Machine Learning Engineer Nanodegree

Capstone Report

Aditya Tandon

I. Definition

Project Overview

Wikipedia defines sentiment analysis as the process that "aims to determine the attitude of a speaker or a writer with respect to some topic". Automated Sentiment Analysis is the process of training a computer to identify sentiment within content through Natural Language Processing (NLP). According to Mechanical Turk, automated analysis can be almost as good as human analysis (or, "as good as it gets"). Analysing 20 documents can be done easily by reading it and not much of human effort would be required. But analysing 50,000 documents will require a lot of effort and labour charges. And by using Automated Sentiment analysis, it can be completed within few minutes. Here is the paper on which machine learning has been applied for the sentiment analysis of the tweets on twitter.

In this project, I aim to train a Machine Learning model, which analyses the sentiments from the Rotten Tomatoes movie dataset by differentiating sarcasm, terseness, language ambiguity and many others using NLP and classification algorithm.

Problem Statement

The goal is to implement and train a classifier that can predict the likelihood of the type of sentiment (multi-class classification) by using Natural Language Processing and Machine Learning Methods. The tasks involved are the following:

1. Load the Data.
2. Train a benchmark model on the data without any kind of preprocessing.
3. Data exploration and visualization. Use insights obtained from this step for data preprocessing.
4. Data preprocessing and Feature Engineering.
5. Trying out various ML algorithms.

6. Picking up the best performer and refining it.
7. Evaluating the final algorithm.

The final application is expected to be useful for determining the sentiments of the movie reviews in just a few seconds. Therefore, this project can be useful building later projects like building an automation sentiment analysis for Food Reviews or Analysis of Twitter tweets.

Metrics

Since this is a classification problem, I am using accuracy as a metric. Accuracy can be defined as following: -

$$\text{Accuracy} = \frac{Total\ number\ of\ correct\ classifications}{Total\ number\ of\ instances}$$

In the neural networks, I will use categorical cross-entropy as a loss function since this is a multi-class classification problem.

Since, this is a classification problem, and our goal is to maximize the number of correct labels to given to the reviews, Ii believe that accuracy is a valid metric for given use case.

II. Analysis

Data Exploration

The Rotten Tomatoes dataset is provided as tab separated (.tsv) file. There are 4 columns in this data: -

1. ***PhraseId*** – The ID of each phrase. Works like an index or Serial number.
2. ***SentenceId*** – The ID of the parent sentence. Multiple phrases are generated from one sentence.
3. ***Phrase –*** The actual text/phrase data.
4. ***Sentiment –*** The target variable. Sentiment has 5 possible values which mean -

0	Negative
1	Somewhat Negative
2	Neutral
3	Somewhat Positive
4	Positive

Total data points: 156,060

Total number of Parent sentences (unique ***SentenceId*** values): 8529

Sentiment wise number of data points

Negative	7072

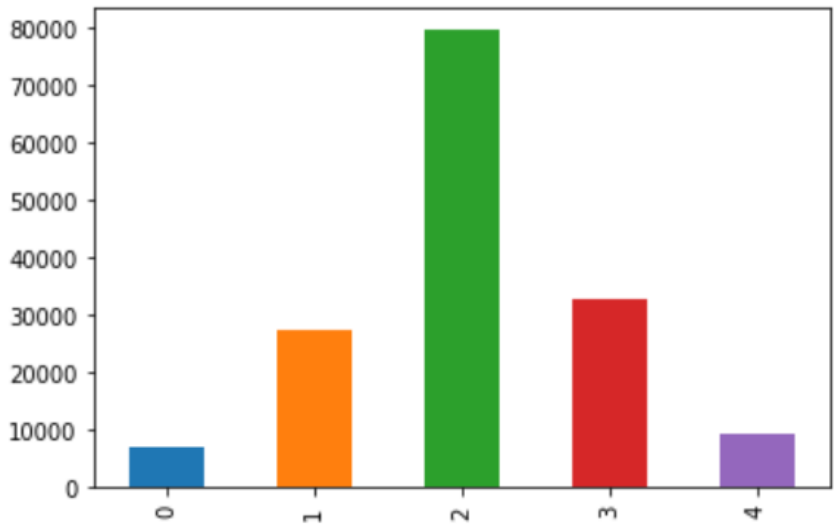
Somewhat Negative	27273
Neutral	79582
Somewhat Positive	32927
Positive	9206

We can see that actual number of positive and negative reviews are quite less.

