SPOOKY AUTHOR ID

Multi-class text classification with vectorizers and classifiers!

Timothy R Advani

SATURDAY, DECEMBER 16, 2017

SPRINGBOARD CAPSTONE 2 FINAL REPORT

INTRODUCTION

"It was a dark and stormy night" may be written by anyone. However, in the spirit of Halloween, the goal of this project is to apply data science concepts to help identify an author based on his or her prose. This review will review deeply into machine learning algorithms, hyperparameter tunings, and conclude about the evaluations and findings. This project will explore the use of natural language processing, which has a strong ability to infer the meaning behind the message.

STAKEHOLDERS

The business problem is to identify the author of a given unlabeled text, using models previously trained with labeled text. This project will explore several classification problems and study processing time and accuracy of various algorithms in order to show which are able to perform the task best. This will lure linguists, communication experts, and people interested in classifying unlabeled text with respect to a given set of possible authors. This will serve as a foundation for applying complicated methods, and see how they perform.

MATERIALS

All work was done on Jupyter Notebooks version 5.0.0, and the following programs:

1. Python 3.4
2. Pandas 0.20.3
3. Numpy 1.12.1
4. Sci-kit Learn 0.18.2
5. Matplotlib 2.0.2
6. Seaborn 0.7.1

DATA

Acquisition

The data source comes from Kaggle.com; in 2017, they hosted a $25K competition to create learning algorithms to classify given portions of text, according to a set of authors (see reference [1]). Data were downloadable as several separate CSV files. This will focus on the training set.

Dataset

The training dataset has three attributes for 19,579 observations or documents (19,579 x 3).

- id (six-character id, 'id4021', primary index, text)
- author (three-character abbreviation, three choices: 'EAP', 'HPL', 'MWS', text, target response)
    - EAP = Edgar Allen Poe
    - HPL = HP Lovecraft
    - MWS = Mary W. Shelley
- text (text corpus, see exploratory analysis for more metrics, target data)

Wrangling and Cleansing

Originally the data is clean. No missing values were present and the classes in the training set are properly identified. However, the text does contain words that could be deleted to improve computational efficiency and produce concise vector representations of textual information. Cleaning the text is essential. Also, a new variable, length, was computed from the text, to serve as part of exploratory data analysis. In modeling, stop words (or the lack thereof) played a role in parameter tuning.

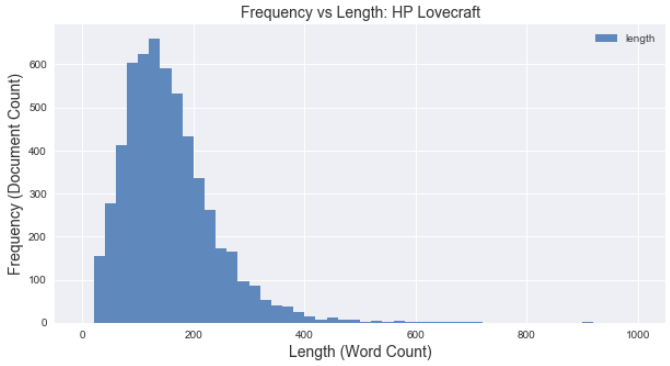
EXPLORATORY ANALYSIS

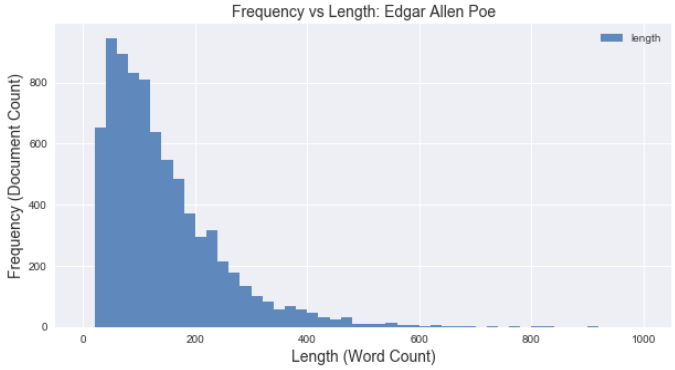
Before diving into the machine learning algorithm, what does the data look like? Below are count and length stats.

