Chatbot Development Summary

Purpose:

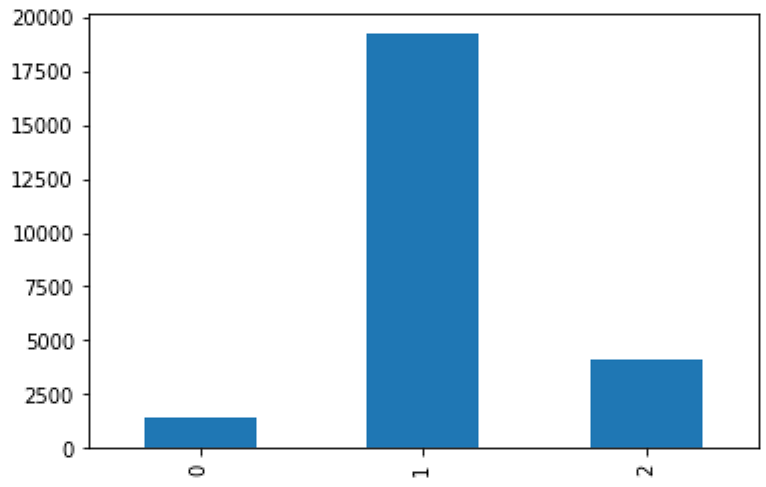
The Buttersbot chatbot was intended to help moderate chat rooms in discord using machine learning to help identify hate speech nonobvious offensive speech. Most bots currently have simple wordlist filters that can find obvious violations whereas language is more complex.

Data Sources

The cleanest dataset that seems recent enough to work with this problem is from Kaggle based on the paper "Automated Hate Speech Detection and the Problem of Offensive Language." by Davidson, Dana Warmsley, Michael Macy, and Ingmar Weber (2017). However, as explained later, a second generic hate speech dataset https://www.kaggle.com/mohit28rawat/hate-speech had to be introduced to solve some balancing problems.

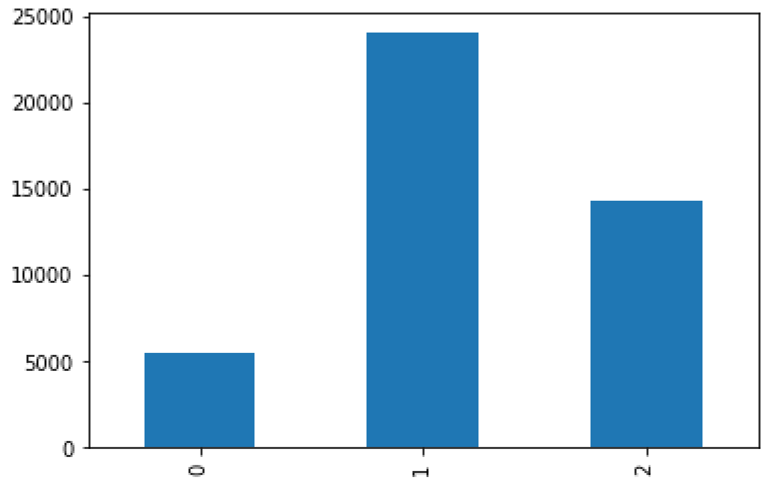
Data Wrangling

https://github.com/jeffnb/springboard-capstone/blob/master/Capstone%20-%20Data%20Wrangling.ipynb
The first dataset was imported into the system via a csv file which was nice and clean. Pandas parched it and had a few columns but we only cared about the class it was categorized as. The original id came in unnamed but was simply renamed to id. There were no bad rows without classes or with invalid classes. However, the dataset was extremely unbalanced with hate speech having approximately 8% of the number of samples as offensive speech. Neither, which represents innocuous tweets was about 15%.



Balancing

The classes being so imbalance meant that something had to be done to increase the relative sample sizes. Another dataset was found on Kaggle that was the same format as the first but with different samples. This dataset had fewer columns but still came out clean with minimal transformation needed to be ready for the next steps. Both datasets then had the columns renamed to match each other. In order, not to worsen the balancing problem, the “offensive” class was filtered out of the second dataset and the remaining 2 classes concatenated onto the first dataset. While not eliminating the problem it did greatly help balance.



Initially, the new dataset looked fine but there was a problem where the main index picked up duplicate ids. It was easy to mix and actually caused some huge problems when machine learning was introduced. While never nailing down the exact reason in the sklearn code, it does appear that when the train and predict happen the internals look to the id to determine if the predictions were correct. The initial runs were 20% lower because of this problem. In order to fix the index on the combined dataset was simply remade dropping the old one. Finally, the dataset was put into a pickle file.

Preprocessing

The samples need to be converted into a standard format. The dataset being tweets meant that it needed to be cleaned up considerably. For a tokenizer, TweetTokenizer from NLTK was used to break up the tweets correctly with mentions and URLs as individual tokens. A number of custom functions, as well as several Gensim preprocessing methods, were used to clean the data for feature processing:

- Lowercase
- Strip extra spaces
- Remove urls
- Remove mentions
- Remove punctuation
- Remove retweet
- Convert accents
- Expanding contractions
- Remove stop words
- Stem

Vocabulary

The words in the corpus were turned into a frequency list then a simple cutoff was placed on any words that occurred less than 5 times in the corpus. The words were then dumped to a pickle file. Several reasons to do this exist. First, the long tail was really long of words and this cuts way down on processing required. Second, if a word only occurs a couple of times the algorithms won't do well trying to understand how it works with the classification.

