Marifel Barbasa
Thursday, August 30, 2018
MLND March 2018 Class

Identifying Horror Authors from Samples of their Writings

Machine Learning Capstone Report
Machine Learning Engineer Nanodegree, Udacity

I. Definition

Project Overview

Authorship attribution or identification is the problem of identifying who wrote a piece of text. Determining the author of a body of written work is an important task in natural language processing (NLP), information retrieval, and computational linguistics, as doing so correctly can help resolve forensic investigation cases, uncover plagiarists, give proper attribution to historical or ancient works, provide readers with recommendations of authors with similar writing styles, and determine anonymous authors or those who pose under a pseudonym. Consequently, solutions to authorship attribution help fight crimes and make it more difficult for people with malicious intent to distribute written content, such as those with the aim of posting fake news under a pseudonym or multiple pseudonyms.

In this project, we will apply NLP and machine learning techniques on the Spooky Author Identification competition dataset hosted by Kaggle[1] in order to tackle the authorship attribution problem. Specifically, our goal is to identify the horror authors Edgar Allan Poe (EAP), H.P. Lovecraft (HPL), and Mary Wollstonecraft Shelley (MWS) from samples of their writings. The dataset calls for a multiclass classification solution to the problem, in which each of the three authors is a target class and the only input is the textual excerpts of their fiction writings. The task is to assign probabilities to EAP, MWS, and HPL, given a sentence from one of their works; in other words, the task is to determine how likely it is that each of the three authors wrote a particular sample.

While the data is real in that the authors truly wrote these pieces, the dataset has been handcrafted. Kaggle has prepared and labeled this dataset of 19,579 labeled samples, where each sample is a sentence drawn from a written work of the author. Kaggle states that, because the "data was prepared by chunking larger texts into sentences using CoreNLP's MaxEnt sentence tokenizer," occasional incomplete sentences may appear in the samples[2].

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

Problem Statement

Given labeled samples of their writings, this project aims to predict the probability that each author in the Spooky Author Identification corpus wrote a piece of text that the machine learning model has not yet seen. This multiclass classification task is quantifiable because it seeks to solve the authorship attribution problem by expecting an output probability distribution across authors per unseen writing sample. The probability predicted per author translates to how likely the author is to have written the sample, and the probabilities predicted per sample should approximately sum to 1. The problem is also measurable because, given a number of unseen samples with the actual labels set aside, the predictions on those unseen samples can be measured using multiclass logarithmic loss or logloss. And the lower the logloss, the better the model is at making accurate predictions. The problem is also replicable because, given the same set of data, one can reproduce the experiments.

In this project, we will employ the typical approach to working with text data, which involves five steps: exploratory data analysis (EDA), text preprocessing, feature extraction, modeling, and refinement. With EDA, we will visually skim the data to see what we're working with and analyze count features such as basic statistics over all the text samples and those per author.

With text preprocessing, we will attempt various cleaning tasks and evaluate what combination works best. These tasks are: lowercasing, removing punctuation, removing stopwords, stemming, lemmatization, and converting British English to American English spelling. After the text is cleaned, we'll tokenize the text into words.

For feature extraction, we will try pre-trained word embeddings and attempt a bag of words with term frequency-inverse document frequency (TF-IDF) scoring. With pre-trained word embeddings, we shall experiment with Stanford GloVe[3] and Facebook fastText[4] embeddings, evaluating what works best.

Once the text has been converted to numerical input vectors, we will construct the following models using Keras during the modeling phase: 1-dimensional (1D) convolutional neural networks (CNNs), recurrent neural networks (RNNs) with either a long short-term memory (LSTM) or gated recurrent unit (GRU) layer, and simple multilayer perceptrons (MLPs). The CNN and RNN models will take the pre-trained word embeddings as input, while the MLP models will take the bag of words as input.

[3] https://nlp.stanford.edu/projects/glove/
[4] https://fasttext.cc/docs/en/english-vectors.html

Each model will be evaluated based on the validation logloss metric using 10-fold cross validation; the lower the logloss, the better the model. To further validate the model, we will make submissions to Kaggle's Spooky Author Identification competition in order to achieve the mean leaderboard (LB) logloss, since the test dataset that Kaggle provides is unlabeled (so that submission is the only means of achieving a score). Apart from manual evaluation, we will run random search to tune certain parameters and hyperparameters for each model for 60 iterations in order to find the best model for this multiclass classification problem of authorship attribution.

Each of the six models that will be tuned is as follows: a CNN with GloVe embeddings, an RNN with GloVe embeddings, a CNN with fastText embeddings, an RNN with fastText embeddings, an MLP with 20,000 unigram and bigram features scored using TF-IDF, and an MLP with all unigram and bigram features scored using TF-IDF. The MLP settings (of 20,000 features as a limit, using unigrams and bigrams, using TF-IDF scoring, and using MLPs) are based on Google's recently released Text Classification guide and are expected to perform rather well for a dataset whose ratio of number of samples to number of words per sample is smaller than 1,500[5], which describes our Spooky Author dataset. Therefore, the expected solution is that the simple MLP models will perform better than the CNN and RNN models, and that all six models will perform better than the benchmark, which is simply guessing based on the probability distribution of samples per author.

Metrics

An evaluation metric that can quantify the performance of the benchmark and solution models is the multiclass logarithmic loss, or logloss, which is well-suited for multiclass classification problems. The logloss is defined as

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

where N is the number of samples, M is the number of classes (there are 3), *log* is the natural logarithm, y_{ij} is 1 if sample i belongs to class j (else 0), and p_{ij} is the predicted probability that sample i belongs to class j[6]. In short, the logloss is the negative of the mean log-likelihood over the number of samples. This means that, for each test sample, select the model's output predicted probability, p, for the true class and take its logarithm. Then add all of these values together and divide by the number of test samples; lastly, negate this value to obtain the logloss for the model. Note that, despite the model outputting a probability for each author per test sample, only the true author's probability will be included in the logloss score.

[5] https://developers.google.com/machine-learning/guides/text-classification/step-2-5

[6] https://www.kaggle.com/c/spooky-author-identification#evaluation

While upon submission to Kaggle, predicted probabilities, `p`, are replaced with `max(min(p, 1 - 10`$^{-15}$`), 10`$^{-15}$`)` in order to avoid the extremes of the log function, we will be using the `categorical_crossentropy` loss function from Keras to calculate the logloss for us, which has its own threshold value for log extremities. We will also be monitoring the accuracy using Keras but will not use it for weight updates or modeling and refinement decisions; accuracy will be monitored purely as a convenience and reference. One important distinction to note is that we will compare models based on *validation* logloss/loss (`val_loss`) and not logloss/loss (`loss`), which stands for the training loss in Keras. This is because loss alone is not a good indicator of model performance; a very low loss could mean that the model is overfitting to training data. A low validation loss is what we need to aim for, since it is a more likely indicator of the model's performance on unseen data.

The logloss works especially well for multiclass classification problems because it aims to quantify accuracy by heavily penalizing false classifications[7]. Imagine a logarithmic function flipped vertically over the x-axis (since we are dealing with the negative log), where the logloss is mapped to the y-axis and the probability of a class is mapped to the x-axis: we note that as the probability increases, the more gradually the logloss decreases, and as the probability decreases, the sharper the logloss increases. Therefore, the logloss seeks to heavily punish low probabilities (incorrect classifications, since only the probabilities for the true class is included in the score) and neutralize (bring as close to 0) high (correctly classified) probabilities.

II. Analysis

Data Exploration

The Kaggle Spooky Author Identification dataset consists of 19,579 labeled training samples and 8,392 unlabeled test samples. The training set contains the features `id` and `text` along with the target `author` column. The test set contains the features `id` and `text` but lacks the target `author` column. The `id` feature is a unique identifier consisting of the string "id" followed by five digits (such as "id25712") and is irrelevant to the task at hand, since we'll be treating this dataset as an NLP problem. The `text` feature is our main focus and consists of a single sentence per sample. Lastly, the `author` column in the training set contains one of the following values: "EAP", "HPL", or "MWS".

Skimming the text data reveals that there are some uncommon words such as "pursy," made-up words such as "drest" (meaning "dressed"), words with multiple spellings such as "until" vs. "untill," and misspelled/accented words such as those used by HPL: "It was all mud

[7] https://datawookie.netlify.com/blog/2015/12/making-sense-of-logarithmic-loss/

an' water, an' the sky was dark, an' the rain was wipin' aout all tracks abaout as fast as could be." There are also mentions of places like "Windsor" and names like "Dr. Johnson." UK English is oftentimes (but not always) used, such as "travellers" or "neighbours." Moreover, EAP sometimes writes in French: "que tout notre raisonnement se rèduit à céder au sentiment." There do not appear to be any numbers; only common punctuation like periods, commas, colons, semicolons, apostrophes or single quotes, double quotes, and question marks are used, and all caps are rare occurrences. Therefore, the data seems relatively clean.

A sample sentence written by EAP is, "But a glance will show the fallacy of this idea." One written by HPL is, "Dr. Johnson, as I beheld him, was a full, pursy Man, very ill drest, and of slovenly Aspect." Another written by MWS is, "They fly quickly over the snow in their sledges; the motion is pleasant, and, in my opinion, far more agreeable than that of an English stagecoach." From these three samples alone, it is difficult to tell the difference between each author's writing styles. We calculated the sample distribution per author, out of 19,579 labeled samples, as follows: 7,900 EAP samples (40.35%), 5,635 HPL samples (28.78%), and 6,044 MWS samples (30.87%).

