Predicting MPAA Content Ratings from Script Content

By Tal Shprecher

Problem:

The Motion Picture Association of America (MPAA) assigns content ratings to movies released in the US. There are five ratings: G, PG, PG-13, R, and NC-17. The MPAA[1] defines each as follows:

Content Rating	Description
G	General Audiences
PG	Parental Guidance Suggested
PG-13	Parents Strongly Cautioned
R	Restricted
NC-17	Adults Only

Filmmakers target a movie for a specific rating, which has implications on box office income. As the ratings move from least to most restrictive, the target audience diminishes. If a movie intended for the PG-13 audience gets rated R, this will adversely affect the expected box office receipts.

By accurately predicting the MPAA rating of feature films from the content in the script, studios can serve their intended audience more predictably without having to appeal[2] the rating or--worse--re-edit the film, which costs time and money. This project aims to solve this problem by building a machine learning classifier that takes a complete movie script and predicts the content rating. The project code including the data, data collection scripts, python notebook, and this document can be found at https://github.com/tshprecher/mpaa_ml.

Data:

Fetching the set of movies with their ratings

Before training a model, data must be collected. First, a set of movies with their content ratings must be selected. Thankfully there exists a Kaggle dataset[3] with over 5000 movies. Of the numerous features needed, this dataset yields three: title, year, and content rating. The content rating is necessary for the target class and the title for script retrieval, but it may not be obvious why the year is necessary. The PG-13 rating was introduced in the 1980s, so I restricted movies to those released since 1990 to eliminate data skewed towards PG or R ratings.

The resulting `movies.txt`[4] file generated from the Kaggle dataset serves as input to the next stage of data collection: fetching entire script content.

Fetching Script Content

Fetching movie script content proved by far the most time consuming process of the project. The entire script must be download for each movie listed in `movies.txt`. This was solved by scraping scripts from the Internet Movie Script Database

(IMSDB.com[5]), which has over one thousand complete scripts. I wrote a scraper `imsdb.go`[6] that reads input from `movies.txt` and scrapes the entire script from IMSDB.com. Data is written to a folder containing the script and metadata for each film (title, year, and content rating). Unfortunately not all scripts existed, but this process downloaded the scripts for 481 films, enough to start training a classifier.

Building feature sets for each movie

Each movie script must be converted into a feature set for each movie. To accomplish this, another script `generate_features.go`[7] was written to take each movie script and convert it into a csv of normalized words and bigrams along with the number of time each feature occurs in the script.

Joining all movies and features into a single dataset

Once each movie's feature set is created, I needed to combine all the features of every movie into a single dataset. This is known as an outer join, but it could scale in Python using the builtin outer joins on DataFrames, so I wrote `join_features.go`[8] to perform the task, filling in zeroes where no occurrence of the feature existed in a film. In addition to computing the full outer join, it prunes the set of features. To avoid overfitting, all features that occur in less than 5% of films were removed. Furthermore, all features that occur in over 90% of films were assumed to contain little signal and were also removed. This greatly reduced the feature set from over 3 million to less than

45K to be further processed and pruned within the Python Notebook. The `movie_features.txt`[9] file serves as the data fed into the Python Notebook.

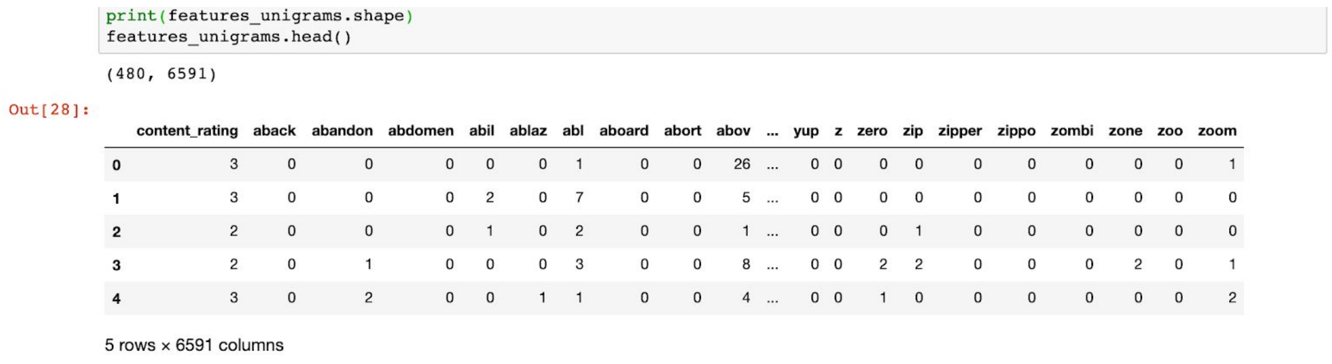
Further dataset pruning in Python

Once enough of the dataset was generated and pruned outside the Python Notebook for scalability reasons, a few more steps were taken in Python to select the final dataset.

