1 Introduction

Our objective in this project is to explore the suitability of Naive Bayes, LSTM and CNN classifiers in predicting the author of excerpts from horror stories by Edgar Allan Poe, Mary Shelley, and HP Lovecraft. This is a Kaggle Competition and we want to try these different predictors and come to a conclusion about why we believe that certain predictors work better than others. Our baseline predictor will be Multinomial Naive Bayes.

2 Data Preprocessing and Word Embeddings

2.1 Data Preprocessing

We used a training dataset of 20000 sentences with labels indicating the author of the sentence. Preprocessing of data was done using NLTK’s word tokenizer, that tokenized the sentences at white-spaces. We intentionally did not remove stop-words or the punctuation marks, since we believe that they contribute to identifying a particular writing style of an author. With the processed data, we created word embeddings and looked at the similarities between pairs of words.

2.2 Word Embeddings

With a dimensionality of 300, we implemented and experimented with two models of word embeddings: 1) Pre-trained GloVe, and 2) Word2vec. We also performed dimensionality reduction with the standard t-SNE technique and visualized our embeddings in two dimensions.

2.2.1 Pre-trained GloVe Embeddings

We used freely available general-purpose predictive model based word embeddings by GloVe [1]. These embeddings are pre-trained on 42 billion tokens, with 1.9 million vocabulary size. We observed that pre-trained word embeddings provide a good start, and can speedup the processes of training a classifier, but they have their own drawbacks. Since GloVe embeddings are trained on a huge dataset of words, a lot of information captured in them is in fact irrelevant to us. On the other hand, few words in our dataset were not present in GloVe’s vocabulary, potentially affecting our accuracy.

2.2.2 Gensim Word2vec

To overcome the above drawbacks of pre-trained embeddings, we also experimented with another approach: generating our own embeddings based on the training dataset. We used Gensim’s count based word2vec algorithm [2] to achieve this. We observed that the word2vec model does not capture information as rich as the GloVe model but, as far as our dataset is concerned, it could reflect the similarities and differences between the words more strongly compared to GloVe. For example, according to GloVe, similarity between the words ‘gold’ and ‘silver’ is 0.866, and between ‘gold’ and ‘water’ is 0.391. But with word2vec, similarity between ‘gold’ and ‘silver’ is just 0.210, and that between ‘gold’ and ‘water’ is merely 0.072. Though GloVe captures that ‘gold’ and ‘silver’ are closely related better than word2vec, word2vec is able to capture that ‘gold’ and ‘water’ are very much unrelated relative to ‘gold’ and ‘silver’.

We also visualized our embeddings in two dimensions using t-SNE dimensionality reduction (fig 1 and 2). We found it really hard to spot similar words being plotted close to each other, even though in 300 dimensions they possess the similarity.
Figure 1: Two dimensional t-SNE visualization of word2vec embeddings

Figure 2: Zoomed region of Fig 1
3 Naive Bayes Classifier

3.1 Introduction
When attempting to build or improve on a classifier, it is important to create a base classifier to measure against. For this we use a Naive Bayes Classifier. This classifier is computationally less expensive than many other classifiers and is known in some cases to perform as well. If our new classifiers perform better than Naive Bayes then we know that it is a viable solution to our problem.

3.2 Theory
A Naive Bayes classifier relies on the results from Bayes Theorem seen in equation 1. If we have a feature vector \( \bar{X} \) and a given class \( C_i \) then we get the equation seen in 2. In our case vector \( \bar{X} \) is a sequence of words and \( n \) is the number of words in the text segment.

\[
P(A|B) = \frac{P(B|A)P(B)}{P(A)} \tag{1}
\]

\[
P(C_i|\bar{X}) = \frac{P(x_1|C_i)P(x_2|C_i)P(x_3|C_i) \cdots P(x_n|C_i)P(C_i)}{P(\bar{X})} \tag{2}
\]

To train the classifier we use the training set to generate the conditional probabilities of \( P(X_1|C_i) \) as well as the class probability known as the prior \( P(C_i) \). Once these are all calculated we can perform the classification which is done by calculating the probability for each class given a feature vector. However, there are some things to look out for. The main one being that when calculating the conditional probability you calculate the probability of every word in your vocabulary occurring in a given class. But, what if a word occurs in a different class as it is part of the vocabulary but is in the current class you are observing? This would create a probability of 0 which would effect your system. So we perform a process known as additive smoothing so that our conditional probability is now given by:

\[
P(x_i|C_i) = \frac{\text{Occurrence}_X_i}{\sum_{X_i \in \text{Vocab}}(\text{Occurrence}_X_i + 1)}
\]