Length of longest review = 53, Length of shortest review = 0

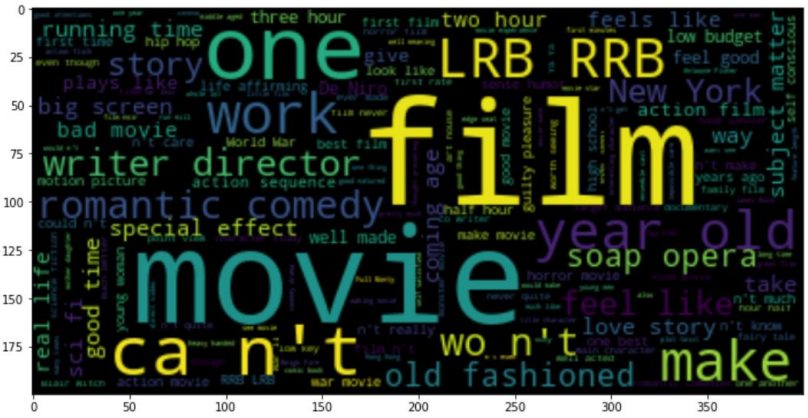
Exploratory Visualization

Distribution of data points among sentiments: -

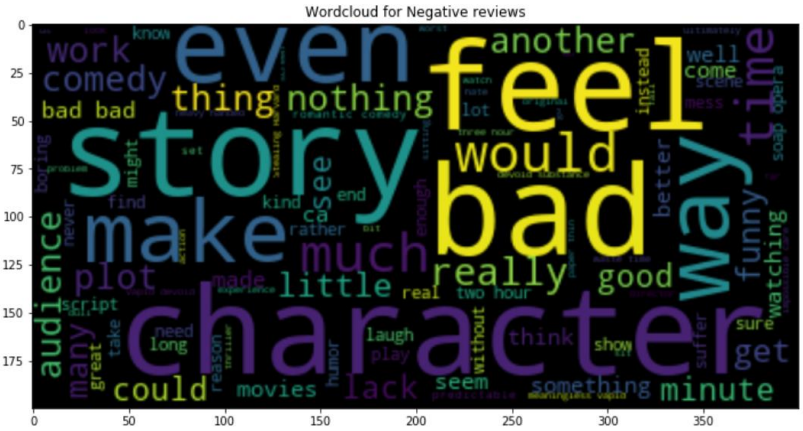


I used the ***wordcloud*** library to display top 100 words for each sentiment.

1. Entire text (All words).



2. Negative Reviews

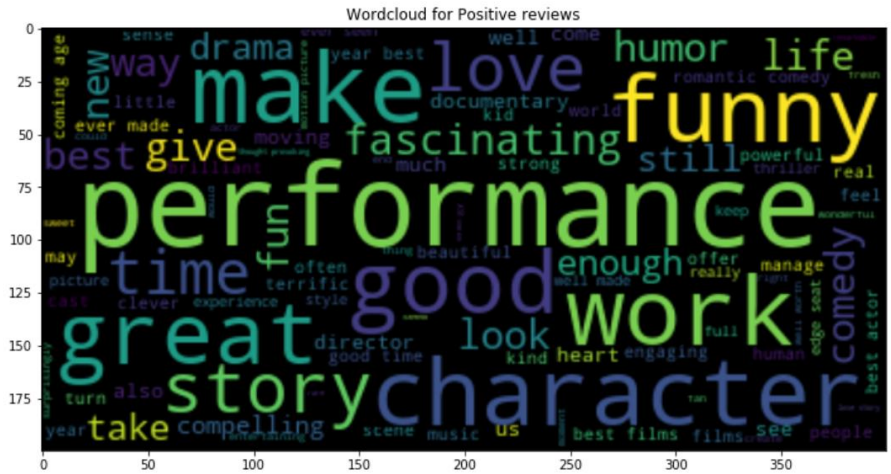


3. Somewhat Negative Reviews

4. Neutral Reviews

5. Somewhat Positive Reviews

6. Positive Reviews



Observations: -

Top Words:

Negative	film, movie, one, LRB, RRB
Somewhat Negative	feel, bad, story, would, way
Neutral	much, story, make, character, work
Somewhat Positive	story, good, make, time, work
Positive	performance, work, good, funny, great

- It can be observed that the most occurring words **do** correspond with the sentiment text.
- On a rough check, data quality seems good.
- Hence, the Machine Learning models will have something to pick up on.

Algorithms and Techniques

For this task I have chosen following 4 algorithms: -

1. **Gradient Boosting Classifier:** Boosting is a type of ensemble learning.
    a. **Ensemble Learning:** Ensemble learning is a method in which multiple models are combined and their mean predictions are considered. This works well in practice because multiple models are trying to predict the target variable in their own way.
    b. **Boosting:** This a type of Ensemble learning in which multiple classifiers are trained sequentially. Each new model tries to fit on the part of data which previous models were unable to. This method reduces both bias and

variance. Although, it is possible to overfit the data with this algorithm, we can control the overfitting through regularization and cross validation.

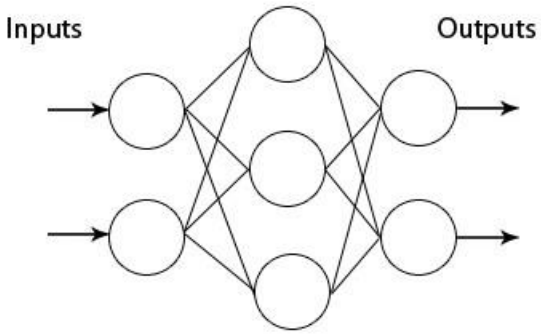
c. **Gradient Boosting:** Gradient boosting is a boosting method which usually uses an ensemble of Decision Trees as weak learner. A weak learner is a model whose predictions are just better than random guessing.

I think this algorithm is useful for given problem because of the following reasons: -

i. It produces multiple models. Each model will understand some part of the reviews that other models won't, giving an overall better accuracy.

ii. It doesn't underfit. Gradient boosting tends to overfit the data. But that can always be controlled using regularization. It is better than to have a model with high bias that is unable to fit the data.

2. **Multi-layer Perceptron (Shallow Neural network)**:



Source: Google Image Search

A multi-layer perceptron (MLP for short) is a type of neural network in which all neurons of a layer are connected to both previous and next except for input and output layers. Every connection between two neurons is called a **parameter** or **weight**. An MLP has at-least 2 layers (1 input layer and 1 output layer). It can optionally have 1 or more hidden layers.

Training an MLP: -

a. The input data flows from Input layer and predictions are generated at the Output layer. This is called **Forward pass**.

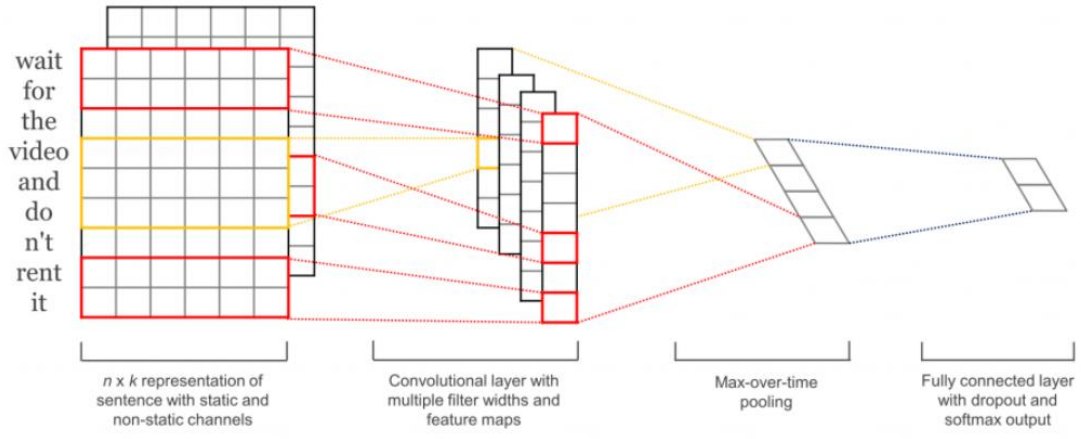
b. The predictions are compared to original labels and error is calculated. The gradient of this errors is passed back with respect to each **weight** at each layer. This process is called **Backpropagation** and this how the Neural

