

Capstone Project Report: Mutual
Trevor Andrus, Lexi Delorey, and Lance Parrish
BYU Data Science Capstone, 2022

Executive Summary

Dating can be a difficult experience at times, which is why the app Mutual was created, as a platform on which individuals may have more luck finding partners. However, many users of the app have exhibited a tendency to use crass, sexually explicit, vulgar, or otherwise inappropriate language that doesn't align with the values of the service. Having to look at reported messages on a case by case basis is an infeasible task, and that doesn't include the number of messages that go unreported. With that in mind, Mutual asked us to use Machine Learning to create a model that would do that. Through our efforts on this capstone project, we have accomplished that.

This was accomplished using a two stage system. The first stage compares the message against a set of words which Mutual did not want users sending to each other on their app. If the message didn't contain those words, it was then passed along to the second stage. This stage entails cleaning, lemmatizing, and vectorizing the message before feeding it into a Machine Learning algorithm which would determine if the message was potentially inappropriate. If the model did not classify the message as inappropriate, we conclude that the message is free of inappropriate content.

We tried many different Machine Learning algorithms, but ultimately landed on a Random Forest Classifier, because it gave the best combination of maintainability, accuracy, and development time needed. We experimented with multiple pre-trained large language models as well, as these are state of the art in the NLP field. However, we ultimately decided against them as they would cost much more to run long term, and are more difficult to fine-tune to our specific task.

The result is a high performing API integrated with AWS as well as the product of the Software Engineering Capstone. This product should prove to be a useful tool to make the Mutual app a positive experience for users.

Introduction

Around 6 months ago Mutual presented the idea for this project: A way to automatically flag messages that are inappropriate to send and let the user know as they are sending it. A competing app had just released a similar feature, and Mutual wanted to implement their own version. The problem was more complicated than that of the competing app, however, due to the unique user base of Mutual. Many of the words and slang used on the app are not common with people outside of Utah, as well as the standard of what is appropriate is stricter. These issues made for an exciting challenge in finding a model that could understand these nuances.

We did intensive research into what kinds of Algorithms and Models would be most useful in this scenario. We investigated several state of the art pre-trained language models (all using LSTMs, specifically versions of BERT, available on the HuggingFace pipeline) as well as many simpler models discussed in class, such as Naive Bayes. The problem presented would best be understood as a branch of sentiment analysis. This is a field that is actively being researched, and we wanted to tap into this research if we could. Simple filters have existed as long as internet chat rooms have, but we needed to go a step further, catching messages that didn't have any obviously offensive terms, but still presented a negative sentiment.

The final step of the project was to host the model on AWS, as that was where the majority of Mutual's other data and APIs were hosted. We looked into several different options for that, including Sagemaker, but decided to use Lambda functions, S3, as well as the API gateway to host the finished model as that was most cost effective and provided a comparable result.

Data

We initially began training with the Hate Speech and Offensive Language Dataset hosted on Kaggle. This dataset consists of thousands of tweets labeled either offensive, hate speech, or neither. This led to discussions about whether such granularity was possible or even useful in our case. Ultimately, we decided it wasn't, and stuck with a simple 3 option response:
Appropriate, Potentially
Inappropriate, Inappropriate.

When training our production model, Mutual provided us with a huge dataset of messages users had sent, stripped of any identifying information for privacy purposes. In total we had around 33 thousand unique messages which were labeled by hand by Mutual staff members. After some experimentation and looking into the data we found several hundred false positives (such

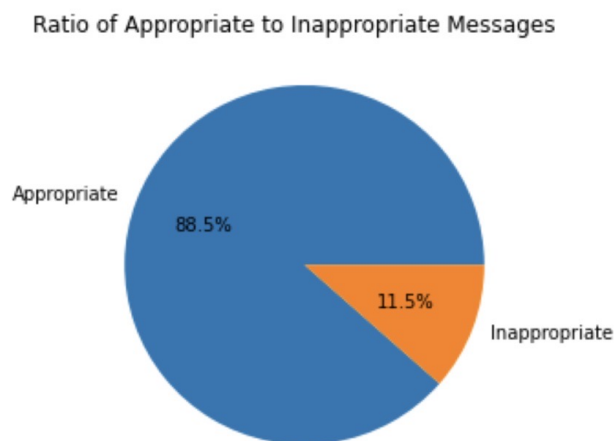


Figure 1: A pie chart of the proportion of appropriate and inappropriate messages found in the dataset given to us from Mutual. As is clear from this visual, we were working with a severely unbalanced dataset, but due to time and cost constraints we were unable to get different data.

as “I’m not temple worthy”) that were confusing the models. To fix this, we combed through the 5 thousand offensive messages to eliminate false positives.

