Machine Learning Engineer Nanodegree

Capstone Project

Japtej Singh Arneja
September 7th, 2019

I. Definition

Project Overview

Natural language processing (NLP) is a subfield of computer science and artificial intelligence that is concerned with interactions between computer and humans[1] and covers manipulation of natural language like speech and text by computers. Text is everywhere – newspapers, web pages, documents, SMS, emails and online surveys are all examples of textual information. Still, the ability to process and extract insights from textual data using computers was limited up until 1990 as researchers mainly relied on bespoke set of hand-written rules[2] e.g. writing rules for grammar, stemming, etc. However starting 1990s, the usage of machine learning for NLP applications gained momentum and widespread industry adoption and especially in the last decade, with the advent of higher computing power (e.g. GPUs), machine learning and deep learning frameworks have penetrated every aspect of NLP achieving state of the art performance on many textual tasks.

In this project, various machine learning and deep learning frameworks are employed for mapping a sentence/excerpt to its respective author in the horror domain. In the process, performance comparison of conventional ML frameworks is also done with some of the more contemporary ones on textual data. The problem is a subject matter of an old kaggle challenge[3] which was launched as part of a Halloween playground competition.

Problem Statement

The problem is to be able to map textual excerpts to their respective (three) authors. The author set (target labels) consists of three authors namely:

1. Edgar Allan Poe (EAP),
2. HP Lovecraft (HPL),
3. Mary Wollstonecraft Shelley (MWS)

Specifically, the excerpts are all taken from spooky stories penned by each of these authors and hence the fundamental topic (horror) is the same – therefore the prediction algorithm has to mainly rely on inferring the writing styles to be able to predict the author. The intended solution is to be able to construct a mathematical classifier that automatically predicts the author of any sentence with high confidence (i.e. high prediction probability). Each sentence belongs to only one author and hence the model performance can be directly evaluated as per the chosen metric. Broadly, the following tasks would be performed:

1. Download and preprocess the data from kaggle[3]
2. Conduct text data exploration/visualization to generate early insights
3. Convert textual data into features that could be used to train a classifier that can predict the author of each sentence
4. Evaluate the model performance on holdout test set (carved from within train set), primarily based on the evaluation metric described in the *Metrics* section below. Chose the model with best performance
5. In the end, provide a critique on the performance of each model class on the textual data at hand

Metrics

For this project, *categorical cross-entropy* (or multi-class logarithmic loss) shall be used as the evaluation metric which is also the metric used in the Kaggle competition[3].

Cross entropy is based on the concept of maximum likelihood of predictions i.e. a model which predicts higher probabilities for the actual observed events is better than models that don't quite does so. Mathematically, cross entropy (or log loss) is given by the expression:

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

where, N = number of observations (text excerpts in this case)
M = number of classes (3 in this case)
y_{ij} = 1 if observation i belongs to class j else 0
p_{ij} = predicted probability that observation i belongs to class j

The theoretical range of cross entropy loss is 0 to ∞.

The benefit of using *cross entropy* instead of other simpler metrics (like accuracy) is that cross entropy rewards more deterministic models that can predict with high confidence (i.e. high prediction probability). Hence, a model that consistently predicts the correct author with ~60% probability (say) would have high accuracy but still have low cross entropy. In summary, the cross entropy metric would avoid choosing model which gives borderline predictions. This is important as the model needs to be robust to slight changes in future writing styles of the authors as writing styles can change slightly with time. A more deterministic model would be able to still show good performance in such variation scenarios.

In addition to the cross entropy metric, *accuracy* would also be tracked as a helper evaluation metric. Accuracy can be defined as:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

In other words, *accuracy* is the sum of true positives and true negatives as a fraction of total predictions.

II. Analysis

Data Exploration

The dataset used is in the form of a *csv* file and is procured from a past Kaggle competition[3] titled *Spooky Author Identification*. The train dataset contains ~19k text excerpts from the novels of three authors viz.:

1. Edgar Allan Poe (EAP),
2. HP Lovecraft (HPL),
3. Mary Wollstonecraft Shelley (MWS)

There is also a test dataset with ~8k excerpts but that would not be used as it doesn't have author labels available (hidden for kaggle scoring purposes) – instead the test data would rather be carved from within the train set (10% of train set). The train dataset contains three columns viz.:

- **id**: unique ID for each sentence excerpt
- **text**: the actual sentence excerpt the author wrote
- **author**: name of author who wrote the sentence (one of EAP, HPL and MWS)

The number of words per sentence have a high variance with a range from 2 to 861.

Below is a sample of the data (first 5 observations):

id	text	author
id26305	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.	EAP
id17569	It never once occurred to me that the fumbling might be a mere mistake.	HPL
id11008	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.	EAP
id27763	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.	MWS
id12958	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.	HPL

Some things/issues to consider are:

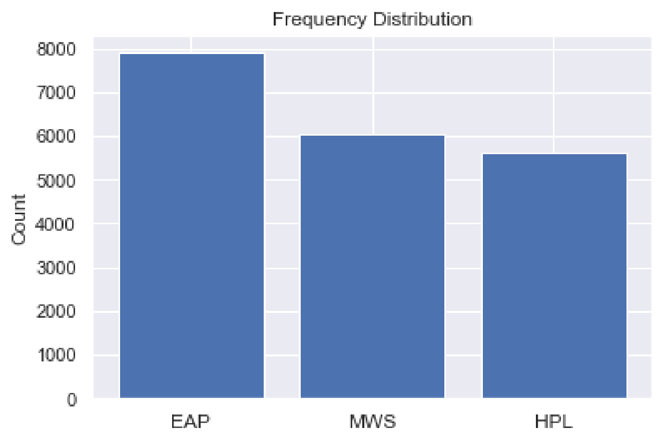
- Textual data needs to be converted to some sort of numerical features in general which can be used as input to ML models
- Sentences contain multiple punctuation and stop word tokens which could possibly be removed to check if they increase model performance
  - Removing punctuation doesn't always increase performance however. For e.g. there are multiple sentences in the train corpus ending with three dots (…), all written by EAP. Hence, the model can possibly use this and other punctuation patterns for better prediction
- Almost all the excerpts are single sentences and hence sentence tokenization is not required
- Word count per sentence have an interquartile range of 20 words (15 to 34 words) with a high overall variance (minimum word count=2 and maximum word count=861). This could be a problem for training neural net models which need fixed sized sentences as input and hence most of the sentences have to padded with lots of 0s to get up to the maximum (same) desirable length, leading to sparse inputs
  - As a remedy, sentences are clipped such that they can have a maximum word count threshold determined by 95th percentile of the word count distribution (i.e. 58 words). Thus, most of the sentences are not clipped but still have a more dense input format
- The train dataset only contains ~19k sentences and word embedding trained on such relatively small corpus might not be able to generalize to test data

