Quandary: An Experiment in Offloading Moral Judgement to Artificial Intelligence

Author: James Robinson

Supervisor: Dr. Greg Lee

Prepared for COMP 4983 WI01

Table of Contents

Introduction

Quandary is a project encompassing an automated web scraper, a text-classifying artificial intelligence model, and a mobile-first ubiquitous application created for the capstone portion of the Bachelor of Computer Science degree requirements at Acadia University. The capstone project is designed to give students an opportunity to apply the skills learned throughout their time at the university to a single, semester-long, independently driven project.

This experience would not have been possible without the help and guidance of my supervisor, Dr. Greg Lee, as well as the organization of Dr. Darcy Benoit, director of the Jodrey School of Computer Science, and the support of my friends throughout the whole process. Thank you all.

In simple terms, Quandary is an AI-driven simulator of the popular online community Reddit's r/AmITheAsshole, wherein users can submit morally ambiguous situations from their own lives and receive judgement from online strangers who can label them as 'The Asshole' or 'Not the Asshole.' This emulation is accomplished using software by gathering and analyzing these posts to train a text classification model that can then be applied to input from the app's users and return one of the two labels.

While the subreddit is framed as a place for real moral advice, and indeed, many of the scenarios presented are serious in nature, Quandary is not meant to achieve this same goal and instead fulfills the secondary purpose of amusement, which is another driving force in the popularity of the online community. Hence, Quandary can be seen more as a toy or entertainment service while still demonstrating proof of if and how concepts of data gathering and machine learning can be applied to achieve similar, if more serious, ends.

This paper will discuss the entire development life cycle of Quandary, from conception to planning through production and completion, with an emphasis on the lessons learned throughout. The initial project proposal will first be revisited from a post-production perspective, including an overview of the research undertaken before production, the anticipated structure of the final product, the originally proposed development timeline, and the project goals presented.

The actual development process of Quandary will then be examined within the context of the proposal. Unexpected aspects of the experience, both beneficial and detrimental, will be detailed, and their impact on the final product will be explored.

Next, each component of the completed project, the web scraper, text classifier, and front end, will be considered and analyzed along with the product as a whole to assess whether expectations were met, which parts were most affected by twists in the development process, and the software's overall utility. An explanation will be given of the technologies used and how the components interact and build on each other.

Future work will then be proposed based on unfinished portions of the delivered project as well as the directions which could be taken were development to continue. The merits and costs of possible further improvements will be weighed to determine the most effective path for continuation.

Finally, there will be a reflection on the learning process of the capstone project and the impact that this will have on future projects. Triumphs and mistakes will be inspected to distill workable grains of truth to improve upon the good and alleviate the bad within the project experience.

