Spooky Author Identification - Text Classification in NLP

Capstone Project
Machine Learning Engineer Nanodegree

Shashank B
Jan, 2018

I. Definition

I.A. Project Overview

Natural Language Processing (NLP) is one of the fields in Artificial Intelligence that involves fruitfully processing natural language data. This project focuses on a sub-field in NLP called document / text classification, which concerns with classifying topic or category of a given article.

The problem statement is posted as one of the recent (2017) competitions from Kaggle, called “Spooky Author Identification”. The dataset includes labeled excerpts from horror stories of 3 authors, and the machine learning model needs to predict the author of the excerpts on unlabeled test dataset.

This project explores various techniques in NLP and machine learning in order build a machine learning model that can solve this problem. These include text preprocessing, various supervised machine learning models, Neural Network models and Stacking.

I.B. Problem Statement

The training dataset (train.csv) includes labeled excerpts from horror stories of 3 authors, i.e. each data point includes its author. The test dataset (test.csv) is not labeled, and the machine learning model built has to predict the author (among the 3 authors) of each of data points in test dataset.

For each sample in the dataset, the prediction of label from the machine learning model should be in the form of probability for each author. So each sample’s output will be 3 columns with probabilities of the respective 3 authors.

In this project, various NLP and machine learning techniques have been used:

1. Feature engineering:
    i. Using the given text to create certain features that can be input into standard sklearn machine learning models.
2. Sklearn non-neural network classifiers:
    i. Convert the text data to numerical format that the training models accept. (This includes generating document vs term frequency matrix)
    ii. Train various classifiers and perform hyperparameter tuning.
3. Keras neural network classifiers:
    i. Convert text data to numerical format on which neural network models work well. (This includes generating word vectors or using pre-trained word vectors.)
    ii. Train neural networks and perform hyperparameter tuning.

4. Stacking:
    i. Use the predictions made from all the above models, and feed it to another model to make final predictions on the test set.

All the above models are write out predictions in the form of probabilities that the sample text belongs to each of the 3 output classes (i.e. authors). The predictions from each of the above models, when good, will be submitted to Kaggle to evaluate the machine learning model.

I.C. Metrics

Since this is a multi-class supervised learning classification problem, Kaggle uses multi-class logarithmic loss to evaluate the accuracy of the model. Each excerpt (also referred as id) has one true class, and the model predicts probability for each author.
The metric is given by formula:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

where:
N is the number of observations in the test set,
M is the number of class labels (3 classes),
log is the natural logarithm,
y_{ij} is 1 if sample *i* belongs to class *j* and 0 otherwise, and
p_{ij} is the predicted probability that sample *i* belongs to class *j*.

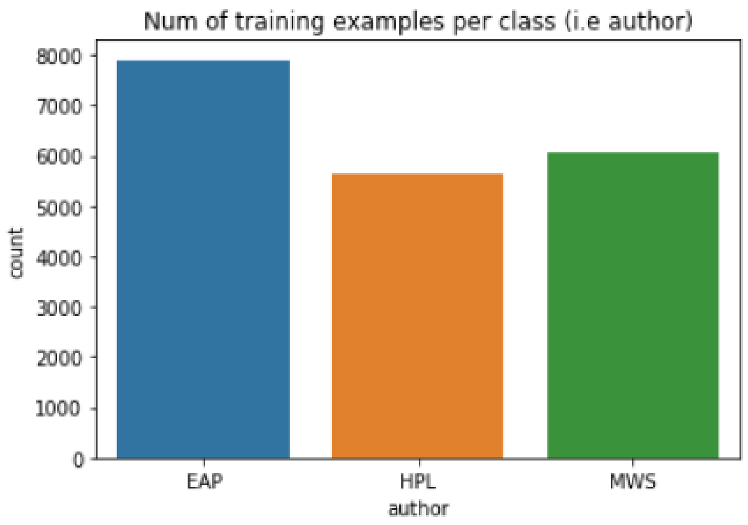
The justification for the above metric is as follows:
- For each sample, the true value ‘y’ is 1 for only 1 class (i.e. correct author). The terms for the other 2 classes (authors) vanish, since the true value ‘y’ is zero for them. This true value ‘y’ for correct class is multiplied by negative logarithm of the predicted probability for that class.
- The negative logarithm goes towards infinity when the probability goes towards zero. So the log loss function is very high when the true value and prediction don’t agree with each other. If the predicted probability is close to 1, the negative logarithm goes towards zero. So, the log loss goes function goes towards zero if the true value and predicted value agree with each other.

The lower the absolute value of this log loss the better is the model. If all the classifications are correct, log loss would be zero.

II. Analysis

II. A. Data Exploration

There are a total of 19579 samples in training data, and 8392 samples in test data.



Lets start with checking if the training data is balanced among the 3 authors. shows that the classes are fairly balanced with EAP, MWS, HPL having 7900, 6044, 5635 samples respectively.

Lets check if there is any information we could extract from the background of the authors. Based on wikipedia, the 3 authors in this data set are:

- Edgar Allan Poe (1809 – 1849), an American writer, editor, and literary critic.
- Mary Wollstonecraft Shelley (1797 – 1851), an English novelist, short story writer, dramatist, essayist, biographer, and travel writer.
- Howard Phillips Lovecraft (1890 – 1937), an American writer who achieved posthumous fame through his influential works of horror fiction.