Additionally, in order to see if extra performance can be gained from the algorithm 2 alternate vocabularies were created, one smaller with a cut off of 10 and one larger with a cut off of 3. These were also saved.

Word2Vec

As a second experiment, a large word2vec word vector dataset trained on twitter was used to attempt to get better results with more context. The word2vec was trained on the corpus from the preprocessing and intersected with the larger dataset. This was then dumped to be used later.

Deep Learning

Finally, since deep learning requires, and sometimes benefits from less preprocessing a separate preprocessing was done. The largest change was the removal of stop words no longer being needed. This separate corpus was also saved to disk.

Features

https://github.com/jeffnb/springboard-capstone/blob/master/Capstone%20-%20Features.ipynb
In order to get the most out of the project for NLP 3 tactics were tried for creating features to be used.

Bag Of Words

The first feature method is to create a bag of words. This is a common and long-running way to try to evaluate the importance of the words in a given sample with a simple frequency of every word's occurrence in the vocabulary. While this creates an explosion of features it is pretty common.

Using the CountVectorizor from scikit-learn with the vocabulary list from the preprocessing a count matrix is created of every sample and every word. Grabbing the feature names from the vectorizer as columns, the dataset is converted into a pandas data frame. Finally, the data is merged with the data frame loaded from the data wrangling and saved.

TFIDF

Term Frequency Inverse Document Frequency is widely used in search and deals with a more complex formula that looks at both the number of times that a given word appears in the sample as well as the number of times a word appears within the entire corpus of words.

The process is almost exactly the same as Bag Of Words except it uses the TfidfVectorizer. The vectorizer produces a matrix of floating-point numbers between 0 and 1. After which the same process is used to create a data frame that has the new data and original data together.

Word2Vec

The word2vec model created in the last step is already partially machine learning. Each word in the vocabulary has 200 features that are trained numbers that represent the word in a massive corpus. For each of the tweet samples, the vectors of each of the words in the sample are then averaged together to get 1 200 point vector representing the entire tweet.

As with the last two methods, the data is merged with the data frame from the original data wrangling exercise giving the ability to run on the set in the machine learning section.

Machine Learning

By far the most time-consuming step, it was chosen to do 3 different models for each of the three different data types from the previous feature creation. The ended up with 9 total all of which had to have hyperparameters tuned. Each different dataset was split with a test size of 20% to ensure accurate testing

Logistic Regression

This method tries to fit a line in multidimensional space to the data samples so that it can predict the different classes. Initially, this method was easily the most promising in all three datasets. Even with small amounts of tuning it was able to come up with decent numbers. Two of the three of the datasets used class weight balancing to account for the small numbers of hate speech samples compared to the other to classes. Word2Vec actually performed better without the balanced setting. Each one was hypertuned to try to squeeze out the best performance.

Random Forest

This method builds hundreds of small decision trees that all are somewhat bad and predicting the outcomes then combining them in a way that makes it possible to develop an accurate algorithm. This method never really did perform up to the other two on any of the datasets. Though it proved to be the fastest to tune the parameters with a large number of varying combinations in the grid search.

XGBoost

This method is the way that most people have won Kaggle competitions using a boosting methodology to determine how to classify a sample. The winner for most accuracy went to this category of algorithm. The downside was that the algorithm was extremely slow to run the grid search on. Many times taking over a dozen hours. However, once tuned it was 1-3% more accurate than any of the others.

Deep Learning

https://github.com/jeffnb/springboard-capstone/blob/master/Capstone%20-%20Deep%20Learning.ipynb

Here we took the leap into a couple of different neural networks to see if deep learning could help with classification.

Data Preparation

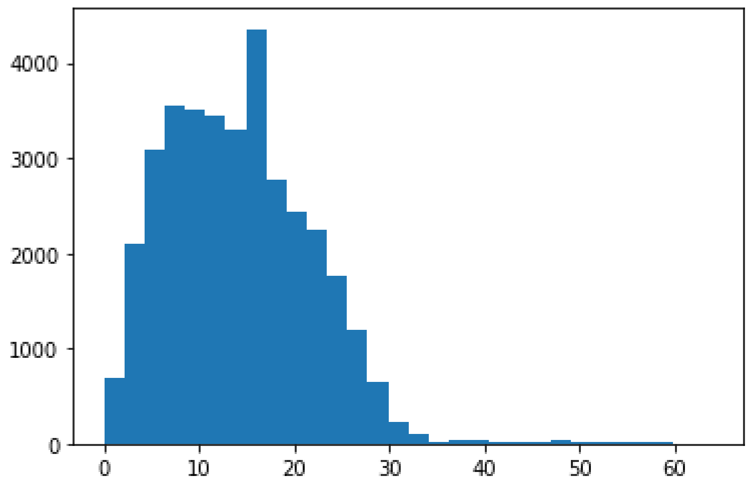
One of the most convoluted problems that was faced is the fact that pandas series have some strangeness with indexes and ordering. It caused the classification numbers to mismatch the features. To prevent this the class values were converted to a list.

