Offensive and Hateful Language Detection in Tweets using Neural Networks

Capstone Project Report

Pardeep Kumar

Mentor: Dipanjan Sarkar

Springborad Career Science Data Track

Table of Contents

Lists of Figures

Abstract

Detecting hateful language in online content is an extremely challenging task, and business organizations still rely on manual (skilled) labor to filter out harmful content. This report presents an experimental study using neural networks classifiers to classify roughly 19,000 tweets as either hateful, offensive, or neutral. Word vector representations such as word embedding and bag-or-words were used to study the impact on performance. In conclusion, the results show that the deep learning methods performed at the same level as the decision tree models such as random forest and gradient boosting, which are often known to perform better than the neural networks [1].

1 Text Classification

In machine learning, a text document is considered a structured sequence of tokens (character, words, long sentences) with a particular order that reflects the semantic meaning of the words. Text classification involves assigning a class to the text document based on its overall content. A text document can be an article, a book, or even a single sentence. As the size of online and offline text corpus grew, manually labeling the materials has become extraordinarily exhaustive and challenging. Text documents with sensitive content often require specialized skills such as categorizing whether a tweet is a racist or offensive which, sometimes gets different opinions even from the linguistic experts.

Natural Language Processing (NLP) combines the field of information theory, statistics, linguistic and computational engineering to accomplish the task of text classification. Machine learning algorithms can assist in such functions if trained on a well-labeled dataset. With the advent of neural networks such as convolution neural networks, recurrent neural networks, and hybrid models, machines can perform labor-intensive jobs with fair accuracy.

In the text classification problems, the dataset consists of the documents (di) from some document space ($\mathcal{D}$), labeled as one of the classes from a fixed class set, $\mathcal{C} = \{c_1, c_2, \dots c_N\}$. In the supervised learning, the task of a classifier (Ψ) is to find a classification function (ψ) that can accurately map each document to the corresponding class-label, $\psi : \mathcal{D} \rightarrow \mathcal{C}$. This report only presents one-on-one mapping, also called *multiclass classification* problem, where are given text input belongs to only one particular class and all classes are independent of each other. Another classification problem called *multilabel text classification* is one-to-many where a text document can have multiple labels. The goal of machine learning algorithms is to train a classifier on a training dataset and evaluate its performance on the test dataset (data set that the classifier has never *seen* before). A simple (linear) model usually result in highly biased output with underperformance on both the test and training datasets, especially when the observations have complex structures. On the other hand, sophisticated machine learning algorithms can overfit, i.e., achieve higher accuracy on the training set but underperform on test datasets. Overfitting hinders the generalization ability of the neural networks.

Performance measures based on the confusion matrix, such as accuracy, precision, F1-score, and area under the ROC curve are used to compare two classifiers. Model accuracy score is the standard performance measures in statistics. In the multiclass classification with imbalanced class distribution, relying solely on the accuracy score could lead to an inferior model selection. For examples, an *always-spam* classifier would be 95% percent accurate *on a training set with 5% non-spam* class frequency. For binary classifications, a confusion matrix compares the count of actual labels and predicted labels. For example, consider a binary

text classification problem of assigning either offensive (1) and the neutral class (0) to a text document. Figure 1.1 shows a confusion matrix for the binary text classification problem, and Table 1.1 lists the four standard performance measures. Note that depending on the task, the invariance of the performance measure may be beneficial or detrimental. A performance measure is invariant if changes in the confusion matrix have no impact on its value. There are two categories of performance measures in case of the multiclass classification problems: *micro-averaging* which sums the count of confusion matrix components to obtain the cumulative values or *macro-averaging* which computes a simple average over all classes. A micro-averaged metric is the measure of effectiveness on broad categories, whereas macro-averaged values give a sense of efficacy on individual levels.

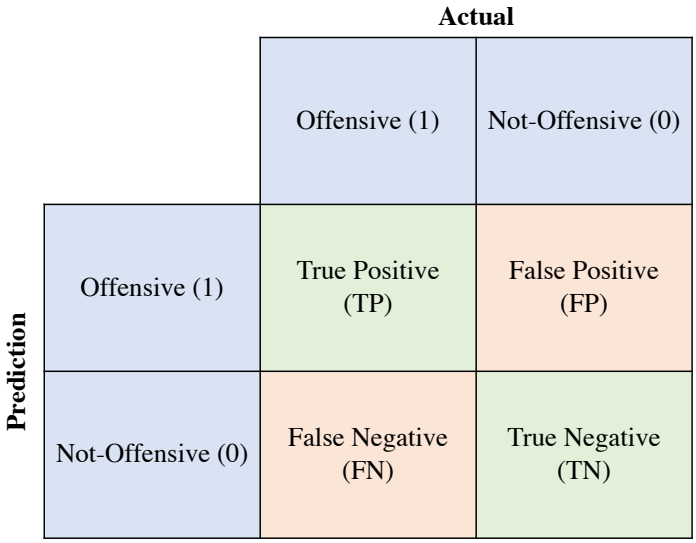

Figure 1.1 Confusion matrix for binary classification

Table 1.1 Performance measures for binary classification

Accuracy	$\frac{(TP+TN)}{(TP+FP+FN+TN)}$	Accuracy score measures the overall effectiveness of a classifier and is most effective when the class size is relatively balanced, i.e., each class has a frequency of at least 10% occurrence.
Precision	$\frac{TP}{(TP+FP)}$	The precision score measures the exactness of the classifier. Precision is high if the number of true positives is high, and false positives are low.
Recall (Sensitivity)	$\frac{TP}{(TP+FN)}$	Recall measures the completeness or sensitivity of the classifier. High recall means that the false-negative count is low and true positives are high.
F1-score	$\frac{2TP}{(2TP+FP+FN)}$	Recall and precision are often at odds. F1-score combines the two by taking their weighted harmonic mean.

Organization of Report

Chapter 2 focuses on the data mining and feature study of the processed data. Chapter 3 dives into the first step of text classification problem- converting word into numeric for the information to be processed by the machine learning algorithms such as Word embedding and bag-of-word (TF-IDF) vectors. Chapter 4 details the baseline model selection using decision trees and different input features. Chapter 5 through 7 are devoted to the three major deep learning algorithms, multilayer perceptron, convolution neural networks, and recurrent neural networks. Chapter 8 concludes the report by comparing the performance of different classifiers and suggested future work.

2 Data Collection and Preprocessing

A text data set is a collection of data objects, which in turn are defined by their attributes that reflect the essential characteristics of objects. For example, Table 2.1 shows the data set that was initially stored in a CSV-file format, uploaded in the *python* environment using *the pandas* library. Each row is the object of the data set (a tweet), and each column represents the categories. Just like any dataset, twitter dataset had two types of attributes:

1. Categorical or qualitative attributes such as the class of each tweet (hate, offensive or neutral), and
2. Numeric or quantitative characteristics such as the length of each tweet.

Most machine learning tasks start with:

1. Collection of the raw data, and
2. Preprocessing the raw data to make for analysis.

Data Collection

Twitter data was collected from three different sources and combined to form a larger dataset. Including more data in machine-learning analysis helps to reduce the model overfitting. The following section discusses each dataset:

1. Collection of 24,784 manually labeled tweets by CrowdFlower [2] users as *hate_speech*, *offensive_langauge*, or *neither* [3] was the first and primary datasets. Table 2.1 shows the first two rows of raw data Table 2.1. Out of the six columns in the table, four fields were dropped, keeping only columns labeled *class* and *tweet*. The class field contained the majority votes a tweet received from the CrowdFlower users out of 0 – hate speech, 1 – offensive speech, and 2 – neutral (neither). As shown in Figure 2.1, the tweet count per class was highly imbalanced Figure 2.1.

Table 2.1 Raw dataset#1 for tweet classification

	count	**hate_speech**	**offensive_language**	**neither**	**class**	**tweet**
0	3	0	0	3	2	!!! RT @mayasolovely: As a woman you shouldn't complain about cleaning up your house. & as a...
1	3	0	3	0	1	!!!!! RT @mleew17: boy dats cold...tyga dwn bad for cuffin dat hoe in the 1st place!!

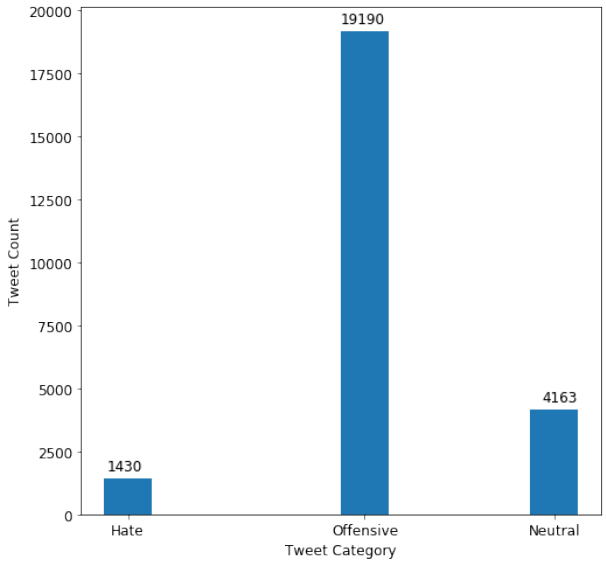

Figure 2.1. Tweet count per class [2]

2. Table 2.2 shows two rows from a set of 31,962 tweets [4]in the second mined dataset [7]. Each row was labeled either 0 (non-racist/sexist tweets) and 1 (racist/sexist tweets) as shown in Table 2.2. When combining with the first dataset, racist/sexist tweets were added under the hate category and remaining to the neutral class.

Table 2.2 Raw dataset#2 for tweet classification

id	**label**	**tweet**
13	0	i get to see my daddy today!! #80days #gettingfed
14	1	@user #cnn calls #michigan middle school 'build the wall' chant " #tcot

3. The third dataset files contained the tweet IDs and the annotations [5]. The individual tweets were obtained by querying the tweet IDs (*Tweepy* API). Each tweet belonged to either racist, sexist, or neither categories. Table 2.3 shows the sample rows from the dataset. Tweets in the racist and sexist classes were added to the hate class and those labeled neither to the neutral category of the big dataset (the combination of dataset 1 and 2). The final data (Figure 2.2) selected for this study was a subset of this larger dataset selected to obtain a balance tweet count, approximately 6580, per class.

Table 2.3 Raw dataset#3for tweet classification [6]

User id	**label**	**tweet**
572345016275771000	sexism	I cannot stop looking at Nikki's dreadful black crooked bra #MKR #MKR2015
572344928069500000	sexism	Trying to find something pretty about these blonde idiots.#MKR

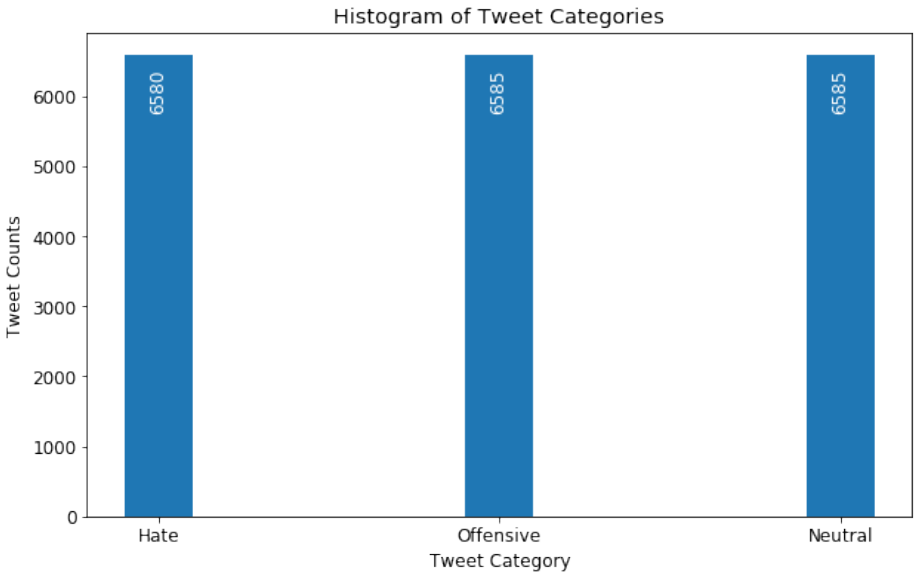

Figure 2.2. Final dataset counts per class

Data Preprocessing

The raw tweets were messy, especially those that belonged to the hate and offensive categories, with poor spelling, grammar, and sentence structure. Often extensive data cleaning is necessary before developing any text classifier. The raw data files were uploaded in the UTF-8 Unicode (byte per character) format preserving the unknown words that are often modified if loaded using ASCII format. The text normalization methods such as stemming or lemmatization were excluded, to study how well classifiers can perform on the moderately processed text.

Following features were removed from the tweet during preprocessing:

1. Username: Usernames were limited to a maximum length of 15 characters after '@' symbol.
2. Web address: Web addresses started either with "*https://*" or directly as "*www…*".
3. Retweet handle: Keyword 'RT' short for re-tweet.
4. Punctuations: Punctuations are the most common feature in the online texts and do not convey any meaning to the machine learning classifiers.
5. Emoticons: Emoji could be useful in the right context but, dropped for this study.
6. Numbers: Numeric data did not add to any text classification relevance.

All uppercase characters were converted to the lowercases. Figure 2.3 shows the character length distribution of the *raw* and *cleaned* tweets. Maximum character length in the unprocessed dataset was 754 characters exceeding the twitter limit of 140 characters (now increased to 250) because of the twitter emojis

in the text. Emoticons are extremely popular in online texts, and their UTF-code expand to long character strings in datafiles. Table 2.4 [7] shows a few twitter emojis and their respective byte codes.