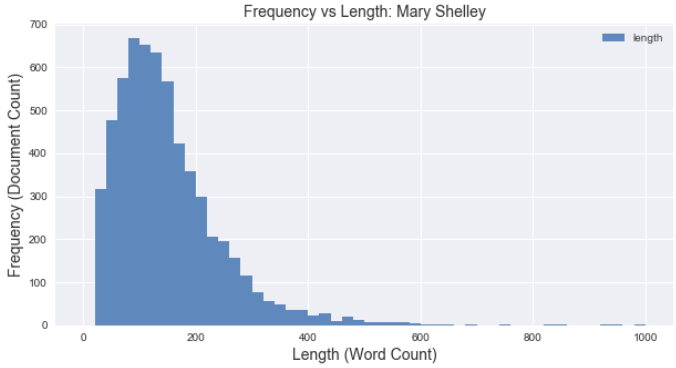
Author	TOTAL	EAP	HPL	MWS
count	19,579	7,900	5,635	6,044
mean	149	142	156	152
std	107	106	82	126
min	21	21	21	21
25%	81	68	98	84
50% / median	128	115	142	130
75%	191	186	196	192
max	4,663	1,533	900	4,663

Quickly, EAP has the most prose in the training set. However, the training set is uniformly distributed because length plays a trivial role; no prose will be smaller than 21 words. The average words per prose is 149, with MWS having 4,663 words in one prose. This provides an idea as to how the features (or grams) will grow.

For each author, below is a frequency representation by length (word count). Notice how all three distributions are skewed right; it is rare yet possible for an author to have over 500 words per prose.


Frequency vs Length: HP Lovecraft
length
Frequency (Document Count)
0
100
200
300
400
500
600
0
200
400
600
800
1000
Length (Word Count)


Frequency vs Length: Edgar Allen Poe
length
Frequency (Document Count)
0
200
400
600
800
0
200
400
600
800
1000
Length (Word Count)


Frequency vs Length: Mary Shelley
length
Frequency (Document Count)
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
Length (Word Count)

FEATURES

For text classification algorithms, there are two fundamental elements: authors and text. The authors attribute is the target of interest. The text attribute is the data that determines which author wrote this. In other words, provided a series of text, which author wrote the following prose by wording style? However, the real features are those from the document-terms matrix. For a training set, it will pass through a vectorizer and return a 'bag of words', or vocabulary unique to the set of texts passed.

The filters or transformers are vectorizers, which translate a text fragment into a vector of features. This project will use two vectorizers from the scikit-learn package:

- **CountVectorizer()**, or *Count Vectorizer*, which creates a document-terms matrix showing the count of words per document
- **TfidfVectorizer()**, or *TF-ID Vectorizer*, which is similar to CountVectorizer, but weights the word within a text relative to the entire corpus using the metric TD-IDF (term frequency - inverse document frequency).

These matrices will pass into the machine learning algorithm to predict each author.

MODEL

Strategy

The strategy focuses on using a combination of classifiers and vectorizers to perform the job. The aim is to evaluate which combinations perform better in this project, and what features were important. The following are guiding questions:

(1) Which **method** of vectorization is better?
(2) What **algorithms** to choose?
(3) How to tune **parameters** to use with these algorithms?
(4) What **metrics** to use in order to evaluate the performance?
(5) How features guide the decision to classify a given portion of text with respect to the target set of authors?

To deal with vectorization, two methods exist. The first method is to split the dataset into training and testing sets, and then vectorize the training set; the training set will have 70% of the entire corpus and will provide a real-world scenario. The second is to vectorize the entire corpus and then split the dataset into training and testing sets; the vectorization will have a larger bag of words and improve context to identify the author who wrote a certain piece of text. These two methods would determine which combinations captures the corpus better and makes stronger predictions.

Either way, each method will follow the same steps:

- Tune the parameters associated with the algorithms that will be used
- Fit the models with best parameters on the training set
- Find predictions for the training and testing sets
- Compute the performance evaluation metric for training and test sets
- Identify the guiding features per author (at best)

In this project, the following elements are combined:

- vectorizer: `CountVectorizer()`, `TfidfVectorizer()`
- classifiers: `MultinomialNB()`, `LogisticRegression()`, `RandomForestClassifier()`

This will create 24 combinations, based on method, vectorizer, and classifier. The first step is to explore the major parameters and determine a proper evaluation field, a set of lists, possibilities, or factors that would make it easy to evaluate the performance in GridSearchCV. This will lead to the optimal parameters to use for the model. Then, this leads to the performance metrics. A classification report provides a comprehensive performance, as does the Out-of-Bag (OOB) score for the Random Forest algorithm. These steps will show which algorithms performed better.

Finally, each algorithm will explore feature importance by showing the top 10 words and their best associations. Please keep in mind that feature importance may differ. Naive Bayes uses probabilities, Logistic Regression uses coefficients, and Random Forest (due to time) focused on a combined feature importance parameter irrespective of class. Nevertheless, it will show which features were important. While accuracy may show which algorithms did well, feature importance will show what words determined which author.