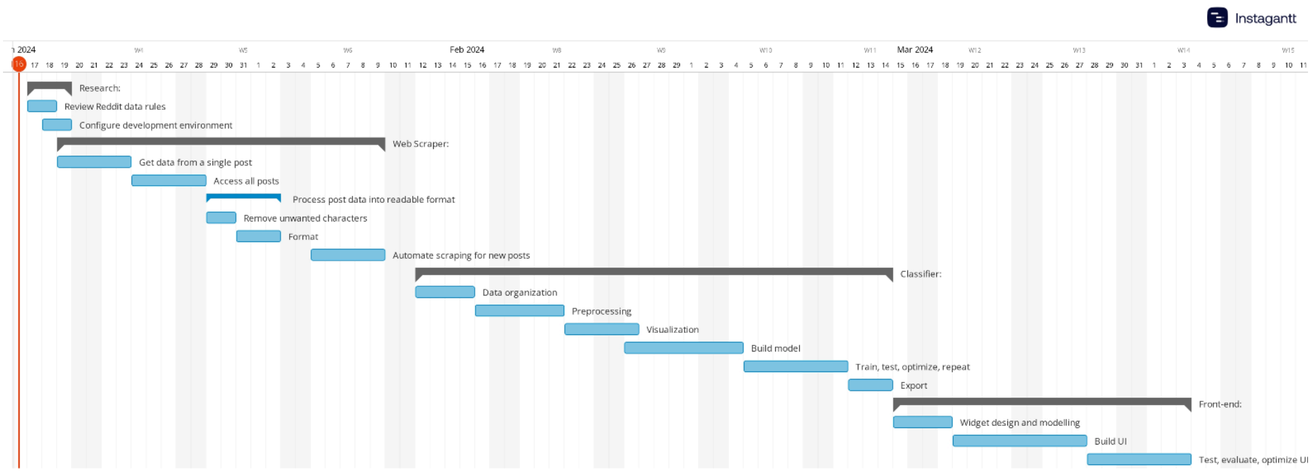
Proposal

The project proposal for Quandary was written in January 2024 while in the early research stages of the development process and consisted of research into similar products and how Quandary could differentiate itself, a proposed tech stack, connections to previous course work, a proposed development timeline, strategies to avoid anticipated pitfalls, and most importantly the deliverable goals of the project. This section will briefly go over key points in the proposal as a framing tool for the actual completed project.

In the proposal, Quandary was asserted to be significantly different from similar existing products in its clear and simple purpose of entertainment rather than a more serious tool with the lofty ambition of creating a universally moral decision-making artificial intelligence. This distinction helps firstly to keep a limit on the scope of the project in its application and also aids in focusing Quandary's design on emulating parts of r/AmITheAsshole.

The proposed coding languages to be used in this project were *Python* for the web scraper and text classifier and *Dart* for the front-end user-facing elements. In addition, the Python libraries *BeautifulSoup, Selenium Webdriver, Pandas, Pyplot,* and *Pytorch* were offered as supplementary tools, with *Pytorch* providing most of the machine-learning model capabilities. The *Flutter* software development kit was proposed as a base for the front-end development as it allows a single project to be built on several platforms without the need for extensive rewriting. Each of these tools and their integration into the project will be discussed in the implementation section of this paper, along with the other libraries and services used but not included in the proposal.

The proposed development timeline of Quandary is represented in the following chart:



This shows how the interdependency of the three components is reflected in the proposed development process; since each piece of software provides input to the next, it would be counterintuitive to work on these pieces out of order. Hence, Quandary's proposed development timeline follows a 'waterfall' approach wherein a more rigid order of production is required in contrast to 'agile' methods that allow for a more loosely defined, adaptive workflow.

The proposal document also includes a section on the anticipated issues that could arise during development. In this section, it is predicted that Reddit's desire to profit from data being gathered from its site, insufficient amounts of data gathered, and unknown aspects or limitations of the proposed tech stack could cause problems, forcing the timeline to shift over time. The main strategy proposed to deal with these issues is introducing small amounts of flexibility to the timeline by allowing for features of each component to be completed in a different order or timeframe while maintaining the overarching plan. As will be discussed later, this strategy enabled an unexpected pivot late in the project to be undertaken without compromising the end product.

The completion and performance targets for Quandary were divided into three tiers: a minimum viable product, a target product, and stretch goals. This division allows the completed project to be assessed not on a pass/fail binary but instead as a list of criteria that more holistically shows in what ways the application succeeded.

The minimum viable product goals were set as a baseline for the success of the project but were still easily achievable enough that total failure was unlikely. The proposed milestones for this level of completion were a web scraper able to gather 1 000 posts worth of data, a text classifier with 80% testing accuracy on unseen posts, and a functional front end that allows users to interact with the other component. These criteria were set in order to better gauge the progress and quality of the project as they are completed.

Quandary's target goals, as per the proposal, are to have 2 000 posts collected by the web scraper, 90% accuracy on unseen data from the text classifier, and a front end that can be distributed and downloaded to mobile devices. This level of completion was set as the expected result at the end of development.

Finally, the stretch goals for Quandary were to have a web scraper that can gather an indefinite amount of post data limited only by the output of content from the site, as well as 95% accuracy from the text classifier and an official app store release for the front end. The purpose of these stretch goals is to provide targets to guide the late stages of production, given the surpassment of the other goals.