1. Since the dataset only contained one NC-17 movie, it was removed from the dataset to avoid bias toward less permissive ratings.
2. Bigrams were eliminated from the feature set. They present an opportunity for more accurate classification, but they also pose a problem in injecting covariance and bias into the set of features since each bigram may be composed of two other existing features if the words that comprise the bigram have not been filtered out in a previous step.
3. The remaining features and counts were merged by the word's stem using the `EnglishStemmer` from the `nltk.stem.snowball` package. This helps combine similar ideas into similar columns.
4. Finally, to avoid overfitting, only a subset of the features were selected and the rest ignored. More specifically, the top quartile of features ranked by correlation to content rating were selected.

Data Summary

Before filtering on the most correlative features, the dataset looks like this:



```
print(features_unigrams.shape)
features_unigrams.head()
```

```
(480, 6591)
```

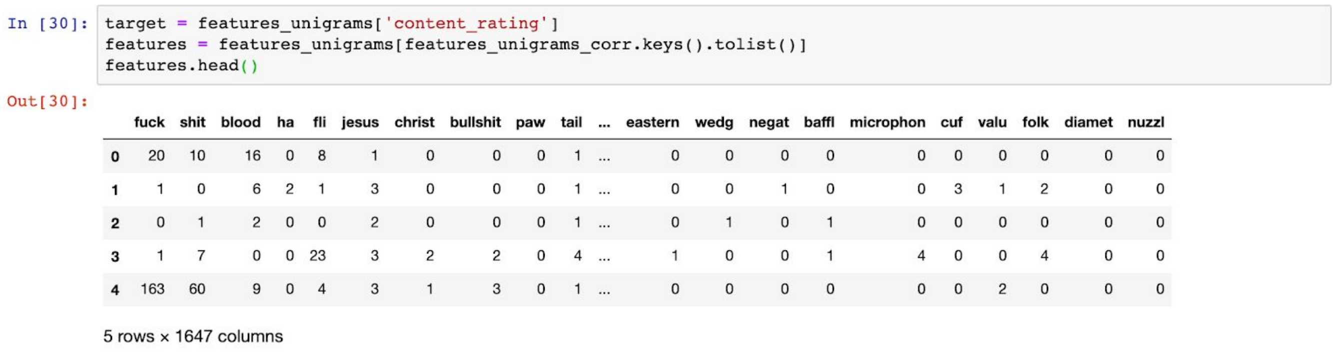
Out[28]:

	content_rating	aback	abandon	abdomen	abil	ablaz	abl	aboard	abort	abov	...	yup	z	zero	zip	zipper	zippo	zombi	zone	zoo	zoom
0	3	0	0	0	0	0	1	0	0	26	...	0	0	0	0	0	0	0	0	0	1
1	3	0	0	0	2	0	7	0	0	5	...	0	0	0	0	0	0	0	0	0	0
2	2	0	0	0	1	0	2	0	0	1	...	0	0	0	1	0	0	0	0	0	0
3	2	0	1	0	0	0	3	0	0	8	...	0	0	2	2	0	0	0	2	0	1
4	3	0	2	0	0	1	1	0	0	4	...	0	0	1	0	0	0	0	0	0	2

5 rows × 6591 columns

The first column represents the ordinal value of the content rating with G=0 and R=3. It serves as the target column for training. The remaining columns are the features and serve as the feature set for training.