Of the 19,579 training samples, the mean word length of a sentence is 27 with a standard deviation of 19 words. The minimum number of words in a sentence is 2, 25% of sentences have 15 or fewer words, 50% of sentences have 23 or fewer words, 75% of sentences have 34 or fewer words, and the maximum number of words in a sentence is 861. Without any text preprocessing (obtaining tokens simply by using a single space ' '), the vocabulary size of the text per author is 26,521 for EAP, 22,527 for HPL, and 20,251 for MWS, with a total vocabulary size of 47,556 between all three authors. After removing stopwords and punctuation, the vocabulary size of the text per author is 15,276 for EAP, 14,576 for HPL, and 11,458 for MWS, with a total vocabulary size of 25,275 between all three authors.

Since we're working with text data from the sole `text` feature column, we need to apply NLP techniques such as a bag of words or word embeddings to extract features from the text before feeding them into a machine learning model. Furthermore, the target `author` string column needs to be transformed into integer indices and then into one-hot encoded vectors representing those integers.

For this project, an important metric that we need to calculate is the ratio of the number of samples to the number of words per sample; based on Google's Text Classification guide[8], this metric will determine whether we should go with a bag-of-words model or use word embeddings. If this ratio is less than 1,500, then Google suggests using a bag-of-words MLP

[8] https://developers.google.com/machine-learning/guides/text-classification/step-2-5

model; otherwise, use word embeddings. We calculated this ratio to be about 732 for our dataset, using the mean word length of about 27 for the number of words per sample. Since this ratio is less than 1,500, per Google, we should use n-grams as input into an MLP. Nevertheless, we will experiment with both n-grams and word embeddings.

Exploratory Visualization

Since we'll be feeding n-grams into an MLP model, it might be helpful to view the most common n-grams per author. Which ones are highly predictive of one author over another? And which ones are commonly used by more than one author? While we're not certain at this point if keeping stopwords and punctuation will help or not, for the purposes of these visualizations, let's remove them and see what actual words might be meaningful or predictive.

Figures 1 and **2** below show the 15 most common word-level unigrams and bigrams per author. We can see that all three authors are fond of the unigrams "one," "could," "would," "time," and "man." We can also see that, more than any other unigram or bigram, EAP is extremely fond of "upon" and "let us," whereas HPL extensively uses the phrase "old man." And all three authors often talk about an "old man." It seems that, apart from "old man," "madame lalande" and "mr crab" are important characters in EAP's works, "old woman" and "herbert west" are notable characters in HPL's works, and "lord raymond" and "earl windsor" are important in the works of MWS. Due to these interesting findings and that each author has a distinctive usage of vocabulary, it appears that unigrams and bigrams will prove to be useful features with fairly decent predictive power.

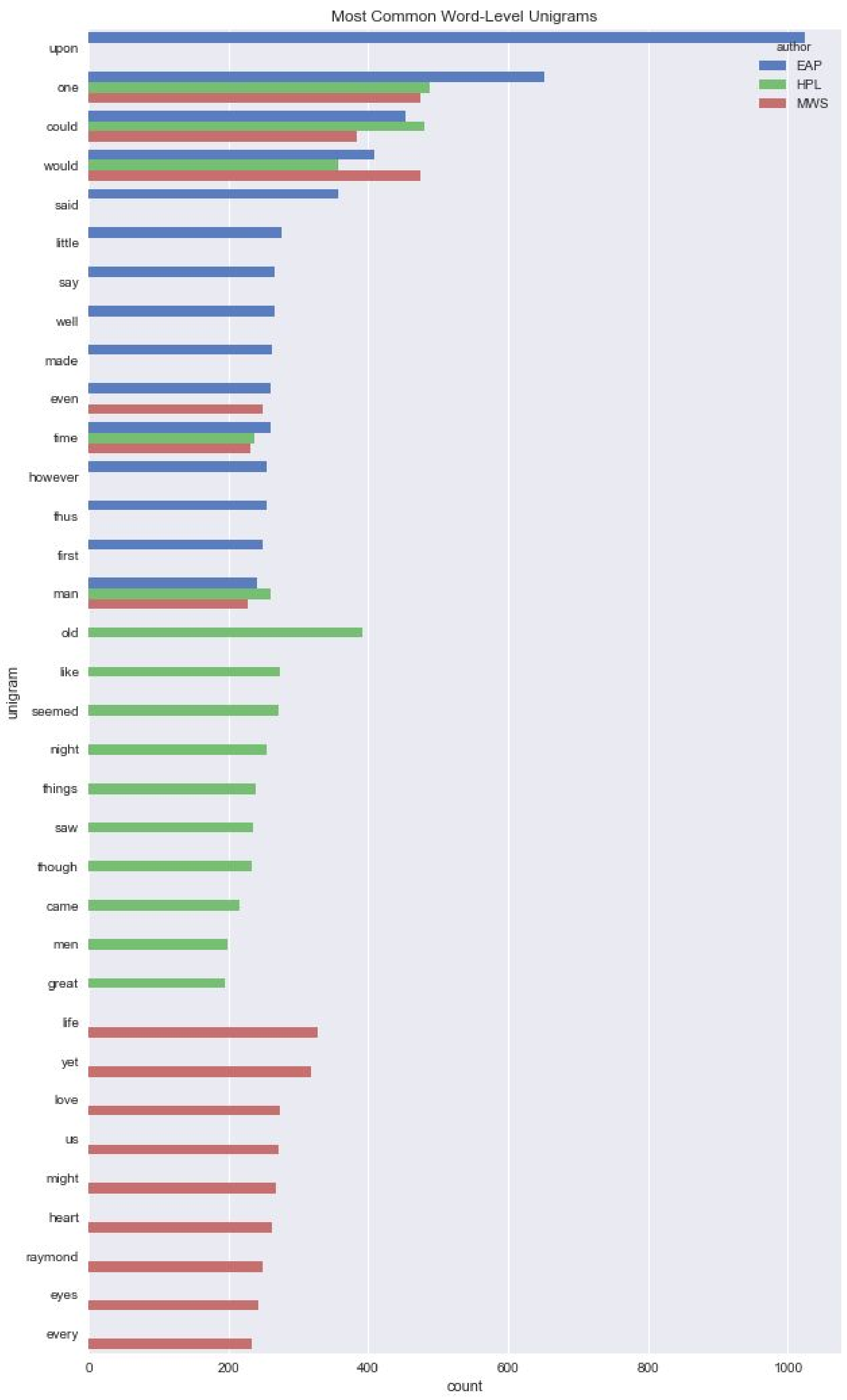

Figure 1: The 15 most common word-level unigrams per author.