All of the goals outlined in the proposal can be summarized in the following table:

Goal	Web Scraper Output	Classifier Accuracy	Front End Viability
Minimum Viable Product	1 000 posts	80%	Functional Demo
Target	2 000 posts	90%	Beta Release
Stretch	Indefinite Posts	95%	Full Release

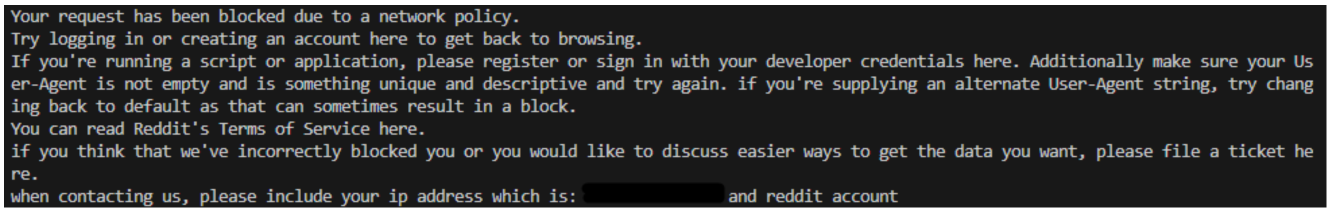
With the context of the proposal, the success of the development process can be better analyzed, along with how such documents can be improved for future projects. In the next section, the implementation of the proposal throughout the development process will be reflected on.

Development

By reviewing the phases of production as they actually happened, the final version of Quandary can be contextualized further, and lessons can be drawn to improve project management practices in the future. This will also aid in assessing the timeline and pitfalls sections of the proposal to determine if accurate predictions were made regarding the development process.

As laid out in the proposed timeline, development began first with work on the web scraper component in mid-January 2024. The initial stages of work were carried out as expected; the libraries used were intuitive despite a total lack of experience using them, and there were no external factors disrupting the development process.

The first major pitfall encountered in the creation of Quandary came while attempting to increase the gathering rate of the web scraper from a proof of concept level to something on par with the minimum viable product goals. Reddit.com will detect automated webdrivers such as the Chrome driver from Selenium used in this project and will block any requests they make. Attempting to scrape from Reddit in this way will result in the following message:

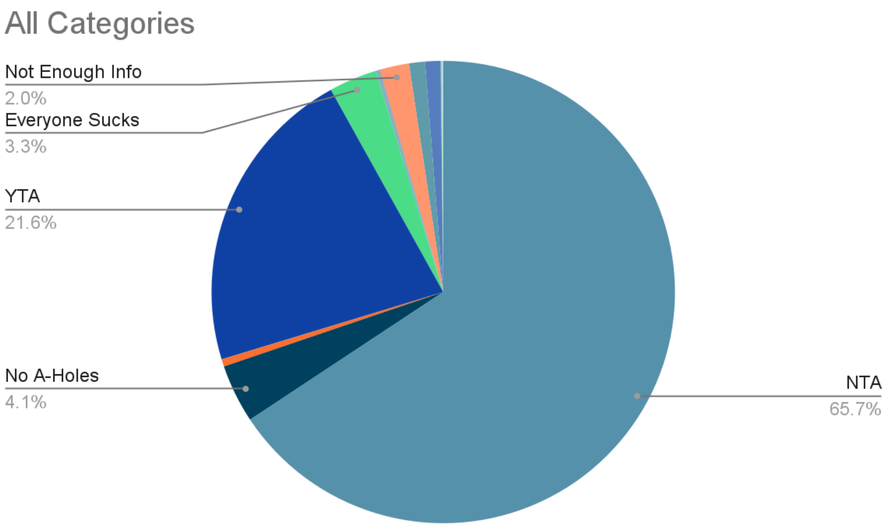


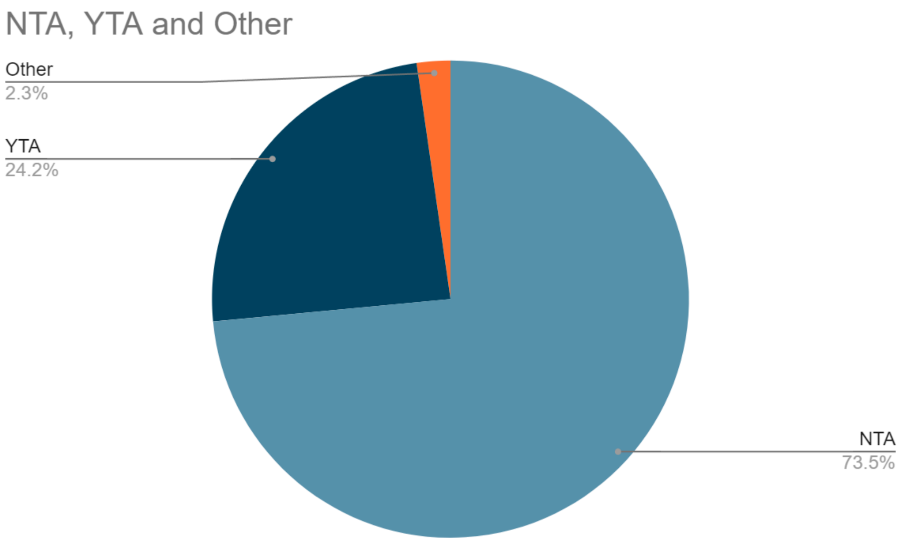
```
Your request has been blocked due to a network policy.
Try logging in or creating an account here to get back to browsing.
If you're running a script or application, please register or sign in with your developer credentials here. Additionally make sure your Us
er-Agent is not empty and is something unique and descriptive and try again. if you're supplying an alternate User-Agent string, try chang
ing back to default as that can sometimes result in a block.
You can read Reddit's Terms of Service here.
if you think that we've incorrectly blocked you or you would like to discuss easier ways to get the data you want, please file a ticket he
re.
when contacting us, please include your ip address which is: [redacted] and reddit account
```