Parameters

Parameters were tuned using GridSearchCV, with one vectorizer and one classifier. Both vectorizers have the same parameter tuning; hence, they were merged. The parameters were:

SCI-KIT LEARN	**Parameters***
CountVectorizer()	**max_df = [0,1,2,3,4,5]** When building vocabulary ignore terms that have a document frequency strictly lower than the given threshold. **ngram_range = [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]** (The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.)
TfidfVectorizer()	**max_df = [0,1,2,3,4,5]** When building vocabulary ignore terms that have a document frequency strictly lower than the given threshold. **ngram_range = [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]** (The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.)
MultinomialNB()	**alpha=[0.05, 0.1, 1.0, 2.0]** additive smoothing parameter
LogisticRegression()	**C=[0.05, 0.1, 1.0, 2.0]** Inverse of regularization strength, small C larger regulation; smoothing parameter

RandomForestClassifier()	**max_features = ['auto', 'sqrt', 'log2']** The number of features to consider when looking for the best split **n_estimators = range(0,200)** The minimum number of samples required to be at a leaf node **min_samples_leaf = range(0,200)** The minimum number of samples required to be at a leaf node **random_state = 42** Keeps the randomization to one seed, steady. **oob_score = True** Whether to use out-of-bag samples to estimate the generalization accuracy. **n_jobs = 1** The number of jobs to run in parallel for both fit and predict

* Descriptions come from sci-kit learn documentation.

For Count, `stop_words` and `ngram_range` tuned to better performance and got better accuracy. The classifiers have smoothing parameters that may get better accuracy. But optimizing multiple parameters increased processing time. Therefore, the parameters 'binary' and 'stop_words' were best defaults when cross-validated and ignored in tuning.

MultinomialNB()

Hyperparameter Optimization

Below are the best parameters per combination, showing wall time (total processing time) and best score (from 0 to 1).

Setup	Wall time	Best score	Vectorizer	Classifier
CountVectorizer() Method 1	16min 15s	0.844	vect__ngram_range: (1, 2) vect__max_df: 0.5	nb__alpha: 0.1
CountVectorizer() Method 2	6min 25s	0.859	vect__ngram_range: (1, 2) vect__max_df: 0.5	nb__alpha: 0.1
TfidfVectorizer() Method 1	2min 48s	0.845	tfidf__ngram_range: (1, 2) tfidf__max_df: 0.5	nb__alpha: 0.05
TfidfVectorizer() Method 2	3min 25s	0.854	tfidf__ngram_range: (1, 2) tfidf__max_df: 1.0	nb__alpha: 0.1

Both the Count Vectorizer and the TF-IDF Vectorizer had better `ngram_ranges` between 1 and 2. For `MultinomialNB()`, the alpha parameter is smaller (i.e. 0.05-0.1) for smoothing.

Classification Reports

Method 1 with CountVectorizer() / Train

	Precision	Recall	F1 score	Support
EAP	1.0	1.0	1.0	5,923
HPL	1.0	1.0	1.0	4,218
MWS	1.0	1.0	1.0	4,543
Avg/Tot	1.0	1.0	1.0	14,684

Method 1 with CountVectorizer() / Test

	Precision	Recall	F1 score	Support
EAP	0.84	0.87	0.86	1,916
HPL	0.85	0.87	0.86	1,290
MWS	0.88	0.83	0.86	1,470
Avg/Tot	0.86	0.86	0.86	4,895

Method 1 with TfidfVectorizer() / Train

	Precision	Recall	F1 score	Support
EAP	1.0	1.0	1.0	5,936

Method 1 with TfidfVectorizer() / Train

	Precision	Recall	F1 score	Support
EAP	0.85	0.82	0.83	2,054

HPL	1.0	1.0	1.0	4,222
MWS	1.0	1.0	1.0	4,526
Avg/Tot	1.0	1.0	1.0	14,684

HPL	0.81	0.86	0.86	1,321
MWS	0.84	0.84	0.84	1,520
Avg/Tot	0.82	0.81	0.81	4,895

Method 2 with CountVectorizer() / Train				
	Precision	**Recall**	**F1 score**	**Support**
EAP	0.99	0.98	0.99	5,925
HPL	0.99	0.99	0.99	4,226
MWS	0.98	0.99	0.99	4,533
Avg/Tot	0.99	0.99	0.99	14,684

Method 2 with CountVectorizer() / Train				
	Precision	**Recall**	**F1 score**	**Support**
EAP	0.99	0.98	0.99	1,975
HPL	0.99	0.99	0.99	1,409
MWS	0.98	0.99	0.99	1,511
Avg/Tot	0.99	0.99	0.99	4,895

Method 2 with TfidfVectorizer() / Train				
	Precision	**Recall**	**F1 score**	**Support**
EAP	1.0	1.0	1.0	5,937
HPL	1.0	1.0	1.0	4,221
MWS	1.0	1.0	1.0	4,526
Avg/Tot	1.0	1.0	1.0	14,684

Method 2 with TfidfVectorizer() / Train				
	Precision	**Recall**	**F1 score**	**Support**
EAP	1.0	1.0	1.0	2,039
HPL	1.0	1.0	1.0	1,300
MWS	1.0	1.0	1.0	1,556
Avg/Tot	1.0	1.0	1.0	4,895

Method 2 with `TfidfVectorizer()` was superior than that with `CountVectorizer()`. However, Method 2 was vectorizing on the entire corpus (training and testing set) so it is sensible. Method 1 with `CountVectorizer()` did better than that with `TfidfVectorizer()` by 0.05. While this is slightly significant, the difference is still minimal to choose either vectorizer. From metrics, the best choice is to apply Method 2 with `TfidfVectorizer()`.

