CIS 520 Fall 2020, Project Report

News Category Classification based on headlines and short descriptions

Team Members:

- Amine Soufaih; Email: asoufaih@wharton.upenn.edu
- Katrina Lee; Email: katrlee@wharton.upenn.edu
- Thanyaphorn Thangthanakul; Email: tthang@wharton.upenn.edu
- Warren Wang; Email: warren.wang.wh22@wharton.upenn.edu

Abstract

In this project, we aim to predict and classify news articles within different preset categories using their titles and descriptions. In order to achieve this goal, we used a publicly available dataset on Kaggle, a news category dataset containing over 200,000 HuffPost articles from 2012 to 2018 with text headlines and short descriptions, among other specific identifiers. To handle this supervised text classification task for most of our methods, we created a bag of words representation of each headline and short description combined and ran a Term frequency-inverse document frequency (tf-idf) scheme through it. Our baseline model was a simple majority classifier with an accuracy of 26.71% - outside of that, we implemented a variety of supervised machine learning models, including Logistic Regression, SVMs (Kernel and Linear), KNN, and different types of Neural Networks. We found the traditional one hidden layer Neural Network to have the highest test accuracy at 80.73%. Other models performed decently well, with accuracies mostly in the high 70s, far exceeding our baseline standard.

https://www.overleaf.com/project/5fafdd24e70260247d0766bf

1 Motivation

In today's society, the amount of information and news on the Internet is growing exponentially. At the same time, our demand for news is also growing rapidly, but only a small subset of the abundance of information is of interest to us. This leads to our question of how do we correctly classify news such that users can access information of interest efficiently and effectively? With the appropriate classification of news articles, users could then be recommended news that they deem the most relevant. Thus, in this project, we aim to implement machine learning models to classify real news articles each into pre-determined categories given their respective titles and short descriptions.

2 Related Work

Since we got the data from Kaggle's News Category Dataset [9], there were many Kaggle notebooks that experimented with this dataset. Most of the projects done using this dataset focused on classifying news articles into different types of articles using selected data such as headlines and descriptions. Some other projects tried to use information from previously read articles to give recommendations on other new articles.

Similar to our project, some notebooks used a Naive Bayes Classifier to predict the news categories of each article. One notebook used a multinomial Naive Bayes model where predictions are made in accordance to a Multinomial given the probability of each event[3]. This model, implemented using the same news articles dataset used in our investigation, relied on the number of time certain events were observed which is equivalent to their probabilities.The event in this case are words which are counted across all categories to estimate the probability of each word per class. Another method that we pursued in this project that has been already explored by other notebooks is Linear SVM[11]. Although these simpler methods were noted

as extremely efficient and high performing for short text classification tasks, when datasets were larger, they understandably tended to underperform more complex models like neural nets.

When it comes to the main methods used to model this data set, most notebooks extensively used generic neural networks and CNNs to perform this classification tasks. One notebook used GloVe for obtaining vector representations for words and then used various methods to model the data, such as CNN, LTSM, GRU, etc [4]. Some other common methods mentioned were the use of HAN (Hierarchical Attention Network) and BERT machine learning framework for NLP [5, 2]. Across a large dataset like this one, these methods performed notably better than their simpler counterparts aforementioned. Additionally, RNNs like LSTM can learn from word sequences rather than bags, which could be important (or not) in this task as well.

One notebook carefully described the detailed steps of removing stop words, equalizing the number of categories, performing EDA, tokenizing and sequencing text, word embedding, etc [1]. These steps are extremely adaptable and could be used as a guideline for our NLP analysis, particularly for neural networks. Sequencing text is occasionally a tricky subject - in some text-related scenarios, the sequence of words doesn't matter as much (for ex. when classification can be determined simply from the vocabulary contained within each text input). Then, a bag of words representation along with classifiers that don't take this word ordering into account would likely trump preprocessing that does. All in all, from a lot of these notebooks, we see that the focus was mostly on the model-building part of the CNN without much description on the motivation behind the use of those models.

3 Data Exploration

The complete News Category Dataset on Kaggle contains 200,853 rows of news category, author, date, link, headline, and short description. The data was obtained from HuffPost from the year 2012 to 2018. The intended purpose of this data is for training the model to identify tags for untracked news articles or identify the type of language used in different news categories. Since we only consider using headline and short description to predict the category of each news article, we dropped the unnecessary columns, such as author and link.

The original data contains 41 news categories. At first glance, we can immediately see that the categories are significantly imbalanced. The top 3 categories (Politics, Wellness, and Entertainment) make up around 33 percent of the entire dataset.

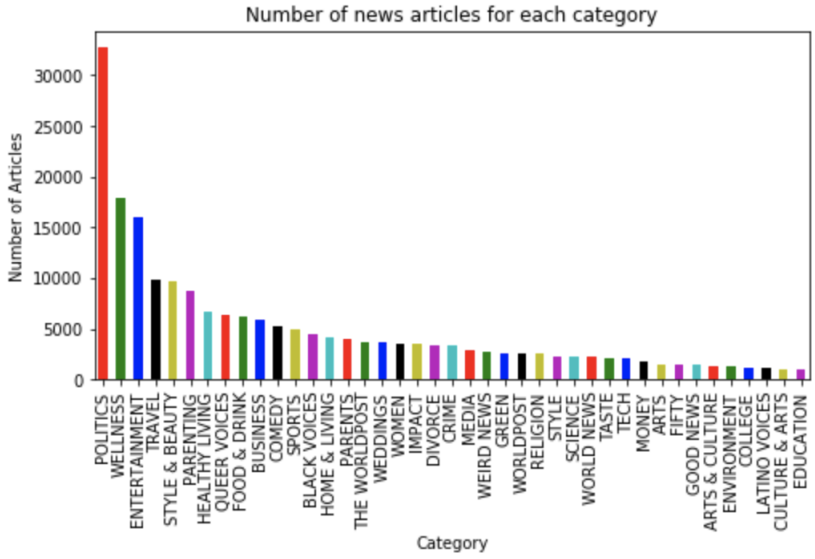

Figure 1: Number of news articles per category in full dataset

We noticed that some small categories have very similar names and/or characteristics, and sometimes one category was a subset of another. For example, Arts, Arts & Culture, and Culture & Arts/ Education

and College/ Style and Taste. There are also some vague categories that can still be hard to classify even with human judgement. For example, one would have difficulty classifying what would be Weird News or Good News. To minimize this confusion and put more emphasis on the model-building aspect of this project, we decided to cut the data down to just the top 10 categories. Those categories were: Politics, Wellness, Entertainment, Travel, Style & Beauty, Parenting, Healthy Living, Queer Voices, Food Drink, and Business.