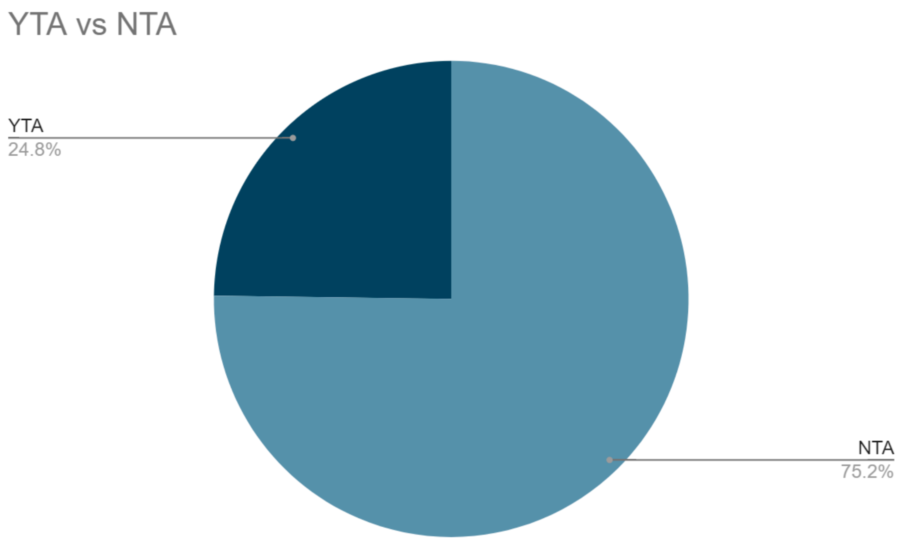
Reddit's API, which would enable the scraper to be used by providing official developer credentials, is priced at $0.24 USD per 1 000 calls. While this price would not in itself critically restrict the project, it would be far preferable to avoid it both for the practical reason of keeping the development free as well as on principle as any value inherent in the data was created by and is owned by users. Instead, the site old.reddit.com, which is a preserved version of Reddit lacking many of the newer features while hosting the same content, was used as the target of the web scraper software. This strategy proved successful, and the project was able to continue as planned with the small caveat that old.reddit.com does not have infinite scrollability and instead will only return up to 1 000 posts for a given set of parameters.

The only other notable deviation in the development of the web scraper component was some difficulty encountered in posts containing emojis. Since the scraper operates by transforming HTML elements to JSON-encoded objects via Python, and the Python codec cannot map many non-standard characters, any posts with emojis, emoticons, or other tokens like currency symbols would initially cause the program to crash. It was simple to catch the encoding error and ignore the text that caused it; however, this leaves behind data which could otherwise be useful; hence, it was instead decided that each error message would be recorded to build up a list of characters to be removed prior to encoding. A similar issue arose with the inclusion of forward slashes (\) in some posts, which are accepted by the encoder but act as escape characters once put into JSON. In this case, the character had to be manually removed.

As development progressed into the text classifier component, issues related to inexperience with the Pytorch library and natural language processing were encountered. The first step in the classifier pipeline is to transform the web scraper's output, a series of JSON files, into a single Pandas dataframe using which the raw data is preprocessed to derive useful information contained within it. In preprocessing, the major difficulty with the classifier, and one of the most challenging aspects of the project, was encountered: post data available on the subreddit is highly skewed towards supporting the original poster. An attempt was made to rectify this by scraping both the top posts as well as the most controversial, however, the asymmetry of the data persisted. Data labels other than the desired binary also had to be either removed or condensed into it; the graphs below show these first preprocessing steps.


