Brian Yi, Vaibhav Malik, Simon, Jingyuan Guo

Professor Khan, Professor Nelson

Analytics Practicum

26 December 2020

User Command Application – Final Report

**Introduction**

This project is a proof of concept for a new feature recommendation for an unreleased software product from an international SAAS provider. Despite an NDA restriction, the company has allowed the reveal of relevant product features in this report as long as the company and product remain anonymous. This unreleased software product, which will be referred to as Product X, has project management features that all focus to optimize workflow within an organization. The main two features of Product X relevant to this project facilitate task and project management by allowing users to keep track of any jobs they need to complete. Consider the following situation where a user is within one of the many features in Product X that is neither the task nor project tabs. They suddenly get a message on their phone that the only software engineer on Project Artemis was fired, and a replacement needs to be found for the project to continue. Currently, Project Artemis has the status of *On Target*, which now needs to be changed to account for this sudden obstacle. In order to update everyone else of this interruption, the user now needs to navigate away from their current tab, open the project tab, find Project Artemis among all the other on-going projects, and proceed to update the respective project status to *Blocked*. After completing this status update, the user now needs to waste time navigating back to the tab where they were previously working. This example shows how a lot of time is unnecessarily wasted when updating tasks and projects within Product X.

The feature recommendation developed to eliminate this time wasted is a user command application that automates status updates for tasks and projects within Product X. The current version of the user command application only supports the aforementioned two use cases: tasks and projects. Both use cases each have five statuses, which differ between tasks and projects, that a user can choose between to reflect the job situation. This user command application was designed to be integrated into Product X as a text interface with a fixed position in the bottom right corner similar to the chatbot feature seen in popular social media platforms like Facebook and Instagram. Unlike a traditional command line where a specific syntax is needed for unique commands, the user can type in a phrase in their preferred grammar that details the status update they want for a specific task or project. This feature uses natural language processing (NLP) to handle any variations in commands between different people to ensure that the user command application properly executes a given command. After parsing the text command through NLP, this feature will then use machine learning (ML) to automate the user’s requested status updates without them ever needing to exit their current workflow.

**Background**

Companies in every industry, sector, and market have become increasingly reliant on digital tools that help maximize productivity for teams within their workforce. Some of these tools have taken the form of visualization software, collaborative project management web applications, and team communication platforms. The more connected team members become and the more information that is readily shareable and accessible between team members, the larger the value they can generate for their businesses. The partner company, a company which provides enterprise clients with user interface control libraries, is in the process of developing a new product, Product X, which provides their customers with a comprehensive suite of project collaboration tools. Product X’s value proposition is based on providing clients’ teams with the ability to maximize their productivity because of the comprehensiveness of the provided tools and the ability for teams to interact with each other through various mediums in the context of project management. Product X also help users manage their workflow and keep teams informed on the statuses of the assignments they are a part of. In order to facilitate an even higher rate of productivity for application users, this project builds a user command application within Product X that can automate the process of a user manually updating the statuses of tasks or projects. A high-level overview of this user command application includes understanding the user’s text command using NLP, identifying the task or project and corresponding status update through ML, and then executing this status update in Product X. By automating this process, users avoid the time lag associated with navigating through Product X while finding the task or project they are updating.

This user command application uses an ensemble approach that combines two layers of ML algorithms to translate a user command to a definitive action. Consider the example command “finished task market research.” Within this example text command, the ensemble algorithm needs to identify two main components: the topic and the action. The first ML layer will determine the correct topic, which is either a *Project* or *Task* in Product X that the user wants to update. The second ML layer has two different models that predict the status update, which is referred to as an action, for the respective task or project. Tasks have the five possible actions: *To Do*, *In Progress*, *In Review*, *Blocked*, and *Completed*, and projects have the five possible actions: *Create*, *On Target*, *At Risk*, *In Danger*, and *Completed*. Figure 1 helps visualize this ensemble approach consisting of the topic, task-action, and project-action models.

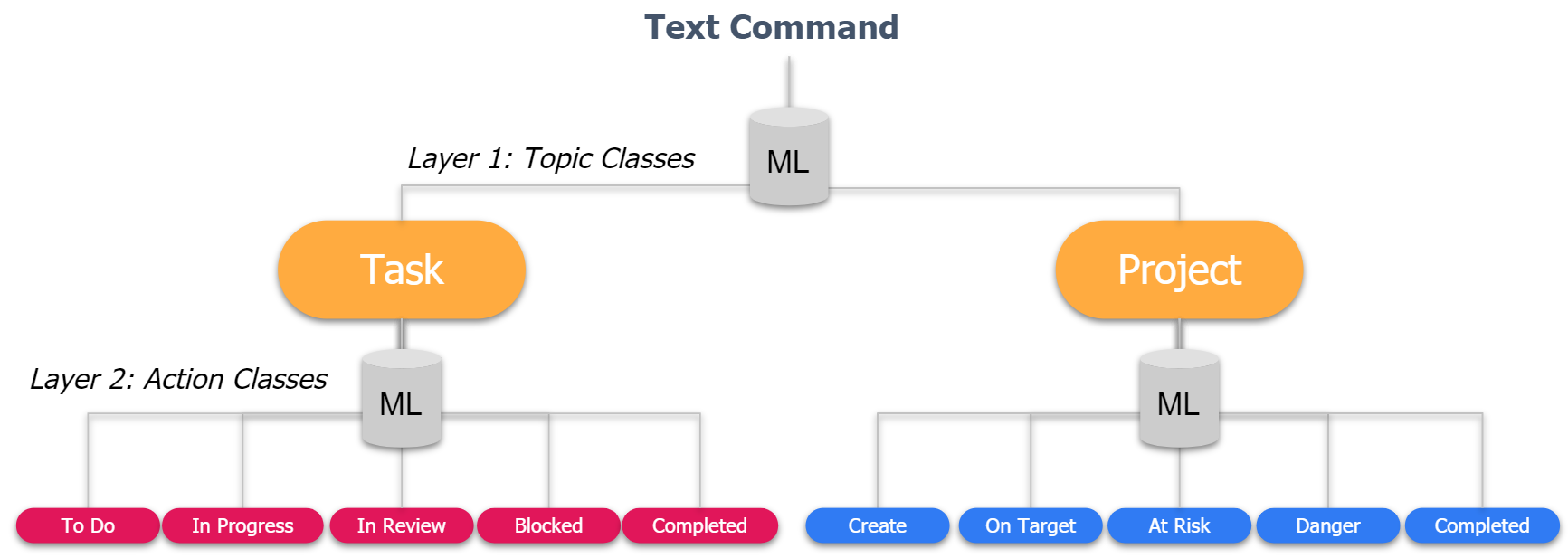


Figure 1. Ensemble ML Approach

In order to have a preliminary validation of this ensemble approach, the Naive Bayes (NB) model was selected as a standard ML algorithm to fulfill this role. NB is based off the Bayes’ formula: , where X and y are the features and classes, respectively2. In this project, the features are the words in a text command and the predicted class is either of a topic or an action. In order for the text command to be in a format suitable for modeling, bag-of-words (BoW) was used to vectorize textual data by counting how often a word appears in a command6. NB was also selected because it easily integrates a BoW approach where each feature is the word frequency of each distinct word that appears in a given text command. BoW can be especially effective in this context where predictions are based on brief and simple text commands in which grammar and sentence structure are not as relevant comparatively to longer texts. Again, consider the example user command “finished task market research.” This is a trivial example for the first layer of prediction between a *Task* and a *Project*, because the NB word frequency approach is most effective in classifying commands that contain the class label for apparent reasons. For the second layer of prediction of the correct action status update for this task command, “finished” is a distinct word in the text command that is very indicative of the correction action – *Completed*. In these text commands where specific verbs are very suggestive of the correct action, the BoW approach is effective in considering these high impact words that will occur more often than other non-stop words in a given text command. As a result, NB serves as a good benchmark to measure how well a BoW word frequency approach is for predicting a topic or action from text commands.

After initial NB benchmarking confirmed that the current ensemble approach was feasible, a more sophisticated algorithm was needed to improve model performance. After some research, a neural network (NN) model was determined to be the algorithm of choice because it can identify semantic nuances within a text command that a BoW approach fails to capture. These additional details within text are captured through word embedding, where words are not just measured by the frequency in which they occur; instead, words are mapped to vectors that represent their location within a vector space6. Words that are closer in distance within the vector space would also be more similar in meaning. The chosen word embedding model for this project is Google’s Word2vec, which is used to vectorize all text commands. In theory, vectorization gives models the ability to accurately make predictions on text commands which convey the same sentiment as text commands that were trained on even if the sentence structure and vocabulary are not exactly the same. Again, consider the command “finished task market research.” If a standard NN model in the first layer of the ensemble approach only uses the last two words to make a topic prediction, it would be a lot less accurate than if it uses the last three words. A Long Short-Term Memory (LSTM) model, which is an advanced recurrent neural network, is used because it has built in memory within the algorithm that can better remember earlier words that may help with making a prediction5.

Note that both machine learning models were also selected due to their unique vectorized outputs that aims in providing more flexibility to actions that can be executed off a more informed prediction. Instead of having just a single class prediction as a model output, both the NB and LSTM algorithms output an array of probabilities that allows for further judgement of the confidence on this prediction. Since the automation of an incorrectly predicted command will result in the user needing to manually update the task themselves, knowing the confidence levels of a prediction is critical in avoiding creating any friction for the user. This type of probabilistic confidence approach is facilitated by the vectorized output of both algorithms.

Returning back to the example user command, “finished task market research*,*” the two-layer ML algorithm has now predicted both the topic (*Task*) and the action (*Completed*). The final step to gathering the necessary information to automate this command in Product X is locating the name of the task, market research, which will be referred to as the identifier. Locating the identifier does not use ML and instead is done solely through descriptive analytics and parts of speech (POS) tagging. A high-level view of this process is tagging each word in the command with its respective POS, finding patterns through descriptive analytics, and then coming up with linguistic rules to best predict the location of the task name3. In a production environment, inaccurate predictions can be avoided because the user can be asked to include quotation marks around the task or project name allowing for trivial identification. For the sake of challenging oneself in an academic setting, this alternative POS identifier tagger was also developed as a part of the project’s approach.