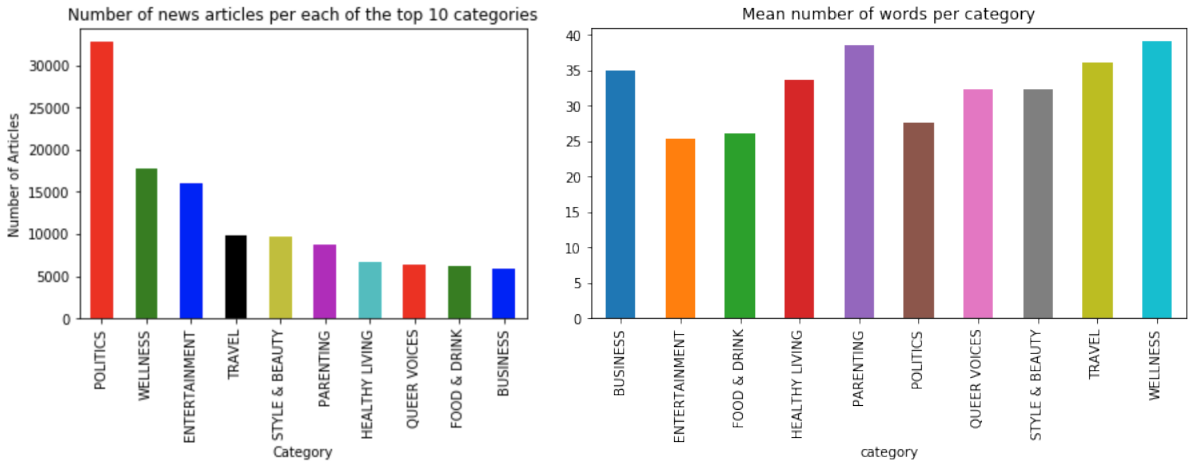

Figure 2: Top 10 Category Visualizations

The truncated dataset obtained from just observations classified in the top 10 categories contains 120,008 rows prior to any preprocessing. For the top 10 categories, we found that the average number of words per news category are quite homogeneous, ranging from around 25 to 39 words. To learn more about the training data, we also looked at the word clouds for top categories such as Politics and Wellness:

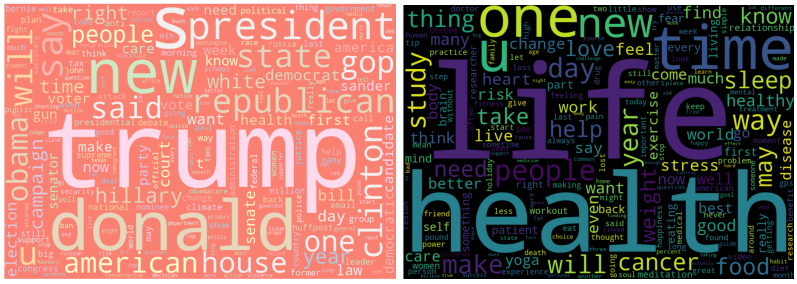

Figure 3: Politics (left) and Wellness (right) Word Clouds

4 Data Preprocessing

To begin, we combined the headline and short description columns into a single attribute, "text", as the input for our classifiers. This was also the method employed in various sources as described in the "Related Work" section. To simplify and standardize, we also dropped the rows with NA values. After these basic high-level decisions, our dataset contained 110,723 observations with the aforementioned "text" column, each labeled with one of the 10 possible news categories. We then proceeded to split the data into 90-10 for train-test, with the validation set for hyperparameter tuning being 10% of the training set as well. The training-testing split was conducted via stratified sampling so our test results would better represent the overall dataset.

4.1 Term Count TFIDF

The overall data preprocessing was mainly focused on turning text content into a numerical representation. We tokenized the text and created a "bag of words" representation using the CountVectorizer method in sklearn. This method builds a dictionary of features and transforms documents to feature vectors. Now with the words' occurrence counts, we had to address the discrepancies between larger documents and shorter documents (larger documents will have larger count values, thus skewing the relative importance of certain words). To ameliorate this issue, we employed Term Frequency times Inverse Document Frequency (TFIDF), which divides the number of occurrences of each word in the document by the total number of words in the document and downscales weights for words that occur in many texts. This works well in rendering the effect of stopwords, for example (ex. "the" and "and") almost negligible, because these common words that appear in many texts should be less informative than those that are unique to only a few texts in the entire dataset. To do so, we used sklearn's TfidfTransformer after CountVectorizer, which is equivalent to the TfidfVectorizer method.

The transformation resulted in a sparse matrix with more than 60,000 features (words in our vocabulary). The high number of features was slightly concerning to us, especially in the context of fitting complex models. The danger of overfitting would be significant, and the extremely long training time for some models would prevent us from getting meaningful results given the project timeline. To address this issue, we attempted several approaches for dimensionality reduction. First, we quickly realized sklearn's PCA algorithm didn't work on sparse matrices. Many sources recommended we utilize the TruncatedSVD method instead, which employs a variation of singular value decomposition (SVD) that has been known to work well on term count/tf-idf matrices and is frequently used as a latent semantic analysis (LSA) technique in NLP. Nevertheless, the TruncatedSVD approach did not yield adequate results - we could not get a satisfactory balance between a high enough cumulative explained variance and a reasonable number of components - even with 3,000 components, less than 70% of the variance was explained. However, probably the biggest reason we chose to forgo the SVD method was that our models actually ran much slower with the newly reduced matrix, probably due to the fact that it wasn't sparse (like our original input matrix) anymore.

Ultimately, to achieve dimensionality reduction, we chose to specify the "min_df" argument in CountVectorizer to be 20. This is a cutoff frequently used in pertinent literature which stipulates to the method that, when building the vocabulary, it should eliminate terms that have a document frequency strictly lower than 20. After this small addition, the 60,000-plus word vocabulary was reduced to 10,306, a much more manageable number, and it allowed us to retain to retain matrix sparsity and most of the important term identifiers anyway.

4.2 GloVe Embedding for CNN

It is important to note that CNN requires different data preprocessing steps than other methods. A CNN learns from sequences of words in text as each filter slides through, as texts have a one-dimensional structure where words sequence matter. While our term count/tf-idf processing captures the relative occurrence of each word, it does not take into account the ordering of words. Therefore, we chose to perform another type of preprocessing specifically for CNN.