After selecting the most correlative features, the dataset looks like this before training:



In [30]:
```
target = features_unigrams['content_rating']
features = features_unigrams[features_unigrams_corr.keys().tolist()]
features.head()
```

Out[30]:

	fuck	shit	blood	ha	fli	jesus	christ	bullshit	paw	tail	...	eastern	wedg	negat	baffl	microphon	cuf	valu	folk	diamet	nuzzl
0	20	10	16	0	8	1	0	0	0	1	...	0	0	0	0	0	0	0	0	0	0
1	1	0	6	2	1	3	0	0	0	1	...	0	0	1	0	0	3	1	2	0	0
2	0	1	2	0	0	2	0	0	0	1	...	0	1	0	1	0	0	0	0	0	0
3	1	7	0	0	23	3	2	2	0	4	...	1	0	0	1	4	0	0	4	0	0
4	163	60	9	0	4	3	1	3	0	1	...	0	0	0	0	0	0	2	0	0	0

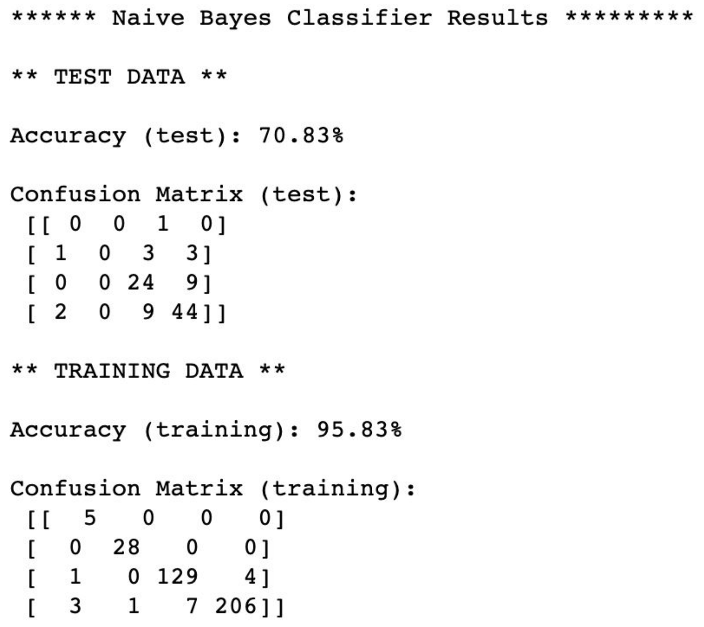
5 rows × 1647 columns

Note that explicit language serves as some of the most important features. This is expected.

Classifying

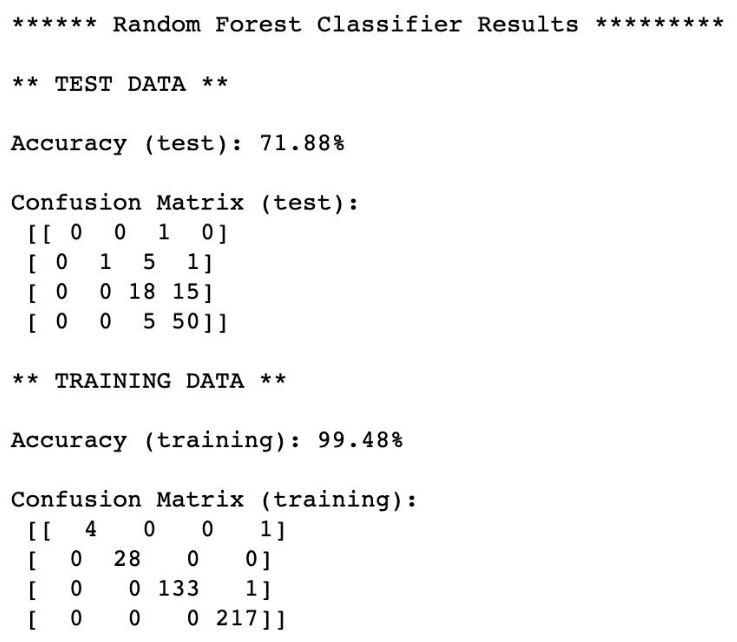
After the target and feature datasets have been established, the challenge becomes training and testing using multiple classification algorithms. The three algorithms selected were naive bayes, random forest, and one-vs-rest logistic regression. The dataset was split 80/20 into training and test data. All three classifiers were trained and tested on the same data.