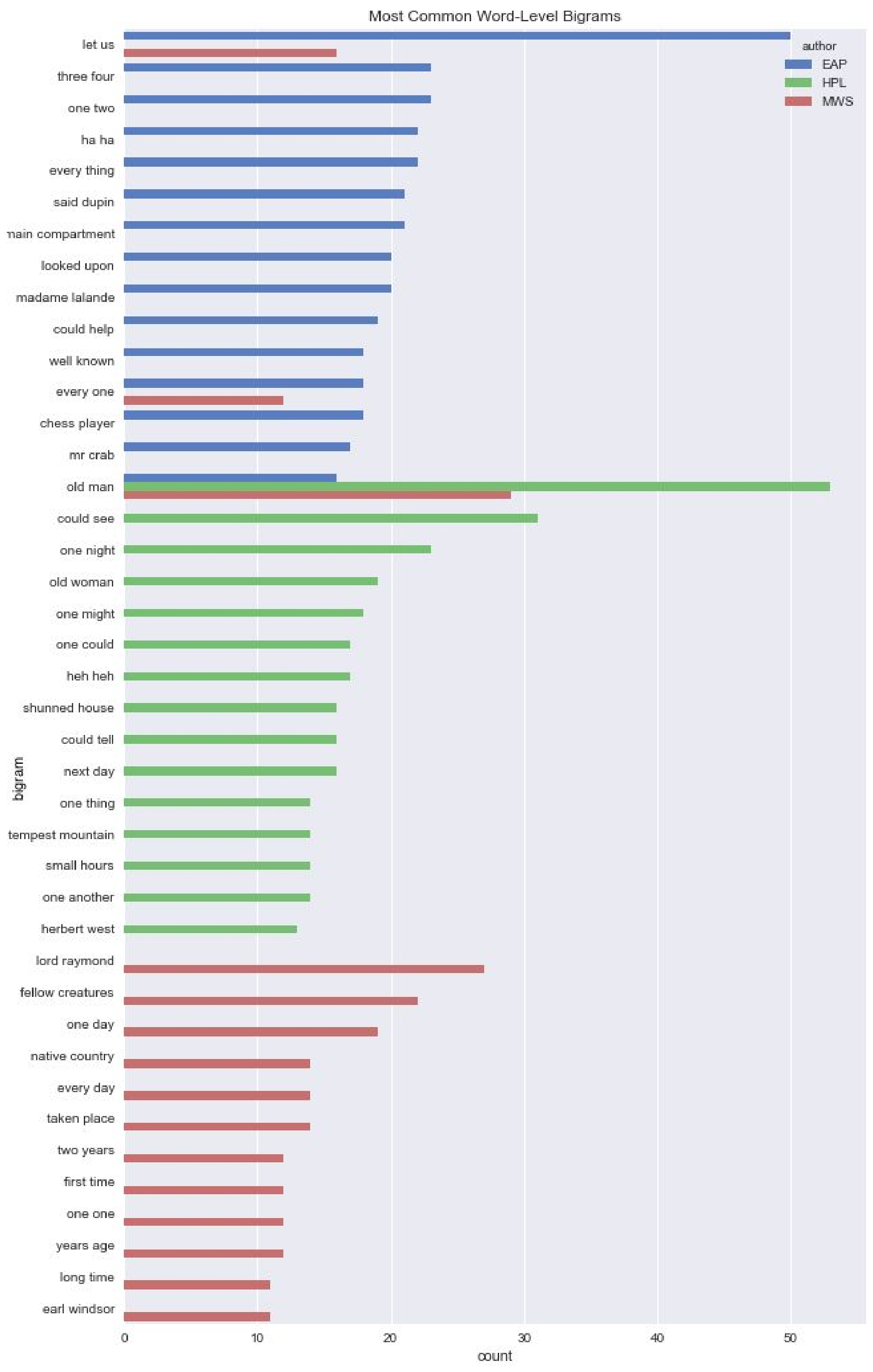

Figure 2: The 15 most common word-level bigrams per author.

Algorithms and Techniques

In this project, we'll apply two NLP feature extraction techniques on the Spooky Author Identification text data: word embeddings and a bag of words. Words can be represented by word embeddings, which are vectors of fixed dimensionality, such as 25, 50, 100, 200, or 300. These vectors form an embedding space in which words that are similar to each other tend to be grouped closer to one another than dissimilar words. This differs from the bag-of-words approach in that a bag of words does not preserve context and word order. A bag of words is simply a "bag" or collection of words with a random but unique integer index assigned to represent each word. In this way, when vectorizing sentences, for example, each vector is as long as the length of the vocabulary and the word indices are mapped to the positions within the vector. A word can then be marked as present in the sentence with a 1 and 0 otherwise; this is called a binary count, which is a bag-of-words scoring method. Already, we can see that the bag-of-words approach can lead to large sparse matrices, which are matrices with mostly 0 elements, since vocabularies in decently sized corpora, such as our dataset, can range in the thousands of words. Conveniently, scikit-learn efficiently handles sparse matrices using `scipy.sparse.csr_matrix` to store the vectorized text[9].

Scoring methods for a bag of words include the following: binary, count, and term frequency-inverse document frequency (TF-IDF). Binary, as discussed, marks whether or not a word is present in the sentence with a 1 at the position (word index) of the word. This means that if a word occurs five times in a sentence, the sentence vector will contain just a 1 at its word index. Count scoring, instead, would mark that word index with a 5. For both binary and count, if a word is not present in a sentence, a 0 marks its place in the sentence vector. Counting, however, is not always that informative, since common words such as "the" and "a" can appear multiple times within most sentences and still be regarded as the most important words due to count alone. A more effective measure of word importance for a document (sentence, in our example) is TF-IDF scoring. Term frequency (TF) is the count of a word in a document (essentially the count score itself), while inverse document frequency (IDF) is the natural logarithm of the total number of documents in the corpus over the count of documents that the word appears in (1 may be added to the numerator and denominator to avoid the natural logarithm of 0, which is undefined); TF and IDF are multiplied and normalized to achieve the TF-IDF score. This rewards a word that appears more frequently in a single document with a higher decimal value but lowers that value if the word becomes too frequent across multiple documents within the corpus. In this way, we know what words are actually meaningful for each document.

[9] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

We will employ the bag-of-words approach using TF-IDF scoring in this project, but we will use bigrams (2-grams) as well, not just unigrams (1-grams). Unigrams are simply all words such as “the,” “cat,” “in,” “the,” and “hat” in “the cat in the hat,” whereas bigrams are every pair of words in sequence order, such as “the cat,” “cat in,” “in the,” and “the hat.” Trigrams (3-grams) are every three sets of words in sequence order, such as “the cat in,” “cat in the,” and “in the hat.” This can be extended to 4-grams, 5-grams, and so forth, and the general term is n-grams. While the n-grams used in the examples are word sequences, n-grams mean any linguistic sequence of components in speech or text, such as characters, syllables, and words. In the project, however, we will only use word-level n-grams; these word-level unigrams and bigrams will then serve as our “word” tokens in the bag of words, and each unique n-gram will be assigned a random unique integer index.

Regarding word embeddings, instead of training our own vector space representations of words from scratch, we will use pre-trained word embeddings, in particular, Stanford’s GloVe embeddings and Facebook’s fastText embeddings. I chose these embeddings because they are the standard and commonly used in NLP research papers and Kaggle competitions. There are a number of pre-trained vectors to choose from, trained on different corpora such as Wikipedia, Twitter, or the Common Crawl. These corpora are much larger, containing billions of tokens, compared to most datasets, such as our dataset for this project. The benefit of using pre-trained embeddings is that we don’t need to train on the same massive dataset to learn useful word embeddings, therefore wasting time and computational resources. This is also beneficial for our dataset in that words that appear only once in the training dataset can still be predictive, since their equivalent vector representations can still be found in the pre-trained embeddings and clustered in vector space with similar words that an author uses.

After each sentence in our dataset has been preprocessed and vectorized into numerical format using word embeddings or a bag of words, we can then feed it into a machine learning algorithm. The machine learning algorithms we will employ in this project are neural networks. A basic neural network is as follows: A number of input neurons (essentially nodes) contain values, each value is multiplied by its weight, and the resulting values are then summed together, to which a nonlinear function is then applied in order to achieve the final output value. Multiple neurons can then be placed together side by side as one layer, taking in the same inputs, and then another layer of neurons can be added as the first layer’s output, taking in as input the output values of the first layer. Further layers of neurons can be added; the first layer is called the input layer, while the last layer is called the output layer, and all the layers in between are called hidden layers (a neural network with multiple hidden layers can be considered a deep neural network). Neural networks in which data flows in one forward direction and all nodes in one layer are connected to every node in the next layer (a fully connected layer), and so forth, are called feedforward neural networks. In this project, we will train the multilayer perceptron (MLP) neural network architecture, which is simply a feedforward neural network with one or more hidden layers. In particular, we will use MLPs with bag-of-word features as input.

Before neural networks can predict a probability distribution or the target classes of "EAP," "HPL," and "MWS," we need to one-hot encode the classes, since neural networks can only output numbers. One-hot encoding transforms a string or number into a sparse vector representation by first taking all the values, assigning each one to a column in the vector, and then marking a 1 in the corresponding column for that value (marking 0 for all other columns). For this project, we will first transform EAP, HPL, and MWS to the numerical labels 0, 1, and 2, respectively. Then we will one-hot encode these labels into the vectors [1 0 0] for EAP (label 0), [0 1 0] for HPL (label 1), and [0 0 1] for MWS (label 2) before feeding them into our neural network models. The reason we want to one-hot encode values as opposed to just assigning a number to them is that machine learning algorithms can assign more or less weight to different numbers, such as "yellow" having higher importance than "green" if "yellow" were 1 and "green" were 0. But this is not necessarily the case, so to prevent this, we can use one-hot encoding so that "yellow" and "green" can weigh similarly.

While we will train simple MLPs with one or two hidden layers for the bag-of-words approach, we need to use a different category of neural network architectures in order to use word embeddings as input. This category involves neural networks that are made for discovering features in sequential or contextual data and includes the convolutional neural network (CNN) and the recurrent neural network (RNN). CNNs are extremely effective at discovering hierarchical features in image data, such as the earlier layers detecting edges, the middle layers detecting shapes, and the later layers detecting faces in a face detection image dataset. A CNN consists of one or more convolutional layers and usually consists of pooling layers along with fully connected layers. A convolutional layer learns local patterns in the input feature space, whereas a fully connected layer learns global patterns[10]. For example, imagine a 16-by-16-pixel image of the number 3: While a fully connected layer would learn patterns involving all the pixels in the entire image, a convolutional layer would learn patterns based on every contiguous window, such as all the 3-by-3-pixel windows that can be made if one were to slide that window across the full image, from left to right, top to bottom, producing feature maps. CNN patterns learned are therefore translation invariant (for example, a pattern learned in the top-right corner of the image can be detected anywhere, such as in the bottom-left) and may form spatial hierarchies (the edges detected in earlier layers can be used to learn larger

[10] *Deep Learning with Python* by Francois Chollet, page 122

patterns in later layers)[11]. Pooling layers are layers that downsample feature maps by taking the average or maximum of the values in each window[12].