On the "text" column, we tokenized each word in the text using a Tokenizer method from Keras. The Tokenizer method creates a vocabulary index based on word frequency so that every word gets a unique integer value. We transformed each word in text to a sequence of corresponding integers. We also used 50 as our padding length for the words. Then, we performed GloVe embedding [6] on those words. We used the 100d pre-trained word vectors of Wikipedia 2014 + Gigaword 5 data from the Stanford NLP GloVe's project website. This embedding was defined as the first layer in our CNN. In doing so, all these preprocessing steps leave us with a embedded matrix that retained the word ordering to be used in training our convolutional neural networks.

5 Problem Formulation

With defined features and labels, news categorization in this case is clearly a supervised learning problem. We have a set of over 110,000 news articles with their total weighted vocabulary according to the TFIDF

scheme described above as an (X) 10,306 feature matrix, which was used to predict one of the 10 possible categorical labels (our y). As seen later, we considered only total and in-class accuracy (what percentage of the final class predictions on the test set were equal to the actual test set labels) when evaluating model performance. Deciding which news articles are more important than others is a tricky issue, and one that doesn't have a clear answer, so we safely assumed the false positive/negative penalties should be relatively uniform. Loss functions varied depending on method (i.e. negative log likelihood loss for neural nets and modified huber loss for linear SVM), but all were selected either via hyperparameter tuning or derived from trustworthy sources.

6 Methods

Since our aim is to classify categories of news articles via supervised learning, we identified various, previously high-performing supervised classification methods to aid our investigation. With an imbalanced dataset and accuracy as our main performance metric, the first method we employed was a simple majority classifier, which served as our baseline model. Since politics is the majority class of our data, the baseline model that predicts every label as politics yielded an accuracy of 26.71%. This accuracy is consistent for all training, validation, and test sets.

The first method we implemented was Multinomial Naive Bayes via sklearn's MultinomialNB() method, which is known to be very suitable for text classification with word counts or a sparse tf-idf matrix. With our "bag of words" representation, Naive Bayes's conditional independence assumption holds and actually becomes very useful, as it allows for an extremely efficient running time. This model served as a reference point for our other considerably more complex models.

We also decided to fit a logistic regression model. While we did not expect a strong linear relationship for our data, we believed it is interesting to experiment with logistic regression. The model is easy to implement, interpret, and efficient to train, especially in our case when the number of observations is greater than the number of features. Logistic regression would also be an interesting model to compare with our next classifier, Linear SVM.

We also decided to use the SVM model, as it is generally a powerful supervised machine learning model for classification problems. We started off with the linear SVM model, the base SVM model. To fit a linear SVM model, we used Sklearn linear model's stochastic gradient descent (SGD) classifier package, which is a set of linear classifiers, including SVM with SGD training. We chose this package instead of the SVM package because, firstly, its implementation works well with data represented as dense or sparse arrays, which is the form of our training and test data. More importantly, this Sklearn model runs a lot faster than Sklearn's regular SVM package, as it uses SGD. To compare with the linear SVM model, we also fit a Gaussian kernel SVM model using Sklearn's SVM package. In cases where the data is not linearly separable, the Gaussian kernel SVM model (by projecting the data into a higher dimensional space where the data becomes linearly separable) could give a higher test accuracy.

Another model that we decided to use is K-Nearest-Neighbors (KNN). As we have seen in class, this is a non-parametric classification and regression method that is very easy to use since it relies the k neighboring points to make a prediction on the given dataset. The way we setup this algorithm is pretty simple. We imported SKlearn K Neighbors classifier and we fit our training data xtrain and ytrain. We then predicted the ypred and calculated the model accuracy. While the default number of neighbors used by the imported SKlearn package is 5, we decided to tune this parameter in order to find the parameter that makes the most sense in our case using this dataset. This parameter achieve a higher accuracy than the default parameter set in the imported package.

Finally, we decided to implement both a generic neural network and a convolutional neural network. Neural networks are extremely popular for text classification problems due to their flexibility and scalability, and given the size of our dataset and the number of text-generated features we had, incorporating them was only appropriate [8]. To construct, train, tune, and test both types of neural networks, we made extensive use of the PyTorch and Keras packages. The basic architecture of our generic neural net consisted of one input layer, one hidden layer, and one output layer, as more than one hidden layer is generally reserved for extremely complex relationships that aren't really seen here. The input layer had 10,306 dimensions, or equal to the number of features/words in our reduced vocabulary, and the output layer included 10