Feature Engineering

The flow for the following was taken almost step by step from Deep TL hack session by DJ Sarker. The first step was to use the tokenizer from Keras' preprocessing module. The tokenizer is fit against the training data and a padding place holder <PAD> was added. The neural networks need numbers, not words to work with so the tokenizer is used to convert the words into sequences of numbers.

Padding

At this point, we take a look at the histogram of how many words are in each tweet so that each sample can be locked into the same length of tokens. In summary, anything shorter than the number of tokens will be padded with tokens and anything longer will be truncated.



Picking 34 seems to cover almost every scenario which seems realistic given the limitations of Twitter. By using the Keras method pad_sequence call the sequences were all pushed to be a unified length.

Embedding

For NLP models one of the first layers is an embedding layer for the neural network to learn how the words relate to each other. We need to prepare the embeddings to use with training. For this, we used the glove twitter fast text file. Each word has 200 different values for the vector so it should give decent results. First, the embeddings need to be loaded up so that there is a dictionary of each word and the array of embedding values. Once this is accomplished then it needs to trimmed down and shaped for the vocabulary data that exists in the corpus. To do this the mean and standard deviation are calculated over the entire set of vectors. An entire matrix of values is created by randomly assigning values within the normal distribution. Finally, the fasttext vectors are used to overwrite all values that exist leaving just the unknown words as random but within a norm of the data.

Encoding Multiple Classes

The way to prevent skew with multiple classes, the classes need to be one hot encoded so instead of 0, 1, 2 it is a matrix similar to Bag Of Words.

Bidirectional LSTM

The first architecture built for this problem was a bi-direction LSTM. With the embedding layer and 2 different LSTM layers to try to understand the context of the tokens. The rest of the architecture is standard dense layers with dropouts. The first struggle was getting the embedding layer to accept the input given the output. In the end only input_dim, output_dim, and input_legth were needed. Without these correctly set the model would throw errors about incorrect shape which turns out to be the form of many errors in neural networks. The second challenge was around the Attention layer. In the original, it was a custom layer that was inserted after the second LSTM step. Keras now has a built-in attention layer but will need to get more help with its use as every way tried returned an error in shape. The output of the model was much better in precision on the hate speech category than most of the machine learning models but not all of them.

Convolutional Neural Networks

This architecture uses a couple of layers of Convolution after the embedding layer to try to get a better handle on context. The max pooling layers collect the data into smaller blocks of data to try to get features from the data. Finally, several layers of dense and dropout make the model more reliable. This CNN model was several points higher in precision in hate speech than the LSTM model but still 2% lower than the best ML model. The biggest struggle with this model was understanding the way max pooling layers change the data as it flows through the network. If the pool_size is ever larger than the input_length the pooling will not be able to divide the tokens and it will throw an error.

Solution

In the end, the best results for hate speech came from the combination of the bag of words and XGBoost. For the hate speech classification, it managed to get an F1 score of 0.73 which is at least 3% higher than the others. Unfortunately, while respectable on the scoring, it was challenging in practice. Several false, offensive words were connected to hate speech as they often appeared in rants in the twitter data. Several other phrases that are clearly offensive weren't classified as offensive either. The reasons intuited for this are described in more detail down below.

Learnings

Data

The data sets used were both based on Twitter posts to determine hate speech and offensive language. While this did provide some acceptable data it also makes training difficult. The required 140 character limit forced users to do strange abbreviations that are not common and can be arbitrary. The dataset is intended to be used in chat conversations which are much closer to long-form English. Additionally, the entire dataset used only had 44,000 samples in it which just isn't enough to be trained well especially in the models. Another issue with the data is that its intention was to identify offensive vs hate speech. This focus leads to almost all of the data being offensive and a much smaller subset being clean or hate speech.

Another hypothesis as to why the models that typically do well underperform is that hate speech is highly contextual. Some words are universally considered racial slurs but a majority of hate speech is more subtle. Twitter makes it more difficult to identify since many of the messages are short. Additionally, swear words that are not hate-speech but commonly used in the same rant tend to get classified in the hate speech category.

An alternate dataset was found with had multi labeled data on forum comments. This data is more closely related to the use case of a chatbot than the twitter data.
https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge

Pandas Indexes

Several times in the project indexes were an issue. Scikit-learn seems to make use of these indexes quite a bit. In the event that the indexes are duplicated or not aligned the resulting learning is extremely poor.

The first time this occurred is in combining the data sets to help bolster numbers in the hate speech and clean categories. The index numbers for some of the samples were duplicated. All models severely underperformed in this case. The theory is that when the comparisons against the predicted values were done the system didn’t know which index to pick and so wouldn’t get the right one.

The second time this was an issue was with deep learning where a series was used. The series has an index as well and the ordering done for the predicted values seemed to shuffle based on that index making the data misalign. The simple solution was to convert it to an array or a list.