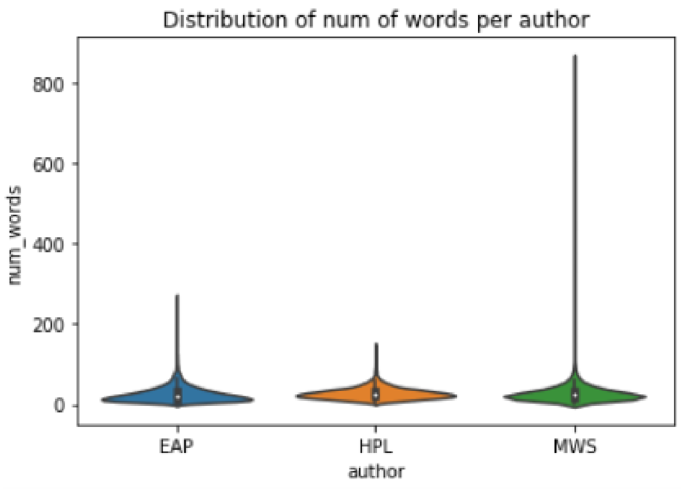
They are from the 1800s, so the word usage and vocabulary could be a bit different from the present. These classes will be referred to as EAP, MWS, HPL from here onwards.
Lets look at some sample texts from each author to understand if there are interesting patterns. They are shown below:

- Author EAP:
  - 'This process, however, afforded me no means of ascertaining the dimensions of my dungeon; as I might make its circuit, and return to the point whence I set out, without being aware of the fact; so perfectly uniform seemed the wall.'
  - 'In his left hand was a gold snuff box, from which, as he capered down the hill, cutting all manner of fantastic steps, he took snuff incessantly with an air of the greatest possible self satisfaction.'
- Author HPL:
  - 'It never once occurred to me that the fumbling might be a mere mistake.'
  - 'Finding nothing else, not even gold, the Superintendent abandoned his attempts; but a perplexed look occasionally steals over his countenance as he sits thinking at his desk.'
- Author MWS:
  - 'How lovely is spring As we looked from Windsor Terrace on the sixteen fertile counties spread beneath, speckled by happy cottages and wealthier towns, all looked as in former years, heart cheering and fair.'
  - 'A youth passed in solitude, my best years spent under your gentle and feminine fosterage, has so refined the groundwork of my character that I cannot overcome an intense distaste to the usual brutality exercised on board ship: I have never believed it to be necessary, and when I heard of a mariner equally noted for his kindliness of heart and the respect and obedience paid to him by his crew, I felt myself peculiarly fortunate in being able to secure his services.'

The number of words and number of punctuations used by each author could be interesting feature. The number of stopwords could also be an interesting feature to check. In NLP, stopwords are usually thrown away since they occur so frequently that they don't add value to classification. Lets plot and check next if the authors have different patterns here.

II. B. Exploratory visualization

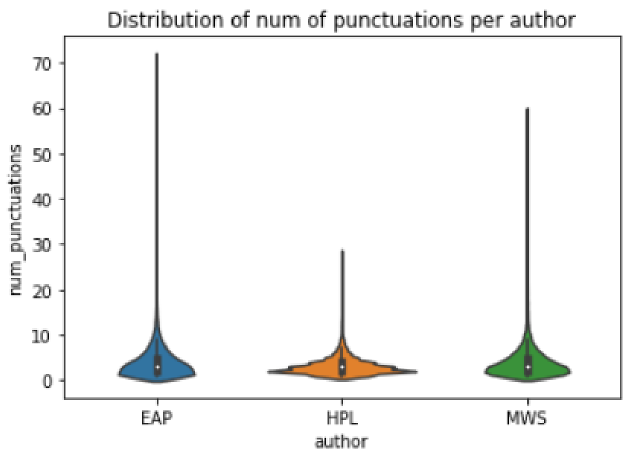
The violin plots here show the number of words, punctuations and stopwords distribution of each author.

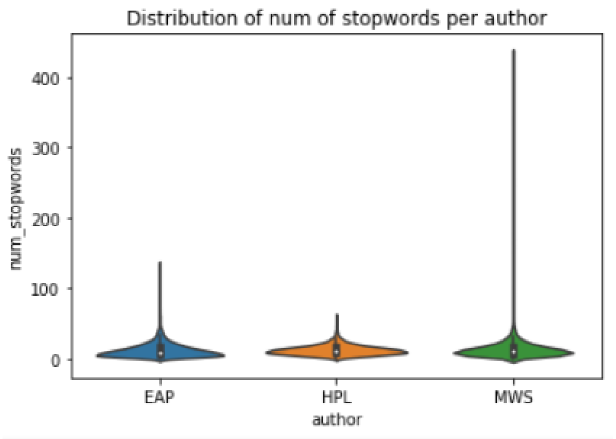


There definitely seems to be some differences in the usage of these 3 features among the authors from these plots.

For example, the mean number of words used by MWS in general is much lower, but there are many samples in 4th quartile. The number of punctuations used by HPL is much more around the mean. The mean number of stopwords used by MWS seems lower than other two classes.

So, we will add each of these as features in the training data.





II. C. Algorithms and Techniques

The following are the pre-processing techniques and algorithms used in this project. The "Appendix" section at the end of the report gives more details about these algorithms and techniques.

Pre-processing techniques:

First, the text data is converted to numerical format that the ML algorithms can use.

i. The standard machine technique in NLP text classification is to **Tokenize** (i.e. separate into words or characters) the text and use the frequency of each token appearing in each sample (called **Term Frequency or Count**) to classify the text. This will be one of the feature pre-processing technique used. This approach is called "Bag of words".

ii. But the words appearing in all the samples can get unnecessary extra importance and may not help in classifying the samples. This needs to be offset and the metric called **i**nverse document frequency (i.e. inverse of frequency of word in the document) usually takes care of it. This is multiplied with Term frequency to obtain **Term frequency-inverse document frequency (TF-IDF).** This is another standard pre-processing technique used in text classification.

iii. Text data usually needs to be "Stemmed" and "Lemmatized". **Stemming** crudely chops the ending of the words in order to reduce the feature space. For e.g. "running", "runs", "run" could be stemmed to "run" so that all 3 words are treated the same by machine learning models. **Lemmatization** on the other hand, gets the base word in a more correct form. For e.g., "leaves, "leaf" could be turned to "leaf" and treated the same. Both Stemming and Lemmatization will be tried before applying the Term Count or TF-IDF transformations.

iv. The words in the text data are usually converted to word vectors or embeddings so that **Neural Networks** can effectively use them for text classification. These word vectors are built based on their neighboring words using Neural network techniques. The following word vectors have been tried in this project. These word vectors are:
   - Either, pre-trained on large corpus like Wikipedia - Glove vectors,
   - Or, they could be trained on the new dataset using tool called "gensim" word2vec.
   - Or, some Neural network models learn them on their own during training.