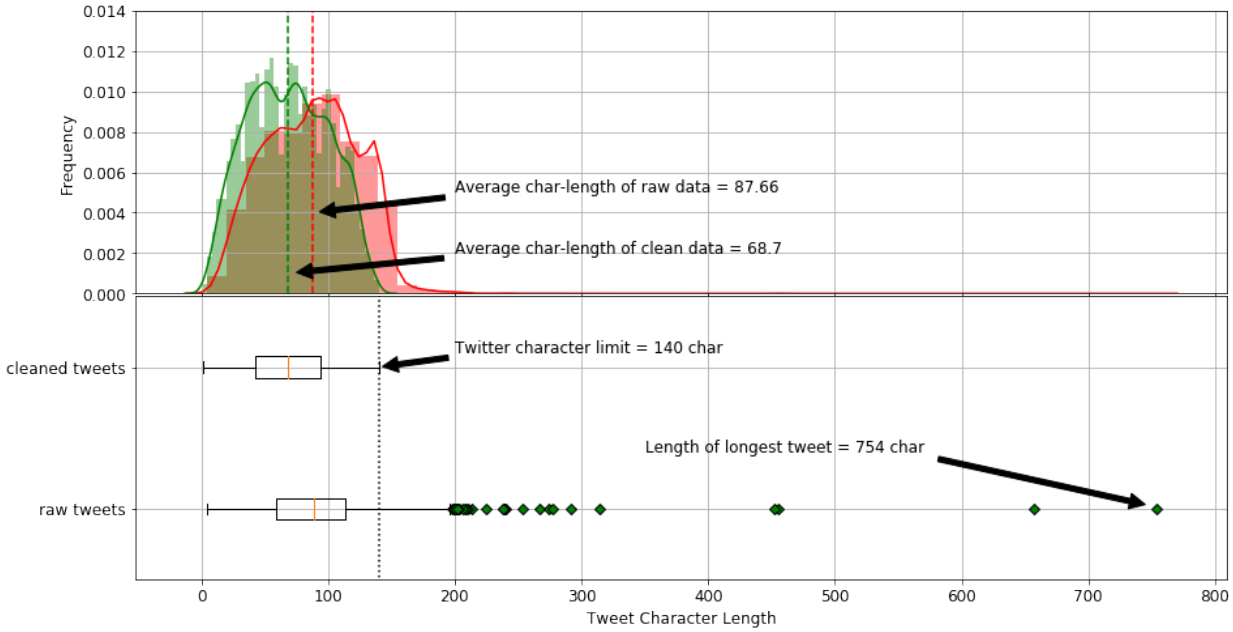

Figure 2.3 Character length comparison of raw- and clean-tweets]

Table 2.4 Emoticons (1F601-1F64F) [7]

Twitter	Unicode	Byters (UTF-8)	Description
😂	U+1F602	\xF0\x9F\x98\x82	Face with tears of joy
😃	U+1F603	\xF0\x9F\x98\x83	Smiling face with open mouth
😄	U+1F604	\xF0\x9F\x98\x84	Smiling face with open mouth and smiling eyes

Table 2.5 shows the statistics of the raw- and cleaned-tweet character length. The average length of the cleaned tweets was about 80% of the original tweets with 14% less standard deviation. Preprocessing resulted in removing the unwanted outliers from the dataset, Figure 2.3.

Table 2.5 Statistics of unprocessed and processed tweets.

	count	mean	std	min	25%	50%	75%	max
Raw tweet length	19750	87.66	36.66	5.0	59.0	89.0	114.0	754.0
Clean tweet length	19750	68.70	31.69	1.0	43.0	68.0	94.0	140.0

The processed data was splits into 2:1 ratio, as the training and the test set (Figure 2.4). For the neural network classifiers, the training set was further divided into the training and the validation sets in ratio 2:1.

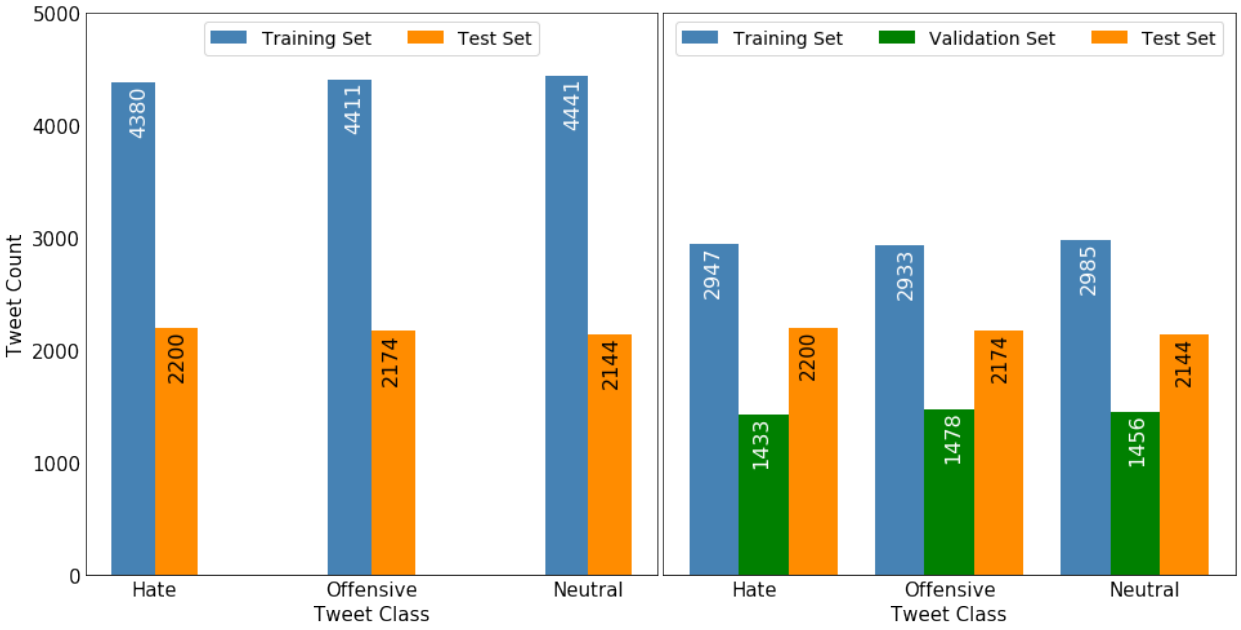

Figure 2.4 Tweet count (per class) in training and test set

3 Word Vectors

Following the text data mining and preprocessing, the text data requires tokenization. Machine learning algorithms can only work with numerical inputs, and each algorithm requires data presented in a specific format. *Tokenization* is the process of splitting the text documents into smaller units called tokens and assign them either a unique index (a number) or a vector. Depending on the problem, tokens can be individual characters, words in the document, or N-grams. N-grams are the groups of overlapping consecutive characters or words. Most of the research in natural language processing has focused on the corpus in the English language. Without due diligence, even state-of-the-art machine learning models might underperform significantly when other languages are involved. For example, if the language contains rare vocabulary words (a word with low occurrence), the character-based tokenization might be a better representation of the input (rather than word-based).

Words that occur in similar contexts tend to have similar meanings [8] [9]. In machine learning, a word vector represents the meaning of a word. Vector similarity can be expressed using the cosine of the angle between two (unit) vectors. For two n-dimensional vectors, $x_i, x_j \in \mathcal{R}^n$, the cosine similarity is computed as the dot product between two vectors (usually unit vectors) as,

$$Similarity_{cosine} = cos(\theta) = \frac{x^i}{\|x^i\|_2} \cdot \frac{x^j}{\|x^j\|_2}$$

If the words are the points in the vector space, words with similar meaning are "closer," and the magnitude of the cosine similarity measure is closer to unity. On the other hands, for the two uncorrelated (out-of-context) words, the cosine similarity measure would be smaller or close to zero. Two common vector-representations are a bag-of-words model (count or co-occurrence based) and word embedding (context-prediction based).

Bag-of-Words

Bag-of-word models hypothesize that relevance of a document to a specific query depends on the word frequency in the document [10]. Bag-of-words model *represent*s a word (token) as a one-hot encoded vector. For example, consider a simple expression *"My dog is always excited and cheerful*." Figure 3.1 shows the one-hot encoded representation of the words. Each word in Figure 3.1 is a 7-dimensional unit vector. As the size of vocabulary increases, so does the length of each one-hot vector. Managing large scale sparse vectors (matrices) can be computationally expensive and often impossible. Also, the bag-of-word vectorization technique is not order-preserving, and the meaningful structure of the sentence is lost when transformed into one-hot vectors. In Figure 3.1, the words '*excited*' and '*cheerful*' are similar in context

but have zero cosine similarity, which would affect the ability of an algorithm to learn the correct document context.

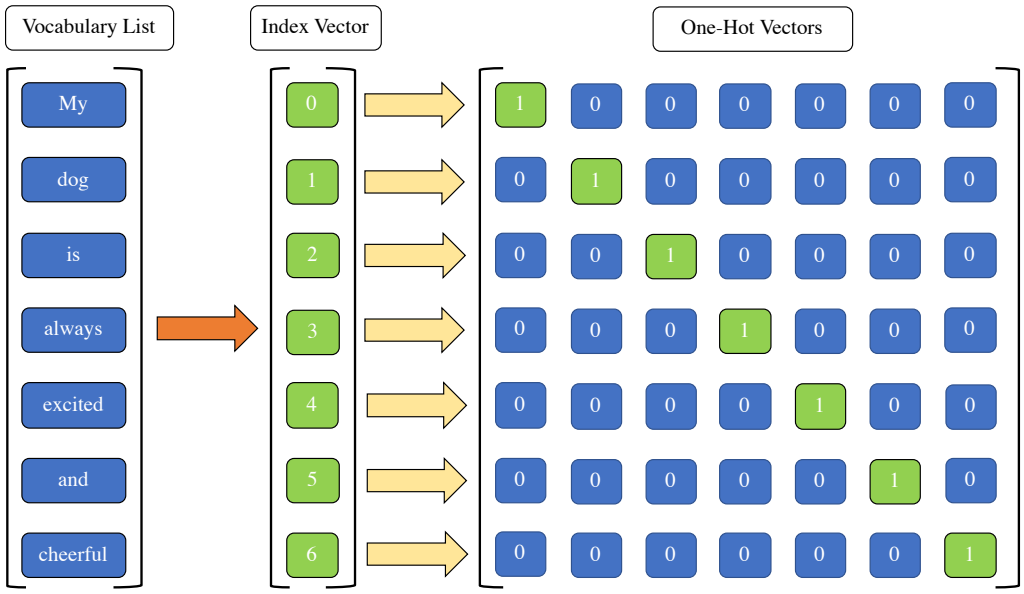

Figure 3.1. One-hot encoding

Figure 3.1 is a simple example where each word occurs only once in the text (document). Large documents comprise of hundreds of words repeated multiple time. For example, the word *the* occurs in the context of almost every noun and has high word-frequency. One approach is to replace the indicator (0,1) for word occurrence with the (0, word count). Another approach is to create a co-occurrence matrix. Co-occurrence matrix counts the number of times (frequency) a word appears in a document with other words in the corpus.

Figure 3.2 shows the co-occurrence matrix of a famous tongue twister. The matrix entries are co-occurrence count (frequency) of a word with other words in the matrix. The matrix in Figure 3.2 had a window of size one i.e., co-occurrence count with words in immediate proximity. For larger document space, term-document matrix, where each entry is the count of the target word in different documents, is more informative than the (word-word) co-occurrence matrix.

Stop words (e.g., *a, the, of*) occur more frequently in the documents and cause skewness in the co-occurrence matrix without adding any relevant information. On the one hand, the count of more frequently co-occurring words is crucial, and on the other hand, words that are more frequent lack of any useful information. **TF-IDF** algorithm can balance the two paradox to obtain an updated new-matrix. Each entry in the updated matrix is the product of the term-frequency and the inverse document frequency.

"she sells sea-shells on the sea shore. the shells she sells are sea shells. if she sells sea-shells on the sea shore, then she sells sea shore shells"

	she	sells	sea	shells	on	the	shore	if	then
she	0	4	1	1	0	0	0	1	1
sells	4	0	3	0	0	0	0	0	0
sea	1	3	0	3	0	2	3	0	0
shells	1	0	3	0	2	1	1	1	0
on	0	0	0	2	0	1	0	0	0
the	0	0	2	2	2	0	1	0	0
shore	0	0	3	1	0	1	0	0	1
if	1	0	0	1	0	0	0	0	0
then	1	0	0	0	0	0	1	0	0

Figure 3.2. Co-occurrence matrix

1. Term frequency is obtained by logarithmic transformation (base 10) of the entries of the co-occurrence matrix of a given document ($C_{t,d}$), such that words that occur more frequently receive a higher penalty.

$$tf_{t,d} = \begin{cases} 1 + \log_{10} C_{t,d}, if\ C_{t,d} > 0 \\ 0 \qquad\qquad otherwise \end{cases}$$

2. The inverse document frequency (idf) assign relative importance to more discriminative words, words that might appear more often in one document but sparsely in the rest of the text-corpus. A large document set is also log-transformed to bound the computed values,

$$idf_{t,d} = \log_{10}\left(\frac{size\ of\ text\ corpus}{number\ of\ documents\ the\ term\ t\ occurs}\right)$$

The elements of the updated matrix are computed as,

$$C_{t,d}^{updated} = tf_{t,d} \times idf_{t,d}$$

TF-IDF vector model can capture similarities between words as they focus heavily on target-context word occurrence. However, the issue of large vector (matrix) size remains a challenge as online text data usually has a large vocabulary. The idea of TF-IDF can be applied to characters or word N-grams.

Word Embedding

A straightforward way to avoid long sparse vectors is to map them onto a smaller dimension. The dense representation of sparse word vector is called *word embedding*. A one-hot vector is trained on a single hidden layer neural network, and the weights of the layers correspond to the dense representation of the word. Each word in the vocabulary is assigned a target and context vector, and the neural network is trained such that similar words have a high value of the similarity measure.