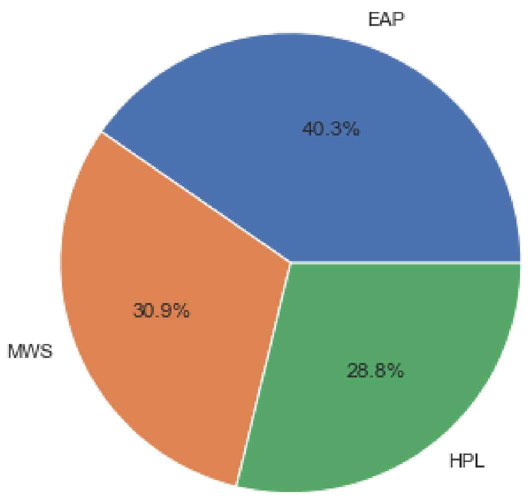
- As a remedy, fixed pre-trained word embeddings are used during neural net model training

Exploratory Visualization

Author frequency distribution

The bar chart and pie plot below show the author distribution in the data. It can be observed that classes are only slightly imbalanced with EAP having 40% share and HPL/MWS closely following at ~30% share each. Hence, as such, sub-sampling is not required to address this slight class imbalance and the data can be taken as-is.

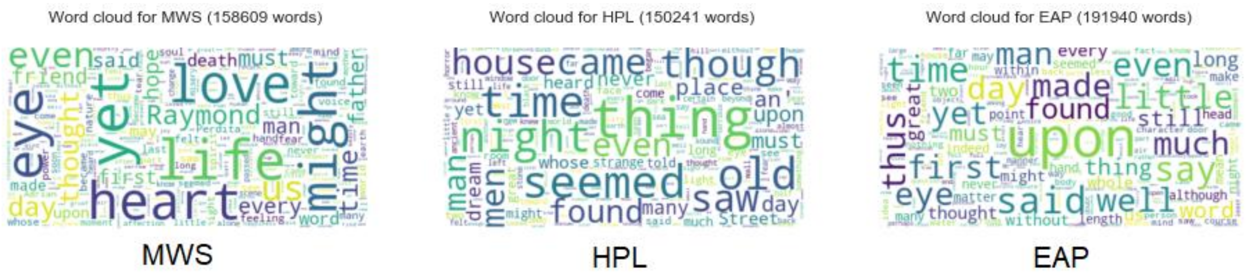




Word clouds for each author

Word clouds for different authors are also drawn in order to gain early insights into the choice of words and hence the writing styles of each author. Note that stopwords have been removed from the below clouds and the clouds only contain top 1000 words (ranked by frequency). It can be observed that usage of words are indeed different among authors (zoom into the word clouds for details):

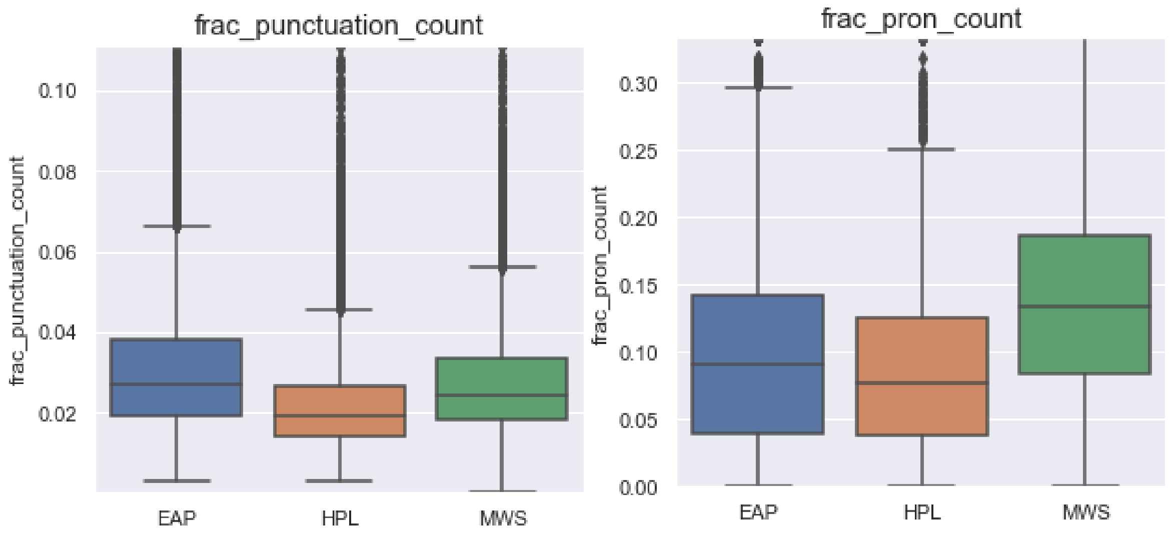
- MWS uses more abstract words like *love*, *heart*, *life*, etc.
- HPL uses words like *night*, *old* and *thing* maybe to create a more spooky plot
- EAP doesn't seem to be loading heavily on specific words and uses a diverse vocabulary


Word cloud for MWS (158609 words)
Word cloud for HPL (150241 words)
Word cloud for EAP (191940 words)
MWS
HPL
EAP

Box plots for text meta features

Finally, distribution of text meta-features across authors is also analyzed in order to gain early insights on the utility of these features for model classification. Couple of meta-features with maximum differences across authors are shown below:

- EAP has the highest fraction of punctuations across all his sentences
- MWS has the highest fraction of pronouns across all his sentences



Algorithms and Techniques

The following classes of algorithms/models are fitted on the text excerpts:

- Conventional ML models
- Stacking ensembles of ML models
- Neural net based models
- FastText model (by Facebook)

Conventional ML models

Here, three models are fitted viz. Multinomial Naïve Bayes, SVC and XGBoost. Each of the algorithms is fitted on three sets of text features, viz. text meta-features, count features and TF-IDF features. Each of the techniques is described in detail below and also how each of them is suitable for the problem at hand:

- ***Multinomial Naïve-Bayes***: Naïve Bayes is an adaptation of Bayes theorem which assumes independence of features. Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred and is stated mathematically as the following equation:

  $$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

  If X is a set of features given by *(x_1, x_2, x_3, ….)*, then Naïve Bayes can reduce the above equation to the below i.e. it naively assumes independence of features and simplifies the problem:

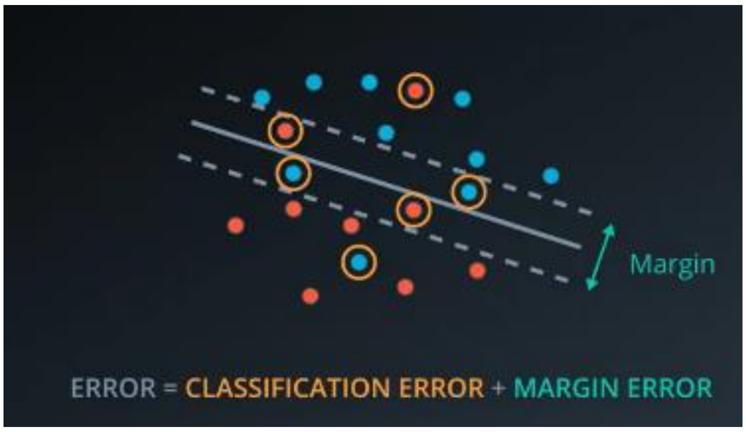
  $$P(y|x_1, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)}$$

  Similar equation can be then developed for the complementary probability of *P((1-y)/ x1, x2, x3, ….)* and since *P(y) + P(1-y) =1*, the probabilities can be solved for.

  Multinomial Naïve Bayes is a variant of Naïve Bayes where feature vectors represent the frequencies with which certain events have been generated by a multinomial distribution. For e.g. in our data, it can help us predict probability of an author (*y*) given that the features contain certain set of words (*X*).

  Benefits of Multinomial Naïve Bayes for our problem: Multinomial Naïve Bayes algorithm is fast and can get really good results when data available is not much (~ a couple of thousand tagged samples) and computational resources are scarce. It also is one of the most used models for text classification as text data generally has large number of features.

- ***Support Vector Classifier (SVC)***: They are similar to Logistic regression however SVC additionally picks the dividing hyperplane that maximizes the margins between both the target classes. Specifically, the points near the margin are called support vectors and are used for finding the best hyperplane. The error term in SVC consists of classification error (as in Logistic model) and also margin error as shown in the figure below.

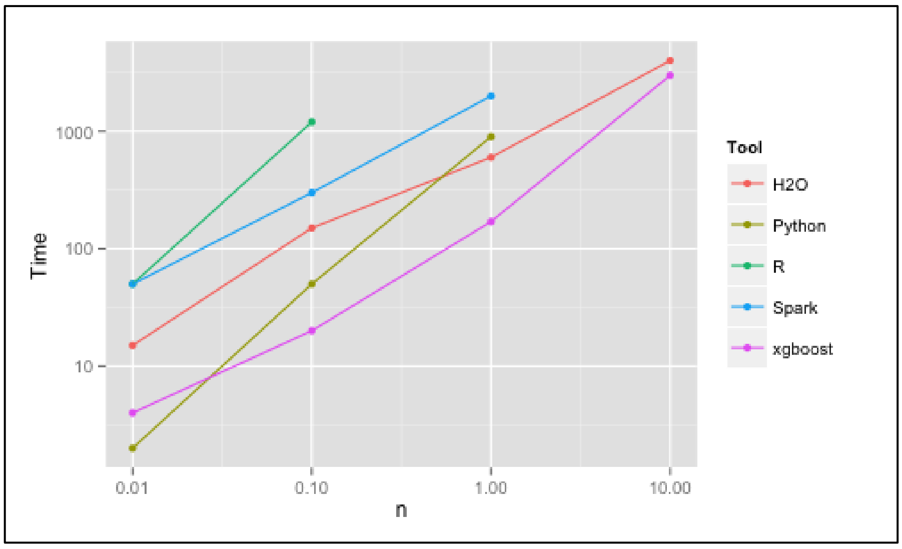


Error components of SVC model

Additionally, SVC also supports non-linear decision boundaries using kernel trick. Specifically, the radial basis function kernel (rbf) has shown great adaptability to a variety of tasks and would also be used for our project.

Benefits of SVC for our problem: Since SVCs use overfitting protection, which does not necessarily depend on the number of features (since they care only about marginal support vectors), they have the potential to handle large feature spaces[4], something which is common in textual data. In addition, SVCs are well suited for problems with dense concepts and sparse instances[4], again something which is common for text classification problems.

- ***XGBoost***: XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. It is similar to gradient boosting machines, however XGBoost uses a more regularized model formalization to control over-fitting, which gives it better performance (see figure below)



Execution speed of XGBoost is highest[5]

Benefits of XGBoost for our problem: Since text data have large number of features, XGBoost's faster performance can come in handy. It is also the algorithm of choice for many kagglers even for text classification[6].

Stacking ensemble of ML models

Ensembling is a powerful technique to combine several models to produce an even better model. In this project, a stacking ensemble would be developed wherein a combiner algorithm is trained to make a final prediction using the predictions of other algorithms as inputs.