All Categories
Not Enough Info
2.0%
Everyone Sucks
3.3%
YTA
21.6%
No A-Holes
4.1%
NTA
65.7%


NTA, YTA and Other
Other
2.3%
YTA
24.2%
NTA
73.5%



Once the initial preprocessing was complete and the natural language processing began, another experience gap was encountered. Practices such as the development of a corpus and vectorization of text data presented a fairly steep learning curve, which was thankfully accounted for in the proposition timeline. Another similar learning curve emerged directly afterwards with the Pytorch neural network library; however, this was lessened due to previous experience with the commonly used alternative TensorFlow, which shares many fundamental similarities.

When starting production on the front end of Quandary, the issue of how to utilize the classifier's output, namely a trained model, was immediately apparent. During initial research, suggestions were found that ONNX models could be run natively on Flutter applications via the FONNX library, and no further research was done at the time. An ONNX model was created from the trained model using Pytorch and imported into the Flutter project; however, the model itself is only one part of the text classifying pipeline, which is essentially useless if it cannot be fed properly vectorized input. Hence, the search for a method to run the classifier's vectorizer on user input in a Flutter app began. It was discovered that a Python object like a vectorizer can be exported and imported using Python's built-in Pickle library. Unfortunately, no applicable method to extract and run the pickled vectorizer was found, and an alternative way to have the user interact with the classifier had to be implemented. The solution found after further research was to offload the work of processing input and returning a prediction from the front end Flutter app to a cloud-hosted Flask application. This additional component is capable of receiving user input from the front end, running both the vectorizer and, subsequently, the classifier on said input, and returning a prediction via an HTTPS GET request.

Despite relatively little experience developing with Flutter, work on the front end component after deciding to offload the actual execution of the classifier was straightforward and caused no issues of note. The structure of programs in Flutter is most similar to web development, which, along with extensive documentation and the simplicity of the application, allowed the component to be created quickly and easily.

Somewhat related to the unexpected structural changes is the issue of unplanned time constraints stemming from the irregularity of course workloads. In a commercial production environment, external factors impacting the amount of time and effort that can be put into a project are limited and easier to account for than in the case of a project-centred course taken simultaneously with others. The challenge, in this case, was that of properly estimating and accounting for the variations in coursework and planning the project in such a way that more flexibility could be introduced to the timeline.

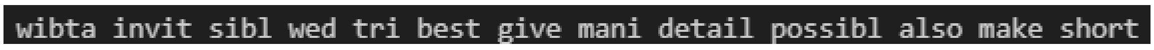
By examining these production challenges and the development process as a whole, Quandary can be viewed more holistically, and its final form better understood. With the knowledge gained from its creation, similar challenges in future projects can be more effectively overcome.

Implementation

This section will discuss the final version of Quandary, which has been delivered in more detail. As in previous sections, this analysis will be done by component in the order of completion, starting with the web scraper, followed by the classifier, and finally, the front end. The unexpected additional back-end component will be considered between the classifier and front end, as its inclusion only directly impacts the implementation of the final component, and it exists between the two in the flow of information when viewing the project as a whole.

The web scraper, as delivered in Quandary's final implementation, easily surpasses the expectations set forth in the proposal and is on par with the stretch goal targets. Using the Selenium Chrome web driver and the BeautifulSoup and Requests libraries, as many posts as could be considered reasonably available are gathered and encoded into JSON. With the limitation on the number of posts returned per parameter set from old.reddit.com, the web scraper is capable of obtaining a maximum of 8 000 posts from the first execution and then up to 2 000 posts per week after. This is accomplished by querying both the 'top' and 'controversial' categories of each time parameter available from the site: weekly, monthly, yearly, and all time. Notably, this strategy will inevitably result in some duplicates across categories, which must be removed in the classifier's preprocessing stage, and it has been noted that, depending on the amount of activity on the site in a given week, there may be significant overlap in the 'top' and 'controversial' sets.