Apart from feeding GloVe and fastText embeddings into CNN models, we will feed them into RNN models as well. An RNN consists of RNN layers and may consist of fully connected layers. An RNN layer consists of RNN cells, which have a loop, where each iteration through the loop is a timestep that passes and the output of the loop is fed back into the loop as input. RNNs are good at remembering the recent past and contain memory elements, unlike fully connected layers. RNNs are not as effective on their own in practice and can't remember many timesteps before, due to the vanishing gradient problem, in which neural networks that are many layers deep become virtually untrainable[13] since the gradient becomes too small. Long short-term memory (LSTM) and gated recurrent unit (GRU) layers were invented to help combat the vanishing gradient problem. An LSTM cell consists of a cell state along with three gates used to control the flow of information to or from that state: an input gate, output gate, and forget gate. Using these gates, an LSTM can choose what to learn, remember, and forget about past information, even if it happened many timesteps ago. A GRU cell is a less computationally complex variant of LSTM (and thus generally runs faster, especially on smaller datasets) in that it has two gates instead of three, since the input and forget gates of the LSTM are combined into a single update gate for the GRU, along with other changes.

Benchmark

The distribution of target classes, of samples per author, on the Spooky Author Identification dataset will be used as a basis for random guessing. Kaggle has already provided that benchmark in the `input/sample_submission.csv` file, and the probability per author (same probabilities across all samples, so that EAP is always about 0.40, HPL is always about 0.29, and MWS is always about 0.31) exactly matches the percentage of samples that each author has in the total training dataset of 19,579 samples (7,900 EAP samples, 5,635 HPL samples, and 6,044 MWS samples). The benchmark's mean leaderboard logloss to beat is 1.08825.

III. Methodology

Data Preprocessing

There are multiple factors to consider when working with text data, such as whether to tokenize text into words or characters, whether to use a bag of words or word embeddings as features, the choice of embeddings and dimensionality of the word vectors, the maximum

[11] *Deep Learning with Python* by Francois Chollet, page 123
[12] *Deep Learning with Python* by Francois Chollet, page 127
[13] *Deep Learning with Python* by Francois Chollet, page 202

number of features to include in the model (number of n-grams or number of word embeddings), the maximum sequence length to be fed into the model, and the modeling itself. In addition, when constructing a bag of words, one must decide on the n-gram range (unigrams, bigrams, trigrams, 4-grams, etc.) and the count mode, such as binary, count, or TF-IDF, and with word embedding features, one must consider whether to learn them from scratch or use pre-trained embeddings, which can be fine-tuned or frozen. From the proposal stage, I had already decided to try both a bag of words and pre-trained word embeddings as features. Then at the project start, I chose to always tokenize text into words throughout the project.

In order to make these decisions for the word embedding features, while not exhaustive, quite a bit of testing was performed for this project, while other decisions were made without experimentation. To start, I chose Stanford's GloVe embeddings and Facebook's fastText embeddings because, as discussed, they are the standard and commonly used in NLP research papers and Kaggle competitions. Since both GloVe and fastText each have a small number of pre-trained embeddings to choose from, I ran 10-fold cross validation on each of the following GloVe embeddings: 400,000 300-dimensional (300d) vectors of uncased tokens (words) trained on 6 billion tokens of the Wikipedia 2014 and English Gigaword Fifth Edition corpora (filename `glove.6B.300d.txt`), about 1.9 million 300d vectors of uncased tokens trained on 42 billion tokens of the Common Crawl corpus (`glove.42B.300d.txt`), and about 2.2 million 300d vectors of cased tokens trained on 840 billion tokens of the Common Crawl corpus (`glove.840B.300d.txt`). I also ran 10-fold cross validation on each of the following fastText embeddings: about 1 million 300d vectors trained on 16 billion tokens of the Wikipedia 2017, UMBC WebBase, and statmt.org news corpora (`wiki-news-300d-1M.vec`), about 1 million 300d vectors trained with subword information on 16 billion tokens of the Wikipedia 2017, UMBC WebBase, and statmt.org news corpora (`wiki-news-300d-1M-subword.vec`), and about 2 million 300d vectors of cased tokens trained on 600 billion tokens of the Common Crawl corpus (`crawl-300d-2M.vec`). The embedding test results can be found in the file `results/embeddings_test.txt`. The embeddings that scored the best (lowest) with validation loss 0.78022 for GloVe and 0.71973 for fastText were `glove.840B.300d.txt` and `crawl-300d-2M.vec`. Note that I skipped testing vectors lower than 300d because I wanted to work with vectors that had captured the most meaning possible (thus higher dimensionality vectors) and thought it would make for a fairer comparison between GloVe and fastText, since fastText only has 300d vectors available. In addition, I skipped the GloVe Twitter embeddings because I felt that Twitter would be a worse representation of 18th, 19th, and early 20th century writing than either Wikipedia or the Common Crawl. Furthermore, I decided only to work with frozen pre-trained embeddings for comparison, since learning embeddings from scratch or fine-tuning pre-trained embeddings would add a lot more to the already full plate of experimentation. In the end, I did not have time to explore learning embeddings from scratch or fine-tuning pre-trained embeddings.

For the pre-trained embeddings, the maximum number of features and maximum sequence length were also decided upon based on experimentation. Using the `glove.840B.300d.txt` embeddings, I ran 10-fold cross validation for each of 9,000, 12,000, 15,000, 18,000, and `None` (meaning no maximum limit) for the maximum number of features and concluded that setting no maximum limit (using all features) performed the best, scoring the lowest validation loss scores; see `results/max_features_test.txt` for detailed results. Using the same GloVe embeddings, I ran 10-fold cross validation for each of 23, 27, 34, 128, 256, 512, 861, and 900 for the maximum sequence length and concluded that a length of 128 performed the best; see `results/max_sequence_length_test.txt` for detailed results.

As for the bag-of-word features, decisions were made following Google's advice in their Text Classification guide[14] and flowchart. Therefore, I chose to use both unigrams and bigrams for the n-gram range and a TF-IDF count mode. Nevertheless, I performed a 10-fold cross validation on an n-gram range of 1 to 3 (so that each unigram, bigram, and trigram is a feature) and concluded that an n-gram range of 1 to 2 performed better, scoring a lower validation loss. The maximum number of features (unigrams and bigrams combined) was set to 20,000 as recommended by Google; regardless, I also experimented with setting no limit on the number of n-grams and concluded that including all features without limit performed the best (see how test 34 performed better than test 33 in `results/kaggle_spooky_author_submission_results.csv`, scoring both a lower mean leaderboard logloss and a lower 10-fold cross validation logloss). In the end, two of the same MLP models were created to compare against both sets of features.

Apart from the above, a series of tests were performed on the actual preprocessing of text, which was neglected in the previous tests, left to the devices of the default Keras `Tokenizer` [15]. The text preprocessing tests, denoted as Z and A through K (with preprocessing settings 1 through 7), can be viewed in the file `results/text_preprocessing_test.txt`, and the results of the tests are located in `results/kaggle_spooky_author_submission_results.csv`, tests 2 through 29. Tests 2 through 5 were baseline tests of the best CNN and RNN models found manually, without any text preprocessing and just the default Keras `Tokenizer`. Tests 6 through 29 were conducted in order to determine the best text preprocessing settings to use for this dataset. Most of the testing was done on the same GloVe and fastText CNN models, while the

[14] https://developers.google.com/machine-learning/guides/text-classification/step-2-5
[15] https://keras.io/preprocessing/text/#tokenizer

only tests performed on the RNN models were when the best preprocessing settings were already chosen: settings `I` for GloVe and `E` for fastText. Tests 28 and 29 confirm that text preprocessing was at least beneficial to the RNN model than without (tests 3 and 4); however, 28 and 29 do not confirm that text preprocessing `I` and `E` in particular are the best for RNN models. It would have been great to test this given more time, but my focus was on the modeling for this project. In short, the best preprocessing settings (by "best," meaning, "at least for the best manually found CNN model") are to keep the casing, keep the punctuation, keep the stopwords, not stem, not lemmatize, and normalize the spelling (by converting British English words to American English spelling) for GloVe CNN and RNN models. For fastText CNN and RNN and bag-of-words MLP models, on the other hand, the preprocessing settings found to work the best are the same, except that the spelling should be left alone (not normalized). These results indicate that casing, punctuation, stopwords, and forms of a word are important in helping a model predict an author's style of writing.

Implementation

The main task of the implementation phase was to decide on neural network architectures to optimize during the refinement phase. Since I didn't have much experience working with text datasets and feeding them into neural networks, I constructed the neural network architectures with help from official or reliable sources such as the Keras blog and guides, a popular Kaggle kernel, a research paper, and Google's Text Classification guide. Architecture test details can be found in `results/cnn_architecture_test.txt` and `results/rnn_architecture_test.txt`.

The CNN architectures used were inspired by a Keras blog post[16] on pre-trained word embeddings and the "Sequence classification with 1D convolution" and "VGG-like convnet" architectures in the Keras Sequential model guide[17]. Without much configuration of hyperparameters, the best CNN model manually found was one with 250 filters, 3 kernels, and a dropout rate of 0.2. It uses the Adam optimizer and a single 1D convolutional layer followed by global max pooling and dense layers, with two dropout layers; we've denoted this architecture as the "special CNN architecture" since it only takes one convolutional layer into account (but for details on the non-special architecture, see `code/models/build_cnn_model.py`). The batch size used for manual testing of neural network architectures and text preprocessing was fixed at 64. Trained for 3 epochs, the best manual CNN model scored a 10-fold cross validation loss of 0.47104 using GloVe `glove.840B.300d.txt` embeddings.