Then to classify we grab the class with the highest probability as our prediction. This is shown mathematically below:

\[
\arg\max_{C_i \in C}(P(C_i|\bar{X})) \Rightarrow \arg\max_{C_i \in C}(P(x_1|C_i)P(x_2|C_i)P(x_3|C_i) \cdots P(x_n|C_i)P(C_i))
\]

We drop the \( P(\bar{X}) \) from the equation because it is constant throughout all of the classes. Next we take the log of both sides for the sake of floating point precision because the probabilities become small quickly.

\[
\arg\max_{C_i \in C}(\log(P(C_i|\bar{X}))) \Rightarrow \arg\max_{C_i \in C}(\log(P(x_1|C_i)P(x_2|C_i)P(x_3|C_i) \cdots P(x_n|C_i)P(C_i)))
\]

\[
= \arg\max_{C_i \in C}(\log(P(x_1|C_i)) + \log(P(x_1|C_i)) + \cdots + \log(P(x_1|C_i)) + \log(P(C_i)))
\]

3.3 Results
It turns out that Naive Bayes did not have the best results. To try to improve the results I tried removing things such as stop words, using only low frequency words or high frequency words, etc. The results can be seen in figures 3 and 4.
These results show that the classifier that performed the best was when stop words were removed and only the 100 most frequently used words in the vocabulary formed from the training documents were considered. This is most likely because stop words are used equally amongst authors so including them will only decrease the importance of unique features of words used by a certain author. Also, by only training on the highest occurring words from the vocabulary allows Bayes to focus on those features only. However, it is important to note that when I took the least frequently used words and only trained with those in the vocabulary we had similar results with the trend being that the more words you add into the vocabulary the less accurate the predictor becomes.

4 Classification using Weka

Waikato Environment for Knowledge Analysis (Weka) is a suite of machine learning software written in Java. We used Weka to perform 7 classification algorithms on our training dataset to check our own implementations. The consolidated results are given in the table below for 19579 instances(66:34::train::validation). Weka’s inbuilt tf-idf module is used for this purpose.
From the table we see that the Random forest algorithm performs best with 44.21 percent accuracy.

5 LSTM Classification

5.1 Introduction
Feed-forward neural networks (FFNN) are very effective in predicting outcomes of various problems. However, these networks never consider possible effects of previous inputs on the current output. Simply, such networks do not know the concept of time. However, most of the structure in sequential data is contained as dependencies between realizations of the random variable. If we say authors have certain use of language (rhetoric), to some extent, this assumption is reasonable. Motivated by this idea, we use a specific type of a neural network that is designed to capture sequential dependencies to model this problem. This section implements the Long Short-Term Memory (LSTM) network to predict the label (author) of sequential text data (sentences from books). Our main focus is understand the effects of hyper-parameters and how they affect the model.

5.2 Design
The idea of a recurrent neural network (RNN) is simple: the output in one state depends on the output of previous state. This distinguishes RNNs from regular FFNNs. A feedback loop enables a neuron to weight the the previous input. Generalizing this allows a network to retrieve information from the sequential structure of data. Below is the diagram that represents this flow. Neural networks are trained by using the gradients that calculates the change in error by back propagation.

‘Vanishing Gradients’ problem becomes problematic in RNNs, where back propagation is both through layers and time. This is tackled in the paper by S.Hochreiter and J.Schmidhuber where they introduce a novel and efficient gradient based method called long short-term memory. LSTM’s are preferred since they handle vanishing gradients better.

5.3 Hyper-parameters of LSTM
The model is a single layer LSTM, trained on mini batches.

- **batchSize**: Size of each input batch. Although this will not affect the model’s predictive power, we expect to see its effect on training. It should affect the noise in the training loss, also number of epochs too, for the model to fit well.

- **lstmUnits**: Number of units in the LSTM layer. This is the most prominent hyper-parameter of the model. It affects both predictive power and training behavior.
• **keepProb**: Keep probability in the drop-out layer. The model ignores the output of randomly selected units, constrains the model from overfitting to training data.

• **maxLength**: This is the sequence length that the model accepts. Both CNN and LSTM use a sequence length of 100 words.

• **Number of LSTM layers**: No clear theoretical support for stacking layers. However, there are various articles and reports claiming stacking layer improves models in certain applications. We do not see any improvement here.

### 5.4 Methodology

The training step uses hold-out validation. The dataset is split in two, validation set is 0.8% of the complete set. In ‘Tensorflow’, fixed size timesteps should be defined for computational graph. However, data consists of sequences of varying length. We define **maxLength** as the number of timesteps. Sequences shorter than this **maxLength** are padded with zeros, longer sequences are trimmed. Inside the computational graph another variable is defined to store the actual length of the sequences padded with zeros. After scanning each word in a sentence, the LSTM produces an output. Thus, padded zeros may introduce noise into the model. The additional variable is fed into the model to supply this information. Also, we need to extract correct output for each input. This is done by flattening the final output of the network and retrieving the output at the last non-zero position. This diagram visualizes LSTM cell’s updates.