Feature Importance:

	EAP	HPL	MWS
Method 1 CountVectorizer() MultinomialNB()	upon 0.116962 one 0.071730 now 0.060422 said 0.044895 will 0.044726 made 0.033418 time 0.033418 well 0.032911 even 0.032574 say 0.032574	one 0.078325 old 0.068386 now 0.052532 man 0.049219 seemed 0.049219 like 0.047089 night 0.043540 time 0.041884 though 0.041410 saw 0.039754	one 0.069270 now 0.058460 will 0.055813 life 0.050739 yet 0.049195 raymond 0.044341 love 0.042797 heart 0.041915 might 0.041253 even 0.040150
Method 1 TfidfVectorizer() MultinomialNB()	upon 0.009923 one 0.006662 now 0.005922 said 0.005776 will 0.005269 say 0.004377 however 0.004355 little 0.003919 well 0.003866 made 0.003659	one 0.007390 old 0.006728 man 0.005220 now 0.005176 night 0.005148 seemed 0.005108 though 0.004830 like 0.004737 saw 0.004720 came 0.004646	will 0.006415 raymond 0.006166 one 0.005755 life 0.005631 now 0.005615 love 0.005389 yet 0.005303 heart 0.004890 perdita 0.004564 us 0.004437
Method 2 CountVectorizer() MultinomialNB()	upon 0.173165 one 0.110717 now 0.080000 will 0.066667 said 0.060253 little 0.046582 say 0.045232 well 0.044895 made 0.044557 even 0.044219	one 0.116422 old 0.092996 now 0.070989 man 0.066493 like 0.064837 seemed 0.064600 night 0.060577 things 0.056555 time 0.056555 saw 0.055845	one 0.105449 will 0.100375 now 0.088021 life 0.073902 yet 0.070373 love 0.060666 us 0.060225 raymond 0.059784 might 0.059563 heart 0.058681
Method 2 TfidfVectorizer() MultinomialNB()	upon 0.012826 one 0.008459 said 0.007855 now 0.007486 will 0.006706 however 0.005647 say 0.005556 well 0.005095 little 0.004956 thus 0.004812	one 0.008994 old 0.008842 seemed 0.006706 night 0.006455 now 0.006442 man 0.006441 though 0.006290 like 0.006266 saw 0.006116 came 0.006010	will 0.008565 raymond 0.008032 now 0.007758 one 0.007662 life 0.007306 yet 0.007261 love 0.006900 heart 0.006448 perdita 0.005989 father 0.005647

Here, both vectorizers are returning similar words. Any writer may associate with these words (i.e. will, one, made). However, Mary Shelley's work has better insight to words, such as 'raymond', 'heart', and 'perdita.' The TF-IDF Vectorizer, despite better context-awareness, makes little progress than Count Vectorizer. So let's explore the other two classifiers before deciding one.

LogisticRegression()

Hyperparameter Optimization

The best parameters and wall time for evaluating different pairs is below.

Setup	Wall time	Best score	Vectorizer	Classifier
CountVectorizer() Method 1	22min 20s	0.812	vect__ngram_range: (1, 1) vect__max_df: 0.75	logreg__C: 1.0
CountVectorizer() Method 2	7min 56s	0.814	vect__ngram_range: (1, 1) vect__max_df: 0.75	logreg__C: 2.0
TfidfVectorizer() Method 1	2min 56s	0.806	tfidf__ngram_range: (1, 1) tfidf__max_df: 0.75	logreg__C: 2.0
TfidfVectorizer() Method 2	3min 39s	0.821	tfidf__ngram_range: (1, 1) tfidf__max_df: 0.75	logreg__C: 2.0

Unlike Naive Bayes comfortable with n-grams of (1,2), the n-grams are (1,1) with a maximum degrees of freedom as 0.75. All perform at least 80% accuracy.

Classification Report:

Method 1 with CountVectorizer() / Train				
	Precision	Recall	F1 score	Support
EAP	0.98	0.95	0.97	6,145
HPL	0.96	0.99	0.97	4,124
MWS	0.95	0.98	0.96	4,415
Avg/Tot	0.97	0.97	0.97	14,684

Method 1 with CountVectorizer() / Test				
	Precision	Recall	F1 score	Support
EAP	0.87	0.77	0.82	2,218
HPL	0.77	0.85	0.81	1,265
MWS	0.78	0.83	0.81	1,412
Avg/Tot	0.82	0.81	0.81	4,895