dimensions, one for each of the possible news categories. For the loss criterion and optimizer, we opted for negative log likelihood loss and the popular Adam optimizer with a learning rate of 0.001. These areas were fixed through all iterations and tuning of the model. A representation of one possible combination of these hyperparameters in neural net architecture form can be seen below:

```
Net(
  (fc1): Sequential(
    (0): Linear(in_features=10306, out_features=128, bias=True)
    (1): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=False)
    (2): ReLU()
  )
  (drop_out): Dropout(p=0.5, inplace=False)
  (output): Linear(in_features=128, out_features=10, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

Figure 4: Neural net architecture for one tuning iteration

Moreover, CNN is a standard deep learning model for NLP, such as text classification and sentiment analysis, which uses a word embedding layer and one-dimensional convolutional layers. It is important to note that CNN requires different data preprocessing steps than other methods, as discussed in the preprocessing section. The basic structure of our CNN consisted of an embedding layer, three sets of convolutional and max pooling layers, a flatten layer, a dropout, and a dense fully connected layer with softmax activation function [10]. We also experimented with having dropout layers of different values right after each max pooling layer and also with batch normalization. For the loss criterion and optimizer for CNN, we chose the sparse categorical cross entropy loss and the Adam optimizer. The embedding layer performs the GloVe embedding for the training data. We used a kernel size of 3, a stride of 1, and ReLu activation for all Conv1D layers. A more in-depth discussion of tuning various hyperparameters, like dropout rate and hidden layer size, and the experimentation process will be addressed in the following section. A representation of one possible combination of these hyperparameters in neural net architecture form can be seen below:

Layer (type)	Output Shape	Param #
input_9 (InputLayer)	[(None, 50)]	0
embedding (Embedding)	(None, 50, 100)	8563400
conv1d_24 (Conv1D)	(None, 50, 64)	19264
max_pooling1d_17 (MaxPooling	(None, 25, 64)	0
conv1d_25 (Conv1D)	(None, 25, 64)	12352
batch_normalization (BatchNo	(None, 25, 64)	256
max_pooling1d_18 (MaxPooling	(None, 12, 64)	0
dropout_15 (Dropout)	(None, 12, 64)	0
conv1d_26 (Conv1D)	(None, 12, 64)	12352
batch_normalization_1 (Batch	(None, 12, 64)	256
flatten_8 (Flatten)	(None, 768)	0
dropout_16 (Dropout)	(None, 768)	0
dense_8 (Dense)	(None, 10)	7690

Figure 5: CNN architecture for one tuning iteration

7 Experimentation & Evaluation

Here we evaluate the results of our classification algorithms based on their accuracies on the test set. As a reminder, the training and validation sets compose 90% of our final dataset (99,650 observations) and the testing set is 10% of our final dataset (11,073 observations). In-class accuracies of the best performing method will be discussed in the final section, as the patterns are similar across all classifiers.

7.1 Naive Bayes

We used the traditional Laplace smoothing parameter of 1, as commonly seen in Naive Bayes for most text classification tasks. There was only one hyperparameter to tune: the presence of class priors or not. The class priors would be how often each class appeared in the entire datset of 110,723 observations. Instead of performing k-fold CV on whether or not we should include class priors, which seems unnecessary in this particular case to tune two possibilities of just one hyperparameter, we simply ran the model twice and found each model's test accuracy. The results are shown below:

Model	Training Accuracy (%)	Test Accuracy (%)
No Class Prior	80.47	76.86
Class Prior	78.00	76.00

Table 1: Training and Test Accuracies for Naive Bayes

Interestingly enough, Naive Bayes performed better with no class prior. An accuracy close to 77% was relatively high and established a higher reference point than our simple majority classifier for other models to beat.

7.2 Logistic Regression

We trained logistic regression models for multinomial classification. The hyperparameter we chose to tune over for logistic regression was C, which is the inverse of regularization strength. We only considered the L_2 penalty in logistic regression. As our data is already a sparse matrix, we believe that we should consider shrinking the parameters rather than zeroing them out. Training on L_1 or elasticnet penalty for base models also took much longer than the given L_2 penalty built into sklearn. We also decided to use the saga solver, as it performed the most efficiently for our sparse and big dataset.

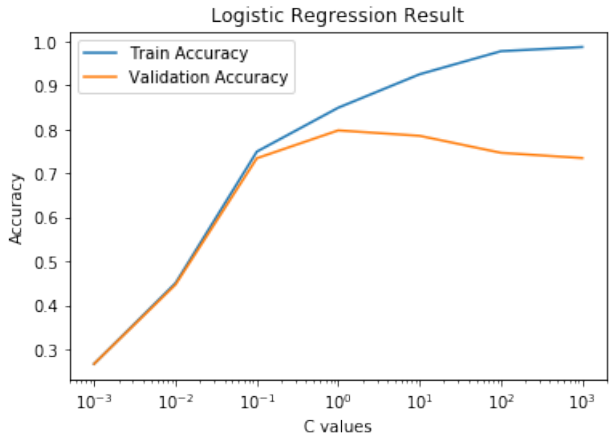

Figure 6: Effect of C on Logistic Regression

From the plot of the effect of C on logistic regression accuracy, we saw that $C = 1$ gave the highest validation accuracy. We tested this model with $C = 1$ on the test data: the training accuracy was 84.93% the test accuracy was 80.01%.

Overall, logistic regression gave quite a high accuracy for our classification task. The reason could be that the data has an underlying linear relationship. The training and test accuracies are very close to those of the linear SVM model, which we will discuss in the next section... Logistic regression is also popular for providing insights about the importance of each feature. However, as our data is a count/TF-IDF matrix, we do not believe that those information would be relevant to our classification task. The per class accuracies are in line with other models.

Hyperparameter	**Training Accuracy (%)**	**Test Accuracy (%)**
C = 1	84.93	80.01

Table 2: Final Logistic Regression Training and Test Accuracies

7.3 Linear SVM / Modified Huber SVM