[16] https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html
[17] https://keras.io/getting-started/sequential-model-guide/

The RNN architectures used were inspired by a popular Kaggle kernel[18] by Vladimir Demidov and one that a Stanford paper[19], entitled "Deep Learning based Authorship Identification," employed. After the input and embedding layers, it begins with a spatial 1D dropout layer followed by a bidirectional GRU layer, then concatenates separate global average pooling and global max pooling layers into the final output layer. Trained for 5 epochs, the best manual RNN model scored a 10-fold cross validation loss of 0.40661 using GloVe `glove.840B.300d.txt` embeddings.

The MLP architecture used was inspired by Google's `mlp_model` implementation for their Text Classification guide[20]. It is essentially the same as Google's architecture, except that I later allow random search to set Adam or RMSProp as the optimizer (while Google has Adam hardcoded). This MLP architecture starts with an input layer for the number of features, followed by a static dropout layer, and then it sets a series of fully connected and dropout layers (configurable based on the number of total layers), before adding on the final output layer. Trained for 1,000 epochs with early stopping in place so that the model only trained for less than 20 epochs, the MLP model with Google's suggested parameters scored a 10-fold cross validation loss of 0.35602 using the top 20,000 unigram and bigram features. The batch size used was 128, there were 64 units, the dropout rate was set to 0.2, the optimizer used was Adam, and there were only 2 fully connected layers (including the final output layer).

Note that only GloVe was used to make comparisons between different neural network architectures; therefore, by "best manual," I mean "the best model architecture manually constructed using GloVe embeddings." This means that these best manual architectures might not necessarily be the best for fastText. Regardless, I wanted both GloVe and fastText to use the same CNN and RNN architectures as a basis for comparison during random search, which is conducted later. And since fastText scored better than GloVe for both manual CNN and RNN architectures, that was a good enough benchmark for me, as far as manually selecting architectures to use for fastText.

As for the metrics, Keras already provides multiclass logloss in the form of one of its loss functions, called `categorical_crossentropy`. This just needs to be specified as a string value to the `loss` parameter in the call to the `model.compile` method. By default, loss and validation loss are monitored metrics, but we also set accuracy as a metric to monitor (and this includes validation accuracy), by supplying the string "accuracy" to the `metrics` parameter of the same compile method. While four metrics are being monitored in total, throughout this

[18] https://www.kaggle.com/yekenot/pooled-gru-fasttext
[19] https://web.stanford.edu/class/cs224n/reports/2760185.pdf
[20] https://developers.google.com/machine-learning/guides/text-classification/step-4

project, we are mainly making decisions based on the validation loss (the multiclass validation logloss, to be precise).

Before the refinement phase, the meat of the implementation for quite a while was the 10-fold cross validation loop. The loop prints out the current fold out of the total number of folds; prepares the splits of data; constructs the model callbacks used for recording the model improvement progress based on validation logloss, early stopping, and logging all the metrics (accuracy, loss, validation accuracy, and validation loss) to a CSV at the end of every epoch; builds the CNN, RNN, or MLP model; saves the Keras model summary to file; trains the model on 9/10 of the training dataset and validates on 1/10; keeps track of the best validation loss per fold and the number of epochs at which they occurred; and constructs a classification report and confusion matrix, saving them to file. After the 10-fold loop, cross validation results based on the lowest validation loss from each fold are recorded; this includes the mean accuracy, mean loss, mean validation accuracy, and mean validation loss across 10 folds, along with their standard deviations. The suggested best epochs per fold (when the lowest validation loss occurred), best scores (all four metrics recorded at the lowest validation loss) per fold, and total and average runtimes across all folds are also logged to file after the 10-fold loop.

There were no major complications that occurred during the coding process. The code is more tedious than it is complex, especially when it came to logging. I took much longer implementing logging than I would have liked, simply because I wanted to ensure that everything that might have been useful was recorded for the random search process that would take place during the refinement phase. So it took me a few iterations, from the contents of the logs to consolidating some of the log files, to get the code prepared enough for random search.

Refinement

My process choice of refinement for improving upon the neural network models was to use random (or randomized) search. In random search, parameters of the model are randomized for every iteration, as opposed to searching every possible combination of parameters in grid search. The philosophy behind random search is that, instead of trying to cover all bases with all possible combinations no matter how much time it takes, it wants to try and cover as much ground as possible in a fixed amount of time. Therefore, random search is more computationally efficient and performs on par with grid search, according to Bergstra and Bengio[21] in their paper, "Random Search for Hyper-Parameter Optimization." In addition, I chose 60 iterations because, with 60 iterations, there is a 95% chance that random search would find parameters that are within 5% of the optimum parameters[22].

[21] http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf
[22] https://stats.stackexchange.com/questions/160479/practical-hyperparameter-optimization-random-vs-grid-search

Parameters that were adjusted include the following:

- CNN models:
    - Batch size: 32, 64, 128, 256, 512
    - Filters: 32, 64, 128, 256, 300
    - Kernel size: 3, 5, 7, 9
    - Dropout rate: 0.1, 0.2, 0.3, 0.4, 0.5
    - Optimizer: Adam, RMSProp
    - Use the normal architecture 80% of the time
        - Number of convolutional stacks: 1, 2, 3
        - Add an extra convolutional layer: True, False
        - Add a dropout layer: True, False
        - Use `Flatten`[23]: True, False
        - Use a global max pooling layer (else a global average pooling layer): True, False
        - Add a final dropout layer: True, False
        - Pool size: 2, 3, 4, 5
        - Final dropout layer rate: 0.1, 0.2, 0.3, 0.4, 0.5
    - Use the special architecture 20% of the time
        - No other parameters
- RNN models:
    - Batch size: 32, 64, 128, 256, 512
    - Use a bidirectional GRU layer (else a bidirectional LSTM layer): True, False
    - Use a global max pooling layer (else that and a global average pooling layer, concatenated): True, False
    - Units: 32, 64, 128, 256, 300
    - Spatial dropout rate: 0.1, 0.2, 0.3, 0.4, 0.5
    - Optimizer: Adam, RMSProp
    - Use one RNN stack 70% of the time
    - Use two RNN stacks 30% of the time
- MLP models:
    - Batch size: 32, 64, 128, 256, 512
    - Units: 32, 64, 128, 256, 300
    - Dropout rate: 0.1, 0.2, 0.3, 0.4, 0.5
    - Optimizer: Adam, RMSProp
    - Number of total layers (including the output layer): 2, 3

[23] https://keras.io/layers/core/#flatten

After automating random search and letting it run 60 iterations for each of the six models, tests 36 through 53 in `results/kaggle_spooky_author_submission_results.csv`, I opened the `summaries` output folder for each model and examined the top three model configurations that scored the lowest (best) mean cross validation (CV) logloss across 10 folds. This validation logloss that I used for comparison is a single value representing the mean of the lowest validation losses achieved per fold. I also submitted each of the top three scoring model configurations to Kaggle in order to score the predictions on the unseen test data, which is unlabeled (and therefore must be scored via Kaggle). After submission, if the competition is already over, Kaggle provides a public leaderboard (LB) logloss and a private LB logloss. In the case of the Spooky Author Identification competition, 30% of the test dataset is calculated in the public LB score, while 70% of the dataset is calculated in the private LB score; taking this into account, we can achieve the mean LB score (see the end of any of the model notebooks in the `code/` folder for details).

After we list the scores for both the mean LB and 10-fold CV, we record the final leaderboard-cross validation (LB-CV) mean, which is the average between these two. If the LB-CV mean is lower, it doesn't necessarily mean that the model performed better. We also need to take into account the difference between the mean LB and 10-fold CV scores: they should not vary too widely. In our case, the scores shouldn't differ by more than 0.05 (an arbitrary choice, based on our observations of the scores). If it does, then the model is unreliable, showing signs of overfitting, either to the local CV or the LB. An extreme example of this is tests 30, 31, and 32 found in `results/kaggle_spooky_author_submission_results.csv`; they each have much lower 10-fold CVs than their mean LB scores.

The final results are displayed in **Figure 3** (see tests 36 through 53 and how they compare to the initial solutions, tests 14, 23, 28, 29, 33, and 34, for more details). Initial solutions are highlighted in yellow, best-of-model random search solutions are highlighted in purple, and the final best (random search) solution is highlighted in green.

Test #	Mean LB[24]	10-fold CV[25]	LB-CV Mean	Description
23	0.46028	0.47104	0.46566	GloVe best manual CNN model
42	0.50489	0.48070	0.49280	GloVe CNN, iteration 47, third[26]

[24] Mean LB: Kaggle has a public and private leaderboard (LB); the mean LB score is a measure of both.
[25] 10-fold CV: Stratified 10-fold cross validation.
[26] By first, second, and third, we mean the top 3 of 60 random search iterations that scored the lowest (by the **10-fold CV** column) for a particular model, such as the GloVe CNN model. 60 iterations were run for each model.