Word embedding can be included in the model either by extracting vocabulary from the training data and (specific to a problem) or by using the pre-trained word embedding. In either case, each word in the dataset is assigned an index and embedding layer maps the index to a dense vector. There are two primary pre-trained word-embedding databases: Word2Vec and GloVe.

3.2.1 Word2Vec

Word2vec is an efficient way of local context-based learning and has been quite useful in the text classifications using deep learning [11]. Word2Vec method yields an embedding layer, which is the input to the classifier. Two main algorithms for getting the word embedding in Word2vec are:

1) Continuous skip-gram model: Given a target word, the continuous skip-gram model learns the vector representation of the context words. A binary classifier is trained to predict the likelihood of a word occurring in the neighborhood of the target word. For example, given the target word *'bank'* it computes the probability of word '*financial'* appearing in the near neighborhood.

2) Continuous bag-of-word model. The continuous bag-of-words model learns the dense vector representation of a target word from the given context. For example, given words like *'money,' 'services,' 'finance*,' what is the probability of word *'savings'* being in that 'bag' of words. Both algorithms are trained using 1) hierarchical *softmax* method or 2) negative sampling.

3.2.2 GloVe

Word2Vec are predictive models of vector representation that focus on minimizing the loss in predicting the target words from the context words. Global Vector Word Representations (GloVe) are count-based models that learn by dimensionality reduction on the global (from large corpus) word-word co-occurrence matrix (frequency of the target words co-occurring with the context words). The large co-occurrence matrix is normalized, log-smoothed, and the factorized to obtain a lower-dimensional matrix. Factorization is done such that the higher-dimensional data variance is well explained by the lower-dimensional matrix [12].

Performance of NLP models using pre-trained word-vector representations based on, either Word2Vec or GloVe corpus has been comparable [13].

For this study, 300-dimensional pre-trained GloVe model was used for the vector representations of the tweets and the tokens [12]. The classifiers were trained on the same training set, and the best model was selected based on the test set performance.

Figure 3.3 shows the distribution of word-counts per tweet using strip and violin plots. The maximum length of tweets in the training set was 34 words (far right green zone in Figure 3.3), and about 50% of the tweets had a word count of 12 or less. Most tweets had 10 words about 45% tweets had word count in range 8 to 14. Note that if the maximum document length in the test set exceeds that of the training set, the text was curtailed to the maximum document length measured in the training set.

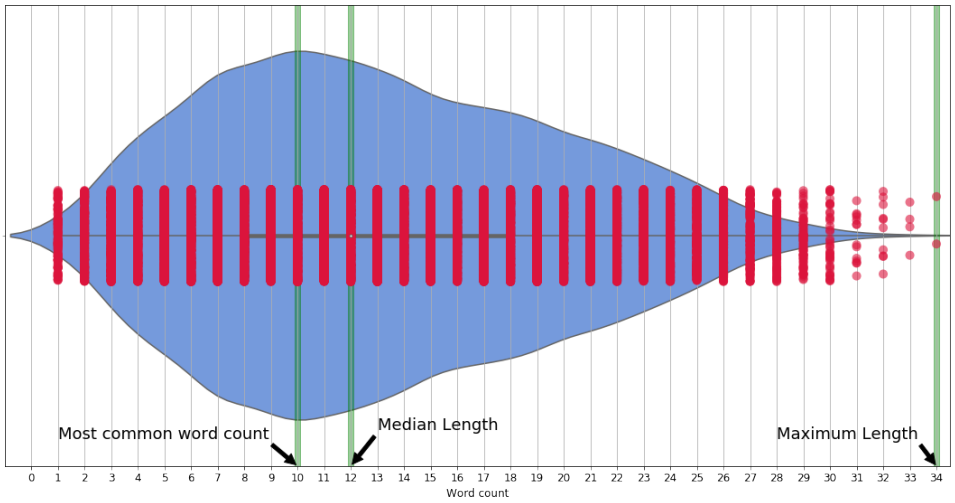

Figure 3.3 Tweet length (word count)

For uniform inputs, the tweet texts were zero-padded near the right end (post, zero-padding). Zero-padding is vital in deep learning algorithms where the length of the input layer should be fixed. Figure 3.4 shows an example of zero-paddings. Assume $(i-1)^{th}, i^{th}$, and the $(i-1)^{th}$ tweet has a word count of 32, 34, and 2. For a uniform input (size-wise), 2 and 32 zeros can be added to the right of the last words of $(i-1)^{th}$ and $(i+1)^{th}$ tweet, respectively.

The input text was tokenized, where the sentences were split into words, and each unique word was assigned an index starting with 1 (index 0 was reserved for zero-paddings). The training dataset had a vocabulary size of 18, 602 words.

The 300-dimensional (pre-trained) GloVe data had about 400,000 unique words and a unique vector for each word. An empty embedding matrix (18,602 x 300) was created. Each word from the training set was

queried against the GloVe vocabulary, and corresponding dense vectors were stored in the embedding matrix. For words with no match in GloVe corpus were initialized with zero vectors. Figure 3.5 shows a simple example with the vocabulary size of 8 and the steps to obtain the corresponding embedding matrix.

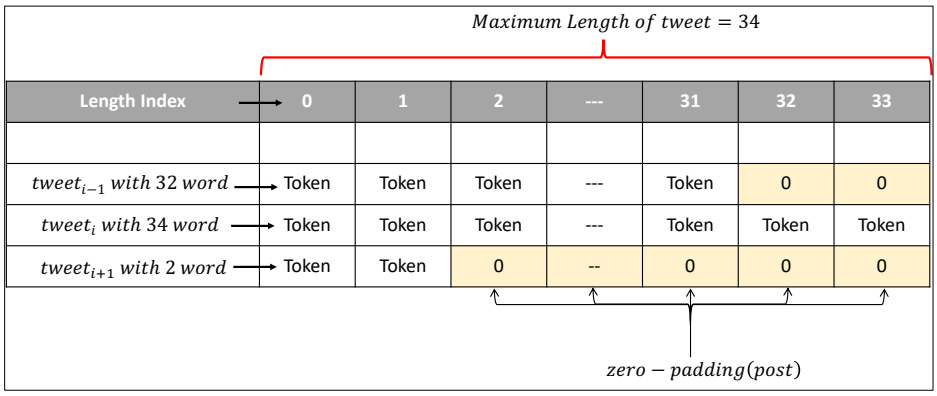

Figure 3.4 Zero-paddings

he	was	tired	of	tohuwabohu	
when	he	was	little	bit	tired

Tokenizer

Words	he	was	tired	of	tohuwabohu	when	little	bit
Index	1	2	3	4	5	6	7	8

Query Embedding Space

Embedding Space

	the	,	.	of	to	and	in	a	"	's	...
0	0.046560	-0.255390	-0.12559	-0.076947	-0.257560	0.038466	-0.44399	-0.297120	0.694700	-0.001272	...
1	0.213180	-0.257230	0.01363	-0.021211	-0.057132	-0.039792	0.12817	0.094049	0.221840	0.365140	...
2	-0.007436	0.131690	0.10306	0.212710	-0.671900	0.082747	-0.25247	-0.096662	0.105260	-0.077363	...
3	-0.458540	-0.042688	-0.10123	-0.722320	-0.380820	-0.389230	-0.18582	-0.344000	0.012382	-0.265590	...

Embedding Matrix

Reserved	he	was	tired	of	tohuwabohu	when	little	bit
0	1	2	3	4	5	6	7	8
0	0.0120	0.2325	0.2283	0.2825	0	0.0494	0.0088	0.3982
0	0.0831	0.2569	0.5450	0.3179	0	0.2064	0.2497	0.1425
0	0.3814	0.1521	0.2658	0.2497	0	0.1952	0.1500	0.3512
0	0.2241	0.2325	0.2283	0.3102	0	0.1828	0.2825	0.2239
…	…	…	…	..	…	…		…

Figure 3.5 Word Embedding

4 Baseline Model

A famous aphorism in statistical learning is "all models are wrong, but some are useful'. Thus, a model can only be evaluated relative terms to another due to lack of an absolute measure in statistics. A baseline model serves as a benchmark against which the performance of other classifiers can be compared. Characteristics such as simple execution and interpretation usually decide a baseline model. Decision tree models were used as the baseline model for this study.

[redacted]Random Forests and Boosting

Tree-based models involve stratification of the feature space to simple regions, where an (inverted) tree is grown using splitting rules imposed on the input features at different points (called nodes). Each split-up path (called branch) leads to the final regions (called leaf nodes). Regression and classification problems can be solved using decision tree methods.

Figure 4.1 shows a simple binary classification tree that predicts whether a document (with multiple sentences) is motivational or not. Two simple splitting rules imposed at each node result in branches that lead to either the positive (motivational) class or the negative (not-motivational) class. The decision tree is fit (grown) on every sentence in the document, and finally, the document is assigned the majority class (model prediction). Switching the two conditional nodes in Figure 4.1 would result in two different trees.

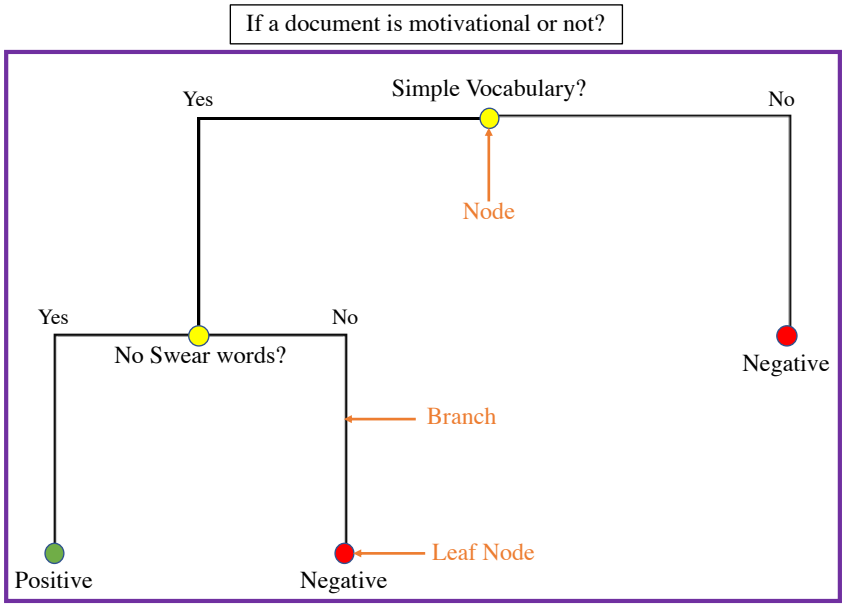

Figure 4.1 Document classification tree

Decision trees are intuitive and easily interpretable. Their interpretability, however, decreases for more complex problems. Decision trees can overfit, where large trees tend to fit well on the training set but perform poorly on the test set due to lack of generalization. Tree pruning can be used to control the depth of the tree and reduce the number of splits, but if the initial splits are relatively less important to the splits down the line, then tree pruning might be detrimental to the classifier.

Averaging the results of multiple, fully-grown trees on different training data sets could result in reduced variance in the output. Unlimited training sets, however, are not available in practice. Recording more observation or generating synthetic data can help solve the high variance issue. Decision trees rely on a simple, yet powerful technique called *bagging* for obtaining larger training dataset. From a given training dataset of size *n*, *bagging* draws random observations with replacement to generate new data sets of size *n*. On average, the bagged subsets contain 2/3rd of the original data. Trees are then fit on the different bagged datasets, and the majority class is the model's prediction.

Bagging is effective in reducing the variance if the resulting trees are uncorrelated. Stronger features are often selected picked first and most, resulting in similar trees on bagged datasets. Having a large number of correlated trees results in the issue of overfit remaining unresolved. *Random forests* overcome this limitation by randomly selecting a subset of the input features to choose from, at each node. Random selection provides an opportunity for the not-so-strong features to be selected built different trees. Out of say F total predictors, a random subset of f predictors is selected at each node. On average, $\left(\ {}^{/}_{(1 - f \ / \ F)} \right)$ splits will not even consider the strong features for splitting decision, which results in decorrelated decision trees. Averages from decorrelated trees are less prone to overfitting, making the analysis reliable. Typically, the subset size f is approximately equal to the square root of the total number of features, i.e., $f \approx \sqrt{\sqrt{F}}$.

Boosting is a sequential process slightly different from the other decision tree algorithms were rather than fitting a new tree on different datasets, trees are fit on the residuals. The process is repeated until the error falls below a predefined threshold. Thus, there is information transfer along the process (similar to RNNs discussed later). Bagging is not required, and neither is growing deeper tress. The maximum tree depth in boosting is less than the random forest models (often, just a stump). Small trees do not overfit on the training data, and the method could focus on slowly reducing the high bias introduced by small trees. A shrinkage parameter is used to reduce the rate of learning, allowing more trees to contribute to the error reduction.

Decision Trees for Tweet Classification

The decision tree classifiers were trained on different feature vectors using TD-IDF algorithm and GloVe embeddings.

4.2.1 Model Parameter Selection

Random forest and gradient boosting classifiers has multiple hyperparameters such as number of trees, depth of tree, type of error criterions etc. The different input options in python for each classfier are shown in the Code Section 4.1 and Code Section 4.2. In order to train the model in grid seach, TF-IDF algorithm was used to obtain word uni-gram input vectors. Code Section 4.1 and Code Section 4.2 also shows the result for a best fit model on the training set and classification report on the test set for the two classifiers. The best fit models were selection based on the minimum error using stratified 5-fold cross-validations.

In the case of the random forest model, the accuracy score did not vary significantly under different hyperparameter combinations. Accuracy score (best) of 0.79 was obtained for the model with 500 random trees and the depth of each tree equal to 100. The recall test score for the *hate* class was low, 0.68, suggesting a relatively higher number of false negatives, hate tweets misclassified as belonging to the other two classes. Models fit on 300 random trees slightly underperformed at all levels.