![Figure 7: LSTM Updates](image)

The loss function is specified as the cross-entropy loss. Adam optimizer is used.

### 5.5 Parameter Tuning

In this part, we tune the parameters of the model. Parameters are tuned in the order of the batch size, number of LSTM cells, drop-out probability and sequence length. This gives us a complete understanding of the model’s capabilities and limitations. We tune each parameter while keeping others constant.

• **batchSize**: We consider batch sizes of 10, 50, 100, 400 and 1000. As expected, all models converge to the almost same minimum loss (there is room for variance due to shuffling of data). A batch size of 400 gives minimum variance in loss.

• **lstmUnits**: Candidates are 12, 24, 50, 100 or 250 units. Performance is similar with 100 and 250 LSTM units, thus prefer lstmUnits = 100. Also the model overfits more quickly with increasing number of units. Resulting plot is given below.
• **keepProb**: Tuning drop-out layers was purely exploratory, we did not see a significant change in model performance. When keepProb = 0.75, the model’s performance is slightly better and training is faster.

• **maxLength**: As we increase the maximum length from 10 to 100, every increment yielded a significant improvement. However, 100 and 200 words achieved a similar performance. Considering the original dataset has very little data with length larger than 100, this seems plausible. We set maxLength = 100 and plot the loss and accuracy of models for the training and validation sets.

• **Number of Layers**: Tune number of layers in the model. We do not see any improvement by adding more LSTM layers, implemented for 1-2-3- layers.
5.6 Discussion and Future Work

After tuning all the parameters, the optimal model is a single layer LSTM that has 100 cells with drop-out probability of 0.25. The model works with batches of size 400. Every input has a maximum length of 100. This model achieves a minimum loss of 0.48 on the test set with 83% accuracy.

The tuning section explored effects of almost all parameters in the model. In addition, we tried stacking layers which did not improve the predictive power. As future work, punctuation can either be removed or replaced with context tags. Also one can remove the stop words to see the effect.

6 Convolution Neural Network

6.1 Introduction

Convolutional Neural Networks (CNN)s are intuitively applied to image processing. The action of cascaded feature extraction of an input using a sliding kernel makes most sense on images. It is, therefore, no surprise that CNNs have rarely been applied to text classification. However, in this section we will explore how the hyper-parameters can affect the performance of CNNs on language problems.

6.2 Hyper-parameters of CNNs applied to NLP

These hyper-parameters affect a CNN’s behavior:

- **Filter Size**: The filter size determines how many words will affect the result of a convolution. Intuitively, the filter size can be used to explore the relationship between contiguous words. A filter size of 1 retains no text context, whilst 2 or more words add different contexts. This is particularly useful for noticing an author’s signature phrases or idiosyncratic expressions.

- **Stride Size**: How many words to jump over at each step. Usually a stride of 1 is used so that each word has a convolution applied. A large stride size shifts the model towards a tree structure seen in Recursive Neural Nets (shown in Figure 11). This can perhaps help emulate an LSTM. We will not explore this parameter.
• **Pooling Layer:** The convolution output is typically pooled over in a sliding window. In NLP, we usually apply max pooling over the complete output, yielding a single number per filter. Why do we pool?
  1. Pooling provides a fixed size output, which is required for classification
  2. The fixed size output also makes it robust to variable length input
  3. Most importantly, a max pooling can highlight certain features like negations in sentences. For example, the phrase ‘not great’ can have a high value, which the pooling layer catches if a window contains that phrase. However, we do lose phrase locality.

• **Channels:** Images have 3 color channels. Similarly, one can add different word embeddings as different data perspectives. This will not be explored in this report.

### 6.3 Methodology

The focus of this study will be training epochs, filter sizes and multilayer vs single layer models. More specifically, we shall:

• compare losses produced by 2 embedding layers: one that is trained on the training set using Gensim and the other trained using GloVe’s algorithm.

• determine the number of epochs for optimum fitting

• determine optimum filter size(s)

• quantify the effect of dropout

• discover the effect of a multilayer CNN vs a flat CNN. The two CNNs we shall compare are:

  1. A 3 layer (multilayer) 1D CNN with alternating layers of convolution followed by local 1D pooling. This was implemented using Keras for convenience’s sake.