As is clear from this experience, the data we used wasn’t perfect. The most major difficulty with the data was the disproportionate amount of messages that were labeled appropriate (see plot to the right). Language Models are most easily trained on datasets that have equal parts of all labels, since they are prone to majority label bias.

Methods

As with most machine learning projects, our first step was to clean and process the data. In the case of the dataset from Mutual, all identifying information was stripped from the messages before we received them, in order to preserve user privacy. Cleaning that information is a big task, and we are grateful to have started with a fairly clean dataset. The only cleaning we had to do was mentioned in the previous section, as well as removing any empty messages or ones that had labels that didn’t match our expected set. From there, there were several steps to process each message to make them useful for the random forest classifier.

The code shown below is the function we used to clean each message. We convert the emojis to their textual meaning, force everything lowercase, tokenize the words, get rid of punctuation, and get rid of commonly used stop words. The final step is to lemmatize the tokens. This is the process by which verbs are changed to their base conjugation to make processing easier. For example: changes, changing, and changed all become change, which allows models to create deeper connections between words that would otherwise be diluted between each form of the verb. The decision to keep the emoji text was made by our sponsors, as they noted that emojis can be used to convey offensive messages.

```
def clean_data(text):
    text = emoji.demojize(text)
    text = text.lower() # coerce data to lower case
    tokens = wordpunct_tokenize(text) # tokenize individual words
    tokens = [tok for tok in tokens if tok.isalnum()] # removing
    punctuation
    tokens = [tok for tok in tokens if tok not in sw] # removing stop words
    tokens = [wn.lemmatize(tok) for tok in tokens] # lemmatizing - reducing
    to base words
    return " ".join(tokens)
```

To determine which model to use, we tested several different common classification methods on our data. At the time we were experimenting with models, we only had access to the Hate Speech and Offensive Language Dataset mentioned earlier. Figure 3 shows the accuracies of the several methods we tried. For the most part they all had comparable accuracies, but since Random Forest was the fastest to train and had the highest accuracy, we decided to stick with it. Figure 2 shows the comparison of accuracies between the models on the Mutual dataset. This was our initial training set up, which did not include the engineered dataset that had an even split of appropriate and inappropriate messages, which is why the accuracies are so high. In practice, the models were performing at around 70-80% accuracy.

There are few models we tried after receiving the Mutual dataset. These are the HuggingFace sentiment analysis pipeline and LightGBM. Neither of these performed significantly well. The sentiment analysis pipeline averaged a 45% accuracy when testing on the data alone, and a 66% accuracy when testing on the data with the dictionary of flag words to remove any obviously inappropriate messages. Light GBM, on the other hand, performed well in training (mid to high 80s in accuracy), but was consistently getting 50% or less in testing. It is clear that the model was overfitting the training data, but even an early stopping criteria did not fix this problem.

After all this analysis, we settled on a Random Forest Classifier. Random Forest Classifiers are models that are built up out of many decision trees, each created based on the training data. Each message in the training data is encoded in a tree as a collection of attributes, having certain words. New messages are then indirectly compared to those messages, and based on its similarity the tree determines if it's inappropriate or not. The Random Forest is a collection of many decision trees, each with slightly different encodings. The final result is a vote of all those trees, and as such is usually more accurate than just one decision tree. However, because of the random nature of the trees, we train the model several times until we see an acceptable accuracy to ensure that only the best trees are in our forest.