Method 1 with TfidfVectorizer() / Train				
	Precision	Recall	F1 score	Support
EAP	1.0	1.0	1.0	5,923
HPL	1.0	1.0	1.0	4,218
MWS	1.0	1.0	1.0	4,543
Avg/Tot	1.0	1.0	1.0	14,684

Method 1 with TfidfVectorizer() / Test				
	Precision	Recall	F1 score	Support
EAP	0.84	0.87	0.86	1,916
HPL	0.85	0.87	0.86	1,290
MWS	0.88	0.83	0.86	1,470
Avg/Tot	0.86	0.86	0.86	4,895

Method 2 with CountVectorizer() / Train				
	Precision	Recall	F1 score	Support
EAP	0.95	0.98	0.97	5,925
HPL	0.98	0.96	0.97	4,226
MWS	0.97	0.95	0.96	4,533
Avg/Tot	0.97	0.97	0.97	14,684

Method 2 with CountVectorizer() / Test				
	Precision	Recall	F1 score	Support
EAP	0.96	0.98	0.97	1,975
HPL	0.98	0.97	0.97	1,409
MWS	0.98	0.96	0.97	1,511
Avg/Tot	0.97	0.97	0.97	4,895

Method 2 with TfidfVectorizer() / Train				
	Precision	Recall	F1 score	Support
EAP	0.99	0.98	0.99	5,925
HPL	0.99	0.99	0.99	4,226
MWS	0.98	0.99	0.99	4,533
Avg/Tot	0.99	0.99	0.99	14,684

Method 2 with TfidfVectorizer() / Test				
	Precision	Recall	F1 score	Support
EAP	0.99	0.98	0.99	1,975
HPL	0.99	0.99	0.99	1,409
MWS	0.98	0.99	0.99	1,511
Avg/Tot	0.99	0.99	0.99	4,895

Here, the metrics support the second experiment, as it vectorizes the entire set. Combining `TfidfVectorizer()` and `LogisticRegression()` is a better choice. Also, overfitting is apparent yet minimal.

Feature Importance:

	EAP	HPL	MWS
Method 1 CountVectorizer() LogisticRegression()	dupin 2.307535 minutes 1.970809 evident 1.886097 color 1.672222 madame 1.578178 ellison 1.570647 lustre 1.568421 pompey 1.557530 drawer 1.533857 balloon 1.523219	gilman 2.284341 west 2.239494 though 2.140185 innsmouth 2.134974 jermyn 2.120193 despite 2.022103 pickman 1.993617 johansen 1.933906 later 1.910848 musides 1.896967	raymond 3.539530 perdita 3.143836 adrian 2.907555 idris 2.426964 sister 2.394821 towards 2.386876 elizabeth 2.192059 windsor 2.189323 miserable 2.131024 plague 2.039970
Method 1 TfidfVectorizer() LogisticRegression()	upon 6.562724 madame 3.764638 however 3.619355 dupin 3.466446 matter 3.426787 lady 3.340734 character 3.201394 altogether 3.184059 mr 3.156012 although 3.146067	though 6.083431 west 4.823636 street 4.567506 later 4.501265 gilman 4.205996 despite 4.125355 old 3.986580 innsmouth 3.831145 men 3.678563 whilst 3.507929	raymond 8.191651 perdita 6.080660 adrian 5.910498 towards 5.586380 love 4.787638 idris 4.634267 plague 4.205655 misery 4.176011 sister 4.062894 cottage 3.972599
Method 2 CountVectorizer() LogisticRegression()	dupin 2.226379 minutes 1.940119 madame 1.841242 precisely 1.788356 lady 1.785235 kate 1.737145 evident 1.730778 jupiter 1.706281 nose 1.695168 amid 1.657188	despite 2.403688 later 2.396820 jermyn 2.327398 gilman 2.320025 birch 2.223038 innsmouth 2.196929 though 2.180848 brown 2.089596 normal 2.079948 johansen 2.077081	raymond 3.711152 perdita 3.289060 adrian 3.082172 idris 2.583584 elizabeth 2.401328 plague 2.367171 windsor 2.158187 cottage 2.152484 towards 2.127779 miserable 2.100854
Method 2 TfidfVectorizer() LogisticRegression()	upon 6.746061 is 3.889144 however 3.838754 madame 3.821593 lady 3.658483 dupin 3.278529 minutes 3.242017 although 3.234435 marie 3.210714 character 3.124110	though 6.164898 west 4.582290 later 4.435798 street 4.219803 despite 4.102376 old 4.017074 uncle 3.968330 gilman 3.936558 men 3.906260 innsmouth 3.705354	raymond 8.194954 adrian 5.920245 perdita 5.732935 her 5.660064 towards 5.109025 love 4.825883 my 4.687107 idris 4.593774 she 4.478862 plague 4.407684

Logistic Regression provides 'richer' words to distinct the authors. Coefficients provide better context. When the coefficient is positive and larger, the more it 'nudges' the decision towards a positive case. The converse is true; when negative, it 'nudges' the decision towards a negative case. It is sensible to concentrate on larger positive coefficients to see how the model drives the decision to one of the three authors. So far, Method 2 using the TF-IDF Vectorizer and Logistic Regression is a better choice.