The SVM algorithm runs to find a hyperplane in an N-dimensional space that distinctly classifies the data points, where N is the number of features. We started with the linear SVM model, which is a parametric model, using Sklearn's SGD classifier package. To tune the model, we ran a 5-fold cross-validation (CV) grid-search over the following hyperparameters:

Tuned	**List of Values**
Loss	[hinge, squared hinge, modified huber]
Penalty	[l2, l1, elastic-net]
Alpha	[1e-5, 1e-4, 1e-3]
Class Weight	[balanced, None]

Fixed	**List of Values**
Max Iterations	100
Learning Rate	optimal
Shuffle	False

Table 3: Hyperparameters for Linear SVM

'Loss' is the loss function that is used. Hinge loss is a convex function that penalizes predictions $y < 1$ linearly, forming the linear SVM model. Squared hinge is hinge but quadratically penalizes $y < 1$ predictions. Huber loss modifies squared hinge loss such that it focuses less on getting outliers correct, by switching from squared to linear loss past a threshold/distance of epsilon. The modified huber loss modifies the huber loss slightly and will be explained in more depth.

Penalty is the regularization term to help the model from overfitting: L2 is Lasso, L1, is Ridge, and elastic-net is a combination of both Lasso and Ridge.

Alpha has two purposes. Firstly, it is a constant that is multiplied to the regularization term so, the higher our alpha, the higher the regularization. Secondly, it is one of the parameters to our learning rate: $1.0/(alpha * (t + t_0))$. The larger alpha is, the lower the learning rate.

Class Weight determines the weights given to each of the 10 classes. 'Balanced' adjusts the weights of each class such that it is inversely proportional to how frequent that class occurs. This can help with unbalanced datasets, such that the dominant class(es) don't necessarily dominate. 'None' means that all classes have a weight of one.

We chose max iterations of 100 as there is minimal difference between max iterations of 50 versus 100 and we also wanted to minimize runtime. The optimal learning rate has the formula of $1.0/(alpha * (t + t_0))$. Shuffle being False means that we do not shuffle the training data after each epoch.

The CV test accuracies from the different hyperparameter combinations were quite similar, ranging from around $75 - 80\%$. The CV training and test accuracies of 4 of the many combinations that we ran with gridsearch are shown on the top of the next page in Table 4 (in this table, penalty is fixed as l2 and alpha is fixed as 1e-4).

Optimal Model: From tuning with gridsearch, we found the optimal hyperparameters to be: modified huber (loss), L2 (regularization), 1e-4 (alpha), and None (class weight). As shown from the above table, this has the highest cross-validation test accuracy. **The final accuracy** of this model on our test set is **80.14%**.

Linear SVM Model	CV Training Accuracy (%)	CV Test Accuracy (%)
Hinge, Balanced	82.58	78.43
Hinge, None	81.97	78.62
Modified Huber, Balanced	86.40	79.43
Modified Huber, None	**85.53**	**80.02**

Table 4: Cross-Validation Training and Test Accuracies for 4 Hyperparameter Combinations

The multi-class confusion matrix for the optimal linear SVM is shown below. The diagonal shows the per class test accuracy of each class. This accuracy means, among all the articles/datapoints that are actually labelled class-k, how many were predicted as class-k? From this matrix, it seems like many of the actual healthy living articles were predicted as a wellness article, and many actual business articles were predicted as politics articles, which caused the low accuracies in the classes business and healthy living. The ROC curve, by plotting true positive rates (TPR) with false positive rates (FPR), is a performance measurement for our classification problem at various thresholds settings.

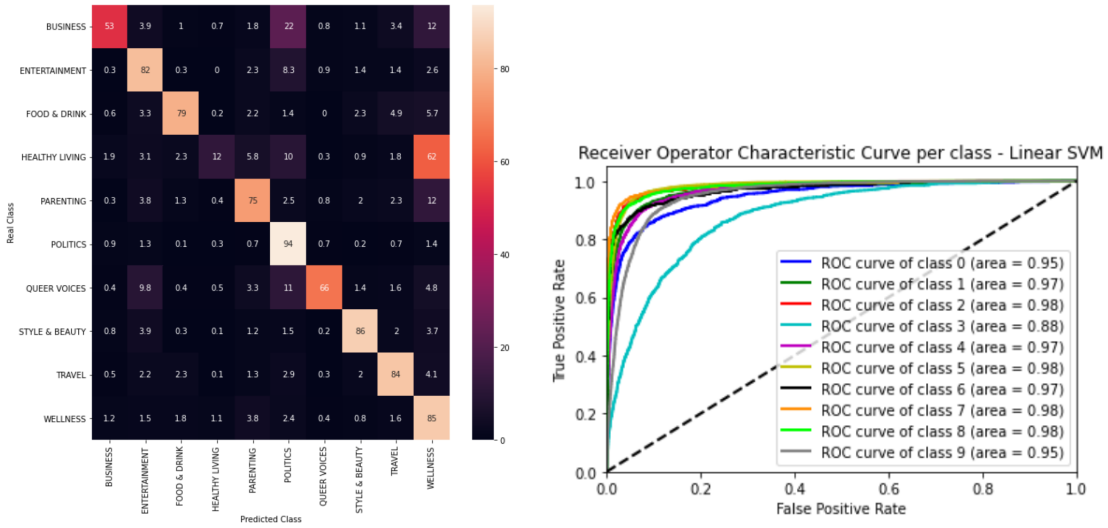

Figure 7: Confusion Matrix and ROC Curve per class - Best Linear SVM

In our optimal linear SVM model, we used the Modified Huber loss[7] instead of the Hinge loss. The Modified Huber loss is defined as[12]: $L = max(0, 1 - yf(x))^2$ for $yf(x) \geq -1$ and $-4yf(x)$ otherwise. As $max(0, 1 - yf(x))$ is hinge loss, this loss modified the squared hinge loss to focus less on getting outliers correct by switching from squared to linear loss past a threshold, as demonstrated by $-4yf(x)$ otherwise. It is less penalizing towards incorrect classifications and outliers compared to the squared hinge loss but more penalizing than the regular hinge loss.

Thus, our optimal "linear SVM" model is not actually linear but is a model with a piecewise nonlinear loss function. The final optimal model is a SVM model with a modified Huber loss.

7.4 Gaussian Kernel SVM

We also ran an Radial Basis Function Kernel SVM model to compare with our modified Huber SVM model. We thought that the Gaussian Kernel SVM would produce better results, as it is more complex and deals with higher dimensions. It essentially projects the data into a higher dimensional space where the data becomes linearly separable. However, it can be computationally expensive and may not be optimal for large datasets like ours.

We built the Gaussian Kernel SVM model using Sklearn's Support Vector Classification (SVC) package

from its SVM package. It defaults to Gaussian (RBF) Kernel. We tuned the model by running a 5-fold CV grid-search over the following hyperparameters:

Tuned	List of Values
gamma	[1e-2, 1e-4]
C	[0.1, 10, 50, 100]

Fixed	List of Values
Kernel	RBF
Class Weight	None
Max Iterations	50

Table 5: Hyperparameters for Gaussian Kernel SVM

Gamma is the kernel coefficient. The default gamma which is 'scale' uses $1/(n_{features} * X.var())$ as a value of gamma. Here, I chose to use 1e-02 and 1e-04, both which give higher CV test accuracies than 'scale'. C is the regularization paramter. It is inversely proportional to the strength of the squared l2 penalty. Like linear SVM, class weight of 'Balanced' versus None was also a hyperparameter we could tune but None gave us a higher test accuracy when running our models initially so we fixed it as None. We used max iterations of 50 for each model in grid-search because we had limited computational power.