network learns to find patterns in the data. All the learning is in the form of final values of all the **weights** of the Neural network.

I have chosen this algorithm since Multi-layer perceptron can work as universal function predictors. If some kind mathematical function exists between the input format of text data and the labels, an MLP should be able to achieve good accuracy after sufficient training.

3. **Convolutional Neural network:** Convolutional Neural Networks are a variant of Neural networks that are generally used in Computer Vision tasks. They work much better than MLPs on Image data because they focus more on local similarity as compared to global similarities in an image. This results in reduction in number of **parameters** and higher accuracy because in Images, nearby pixels are more similar than far away pixels.
   Although used mostly for images, they perform well on NLP tasks as well. For more information, you can check out the below post about how Facebook used CNN for Neural machine translation.
   Source: https://code.facebook.com/posts/1978007565818999/a-novel-approach-to-neural-machine-translation/

   My theory is that similar to images, local words in a sentence tend to point toward similar sentiment.

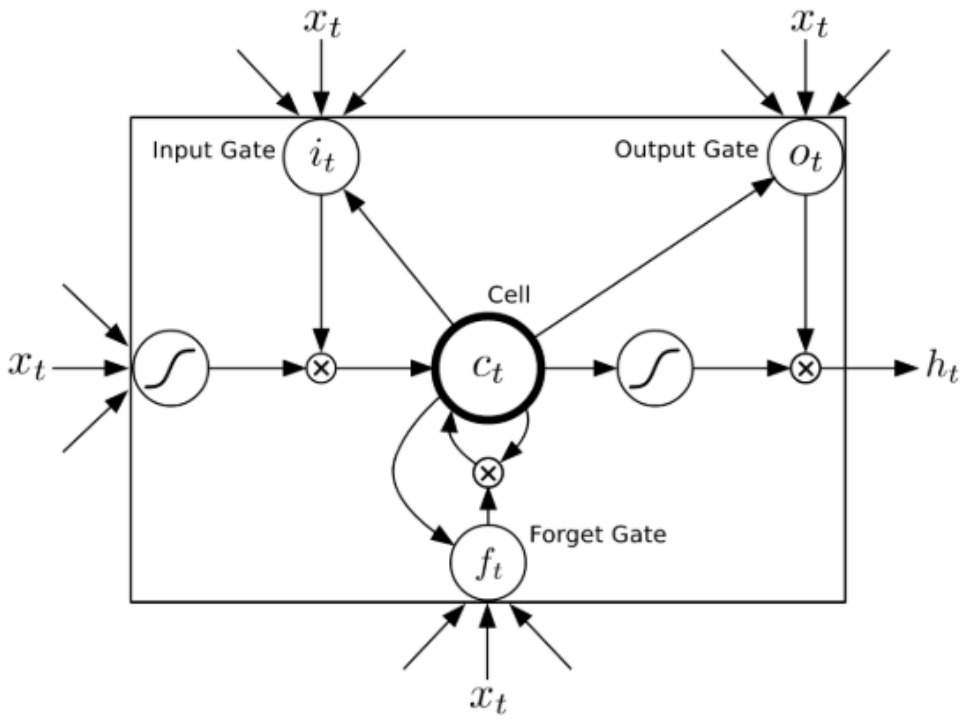


Source: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

Therefore, I am testing CNN on my data for following reasons:

i. Their ability for local feature extraction.
ii. It has been already established by Facebook that CNNs work well for NLP tasks.
iii. To compare the performance of CNN against LSTM, which are considered state of the art for Sequential data like text.