RandomForestClassifier()

Strategy

The Random Forest Algorithm is an ensemble classifier that estimates by different decision trees, computational expensive yet robust. However, they are one of the best supervised classification algorithms to implement, especially for natural language processing. Let's see how the Random Forest Algorithm performs against the previous two classifiers.

Hyperparameter Optimization

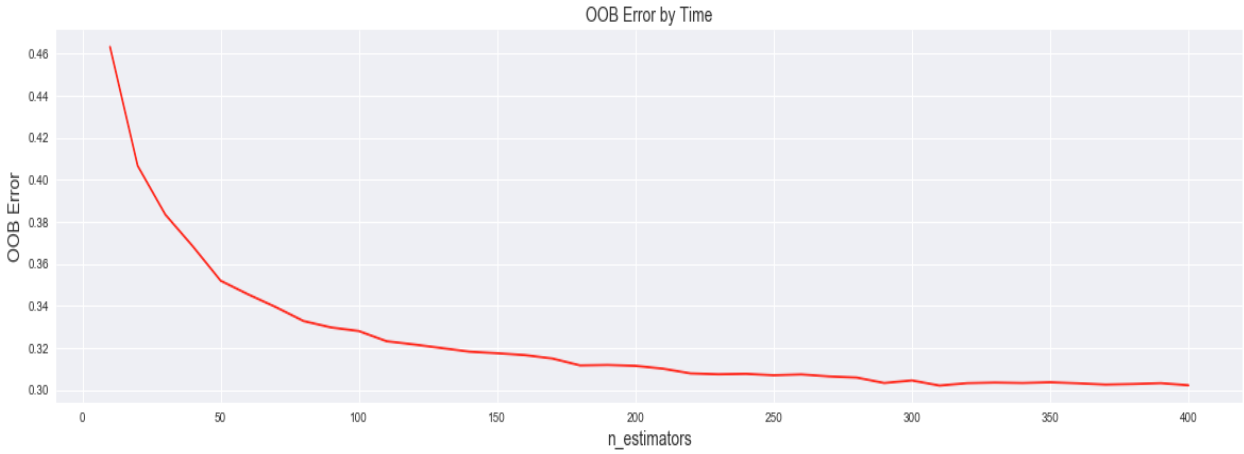
Since Random Forest has many parameters that could influence the model, two approaches were needed. One was experimental, and the other follows suit to the previous strategy (two methods, Count or TF-IDF Vectorizer).

Below shows the executions for the experimental approach. Here, processing time is longer.

	Wall time	Best score	Vectorizer	Classifier
As-is trial w/ Random Forest	5.02 s	0.619	*None*	*None*
Minor trial w/ Random Forest	54.1 s	0.700	*None*	n_estimators = 100, oob_score = True, warm_start = True
Actual trial w/ Random Forest	53.8 s	0.720	max_df=0.5, ngram_range=(1,1)	n_estimators=100, oob_score=True, warm_start=True, **max_features='log2'**

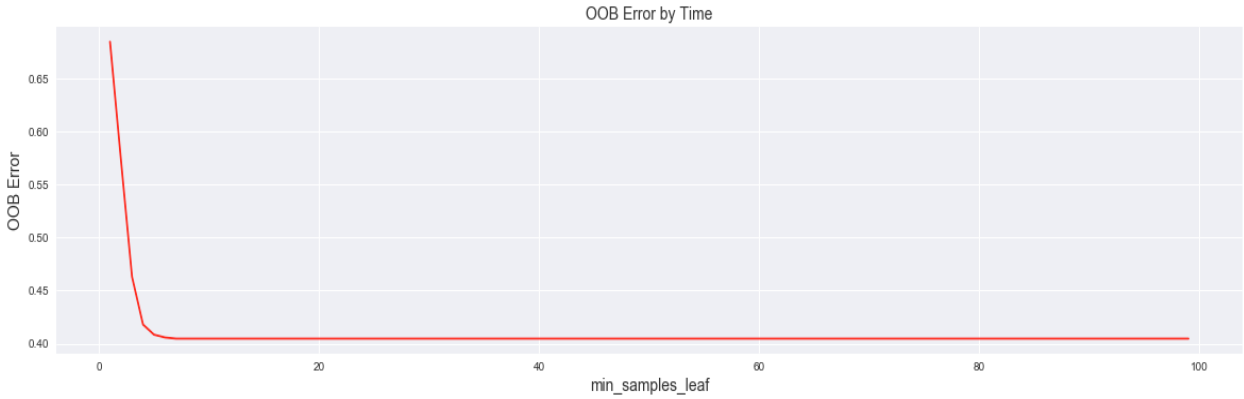
Before exploring different parameters, Random Forest is a tree-based method and is a bagging (bootstrap-aggregation) model. This type of model repeatedly fits to bootstrapped samples of the observation so using the test metrics is not going to work. Fortunately there is an easier way to deal with this; it is called the *out-of-bag error* estimation (or OOB error). It is a valid estimate of the test error and one used over classification reports.