After developing the two-layer ML approach and identifier tagger, predictions had to be somehow linked to Product X where commands can be automated. As a result, all models and scripts were hosted on a cloud platform (Google Cloud) to allow for an easy connection with Product X. A front-end UI of the user command application was also developed to show how this feature would look like in a production environment. The POS identifier tagger, cloud deployment, and front-end UI design will all be explained in more detail under additional work.

**Literature Review**

Since the entire dataset used in this project was manually synthesized, this project needed to identify ways to augment the data. The paper, “EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks,” presented many different techniques to improve the performance of text classification by augmenting the data7. The four main easy data augmentation (EDA) techniques are synonym replacement, random insertion, random swap, and random deletion. The benefits of these EDA techniques are that they do not have a high cost of implementation, which was perfect for use in this current project where computing power was limited. SR was integrated by “randomly [choosing] *n* words from the sentence that are not stop words [and replacing] each of these words with one of its synonyms chosen at random.” The number of words to be changed increased for longer sentences with more words because they can absorb more noise. This project implements a very similar version of SR with small differences due to the consistent and brief lengths of user text commands. As a result, all words within the text command (excluding stop words) were translated, and each synonym had up to five synonyms depending on how many semantically similar synonyms were available. SR proved to be invaluable for this project since it was the main method of augmentation that introduced the bulk of new instances while back-translation served more of the use in introducing syntax variety into the data.

This project’s first benchmark NB model employs a BoW approach, which this paper, “Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval” details in its methods for vectorizing text data2. This vectorization method employs the BoW model where all the unique words from a collection of documents are given a numerical index. Essentially, this model transforms text into a vector of length *n*, the number of terms in the BoWs, with each element in the vector being a binary value that indicates whether the term in the BoWs with the corresponding location index occurs in the document. This paper also suggests a slight variation to this representation and assigns a weight to the elements in the message vector where the weight is the relative frequency of the corresponding word in the messages. It also adds the words and terms from concepts in a domain ontology for a specific topic to the BoWs. A vector for a specific topic is created where the weights for elements in this vector represent the probability that a document containing the corresponding word is related to that topic. These weights are initialized through the application of Term Frequency Inverse Document Frequency (TFIDF) method on documents associated with domain ontology concepts. Since both vectors are of the same length and the indexes of elements in both correspond to the same words and terms, the dot product of the two vectors is calculated and the result represents the degree of relation between the message text and topic. An arbitrary threshold is set to determine whether the calculated degree of relation is significant enough to indicate that the user messages are related to the topic. Once topic(s) are identified, the topic with the highest degree of relation is chosen and digital documents related to that topic are recommended to the user. Even though TFIDF and the weights corresponding to a domain ontology was not implemented into the NB algorithm in this project, these are methods that can tune the current NB model that employs the basic BoW approach.

Even though the BoW model has been dominant in sentiment analysis, this project also explores the alternative word embedding approach through the paper, “More than Bags of Words: Sentiment Analysis with Word Embeddings6.” This paper uses word embeddings during their supervised machine learning models to estimate the level of negativity in parliamentary speeches. Since sentiment analysis has been a very important area of communication research, many studies have shared in this BoW approach. However, the strength of word embeddings lies in its ability to capture the similarities in semantics between comparable words. While the BoW model treats each word as independent units, word embedding attempts to include both semantic and syntactic relations between words as well. One of the largest benefits of word embeddings is that when a new word is seen in the testing set that is missing in training sets, it can still be used to make a prediction through similar words that are used as a replacement. Consider the example where the words “good” and “bad” are seen in the training set and the word “great” is not included. If the word “great” is seen in the testing set, a word embedding model is still able to perceive the sentiment correctly because the numerical vectorizations of the words “good” and “great” are more similar than those of “bad” and “great.” There are many pre-trained word embeddings corpora including Word2vec, GloVe, and Polyglot, which is the chosen German corpus that this paper uses. Each word embedding in the Polyglot corpus has 64 dimensions with each dimension describing the language structure and meaning of the corresponding word. One example to describe the relationships within these embeddings is how the word “man” is related to “woman” as “king” is related to “queen.” The type of information is invaluable for the purposes of this project where synonym replacement is used to augment the data, and word embedding would still allow the model to capture the semantic relations between synonyms.

This project implements word embedding using LSTM neural networks, which details are described in the paper, “Actionable and Political Text Classification using Word Embedding and LSTM5.” This paper applies both word embeddings and LSTM to binary text classification by predicting whether social media messages from customers of service providers are *Actionable* or *Non-Actionable*. This study chooses to analyze whether a customer request is actionable over how happy a customer is, because within the industry context, it is much more useful for customer service to know which requests they can provide actionable results towards rather than which customers are more upset. This paper also classifies social media messages based on political bias by determining whether the messages are *Democratic* or *Republican*. This classification was inherently difficult because not every social media message is purely left-leaning or right-leaning due to the mixed sentiment of many people. There are many benefits of word embedding in conjunction with LSTM for the use of text classification is emphasized throughout this paper. Input messages for models are first tokenized, where vector length is large enough to account for the maximum length of a message and zero index values are used when it is shorter than the maximum length. This input is then fed into the RNN, which has internal states that keep information from previous events, which in the case of text classification, the events are previous words. The RNNs ability to preserve information makes it very useful in processing sequential data, which is in the inherent nature of textual applications. While traditional RNNs have the downside of not being able to learn long relationships in data due to diminishing gradients, LSTMs overcome this obstacle with its unique memory units that integrate input, output, and forget gates. Even though most user text commands are brief in the context of this project, the ability to preserve information from the first word to the last word of longer ten word text commands can still be very useful in classification accuracy.

One of the most important parts to this project is analyzing the output of the LSTM in order to make a final accurate prediction. The paper, “Towards Open Set Deep Networks,” is very useful in detailing how the SoftMax layer is typically used for LSTMs to process model outputs1. The output from the neural network is inputted into a SoftMax function, which then produces a probability distribution across *n* class labels. This transformation from a numerical vector into a probability distribution via a SoftMax layer is particularly important when using metrics such as cross entropy loss. As a result, the current LSTM approach employed in this project uses a SoftMax layer to transform model outputs into confidence probabilities the user command application can then use to decide whether to execute a given user command. However, consider the situation where the user types “Hello World” into the user command application. Ideally, this unknown user text command that is unrelated to the application purpose should not have a model prediction from the known *n* class labels at all. Furthermore, the SoftMax function will simply scale up the low LSTM outputs into a larger probability, which can give the illusion that the model is confident in this prediction. For this reason, setting a confidence threshold to weed out unconfident predictions does not work very well. This paper introduces an algorithm that accounts for unknown text commands known as the OpenMax function. OpenMax is different from SoftMax because it estimates the probability that any given input belongs to an unknown class not represented in the known *n* classes. Essentially, by being able to identify inputs that most likely belong to an unknown class, the model will no longer make a mistaken prediction and can now effectively provide a separate output for these unique predictions. In the context of when a user enters an unrelated command into the user command application, the OpenMax would allow the application to return a message to the user indicating that the command’s content does not fall under the capabilities that the application can provide.

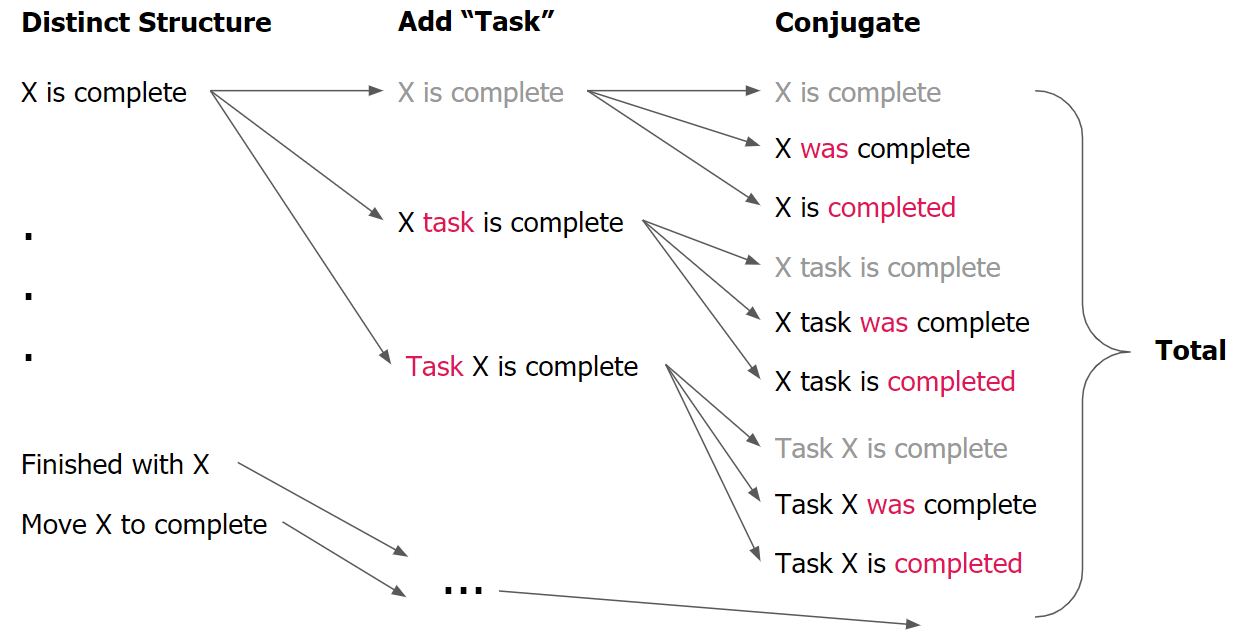
The paper, “Machine Learning and Cloud Computing: Survey of Distributed and SaaS Solutions,” details the various cloud ML technologies that are available today4. These cloud alternatives are very useful in this era of big data because traditional ML libraries do not process data and run functions while accounting for data size. The solution to this issue of long runtimes is deploying ML on the cloud, where support for big data is integrated to the design of these cloud frameworks. As a result, many popular analytic tools and libraries in R and Python have become the basis of many of these cloud ML frameworks. For example, consider the popular cloud Apache Hadoop framework that was developed to handle big data. After Hadoop and HDFS were developed to handle basic analytics via MapReduce, many people wanted to run more complex ML algorithms on top of this big data infrastructure, resulting in Apache Mahout. Many ML frameworks on the cloud also vary in complexity and feature support depending on the different needs of users. This cloud deployment of ML was very useful for this project because it was the easiest way to deploy the LSTM models to service Product X. The paper references the Google Prediction API, which is one of Google’s available cloud-based ML platforms. The data is stored in Google Cloud Storage, where the user can then use a variety of languages including Python to build a model that is trained on that stored data. This project uses a similar cloud platform from Google known as TorchServe that specializes in hosting deep learning models. More details to cloud deployment will be provided in the additional work section.