<u>Algorithms:</u>

The following algorithms or models have been explored in this project:

- Logistic Regression
  - With Term Count/ TF-IDF
  - With/without Stemming Lemmatization.
  - With/without stopwords.
- Multinomial Naive Bayes
  - With Term count/ TF-IDF
  - With/without Stemming Lemmatization.
  - With/without stopwords.
  - Character level tokenizing
- Random Forests Classifier
  - With TF-IDF, and With/without stopwords.

For Neural networks, the following models are used:

- FastText (from Facebook team) with Embedding layer
  - Using no input word vectors
  - Using word vectors (that I created from training data using gensim)
  - Using Glove vectors
- Convolutional Neural networks (CNN) with Embedding layer
  - Using no input word vectors
  - Using word vectors (that I created from training data using gensim)
  - Using Glove vectors

For Stacking, the predictions from the above models have been stacked and run through

- XGBoost

The reason for choosing Logistic Regression is that its a quick linear model for classification. Multinomial Naive Bayes is generally good at text classification and is quick to train due to its assumption of words being independent from each other. RandomForests is a Decision tree ensembling approach that is different from the previous approaches.

The reason for the choice of FastText and Convolution Neural Networks is that both have a relatively smaller number of parameters to train than a Fully connected Dense network which could result in overfitting the data. (It has been empirically tested in this project)

XGBoost is a fast ensembling gradient boosting algorithm that is used by many Kaggle competition winners. The idea for stacking all the previous predictions here is to get the goodness from all these models and let XGBoost choose weights to assign to each of these models to get the final prediction.

Feature Engineering:

The following features have been created from the text and added to final Stacking model:

- Number of words in a sample
- Number of punctuations in a sample
- Number of stop words in a sample

II. D. Benchmark

In ideal case, if all samples are correctly classified, log loss would be 0.

For baseline accuracy, we can predict majority class in training set (i.e., EAP) for all the samples. When I made a submission to Kaggle, this gave a logloss score of 21.05 on private leaderboard (i.e. 70% of test data).

For benchmark accuracy, we can use Kaggle private leaderboard scores since the competition finished a month ago in December 2017. The log losses for some of the ranks are below :

Rank 1	:	0.024
Rank 50	:	0.265
Rank 100	:	0.289
Rank 500	:	0.385

III. Methodology

III. A. Data Preprocessing

1. Count vectorizer:
The standard machine learning algorithms like Logistic Regression, Multinomial Naive Bayes and Random Forests accept document term matrix. This matrix has each sample text translated into a sparse row of numbers. Each column is a word from the entire dataset and value represents number of times that word appears in this sample text. This is implemented using sklearn CountVectorizer.

2. TF-IDF vectorizer:
Re-weighting is done on top of above for each word count, based on number of times the word appears in the entire dataset. Frequent words appearing in all samples get less weight while words appearing in only few samples get more weight. Here sklearn TfidfVectorizer has been used. This also supports tokenizing characters instead of words, and I explored this too.

3. Stopwords:
Both the above pre-processors accept a list of stopwords. These stopwords can be excluded from the samples before building the matrix. NLTK package has list of “english” stopwords and it has been used here.

4. n-grams:
The number of words that need to be considered together when counting the frequency. Both the above models have options to specify this. For creating word embeddings or vectors, I’ve done pre-processing to include this in each sample so that n-grams are used by neural networks.

5. Stemming and Lemmatize:
I’ve overriden the build_analyzer function in CountVectorizer and TfidfVectorizer to evaluate if Stemming and Lemmatizing helps the model. This didn’t help improve performance.

6. Tokenizing for Keras neural networks:
Keras has a tokenizer that can tokenize each sample into words. A couple of pre-processing steps have been added to this based on the performance benefits seen. First step is retaining punctuations in the data and not filtering them out during tokenizing. Second step is adding 2-grams with “--” in between pairs of tokens (words or punctuation) for each sentence.

7. Wordvectors:
The gensim package has been used to create word vectors from the entire training dataset. Each word is represented as a 20 dimensional vector. The words that appear together or have similar context in the training dataset are located closer together in this 20 dimensional space.

8. Pre-trained Glove word vectors:
The pre-trained glove vectors from Stanford University website have been downloaded to be used during Neural network model training of Embedding layer.

III. B. Implementation and Refinement

At first, the training data has been first split into training and test (artificial) sets. The training set has been further split to train and cross-validation sets. The cross-validation set will be used for hyper-parameter tuning and the test (artificial) set will be used to locally evaluate the performance before submitting it to Kaggle.

Regular supervised learning models:

The following pre-processing techniques and their hyperparameters have been tried out:

a. CountVectorizer

b. Tfidf Vectorizer

Hyper-parameters:

- Include/ exclude stopwords
- ngrams: 1, 2, 3, 4
- Tokenize at character level or word level
- Include / exclude stemming and lemmatization

The following algorithms and their hyper parameters have been tried:

a. Logistic regression

- 'C': [0.1, 1.0, 10.0, 100.0]

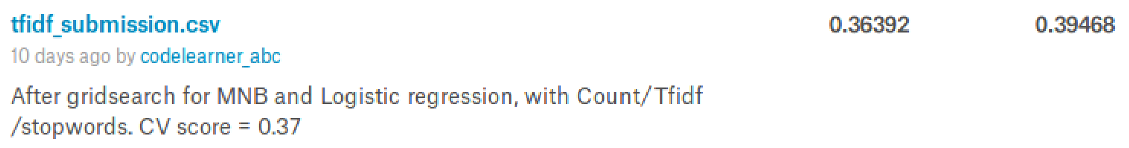
b. Multinomial Naive Bayes:

- 'alpha': [0.01, 0.03, 0.1, 0.3, 0.6, 1.0]

c. Random Forests:

- ‘n_estimators’: [10, 50]