4. **<u>LSTM (Long-Short Term Memory) Neural network:</u>** LSTM is a variant of RNN (Recurrent Neural Network). Recurrent Neural Networks were designed with memory in mind. Many types of data like text, time series occur as a sequence of dependent data points. Analysing data points individually causes data loss. RNNs have a feedback loop which helps them maintain memory over time. This is extremely important for analysing text sentences.
   LSTM is a variant of RNN for remembering long temporal sequences.



As we can see from the architecture, an LSTM cell has an Input gate, Output gate and a Forget gate. The Forget gate decides which data to keep and which to discard. This proves very useful for working with long sequences like movie

reviews. Therefore, for following reasons, I am including LSTM in my list of chosen algorithms: -

i. Designed specifically for sequential data.
ii. Every LSTM cell maintains a hidden state which works as memory.
iii. LSTM can discard data when it is no longer necessary. I think this would help in discarding words in previous sentences in multi-sentence reviews which no longer offer any meaning to current context.
iv. They already been used in several NLP applications and proven to be state of the art for handling text data. For further studies, one can refer to the following excellent article by Andrej Karpathy: -
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Benchmark

For Benchmark model, I am going to use the Naïve Bayes Classifier. Probabilistic models like Naïve Bayes have been used for NLP tasks for a long time. Also, Naïve Bayes is easy to understand and simple to implement. It will also provide a benchmark against how modern methods have evolved to give higher accuracy. For these reasons, I think Naïve Bayes is the best choice for a Benchmark model.

Benchmark Results: -
Number of training instances = 124,848
Number of test instances = 31,212
Input format = Count Vector with max 64 features
Time taken to fit = **0.22** seconds
Test accuracy = **44.33%**

III. Methodology

Data Preprocessing

For Data preprocessing, the major step was removing the stopwords like “a”, “an”, “the”, “I”, “could”, etc. which harshly affect the performance of frequency-based feature vectors. This is because that these words occur in a large frequency in almost all types of text data but don’t provide any important information related to sentiment.
I used the stopwords available from **nltk** package in Python and added punctuation and some other common words that I saw in the Wordcloud (Exploratory Visualization) for all text to create my custom set of stopwords.
After removing these stopwords, I dropped all the reviews with length 0 and recreated the text data from remaining words.
The number of records reduced from 156,060 to 154,726. This is only an 0.85% loss of data from original dataset and I believe that it won’t affect the classifier accuracy in major way.

I split the data in training and testing sets with 20% of data in the test set.
Number of Instances in Training set = 123,780
Number of Instances in Test set = 30,946

Feature Engineering:

Next important step was to convert text data into feature vectors that Machine learning models can understand. For this step, I used 3 different types of vectors.

1. **CountVectorizer:** This method converts the collection of reviews into a matrix of token counts such that each row represents a review and each column a token. The corresponding cell shows the frequency of that token in that review. I designed the Count Vectorizer such that it only considers the top 1024 terms across the entire corpus.
2. **TF-IDF (Term Frequency – Inverse Document Frequency):** Count Vectorizer usually yields a sparse matrix. TF-IDF converts it to a normalized representation.

Applying **TfidfVectorizer** is same as applying **CountVectorizer** followed by **TfidfTransformer**. The goal of using TF-IDF is to give more weightage to features that occur less frequently and can be informative. I thresholded Tf-idf to a max of 1024 features.

3. **Sequence of words of same length:** For Deep learning models, the data format is a sequence of tokens. To generate this, I used the **keras** library. Steps:-
    a. Use a tokenizer to generate a word index from given reviews.
    b. Use the tokenizer to convert reviews to a sequence of words.
    c. Pad the sequences such that they are of the same length.