**Data**

Since the proposed user command application feature is a novel recommendation, there was no user command dataset available from Product X. Furthermore, the user text commands are unique to Product X features so opensource text command data sources that fit the desired use cases are scarce. As a result, this project needed to manually synthesize all the data used for modeling from scratch. The team needed to brainstorm user text commands that update the statuses for tasks and objects in Product X. The example command from earlier, “finished task market research,” is one such example where the user wants to change the status of the market research task to a *Completed* status. However, another user who wants to make the same status update might type a different command such as “completed market research task.” Since the user command application needs to be robust with interpreting the various ways a user might structure text commands, the manual synthesis process needs to be as comprehensive as possible in covering all of the possibilities.

Since different team members were involved in brainstorming their individual commands, a systematic approach needed to be implemented to avoid duplicates. Text commands needed to be generated for each of the five actions for projects and tasks. The synthesis was done according to the following process for each action at a time. Figure 2 shows how the data synthesis process was done.

Figure 2. Data Synthesis Process



Note that task and project names were designated as X while synthesizing data because the name was irrelevant to the syntax of the command; the names of tasks or projects were added after the entire synthesis process was complete prior to augmentation. First, text commands with different syntax were brainstormed. For each text command, the topic that it refers to was then added to the command, whether that was a task or project; the command in Figure 2 was referring to a task so the word “task” was added to the original command to create two more variations. The third and final step was conjugating all verbs within each command from step two. From the one command “X is complete,” there are now nine unique variations. Every distinct command brainstormed in step one underwent these this three-step process to create all the commands within a given action. This entire procedure was then repeated for all five actions for tasks and projects for a total of 609 manually synthesized instances. Despite the difficulties that were faced with manually brainstorming unique instances, this also provides control over how many commands were generated for each class, eliminating any possibility of an imbalanced dataset.

Due to the simplistic nature of user text commands, there is a limit to how many realistic instances that human creativity can give birth to. As a result, the initial data needed to be augmented to a size that is suitable for modeling. The first method found from literature research was back-translation, which translates a text command to another language before translating it back to English8. Back-translation is useful because it introduces new sentence structures to commands in order to mimic users in real life with different language backgrounds. Furthermore, translating a command to a language with a very different syntax compared to English alters the sentence structure of existing commands; this project uses Japanese as the translation language for this reason. As a result, back-translation also introduces commands with erroneous syntax to help keep models robust.

Since many accurate and bulk translation frameworks were not cost-friendly, back-translation was implemented through a free python package Googletrans that uses Google Translate, a proven translation engine. For each back-translation iteration, the entire dataset is increased two-fold, theoretically allowing for the data to be easily scaled up. However, the simplistic nature of these user commands only allows for so many variations; as such, back-translating the same five-word command to three or four languages will result in duplicates and diminishing returns. The second issue with this scaling method is that the translation package, Googeltrans, is unofficial and had a few issues when it came to making requests through its use of Google Translate’s API. Mainly, if too many translation requests were made within a short time frame, only a portion of the data was translated. This was the greatest obstacle when it came to creating a back-translation script using the Googletrans package. After attempting various fixes, the final solution was to make batch translation requests with a grace period between each request. The aforementioned issues combined with lengthy model runtimes resulted in a one back-translation limit for augmentation purposes.

The second method used to augment data was synonym replacement for non-stop words in text commands7. Note that the use of synonyms to create new commands in the initial manual synthesis process was disallowed because a synonym-enhancement script can efficiently automate this process. A word-finding API, Datamuse, was incorporated into an open-source script to find the synonyms needed for replacement. The synonyms used were from a large lexical database, WordNet, where synonyms are selected based on its semantic and lexical relationship with the original word. The synonym replacement script reads through every word in a text command and creates a new text command for each word that is replaced with a corresponding synonym. The script was modified to include stop words that did not need to be replaced, such as “on” and “in” where the WordNet synonyms were not semantically accurate. Furthermore, a limitation of five synonyms per word was introduced to prevent the text commands from replacing words with synonyms with a meaning that is different from the original. This two-step data augmentation process of back-translation and synonym replacement scaled the initial dataset of 609 instances to a total of 6979.

**Data Screenshot**

Table 1. Augmented Data

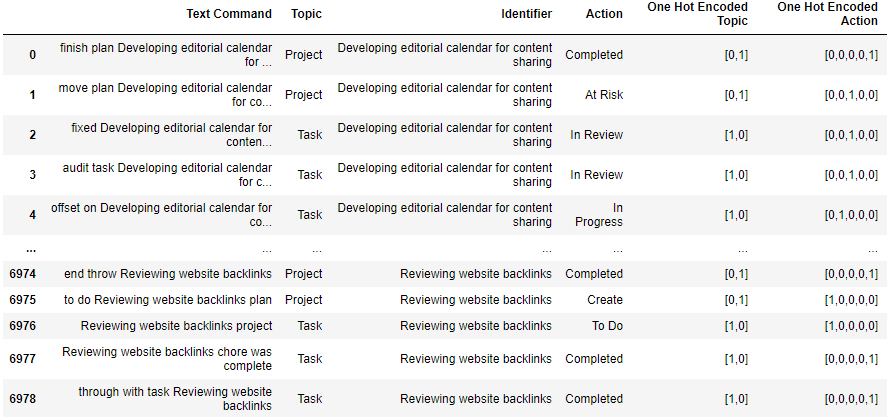


Table 1 is a screenshot of the final synthesized and augmented dataset that was used for modeling. Including a separate table detailing attribute metadata was trivial because the individual word frequency features used by the NB algorithm were not even determined until modeling, and all LSTM features were in vectorized numeric lists using Word2vec word embedding.

Note that the data contains the input features that were extracted from the “Text Command” column. The “Topic” and “Action” columns are the labels to be predicted, and their one-hot encoded counterparts were used during modeling to calculate cross entropy loss. The last column, “Identifier,” is the name of the task or project within Product X that is being referenced by user commands; this column was only used later for locating the identifier within text commands.

Data synthesis took place within an Excel sheet that was imported as a csv file for augmentation purposes. Aside from the manually synthesized set, three main augmented data sources were stored throughout the project: back-translated, synonym replaced, and back-translation and synonym replaced. Note that the final version of the back-translated and synonym replaced dataset in Table 1 was used for all modeling results. Some older versions of data were stored for fault tolerance and project replication purposes.

**Project Methods**

As mentioned in the background, the two models used in the project were the NB and the LSTM algorithms; furthermore, each model was used twice due to the two-layer nature of the ensemble approach that first predicts the topic and then predicts the action for that topic. The NB model was natively coded in Python without the use of any ML libraries like scikit-learn, because the features were individual word frequency probabilities that needed to be extracted from each input text command. Note that features were not one-hot encoded despite the NB model incorporating a BoW approach simply because word frequency can be calculated without the need of vectorization within the NB algorithm2. The extent of the attribute selection process for the NB algorithm was invalidating stop words in the text command that were considered to have little influence on predicting a topic or action. Consider an example NB model from the first layer of the ensemble approach where the model needs to predict whether the user command is referring to a *Task* or *Project* in Product X. For each text command, NB would calculate the probability that the command is referring to each class and output an array of probabilities that sum to one. In this case of predicting a topic, there are two classes to predict between so the output would be an array of two probabilities; when predicting between five actions in the second layer of modeling, there would be an array of five probabilities. The largest probability in the output array is the most likely class and is then used as the model’s final prediction.

LSTM modeling differs from NB in that the text commands need to first be vectorized using Word2vec. Each text command was transformed into a 2D numerical array with *n* columns that represent the number of words in a given command6. For each column, there is a feature dimension of size 350 to characterize the word in the vector space. This array of vectorized commands was then inputted into the LSTM model built using PyTorch. Again, the extent of the attribute selection process for LSTM modeling was not vectorizing certain stop words in the text command that were considered to have little influence on predicting a topic or action. The LSTM consists of a single sequence of ten nodes, which corresponds to the length of the longest text command. Each node corresponds to a vectorized word in the text command and contains a neural network that outputted a value based on the input vectorized word. This value was propagated to the next node where the process was repeated all the way until the last node. For text commands that have less than 10 words, the extraneous nodes did not change the value and simply passed the value to the end of the sequence5. The final output from the sequence of nodes was transformed into an array of numeric predictions that each represent a class. These numerical predictions were then inputted into a Log Softmax layer and converted into a natural log of corresponding probabilities1. The prediction with the largest value was then considered as the LSTM’s final prediction. Unlike the NB model where training and testing a model is not an iterative process, the LSTM needs to go through many iterations (epochs) of training and testing before it can learn to predict on the training data accurately. Note that mini-batch gradient descent was used within each epoch to update LSTM model parameters.