Gradient boosting models had slightly better performance than the random tree forests and were more sensitive to changes in the magnitude of grid parameters. As the learning rate increased from 0.01 to 0.1, the accuracy score increased by 4.5%. At learning rate 0.01, the net change in average accuracy score was 5.6% as the maximum depth, and the number of estimators increased from 3 to 5 and 300 to 800, respectively. In comparison under learning rate of 0.1, for the same changes in maximum depth and number of estimators, the average accuracy score increased only by 0.9%. Gradient boosting had higher precision, recall, and f1-score (on average and per class basis) relative to the random forest model.

Code Section 4.1 Best random forest model fit using grid search

Model parameters

```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=RandomForestClassifier(bootstrap=True, class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators='warn', n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='warn', n_jobs=-1,
             param_grid={'bootstrap': [True, False],
                         'max_depth': [90, 100, 110],
                         'n_estimators': [300, 500, 800]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=2)
```

Accuracy scores for each grid search

```
Best: 0.790432 using {'bootstrap': False, 'max_depth': 100, 'n_estimators': 500}
0.785142 (0.008365) with: {'bootstrap': True, 'max_depth': 90, 'n_estimators': 300}
0.784764 (0.013018) with: {'bootstrap': True, 'max_depth': 90, 'n_estimators': 500}
0.785973 (0.009922) with: {'bootstrap': True, 'max_depth': 90, 'n_estimators': 800}
0.782044 (0.009169) with: {'bootstrap': True, 'max_depth': 100, 'n_estimators': 300}
0.783782 (0.010235) with: {'bootstrap': True, 'max_depth': 100, 'n_estimators': 500}
0.786880 (0.008983) with: {'bootstrap': True, 'max_depth': 100, 'n_estimators': 800}
0.784537 (0.010203) with: {'bootstrap': True, 'max_depth': 110, 'n_estimators': 300}
0.784008 (0.008408) with: {'bootstrap': True, 'max_depth': 110, 'n_estimators': 500}
0.785671 (0.007867) with: {'bootstrap': True, 'max_depth': 110, 'n_estimators': 800}
0.783706 (0.008839) with: {'bootstrap': False, 'max_depth': 90, 'n_estimators': 300}
0.788014 (0.007047) with: {'bootstrap': False, 'max_depth': 90, 'n_estimators': 500}
0.789525 (0.009312) with: {'bootstrap': False, 'max_depth': 90, 'n_estimators': 800}
0.784386 (0.005450) with: {'bootstrap': False, 'max_depth': 100, 'n_estimators': 300}
0.790432 (0.009400) with: {'bootstrap': False, 'max_depth': 100, 'n_estimators': 500}
0.788845 (0.009999) with: {'bootstrap': False, 'max_depth': 100, 'n_estimators': 800}
0.782270 (0.007470) with: {'bootstrap': False, 'max_depth': 110, 'n_estimators': 300}
0.788845 (0.005331) with: {'bootstrap': False, 'max_depth': 110, 'n_estimators': 500}
0.788392 (0.007252) with: {'bootstrap': False, 'max_depth': 110, 'n_estimators': 800}
```

Classification Report of the best fit model on the test set

	Precision	recall	f1-score	support
hate	0.79	0.68	0.73	2200
offensive	0.81	0.83	0.82	2174
neutral	0.76	0.85	0.81	2144
accuracy			0.79	6518
macro avg	0.79	0.79	0.79	6518
weighted avg	0.79	0.79	0.79	6518

Code Section 4.2 Best gradient boosting model fit using grid search

Model parameters

```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=XGBClassifier(base_score=0.5, booster='gbtree',
                                     colsample_bylevel=1, colsample_bytree=1,
                                     gamma=0, learning_rate=0.1,
                                     max_delta_step=0, max_depth=3,
                                     min_child_weight=1, missing=None,
                                     n_estimators=100, n_jobs=1, nthread=-1,
                                     objective='binary:logistic',
                                     random_state=0, reg_alpha=0, reg_lambda=1,
                                     scale_pos_weight=1, seed=None, silent=True,
                                     subsample=1),
             iid='warn', n_jobs=-1,
             param_grid={'learning_rate': [0.1, 0.01], 'max_depth': [3, 5],
                         'n_estimators': [500, 800]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=2)
```

Accuracy scores for each grid search

```
Best: 0.817715 using {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 800}
0.810837 (0.004461) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 500}
0.812878 (0.006458) with: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 800}
0.814994 (0.006503) with: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 500}
0.817715 (0.007118) with: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 800}
0.755895 (0.008485) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 500}
0.778265 (0.009035) with: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 800}
0.786049 (0.006886) with: {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 500}
0.797990 (0.008162) with: {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 800}
```

Classification Report of the best fit model on the test set

	precision	recall	f1-score	support
hate	0.81	0.74	0.77	2200
offensive	0.85	0.87	0.86	2174
neutral	0.81	0.86	0.83	2144
accuracy			0.82	6518
macro avg	0.82	0.82	0.82	6518
weighted avg	0.82	0.82	0.82	6518

4.2.2 Bag-of-Words (TF-IDF Vectors)

The input tweets were first vectorized using the TF-IDF algorithm. Twitter data often contains slangs that are not included in English vocabulary or might be from a different language. Character level feature space can sometimes result in a better classification model [14]. Figure 4.2 shows the distribution of the character length of each word in the vocabulary (training set). The average length of vocabulary words was 7.25, close to the median values of 7. The input features of different character and word combinations (N-grams) were used to train the classifier. The most common word length was 6 characters, and more than half the vocabulary words had character length between 5 to 8. Based on this observation, 11 different N-gram features were used to evaluate the random forest classifier:

1. Character level N-grams where $N \in \{2, 3, 4, 5, 6, 7, 8\}$
2. Word level N-grams where $N \in \{1, 2, 3\}$

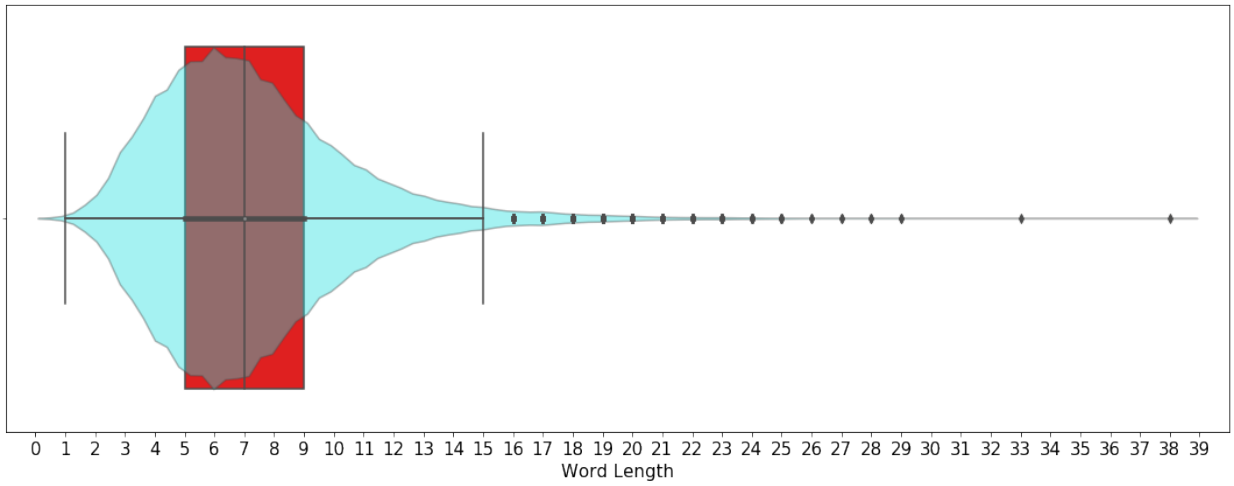

Figure 4.2. Character-length distribution of vocabulary words

For consistency, the same hyperparameters obtained as the best fit for the word embedding models were used to train on TF-IDF vectors. Table 4.1 and Table 4.2 lists the performance of random forest model on different input features. Character level N-gram features performed better than word-level features. Overall performance of character 4-grams was highest while word trigrams had the worst values of the metrics, in line with another study [14]. Word unigrams also performed comparably well.

Table 4.3 and Table 4.4 shows the performance of the gradient boosting model on the TF-IDF feature space. Similar to the random forest, the boosting trees performed slightly better on the character level than the word level. Performance measures of the character tri-, 4- and 5-gram, and word unigrams had almost the same magnitude. The performance metric (macro F1-score) of the gradient boosting fell sharply (almost 50%) when the feature size increased from word unigram to word trigrams. Gradient boosting trees trained

on the word trigrams had the worst performance among all the feature sets. In comparison, at character level n-grams the gradient boosting performance only had loss of about 4%. Overall, the gradient boosting models performed slightly better than the random forest model, but at higher computational costs.

Table 4.1 Performance of the random forest algorithm on character N-grams

Performance Measures		Character N-grams						
		Bigrams	Trigrams	4-grams	5-grams	6-grams	7-grams	8-grams
Micro	Precision	0.80	0.84	0.85	0.85	0.82	0.81	0.77
	Recall	0.80	0.84	0.85	0.85	0.82	0.81	0.77
	F1- Score	0.80	0.84	0.85	0.85	0.82	0.81	0.77
Macro	Precision	0.81	0.84	0.85	0.84	0.82	0.81	0.78
	Recall	0.80	0.84	0.85	0.85	0.82	0.81	0.77
	F1- Score	0.79	0.84	0.85	0.84	0.82	0.81	0.78

Table 4.2 Performance of the random forest algorithm on word N-grams

Performance Measures		Word N-grams		
		Unigrams	Bigrams	Trigrams
Micro	Precision	0.83	0.65	0.50
	Recall	0.83	0.65	0.50
	F1-Score	0.83	0.65	0.50
Macro	Precision	0.83	0.67	0.66
	Recall	0.83	0.65	0.50
	F1-Score	0.82	0.65	0.47

Table 4.3 Performance of the gradient boosting algorithm on character N-grams

Performance Measures		Character N-grams						
		Bigrams	Trigrams	4-grams	5-grams	6-grams	7-grams	8-grams
Micro	Precision	0.82	0.85	0.85	0.85	0.82	0.79	0.78
	Recall	0.82	0.85	0.85	0.85	0.82	0.79	0.78
	F1- Score	0.82	0.85	0.85	0.85	0.82	0.79	0.78
Macro	Precision	0.82	0.85	0.85	0.85	0.82	0.81	0.78
	Recall	0.82	0.85	0.85	0.85	0.82	0.80	0.79
	F1- Score	0.82	0.85	0.85	0.85	0.82	0.80	0.79

Table 4.4 Performance of the gradient boosting algorithm on word N-grams

Performance Measures		Word N-grams		
		Unigrams	Bigrams	Trigrams
Micro	Precision	0.84	0.66	0.48
	Recall	0.84	0.71	0.64
	F1-Score	0.84	0.66	0.48
Macro	Precision	0.85	0.71	0.64
	Recall	0.84	0.66	0.47
	F1-Score	0.84	0.66	0.44

Final Baseline Model

Table 4.5 compares the macro F1-score, measured on the test data, for the two decision tree models on various inputs. Although the performance of gradient boosting models was slightly better, the computation time was higher than the random forest. Without loss of too much performance, random forests are comparable to the gradient boosting trees. In terms of word vectorization, TD-IDF character N-grams were superior to any other input features. Since the other classifiers (neural networks) were trained using word-level vectorization, word-unigram + gradient boosting model was selected as the *baseline model* for this study.

Table 4.5 F1-score comparison of random forest and gradient boosting models

	Token Vectors	Random Forest	Gradient Boosting
1	Character Bigrams	0.79	0.82
2	Character Trigrams	0.84	0.85
3	Character 4-grams	0.85	0.85
4	Character 5-grams	0.85	0.85
5	Character 6-grams	0.82	0.82
6	Character 7-grams	0.81	0.80
7	Character 8-grams	0.78	0.79
8	Word Unigram	0.82	0.84
9	Word Bigrams	0.65	0.66
10	Word Trigrams	0.47	0.44

4.3.1 Performance Measures

Figure 4.3 shows the overall confusion-matrix of the baseline model on the test data and the per-class confusion matrices derived from it. Table 4.6 lists different performance measures computed with confusion matrix results. Due to balanced classes (in terms of data size per class), the macro- and micro-performance measures along with accuracy were the same magnitude. For comparing different classifiers, F1- score (includes the effect of both precision and recall) was selected as the primary performance measure. The model has a higher success rate (true positives) in identifying offensive and neutral class tweets. Figure 4.4 shows the ROC curve, a plot between the true-positive rate and the false-positive rate. The area under the ROC curve often reflects the level of model performance, which AUC for a perfect model being close to 1. Based on all the magnitude of AUC performance measures, the performance of gradient boosting model in classifying the tweets into three categories was better.