Neural net based models

Here, three neural net models classes are fitted viz. LSTM, GRU and CNN. Each of the architectures is fitted on pre-trained GloVe word embeddings (42B tokens 300-dimensions) as neural net models are known to work best with dense representations like word vectors. Also since our train corpus is not very big (~19k sentences), the pre-trained embeddings are not trained further (as part of neural net training) to improve generalization and avoid overfitting. Each of the techniques is described in detail below and also how each of them is suitable for our problem:

- ***LSTM***: LSTMs (Long Short Term Memory) are a special class of RNN that are capable of learning long-term dependencies. LSTMs have inbuilt protection against the vanishing gradient problem, something which is common in vanilla RNN networks. LSTMs have the ability to remove or add information to the cell state memory, carefully regulated by structures called gates[7]. There are three types of gates viz. input gate (determines if current cell matters), forget gate (determines if past needs to be forgotten) and output gate (determines the extent to which the cell need to be exposed). One problem with vanilla LSTMs is that they only store past information. To overcome this, an adaptation of LSTMs called *Bi-directional LSTMs* can be used that stores information both ways.

  Benefits of LSTM for our problem: LSTMs have shown great performance on sequential data including textual data and can use variable past contextual information for making predictions, which is extremely handy for text classification. LSTMs control past context by regulating the flow of information using its gates.

- ***GRU***: GRUs (Gated Recurrent Units) are similar to LSTMs as both have the same goal of tracking long-term dependencies effectively while mitigating the vanishing/exploding gradient problems, but GRUs are simpler as they only have two gates viz. update gate (determines previous memory to keep around) and reset gate (determines how to combine new input with previous value).

Benefits of GRU for our problem: GRU have most of the useful properties of LSTMs for textual data. Additionally, as GRUs have fewer gates and hence fewer parameters, it can train faster and generalize well even on a small train corpus. This suits well for the data at hand as relatively small corpus of ~19k sentences exists in this case.

- ***CNN***: CNNs (Convolutional Neural networks) are regularized versions of multilayer perceptrons[8]. In general, CNNs use the spatial arrangement of the data to its advantage and shine in areas like image recognition as images have spatial properties. In the NLP world, CNNs have shown good performance with text classification as it can take advantage of word sequence ordering within sentences and find word patterns irrespective of their location in the sentence.

  Benefits of CNN for our problem: Since CNN have fewer parameters, they are one of the fastest neural nets to train on textual data. Moreover, they particularly shine in text classification tasks as they are adept in finding location invariant word patterns in the sentences that are useful in text classification.

FastText (by Facebook)

Finally, I also explored character based n-gram models like Facebook's FastText, which are faster even while running on CPU, work better for small train corpus and also address out of vocabulary (OOV) problem to a large extent. The *fasttext* package in python requires input in a special format i.e. complete observation in one line with labels (preceded by the literal *_label_*) before the sentence. The FastText python implementation automatically forms features based on external train file path.

Benefits of FastText for our problem: Since FastText model uses character n-grams, it can help with out-of-vocabulary problem for words that the model hasn't seen in train set. Also, the fast execution speed of the algorithm implies that several hyperparameter combinations can be tried and tuned in relatively quick time.

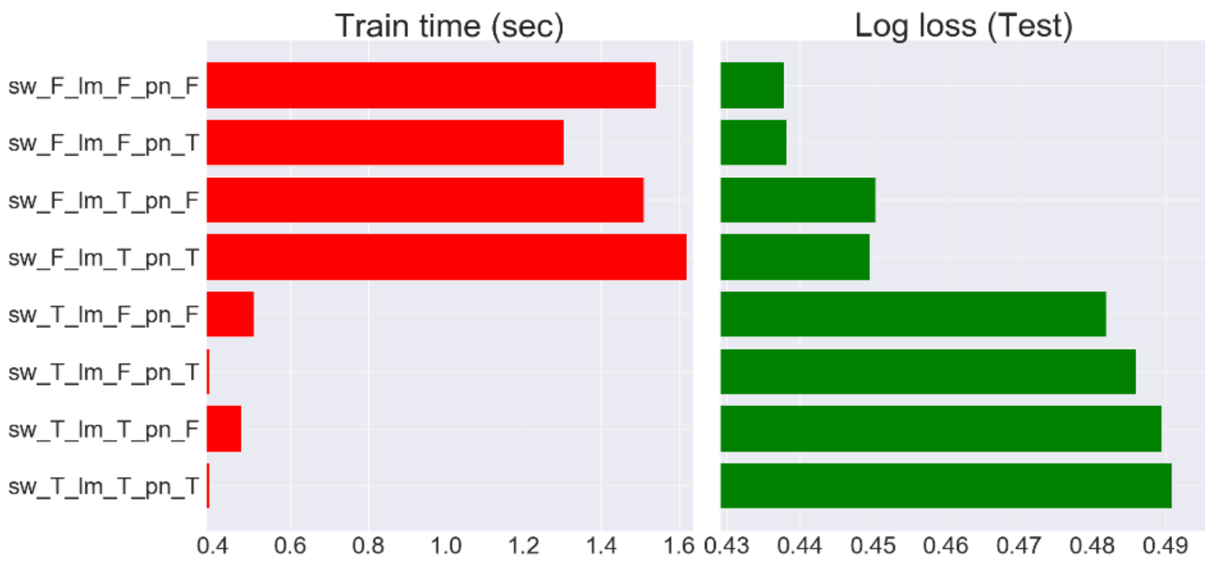
Benchmark

The benchmark model was multi-class logistic regression model fitted on single words (1-gram) bag of words input. The features were obtained by running the *CountVectorizer* function of *sklearn* (with *ngram_range*=(1,1)) on pre-processed text sequences and then passed to a standard Logistic classifier. Various pre-processing schemas were tried before fitting the *CountVectorizer* to our sentences.