Now, having a fully trained model, we implemented our two step approach. The first step checks each word in the message against our dictionary of immediate flag words. Using a python dictionary for this lookup, we are almost certain that this lookup will happen in constant time, though we recognize that python dictionaries have a worst case lookup time of $O(n)$. Even with that in mind, the dictionary seemed to be the best approach. If a flag word was found in the message, it is flagged as inappropriate, and that label, as well as the flag word found, are returned to the user. If no flag words are found, the message is cleaned using the `clean_data` function above, and then vectorized. It is important to note that we used the

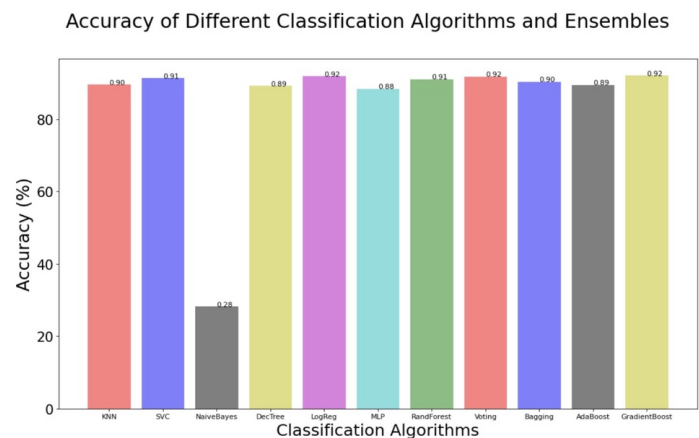


Figure 2: The accuracy of several models in classifying the Mutual dataset

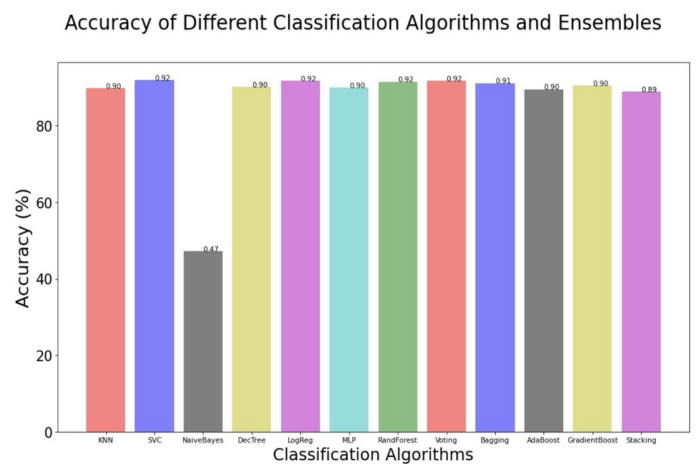


Figure 3: The accuracy of several models in classifying the Kaggle dataset

same CountVectorizer in this step as we did in the training and testing, to ensure messages are all vectorized in the same shape and manner. This message is then passed to the classifier. If the classifier finds the message inappropriate, the label “Potentially Inappropriate” is returned to the user. We chose to return potentially inappropriate as opposed to inappropriate because we recognize the model can make mistakes. We hope that potentially inappropriate messages will be reviewed by staff members if a user collects several of them, to ensure that any action taken against them for inappropriate language is valid. If the classifier doesn’t find the message inappropriate, then the label “Appropriate” is returned to the user.

Results

Overall, we created a model that, when used in conjunction with the flag words dictionary, correctly classifies messages as inappropriate 82% of the time. When considering the data we were working with, as well as the nuances in what is inappropriate and appropriate, we are very proud of this result. In addition to this, we created an API that, after an initial cold start, classifies the message in under 200ms. This is within the realm of appropriate latency that was set out for us in the project description.

The images below show examples of the response the frontend of the Mutual app will get back when pinging our API, and the time it took to get a message back. The “Inappropriate” example also returns the flagword that caused the message to be classified as inappropriate. Conversely, the “Potentially Inappropriate” example does not do this, since all messages labeled potentially inappropriate do not have any flag words in them.