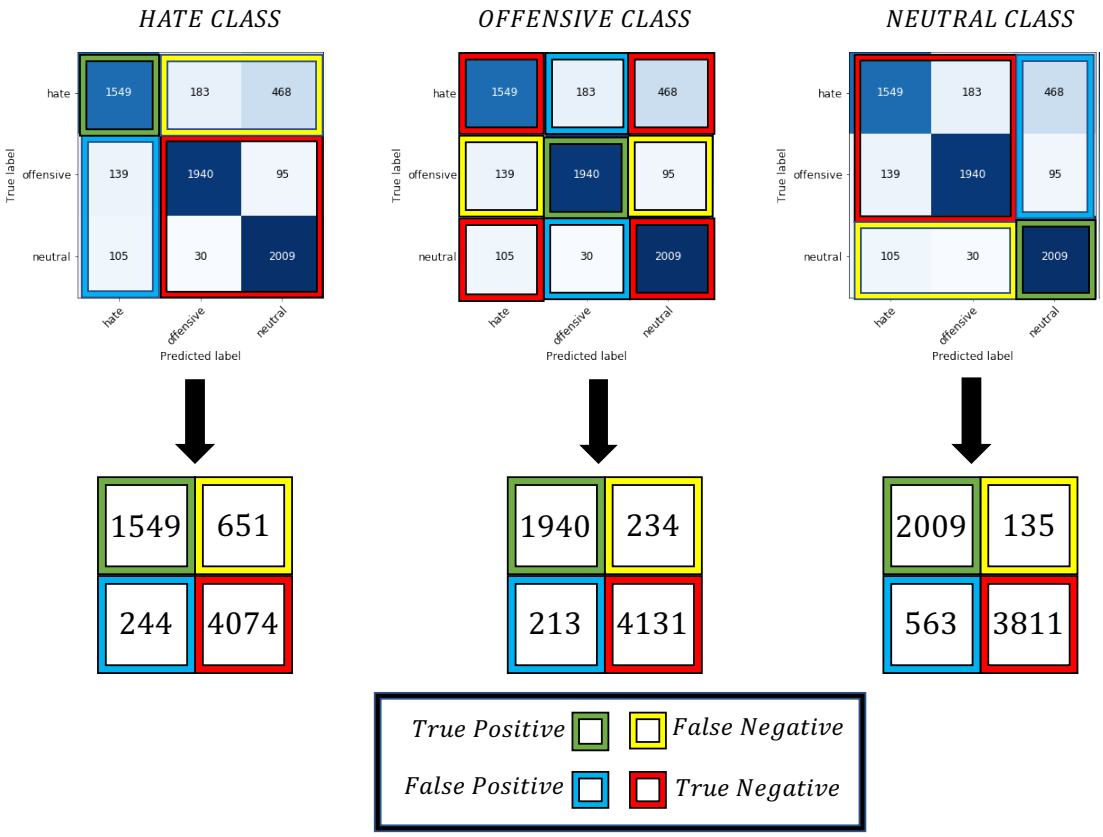

Figure 4.3. Overall and per-class confusion matrices of the baseline model

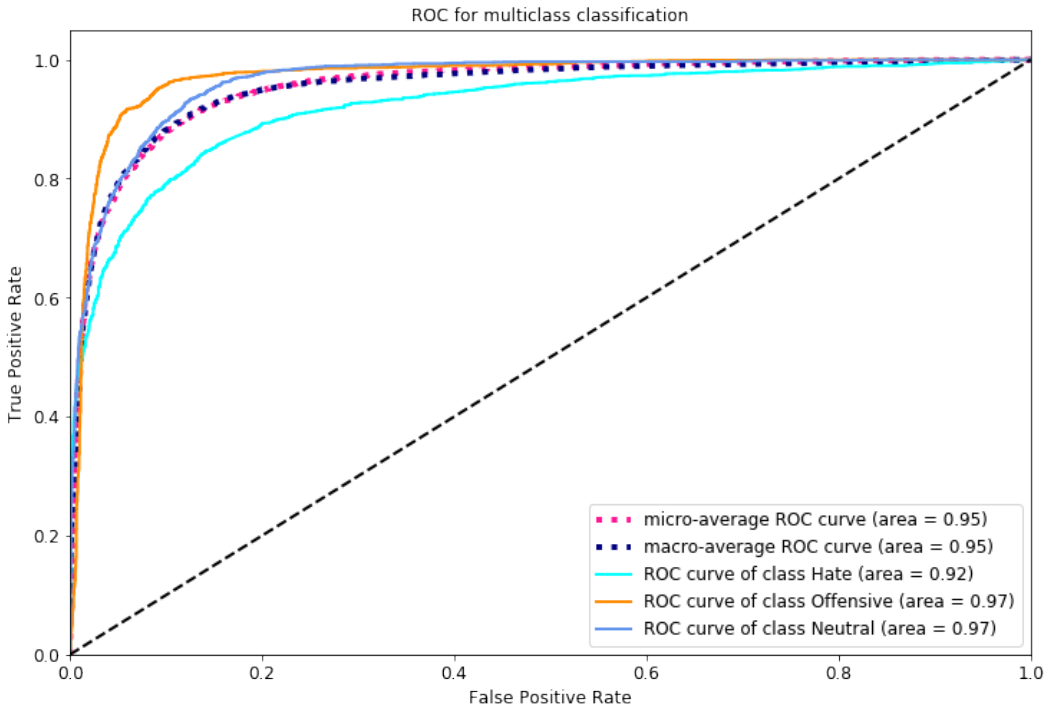

Figure 4.4. ROC for multiclass classification

Table 4.6 Performance metric computations for the baseline model

Accuracy
$Accuracy_{macro}^{test} = \frac{1}{3}\sum_{class}\left(\frac{TP_{class}+TN_{class}}{TP_{class}+TN_{class}+FP_{class}+FN_{class}}\right) = \frac{1}{3}(0.8627+0.9314+0.8929) = 0.8957$
$Accuracy_{overall}^{test} = \sum_{class}\left(\frac{TP_{class}}{TP_{class}+TN_{class}+FP_{class}+FN_{class}}\right) = 0.2376+0.2976+0.3082 = 0.8435$
Precision
$Precision_{micro}^{test} = \frac{\sum_{class=0}^{2} TP_{class}}{\sum_{class=0}^{2}(TP_{class}+FP_{class})} = \frac{1549+1940+2009}{(1549+244)+(1940+213)+(2009+563)} = 0.8435$
$Precision_{macro}^{test} = \frac{1}{3}\sum_{class=0}^{2}\left(\frac{TP_{class}}{TP_{class}+FP_{class}}\right) = \frac{1}{3}\left(\frac{1549}{1549+244}+\frac{1940}{1940+213}+\frac{2009}{2009+563}\right) = 0.8487$
Recall
$Recall_{micro}^{test} = \frac{\sum_{class=0}^{2} TP_{class}}{\sum_{class=0}^{2}(TP_{class}+FN_{class})} = \frac{1549+1940+2009}{(1549+651)+(1940+234)+(2009+135)} = 0.8435$
$Recall_{macro}^{test} = \frac{1}{3}\sum_{class=0}^{2}\left(\frac{TP_{class}}{TP_{class}+FN_{class}}\right) = \frac{1}{3}\left(\frac{1549}{1549+651}+\frac{1940}{1940+234}+\frac{2009}{2009+135}\right) = 0.8445$
F1-Score
$F1score_{micro}^{test} = \frac{2\times Precision_{micro}^{test}\times Recall_{micro}^{test}}{\left(Precision_{micro}^{test}+Recall_{micro}^{test}\right)} = \frac{2\times 0.8435\times 0.8435}{(0.8435+0.8435)} = 0.8435$
$F1score_{macro}^{test} = \frac{1}{3}\sum_{class=0}^{2}\left(\frac{2\times Precision_{class}\times Recall_{class}}{Precision_{class}+Recall_{class}}\right)$ $F1score_{macro}^{test} = \frac{1}{3}\left(\frac{2\times 0.8639\times 0.7041}{0.8639\times 0.7041}+\frac{2\times 0.9011\times 0.8924}{0.9011+0.8924}+\frac{2\times 0.7811\times 0.9370}{0.7811+0.9370}\right) = 0.8415$

5 Multilayer perceptron

Stacking an input layer, one hidden layer with one neuron and one output layer form a simple neural network. A linear combination of features, weights (and biases) is input into the neuron unit which transforms the affine function into one single values as the network output. For a binary classifier with supervised data ($(\boldsymbol{x}, y) \in \boldsymbol{\mathcal{R}}^m \times \boldsymbol{\mathcal{R}}$), the one-neuron network computes the output as follows,

$$\hat{y} = f(w_1 x_1 + w_2 x_2 + \cdots + w_m x_m + b)$$

In the above equation, w_i's are the weights, b is the bias, and f is a non-linear activation function.

Network Architecture

A **multilayer perceptron** (**MLP**) is the combination of multiple neurons distributed (arranged) in multiple hidden layers with each layer containing at least one neuron. All computations in MLP are done sequentially without any cyclic information exchange system such as the recurrent neural networks (discussed later in the report). This non-cyclic graph of sequential operations is called the *feed-forward neural network*. Figure 5.1 shows a sample MLP with depth (number of layers) equal to 4 and width of each layer is 3 (input layer), 2 (first hidden layer), 4 (second hidden layer) and 2 (output layer). Each neuron unit is fully- connected with all the neurons from previous layers, called a dense connection. Note that the activation functions may vary from one hidden layer to another (f^l) but are kept the the same in a given layer. The computations in the graph (Figure 5.1) can also be expressed in vector-matrix notations,

$$\boldsymbol{x} \rightarrow \boldsymbol{a^1} = f^1\left(\boldsymbol{W^1 x + b^1}\right) \rightarrow \boldsymbol{a^2} = f^2\left(\boldsymbol{W^2 a^1 + b^2}\right) \rightarrow \boldsymbol{\hat{y}} = f^3\left(\boldsymbol{W^3 a^2 + b^b}\right)$$

$$\begin{Bmatrix} z_1^1 \\ z_2^1 \end{Bmatrix} = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} + \begin{Bmatrix} b_1^1 \\ b_2^1 \end{Bmatrix} \quad \rightarrow \quad \begin{Bmatrix} a_1^1 \\ a_2^1 \end{Bmatrix} = \begin{Bmatrix} f^1(z_1^1) \\ f^1(z_2^1) \end{Bmatrix}$$

$$\begin{Bmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \\ z_4^2 \end{Bmatrix} = \begin{bmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \\ w_{31}^2 & w_{32}^2 \\ w_{41}^2 & w_{42}^2 \end{bmatrix} \begin{Bmatrix} a_1^1 \\ a_2^1 \end{Bmatrix} + \begin{Bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \\ b_4^2 \end{Bmatrix} \quad \rightarrow \quad \begin{Bmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \\ a_4^2 \end{Bmatrix} = \begin{Bmatrix} f^2(z_1^2) \\ f^2(z_2^2) \\ f^2(z_3^2) \\ f^2(z_4^2) \end{Bmatrix}$$

$$\begin{Bmatrix} z_1^3 \\ z_2^3 \end{Bmatrix} = \begin{bmatrix} w_{11}^3 & w_{12}^3 & w_{13}^3 & w_{14}^3 \\ w_{21}^3 & w_{22}^3 & w_{23}^3 & w_{24}^3 \end{bmatrix} \begin{Bmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \\ a_4^2 \end{Bmatrix} + \begin{Bmatrix} b_1^3 \\ b_2^3 \end{Bmatrix} \quad \rightarrow \quad \begin{Bmatrix} a_1^3 \\ a_2^3 \end{Bmatrix} = \begin{Bmatrix} f^3(z_1^3) \\ f^3(z_2^3) \end{Bmatrix} = \begin{Bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{Bmatrix}$$

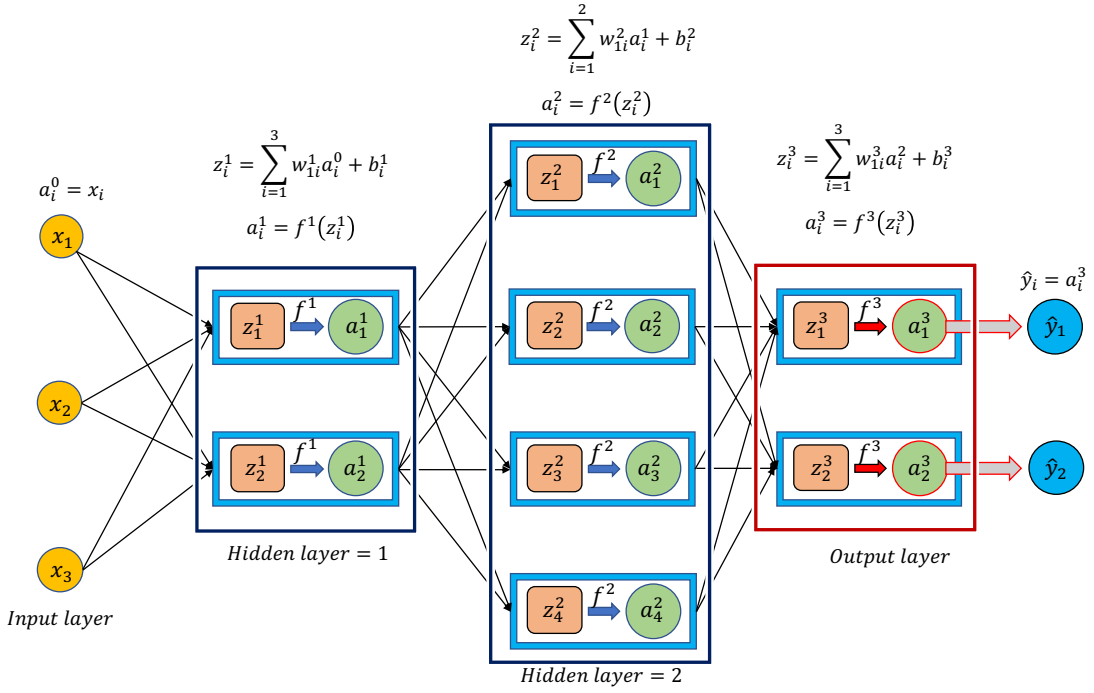

Figure 5.1 Multilayer perceptron with two hidden layers

Backpropagation

Figure 5.2 Input to a neural network can be a simple vector (one training example at a time) or a matrix (batch input). Batch size of the input data, depth of the neural network, and the width of each hidden layer are three common hyperparameters often fine-tuned using k-fold, stratified, cross-validation. For multiclass classification, the final activation unit is generally *a softmax* function which outputs a vector (per input), $a^L = \hat{y}^i \in \mathcal{R}^C$, where C is the total number of classes. Each component of a_c^L is the probability that input x^i belongs to the class, c. The vector component with the highest probability is the final prediction of the MLP for a given input, $arg\ \underset{c}{max}\ a_c^L$.