For reference, an example schema nomenclature is as follows:

sw_T_lm_F_pn_F: schema with stopwords removed (T), without lemmatization (F) and without punctuation removal (F)

The results were as follows:



As can be seen, the best model (lowest log loss) in fact had no preprocessing (*sw_F_lm_F_pn_F).* This could be due to few reasons:

- Stopwords usage is significantly different among authors and the model is able to infer important writing patterns by including those
- Choice of verb form also is an important parameter in author prediction which is lost on performing lemmatization
- Punctuation usage as well is different among authors and helps in better predictions. For e.g. there are multiple sentences in the train corpus ending with three dots (…), all written by author EAP

Hence, the Logistic single n-gram model **with no preprocessing** was used as the benchmark model. Following was the performance of the final benchmark model chosen:

- Log loss (evaluation metric): **0.438**
- Accuracy (helper metric): **0.831**

III. Methodology

Data Preprocessing

Following were the pre-processing and feature extraction steps used before model implementation:

- ***Encoding target author labels***: First, encoding the author classes into numeric labels was done using *LabelEncoder* function of sklearn. The final encoding obtained was as follows:
    - *EAP*: 0
    - *HPL*: 1
    - *MWS*: 2

  A one hot encoding was also additionally done using *np_utils.to_categorical* function of keras. The one hot encoded features were specifically used by the neural net models.

- ***Splitting data into train and test set***: The data was then split into 10% test data and 90% train data. Following were the number of observations in each of the sets after splitting:

    - *Train data*: 17621 observations
    - *Test data*: 1958 observations
- ***Text cleaning/preprocessing***: A function for textual pre-processing was built for stopwords removal, lemmatization, punctuation removal and lowercasing, but wasn't eventually used as it was leading to worse performance (as mentioned in the Benchmark model discussion above). Also spelling corrections were not required as the texts were clean excerpts picked from novels

- ***Feature extraction for ML and stacked ensemble models***: As ML models need numeric features for model fitting, the following numerical features were constructed on the textual excerpts:
    - Text meta-features which provide a broad overview of the sentences. They included:
        - General meta-features including word count, character count, average word length, punctuation count, upper case word count and title case word count, both in absolute and fractional form
        - Parts of Speech (PoS) based features including count of nouns, verbs, adjectives, adverbs and pronouns, both in absolute and fractional form

    - Count based bag of words features were obtained by running sklearn's *CountVectorizer* function on the text. This resulted in a sparse matrix of word token counts. The count features were only based on single-gram words as any higher n-grams were leading to worse performance
    - TF-IDF feature were also obtained as they are generally more robust than count based features. They were obtained by running sklearn's *TfidfVectorizer* function on the text. This again resulted in a sparse matrix of TF-IDF word features. The TF-IDF features were also only based on single-gram words as any higher n-grams were leading to worse performance

- ***Feature extraction for Neural net models***: Since neural net models can consume and capitalize on the sequential nature of word order in sentences, they needed sequential features. For this purpose, padded fixed length integer sequences of words (for each sentence) were built using *Tokenizer* API of keras and used as inputs to neural net models. The padded fixed length sequences were fed into the Embedding layer of keras which also was the first input layer of the model. Also, pre-trained GloVe embeddings (42B token 300-dimensional) were used to initialize the word vectors in the keras Embedding layer
    - The sentences had a huge word count variance from 2 words to 861 words. However, the upper end was mostly outliers as even the 95th percentile word count was only 58 words. Since keras Embedding layer requires integer encoded word sequences of same length which is typically equal to largest sentence size (i.e. 861 words), the padding required was rendering most sentences as sparse with lots of 0s at the end. This was leading to worse model performance. To avoid this problem, a maximum word length cutoff equivalent to the 95th percentile (i.e. 58 words) was chosen and longer sentences were clipped to those many words. This helped built more dense input sequences without losing much data as most sentences had <58 words

Implementation

The implementation was carried separately for the following model classes:

- Benchmark model
- Conventional ML models
- Stacked ensemble of ML models
- Neural net models
- FastText model

Benchmark model

For the benchmark model, the below steps were followed:

- All possible text pre-processing schemas were prepared
    - For e.g. one schema was sw_T_lm_F_pn_F which implies a schema with stopwords removed (T), without lemmatization (F) and without punctuation removal (F)
- The following steps were performed for each text processing schema:
    - A *CountVectorizer* feature vector is prepared on the processed words with single-gram bag of words setting
    - A *Logistic* classifier is then fitted onto these count based features for the train dataset. The solver for the classifier was '*liblinear*' and multi_class setting was set to '*ovr*'
    - The log loss (evaluation metric) and accuracy (helper metric) is evaluated on test set for the fitted model
- After fitting a model for each text pre-processing schema, a comparison of test data log loss and accuracy is done across all schemas
- Finally, the schema with the minimum test log loss was selected as the benchmark model

Conventional ML models

For ML model fitting, an exhaustive function called '*train_and_compare_ML_models*' was defined that did the following:

- Divided the train samples into subsets of 1%, 10% and 100%. The intent was to observe the performance and scalability of the model training as the amount of train data iteratively increases
- Fitted the following ML classifiers on each train partition. The classifiers were:
    - Multinomial Naïve-Bayes
    - SVC
    - XGBoost
- Calculated the following metrics for each combination of classifier and train partitions:
    - Training time
    - Prediction time (on test set)
    - Accuracy on train and test set (to check overfitting)
    - Log loss on train and test set (to check overfitting)
- Provided a visual plot for all the above performance metrics for each train partition and classifier combination

The following steps were performed for ML model fitting:

- The three ML classifiers were fitted on the following features to check for best performance. These features are described in detail in the Data pre-processing section above. They were:
    - Text meta-features like word count, character count, noun count, etc.
    - Count based bag of words features
    - TF-IDF features based on words

  The '*train_and_compare_ML_models*' function described above was used for evaluating performance of each classifier on each these features.

  No text pre-processing (stopwords removal, punctuation removal, etc.) was done as it was leading to worse model performance (as explained in the Benchmark section). Also, we only used single n-grams for both count and TF-IDF based features as higher n-grams were giving worse performance (overfitting). A probable reason for worse performance of higher n-grams could be that they provide features that are very specific and might not be really observed on unseen corpus.