GridSearchCV has been used to search for the hyperparameters.
Multinomial Naive Bayes with TF-IDF of 2-grams, including stopwords performed the best with a CV score of 0.37. This has been submitted to Kaggle and similar performance is observed on unseen test data. Snapshot is shown below. (Left score is private score, i.e. 70% of test data, while right score is public score, i.e. 30% of test data)


tfidf_submission.csv 0.36392 0.39468
10 days ago by codelearner_abc
After gridsearch for MNB and Logistic regression, with Count/Tfidf /stopwords. CV score = 0.37

Neural network models:

Next, I’ve tried using neural network models. Keras Embedding() layer needs the text samples to be tokenized (using Tokenizer) and all the sentences (or samples) to be padded to same length (using pad_sequences). The hyper-parameters tuned here are:

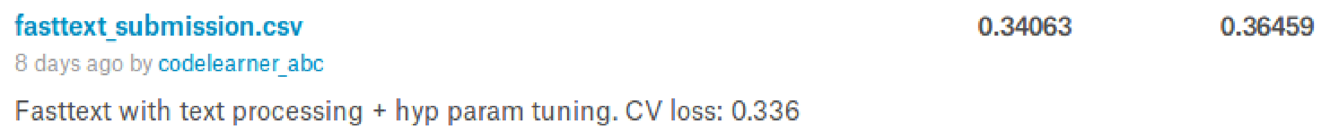
- Include or exclude punctuations as tokens
- n-grams during tokenizing
- Number of words used for tokenizing
- Word level or character level tokenizing
- Maximum length of sentence (chop longer ones and pad shorter ones with 0)
- Give or not give pre-trained word vectors to Embedding layer.

FastText: The number of layers in neural network after the Embedding layer is another hyperparameter. Adding a Dense() layer with any number of nodes and amount of Dropout() resulted in overfitting. This was noticed using Earlystopping and using a validation split of 0.05 percent. So, I used FastText model from Facebook which only has 1 GlobalAveragePooling() layer and Dense(3) layer with softmax at the output. The optimizer was set to 'adam'.

1. I've started with no pre-trained word vectors, and tuned the rest of hyper parameters mentioned above. The following set of values for hyperparameters with FastText model gave good perfomance:

- Include punctuations as tokens
- 2-grams during tokenizing
- Number of words used for tokenizing – only words that appear more than twice
- Word level tokenizing
- Maximum length of sentence - 99th percentile value among all training data
- No pre-trained word vectors to Embedding layer.

It gave a CV log loss of 0.336 and submission to Kaggle showed similar results:


fasttext_submission.csv 0.34063 0.36459
8 days ago by codelearner_abc
Fasttext with text processing + hyp param tuning. CV loss: 0.336

2. I've then downloaded pre-trained glove vectors from Stanford and created a embedding matrix that can be input to Embedding layer. The intent here is to provide a better starting point for the optimizer. But the performance on FastText with glove vectors hasn't been good with CV log loss of around 0.80. The reason here could be the vocabulary / word usage / context difference between Wikipedia on which Glove vectors are trained from the vocabulary in 1800's time period from which atleast 2 out of 3 authors belonged.

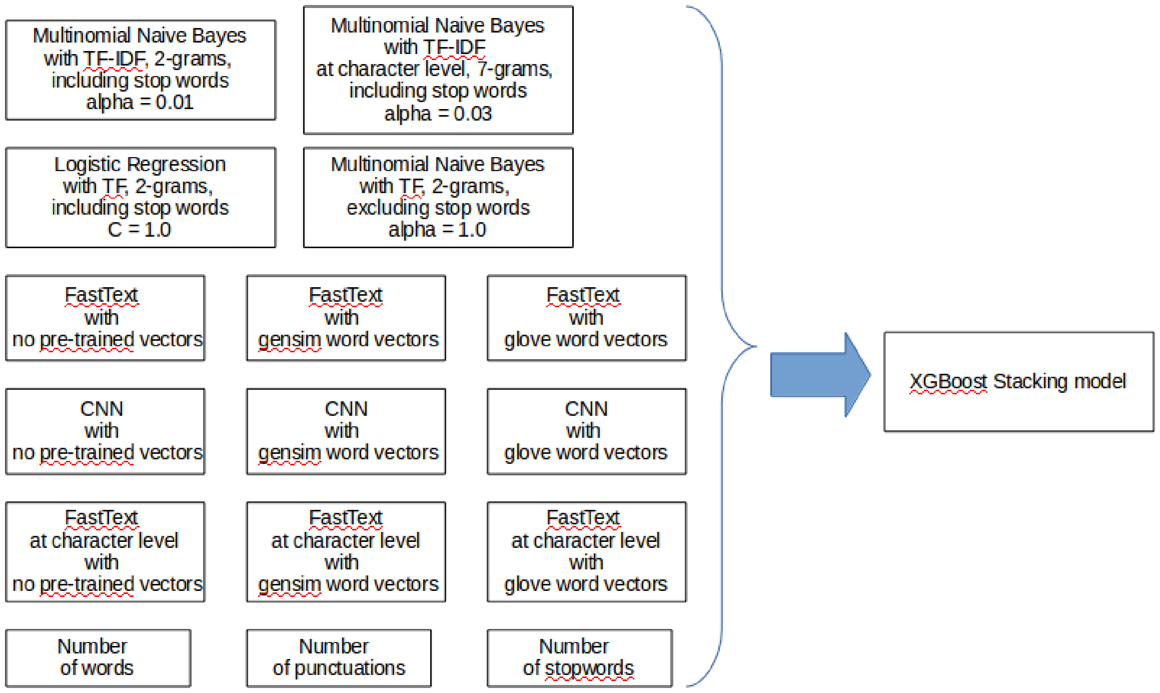
3. So, I then created word vectors based on the given training data. This involved installing gensim package and using the training data text columns as input to generate word vectors. The performance using these word vectors on FastText model gave similar results to using no vectors.

CNN: The other neural network model that above was Convolutional neural network. The number of nodes in CNN had to kept at a relatively small number of 32, with kernel size 4 and Relu activation to avoid overfitting. It was followed by a MaxPooling layer and a Dropout layer with high dropout of 0.8 to prevent overfitting. It is followed by a GlobalAveragePooling layer before a final Dense layer with softmax activation. These hyperparameters were obtained after lot of tuning to avoid overfitting. Earlystopping with validation split of 0.05 was used to to prevent overfitting.