43	0.47489	0.47942	0.47716	GloVe CNN, iteration 15, second
44	0.48029	0.46342	0.47186	GloVe CNN, iteration 56, first
28	0.41285	0.40661	0.40973	GloVe best manual RNN model
45	0.41041	0.39837	0.40439	GloVe RNN, iteration 17, third
46	0.40793	0.39675	0.40234	GloVe RNN, iteration 33, second
47	0.42329	0.39570	0.40950	GloVe RNN, iteration 58, first
14	0.43710	0.43531	0.43621	fastText best manual CNN model
48	0.53931	0.43071	0.48501	fastText CNN, iteration 49, third
49	0.50460	0.42728	0.46594	fastText CNN, iteration 35, second
50	0.43736	0.42612	0.43174	fastText CNN, iteration 07, first
29	0.40828	0.37966	0.39397	fastText best manual RNN model
51	0.45846	0.37522	0.41684	fastText RNN, iteration 45, third
52	0.39381	0.37450	0.38416	fastText RNN, iteration 24, second
53	0.37814	0.37397	0.37606	fastText RNN, iteration 08, first
33	0.36098	0.35602	0.35850	Top 20k unigrams and bigrams, Google's suggested MLP model
39	0.35351	0.34488	0.34920	Top 20k unigrams and bigrams MLP model, iteration 10, third
40	0.34130	0.34193	0.34162	Top 20k unigrams and bigrams MLP model, iteration 11, second
41	0.34145	0.34165	0.34155	Top 20k unigrams and bigrams MLP model, iteration 15, first
34	0.34003	0.30814	0.32409	All unigrams and bigrams, Google's suggested MLP model

36	0.33373	0.30226	0.31800	All unigrams and bigrams MLP model, iteration 32, third
37	0.30494	0.30108	0.30301	All unigrams and bigrams MLP model, iteration 24, second
38	0.30781	0.30098	0.30440	All unigrams and bigrams MLP model, iteration 34, first

Figure 3: Kaggle Spooky Author Identification dataset mean leaderboard submission logloss (Mean LB), 10-fold cross validation logloss (10-fold CV), and their mean (LB-CV Mean).

From the figure above, we can see that the best and final solution is test submission 37, with an LB-CV mean of 0.30301, obtained on the 24th random search iteration of the MLP model with all unigrams and bigrams. While it only scored second-best in terms of its 10-fold CV score (compared to test 38), its mean LB score is much lower, leading to its lower and thus better LB-CV mean. For details on the parameters that powered the random search models, see `results/random_search_model_evaluations.txt`.

IV. Results

Model Evaluation and Validation

The parameters of our final model are as follows:

- Batch size: 512
- Units: 64
- Dropout rate: 0.5
- Optimizer: RMSProp
- Number of total layers (including the output layer): 2

The resulting architecture appears as follows:

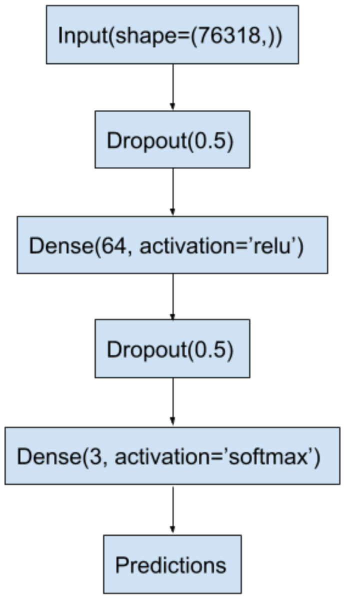

Figure 4: Final MLP model using all unigram and bigram features as input.

In comparison, Google's own suggested parameters are similar, except that it uses a batch size of 128, a dropout rate of 0.2, and the Adam optimizer. The main difference between our model and Google's is that our model uses all unigram and bigram features, whereas Google's suggestion is to use only the top 20,000 features. Comparing our tuned best model of test 37 with an LB-CV mean of 0.30301 to Google's suggested params in test 33 with an LB-CV mean of 0.35850, our model only gained about 0.05549 increased performance in logloss. In general, the untuned test 33 Google model still outperforms all the GloVe and fastText tuned and untuned CNN and RNN models.

The final results support Google's conclusions, based on their own experimentation with text classification tasks, that a simple MLP model with bag-of-word features as input would work best for this particular dataset. They have selected 20,000 features because there appears to be a plateau in performance for that number; however, they do mention that their recommendations are sufficient only to provide a strong model for a set computation time and not necessarily the best model, given more computational resources. Thus, as we have found for our own tuned model, adding more features improved performance.

The final result is sufficiently strong enough to be useful, and we can be confident that this is one of the best sets of parameters for the model because 60 iterations of random search was done, and for each iteration, 10-fold cross validation was performed for the random set of model parameters. Even on an unseen set of data (the test set), the model was able to score well and comparatively close to the 10-fold cross validation score. Therefore, this model is robust enough for the problem, since small perturbations in the training data does not greatly affect the results. In addition, the model has a large number of features (76,318) and thus is not sensitive due to a small number of features.

Justification

The final results of the random search (the tests highlighted in purple and green in **Figure 3**) are all indeed stronger (better and thus lower) than the benchmark mean leaderboard logloss of 1.08825, displayed as test 1 in `results/kaggle_spooky_author_submission_results.csv`. Furthermore, the final model and solution (highlighted in green) is significant enough to have adequately solved the problem because it far surpassed the benchmark model and does not show signs of overfitting, as can be seen in **Figures 5** and **6** below. In addition, this model was arrived at via 60 iterations of random search, and for each iteration, 10-fold cross validation was run. 10-fold cross validation makes the scores much more stable than validating with a single held-out validation set; therefore, we can be confident that the final model did not score well merely on luck. Moreover, the final model also beat out a lot of other model configurations: Five other models were run with random search and compared against the final model (one other MLP but with a different set of features, two CNNs, and two RNNs), yet the final model performed the best, scoring the lowest LB-CV logloss. Thus, the model has not only beaten itself but other types of models as well, using multiple rounds of validation.

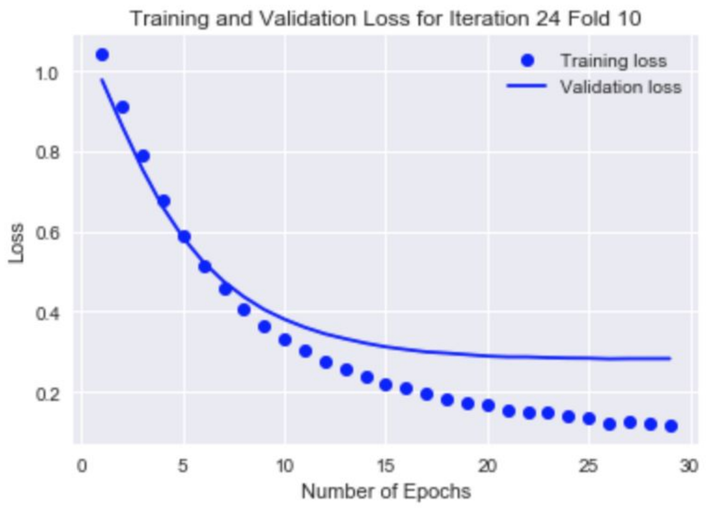

Figure 5: Final Best Model, Test 37: Training and Validation Logloss for Iteration 24 Fold 10.

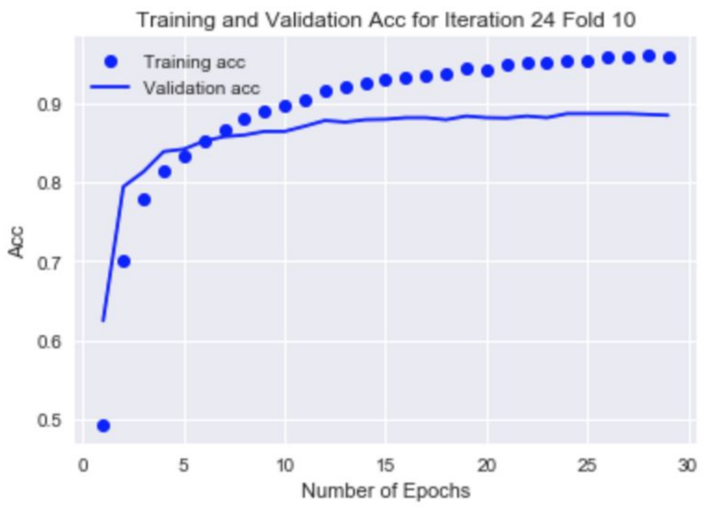

Figure 6: Final Best Model, Test 37: Training and Validation Accuracy for Iteration 24 Fold 10.

V. Conclusion

Free-Form Visualization

Selecting the most important results from **Figure 3**, I constructed four different plots shown as **Figures 7, 8, 9,** and **10** below (the code for which is in the `code/bow_mlp_34_model.ipynb` file). For details on the models represented by the test numbers, see the **Figure 3** table in the **Refinement** section.

Overall, from the plots below, we can see that the MLP models outperformed the CNN and RNN models, and that the RNN models outperformed the CNN models. We can see that fastText embeddings performed better than GloVe embeddings when comparing the same models, and that MLP models using all unigrams and bigrams outperformed MLP models that used only the top 20,000 unigrams and bigrams.

In addition, for the most part, from the **Figure 3** table, the best-of-model random search solutions highlighted in purple have lower (better) LB-CV mean scores than their yellow counterparts (except for the GloVe CNN model, in which the manually found initial solution performed the best). This is clearly illustrated in **Figure 10** below by the blue-green bar pairs (blue and dark blue for the last pair) from left to right.

