NLP with South Park

Final Report

Taylor Willingham
September, 2019

Overview

This project combines Natural Language Processing and classification to predict characters when given a line of dialogue. Specifically, the focus here is to test whether or not a line of dialogue from the show South Park belongs to Cartman. He has the most lines by far and, from simply an anecdotal perspective, has the most distinctive delivery. With that in mind, the hope is that we can build a supervised machine learning model to successfully predict which lines are associated with this character. The main steps were to create the word corpus and preprocess the text -- including remove punctuation, expand contractions, and lemmatize the words --

convert the processed lines of dialogue to vectors with CountVectorizer or TfidfVectorizer, and, finally, use those vectors to train various models in an attempt to find the best one. The algorithms used include Logistic Regression, SVM, Boosting and others. In the end, while all of the algorithms tested exhibited similar levels of baseline accuracy, between 86% and 87%, I think SVM out-performed the others in terms of prediction and speed.

The methodology will be explained, but here are the final results:

Algorithm	Accuracy	Target F1 Score	Runtime
MultinomialNB - with CountVectorizer	0.863	0.18	527 ms
MultinomialNB - with TfidfVectorizer	0.862	0.03	246 ms
Random Forest (balanced weights)	0.767	0.36	1 min 17 s
SVM (adjusted weights)	0.786	0.39	1.1 s
Logistic Regression	0.867	0.21	24 s
Gradient Boost	0.866	0.06	1 min 48 s
AdaBoost	0.761	0.36	2 min 30 s

Project Concept

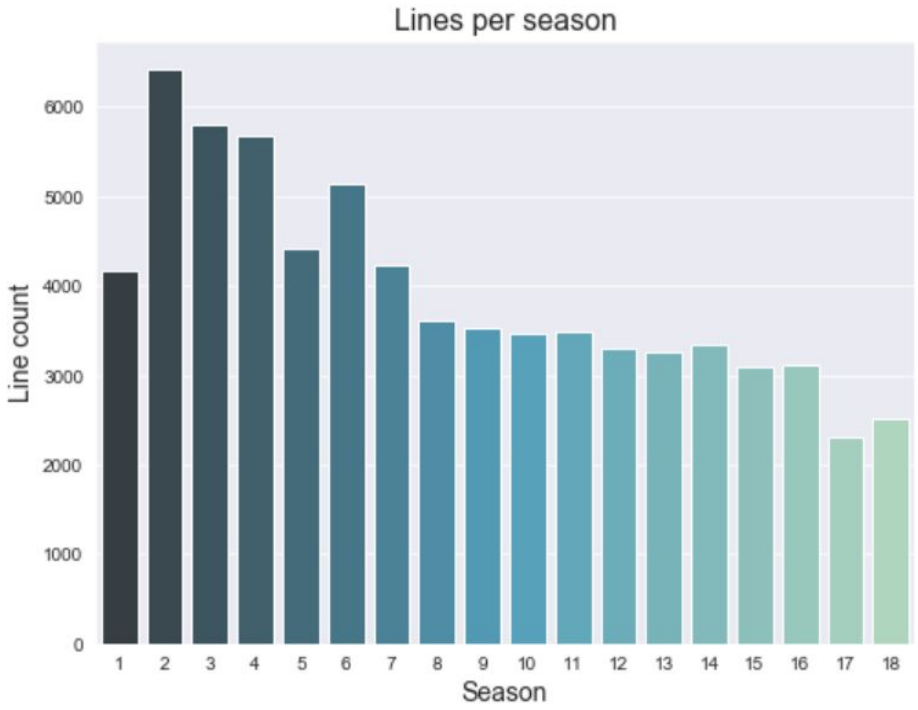
The general intent for this project was to dive into the methods and techniques of natural language processing to test their effectiveness in terms of classification. In order to do so, I chose a data set of scripts broken down line by line from episodes of the television show South Park. With this data, my aim was to train a model that, when given a line of dialogue, would be able to accurately predict which character is speaking.

Now, let's be clear: this is more of a fun passion project versus anything substantial. Knowing which South Park character is speaking probably isn't very important, and if you're hearing the line for the first time you can probably already see who is delivering the line. However, the techniques used here have infinite potential applications. It boils down to a classification problem, where labels are predicted based on text. This could be used to sort types of documents, categorize articles by subject, seek out examples of plagiarism, and countless other things. Not to equate South Park with Shakespeare, but it's not all that different from scholars trying to

determine whether or not an anonymous piece of literature is from the hand of the great poet and playwright. So, even though this project focuses on a comedy TV show, the potential implications reach much further.