Both NB and LSTM modeling used the same validation and evaluation methods due to similar model outputs. Cross validation was integrated in its simplest form through an 80/20 split between the training and testing set. This holdout method was used in order to evaluate model efficacy based on data that the model had not seen before. The NB and LSTM models were validated on 100 and 20 iterations, respectively, to decrease variance. The two evaluation metrics used to measure model performance were cross entropy loss and accuracy. Accuracy was used as an overall metric to preliminarily judge how many user text commands were predicted properly. However, the consequence of executing an incorrect user command is disastrous with respect to user experience, so a metric that provides a non-deterministic outcome would give the user command application more information on whether to execute a given prediction. As a result, cross entropy loss was the metric chosen because it specializes in evaluating the performance of probabilistic classification models such as NB and LSTM. Essentially, cross entropy loss measures the difference between the probabilistic predictions and actual label probabilities. Cross entropy loss can also be determined for each individual class to see if models have more predictive power for certain classes in comparison to others. A lower cross entropy loss value indicates that the model has a higher probabilistic confidence in that particular class’s prediction. Having this probabilistic model output now provides more information to the user command application to decide whether to execute its prediction depending on prediction confidence. A confidence threshold was also integrated into both the NB and LSTM algorithms such that any predictions with a probability underneath this threshold would not be automated to ensure maximum user satisfaction.

Model tuning was done for both NB and LSTM, where the beginning phase of the project was dedicated to tuning the benchmark NB models. Since the NB algorithm used word frequency as features, one way to improve the model for production aspirations was enriching the data; this required synthesizing commands to cover major key verbs or nouns that would cover each topic and action user command. This process was mainly done through brainstorming the various ways that a user would use a specific status for a task or project. For example, if a user uses the *Blocked* action for a project to represent a situation where a critical team member became unavailable, the team had to then conceptualize the various user commands corresponding to this event. Aside from improving data quality, stop words were also introduced into the algorithm to make sure that word frequency of certain words such as “the” or “and” were not being considered when predicting an action or topic. Furthermore, Laplace Smoothing was also a necessary implementation to improve accuracy on text commands with new words that the NB algorithm had not previously encountered for a given class. Since NB was more of a benchmark model, model tuning was briefly done during the initial phase of the project before quickly considering better models that would fundamentally outperform NB with respect to robustness when data scaling.

The model tuning process for LSTM was limited due to the long run-times of LSTM training. Instead, the conscious decision was made to focus on creating a more realistic dataset to create a model that would perform the best in a production environment. This project prioritized results that would benefit the partner company over negligible tuning of models on a more limited dataset for a higher performance. The steps taken to create a more realistic dataset was communicating with the partner company to discuss various use cases, and then going back to the data synthesis step to create commands that covered the various scenarios. Note that in the future, the team considered running variations of the grid search algorithm as the initial approach to LSTM hyperparameter tuning.

**Results**

NB and LSTM modeling results were from the three different ML models: topic, task-action, and task-project (referenced in Figure 1). First, the benchmark NB model performances were evaluated. NB models were run for 100 iterations with an 80/20 split where accuracy and cross entropy values were calculated by taking the mean value over all iterations.

Table 2. NB Topic Model Performance

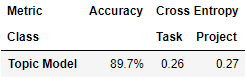


Table 2 shows the test accuracy for the NB topic model and test cross entropy values for individual class predictions. Overall model accuracy is reasonably high at 89.7%. The cross entropy values are somewhat low for both predicted classes, indicating that the model is reasonably confident when it comes to distinguishing whether a command is referring to a *Task* or *Project*.

Table 3. NB Task-Action Model Performance

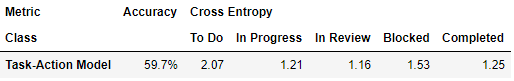
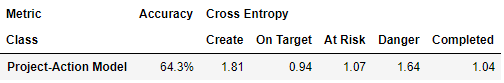


Table 4. NB Project-Action Model Performance



Tables 3 and 4 show the performance of the two NB action models, which have much lower accuracies compared to the topic model. This decreased performance is also reflected in the higher cross entropy values indicating that these models are not very confident in determining which action a text command is referring to. Even though the topic model has a similar confidence in predicting between its classes, the action modeler has a much greater variance in its prediction confidence between classes. Notably, the action models are far less confident when identifying text commands that create either a task or project in Product X, as reflected by the high cross entropy values for the *To Do* and *Create* classes.

Next, the LSTM modeling was evaluated. Due to lengthy model runtimes, LSTM models were run for only 20 iterations, where accuracy and cross entropy values were once again calculated by taking the mean value over all iterations. Within each iteration, the LSTM was trained over many epochs to learn how to predict on the provided data. The visualizations of each model show the testing accuracy and the mean training loss per epoch. Note that the mean training loss was calculated by taking the LSTM’s mean training loss of the prediction classes.

Figure 3. LSTM Topic Training Loss and Testing Accuracy Per Epoch

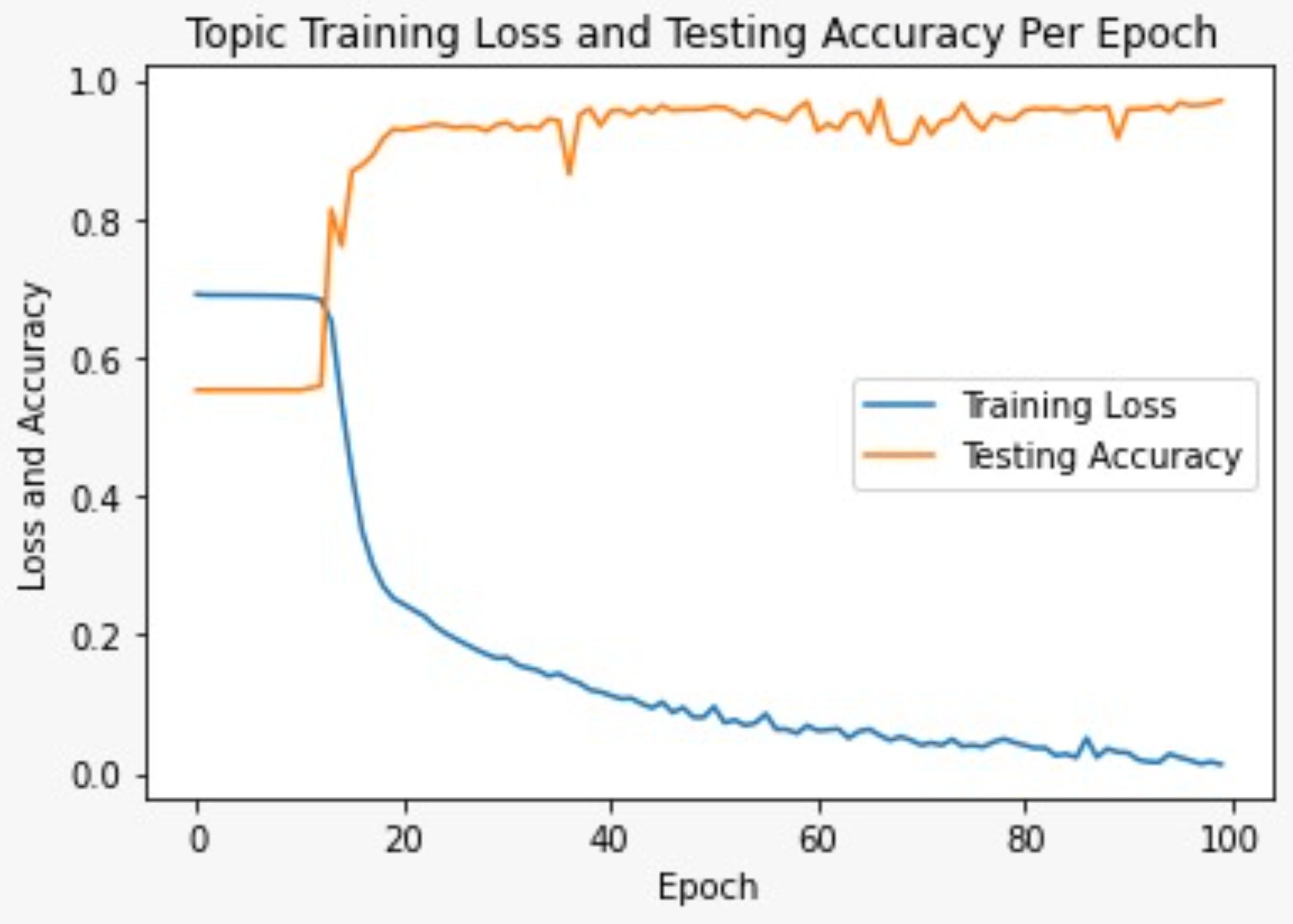


Figure 3 visualizes the topic model’s mean training loss and test accuracy per epoch. For about the first 15 epochs, the LSTM seems to not recognize any patterns before suddenly increasing in performance whereby the 20th epoch, the model already has a training loss of under 0.30. For the remaining 80 epochs, the model manages to decrease the training loss to below 0.10 by the 100th epoch. As expected, when training loss decreases, the testing accuracy increases.