Lastly, as discussed in the **Refinement** section, the best and final solution is test submission 37 (the last bar in each of the four plots below), with an LB-CV mean of 0.30301, which is an MLP model using all unigram and bigram features.

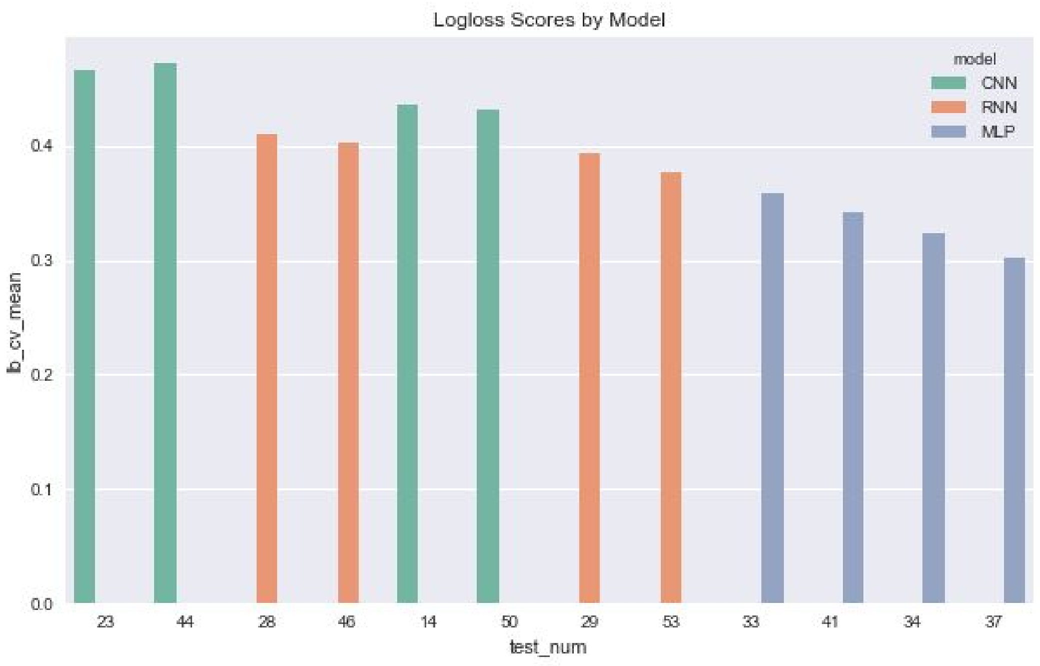

Figure 7: LB-CV Mean Logloss Scores by Neural Network Model (CNN, RNN, or MLP).

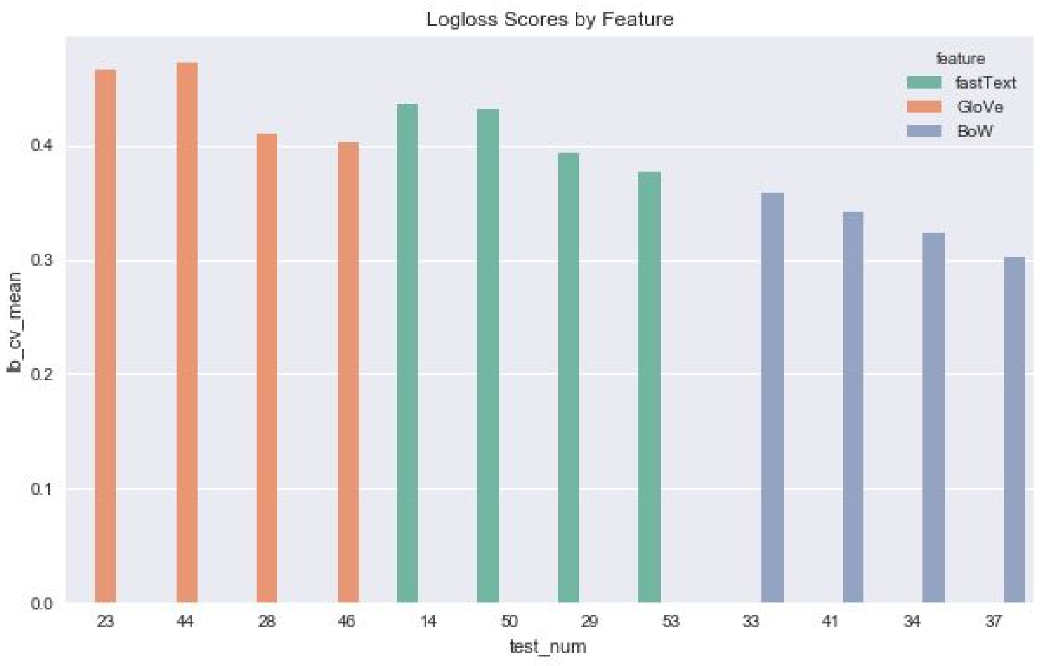

Figure 8: LB-CV Mean Logloss Scores by Feature (GloVe, fastText, or bag of words).

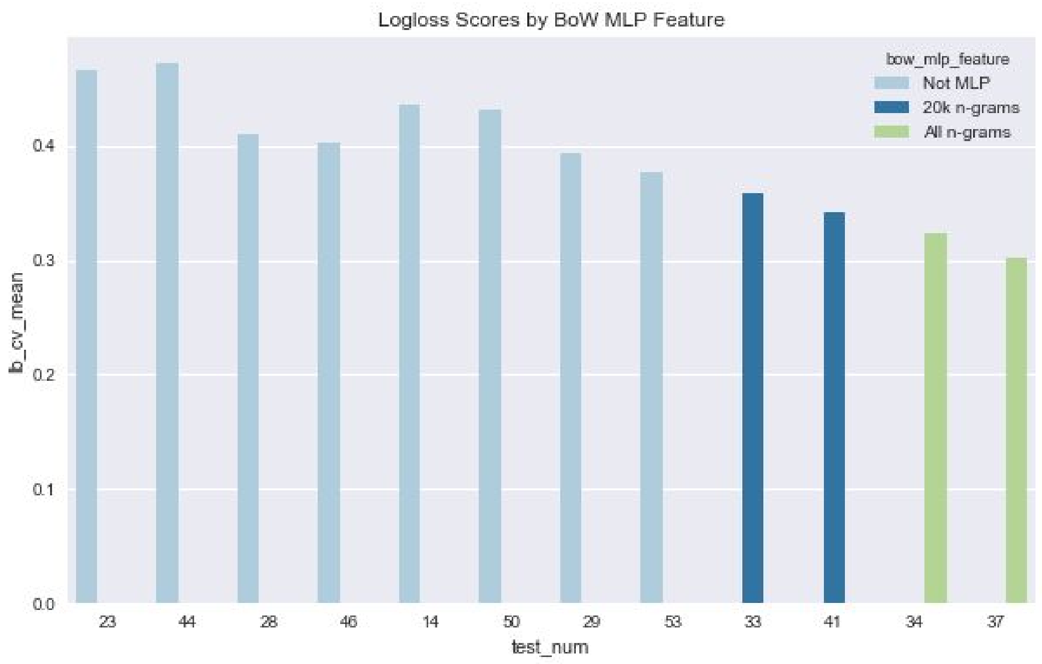

Figure 9: LB-CV Mean Logloss Scores by Bag-of-Words MLP Feature (20k or all n-grams).

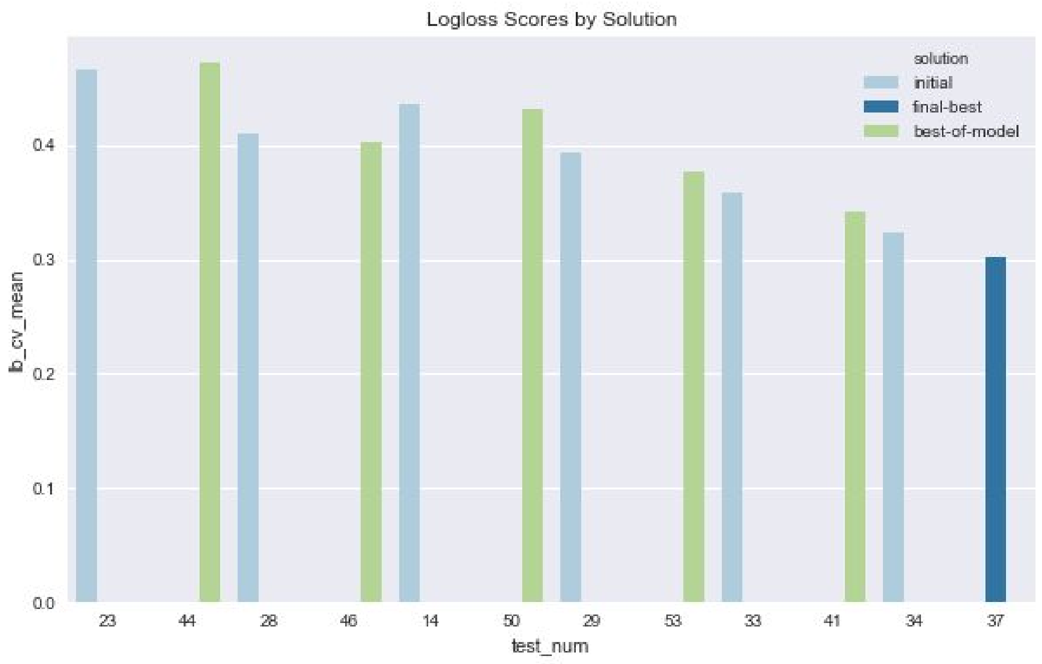

Figure 10: LB-CV Mean Logloss Scores by Solution (initial or best-of-model, then final-best). Note that the groupings appear off due to seaborn's `hue` param (empty space is being

allocated for each of the three solutions, for each test number). Blue vs. green (tests 23 vs. 44), then blue vs. green (tests 28 vs. 46), and so forth, are the bars that should be compared.