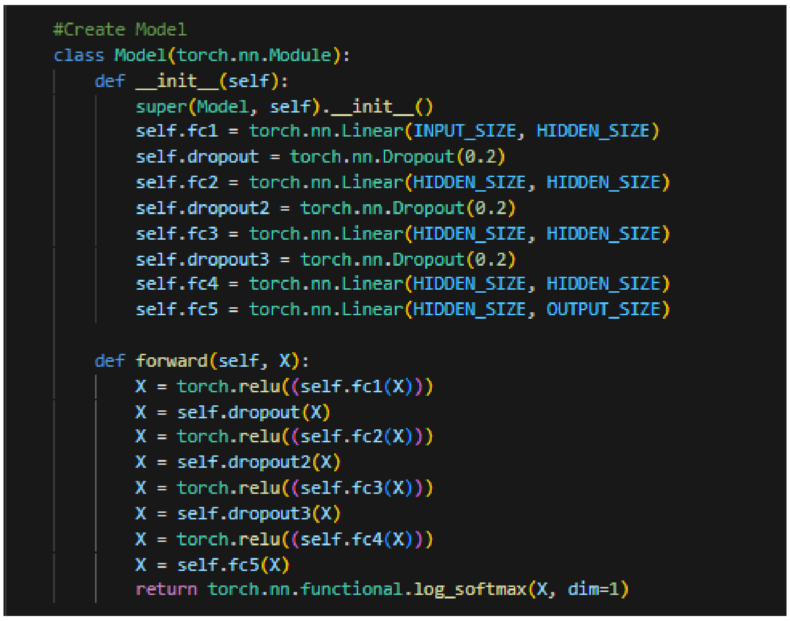
The text classifier component, in its final implementation, can be divided into six distinct phases: data amalgamation, preprocessing, building the model, training, testing, and exporting. To start, the series of JSON files created by the web scraper is condensed into a single file and then read into a Pandas dataframe object. Next, posts which cannot be used because they are empty, duplicates, unwanted labels (ex: 'not enough info'), or caused other errors when being scraped are removed; this reduces the raw dataset to approximately 60% of its original size, largely due to duplicate posts. The remaining posts in the dataframe are further processed into a corpus with the help of the RE and NLTK libraries, which enable the removal of formatting and stopwords as well as the stemming of remaining words to their root form, resulting in sentences like the example below.



```
wibta invit sibl wed tri best give mani detail possibl also make short
```

When a source text is reduced to its simplest form, it can be then transformed into a matrix representation using a TFIDF vectorizer from the Scikit-Learn library. TFIDF, short for Term Frequency Indirect Document Frequency, gives a weight to the occurrence of each word in a document depending on how often it is found across the entire corpus. Numericizing each data point in this way allows it to be passed to a neural network that can predict the associated label based on patterns across the vectors.

The classifier model itself is defined as a Python class extending the Pytorch Module class with eight layers, five linear layers and three dropout layers to help combat overfitting and a forward function that passes the input vector through and returns the prediction value.



```
#Create Model
class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.fc1 = torch.nn.Linear(INPUT_SIZE, HIDDEN_SIZE)
        self.dropout = torch.nn.Dropout(0.2)
        self.fc2 = torch.nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE)
        self.dropout2 = torch.nn.Dropout(0.2)
        self.fc3 = torch.nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE)
        self.dropout3 = torch.nn.Dropout(0.2)
        self.fc4 = torch.nn.Linear(HIDDEN_SIZE, HIDDEN_SIZE)
        self.fc5 = torch.nn.Linear(HIDDEN_SIZE, OUTPUT_SIZE)

    def forward(self, X):
        X = torch.relu((self.fc1(X)))
        X = self.dropout(X)
        X = torch.relu((self.fc2(X)))
        X = self.dropout2(X)
        X = torch.relu((self.fc3(X)))
        X = self.dropout3(X)
        X = torch.relu((self.fc4(X)))
        X = self.fc5(X)
        return torch.nn.functional.log_softmax(X, dim=1)
```

The input size reflects the number of words represented by the vectorizer, in this case 1 000 per post. The output size represents the two possible labels, and the hidden size is the number of nodes per internal layer—in this case, 4 000 was chosen after some experimentation.

Training the model is accomplished by randomly selecting 80% of the posts scraped to create a training subset with the remaining data set aside for testing. The Adam optimizer and cross-entropy loss function were selected to perform the 'learning' functionality for the model that adjusts the model weights based on correct or incorrect predictions of the training data labels. The training data is iterated over by the model for 200 epochs, with the loss (typically) falling each time.