Table 5. LSTM Topic Model Performance

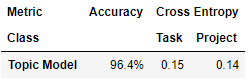
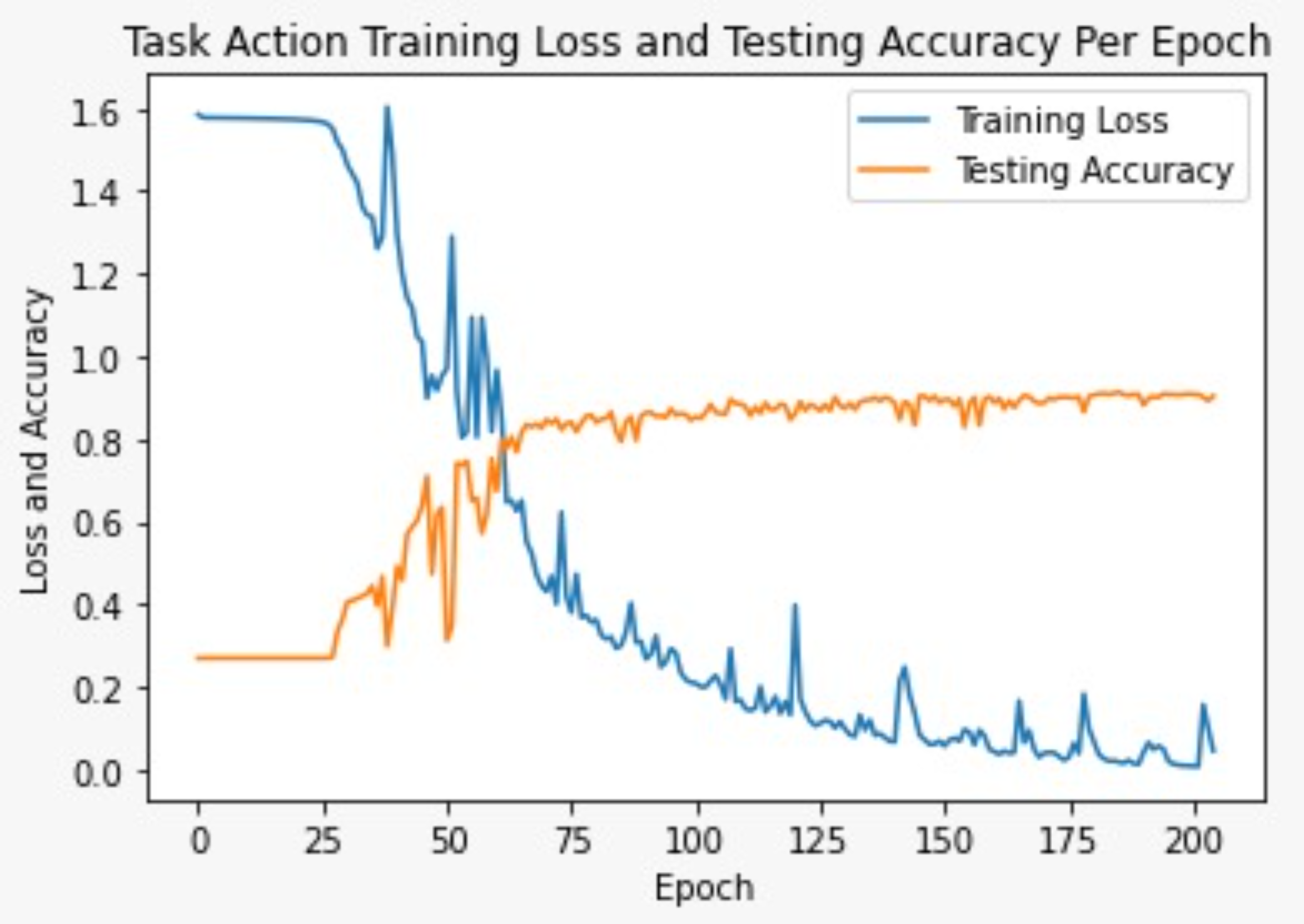


Table 5 shows the accuracy and cross entropy of the test set of the fully trained topic model over 100 epochs. The accuracy of 96.4% is an improvement over the 89.7% accuracy of its NB counterpart. Furthermore, the cross entropy values for *Task* and *Project* are also lower than that of the NB model, which indicates that the LSTM topic model is better at determining whether a task command is referring to a *Task* or *Project*.

Figure 4. LSTM Task-Action Training Loss and Testing Accuracy Per Epoch



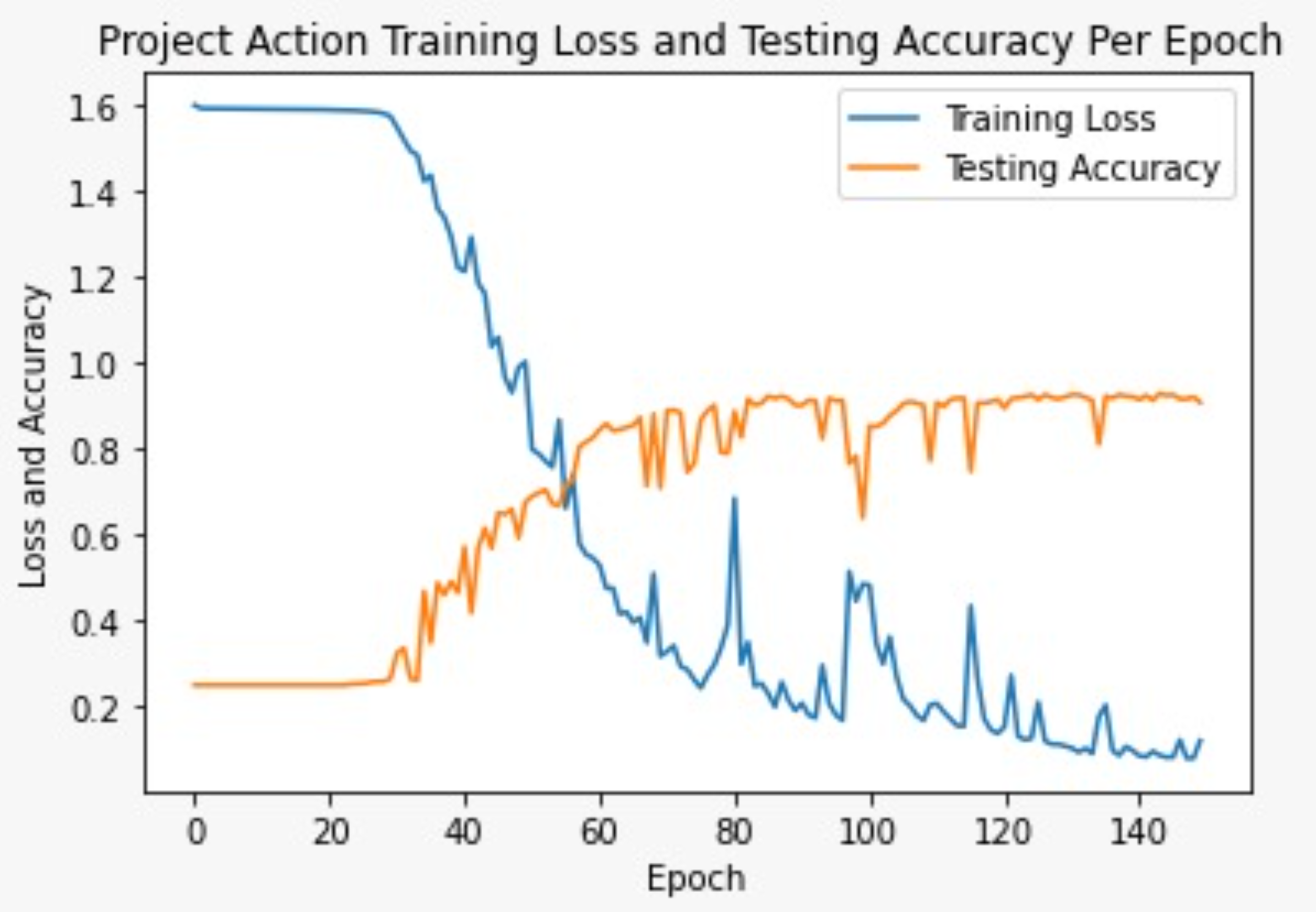


Figure 5. LSTM Project-Action Training Loss and Testing Accuracy Per Epoch

Figures 4 and 5 visualize the action models’ mean training loss and test accuracy per epoch. The same trend from the topic LSTM where there is no increase in performance for the first 25% epochs reappears. At some point, the LSTM seems to start figuring out some patterns and the mean training loss starts to decrease. Compared to the topic LSTM, the action LSTMs take longer to distinguish between its predicted classes; the task-action and project-action models take 200 and 140 epochs, respectively, to reach the same performance that the topic model reaches in 100 epochs. Furthermore, there are many more epochs where the training loss suddenly spikes up due to variance in the mini-batches used to update LSTM parameters.

Table 6. LSTM Task-Action Model Performance

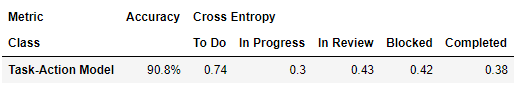
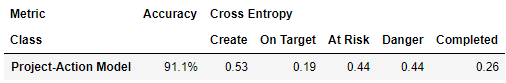


Table 7. LSTM Project-Action Model Performance



Tables 6 and 7 show the accuracy and cross entropy of the test set of the fully trained action models. The accuracies of both LSTM action models are dramatically higher than their NB counterparts with an increase of 31.1% and 26.8% for the task-action and project-action models, respectively. The cross entropy values are much better comparatively to NB as well. There are many similar patterns that hold for both the NB and LSTM, one of which is that there is greater variance in prediction capability between the classes when compared to the topic model. Furthermore, NB and LSTM both perform the worst when predicting the creation of task or projects, as reflected by the *To Do* and *Create* cross entropy values. Unlike its NB counterparts, the LSTM performs best when predicting for tasks or projects that a user has started on (*In Progress*) or gotten back on track with (*On Target*).

Overall, LSTM is an improvement over NB in terms of model performance for all three models. This evaluation holds true when using accuracy or cross entropy loss as the metric for performance. However, when it comes to time efficiency, the NB model takes a fraction of the time to train comparatively to the LSTM, even with a dataset of only 6979 instances.

**Discussion**

The NB models display the limited effectiveness of the BoW approach when parsing brief text commands to extract the topic and action. Overall, the topic model performed much better than the action models in accuracy and cross entropy values for reasons related to the data. When synthesizing commands, the topic classes *Task* and *Project* words were included in some instances, because there is a good probability that users will likely include these words in their commands. As a result, 48.7% of the commands in the data contain either the word *Task* or *Project*. With so many commands that include the class name within the text, the word frequency approach performs very well when it comes to classifying commands between the two, resulting in higher accuracy and cross entropy values for the topic model. On the other hand, consider one of the two action models, the task-action model, which has the following classes: *To Do*, *In Progress*, *In Review*, *Blocked*, and *Completed*. Only 4.0%, 28.6%, 14.5%, 22.5%, 20.0% of the *To Do*, *In Progress*, *In Review*, *Blocked*, and *Completed* classes, respectively, contain the class name within the text command. Even though there are many other factors that play a part in why the topic model outperforms the action models, this trend in the data suggests that BoW can fall short due to its simplistic word frequency approach.