4. **Word2Vec:** Word2Vec are used to generate word embeddings. In this method, shallow neural networks are trained to generate word vectors. These word vectors are used are positioned in vector space such that words that share common contexts in corpus are close together. Word Embeddings can be useful in Deep Neural Networks.
   To improve efficiency, I used pretrained 1 million Word vectors trained on Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset (16B tokens). I loaded the word vectors of all the words that are common with this corpus of the reviews.
   Link:- https://s3-us-west-1.amazonaws.com/fasttext-vectors/wiki-news-300d-1M.vec.zip

Implementation

1. **Gradient Boosting Classifier:** I used the ***GradientBoostingClassifier*** from scikit-learn library. I used the default configuration except for ***random_seed*** parameter for reproducibility. Some of the important parameters are: -
    a. Learning rate = 0.1
    b. Number of estimators (weak learners) = 100
    c. Maximum depth of the tree = 3
2. **Multi-Layer Perceptron:** For this implementation, I used the ***MLPClassifier*** class with ***random_state*** set for reproducibility. Other important parameters are:
    a. Size of hidden layer = 1 layer with 100 units
    b. Activation = ReLU
    c. Solver = Adam

d. Learning rate = 0.001
e. Learning rate decay = None (***learning_rate = "constant"***)
f. Max iterations = 200

3. **<u>Convolutional Neural Network:</u>** I used the **Keras** library with **Tensorflow** backend for implementing the Convolutional Neural Network. I used an embedding layer in the network to utilize my Word2Vec embeddings. Architecture: -

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32)                0
_________________________________________________________________
embedding_1 (Embedding)      (None, 32, 300)           4566600
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 30, 32)            28832
_________________________________________________________________
average_pooling1d_1 (Average (None, 15, 32)            0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 13, 64)            6208
_________________________________________________________________
average_pooling1d_2 (Average (None, 6, 64)             0
_________________________________________________________________
flatten_1 (Flatten)          (None, 384)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               49280
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 645
=================================================================
Total params: 4,651,565
Trainable params: 84,965
Non-trainable params: 4,566,600
_________________________________________________________________
```

It has 1 input layer, 1 embedding layer, 2 (Convolutional + AveragePool) blocks, 1 flatten layer to convert output to 1D Tensor and 2 Fully connected layers. The neural network has 4,651,565 parameters out of which 4,566,600 are Word embeddings and are not supposed to change values during Training, making them non-trainable parameters. The total number of trainable weights (and biases) is 84,965.
Number of training epochs = 5
Batch size = 64

Optimizer = Adam
Loss function = Categorical Cross entropy

4. **LSTM (Recurrent Neural Network):** This architecture too was implemented using the **Keras** library with **Tensorflow** backend. I used the Word2Vec embeddings in this architecture as well.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 32)                0
_________________________________________________________________
embedding_2 (Embedding)      (None, 32, 300)           4566600
_________________________________________________________________
lstm_1 (LSTM)                (None, 128)               219648
_________________________________________________________________
dense_3 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_4 (Dense)              (None, 5)                 325
=================================================================
Total params: 4,794,829
Trainable params: 228,229
Non-trainable params: 4,566,600
_________________________________________________________________
```

The architecture is not too deep. I have used an Input layer, Embedding layer for Word2Vec embeddings, 1 LSTM layer and 2 Fully connected layers. Total number of trainable parameters in this network is 228,229.
Number of training epochs = 5
Batch size = 64
Optimizer = Adam
Loss function = Categorical Cross entropy

Summary of Results

MODEL	TRAINING TIME	TEST ACCURACY
GRADIENT BOOSTING	**Count Vectorizer** = 30.5 seconds	**50.42 %**
	TF-IDF = 30.5 seconds	**50.42 %**
MULTI LAYER PERCEPTRON	**Count Vectorizer** = 21.84 seconds	**46.89 %**
	TF-IDF = 21.94 seconds	**46.89 %**

CNN	19 seconds/epoch	**58.95%**
LSTM	94-95 seconds/epoch	**66.12%**

As we can clearly see, Deep learning models outperform the remaining methods with **LSTM** taking the lead. In fact, it can be observed that Multi-Layer Perceptron is the weakest model here.