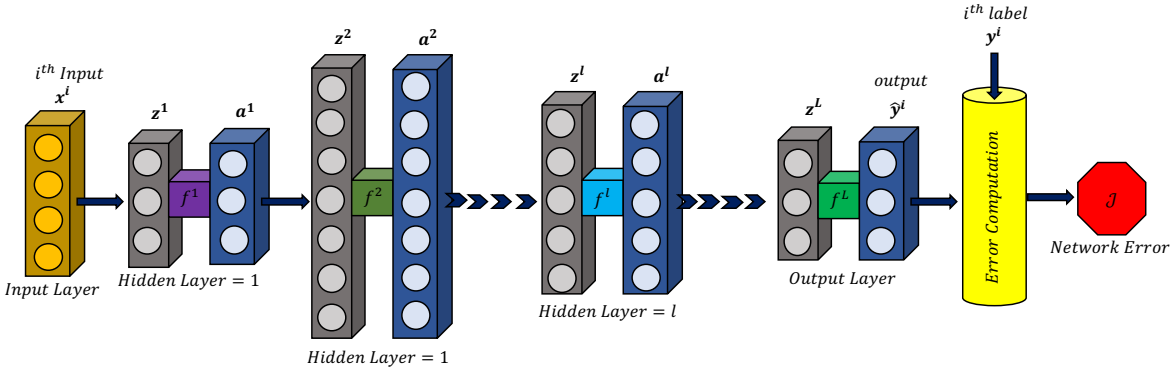

Figure 5.2 MLP architecture with error computation

Figure 5.2 shows the general MLP architecture along with one extra layer added at the right end, for error computation. The layer accepts the network outputs and actual input labels and calculates an error measure ($\mathcal{J}$). In multiclass classifications, **cross-entropy loss function** (also called *log-loss*) is a common error measure. It imposes a heavy penalty on network outputs that are significantly farther from the actual labels. For a batch size of *M*, the cross-entropy loss is computed as,

$$\mathcal{J} = -\sum_{i=1}^{M}\sum_{c=1}^{C} y_c^i \, ln \, \boldsymbol{a_c^{L,i}}$$

If the actual labels are one-hot encoded, the inner summation represents the overall logarithmic-component of the network output vector (a^L). If the network predictions are correct, $then\ a_c^{L,i} \approx 1, ln\, a_c^{L,i} \approx 0$. If the network outputs are far from the expected values, the error magnitude is immense, $a_c^{L,i} \approx 0, ln\, a_c^{L,i} \to -\infty$ (at least in theory). **Backpropagation rule** distributes the final error backward, from output layer towards the input layer, adjusting all the network weights and biases during this propagation to minimize J. The weight adjustment at a given node is directly proportional to the distributed errors in its fully connected units. The weights (and biases) are updated using the gradient descent method, that follows the steepest gradient descent in the weight space $\Delta w_{ij}^l = -\eta \frac{\partial \mathcal{J}}{\partial w_{ij}^l}$. The error differential during backpropagation uses the chain rule,

$$\mathcal{J} = f^{L+1}\left(\boldsymbol{a^L}\right) \to \frac{\partial \mathcal{J}}{\partial \boldsymbol{a^L}} = \begin{bmatrix} \frac{\partial \mathcal{J}}{\partial a_1^L} & \dots & \frac{\partial \mathcal{J}}{\partial a_C^L} \end{bmatrix}$$

$$\boldsymbol{a^L} = \boldsymbol{f^L}\left(\boldsymbol{z^L}\right) \to \frac{\partial \mathcal{J}}{\partial \boldsymbol{z^L}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{a^L}} \cdot \frac{\partial \boldsymbol{a^L}}{\partial \boldsymbol{z^L}} \to \boldsymbol{z^L} = \boldsymbol{W^L a^{L-1}} + \boldsymbol{b^L} = \boldsymbol{g}\left(\boldsymbol{W^L}, \boldsymbol{a^{L-1}}, \boldsymbol{b^L}\right)$$

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{W^L}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{a^L}} \cdot \frac{\partial \boldsymbol{a^L}}{\partial \boldsymbol{z^L}} \cdot \frac{\partial \boldsymbol{z^L}}{\partial \boldsymbol{W^L}} \;\&\; \frac{\partial \mathcal{J}}{\partial \boldsymbol{b^L}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{a^L}} \cdot \frac{\partial \boldsymbol{a^L}}{\partial \boldsymbol{z^L}} \cdot \frac{\partial \boldsymbol{z^L}}{\partial \boldsymbol{b^L}}$$

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{W^{l-1}}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{a^L}} \cdot \frac{\partial \boldsymbol{a^L}}{\partial \boldsymbol{z^L}} \cdot \frac{\partial \boldsymbol{z^L}}{\partial \boldsymbol{a^{L-1}}} \cdot \dots \frac{\partial \boldsymbol{a^{l-1}}}{\partial \boldsymbol{z^{l-1}}} \cdot \frac{\partial \boldsymbol{z^{l-1}}}{\partial \boldsymbol{W^{l-1}}} \;\&\; \frac{\partial \mathcal{J}}{\partial \boldsymbol{b^{l-1}}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{a^L}} \cdot \frac{\partial \boldsymbol{a^L}}{\partial \boldsymbol{z^L}} \cdot \frac{\partial \boldsymbol{z^L}}{\partial \boldsymbol{a^{L-1}}} \cdot \dots \frac{\partial \boldsymbol{a^{l-1}}}{\partial \boldsymbol{z^{l-1}}} \cdot \frac{\partial \boldsymbol{z^{l-1}}}{\partial \boldsymbol{b^{l-1}}}$$

Activation Functions

Activation functions are the most important part of neural network architecture and set them apart from other statistical models such as linear or logistic regression and SVM. The differential of the output of the activation unit with-respect-to the input ($\frac{\partial \boldsymbol{a^l}}{\partial \boldsymbol{z^l}}$) depends on the choice of the activation function. The differential can impact the performance and learning abilities of a neural network.

These days, non-linear activation functions are the standard because of their ability to handle the complex structures in the input data. The four easily accessible activation functions in deep learning algorithms are:

1. **Sigmoid Function**: Sigmoid function is a non-linear function with all real numbers as its domain $(-\infty, +\infty)$ and output range between (0,1). The sigmoid function compresses any real-valued input to

a value between (0,1). The sigmoid function is most common in the binary classifiers with its output representing the probability distributions.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid functions are differentiable everywhere and are monotonic in nature (non-increasing or non-decreasing over the entire domain). The first differential of the sigmoid function is,

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big)$$

The simple differential computation is extremely useful during backpropagation as the sigmoid output was already computed during the forward pass. The derivative of the sigmoid function is bell-shaped and is not monotonic (Figure 5.3).

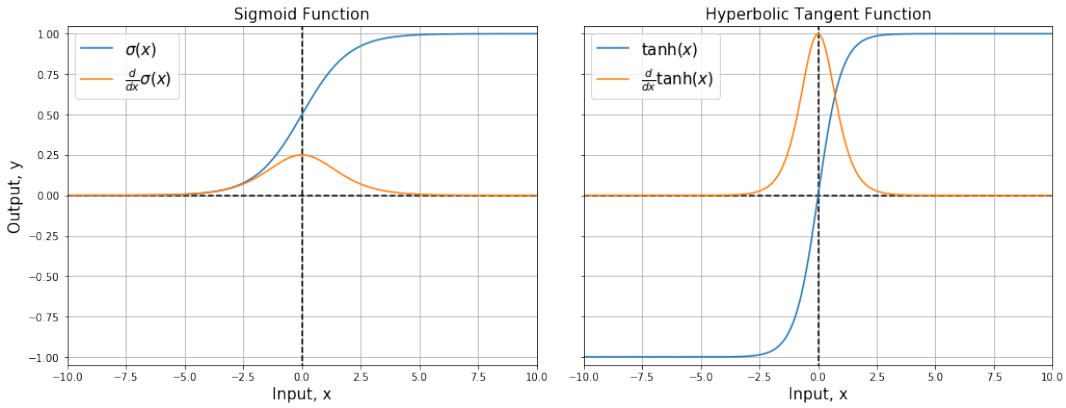

Figure 5.3 Sigmoid and hyperbolic tangent functions

2. **Hyperbolic Tangent Function**: Hyperbolic functions are similar to the sigmoid functions with real-valued inputs compressed between the range (-1, 1). The models with hyperbolic tangent functions can outperform the sigmoid activation units and are relatively easy to train. The hyperbolic functions are written as,

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} = 2\sigma(2x) - 1$$

The function is differentiable everywhere and monotonic in nature with differential,

$$tanh'(x) = 1 - tanh^2(x)$$

Compared to the sigmoid function, the hyperbolic functions are more sensitive around the neighborhood of zero. Figure 5.3 shows the plots of *tanh(x)* and its derivative.

Figure 5.3 also highlights the two limitations of sigmoid and hyperbolic tangent functions. Both functions can quickly saturate i.e., as all large positive and negative values 1 and 0 (for *sigmoid*) or -1 (for *tanh*), respectively the network is unable to learn anything during the backpropagation. Both

activation functions are only sensitive (small change is easily detected) around 0.5 (for *sigmoid)* or 0 (for *tanh*), any changes near the outer domain are almost undetectable.

3. **Rectified Linear Function**: Rectified Linear function (ReL) is a non-linear function and the neuron with ReL as its activation function is called Rectified Linear Unit (ReLU). ReLU act as a filter that only allows the positive values of input to pass through, replacing all negative values with zeros.

$$ReLU(x) = max(0, x), ReLU'(x) = \begin{cases} 1, \forall x > 0 \\ NaN, x = 0 \\ 0 \; \forall x < 0 \end{cases}$$

ReLU is most used activation function neural networks. The output range of ReLU is between zero and $+\infty$, and it is non-differentiable at zero. The differential of ReLU is a simple step function making the backpropagation extremely fast.

Various modified versions of ReLU function are available, to improve the model performance, such as $ReLU_{Leaky}$ where the negative values are in the neighborhood of but not exactly zero. ReLULeaky is preferred when a network is unable to learn from the data with a large count of negative values replaced by zeros (saturation issue). Both ReLU and $ReLU_{Leaky}$, along with their respective derivatives, are monotonic.

$$ReLU_{Leaky}\ (x) = \begin{cases} x \; \forall \; x \geq 0 \\ ax \; \forall \; x < 0 \end{cases}, ReLU'_{Leaky}\ (x) = \begin{cases} 1 \; \forall \; x \geq 0 \\ a \; \forall \; x < 0 \end{cases}$$

In the above equation, slope parameter *a* is usually small (0.001).

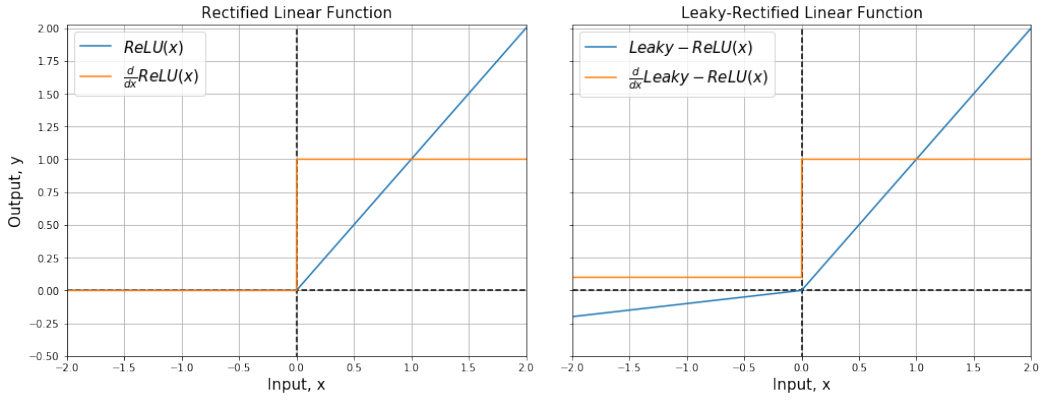

Figure 5.4 ReLU and $ReLU_{Leaky}$

4. **Softmax Function**: *Sigmoid activation functions* are mostly limited in their applications to binary tasks, and the *softmax* function is more appropriate for the multiclass classifier. The input vectors projected onto an exponential space and normalized. Exponentiation increases the relative distance between the input components and normalization compresses all outputs within the range 0 to 1, which is more relatable to probabilities.

$$softmax(\boldsymbol{x})_i = P(y = i|\boldsymbol{x}) = \frac{e^{\boldsymbol{W}_i \boldsymbol{x}}}{\sum_{c=1}^{C} e^{\boldsymbol{W}_c \boldsymbol{x}}}$$

In the above equation, $\boldsymbol{W}$ is the weight matrix with each row corresponding to a class, $\boldsymbol{x}$ is the input vector. The *softmax* function calculates the probability of given input ($\boldsymbol{x}$) belonging to a class i ($1 < i < C$), Figure 5.5. The input-output size of the *softmax* function is the same as the total number of classes, C. All components of the output must sum to 0.

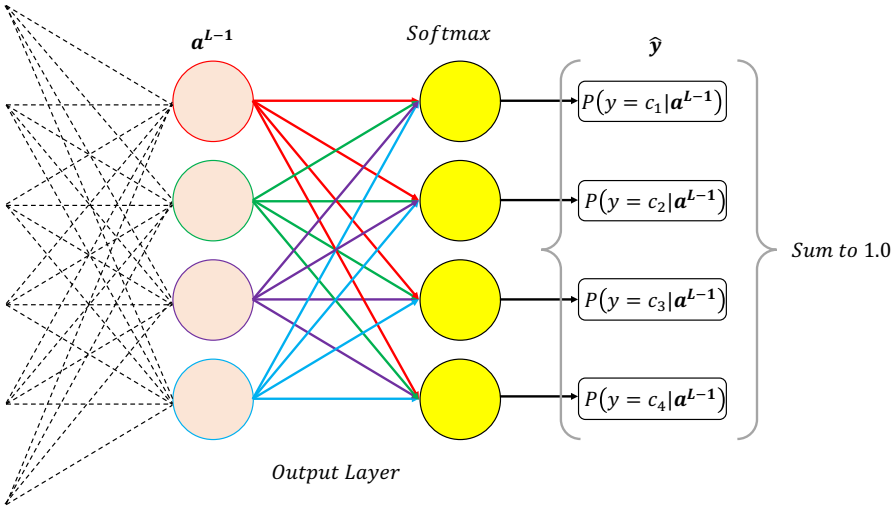

Figure 5.5 *Softmax* activation function

Regularization

Neural networks tend to overfit on the training data, and therefore, multiple regularization techniques are available to make the model more reliable.