The variance in performance between classes for the NB action models can also be rudimentarily explained as well by the word frequency distributions. Once again, consider the task-action model, which is the least confident when predicting the *To Do* class while being the most confident when predicting the *In Progress* class. This result corresponds to how *To Do* appears in the least percentage of text commands while *In Progress* appears in the largest percentage of text commands comparatively to all five classes. Again, use of the class label distribution in a dataset to prove differences in model performance was not tested through hypothesis tests, but these correlations do bring attention to how the NB model’s BoW performs with respect to the frequency of key words that appear in the dataset.

The NB action models performed much better in earlier iterations of the data when the data was not as enriched with more diverse commands. One example structural command that was added into all five action classes followed the pattern: “Assign status <action> to <identifier>.” The inclusion of a text command with the same words to all five action classes makes it hard for the NB algorithm to determine which class this particular command belongs to. The inclusion of data that is similar across different classes again shows the limitations of a BoW approach.

The LSTM models are an upgrade from the NB models because fundamentally, deep learning is simply more accurate than traditional ML given a large enough training dataset. Luckily, there was enough data to train the LSTM to a high enough performance within 200 epochs. Between the three LSTM models, there was a correlation between model performance and the number of epochs needed to fully train the model. The best performing topic model took the least number of epochs at 100, the next best performing task-action model taking 140 epochs, and the worst performing project-action model taking 200 epochs to decrease the training cross entropy loss to below 0.10.

As expected, the LSTM models outperformed the NB models by a wide margin with respect to accuracy and cross entropy values. Specifically, the NB algorithm was not sophisticated enough to accurately predict between different actions in its action models whereas the LSTM action models performed much better in comparison. This increased performance is the result of the LSTM’s ability to capturing the semantic meaning behind individual words as well as its consideration of all the words in a text command. The nature of word embedding allows words to be vectorized with 350 different dimensions, which provides so much more information comparatively to a single dimension word frequency value from the BoW approach. Furthermore, LSTMs have a built-in memory component to the algorithm that allows it to consider words that are sequenced earlier in the command.

The similarities and differences between BoW and word embedding is also showcased in which classes the action model is most confident in for the NB and LSTM algorithms. Once again, the *To Do* and *In Progress* classes for the LSTM task-action model have the highest and lowest cross entropy values, respectively. Furthermore, both the NB and LSTM algorithms seem to struggle to differentiate tasks that are *In Review* and *Blocked* for tasks and *At Risk* and *In Danger* for projects. This assumption in correlation is based on the similarity while generating the commands between those classes. If distinguishing between two actions is difficult and the prediction confidence is low, the user command application can ask the user whether they are referring to one action or the other. However, similarities stop there between the two algorithms because the additional information from word embedding and RNN’s memory function changes the LSTM action model’s behavior compared to that of NB. The LSTM task-action model has cross entropy values of 0.43, 0.42, and 0.38 for *In Review*, *Blocked*,and *Completed*, respectively. Meanwhile, the NB task-action model have a different pattern of cross entropy values of 1.16, 1.53, and 1.25 for *In Review*, *Blocked*,and *Completed*, respectively. A similar difference in pattern is observed for the project-action models between the two algorithms as well.

Even though the LSTM outperforms NB in terms of model performance when it comes to predictive capability, it has its downsides in terms of big data necessity and long training times. While the NB action models perform too poorly to ever be considered over its LSTM counterparts, the NB topic model has reasonable accuracy and cross entropy values. As a result, if there is even an issue with runtime or cost of hosting LSTM models, the NB topic model can be considered as an alternative to the more expensive LSTM model.

**Additional Work**

After either the NB or LSTM models were used to predict the topic and action, developing the non-ML POS identifier tagger was the final step to automating a given user text command. Returning to the earlier example command “finished task market research,” the topic is *task,* and the action is *completed*. Next, descriptive analytics is used to locate the identifier, market research,in order to automate the entire command. This process is detailed in Figure 6.

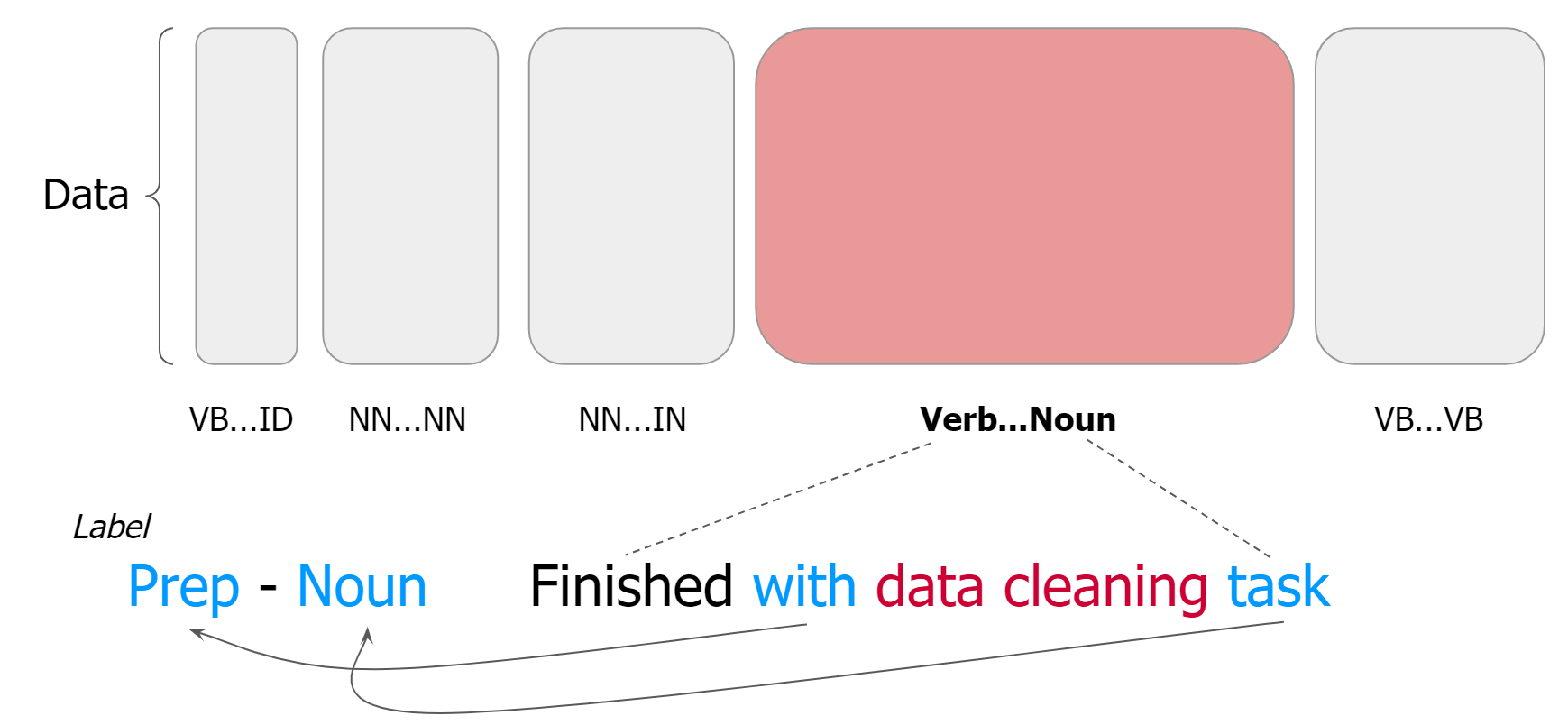
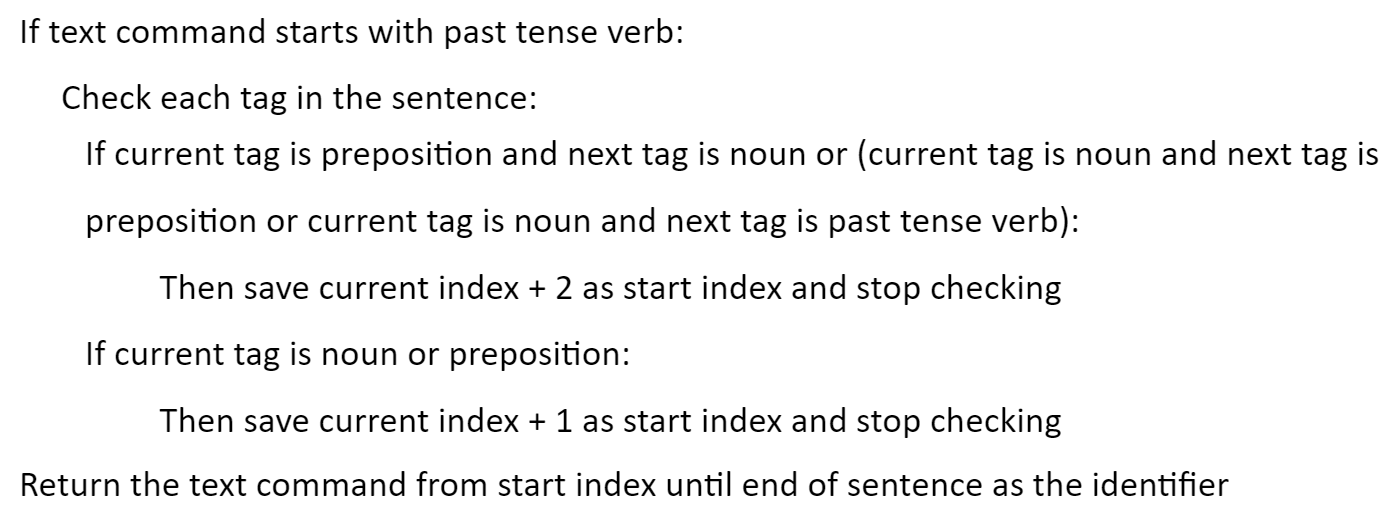