  2. A flat single layer CNN with 1 convolution followed by global pooling. We can vary the number of filters: for example we can convolve the whole batch using a filter of size 3 words, a second filter of size 4 words and a third filter of size 5 words, potentially extracting 3 different features. This was implemented in TensorFlow based on Denny Britz WildML tutorial[^3]. Figure **12** represents the flat model.
The correct methodology for tuning the listed parameters is performing K-fold cross validation. Due to limitations in hardware and time this was not practical. Therefore, the alternative chosen was to train the different models on the whole training set, generate probabilities which are then scored on the hidden test set by Kaggle. For the purposes of this section, we shall use the the score (= loss) reported by Kaggle as our evaluation criteria.

6.4 Results

6.4.1 Log Loss vs Training Epochs

This author varied the number of training epochs. 4 models were compared:

- **3-Layer CNN, Gensim** - A 3 layer CNN using Gensim embedding
- **3-Layer CNN, GloVe** - A 3 layer CNN using GloVe embedding
- **1-Layer CNN, Gensim** - A single layer CNN using Gensim embedding
- **1-Layer CNN, GloVe** - A single layer CNN using GloVe embedding

The filter size for each model was set to 3. Only max pooling was used.
Figure 13: How each model is overfitted at high number of iterations. The flat 1-layer CNNs have optimum fitting at 10 epochs. The multi-layer models seem to degrade as the number of optimizations increases. The optimum is 1 to 2 epochs. This suggests to us that repeated cascaded convolutions does not help in feature extraction of sentences; performing a single convolution is the better approach.

Another observation is that the two different embedding algorithms, GloVe and Gensim, do not seem to have substantial differences. They use the same criteria for semantic similarity.

6.4.2 Effects of Filter Size

Seeing as the embedding algorithm did not have any effect, and that a flat CNN model is more appropriate for our purposes, the following experiments were run on the 1-Layer CNN, GloVe model.

Now we see how filter sizes affect the log loss. We expect that there will be an optimum filter size which minimizes loss, thus the graph of loss vs filter size will have a V shape: if the filter size is too small, then we will not pick up important features across phrases. If filter is too large then smaller but important features within a window can be wiped out by a subsequent pooling layer. The results, shown in Figure 14, display this behavior. A filter size of 2 is the optimum selection. A filter size of 4 is slightly anomalous since it produces an error worse than filters of size 3 and 5. This may be attributed to the structure of the training text, noise and random selection of the training batches. Why an optimum size of 2? This suggests that the model is picking up 2 word negations or stop word pairs. Filter size and pooling function collaboratively determine feature extraction.

6.4.3 Multiple Filters, Dropout, Punctuation

What happens if we use multiple filters across the same input? Figure 15 shows us the results of using 2 more filters of varying sizes. We also check if dropout and keeping punctuation can help us improve our results.
Using multiple filters improves the accuracy by 1%. The best option according to Figure 15 is to use 2 filters of size 2 and 3. Dropout, unexpectedly, does not reduce the loss. This suggests that the dropout rate chosen was too high and caused the model to underfit. Another test done was to see if including all punctuation except ‘tab’ and ‘newline’ improved on the best result. The answer was a no: the model could not find any correct correlation between punctuation use and author. Both the model and embedding matrix are culpable for this. Finally, we checked if our custom embedding was better for our task than GloVe’s pre-trained embedding. Our embedding performed better due to training the embedding specifically on the texts we want to classify, thus allowing the relationship between words to be better defined. This all culminated in a best score of 0.39822.

### 6.4.4 Future Work

To more fully understand how a CNN solves an NLP problem, we need to look more closely at the pooling layer. In particular, different size filters should have different pooling functions: max pooling over 2 words may be appropriate to find negations but we do not want the negation to override features collected over 5 words. The second avenue to improving the score is using multiple embeddings: create an embedding for each author based on the training set, or replace punctuation with tags that describe a certain context. An example of this might be exclamation marks being replaced by ¡fright¿, ¡surprise¿ or ¡excite¿ depending upon the context.

### 7 Conclusion

We observed that GloVe and Gensim algorithms perform with very similar accuracies. GloVe takes up huge amount of memory to run, while Gensim requires more time to train.

The best score achieved by the CNN in spooky author identification was 0.39822. This was done using a flat single layer of 2, 3 filters convolution followed by a global max pooling layer. Conversely, the best LSTM scored 0.48120. This model had 100 cells, a drop-out of 0.25 and batch size of 400. Both were vast improvements over Naïve Bayes.

Contrary to our expectations, the CNN performed better than the LSTM. This may be explained by the data set favoring the short filtering of the CNN. Having just scratched the surface of NLP, we have learned that fine tuning the models requires deep knowledge of the data set and the effects of the hyper-parameters.
References