Reflection

In summary, we aimed to solve the problem of authorship attribution in this project using neural network models with bag-of-word and pre-trained word embedding features. These were the steps taken to arrive at the final problem solution:

1. Decided on multiple factors for the pre-trained word embeddings used in the CNN and RNN models:
    a. Tokenized text into words instead of characters
    b. Tested multiple Stanford GloVe and Facebook fastText 300d embeddings using 10-fold cross validation, resulting in `glove.840B.300d.txt` and `crawl-300d-2M.vec` performing the best
    c. Ran 10-fold cross validation for each of 9000, 12000, 15000, 18000, and `None` (meaning no maximum limit) for the maximum number of features and concluded that setting no maximum limit performed the best
    d. Ran 10-fold cross validation for each of 23, 27, 34, 128, 256, 512, 861, and 900 for the maximum sequence length and concluded that a length of 128 performed the best
2. Used the `glove.840B.300d.txt` embeddings and the settings above in a series of manual parameter configurations to find the best manually tuned CNN and RNN models using 10-fold cross validation
3. Made decisions based on Google's advice in their Text Classification guidelines and flowchart for the MLP model:
    a. Used their default MLP model configuration
    b. Used unigrams and bigrams for the n-gram range
    c. Used TF-IDF as the count mode
    d. Set the maximum number of features to 20,000
4. Regarding the MLP model, ran 10-fold cross validation on setting no limit on the number of features; this performed better than setting a maximum limit of 20,000
5. Ran various preprocessing tests using the best manually found GloVe CNN model and concluded that the best settings were to keep the casing, keep the punctuation, keep the stopwords, not stem, not lemmatize, and normalize the spelling (by converting British English words to American English spelling) for GloVe CNN and RNN models
6. For fastText CNN and RNN and bag-of-words MLP models, on the other hand, the preprocessing settings found to work the best are the same, except that the spelling should be left alone (not normalized)

7. Based on the best manually found CNN and RNN models and Google's MLP model, ran 60 iterations of random search for each of the following six models:
    a. GloVe `glove.840B.300d.txt` embeddings with a CNN model
    b. GloVe `glove.840B.300d.txt` embeddings with an RNN model
    c. fastText `crawl-300d-2M.vec` embeddings with a CNN model
    d. fastText `crawl-300d-2M.vec` embeddings with an RNN model
    e. 20,000 unigram and bigram features with an MLP model
    f. All unigram and bigram features with an MLP model
8. Concluded that all unigram and bigram features with an MLP model performed the best, surprisingly better than sequence models with pre-trained word embeddings.

I thought it interesting that, for the longest time, I feared random search itself would be difficult to implement, when in fact, it was the logging in preparation for random search that took me the longest time to complete, and that adding on random search afterwards was much less of an undertaking in comparison. On another note, one aspect of the project that was unexpectedly difficult was discovering, understanding, and choosing between the many number of ways to perform the same task: scikit-learn's `CountVectorizer` followed by `TfidfTransformer` vs. `TfidfVectorizer`, manually handling text preprocessing (string `split` method or NLTK's `word_tokenize`) vs. using the Keras `Tokenizer`, and so on. In the end, I had to pick a method or combine multiple methods and learn not to overthink my decisions; then once the entire machine learning pipeline was done, I was able to return to certain steps and overthink the things that actually needed overthinking, before running random search.

Improvement

While there were multiple steps involved to arrive at the final model and solution, such as all the details outlined in the **Data Preprocessing** section, manual selection of the neural network architectures, and random search, they did not account for everything. There are many aspects of this project that can still be improved and various NLP techniques to apply that have not yet been explored. Potential improvements are as follows (mainly, these are enhancements to the current process employed in the project):

- Originally, I wanted to use AI2's ELMo[27] and Salesforce's CoVe[28] word embeddings, but I realized it would take a lot more time to learn how to implement and truly understand those, on top of setting up the machine learning pipeline and everything else; I also would have had to learn PyTorch and TensorFlow to get the most out of using them

[27] https://allennlp.org/elmo
[28] https://einstein.ai/research/learned-in-translation-contextualized-word-vectors

- As discussed in the Deep Learning based Authorship Identification paper[29] from Stanford, we could try:
  - Passing in sentence vectors (such as the average of all the word vectors in a sentence) as opposed to the word vectors we've been passing in
  - Add more parameters to tune, such as regularization and learning rate
- Test whether various text preprocessing tasks improve the final model:
  - Use k-fold cross validation instead of purely Kaggle submissions to test the text preprocessing tasks as performed in the project, so that results don't vary too widely (such as what happened with the fastText text preprocessing `E` take 2 test)
  - Try other lemmatizers besides `WordNetLemmatizer`
  - Detect the language used and add it as a feature (for example, EAP sometimes writes in French)
  - Account for misspelled/accented words such as those used by HPL: "It was all mud an' water, an' the sky was dark, an' the rain was wipin' aout all tracks abaout as fast as could be."
  - Use named entity recognition as a feature, since there are mentions of places like "Windsor" and names like "Dr. Johnson"
  - Train our own embeddings or fine-tune the pre-trained embeddings
  - Handle out-of-vocabulary (OOV) words (meaning, all those words in which no word vectors were found from the pre-trained embeddings) in a better way than initializing their vectors to 0
- Include the embedding choice, max features, max sequence length, and text preprocessing tests as part of the random search iterations and increase the number of iterations
- If random search did not take as long as it did, even with a 1080 Ti behind it, I would have run a wider search on parameters of the neural networks (such as not using discrete numbers, but rather, continuous uniform distributions), along with other types of MLP, CNN, and RNN architectures. Then I would decrease the search space (eventually using discrete numbers) and run more iterations of random search.
- Learn and apply model ensembling, including models besides neural networks, such as naive Bayes, support vector machines (SVMs), and gradient boosted decision trees (GBDTs).

Even if our final solution of the MLP model with all the unigram and bigram features were used as the new benchmark, I believe an even better solution still exists, and the best first step towards finding that would be to find a better way to handle OOV words, rather than initializing

[29] https://web.stanford.edu/class/cs224n/reports/2760185.pdf

their vectors to 0. Starting out at particular vectors in the embedding space is expected to perform much better than initializing them to 0 because the pre-trained word embeddings are frozen to begin with and 0 carries much less meaning than, say, switching out the unknown words with their synonyms, for which word vectors may exist.

VI. References

- Bergstra, J., & Bengio, Y. (2012, March 11). Random Search for Hyper-Parameter Optimization. Retrieved from http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf
- Chollet, F. (n.d.). Getting started with the Keras Sequential model. Retrieved from https://keras.io/getting-started/sequential-model-guide/
- Chollet, F. (2016, July 16). Using pre-trained word embeddings in a Keras model. Retrieved from https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html
- Chollet, F. (2018). Deep Learning with Python. Shelter Island, NY: Manning Publications.
- Collier, A. B. (2015, December 14). Making Sense of Logarithmic Loss. Retrieved from https://datawookie.netlify.com/blog/2015/12/making-sense-of-logarithmic-loss/
- Demidov, V. (2018, February). Pooled GRU + FastText. Retrieved from https://www.kaggle.com/yekenot/pooled-gru-fasttext/code
- Firebug, & Bar. (2016, April 26). Practical hyperparameter optimization: Random vs. grid search. Retrieved from https://stats.stackexchange.com/questions/160479/practical-hyperparameter-optimization-random-vs-grid-search
- Google. (2018, July 23). Text Classification. Retrieved from https://developers.google.com/machine-learning/guides/text-classification/
- Kaggle. (2017, November). Spooky Author Identification. Retrieved from https://www.kaggle.com/c/spooky-author-identification
- Keras Documentation. (n.d.). Retrieved from https://keras.io/
- McCann, B. (2017, July 31). Learned in translation: contextualized word vectors. Retrieved from https://einstein.ai/research/learned-in-translation-contextualized-word-vectors
- Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., & Joulin, A. (2017, December 26). Advances in Pre-Training Distributed Word Representations. Retrieved from https://fasttext.cc/docs/en/english-vectors.html
- Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global Vectors for Word Representation. Retrieved from https://nlp.stanford.edu/projects/glove/

- Peters, M. E., Neumann M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018, March 22). Deep contextualized word representations. Retrieved from https://allennlp.org/elmo
- Qian, C., He, T., & Zhang, R. (2018, April 4). Deep Learning based Authorship Identification. Retrieved from https://web.stanford.edu/class/cs224n/reports/2760185.pdf
- The SciPy community. (2018, May 5). Scipy.sparse.csr_matrix. Retrieved from https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html