The CNN was also trained with same set of hyperparameters for pre-processing as the FastText, as they resulted in least overfitting. Also, all the 3 variants of word vectors i.e., no pre-trained word vectors, word vectors from gensim and glove vectors, have been explored. CNNs in general produced similar or worse results than FastText. The reason here could be the relatively small data set compared to large set of parameters that needs to be trained.

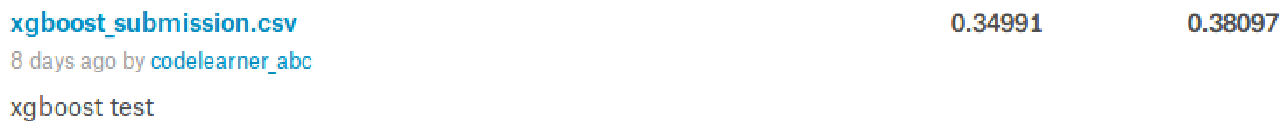
Final Stacking model

The predictions on training data from the above models have been added as features each of training samples, and only these predictions have been given as features to XGBoost, along with their labels, to get the final predictions. The idea here is to let XGBoost choose the goodness from all these models and train the weights to be assigned to each of these models. The architecture is shown below.



Multinomial Naive Bayes with TF-IDF, 2-grams, including stop words alpha = 0.01
Multinomial Naive Bayes with TF-IDF at character level, 7-grams, including stop words alpha = 0.03
Logistic Regression with TF, 2-grams, including stop words C = 1.0
Multinomial Naive Bayes with TF, 2-grams, excluding stop words alpha = 1.0
FastText with no pre-trained vectors
FastText with gensim word vectors
FastText with glove word vectors
CNN with no pre-trained vectors
CNN with gensim word vectors
CNN with glove word vectors
FastText at character level with no pre-trained vectors
FastText at character level with gensim word vectors
FastText at character level with glove word vectors
Number of words
Number of punctuations
Number of stopwords
XGBoost Stacking model

However, the performance did not improve as I expected. I realized the reason for this is a subtle data leakage. I will discuss more about it in the “Further Refinement” section.



xgboost_submission.csv 0.34991 0.38097
8 days ago by codelearner_abc
xgboost test

Coding complications

Working with Pandas dataframes introduced some challenges for implementing stacking model. If the training dataset is shuffled and split into training and CV sets and we’d like to add predictions of machine learning models as features, these features need to be added added to correct rows in the original training data. I could achieve this by efffectively using “.iloc”, “.loc”, etc attributes of data frames. However this complication goes away due to re-coding I did in “Further Refinement”.

III. C. Further model Refinement and Code Re-architecture

After noticing that the performance in stacked model is not improving, I realized a data leakage issue during stacking and re-architected the training method and code.

Issue with my previous Stacking implementation:
The training data was all seen by each of the supervised and neural network models when they made the predictions. These predictions have all been fit by the learning models. When such data is given to Stacking model (XGBoost here), the stacking model can overfit them. Stacking model should only be given predictions that are not used during training by the previous layer's models. Only then can it appropriately identify which models make better predictions, and can assign appropriate weights to them.

Solution – Use K-fold training:
The solution here is to use divide the training dataset into K folds. I've used K=5.

- In each iteration of training, 4 folds will be used for training and predictions will be made on the unseen 5th fold. This way the 5th fold was not directly seen by the training model.
- There will be 5 such training iterations per model. At the end of the 5 iterations, we can get a complete set of predictions on training data that was not directly seen by our models.
- This is also an implementation for K-fold cross-validation where the 5th set is used for CV. The mean CV log loss score from 5 iterations could be used as a rough metric for each model.

Code Re-architecture:
The final ipynb code submitted is thus completely re-architected to reflect this. A function called "run_kfold_training" has been coded which implements the above functionality. It accepts a couple of functions as parameters.

- The first function does pre-processing to generate data that could be used by training algorithms. This is needed for text documents as was mentioned before.
- The second function runs the actual model to generate predictions. "run_kfold_training" accumulates all these predictions.
- Another function "add_pred_features" is coded to add these features back to the training data. The coding complication with pandas Dataframes mentioned in earlier "Implementation" section does not appear in this implementation and that is an added benefit.

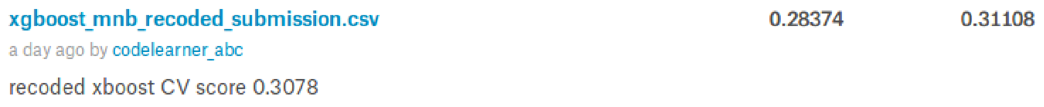
IV. Results

IV. A. Model evaluation and Validation

Model results

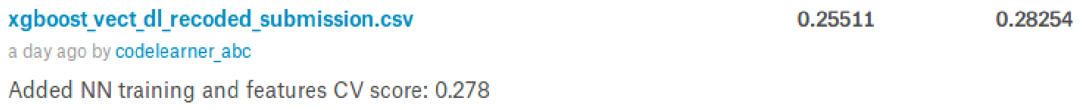
Each of the above individual training models had similar mean log loss scores after changing the implementation to K-fold training approach. This is because the original training for hyperparameters was done with appropriate use of training, cross-validation and test (artificial) sets from the training data. The results from Stacking model improve significantly after changing the implementation to K-fold training.

- The Stacking model from predictions of regular machine learning algorithms (i.e., no Neural network predictions) gave a CV log loss of “0.3078”. The Kaggle submission for it showed “0.283” private log loss score (70% test data).