The Random Forest Algorithm has over several parameters worth exploring. First, n_estimators is the number of trees in a random forest. The larger the parameter, the larger the forest. Focusing on n_estimators alone, the OOB error will stabilize around 0.30. Below shows the OOB Error as the number of trees grow from 10 to 410.



How about maximum features? Seeing three different choices ('sqrt', 'log2', 'none') would create disparities. The 'none' random forest has the same behavior as shown in the graph above. But 'log2' and 'sqrt' have done a better job to reduce the OOB error across the estimators.

Finally, the parameter 'min_samples_leaf', or the minimal number of leaves on a node, was important. A rule of thumb is to use around 50; this would reduce computation. However, as the leaves got populated, the OOB error grew in size. Hence, it was better to keep it as 1.



Classification Report

While overfitting is still present in the dataset, the hyperparameter tuning makes it better for random forest to classify certain prose between the authors. By enabling OOB scores and warm_starts (allows to add trees for each iteration, or starts a new tree), it has improved the accuracy and precision of the random forest to classify the prose with the respective author.

Method 1 with CountVectorizer()
OOB Score: 0.5835

	Precision	Recall	F1 score	Support
EAP	0.61	0.84	0.71	1,975
HPL	0.70	0.55	0.62	1,409
MWS	0.76	0.54	0.63	1,511
Avg/Tot	0.68	0.67	0.66	4,895

Method 2 with CountVectorizer()
OOB Score: 0.713

	Precision	Recall	F1 score	Support
EAP	1.00	1.00	1.00	1,975
HPL	1.00	1.00	1.00	1,409
MWS	1.00	1.00	1.00	1,511
Avg/Tot	1.00	1.00	1.00	4,895

Method 1 with TfidfVectorizer()
OOB Score: 0.6804

	Precision	Recall	F1 score	Support
EAP	0.58	0.95	0.72	1,975
HPL	0.89	0.46	0.61	1,409
MWS	0.88	0.56	0.68	1,511
Avg/Tot	0.76	0.69	0.68	4,895

Method 2 with TfidfVectorizer()
OOB Score: 0.7181

	Precision	Recall	F1 score	Support
EAP	0.64	0.90	0.75	1,975
HPL	0.83	0.59	0.69	1,409
MWS	0.82	0.62	0.70	1,511
Avg/Tot	0.75	0.72	0.72	4,895

Feature Importance

	Experiment 1 CountVectorizer()	**Experiment 1** TfidfVectorizer()	**Experiment 2** CountVectorizer()	**Experiment 2** TfidfVectorizer()
Feature Importance	raymond 0.01536 father 0.01309 windsor 0.01227 though 0.01024 thus 0.00934 upon 0.00802 adrian 0.00795 idris 0.00692 thing 0.00691 sister 0.00661	raymond 0.017322 upon 0.015225 perdita 0.011584 love 0.010100 though 0.009888 adrian 0.008734 father 0.007541 heart 0.007430 old 0.007167 west 0.006363	upon 0.001877 raymond 0.001590 love 0.001330 perdita 0.001265 adrian 0.001253 father 0.000988 though 0.000964 old 0.000926 heart 0.000900 life 0.000838	upon 0.006144 raymond 0.005534 perdita 0.003854 though 0.003818 love 0.003631 adrian 0.003392 old 0.003087 father 0.002661 towards 0.002489 life 0.002411

Unlike the first two algorithms, the feature importance for Random Forest does not break down by class. Any author may have written a particular text. While stop words were included in both vectorizers, the feature 'upon' is not a 'stop word' and appears more often. Same for 'will.' Like Logistic Regression, Random Forest provides significant words that impact the identification.

FINDINGS

Recommendations

The choices are infinite to tackle author identification (see Future Work). Three observations come from these analyses. First, using the entire corpus has its benefits and consequences. While it provided better accuracy on the test set, it leads to data leakage and tunes the performance towards this data. It is not used to relearn itself in face of new and unknown data.

Second, suitable and complex algorithms did beat the simpler ones. Logistic Regression was better and preferable vectorizers in not only weighing features by coefficients but did show significant features. With Naive Bayes, it was looking at each word and word pair to provide a better probability. It didn't suit well. Also, the TF-IDF Vectorizer makes more sense than Count Vectorizer. Whereas the latter focused on the frequency of a word, the former extended it further and tried to show the relationship between the word in its text and in its corpus. It wins!

Finally, stay away from Random Forest. Not only did it take the longest to compute, but it didn't provide significant results to make a valid decision on for this set. Further research may be needed to show Random Forest as a preferred method in this dataset. However, the feature importances for all classes bring significant features, though the features by class was not available. Also, the testing accuracy with computing time makes the Random Forest algorithm slightly infeasible.