Overfitting is the ability of a machine learning model to overly adapt to the training data resulting in poor performance during the test phase. An overfitted model would have low training error but high-test error and lacks generalization. Regularization methods in neural networks penalized the weight matrices. Three extensively employed regularization-techniques in this study were:

1. **Dropout**: Dropout layers randomly select and delete a fraction of neurons from the system, along with all incoming and outgoing connections to them. The dropout fraction is a hyperparameter referred to as the dropout layer parameter.

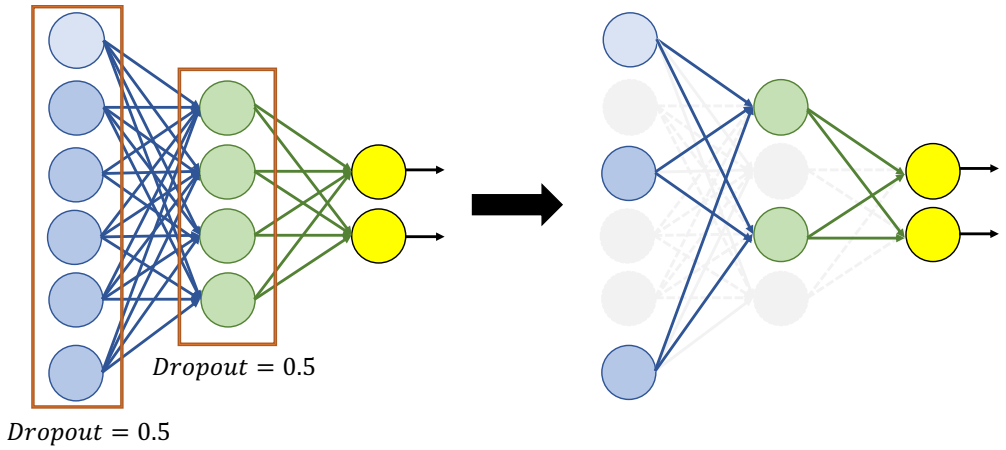

Figure 5.6 Before and after dropout layer

2. **Kernel Regularization**: Kernel regularization penalizes the weight matrices by including them in the loss function calculation. Loss functions represent the overall model fit and are modified as follows,

$$\mathcal{J}_{new} = \mathcal{J} + \lambda \times \|w\|_{norm}$$

The two norms used for regularization are L1 and L2-norm. L1 regularization penalizes the absolute values of the weight, whereas the L2 criterion penalizes the sum-of-squares. L2-norms can learn intricate patterns in the data compared to L1-norm. However, L1-norm is interpretable and straightforward. Both norms include multiplication with the regularization parameter (λ), which controls the extent of penalization.

$$\mathcal{J}_{new} = \begin{cases} \mathcal{J} + \lambda \sum |w_i| \rightarrow L_1 - norm \\ \mathcal{J} + \lambda \sum w_i^2 \rightarrow L_2 - norm \end{cases}$$

3. **Validation Sets**: In neural networks, *early stopping* is included with the validation score to guide how much to train the model. In this study, *early stopping* with two parameters was used: *monitor* and *patience*. Validation loss was used as the monitor to track network performance per epoch. The patience parameter was used to wait out any local features impacting the validation loss. If validation loss did not show any reduction over three consecutive runs, the model execution was interrupted, and the model fit with minimum validation loss was the best model.

MLP Model for Tweet Classification

The data was split into the training-validation-test set ratio $(^4/_3 : ^1/_3 : 1)$ was shown in Figure 2.4, followed by word embedding, section 3.2.2. A sequential model in python's Keras package with dense layers was used for the MLP classifier architecture. Grid search on different hyperparameters was done to select the best fit model. Table 5.1, Table 5.2, and Table 5.3 show the model performance under different combinations of the network architectural hyperparameters. The classifier's performance on the validation set was not too sensitive to these parameters. An MLP network with three hidden layers containing 128-> 64 ->16 neurons, input batch size of 128 and ReL activation function was selected. The selection criterion was to keep the number of parameters (computation time) low without significantly affecting the model performance. Code Section 5.1 shows the model fit on the validation set. The selected model architecture had validation accuracy score of 0.81.

Table 5.1 Fine-tuning the number of hidden layers and neurons per hidden layer

#Neurons in hidden layer 1	#Neurons in hidden layer 2	#Neurons in hidden layer 3	#Neurons in hidden layer 4	#Neurons in hidden layer 5	F1-score
512	256	128	64	16	0.79
256	128	64	16	8	0.80
128	64	32	16	8	0.78
64	32	16	8	x	0.79
128	64	16	8	x	0.80
128	64	16	x	x	0.80
128	32	8	x	x	0.79

Table 5.2 Batch size hyperparameters

#Neurons in hidden layer 1	#Neurons in hidden layer 2	#Neurons in hidden layer 3	Batch size	F1-score
128	64	16	512	0.79
128	64	16	256	0.80
128	64	16	128	0.80
128	64	16	64	0.80
128	64	16	8	0.80

Table 5.3 Model performance for ReLU and ReLULeaky

#Neurons in hidden layer 1	#Neurons in hidden layer 2	#Neurons in hidden layer 3	Activation function	F1-score
128	64	16	ReLU	0.80
128	64	16	$ReLU_{Leaky}$ (slope=0.001)	0.80
128	64	16	$ReLU_{Leaky}$ (slope=0.01)	0.80

Code Section 5.1 MLP Model fit

Validation Fit – batch size = 128, Monitor – Validation loss, Pateince = 2

```
Train on 8865 samples, validate on 4367 samples
Epoch 1/30
 - 5s - loss: 0.6996 - acc: 0.7032 - val_loss: 0.5239 - val_acc: 0.7880
Epoch 2/30
 - 5s - loss: 0.2668 - acc: 0.9058 - val_loss: 0.5233 - val_acc: 0.8067
Epoch 3/30
 - 5s - loss: 0.0961 - acc: 0.9726 - val_loss: 0.6020 - val_acc: 0.8010
Epoch 4/30
 - 5s - loss: 0.0340 - acc: 0.9920 - val_loss: 0.6780 - val_acc: 0.8063
```

Dropout layer, weight constraints, and kernel weight regularizer were included in the model to reduce overfitting. Code Section 5.2 shows the grid search results and the best model selected on the basis of accuracy score. Adding the dropout layer increased the model performance while the magnitude of weight regularizer did not have any considerable effect.

Code Section 5.2 Gridsearch on overfitting hyperparameters

```
param_grid = dict(dropout_rate=[0.0,0.5,
                  weight_constraint=[1,2,3],
                  kernel_reg = [0,0.01,0.001])
```

```
Best: 0.797406 using {'dropout_rate': 0.5, 'kernel_reg': 0.001, 'weight_constraint': 3}
0.778116 (0.011551) with: {'dropout_rate': 0.0, 'kernel_reg': 0.0, 'weight_constraint': 1}
0.786689 (0.008572) with: {'dropout_rate': 0.0, 'kernel_reg': 0.0, 'weight_constraint': 2}
0.785900 (0.007050) with: {'dropout_rate': 0.0, 'kernel_reg': 0.0, 'weight_constraint': 3}
0.772025 (0.009882) with: {'dropout_rate': 0.0, 'kernel_reg': 0.01, 'weight_constraint': 1}
0.774168 (0.011153) with: {'dropout_rate': 0.0, 'kernel_reg': 0.01, 'weight_constraint': 2}
0.772927 (0.015919) with: {'dropout_rate': 0.0, 'kernel_reg': 0.01, 'weight_constraint': 3}
0.786802 (0.002228) with: {'dropout_rate': 0.0, 'kernel_reg': 0.001, 'weight_constraint': 1}
0.786802 (0.009087) with: {'dropout_rate': 0.0, 'kernel_reg': 0.001, 'weight_constraint': 2}
0.780147 (0.012834) with: {'dropout_rate': 0.0, 'kernel_reg': 0.001, 'weight_constraint': 3}
0.797067 (0.009298) with: {'dropout_rate': 0.5, 'kernel_reg': 0.0, 'weight_constraint': 1}
0.796390 (0.007143) with: {'dropout_rate': 0.5, 'kernel_reg': 0.0, 'weight_constraint': 2}
0.797293 (0.010324) with: {'dropout_rate': 0.5, 'kernel_reg': 0.0, 'weight_constraint': 3}
0.791427 (0.009202) with: {'dropout_rate': 0.5, 'kernel_reg': 0.01, 'weight_constraint': 1}
0.793119 (0.005565) with: {'dropout_rate': 0.5, 'kernel_reg': 0.01, 'weight_constraint': 2}
0.784884 (0.018462) with: {'dropout_rate': 0.5, 'kernel_reg': 0.01, 'weight_constraint': 3}
0.789397 (0.006603) with: {'dropout_rate': 0.5, 'kernel_reg': 0.001, 'weight_constraint': 1}
0.796954 (0.009471) with: {'dropout_rate': 0.5, 'kernel_reg': 0.001, 'weight_constraint': 2}
0.797406 (0.007111) with: {'dropout_rate': 0.5, 'kernel_reg': 0.001, 'weight_constraint': 3}
```

Figure 5.7 shows the final MLP architecture selected for the tweet classification with model fit and performance details in Code Section 5.3. The validation accuracy corresponding to min validation loss was about 0.81. The MLP model slightly underperformed than the baseline model with performance metrics about 2% less.

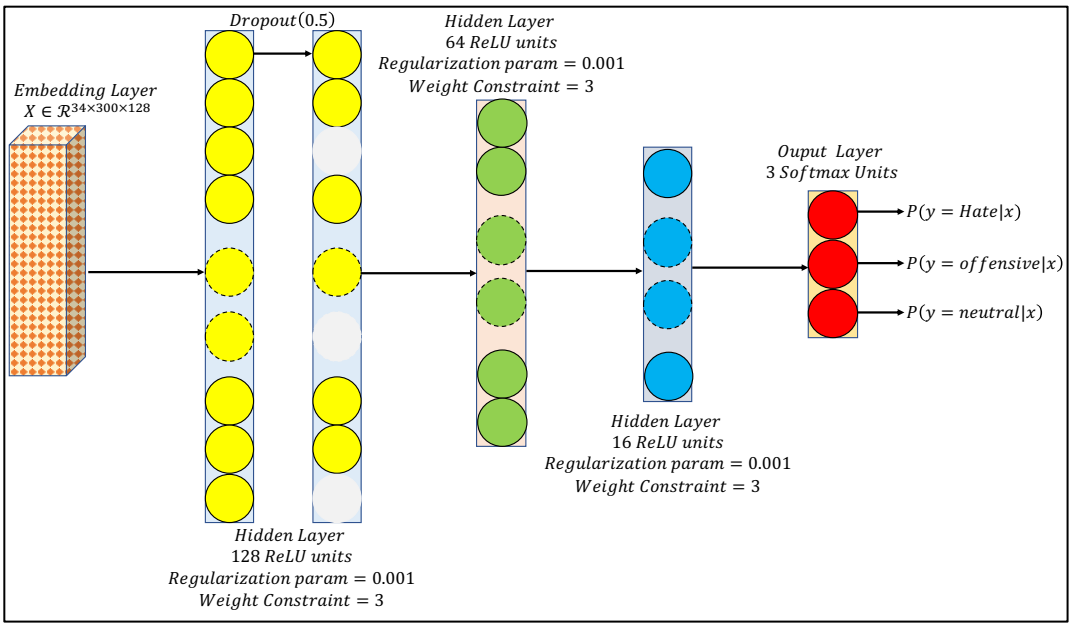

Figure 5.7 Final MLP architecture

Code Section 5.3 MLP Model fit

Model Summary

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_9 (Embedding)      (None, 34, 300)           5580600
flatten_9 (Flatten)          (None, 10200)             0
dense_21 (Dense)             (None, 128)               1305728
dropout_6 (Dropout)          (None, 128)               0
dense_22 (Dense)             (None, 16)                2064
dense_23 (Dense)             (None, 3)                 51
=================================================================
Total params: 6,888,443
Trainable params: 6,888,443
Non-trainable params: 0
```

Model Fit