The CV training and test accuracies of 4 of the above combinations that we ran with gridsearch are shown:

Gaussian Kernel Model	CV Training Accuracy (%)	CV Test Accuracy (%)
gamma=1e-02, C=10	**52.94**	**49.85**
gamma=1e-02, C=50	49.32	47.92
gamma=1e-04, C=10	48.73	48.18
gamma=1e-04, C=50	49.90	48.19

Table 6: Cross-Validation Training and Test Accuracies for 4 Hyperparameter Combinations

Optimal Model: From tuning with gridsearch, we found the optimal hyperparameters to be: C=10 and gamma=1e-02. As shown from the above table, this has the highest cross-validation test accuracy. Using these hyperparameters, we ran the Gaussian Kernel SVM model with 1000 maximum iterations. This gave us a **final test accuracy** of **74.29%**.

Compared to the results from the modified Huber SVM, the Kernel SVM model does not perform as well, with a lower final test accuracy. From the confusion matrix of Kernel SVM, which is shown in the notebook, the Kernel SVM model has the 2 classes that perform the best and the same 2 classes that perform the worst. Their misclassifications between the classes are also similar, but Kernel SVM model makes more mistakes overall.

While, originally, we thought that Kernel SVM would perform better than the modified Huber SVM, possible reasons why this might not be the case are, firstly, that the Kernel SVM model could not reach its full potential. While running our final model on our test set, we already set max iterations to 1000, which already reached out computational limit. We could not increase max iterations any magnitudes higher but we noticed that the final test accuracy increased significantly from 50% to 70% when max iterations increased from 100 to 1000. Thus, it's possible that if we a much higher number of iterations, that the test accuracy could have surpassed that of modified Huber SVM. In addition, we also had the computational capacity to tune our hyperparameters a lot more for modified Huber SVM than our Kernel SVM model, allowing our linear SVM model to perform better. The hyperparameters for our Kernel SVM model may not have been optimal. Sklearn also states that the fit time scales "at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples". Thus, ultimately, it was not computationally optimal for us to fit a Kernel SVM on our large dataset with 99,650 observations, although it theoretically could have generated better results.

7.5 K-Nearest Neighbors

In the KNN model, we are making predictions based a similarity measure between points and their neighbors. An common way of classifying points using KNN is calculating distances with neighboring points. The main parameter that affects how we implemented our model and its accuracy is the "K" number which is the number of neighboring points. As we learned in class, having a low K results in the closest points have a lot of impact on the outcome (even if they might be just noise) and having a very large value for K becomes computationally intensive and might result in lower model accuracies. In our case, we tried to tune our model in such a way that increase the testing accuracy while lowering the complexity of the model.

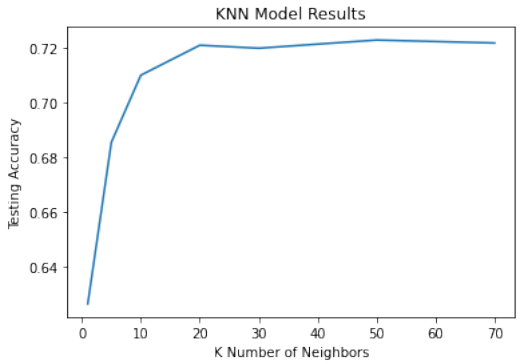

Figure 8: Effect of Number of neighbors on KNN

Based on the plot, it seems like the testing accuracy plateaus at 72% after 20 nearest neighbors. For very large values of K, the accuracy drop below 70%. From 20 to even 70 nearest neighbors, all K numbers result in a very similar accuracy of around 72%. However, the higher the value of K, the more computationaly expensive it becomes to train the data. Because for each point, you have to compute K times the similarity to the point so very large values of K are not ideal. Therefore, we decided to tune the model to fit K = 20 because it still resulted in the best testing accuracy. Here is the training accuracy and test accuracy of the model fitted using the parameter 20.

KNN Model	**Training Accuracy (%)**	**Test Accuracy (%)**
N of neighbors = 20	75.58	72.09

Table 7: Training and Test Accuracies

7.6 Traditional Neural Network

Below is a list of the hyperparameters we selected. Refer to Figure 4 and the "Methods" section above for the overarching model architecture.

Tuned	**List of Values**
Hidden Layer Size	[64, 128, 256, 512]
Dropout Rate	[0.1, 0.3, 0.5, 0.7, 0.8]

Fixed	**List of Values**
Batch Size	50
Epochs	5
Optimizer	Adam, learning rate = 0.001
Loss Function	Negative Log Likelihood Loss
Activation Function	ReLU
Batch Normalization	Yes

Table 8: Hyperparameters for Traditional NN

Mini-batch learning was used for the neural network training process, with each batch containing 50

samples from the training data. The training accuracy and loss were computed for each batch and printed every 40 batches. During the process, the average accuracies and losses for all batches in each epoch were computed and used to display the neural net learning curve, as seen below:

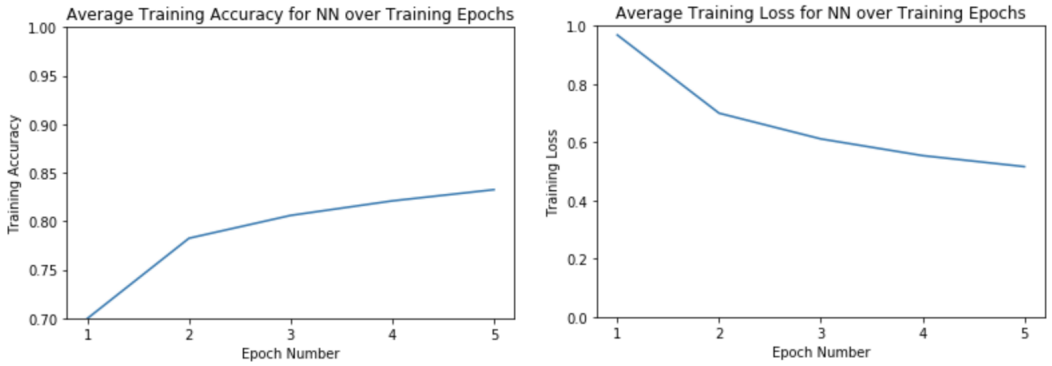

Figure 9: Neural Net (hidden layer size = 128, dropout rate = 0.7) Training Performance over 5 Epochs

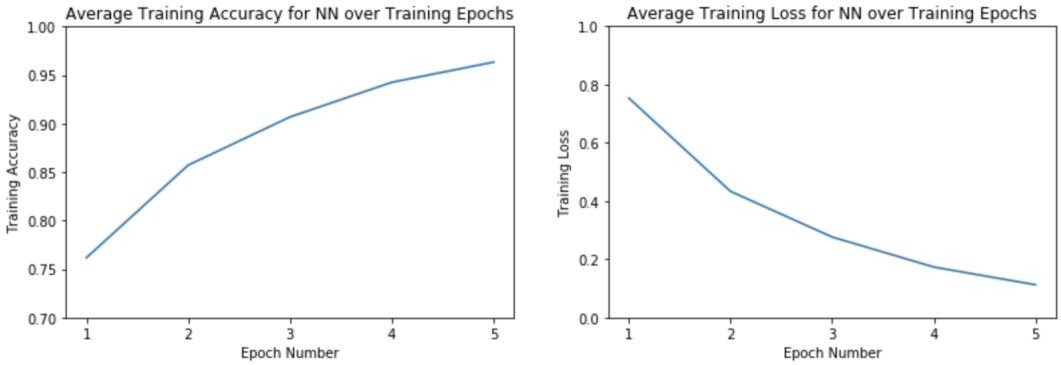

Figure 10: Neural Net (hidden layer size = 256, dropout rate = 0.3) Training Performance over 5 Epochs