Each post in the 20% share set aside for testing is put through the trained model and evaluated based on whether it was classified correctly and which category it belongs to. This allows for the easy creation of a confusion matrix, which reveals that the classifier is substantially more likely to incorrectly label 'a-hole' posts, whereas 'not' posts are classified much more accurately.

Finally, the model and vectorizer must be saved and exported for use in the Flask API. Pytorch has built-in functionality to import and export trained models as .pt files, which is simple and straightforward but has the slight drawback that the importing target must also have the library downloaded and installed. Instead, the vectorizer is saved using Python's pickle library, which converts objects to and from byte-streams.

The performance of the text classifier in its current form does not meet the standards set forth by the minimum viable product in the project proposal. Based on the time spent adjusting the model parameters to try and achieve an optimal result, it can be concluded that the poor performance must stem from either insufficient data to accurately infer the complexities of online moral judgement or, conversely, from an inadequate knowledge of text-classifying neural network design. As will be discussed in the future work section, there are distinct steps that can be reasonably expected to improve the classifier's performance. It could be argued that improvements to the accuracy of the text classifier do not impact the end-user experience; however, one of the goals of this project is to replicate the likely result from Reddit itself as accurately as possible.

Quandary's Flask API, the unexpected fourth component of the project, is a simple web server that corresponds with the front end using HTTPS GET requests. The hosting site chosen for this component was pythonanywhere.com, a low-cost cloud hosting service designed for Python projects. The vectorizer and classifier exported from the previous component are uploaded to the site directory and imported by the root Python file. The Flask program takes users' text input using a URL parameter and replicates the preprocessing and classification process used in training and testing the model. The result from the classifier is transformed into a JSON object and then returned to the front end. Since this component was not present anywhere in the proposal, it was not explicitly given criteria for success. Hence, to evaluate the API, the functionality it adds to the front end must be considered. As the performance of this component is not able to be viewed on a scale, it either executes the functions and returns the result or fails to do so, evaluation must be a binary criterion. In this case, the component is considered successful since it will return a value when given an input parameter.

The final component of Quandary is a simple Flutter application. It is designed to be mobile-first but can be built as a PC or web application if desired. The front end consists of two pages; the first is a landing page containing information about the project along with a disclaimer and a button to navigate to the application page. The application page contains a dynamically sized, scrollable text box for user input and a submit button that executes the GET request to the API's URL. Flutter was chosen as the framework for this part of the project because it enables multi-platform releases from a single source and provides intuitive and powerful functionality via the widget-based structure. This component meets the target goals set in the proposal - the application can be built on multiple platforms and distributed manually; however, no official release onto Android or iOS app stores was completed, meaning the stretch goal target was not met.

Future Work

This section will detail how Quandary could be improved if work were to continue on the project. While a decision has not yet been made regarding whether or for how long maintenance and improvements will be made to the application, the exercise of determining how best to continue production is a valuable practice both to reflect on one's own work while also providing guidelines for whoever may continue or draw from the project in the future.

The possibility of gathering additional data from other subreddits was explored earlier in development; however, none were found that matched r/AmITheAsshole's flaring practice of determining a consensus decision and would thus require manual analysis of comments to determine a label for the post. Improving the web scraper in this way would potentially lead to a vastly larger dataset as well as open the door for wider applicability given the different communities' content focus. This addition to the web scraper would be fairly simple, given the work already completed, and since each newly added subreddit could multiply the number of posts gathered per week, it would be very beneficial if the amount of data is or becomes a limiting factor.

The main avenue for future work that would be exciting to explore is the possibility of making the entire project cloud-based and automated. The web scraper is already set to run regularly on a local machine and could very simply be made to do the same using a service like pythonanywhere. Similarly, the program which builds and trains the classifier could be scheduled to run regularly as new data is gathered. These changes would incur additional and potentially substantial costs for both storage and computation; the classifier, in particular, tends to be resource-intensive. The Flask API and front end would need little to no changes to support a fully online model since, assuming all of the components are stored using the same service, the updated vectorizer and model could simply replace existing versions used by the Flask project and the front end could continue to use the same URL regardless of any updates made. The advantage of taking this approach would be that, in theory, the performance of the classifier would be constantly improving without any manual input, and functionality would only rely on the upkeep of the hosting service, which is, in theory, much more reliable than a development environment.