The Data

As for the data used for this project, it can be found here. Each line of dialogue is treated as a separate observation. Fortunately, South Park is a long running show, so there are quite a few episodes to source from, and this data set contains dialogue covering the first 18 seasons. Here's a rough idea of how many lines, or documents for the purpose of this project, there are within the data:



There are only two columns that really matter: the character column, which is the label of interest, and the line column, which contains the actual dialogue. In addition to these, there are two other columns for the episode number and the season number for when the line occurred.

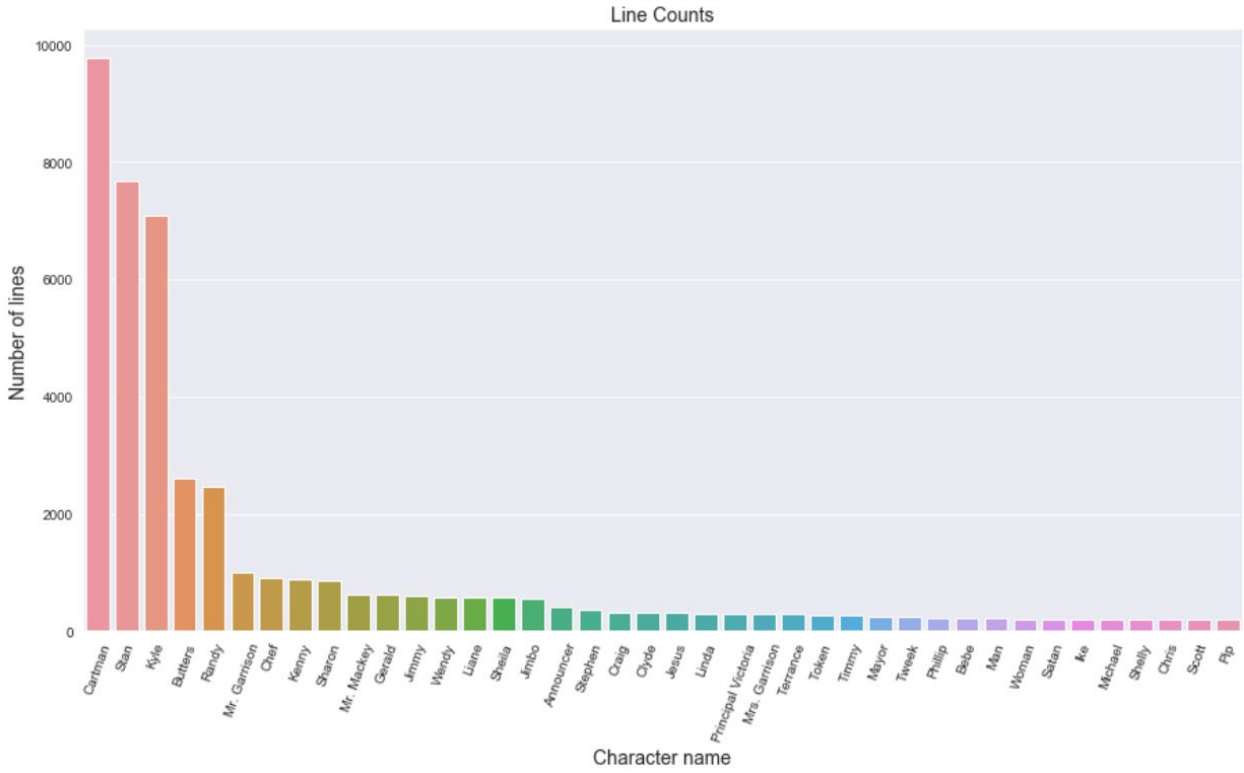
	Season	Episode	Character	Line
0	10	1	Stan	You guys, you guys! Chef is going away. \n
1	10	1	Kyle	Going away? For how long?\n
2	10	1	Stan	Forever.\n
3	10	1	Chef	I'm sorry boys.\n
4	10	1	Stan	Chef said he's been bored, so he joining a gro...

Establishing the Question of Interest

Initially, without any exploration, the idea for this project was to be able to predict all characters. However, a quick count of the labels revealed nearly 4,000 unique characters. This makes sense given that South Park, like any show, has many temporary and minor characters. However, this is way too many labels to deal with, so a decision needed to be made. We can focus on a particular subset of characters and disregard the rest, we can group characters based on certain criteria, like primary versus secondary characters, or we can compare one character to the others to make it a binary classification problem.