Naive Bayes Results



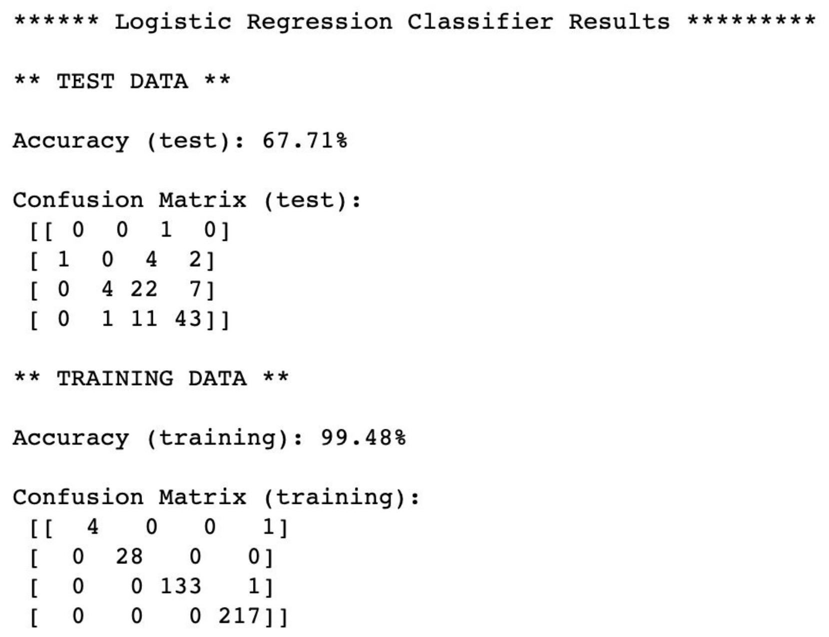
```
****** Naive Bayes Classifier Results *********

** TEST DATA **

Accuracy (test): 70.83%

Confusion Matrix (test):
 [[ 0  0  1  0]
 [ 1  0  3  3]
 [ 0  0 24  9]
 [ 2  0  9 44]]

** TRAINING DATA **

Accuracy (training): 95.83%

Confusion Matrix (training):
 [[  5   0   0   0]
 [  0  28   0   0]
 [  1   0 129   4]
 [  3   1   7 206]]
```

Random Forest Results



```
****** Random Forest Classifier Results *********

** TEST DATA **

Accuracy (test): 71.88%

Confusion Matrix (test):
 [[ 0  0  1  0]
 [ 0  1  5  1]
 [ 0  0 18 15]
 [ 0  0  5 50]]

** TRAINING DATA **

Accuracy (training): 99.48%

Confusion Matrix (training):
 [[  4   0   0   1]
 [  0  28   0   0]
 [  0   0 133   1]
 [  0   0   0 217]]
```

One-Vs-Rest Logistic Regression



```
****** Logistic Regression Classifier Results *********

** TEST DATA **

Accuracy (test): 67.71%

Confusion Matrix (test):
 [[ 0  0  1  0]
 [ 1  0  4  2]
 [ 0  4 22  7]
 [ 0  1 11 43]]

** TRAINING DATA **

Accuracy (training): 99.48%

Confusion Matrix (training):
 [[  4   0   0   1]
 [  0  28   0   0]
 [  0   0 133   1]
 [  0   0   0 217]]
```

Conclusion

The random forest classifier is the most accurate of the three with nearly 72% accuracy, but I consider the naive bayes classifier slightly better. The other two have accuracies of over 99% on the training data, which suggests to me they are more likely to be overfitting the dataset. At 71% accuracy, the naive bayes classifier is nearly as good as random forest and is probably the least overfitting and therefore more generalizable.

I suspect further tweaks to the feature set could see even better results. For example, there is probably a better way of filtering out the most important features. Sorting by correlation and choosing the top quartile seems sensible, but it yielded some unusual results. For example, words like *whale* and *cloud* became features. I suspect there could be some covariance issue between such seeming innocuous features and other truly rating-sensitive features. If there is an efficient and automated way to compute and filter out features based on covariance over thousands of features, it's something that should be explored. In addition, the models used here ignore bigrams, which could also improve the accuracy. Doing so would have the risk of adding bias since bigrams by definition have covariance with other features.

References:

1. https://en.wikipedia.org/wiki/Motion_Picture_Association_of_America_film_rating_system
2. https://deadline.com/2018/10/mpaa-50th-anniversary-ratings-system-appeals-more-rare-1202491191/
3. https://www.kaggle.com/carolzhangdc/imdb-5000-movie-dataset
4. https://github.com/tshprecher/mpaa_ml/blob/master/data/movies.txt

5. https://imsdb.com
6. https://github.com/tshprecher/mpaa_ml/blob/master/scripts/imsdb.go
7. https://github.com/tshprecher/mpaa_ml/blob/master/scripts/generate_features.go
8. https://github.com/tshprecher/mpaa_ml/blob/master/scripts/join_features.go