- The best feature and classifier combination was further fine-tuned using hyperparameter tuning by employing *GridSearchCV* function of sklearn
- Finally, a confusion matrix was plotted for the final tuned model

Stacked ensemble of ML models

Here the *mlxtend* package and specifically the *StackingCVClassifier* function was utilized for building the stacking ensemble classifier. This function uses cross-validation to prepare input for the combiner algorithm and hence prevents overfitting. Two different types of stacking ensembles were fitted:

- ***Stacking of multiple base learners on same features***: Here three different base learners were used (viz. SVC, XGBoost and Naive Bayes) and their predictions (i.e. probabilities) were combined using a combiner algorithm (viz. Logistic regression). The TF-IDF features were used for fitting the ensemble as they showed best performance on the individual ML learners
- ***Stacking of same base learner trained on multiple features***: Here the faster (yet performant) Naive Bayes algorithm was fitted on multiple features (viz. count/TF-IDF word/character features) and XGBoost was fitted on text meta-features. Then, the predicted probabilities of Naïve Bayes and XGBoost were used as first stage base learners and finally combined using XGBoost classifier.

Neural net models

The input features used for neural net model fitting were described in the Data pre-processing section above. Following steps were undertaken:

- Five different neural net models were explored:
    - LSTM
    - Bi-directional LSTM
    - GRU
    - Stacked GRU
    - CNN
- For each of the neural net models, following steps were conducted:
    - Defined the network architecture and training parameters. Few common settings across all architectures were:
        - Spatial dropout of 0.3 after the Embedding layer
        - A dense layer of 512 units with a dropout of 0.3 as the penultimate layer (before the softmax layer). The only exception was CNN model which used a smaller dense layer of 10 units
    - Defined the loss function and evaluation metric. Since we were anyways using *log loss* as our evaluation metric (which can be tracked by the model loss), we used *accuracy* as our tracking metric
    - Set a *ModelCheckpoint* object to save the network weights associated with minimum *log loss* during training
    - Padded integer sequences of same length were initialized with pre-trained *GloVe* embedding vectors. Additionally, the pre-trained embeddings were frozen to avoid overfitting during model training (as we had relatively small train corpus)
    - The padded integer sequences were input to a keras' Embedding layer and the model was trained while logging the loss and accuracy for each epoch. Few settings for model training were:
        - *validation* split of 20%
        - *optimizer*: adam
        - *loss metric*: categorical cross entropy
    - The model performance was then evaluated on the test set and the logged epoch loss/accuracy was plotted to examine model performance and check for overfitting
    - The process was repeated for different architectures until no significant improvement in performance could be obtained

FastText model

Following steps were performed for FastText implementation:

- Minimal data processing was done viz. lowercasing and punctuation removal
- Train dataset was saved as an external *txt* file in a format required by FastText i.e. complete observation in one line with labels (preceded by the literal *_label_*) before the sentence
- The *train_supervised* function from the *fasttext* package was used to fit the model directly on the external saved *txt* file
- The model was then tuned for different combinations of hyperparameters and the best model was obtained

Complications

Some of the additional complications were:

- The hyperparameter tuning of the SVC model took a long time as number of features were large. As a remedy, I had to port my code to a bigger machine and increase the number of parallel cores using the *njobs* parameter of *GridSearchCV* object
- The stacking ensemble showed best results but took >60 minutes to train for each run. Due to want of time, the ensemble model could not be tuned for best hyperparameters but it still holds great potential for performing even better
- The neural architectures involving LSTM and GRUs were exhibiting decent performance but were taking ~15 minutes for each run even on a GPU. Hence, doing hyperparameter optimization was not a possibility for the limited time I had. Hence, I had to rely on inspirations from online sources to build the final neural net architecture

Refinement

The refinements were done separately for the ML models, neural net models and FastText model.

For **ML models**, the single ML model that showed best performance was SVC classifier trained on TF-IDF features and had a log loss of 0.44 and accuracy of 0.82.

To improve upon this, hyperparameter tuning was performed using *GridSearchCV*. The parameters that were adjusted were:

- *C* (penalty parameter): exponential values were tried viz. [0.1, 1, 10, 100, 1000]
- *gamma* (kernel coefficient for rbf): exponential decreasing values were tried viz. [1, 0.1, 0.01, 0.001, 0.0001]
- *kernel* chosen was rbf only as it was showing consistently good performance

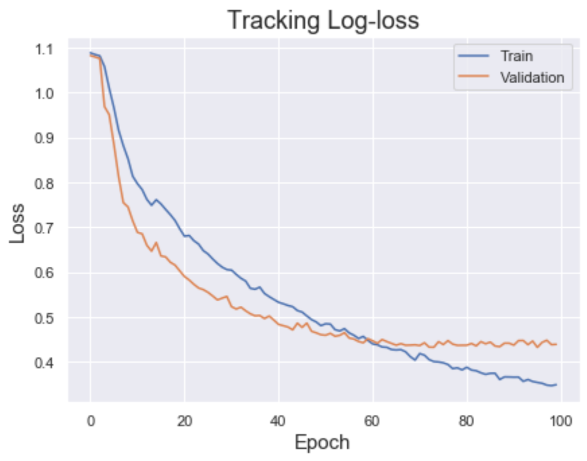
The tuned SVC model based on TF-IDF features then had the following performance:

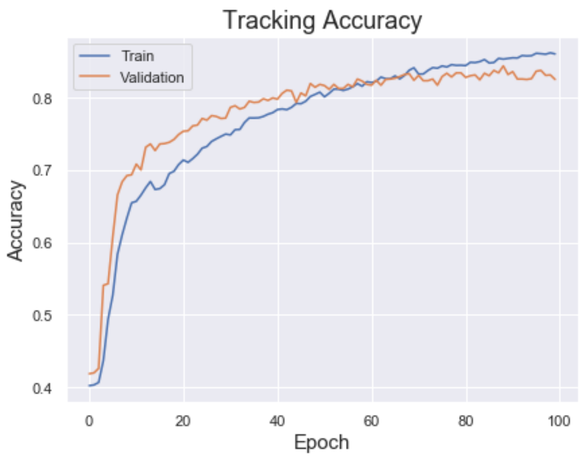
- Log loss: 0.42
- Accuracy: 0.84

Also, in general for ML models, better solutions were obtained by the following:

- Retaining stopwords and punctuation as removing them was worsening the performance
- Using only single-grams as introducing bi-grams and tri-grams was leading to overfitting (as explained in earlier sections)

For **neural net models**, the single layer GRU model gave a log loss of 0.47 and accuracy of 0.81. This was further refined by training in iterative fashion by plotting the train/validation loss/accuracy log like the below:





The final parameters were tuned manually (for reasons cited in complications section) and included:

- Changing dropout rate in layer
- Adding spatial dropout
- Changing the optimizer for model fitting
- Changing the batch size

The final model that performed best was a stacked GRU with two stacked GRU layers. Its performance was:

- Log loss: 0.45
- Accuracy: 0.82

However, this optimized neural net model was still below par compared to benchmark model.