I toyed around with the idea of just focusing on the main characters and disregarding the rest, but this still proved to be problematic. In the end, I decided to narrow the focus to Cartman, comparing his dialogue to all other characters.

Here is a visual distribution comparing the 40 most common characters according to their appearances within the data:



We can see above that Cartman clearly has the most lines, and one could argue that he is the most prominent character.

Data Wrangling and Feature Extraction

Because machine learning algorithms only work on numerical data, the ultimate goal is to convert the dialogue to numerical vectors. This was achieved through various text preprocessing steps, and then using Scikit Learn to create vectors from the adjusted corpus. However, the first step was to create a column to identify whether or not lines of dialogue belong to Cartman. Here's what that looks like:

	Season	Episode	Character	Line	is_cartman
0	10	1	Stan	You guys, you guys! Chef is going away. \n	0
1	10	1	Kyle	Going away? For how long?\n	0
2	10	1	Stan	Forever.\n	0

The next step was to create the word corpus. After compiling each line of dialogue into one list and removing the new line figure (\n) from the end of each line, I then

used the *re* and *string* packages in Python to lowercase all letters and remove punctuation, leaving us with the following format:

```
["chef said he's been bored so he joining a group called the super adventure club",
 'wow',
 'chef what kind of questions do you think adventuring around the world is gonna answer',
 "what's the meaning of life why are we here",
 "i hope you're making the right choice"]
```

With that done, I installed *NLTK* and a lesser known *contractions* package to expand all contractions and lemmatize verbs and nouns to get words in their base form. Here is the result from executing those steps:

```
['chef say he be be bore so he join a group call the super adventure club',
 'wow',
 'chef what kind of question do you think adventure around the world be go to answer',
 'what be the mean of life why be we here',
 'i hope you be make the right choice']
```

From the examples we can clearly see the drastic change from the original lines to the altered ones. The lines in their final form could barely pass as complete sentences, but that is okay. The point is to homogenize the words so that all variations of the same root word are treated similarly. Things like punctuation and letter case would falsely indicate the presence of more words in the word corpus than there actually are. The concept is similar for lemmatization and the other steps. We really just want all the unique root words rather than all the unique variations, as that is the best way to normalize the data.

So to recap and summarize, here are the steps taken to create the word corpus:

- Compile all strings, or lines of dialogue into one list
- Remove the new-line figure (\n) from each line
- Convert each word to lowercase and remove punctuation
- Expand all contractions to their whole words
- Lemmatize all nouns and verbs so that only root words are present

Stop words

With the corpus cleaned, I next examined words with the highest frequency to search for potential stop words. To do this, I used a Counter and an OrderedDict to order words by their total frequencies. I then created two functions to compare the words individually to confirm whether or not they should be treated as stop words. In essence, words that are not evenly distributed across all classes but tend to distinguish one class from the other have more significance and should not be

treated as stop words. This is similar to the concept utilized by tf-idf, which is implemented later. I did make a conscious decision to leave a few common words because of their significance. I chose to keep 'my' and 'me' because they had slightly imbalanced distributions, which might owe to Cartman's selfish nature. I also chose to keep 'no' and 'not' because those might be useful later when moving beyond a simple bag-of-words model. Here is the list of stop words:

```
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']
```

