NLP with South Park

Milestone Report - Machine Learning

Taylor Willingham
September 10, 2019

Overview

With the word corpus established and data explored, this report covers the process of using the data set for machine learning and trying to build a predictive model. At a high level, this involved using a preprocessing algorithm to convert the corpus into mathematical vectors, and then using those vectors to train various algorithms for prediction. Unfortunately, as addressed in this report, the results so far have not been very promising. Regardless of the algorithm used, all of the accuracy scores have been fairly low and not at the level that I was hoping to see.

Initial issues to confront

After an initial exploration of the training data using CountVectorizer and Multinomial Naive Bayes, I quickly realized that a high level of accuracy was going to be difficult to achieve. In an attempt to confront this, I had to back-track a bit and dive deeper into the data preprocessing. I felt the word corpus might need a little more shaping, so before moving further with hyper-tuning and testing I went back to make a few more tweaks.

To start, I found a python package to expand and split contractions. I initially wasn't sure if this was necessary, but I decided it would be better to treat examples like "don't" vs. "do not" as the same word pattern. On top of that, I also used the NLTK library to lemmatize all verbs and nouns. As opposed to stemming, this translates all variations of a word to a single uniform root, decreasing variations in the data. As a consequence, there are less features once the vectors are created, hopefully equating to less noise.

Finally, with an adjusted word corpus, I also needed to go back and reassess the list of stop words. Primarily because some words in the original list were no longer in the corpus, and also because the word frequencies had changed. Without contractions and variations of the same root words, the stop word list is now smaller, but hopefully more accurate.

```
sw = ['be', 'you', 'i', 'to', 'the', 'do', 'it',\
        'a', 'we', 'that', 'and', 'have', 'go', 'what',\
        'get', 'of', 'this', 'in', 'on', 'all', 'just',\
        'for', 'he', 'know', 'will', 'but', 'with', 'so',\
        'they', 'now', 'well', "'s", 'guy', 'u', 'come',\
        'like', 'there', 'at', 'would', 'who', 'him',\
        'them', 'his', 'thing', 'where', 'should', 'an',\
        'please', 'maybe', 'their', 'even', 'any', 'than']
```

Establishing the Vectors

Before any machine learning could be executed, the word corpus needed to be converted into numerical vectors. There are different tools to do this, but I experimented with both CountVectorizer and TfidfVectorizer.

With CountVectorizer, basically a simple bag-of-words dictionary is created, and each observation in the data is converted to a vector containing word counts as determined by the vocabulary in the dictionary. Once the observations are converted into vectors, the subset of training data can be used to train a model. The first model I tried was set up using a Multinomial Naive Bayes classifier. Of course, the results were not very good.

To try and improve the model, I focused on two things: experimenting with different n-gram ranges when establishing the CountVectorizer, and searching for the best value of *alpha* for MultinomialNB classifier. After experimenting, I concluded the basic bag of words n-gram range of (1,1) performed the best, as anything more only worked to create more features and noise without any added benefit. And unfortunately, even using GridSearchCV to find the best value for *alpha* did not improve the results by much. With all the adjustments, the accuracy was still only at 41%.

Tf-idf

In an attempt to improve upon CountVectorizer, I also used TfidfVectorizer, which weights words, or features, based on their frequency and document frequency. In essence, the observations are converted into vectors, but instead of a simple bag-of-words, different words have different degrees of importance.

However, even though tf-idf often out performs CountVectorizer, that was not the case here. When combined with a MultinomialNB algorithm, the accuracy score was slightly lower than the previous one, which was disappointing to see.

Trying other algorithms

Since the results with MultinomialNB were less than ideal, the next logical plan of action was to try some other algorithms to try and find one that would perform better.

The first one I tried was Random Forest. Using the oob (out of bag) score to validate the performance of the trained model, there was no improvement. The next one was a variation of Support Vector Machines using Scikit Learn's LinearSVC. Again, even after tuning the hyper-parameters the accuracy only improved to 43%. Finally, I tried multiclass Logistic Regression, but the performance was similar.

Further thoughts

Regardless of the model used, the results always fell within the 40-43% range. Perhaps a Neural Net or some other more complex model might be able to improve the outcome, but

from all of the different techniques I tried it doesn't seem likely. Perhaps, for this data set, maybe there just isn't enough differentiation between the classes, or maybe having six target labels is too many. It also seems plausible that document length could be an issue. If a large portion of documents are only one to three words long, that might complicate the predictions. Under the current set up, it doesn't seem that we can reliably predict characters based on dialogue.