Future Work

Several ideas stem from this analysis, as they may provide keys and clues to the author or create a reinforcement learning algorithm that may identify future authors and other genres.

Sentiment Analysis: Very popular with challenges leveraging existing text, this may explore how spookier the authors were. All three authors write horror, but which one is the scariest? Sentiment analysis could decipher some syntax and perform a deeper analysis into this.

Non-Negative Matrix Factorization (NMF): Non-negative matrix factorization helps explore the latency or underlying interactions between two entities. Popular for recommendation systems, it is excellent for text mining. This would help explore topic selection and create text cases.

Boosting Algorithm: With Random Forest is a boosting algorithm, it would delve deeper into other boosting algorithms such as XGBoost and AdaBoost. These would compare between the algorithms and provide a clear picture as to what boosting does with the text.

Other Algorithms: In this set, MultinomialNB() was used. But there were other Naive Bayes algorithms worth trying, including GaussianNB() and BernoulliNB(). Also, Support Vector

Machines and Stochastic Gradient Descent would be excellent algorithms worth playing around in this dataset.

Other Vectorizers: CountVectorizer and TF-IDF Vectorizer was implemented here, but there are other ways to vectorize the text corpus. DictVectorizer is also part of the same family, and Word2Vec would also be explored in managing words as vectors.

Built-in NLP Kits: Since NLP has been in the market for a long time, external libraries are available. The best five NLP libraries are NLTK, SpaCy, Gensim, CoreNLP, and TextBlog. These would have been great to explore with this dataset.

Advanced Trials: Finally, other modes of work include reinforcement learning, where if feeding new text data would make the model learn and adapt. This way it would resemble a growing AI-like algorithm. Also, implementing on another corpus would have made it wonder if any business-based authors are horror authors as well!

VERSIONS

0.0 = Final report, version 0: includes BASIC machine learning analysis

0.1 (12/20/2017) = Final report, version 0.1: Random Forest

0.2 (12/22/2017) = Final report, version 0.6: Word Vectorizer

0.3 (12/23/2017) = TF-IDF

0.4 (12/14/2017) = Most Predictive Features

1.0 (12/25/2017) = Final Report

1.1 (01/27/2017) = Updated report with syntactical errors

REFERENCES

1. "Spooky Author Identification: Share code and discuss insights to identify horror authors from their writings." Kaggle.com. Published 10/25/2017. Retrieved 12/07/2017. URL: https://www.kaggle.com/c/spooky-author-identification.

2. R. Tatman. "Beginner's Tutorial: Python". Kaggle.com. Published 10/25/2017. Retrieved 12/07/2017. URL: http://bit.ly/2CJnK4b.

3. S. Dane. "Intermediate Tutorial: Python". Kaggle.com. Published 10/25/2017. Retrieved 12/07/2017. URL: http://bit.ly/2yKUzKx.

4. K. Markham. "Machine Learning with Text in sci-kit learn". PyCon Portland 2016. Published 06/08/2016. Retrieved 12/08/2017. URL: http://bit.ly/2yPaYB4.

5. T. Srivastava. "Tuning Random Forests Model". Analytics Vidhya. Published 06/09/2015. Retrieved 12/20/2017. URL: bit.ly/2BaBiV4.

6. K. Ho, G. Louppe, A. Mueller. "OOB Errors for Random Forests". Sci-Kit Learn. Retrieved 12/19/2017. URL: http://bit.ly/2COv4dT.

7. A. Collier. "Making Sense of Logarithmic Loss". Exegenic.biz. Published 12/14/2017. Retrieved 12/20/2017. URL: http://bit.ly/2z7jSXK.

8. C. Perone. "Machine Learning :: Text feature extraction (tf-idf) – Part II". Published 03/10/2011. Retrieved 12/23/2017. URL: http://bit.ly/2C6BmJ0.

9. S. Sauer. "Convert logic to probability." Sebastian Sauer Stats Blog. Published 01/24/2017. Retrieved 12/24/2017. URL: http://bit.ly/2pse1wx.

10. T. Burhmann. Analyzing tf-idf results in scikit-learn. Blog. Published 06/22/2015. Retrieved 12/24/2017.URL: https://buhrmann.github.io/tfidf-analysis.html.

11. A. A. Yeung. "Matrix Factorization: A Simple Tutorial and Implementation in Python". Quuxlabs. Published 09/16/2010. Retrieved 12/24/2017. URL: http://bit.ly/1mEeFxH.

12. A. Saabas. "Selection Good Features, Part 3: Random Forests". Diving Into Data. Published 12/01/2014. Retrieved 12/25/2017. URL: http://bit.ly/2lauqAj.

13. G. James, D. Witten, T. Hastie, R. Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer, New York. 2013.