Figure 6. High-Level View of POS Identifier Tagging

The high-level view behind locating the identifier involves first tagging the words in the entire command with its respective POS to observe patterns3. Based on the POS patterns found in the data, linguistic rules were then created in order to pinpoint the identifier. Since it is impossible to develop linguistic rules through non-ML means that can locate the identifier uniformly across a diverse dataset, the data was first divided into many partitions based on the POS of the first and last words in the text command. Consider the subset where text commands begin with a verb and end with a noun and the example user command: “finished with data cleaning task.” Next, the POS of the words before and after the identifier were used to create a label corresponding to this text command; in this case, the label would be a *preposition*-*noun*. The user commands in this Verb-Noun subset were then visualized as a bar chart to see the label distribution. Based on the most common labels, the next step was to manually create linguistic rules that are partial to the most common commands and would locate the identifier. For example, consider the following pseudocode for the linguistic rule that was created for the Verb-Identifier (last word in a command is the identifier itself) subset:



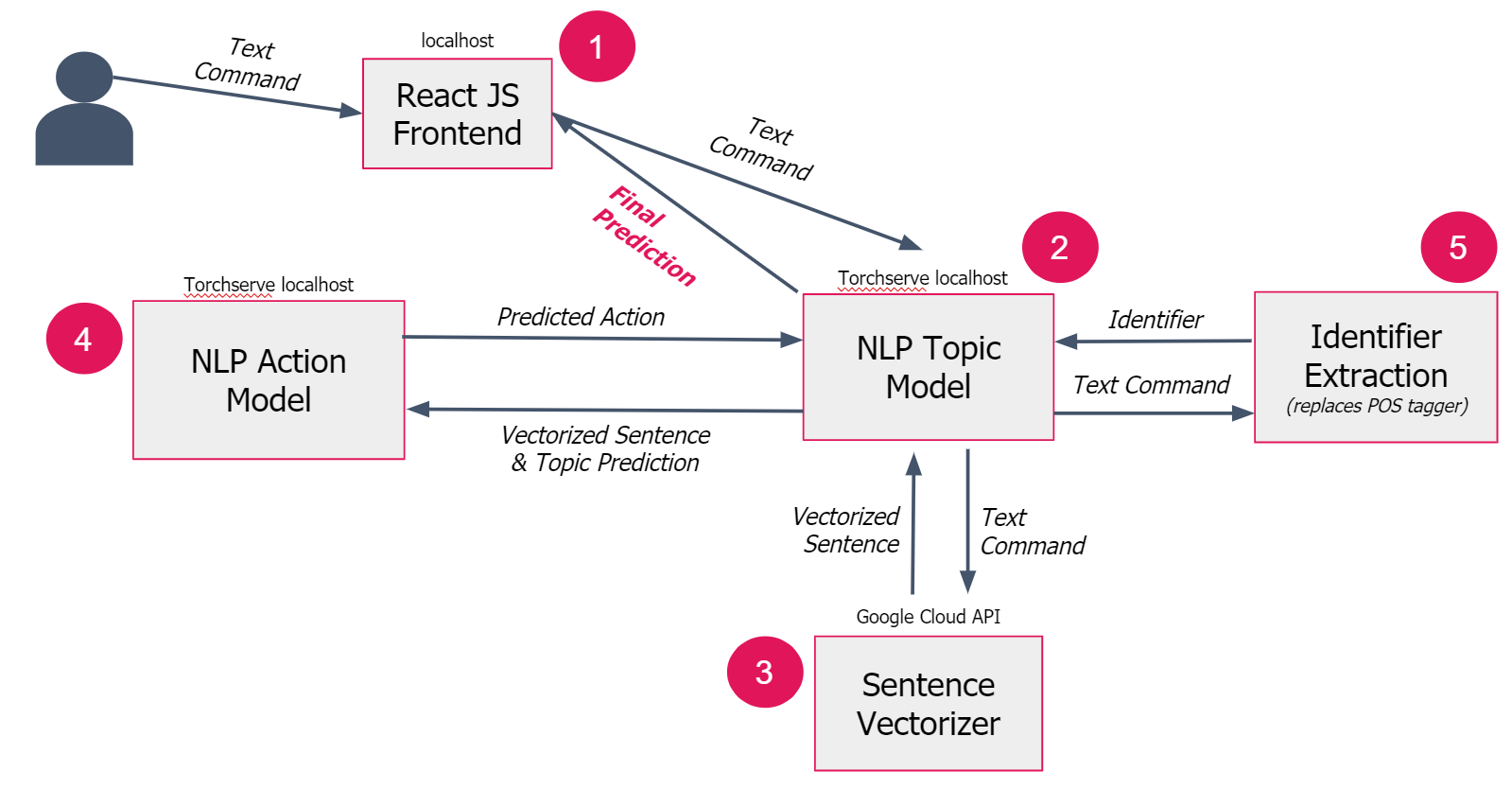
The rule above is 71% accurate in locating the identifier within text commands in the Verb-Identifier subset. The rule itself was uniquely created based on linguistic patterns found in the POS tags observed in text commands within this particular subset. There were efforts put towards brainstorming some generic rule-making system that would locate POS without the use of ML, but the diverse POS ordering of the entire dataset indicated that this approach would result in a low accuracy. As a result, in order to locate the identifier in the entire dataset, this manual and more linguistic-based process would need to be repeated on every unique data split. Lemmatization and stemming were attempted on the data in order to improve the accuracy of the linguistic rules3. However, these methods had lackluster results because many times the transformed word does not have a different POS, which is what the rules are based off of. Due to the time frame of the project, and the fact that in a business setting, locating the identifier would have a 100% accuracy if the user surrounded the identifier in quotes, further optimization of the accuracy of these linguistic rules was not the focus. Note that data was enriched after the creation of this POS identifier tagger so the data that the aforementioned rule was created from was stored as an older version for replication purposes.

Now that all three parts of a text command can be located, there were two final components to the project that needed to be addressed to improve user experience. The whole purpose of the user command application is to save the user time from navigating through Product X in order to update the status of a task or project. However, there are scenarios to consider where the application makes a wrong prediction or decides to make no prediction (when the user inputs an unrelated command like “Hello World”). As a result, a formula that uses time saved as the foundation was created in order to determine whether the ML models were saving the most time for the user.

Depending on the number of correct, incorrect, and no predictions from a given ML model version, this formula evaluates how effective the model is saving time for users. Another way to look at this approach is adding a cost function to model evaluation where the cost is the user’s time. The time taken to type a text command can be extracted by recording how long it takes for users to type a text command into Product X and creating individual user cost functions; alternatively, a generic cost function based on the average time it takes to type a command for the entire userbase can be used as well.

The final step for this project was to build a full stack demonstration for the partner SaaS company to visualize how this user command application would look like in production. The front end of the application was designed using React JS in a design that mimics Product X’s layout. The ML models were then deployed using TorchServe on Google Cloud through a localhost server4. Other necessary components to this pipeline such as the text preprocessing and the POS identifier tagger were also connected to Google Cloud with APIs when needed. The steps on how a user text command is automated through this cloud deployment can be articulated in Figure 7.

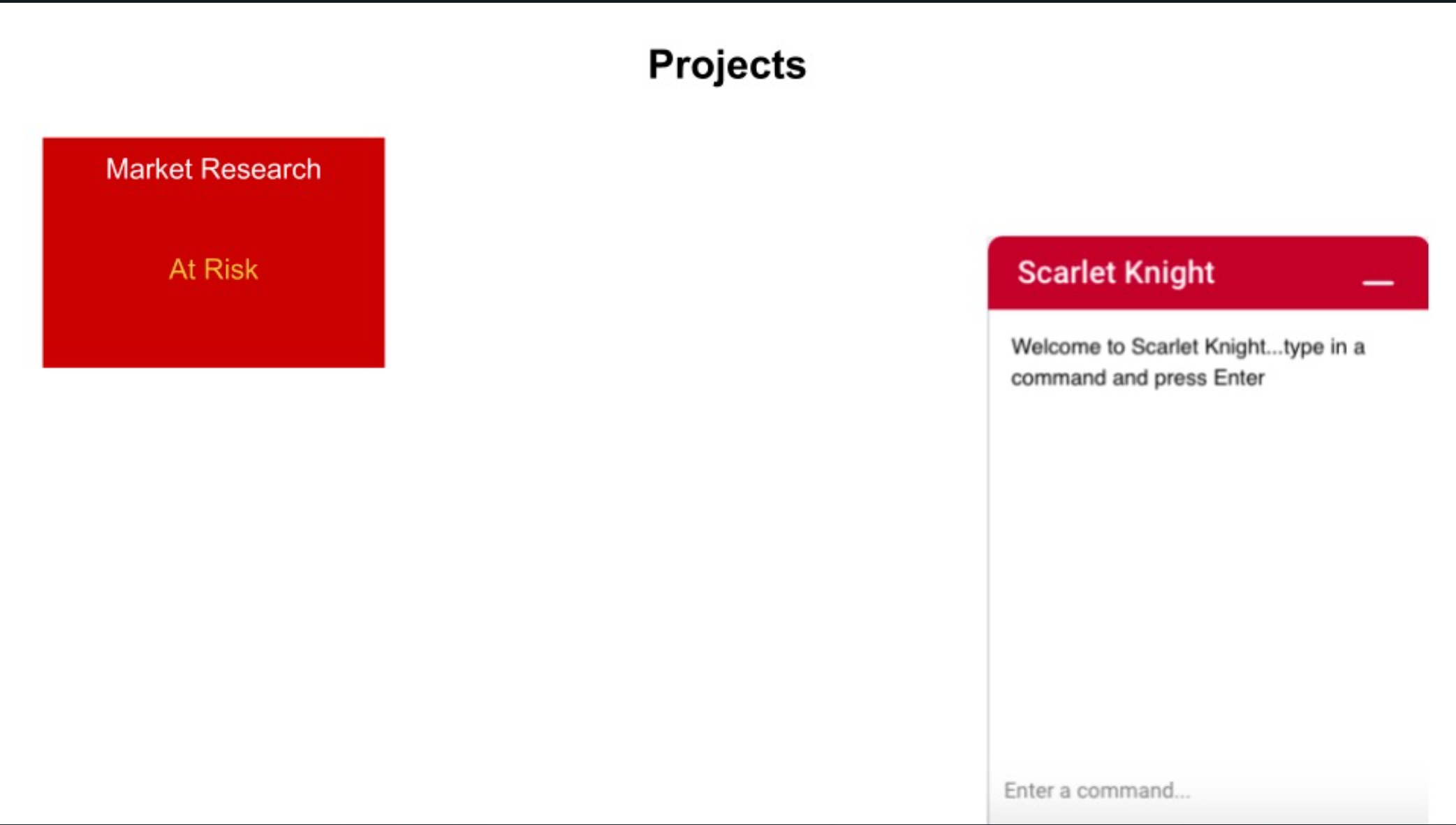
Figure 7. Prediction Process using Cloud Deployment