From the Figures 9 and 10, it's clear that the accuracy increases and loss decreases are beginning to plateau after epoch 5. To test this hypothesis empirically through trial and error, we tried out higher epoch values of 10 and 20 as well. There, although the training losses were still decreasing slightly and the training accuracies were still increasing slightly, the test accuracies (described in the next paragraph and table) began to slowly decrease. This is pretty overt evidence of overfitting, so we decided that 5 epochs would indeed be best to mitigate this effect. Another thing to note that's also reflected in the figures above is that the training performance of higher dropout rates and lower hidden layer sizes were significantly lower than those of higher hidden layer sizes and lower dropout rates. However, as we'll see in the next section, this pattern that we see is mostly due to overfitting to the training set once again, and not necessarily a reliable indicator for model efficacy.

Using the fixed hyperparameter structure defined earlier, we tuned network with by varying hidden layer size and dropout rate. This was mostly done through a trial and error process again with the testing set (rather than computing cross-validation, which has been cited by numerous sources including Professor Ungar as too expensive for neural nets) and prior knowledge of what optimal values could be. These testing accuracies, computed across all 10,000 samples in the testing set, for each hidden layer size and dropout rate are shown in Table 9 on the next page.

Based on Table 9, it's pretty clear that larger dropout rates (0.7 and 0.8, particularly) yield the best test results. Afterwards, when examining a certain model's testing accuracy in tandem with their corresponding training loss and accuracy graphs similar to the ones displayed above, we noticed that low dropout rates

Layer Size, Dropout Rate	Test Accuracy (%)
64, 0.3	78.53
64, 0.5	79.40
64, 0.7	80.11
64, 0.8	79.13
128, 0.3	78.24
128, 0.5	78.53
128, 0.7	**80.73**
128, 0.8	80.48
256, 0.3	77.80
256, 0.5	78.97
256, 0.7	80.16
256, 0.8	80.38
512, 0.3	77.51
512, 0.5	77.71
512, 0.7	79.86
512, 0.8	80.62

Table 9: Testing accuracies for various hyperparameter values

corresponded with extremely high training accuracies and low losses at early epochs - however, when generalizing to the test set, they performed slightly worse relative to networks with higher dropout rates. Clearly, as stated before, low dropout rates tend to induce overfitting. Unlike dropout rate however, hidden layer size didn't seem to have an immediately obvious effect on test accuracy; holding dropout rate constant, the models with hidden layer sizes 64-512 performed pretty similarly. For the purposes of model comparison, we finalized our neural network as the one with the highest test accuracy - layer size: 128 and dropout rate: 0.7. In-class accuracies for this model for each of the 10 possible news categories will also be shown later.

7.7 Convolutional Neural Network

Below is a list of the hyperparameters we selected. Refer to the Figure and "Methods" section above for the overarching model architecture.

Tuned	List of Values
Hidden Layer Size	[32, 64, 128]
Dropout Rate	[None, 0.1, 0.3, 0.5, 0.7]
Batch Normalization	[Yes, No]

Fixed	List of Values
Kernel Size	3
Max Pooling Size	2
Batch Size	128
Epochs	10
Optimizer	Adam, learning rate = 0.001
Loss Function	Sparse Categorical Cross Entropy
Activation Function	ReLU, Softmax

Table 10: Hyperparameters for CNN

Similar to traditional neural network, the average accuracies and loss for all batches in each epoch were computed as the learning curve, as seen on the next page below.

The accuracy increases and the loss decreases steeply at first. Epoch 10 seem to be a reasonable point that both graphs plateau. Using the fixed hyperparameter structure defined earlier, we tuned network with by varying hidden layer size, dropout rate, and batch normalization. Similarly, this was mostly done through

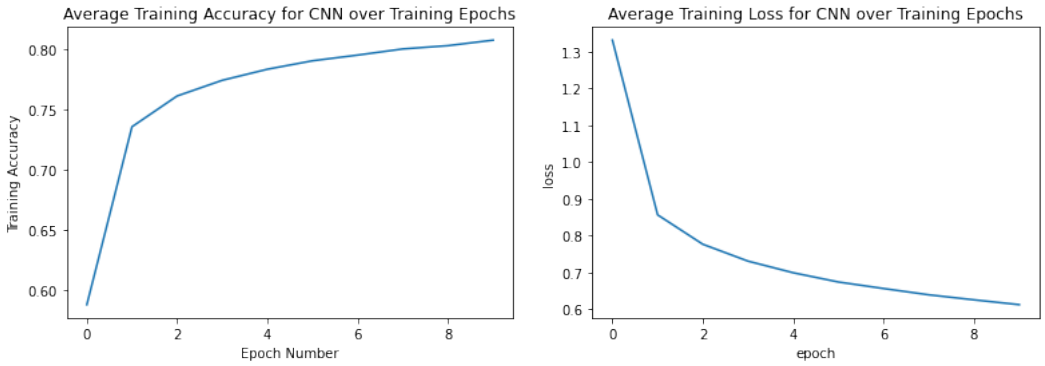

Figure 11: CNN (hidden layer size = 64, dropout rate = 0.5, with Batch Normalization) Training Performance over 10 Epochs

a trial and error process. These testing accuracies, computed across all the testing set, for some selected hyperparameters are shown in Table 12 below:

Layer Size, Dropout Rate, Batch Normalization	**Test Accuracy (%)**
32, (0, 0, 0.5), No	79.22
64, (0, 0, 0.5), No	79.27
64, (0, 0.5, 0.5), No	80.05
64, (0.5, 0.5, 0.5), No	78.51
64, (0, 0.1, 0.1), No	79.77
64, (0, 0.3, 0.3), No	79.86
64, (0, 0.7, 0.7), No	79.15
64, (0, 0.5, 0.5), Yes	**80.14**
64, (0, 0.3, 0.3), Yes	78.82
64, (0, 0.7, 0.7), Yes	79.90
128, (0, 0.5, 0.5), Yes	79.89

Table 11: Testing accuracies for various hyperparameter values

Based on our experiment and the results on the table, the dropout rate of 0.5 yield the best test results so far. When examining the epoch history, we saw a general trend that low dropout rates corresponded to high training accuracies and low losses, but the test accuracies were significantly lower than the training's. Meanwhile, high dropout rate of 0.7 in this case led to a slow increase in training accuracy and also resulted in a similarly low test accuracy. Similar to our previous findings, low dropout rates induced overfitting and high dropout rates induced underfitting. After experimenting with the number of dropout layers, we also found that having 2 layers gave the best test result. Overall, the test accuracy shown here are relatively similar, ranging from around 78 to 80. We could only see minor effect of increasing layer size or batch normalization on the test accuracy. For the purpose of model comparison, we finalized our CNN as the one with the highest test accuracy - layer size: 64, dropout rate: (0, 0.5, 0.5), and with batch normalization.

8 Discussion & Conclusion

Using the data from Kaggle's News Category Dataset, we performed EDA and ran machine learning models to classify every article as one of the ten news categories. A summary of the final testing accuracies for each tuned model is shown in Table 12. The model with the highest accuracy is Neural Network, with a final test accuracy of 80.73%, although Logistic Regression, the Modified Huber SVM, and CNN produced very similar final test accuracies of around 80%. NN performs well with large datasets like our dataset, as it has the flexibility to fit highly complex data and capture patterns in the data that other models may be too simple to capture. At the same time, the fact that logistic regression produced a high test accuracy