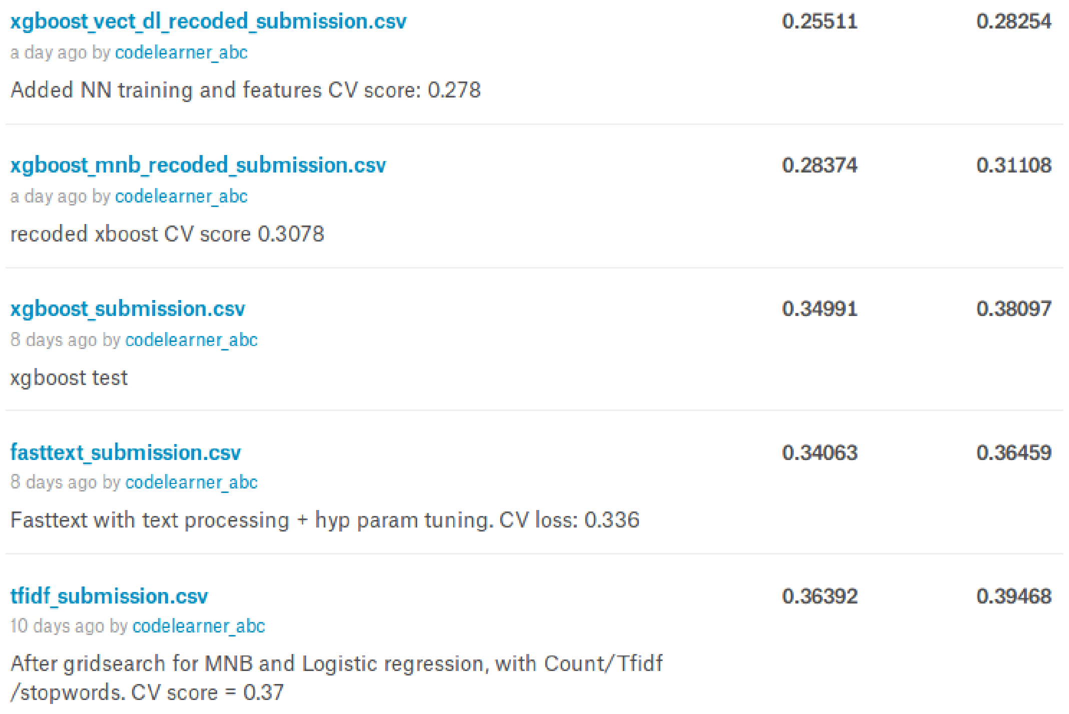
xgboost_mnb_recoded_submission.csv 0.28374 0.31108
a day ago by codelearner_abc
recoded xboost CV score 0.3078

- Including the Neural network predictions to Stacking model improved the CV log loss to “0.278” and the Kaggle submission showed “0.255” private log loss score (70% test data).



xgboost_vect_dl_recoded_submission.csv 0.25511 0.28254
a day ago by codelearner_abc
Added NN training and features CV score: 0.278

- As can be seen, the results have been gradually improved from using regular supervised machine learning algorithms at bottom, to neural networks in between, to Stacking model at the top.



xgboost_vect_dl_recoded_submission.csv a day ago by codelearner_abc Added NN training and features CV score: 0.278	0.25511	0.28254
xgboost_mnb_recoded_submission.csv a day ago by codelearner_abc recoded xboost CV score 0.3078	0.28374	0.31108
xgboost_submission.csv 8 days ago by codelearner_abc xgboost test	0.34991	0.38097
fasttext_submission.csv 8 days ago by codelearner_abc Fasttext with text processing + hyp param tuning. CV loss: 0.336	0.34063	0.36459
tfidf_submission.csv 10 days ago by codelearner_abc After gridsearch for MNB and Logistic regression, with Count/Tfidf /stopwords. CV score = 0.37	0.36392	0.39468

Sensitivity analysis

During the 5-fold training, only 4/5th of shuffled training data is used in each iteration. The CV scores per each iteration have been fairly close to the mean score of all the 5 folds. Below is snippet from the final stacking model (rounded to 5 decimal places)

```
Cross-val scores are: [0.27978, 0.28328, 0.27474, 0.27948, 0.27504]
Mean cross-val scores is: 0.27847
```

This indicates in some form that the model is fairly stable and not very sensitive to different subsets of training data.

IV. B. Justification

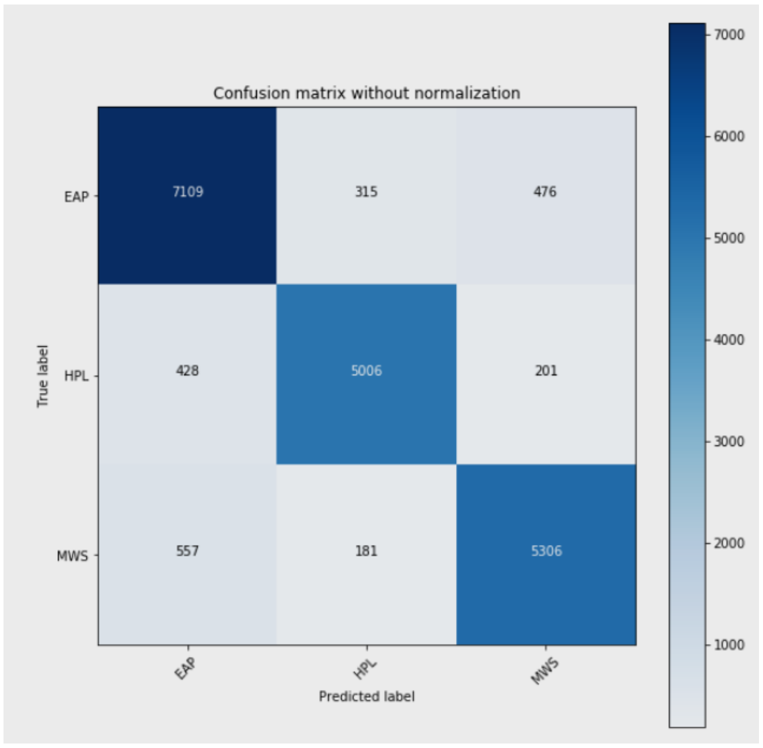
The model performance of “0.255” score on Kaggle falls short of the Rank 1 log loss of “0.024” which is close to ideal log loss of “0”.