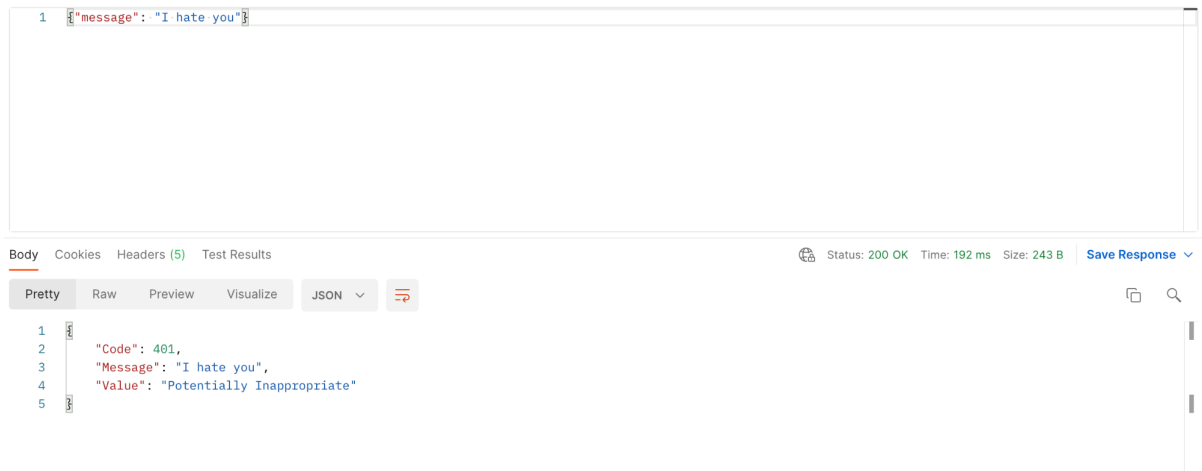
The screenshot shows a REST client interface with a request body containing a JSON object: `{ "message": "What's the deal with airline food?" }`. The response status is 200 OK, with a time of 159 ms and a size of 253 B. The response body is displayed in a pretty-printed JSON format:

```
1 {
2   "Code": 400,
3   "Message": "What's the deal with airline food?",
4   "Value": "Appropriate"
5 }
```



The screenshot shows a REST client interface with a request body containing a JSON object: `{ "message": "I could sure go for a ncmo right now" }`. The response status is 200 OK, with a time of 107 ms and a size of 279 B. The response body is displayed in a pretty-printed JSON format:

```
1 {
2   "Code": 402,
3   "Message": "I could sure go for a ncmo right now",
4   "Value": "Inappropriate",
5   "Flag_word": "ncmo"
6 }
```



```
1 {"message": "I hate you"}

Body Cookies Headers (5) Test Results
Status: 200 OK Time: 192 ms Size: 243 B Save Response

Pretty Raw Preview Visualize JSON
1 {
2   "Code": 401,
3   "Message": "I hate you",
4   "Value": "Potentially Inappropriate"
5 }
```

Conclusion

To conclude, we achieved the original goal within the time, budget, and development limitations we were given. Mutual now has a working API with a fast response time and high accuracy in classifying messages. Despite this success, Things can still be improved, and we'd like to prescribe a few steps that could be taken to achieve lower latency or higher accuracy.

The first is to add more words to the initial filter. This would improve latency, since dictionary lookup is faster than the random forest classifier. A downside of this is the potential to lower accuracy, since flagging any potentially offensive word could lead to an increase in false positives.

The second is retaining the random forest with fewer decision trees. As the model currently stands, there are 100 trees in the forest. Having less trees would decrease latency in classification from the model, especially if the number of trees was decreased significantly. As with our last suggestion, this could also lead to a decrease in accuracy. Having less trees means less chance to build the exact right trees. We would only recommend this if response time is significantly slow. Increasing the number of trees would be a good approach if accuracy needs to be increased, but that would also decrease latency.

Third, it should be noted that adjusting the dataset and the model to account for the context of the entire conversation could increase the ability of the model to catch more offensive language, but due to the extra complexity this would have incurred, this was out of scope of the project. The model we used would not be appropriate for this, with most similar projects using large recurrent neural networks, which would take more time to design and train, as well as much more processing power to run.

Finally, we would recommend that over time, flagged messages be recorded and reviewed. Our model certainly isn't close to perfect, so it would be wise to occasionally review a sample of classifications to ensure they are accurate. In addition to this, as users file reports for abusive language, the API should be tested to see if it catches the messages in question. If not, an updated dataset should be created and the model retrained after a fair number of them have been compiled. We believe this will be the most effective long term maintenance, as the one area we struggled in was a lack of labeled data.