Model	Test Accuracy (%)
Majority Class (Baseline)	26.71
Naive Bayes	76.86
Logistic Regression	80.01
Modified Huber SVM	80.14
Gaussian Kernel SVM	74.29
KNN	72.09
Neural Network	80.73
CNN	80.14

Table 12: Final Testing Accuracies for All Tuned Models

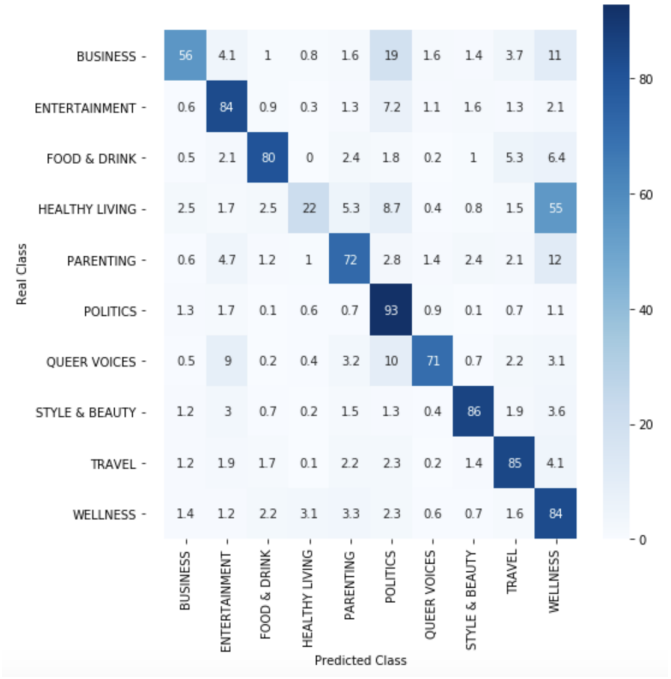

Figure 12: Test Confusion Matrix for Final Tuned Neural Network

could suggest that there are signs of linearity in the data. And support vector machines are generally good supervised classification models - the Modified Huber SVM performed the second best after NN.

CNN also gave the second best test accuracy. Instead of fitting the tf-idf matrix, CNN learns from sequences of words with GloVe embedding. Thus, CNN is able to capture word sequence and sentence structures in the articles that other methods cannot. However, while we have tuned many hyperparameter combinations for CNN, there are still many possibilities to improve its performance. For example, one can use other embedding methods, create multi-channel CNN, or add more layers to it. Moving forward, with a deeper understanding of CNN, the structure/hyperparameters of CNN could be further optimized to produce even better results.

On the other hand, models that gave the lowest test accuracies beside the baseline majority class model are Gaussian Kernel SVM and Random Forest. The commonality between these two models is that they are computationally too expensive to run on our large dataset with over 99,000 observations. Thus, these two models were difficult to tune, leading to non-optimal hyperparameters. For example, we could not increase the max iterations for Gassian SVM past 1000. Computational power put a ceiling for these models' test accuracy.

The confusion matrix for NN is shown in figure 10. The diagonal shows the per-class accuracy for each of the 10 classes. The top two performing categories are Politics and Style and Beauty with accuracies of 93%

and 84% respectively. This could be because these two categories both have a set of words that are highly unique to them, making them highly identifiable. For example, from the word cloud of Politics, we see that it has unique words such as 'Trump', 'Donald', 'president', and 'Republican'. Additionally, as Politics is the majority class, its prediction accuracy could be slightly inflated.

In comparison, the two lowest performing categories are Business and Healthy Living with accuracies of 56% and 22%. This is not surprising for Business as there could be Business-related topics for many other categories, causing overlap. As for Healthy Living, many of its articles have been classified as Wellness, shown by the 55 in the right column. Although we originally thought that Healthy Living could be more specific to food and exercise (physical), while Wellness refers more to general mental and physical well-being, our results say otherwise. Words such as "life" and "health" were two big common words between these two categories. The confusion matrix of Modified Huber SVM show that Modified Huber SVM has the same best-performing and worst-performing categories. We also realised that this is a commonality among our models. Thus, next time, we could perform more robust data exploration and preprocessing to obtain more distinct class labels, to ultimately produce higher accuracies.

Future Scope of Work Summary:

- Improving our data preprocessing to obtain more distinct class labels
- Improve handling of imbalanced data by potentially setting hyperparameters etc. that make classifications less imbalanced. What we see in our final confusion matrix is that, although Politics has a high accuracy, there were various other categories such as Business, Queer Voices, and Healthy Living that were often predicted as Politics. Since Politics is the majority class, our models tend to predict it even more often.
- Improve the structure of CNN: With a deeper understanding of CNN, we could explore other embedding methods, create a multi-channel CNN, and/or add more layers to improve our CNN model.
- Implement other more complex models such as Recurrent Neural Networks (RNN), including Long Short Term Memory networks (LSTM). Among our models, only CNN considered word sequences and structures, which models like RNNs captures. Thus, when optimally tuned, RNNs could learn more information from the texts and produce better results.
  - For example, LSTM networks are well-suited to learning the series of data and is popular for classification tasks. LSTM networks are very good at holding both long term and short term memories. That is, the prediction of nth sample in a sequence of test samples can be influenced by an input that was given many time steps before. As sequences of words would have long term dependencies in it, we believe that LSTM would be an interesting future model to consider.

References

[1] Imdevskp Devakumar. News category classification, Jun 2020.

[2] Foolofatook. News classification using bert, Jul 2020.

[3] Salimbeni Guido. Multinomial naive bayes, 2019.

[4] Hengzheng. News category classifier (val acc 0.65), Jul 2018.

[5] Hsankesara. News classification using han, Dec 2018.

[6] Jonathan Hui. Nlp - word embedding & glove, Jul 2020.

[7] Masayuki Karasuyama. Nonlinear regularization path for the modified huber loss support vector machine, Oct 2010.

[8] Minaee. Deep learning based text classifcation, Nov 2020.

[9] Rishabh Misra. News category dataset, Dec 2018.

[10] Sumit Saha. A comprehensive guide to convolutional neural networks, Dec 2018.

[11] Sadangi Siddhant. Classification using linearsvc ($\text{val}_a cc$ 64%), 2019.

[12] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms, Jul 2004.