Word clouds

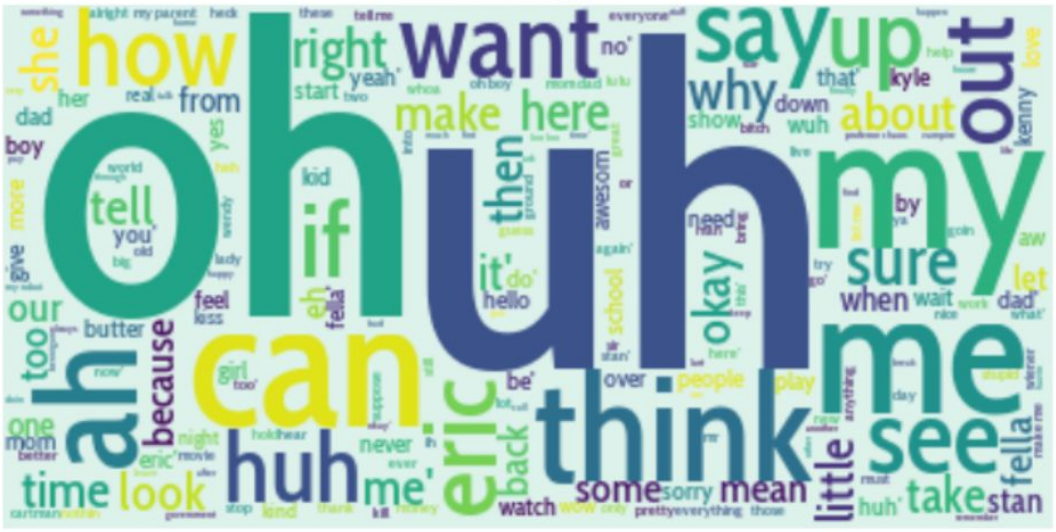
Beyond simple line counts, word clouds are a useful way to visualize word frequencies. This can be helpful to see how certain characters might favor certain words, illustrating differences in dialogue. I wrote a function to create character-specific word clouds along with a few examples shown below.

Cartman

Kyle

Stan

Butters



The word clouds exhibit some similarities and some differences. The similarity between the word clouds for Stan and Kyle is striking, while Cartman seems to really favor 'me' and 'my' comparative to the others.

Modeling

After prepping the word corpus, it's time to move on to modeling. Before a model can be trained though, the word corpus needs to be converted into mathematical vectors. To achieve this, I used CountVectorizer and TfidfVectorizer from Scikit

Learn. Once the vectorized dictionary was established, I split the data into test and training sets in order to train a model. With each of the different vector types I first started with a Multinomial Naive Bayes algorithm. From there, I also used the tf-idf vector to train other models using Random Forest, SVM, Logistic Regression and variants of Boosting. The accuracy levels did not vary much between the models, but the confusion matrices varied quite a bit.

Establishing the CountVectorizer

This is essentially a bag of words, where documents are assessed based on how many times each word appears. Higher usage of certain words will theoretically be more indicative of certain characters, in this case Cartman. Also of note is the *ngram_range* parameter, which accounts for unigrams, bigrams and so on. This allows for the option to consider not just single words, but groups of words as well. I tested for several ranges, but the range of (1,3) yielded the best results with CountVectorizer.

ngram_range	Number of features	Mean cv score
(1,1)	20,939	0.856
(1,2)	232,654	0.860
(1,3)	536,629	0.862
(1,5)	1,060,181	0.849
(2,2)	211,715	0.812
(3,3)	303,975	0.720

These results are from using MultinomialNB without any tuning, but we can see that (1,3) had a slight edge. We also see the large increase in features as the range increases beyond (1,1) because the model must consider individual words and combinations of words.

Also, by setting different minimum and maximum ranges, we can manipulate the CountVectorizer to examine common phrases and word combinations for Cartman. The idea is that this will allow insight into his speech patterns. Here are the results:

Single words: (1,1)	Two word phrases: (2,2)	Three word phrases: (3,3)
not	can not	oh my god
my	my got	can not believe
can	oh my	not tell me
me	not want	no no no
oh	not think	can not wait
your	south park	can not let
no	no not	let me see
yeah	why not	dude can not
here	tell me	no can not
right	no no	why can not

The entries in the table above aren't as distinct as I had hoped. All the negative words kind of get in the way but, on the other hand, it shows how central the negations are to the tone of the dialogue. The sentiment of these phrases would be much different if 'no' and 'not' were removed from the corpus. Also, the expanded contractions and other preprocessing techniques kind of dilute the phrases.

Tf-idf

To try and make further improvements, we can use a weighted tf-idf vectorizer instead of a basic CountVectorizer. Stop words aren't as big of a concern here due to the inherent adjustments, but since a list was already created, I decided to go ahead and use it. I again tried different n-grams for the new vectorizer, but (1,1) proved best in this scenario.

Somewhat surprisingly, using MultinomialNB to compare the results for CountVectorizer and TfidfVectorizer, the tf-idf model slightly under performed. While accuracy with CountVectorizer was 0.863, it was 0.862 with tf-idf. The f1-score, another performance metric, also decreased, as can be seen in the final results table. Tf-idf often outperforms a basic bag-of-words model, so these results were disappointing to see.