Rank 1: 0.024 ; Rank 50: 0.265 ; Rank 100: 0.289 ; Rank 500: 0.385

However, it is a big improvement from baseline log loss score of “21.05”. This puts this model at rank 35 (top 3%) among 1244 participants if submission was made to the competition by December 15, 2017.

The justification for the last bit of performance improvement is due to Stacking the models together and re-architecture explained in “Section III.C”. Due to different nature of each of the learning models, each model would have got correct predictions for different samples. The Stacking model helps in combining these correct predictions to get a final model that has higher performance.

The accuracy of this model is 88.9% on the training set. 17421 out of 19579 samples are correctly predicted without overfitting too much. 2158 samples are classified incorrectly. The confusion matrix for this looks as follows, which shows that the model is not heavily biased to one class.



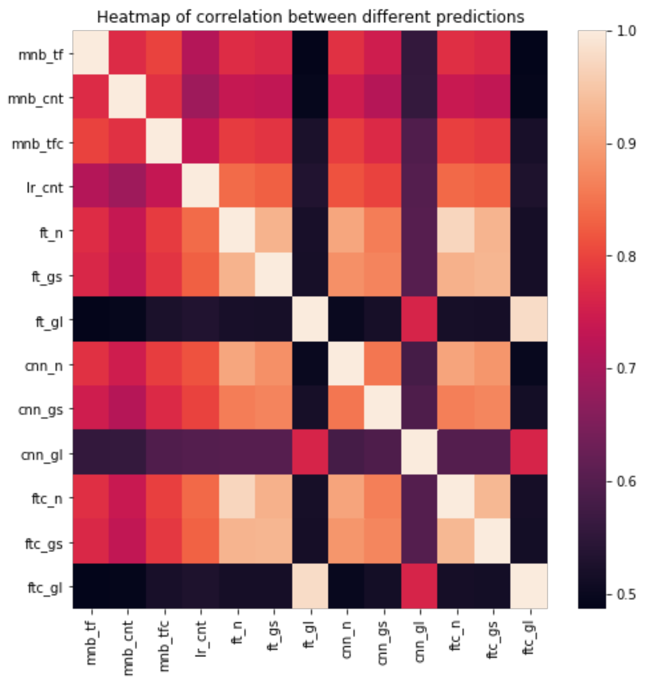
V. Conclusion

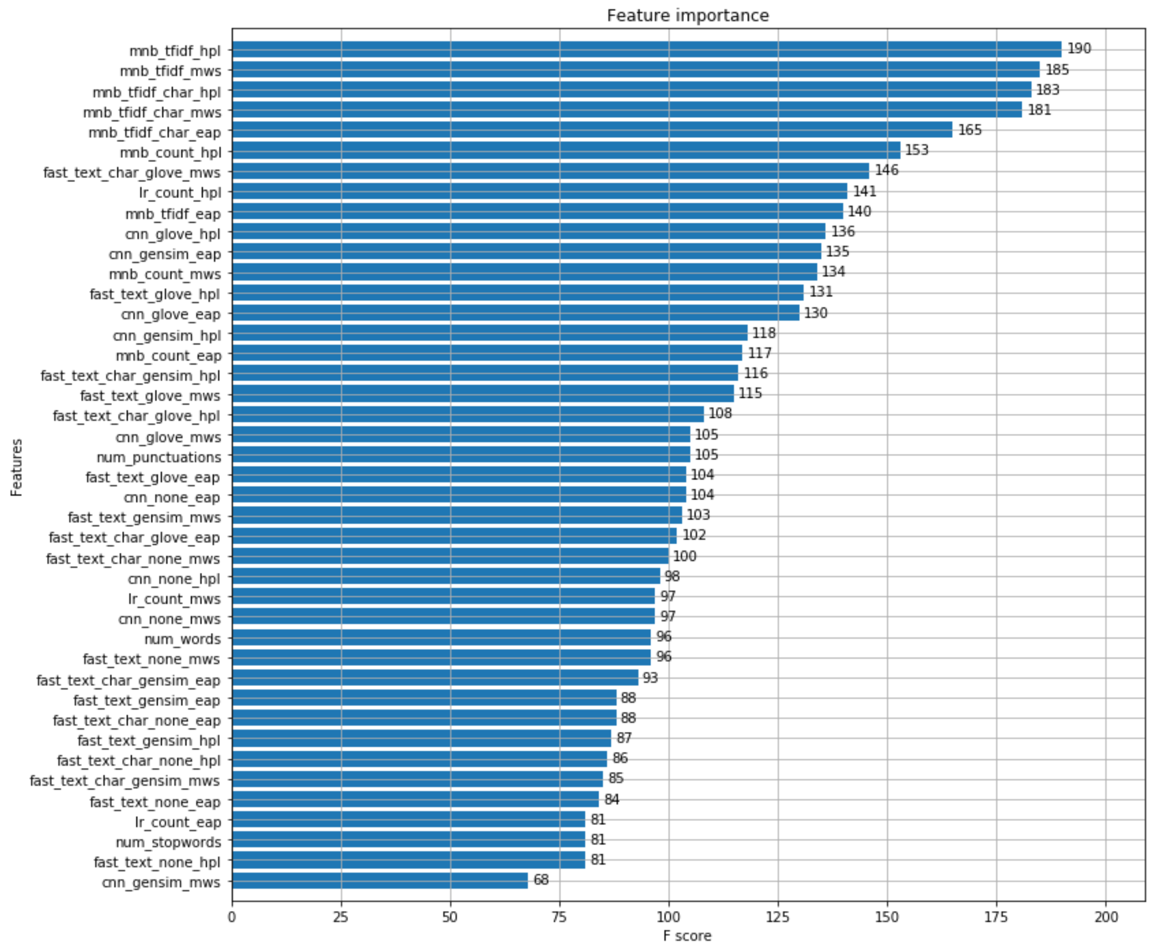
V.A. Free-form visualization

There are a few interesting observations seen:

1. The predictions made by different models don’t necessarily correlate. The heat map here shows the correlation between predictions between the various models that are input to the final Stacking model. This lack of correlation is good for Stacking model since it helps in capturing the goodness of these different models and avoids overfitting.