Refinement

Due to superior performance, I chose **LSTM** as my final model. I took the following steps to refine the model further: -

1. Increase number of LSTM units from 128 to 256 and add a Dropout with dropout factor of 0.25.
2. Add Batch Normalization after LSTM layer.
3. Add 1 Dense layer with 128 units.
4. Applied Dropout to both Dense layers with a higher dropout for larger layer.
5. Increase number of training epochs from 5 to 50.
   Final Architecture: -

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_11 (InputLayer)        (None, 32)                0
_________________________________________________________________
embedding_11 (Embedding)     (None, 32, 300)           4566600
_________________________________________________________________
spatial_dropout1d_7 (Spatial (None, 32, 300)           0
_________________________________________________________________
lstm_8 (LSTM)                (None, 256)               570368
_________________________________________________________________
batch_normalization_4 (Batch (None, 256)               1024
_________________________________________________________________
dense_22 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_9 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_23 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_10 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_24 (Dense)             (None, 5)                 325
=================================================================
Total params: 5,179,469
Trainable params: 612,357
Non-trainable params: 4,567,112
_________________________________________________________________
```

The number of training parameters increased from 228,229 to 612,357.

Other hyperparameters: -

Loss = Categorical cross entropy
Optimizer = Adam
Batch size = 512

Steps for Reproducing the results and challenges faced: -

1. A very important step is to remove stopwords (Words like “a”, “the”, etc) which occur a lot in the language but are useless for sentiment classification. The **nltk** library provides utility for loading stopwords.
2. Also, we can add our own stopwords after visualizing the data. You can verify from the code that I have added a few more words like “Movie”, “RRB” to the stopwords along with punctuation.
3. In general, it would be a good practice to analyse each type of corpus because different categories of text like Movie Reviews, Tweets, Business articles will have a few stopwords of their own.
4. Drop null reviews (zero length) to prevent feeding Garbage data to the model.
5. Analyzing class distribution is also an important step. If you’re doing cross-validation and the classes are imbalanced, it is advisable to perform **Stratified K-Fold** over normal **K-Fold** so that each subset reflects the class distribution.
6. Use the **random_state** argument when splitting the data in training and test sets and in all the Classifiers, wherever provided for reproducing the same results.
7. Keep the maximum length of Count vectors, TF-IDF vectors and text sequences the same as specified. Otherwise, the model will generate different output.
8. Also, the max length of text sequences and the Input layer size of CNN and LSTM must be same, else it will raise an Error.

IV. Results

Model Evaluation and Validation

The model was trained on 123,780 reviews and tested on 30,946 reviews.
Training time of the final model = 128 to 132 seconds/epoch for 50 epochs
Test accuracy = **67.39%**

Justification

MODEL	TEST ACCURACY
NAÏVE BAYES (benchmark)	44.33%
LSTM (Untuned)	66.13%