In order to justify the costs of storing and running the entire project from a service like pythonanywhere, as suggested above, some form of monetization would need to be introduced to Quandary. There are a few ways this could be accomplished: either by releasing an app version for sale, by keeping the service free but running advertisements, or by locking additional functionality or uses behind in-app purchases. The first option has been more researched over the course of development since an official release was one of the stretch goal targets, but this kind of monetization model for mobile apps is largely outdated and might see limited success. Advertising, in contrast, would remove the cost barrier to users accessing the app but may negatively influence the user experience, particularly if higher-engagement advertisements like popups are used. Another benefit to the advertising model is that both costs and revenues would scale with increased use of the app via increases in computation time and ad impressions, respectively. Charging users based on the number of calls made also carries this scalability benefit but could, conversely, disincentivize users to use the app more and introduce a financial barrier to users. Based on these considerations, the advertising model of monetization is likely the best choice for Quandary, but more research must be undertaken before any of these are implemented.

A different approach to improving on Quandary would be to try and return to the originally proposed plan by determining a method by which both the vectorizer and classifier can be run natively on the front end. Community-sourced Flutter extensions are common and reasonably well-supported, so it is possible that one which would allow for this structure exists or is already in development. Another option would be to create a custom extension to serve Quandary's needs, but this vastly increases the scope of the project and introduces another slew of skills and technologies to learn. Either way, this alteration would further complicate the aforementioned goal of automated updates since the program to build the classifier and the program to run it would no longer be in the same place, and a more robust automated update system for the application would also need to be built.

Finally, significant improvements could still be made to the look and feel of Quandary's front end. Since, during development, much more focus was placed on functionality than form, the appearance of the front end can be overly simplistic and dull while offering little in terms of responsiveness or other enhancements to the user experience. To improve upon this aspect of the project, it would be beneficial to take an iterative approach based on user testing to discover and implement the best next steps. By involving potential users in the design process, the most impactful changes can be prioritized, and creative ideas can be effectively crowdsourced.

Any and all of the changes proposed in this section would serve to improve the performance, maintainability, scalability, and user experience of Quandary, though not necessarily in alignment with the original proposal. Regardless of when or if the proposed future works are undertaken, they give a better idea of what the delivered product is and is not, along with the long-term goals at the heart of the project ethos.

Reflection

Since the main purpose of the capstone project at Acadia is to give students a chance to foster independent learning, this section of the paper will be dedicated to the lessons learned, both technical and more general, throughout the development process.

It goes largely without saying that, given the range of new technologies used, much was learned about the particularities of several frameworks, tools, and services. Briefly, the practice of web scraping, Flask projects, and Flutter were almost entirely unknown before the start of this project, while Pytorch, text classification, and API development were also previously unused but are similar enough to more familiar skills that learning was somewhat accelerated. Learning these technologies, either for the first time or more in-depth, is not only useful for future projects involving them but also hones the skill of quickly learning and understanding more new technologies, which is valuable regardless of the nature of future projects or endeavours.

Another lesson from this project is that of effective and extensive research practices prior to beginning development on a project, particularly if new tools will be used. In the case of Quandary, more research into the capabilities of Flutter to run machine learning models could have avoided the need to restructure the project late into development, were it discovered that the additional component would be required. In the future, more in-depth research within the context of the functionality proposed will be undertaken before any work starts on the project. This can be accomplished by setting aside a longer period dedicated to research in the project timeline.

Similar to this lesson is that of time management, particularly when working on multiple projects simultaneously. As was brought up in the development section, Quandary was not made in isolation or alongside a consistent level of other work. While this did not cause either the project or, indeed, other coursework to fail catastrophically, more planning around the busier times throughout the semester could potentially have allowed much more to be accomplished with much less stress. Once again, a dedicated exercise in planning out how time would be divided during the initial planning phase could have saved time later on in the process.

Hopefully, these lessons can be internalized and carried forward for the betterment of all future work. The practice of reflection in and of itself should aid in enshrining better practices and building good development habits.

Conclusion

To conclude, the capstone project at Acadia has been an intensive and rigorous but also rewarding and fulfilling experience. Quandary has been a monumental personal success from inception to completion despite the challenges faced along the way. Not only is the final product functioning and complete with reasonable success, but the creation process has also shaped who I am as a programmer.