For **FastText model**, the standard model using default parameters had a log loss of 0.50 and accuracy of 0.82.

To improve upon this, hyperparameter tuning was performed using an exhaustive grid search. The parameters that were adjusted were:

- *epochs*: [3, 5, 10, 20, 50]
- *learning rate*: [0.05, 0.1, 0.3, 0.7, 1]
- wordNgrams: [1, 2, 3]

The tuned FastText model then had the following performance:

- Log loss: 0.48
- Accuracy: 0.84

However, this optimized FastText performance was still below par compared to benchmark model.

IV. Results

Model Evaluation and Validation

The final model we chose was the **stacking ML ensemble with same learner but different features**. This model was chosen since it gave the best log loss (~0.37) and also the best accuracy (~0.86) on the holdout test set. Although, it takes some time to train (>60 minutes) but that is not our evaluation metric since we are mostly concerned with the best model performance. Also, the prediction on the test set (~2k observations) was done in under a minute and since our predictions are not time sensitive, this model would perform fine in production.

The highlights of the final stacking ensemble model chosen are:

- ***Features***: This model receives inputs from five sets of features, viz.:
    - Count features based on words
    - Count features based on characters

    - TF-IDF features based on words
    - TF-IDF features based on characters
    - Text meta-features like word count, noun count, etc.

  Following are additional details around word/character feature used as inputs to classifiers:
    - Word based features only had single n-grams as higher n-grams were leading to worse performance (overfitting). Also no text cleaning (stopword removal, punctuation removal, etc.) was performed as that was leading to worse performance (as discussed earlier)
    - Character based features had a n-gram range of 1 to 5 characters
- ***Classifiers***: Following classifiers were used:
    - Four different Naive Bayes classifier were used as the first stage classifiers for each of the word/character count/TF-IDF feature sets
    - XGBoost was used as the first stage classifier for text meta-features
    - Finally, XGBoost was again used as the combiner model for combining the prediction probabilities of the 5 base learners

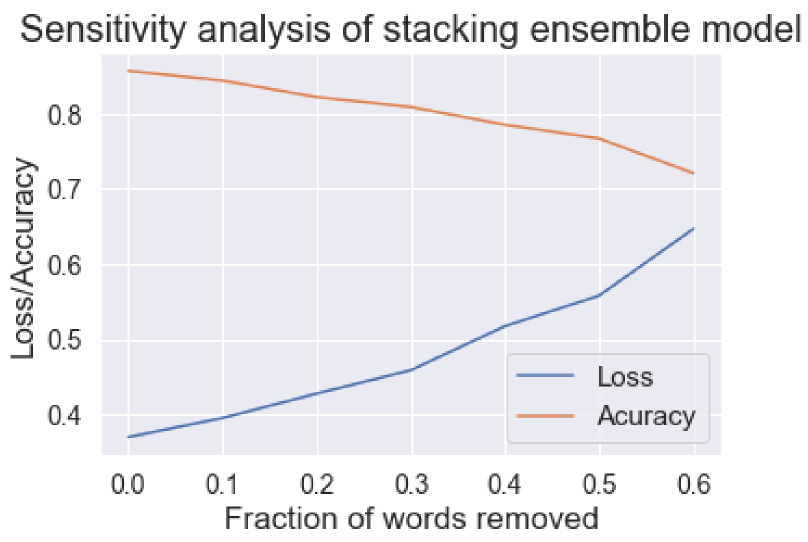
    All classifiers had default settings

We were not able to tune the stacking ensemble classifier as it takes significant amount of time to train for each run (>60 minutes). Nevertheless, the un-tuned classifier itself produces best in class performance on our data. With some more time at hand, the model can be further tuned and even better accuracies can be obtained.

The power of this model is that it finds a way to marry multiple features which are very different and cannot be used by a normal ML model. For e.g. sparse word/character features are high in number and are different from dense text meta-features which are few in number. The stacking ensemble model can deal with all these heterogeneous sets of features with ease.

Further to check the **robustness of the final chosen** model, a sensitivity analysis was conducted. The idea was to randomly remove a fraction of words from each sentence in the test data and check its effect on model performance. If the model is robust, it's performance should not be greatly affected by slight changes in the test data. Below table and plot shows the results.

Fraction of words removed	Loss	Accuracy
0%	0.370	0.856
10%	0.395	0.844
20%	0.428	0.822
30%	0.459	0.808
40%	0.517	0.785
50%	0.558	0.767
60%	0.647	0.721



We observe the following from the above table and plot:

- Model log loss still beats the benchmark model (log loss ~ 0.43) even on removing 20% of words in each sentence
- Model performance only starts to significantly degrade on removing 40% of words (log loss ~ 0.52) or more

Hence, we conclude that the stacking ensemble model chosen is **robust** under small variations in the input data.

Justification

Below is a comparison of some of best models obtained during the project in descending order of performance i.e. best to worst (the final model chosen and benchmark model have been highlighted).

Model Name (best to worst)	**Log Loss**	**Accuracy**
Stacked ML ensemble (same learner different features)	**0.37**	**0.86**
Stacked ML ensemble (multiple learners same features)	0.39	0.84
SVC TF-IDF model (tuned)	0.42	0.84
Benchmark model (Logistic regression on count features)	**0.44**	**0.83**
Stacked GRU	0.45	0.82
FastText model (tuned)	0.48	0.84

It can be clearly seen that the final chosen model (stacking ensemble with same learner different features) has the best log loss (~**0.37**) and the best accuracy (~**0.86**) and significantly beats the benchmark model with a log loss and accuracy of ~0.44 and

~0.83 respectively. Although the bump up in accuracy is not great (+3%) but the improvement in log loss (-7%) is still significant – since our evaluation metric is log loss, our objective was not to optimize for accuracy.