Trying other algorithms

Since MultinomialNB had its faults, I did try a few other algorithms in the hopes of making an improvement, namely Random Forest, SVM, Logistic Regression and variants of boosting. In terms of accuracy, all of the models tended to perform in the upper eighties range, similar to MultinomialNB. Tuning hyperparameters did help, but usually only marginally. However, despite the consistent accuracy scores, the models varied in terms of label prediction distributions, evidenced by referencing the confusion matrix and the f1-score for each one.

Naturally, the problem was largely due to the class imbalance. The models fared well with both precision and recall when predicting the majority class, or the one representing all characters other than Cartman. Of course, the scores were lower for the minority class. I tuned various parameters in an attempt to improve the results, but this usually only resulted in a tradeoff between precision and recall. I considered trying over-sampling the minority class or under-sampling the majority class, but I feared that wouldn't properly reflect a real world application. The class imbalance in the training data is likely similar to the imbalance found in an actual episode of South Park where the majority of the dialogue will not come from Cartman. Instead, I adjusted the weights for each class in an effort to have the models adjust and prioritize the minority class. This presented another tradeoff between accuracy and the f1-score. With added emphasis on the minority class, the accuracy suffered, but the other metrics for the minority class typically improved. In my opinion, because the minority class is the area of interest, it is more important to improve class recall at the cost of a few accuracy points. This, in part, is why I determined the model utilizing SVM to be the best performer. The accuracy decreased a bit after adjusting the weights, but it had the highest f1-score.

On another note, one other thing I did notice while validating these models is that they did tend to overfit a bit in terms of precision and recall. The accuracy score wouldn't noticeably change, but the distribution of predicted labels would often be very different between the training and test sets. Judging just by the training data, before adjusting class weights, Random Forest looked clear and away like the best choice. These discrepancies made it challenging to choose the best model, but that is ultimately the purpose of validation. This table illustrates the end results:

Algorithm	Accuracy	Target F1 Score	Runtime
MultinomialNB - with CountVectorizer	0.863	0.18	527 ms
MultinomialNB - with TfidfVectorizer	0.862	0.03	246 ms
Random Forest (balanced weights)	0.767	0.36	1 min 17 s
SVM (adjusted weights)	0.786	0.39	1.1 s
Logistic Regression	0.867	0.21	24 s
Gradient Boost	0.866	0.06	1 min 48 s
AdaBoost	0.761	0.36	2 min 30 s

I think it's clear to see that of all the algorithms tried, SVM performed the best. It's light and fast, and the results met or far exceeded the results from any of the other algorithms. I must say, I am a bit bummed that none of the models performed better. It would be nice to see accuracy in the 90% range, and see a little less conflict between the precision and recall. As it stands, it's kind of a case of pick your poison. Some are more accurate while some are better at predicting Cartman's lines. In my judgement, SVM strikes the best balance.

Final Thoughts

We managed to take the data set of dialogue, clean it up and preprocess it, and use it to create vocab vectors. The accuracy scores were pretty good, but the confusion matrices revealed some issues, likely due to the imbalance in data. Despite the results, on a macro level, I think it was beneficial to see the potential of these techniques and get a grasp of the NLP pipeline. As for why the models weren't successful, I think it likely has to do with the nature of the data, the way it is structured and the approach to the problem.

In regards to the nature of the data, we have to keep in mind that South Park is essentially created by only two people, Trey Parker and Matt Stone. So even though we have multiple characters, many of those characters likely have the same person behind them crafting the dialogue and delivering the voice-overs. This means that some of the boundaries of distinction between characters might get blurred, making classification difficult.

The other thing to consider is the structure of the documents. We are dealing with lines of dialogue, which can often be very short, many times only consisting of one or two words. If a character responds to a question with a simple "yes" or "no", that one word answer would still be considered as an entire document. Trying to classify such a document would be little more than guesswork. It's likely that this was a contributing factor to the low levels of recall.

To address the issue of document length we could drop all lines below a certain threshold. That might cut down on the total number of observations, but it might help ensure there is enough information within each vector. On the other hand, we might lose certain catch phrases that may be short but could still help identify a certain character. It seems like there are tradeoffs for any approach we might choose.

To try and improve the results, there are a few next steps we might attempt:

- Test thresholds for document length
- Omit some of the preprocessing steps -- maybe contractions are used differently by certain characters
- Use a more complex model involving deep learning
- Narrow the focus by omitting unimportant characters that might dilute the data

Then again, maybe the answer is to just use the Simpsons for our data set: a show with so many seasons that there's bound to be enough data to solve our problems.