LSTM (Tuned)	67.39%

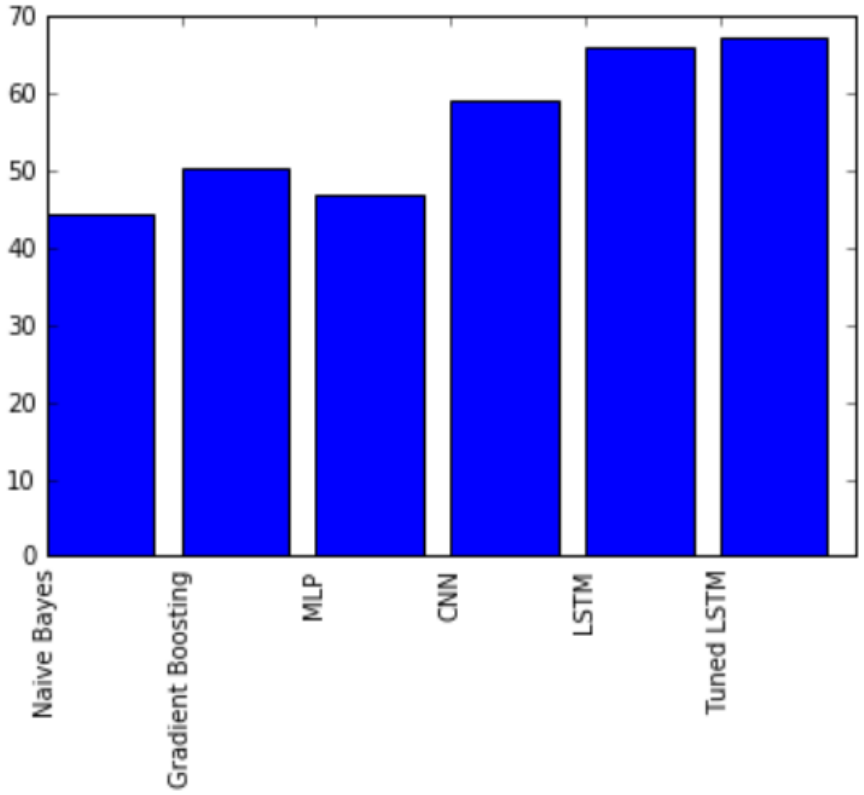
Observations: -

- Accuracy gain compared to benchmark = 23.06%
- Accuracy gain compared to original LSTM = 1.26%
- The model clearly performs much better than the benchmark model.
- It also maintains an accuracy higher than original LSTM architecture.

From these observations and the fact the Sentiment classification can be sometimes hard even for humans, we can conclude that the final model is justified for the given problem.
As for robustness of the model, I think one important factor that will make it robust to small changes in data is the inclusion of Word Embeddings from pre-trained Word2Vec model. Since, it is trained on a huge corpus of documents, those results are robust.
One small change to make it robust to outliers is include all the Word embeddings instead of only those words that were encountered in our reviews. That way, this model will be robust to outliers. But even current scenario, I believe it will generalize well due to Word2Vec embeddings.

V. Conclusion

Free Form Visualization



The above graph displays accuracies of all the models. Observations are as follows: -

1. Benchmark model has the lowest accuracy.
2. Final LSTM model has the highest accuracy.
3. Neural networks don't always have the highest accuracy. (Gradient Boosting > MLP).
4. Deep learning models perform well on Sentiment classification.
5. Although CNNs are mostly used in Computer Vision, their performance in NLP is good. This shows that CNN are good feature extractors.
6. Hyperparameter tuning in LSTMs can be tricky. This can be seen from the fact that accuracy gain from LSTM to deeper and more sophisticated LSTM is relatively minor.

Reflection

End to end problem solution: -

1. Loading the data.
2. Analysing the words in data and removing stopwords.
3. Removing null data points i.e. reviews with 0 length.
4. Training a benchmark model for performance comparison.
5. Feature Engineering – Count vectors, TF-IDF vectors, padded text sequences.
6. Loading word vectors and selecting those available for words in current corpus.
7. Training 2 classical ML models and 2 Deep learning models.
8. Hyperparameter tuning on the Most accurate model for higher accuracy.

This project was an amazing learning experience for me. Through this project I understood how to use Word vectors efficiently for model and got idea about how to use pre-trained models. I also came to understand that neural networks are not always one-shot solution for all the problems unless used properly. This was also the first time I trained an LSTM and surprisingly, it worked really well, giving me more confidence to solve NLP tasks.

Improvement

Some of the improvements I can think of is: -

1. Using Deeper LSTMs i.e. models having more than 1 LSTM layers.
2. Removing all the words whose word embeddings were absent from the Wiki 300 model. Upon analysis, I realized that they were mostly the name of actors, removing them would not have affected model performance in my opinion.
3. One thing I read about very late was the negative effects of using Regularization too much. As we can see, my final model has a lot of Regularization including Dropout at all layers and Batch normalization. Also, the number of Dense layers is very high and the performance gain is not enough considering the model was trained for 50 epoch compared to initial 5 epochs. Preventing over-regularization could be a major improvement and higher accuracy.
4. Use a different set of metrics like True positive rate and Area under ROC.