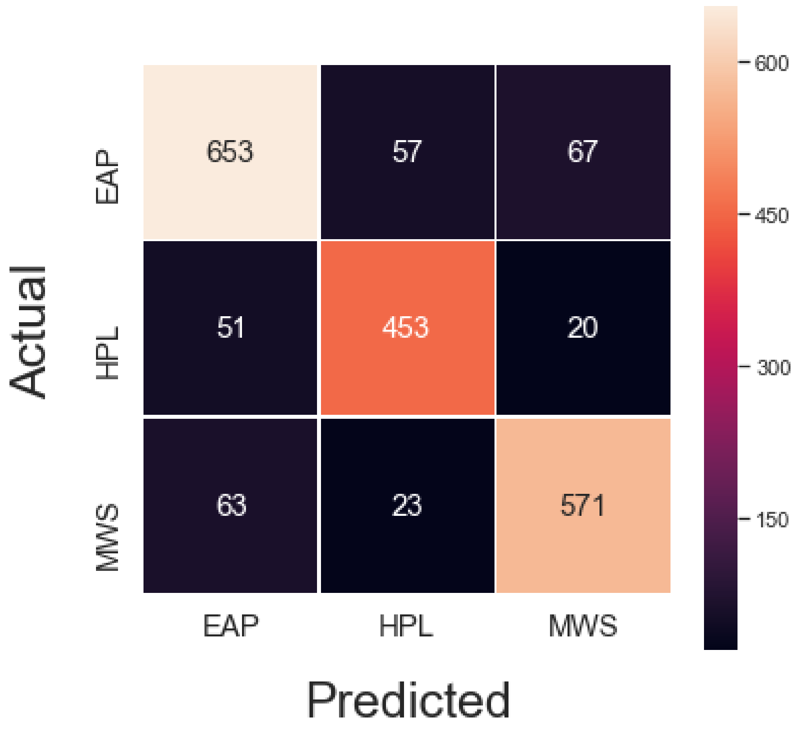
Moreover, as can be seen in the Model Evaluation section, the stacking ensemble model is robust to changes in input conditions.

On the flip side, as the model starts to degrade on removing 40% of words or more, hence the final solution only works in the limited domain in which it is trained. If in the future, the writing styles of authors significantly change or if texts of new authors are desired to be predicted, the model may have limited utility.

V. Conclusion

Free-Form Visualization

Below, we plot the confusion matrix of the final solution i.e. stacked ensemble model:



We can observe the following from the above confusion matrix:

- Most of the sentence counts are along the diagonal which means that author for most sentences are correctly predicted
- There are also some sentences that are getting wrongly attributed to false authors. Further research can be done in order to search for any fundamental differences in these wrongly classified sentences. If found, then corresponding adjustment can be made to the features or to the classifiers in order to obtain even better performance.

Reflection

The process used for this project can be summarized into the following steps:

1. An initial problem and relevant, public dataset was found from kaggle
2. The data was downloaded and exploratory analysis was conducted to gain early insights on the data
3. The textual data was then preprocessed and feature engineering was done to convert textual features to numerical features that can be fitted to various ML classifiers
4. Various ML/DL classifiers were fit on the obtained features and their performance was evaluated on the holdout test set. The classifiers belonged to multiple classes viz. ML based, stacking ensembles, neural net based and character based (FastText)
5. Tuning was performed on promising models although some of the best models (like stacked ensembles) could not be tuned due to want of time (as they take long time to train)
6. The final model having best log loss and accuracy was then selected
7. Sensitivity analysis was also conducted on the final solution in order to establish the robustness of the chosen model on small variations in input data

I found the 4th step to be the most difficult and time consuming. Since textual data has large number of features, most models typically took large amount of time. Hence, I had to optimize my runs so that I can get to best models in limited number of runs. Still, I could not optimize some of the models which took a lot of time to train for even a single run.

On the interesting aspects of the project, everything was interesting! In the ML nanodegree, NLP was not covered specifically. Although, the nanodegree did a great job in explaining the fundamentals around some of the most used ML and DL implementations, the nuances of NLP are very different than other domains. Since NLP is something that interests me more than any other domain, I had to read through a lot of concepts in detail and watch course lectures from top universities to get a strong understanding of the same. In the process, I learnt new types of methods for feature engineering and models, and ventured into a completely new domain. In the end, I am glad that it was a great knowledge enhancement experience for me!

On the general applicability of the model, as discussed in the previous sections, the model is trained on the works of specific authors in horror domain. This model as such may not generalize to other domains or other authors. In the limited context in which this model is trained, it performs well.

Improvement

To improve the model performance further, the following can be done:

- More capable hardware can be used which can run models faster and enable tuning of hyperparameters faster. We could also employ techniques that tune hyperparameters more efficiently like Bayesian methods. This would enable tuning of models that could not be tuned in this project
- Contemporary language models like BERT can be employed which have shown state of the art performance on many textual tasks. Although, I mentioned in the proposal that I might try this out if I had time, but could not. In the future, this model can be used on the data at hand to try to bump up the performance even further
- Transfer learning can be further used as that has provided significant improvement on many NLP tasks. We did use transfer learning in limited context by making use of pre-trained GloVe embeddings but we can make use of more recent and better embeddings like ELMO, etc.

In general, if we were to make the final model as the benchmark model, then most certainly a better model exists that can still beat this model using some combination of the above steps!

References

[1] https://en.wikipedia.org/wiki/Natural_language_processing

[2] https://www.forbes.com/sites/forbestechcouncil/2018/11/06/the-evolution-of-natural-language-processing-and-its-impact-on-ai/#131b51b51119

[3] https://www.kaggle.com/c/spooky-author-identification/data

[4] https://www.cs.cornell.edu/people/tj/publications/joachims_98a.pdf

[5] http://datascience.la/benchmarking-random-forest-implementations/

[6] https://news.ycombinator.com/item?id=18769363

[7] https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[8] https://en.wikipedia.org/wiki/Convolutional_neural_network