2. Even though performance of neural network models like FastText or CNN using glove vectors was not good (around 0.80 log loss), the F score (used to represent feature importance) in the final XGBoost stacking model of some of these glove vector predictions is high. This could be because F score is a naive metric for feature importance, or glove vectors had some good predictions which were used while splitting trees for XGBoost.





V.B. Reflection

In conclusion, various NLP and machine learning techniques have been explored in order to solve this text classification problem.

Both regular supervised learning algorithms and neural networks have been explored, along with a Stacking model that uses predictions from these models to generate final predictions.

The machine learning algorithms need pre-processing of text to numerical form - as TF or TF-IDF for regular supervised models, and word vectors or embeddings for neural network models. There are many hyper-parameters that need to be tuned here like n-grams, handling of stop words and punctuations. Neural network pre-processing has more such hyperparameters like the length of the word vector, maximum length of each sample and number of words to use for tokenizing.

The machine learning algorithms themselves have various hyper-parameters that need to be tuned like “alpha” for Multinomial Naive Bayes, ‘C’ for Logistic regression, ‘Number of layers, nodes per layer, dropout, and pooling layers’ for Neural network models. Early stopping using cross validation have been used to observe the training and CV error behavior, and hence avoid overfitting.

In the end, extra care should be taken to avoid data leakage as much as possible into the Stacking model. This leakage could result in no performance benefits in best case, to overfitting the training data in worst case. The K-fold training approach explained earlier has been used to mitigate this issue and improve performance.

V.C. Improvement

Some of the potential models that can further help improve the performance could be the state of the art Recurrent Neural Networks (RNNs) and Long-short-term-memory (LSTM) models. These can learn the relationship and patterns between words that appear close together and could potentially identify more complex patterns. In the bag of words approaches used in this project, the order or relationship is not completely retained. It only appears in some form when n-grams are considered.

Another thing to try would be to analyze the misclassified samples in training data and perform further feature engineering (i.e. add features which could classify them correctly) to improve the models.

VI. References:

https://en.wikipedia.org/wiki/Natural-language_processing
https://www.kaggle.com/c/spooky-author-identification
https://en.wikipedia.org/wiki/Edgar_Allan_Poe
https://en.wikipedia.org/wiki/Mary_Shelley
https://en.wikipedia.org/wiki/H._P._Lovecraft
http://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction
https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html
https://arxiv.org/pdf/1607.01759.pdf
http://blog.kaggle.com/2016/12/27/a-kagglers-guide-to-model-stacking-in-practice/

VII. Appendix
Details of NLP and Machine Learning Techniques

Tokenization
In an NLP approach called “Bag of words” approach, the given sample (or document from here on) is usually converted first into tokens by removing the spaces or punctuations. The tokens could be words or characters, and the punctuations could be also be optionally added as tokens.

TF-IDF
The number of times (i.e. count) each word appears in a particular document can be used to classify the document. This is referred to as Term-frequency (TF). But words that appear frequently in all the documents in the corpus (i.e. training dataset) could not additional value to classification. So, the term frequncy is multiplied by Inverse of the document frequency (i.e. number of times word appears in entire corpus) to obtain Term-frequency Inverse document frequency (TF-IDF) which could be a better featire for each word.

Stemming and Lemmatization
Stemming and Lemmatization reduce the inflectional and derivational forms of words to their base forms. For example we may want to treat “running, “runs” and “run” the same way. Stemming is a crude heuristic that chops the end of the words, while Lemmatization is a more proper heuristic that uses more analysis techniques to come up with base form. Stemming can convert all the 3 above forms to “run”, but may not work on a word like “saw” or “leaves”. It may return “s” and “leav”. Lemmatization on other hand can possibly return “see” and “leaf” for example.

Multinomial Naive Bayes
Multinomial Naive Bayes is an extension of Naive Bayes to multi-class classification problem. Bayes model can be used to solve text classification problems. The probability of a class given a document can be found using the probability of the document given the class and the probability of the class. Naive Bayes is a form of Bayes model in which the variables are assumed to be independent of each other. The variables in this case are the words in a document. Due to Naive bayes assumption, the probability of a document given a class can be obtained by multiplying the probability of each of the words appearing in the document of a particular class. These probabilites can be easily found from the corpus and therefore solve the classification algorithm.

Logistic Regression
It is a linear classification algorithm in which a linear model is passed through sigmoid (or logistic) function to obtain the probability of the output class. The decision boundary learnt here is still a linear one.

Neural Networks
Neural networks have several logistic regression units stacked in single layer and several such layers stacked one after another to make final prediction. Since the linear logistic model predictions are passed through several non-linear sigmoid functions, neural networks can learn complex non-linear decision boundaries.

FastText
This is a model from the Facebook AI team, wherein an Embedding layer (in Keras) is followed only by 1 GloabalAveragePooling layer and then a Dense output softmax layer. The Embedding layer converts the text sequences in training data and converts them to vectors that could be used by following layers of Neural network, under the hood. The simplicity of just 1 GlobalAveragePooling layer significantly reduces the number of parameters to learn and avoid overfitting when only relatively small dataset is available.

CNNs
In Convolution Neural networks, nodes at each layer are connected to only a few number of nodes in previous layer compared to Dense fully connected Neural networks where nodes are connected to all nodes in previous layer. This is especially relevant when patterns are localized to neighboring dimensions of a data point, like in image classification. In text classification, this can be analogous to n-grams. This fewer number of connections helps to reduce the number of parameters and hence overfitting the data.

XGBoost
Boosting is an ensembling approach in which several weak learners (usually small decision trees) are learnt in an iterative fashion, where each subsequent weak learner attempts to put additional effort to learn mis-classifications of their predecessor weak learner. Gradient boosting generalizes this approach as an optimization problem on arbitrary differentiable loss function, which then allows it to apply techniques like gradient descent. XGBoost is a fast and computationally efficient implementation of gradient boosting model.

Appendix for references

https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html
https://en.wikipedia.org/wiki/Gradient_boosting
http://xgboost.readthedocs.io/en/latest/model.html