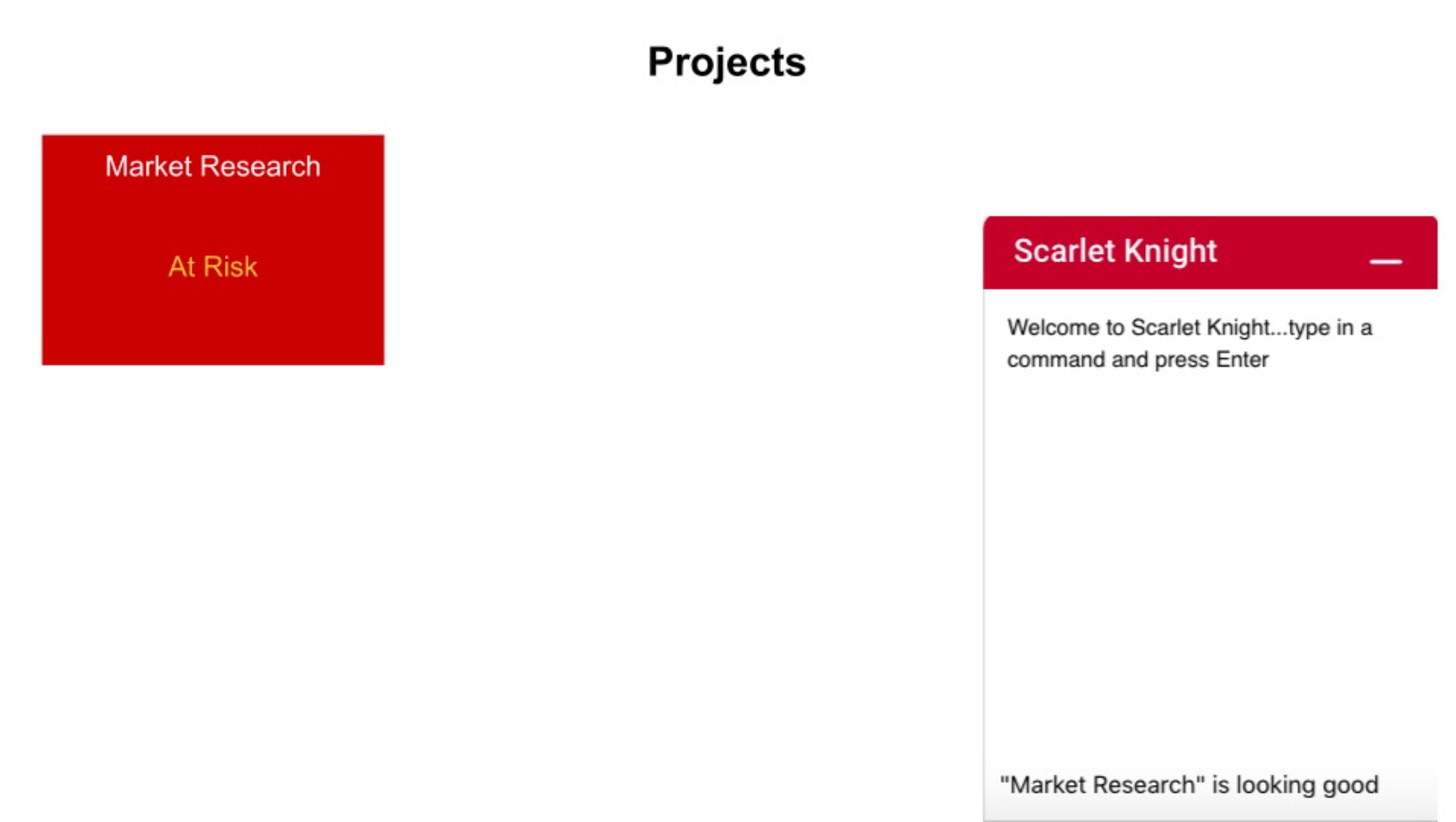
Cloud deployment can be divided into five main steps where in the first step, the user types the text command into the React JS front end. This text command is then set along to the topic model and then to the sentence vectorizer in order to pre-process the text data into a format suitable to input into the LSTM model. Next, this vectorized sentence is inputted into the topic model, which determines whether this text command is referring to a *Task* or *Project*. Depending on the prediction, the vectorized sentence is sent to its respective action model, which then predicts the action within the command. The original text command is sent to the POS identifier tagger, which in the cloud deployment case is a script that locates the identifier based on the quotation marks surrounding it. Finally, the topic, action, and identifier are all sent back to the React JS front end, which automates the user command accordingly. Note that there were many obstacles when hosting the LSTM models on the cloud due to the relatively new nature of the TorchServe platform that is available on Google Cloud. The final demonstration was hosted on a local server because of the inability to work out certain bugs preventing all the models from properly functioning on the cloud. The following screenshots show how this process would look to the user in the front end.



*Caption 1: This screenshot represents what the user is looking at while using Product X. The application manager, dubbed as Scarlet Knight in these screenshots, is integrated into the bottom right at all times. Text commands typed into this terminal are then sent to the models deployed on the cloud to be processed and for a prediction to be made. This prediction is then sent back to Product X, where the text command is automated. At the start of this demonstration, there is a project named “Market Research” that is currently “At Risk”.*



*Caption 2: The Scarlet Knight can be easily opened at all times and will pop up as shown here, similar to how text chats are in popular social medial platforms.*



*Caption 3: The user just learned that the team member that was holding the “Market Research” project back had just submitted their portion of their workload and caught up with everyone else. As such, the user is typing a command to update the status of the “Market Research” project into the Scarlet Knight.*



*Caption 4: Scarlet Knight then automates the command and returns a message about the action that was executed. In this particular example, the status of the “Market Research” task was set from “At Risk” to “On Target,” which matches what the user intended.*

**Recommendations for Future Work**

As mentioned previously, the current integration of a confidence threshold to prevent the user command application from automating an unconfident prediction is not the most effective due to the LSTM’s SoftMax layer. When the model is not very confident in a prediction, the cross entropy loss output vector reflects that numerically. However, the moment this output goes through the SoftMax layer, this lack of confidence becomes less impactful since all the values are normalized such that the new vector of probabilities sums to one. As a result, in future work, the integration of an OpenMax layer would be much more useful in stopping the user command application from automating unknown commands with a low confidence1.

**Conclusion**

Initially, the SaaS company approached the team with an open-ended suggestion asking for recommendations on how to improve Product X using NLP and ML. The various features of Product X were considered along with their pros and cons before deciding on this user command application recommendation that serves to reduce the time users spend navigating through the product. This project’s goal was to design an end-to-end proof of concept on how to manually synthesize and augment the data, design the ML models used to predict the necessary parts in a command, and build the final deployment infrastructure on how this feature is to be integrated into Product X and execute user commands. Based on model performances, the user command application has been proven to be able to save time for many distinct user commands at a reasonable accuracy. In order for this new feature to be ready for production, the accuracy and overall model performance needs to be polished for certain classes such as “Create” and “To Do” for projects and tasks, respectively. Furthermore, there are two things to consider when it comes to using cloud platforms in one’s technology stack. First, it would be whether the algorithms and platform can handle data at scale if Product X is being used at a larger company. The algorithms need to be able to process data and also provide predictions in a fast enough manner; if the user command application is not receiving a prediction fast enough from the cloud ML models, that defeats the purpose of the whole time-saving characteristic this feature is based on. The second consideration is the cost of maintaining the cloud computing instances that need to be initiated to contain the models. If the cost of running the cloud is not worth the benefits, then alternative ways to connect the ML models to Product X would need to be considered. One of the largest challenges in this project was balancing the proof of concept of the user command application with the needs of the client that lay beyond the immediate scope of the project. If this project were solely done in an industry setting, the client’s needs would be prioritized to bring about the biggest business impact.

**References**

1 Bendale, Abhijit, and Terrance E. Boult. “Towards Open Set Deep Networks.” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, doi:10.1109/cvpr.2016.173.

2 Lewis, David D. “Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval.” *Machine Learning: ECML-98 Lecture Notes in Computer Science*, 1998, pp. 4–15., doi:10.1007/bfb0026666.

3 Partalidou, Eleni, et al. “Design and Implementation of an Open Source Greek POS Tagger and Entity Recognizer Using SpaCy.” *IEEE/WIC/ACM International Conference on Web Intelligence*, 2019, doi:10.1145/3350546.3352543.

4 Pop, Daniel. “Machine Learning and Cloud Computing: Survey of Distributed and SaaS Solutions.” *ArXiv*, vol. 1603.08767.

5 Rao, Adithya, and Nemanja Spasojevic. “Actionable and Political Text Classification Using Word Embeddings and LSTM.” *ArXiv*, vol. 1607.02501.

6  Rudkowsky, Elena, et al. “More than Bags of Words: Sentiment Analysis with Word Embeddings.” *Communication Methods and Measures*, vol. 12, no. 2-3, 2018, pp. 140–157., doi:10.1080/19312458.2018.1455817.

7  Wei, Jason, and Kai Zou. “EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks.” *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, doi:10.18653/v1/d19-1670.

8  Zheng, Renjie, et al. “Robust Machine Translation with Domain Sensitive Pseudo-Sources: Baidu-OSU WMT19 MT Robustness Shared Task System Report.” *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, 2019, doi:10.18653/v1/w19-5367.