```
Train on 8865 samples, validate on 4367 samples
Epoch 1/30
 - 7s - loss: 1.0195 - acc: 0.6347 - val_loss: 0.7491 - val_acc: 0.7859
Epoch 2/30
 - 6s - loss: 0.6289 - acc: 0.8371 - val_loss: 0.6839 - val_acc: 0.8125
Epoch 3/30
 - 6s - loss: 0.4379 - acc: 0.9168 - val_loss: 0.7121 - val_acc: 0.8154
Epoch 4/30
 - 6s - loss: 0.3521 - acc: 0.9454 - val_loss: 0.7323 - val_acc: 0.8104
```

Classification Report

	f1-score	precision	recall	roc_auc	support
Hate	0.771448	0.776601	0.766364	0.905481	2200.0
Offensive	0.852106	0.848222	0.856026	0.951435	2174.0
Neutral	0.832674	0.830934	0.834422	0.950097	2144.0
micro avg	0.818656	0.818656	0.818656	0.938013	6518.0
macro avg	**0.818743**	0.818586	0.818937	0.935671	6518.0
weighted avg	0.818490	0.818361	0.818656	0.935484	6518.0
samples avg	0.818656	0.818656	0.818656	0.889230	6518.0

6 Convolution Neural Networks

Originally developed for computer vision tasks like image classifications, Convolution Neural Networks (CNNs) have been successful in natural language processing. CNNs are extremely versatile in handling higher-order tensors (color images are 3rd order tensor). In language processing, the inputs are word vectors, one-hot encoded or dense representations. Figure 6.1 shows a powerful pre-trained deep CNN model [15]. A CNN can be an entire neural network in itself or a small component of a larger neural network.

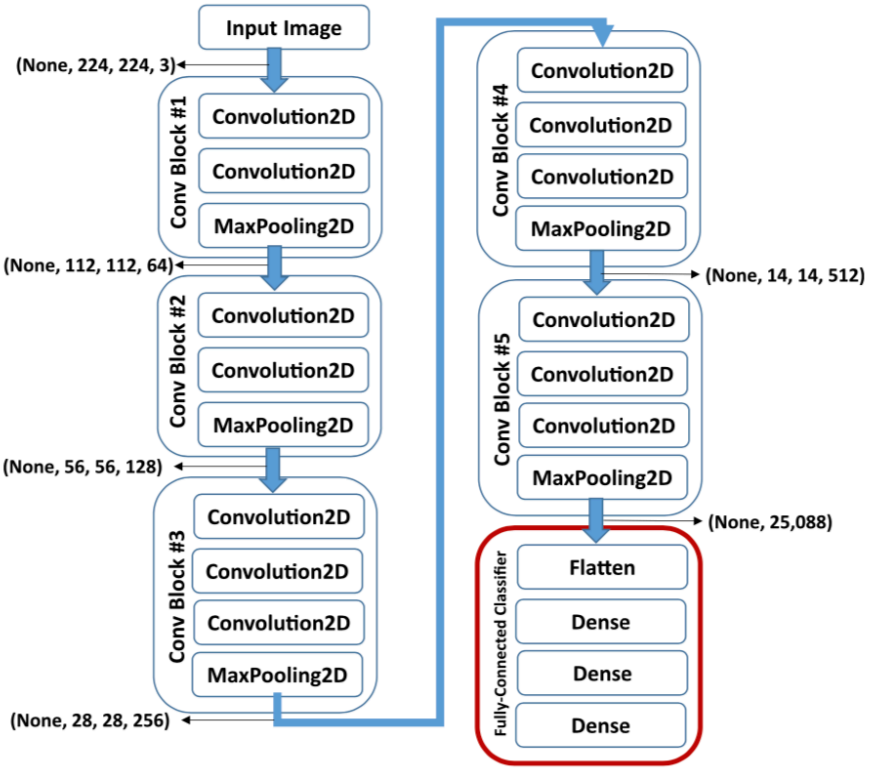

Figure 6.1. A schematic of the VGG-16 Deep Convolutional Neural Network (DCNN) architecture trained on ImageNet database [16]

Convolution Layer

Convolution is a mathematical operation on two functions to produce a third function that expresses how the shape of one is modified by interacting with the other function [17]. An input $x^k \in \mathcal{R}^{M^k \times N^k}$ to a convolution layer $f^k \in \mathcal{R}^{M \times N \times F}$ would results in an output tensor $x^{k+1} \in \mathcal{R}^{M^{k+1} \times N^{k+1} \times F^{k+1}}$. The convolution layer is projected on top of the part of the input matrix. The convolution includes element-wise multiplication followed by their addition to produce a single number (a component of x^{k+1}). A bias is added to the output. In the next step, the kernel strides to the right along the row (or downwards along the column).

Figure 6.2 and Figure 6.3 show the convolution process with stride-values of 1 and 2, respectively. When the stride is 1, the model convolves on every single spatial location of the input. If the convolution stride

set to 2 (>1), the kernel would skip 1 (*s-1*) pixels in the next step. From Figure 6.2 and Figure 6.3, it is clear that stride size affects the size of the output. In Figure 6.3, $x^{k+1} \in \mathcal{R}^{2\times2}$, with the second row and column of operations skipped from the pipeline. For text classification, the convolution process remains the same except the layers are one-dimensional that extract information from local one-dimensional data chunks, 1-D covnets.

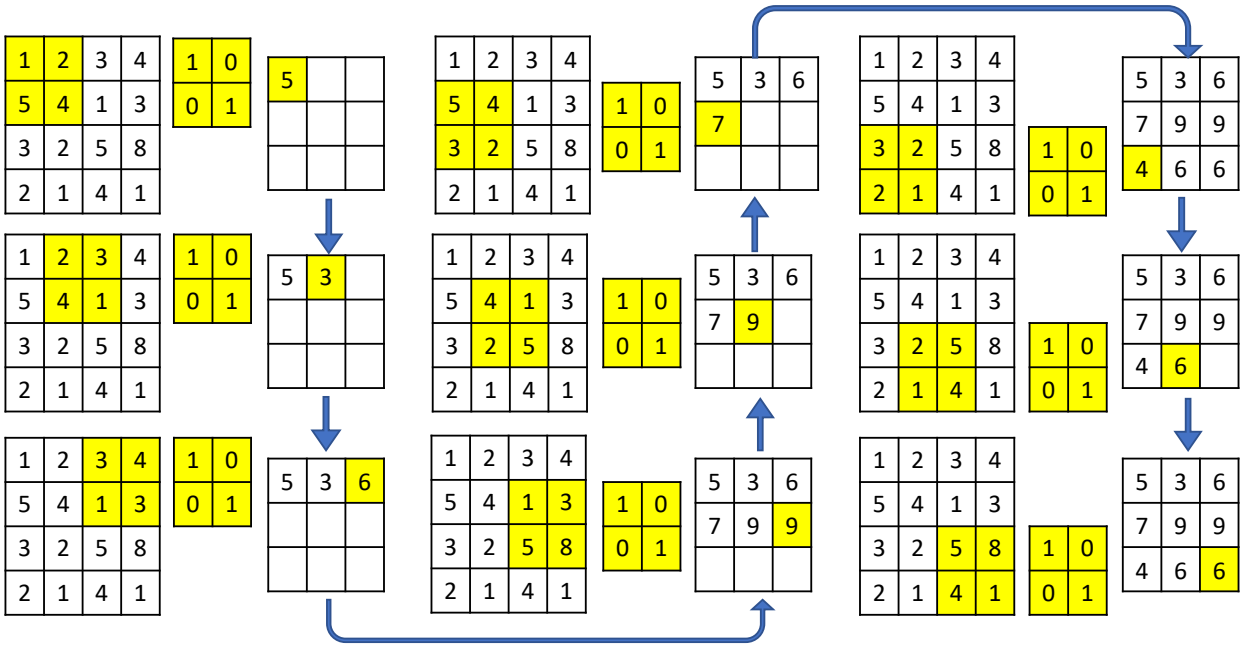

Figure 6.2 Convolution process (with stride 1)

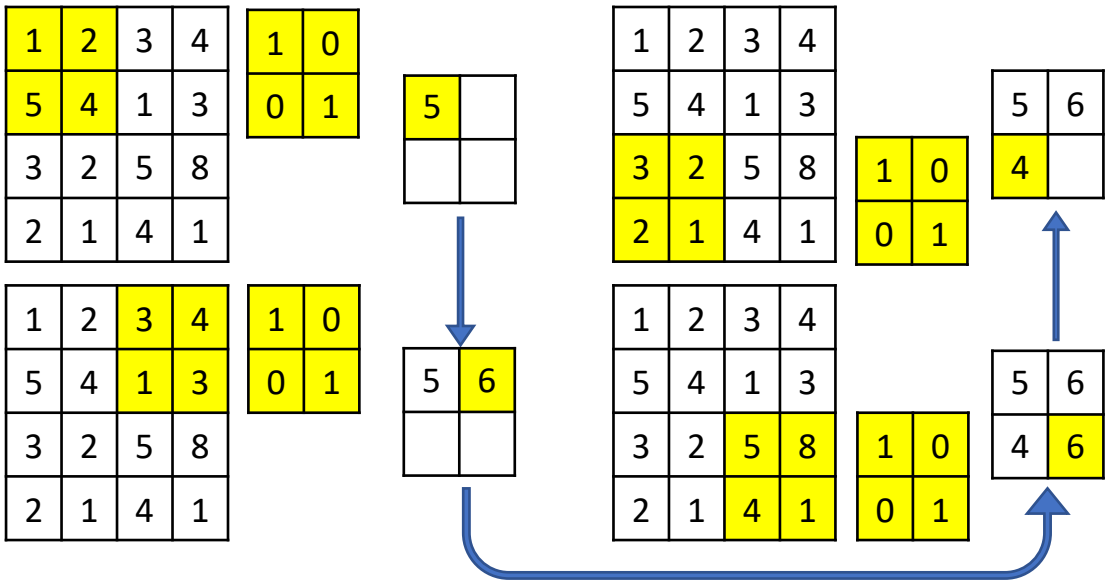

Figure 6.3 Convolution process (with stride 2)

ReLU Layer

Rectified-Linear Unit (ReLU) is a simple non-linear function which has proven to be effective in deep learning applications (section 5.3).

$$x_i^{k+1} = ReLU\left(x_i^k\right) = max\left(0, x_i^k\right)$$

ReLU is an element-wise operation that preserves the dimension of the input tensor and copies non-negative values in the output vector, with zeros for the rest of the components, Figure 6.4. ReLU layer does not have any parameter to train, which results in faster computations during the backward propagation.

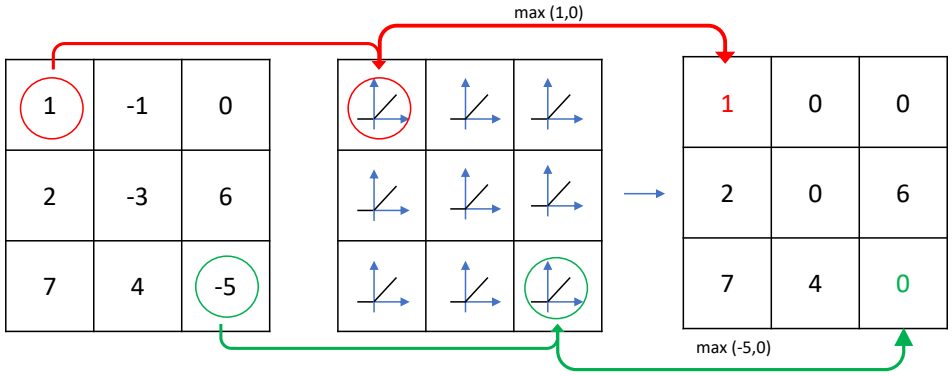

Figure 6.4 ReLU activation layer

Pooling Layer

The pooling operation is similar to the convolution process. A portion (filter -size) of the input (x^k) passes through a filter-screen, which aggregates the information into one single value. The output can either be the maximum value of data chuck or its overall average. The filter then moves a specified amount (stride-size) to the right. Depending on the aggregation function, the pooling layer is called the max-pooling layer or the average pooling layer. Pooling layer does not have any parameters to train. An input, $x^k \in \mathcal{R}^{M^k \times N^k}$ when passed through a pooling layer $f^k \in \mathcal{R}^{M \times N}$, outputs $x^{k+1} \in \mathcal{R}^{M^{k+1} \times N^{k+1}}$ where $(M^{k+1}, N^{k+1}) = \left(\frac{M^k}{M}, \frac{N^k}{N}\right)$. Figure 6.5 shows the application of a 2x2 max pooling to a 4x4 input tensor resulting in a 2x2 output. Same as 1-D convolution layer, one-dimensional pooling layers can process the sequence data.

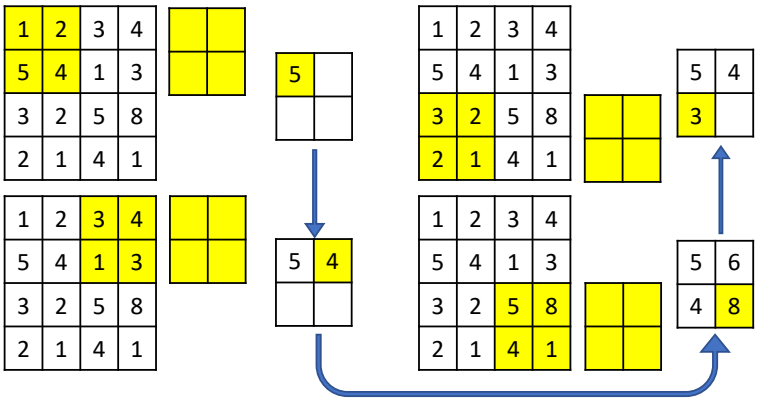

Figure 6.5 Max-Pooling layer

CNN for Tweet Classification

Figure 6.6 shows three 1D *covnets* with $128 \rightarrow 64 \rightarrow 32$ filters of size 4. The output from convolution layers was input to a flat layer which was connected to a dense layer, followed by the final output layer with a *softmax* activation function. Kernel weight regularizer and dropout layer were used to evaluate the model overfitting. Based on the results of the grid search, model with a dropout rate of 0.5 and without any weight regularization was the best fit on the validation dataset.

Code Section 6.1 shows the outputs using 5-fold, stratified cross-validation method followed by a separate Figure 2.4evaluation on the validation set for improving the validation accuracy and minimizing the validation loss. Weight constraint was set to a value of 3. The accuracy score based on the cross-validation data was about 0.84. Higher dropout rate resulted in better model performance, whereas an increase in weight penalization leads to a slight decrease in the accuracy score. Figure 6.6 shows the final CNN architecture selected for the tweet classification with model fit and performance details in

. The validation accuracy of the best fit model was 0.85. Macro F1-score on the test data was 0.84 same as F1-score of the gradient boosting model. Figure 6.7 shows the confusion matrices of the test results. Compared to the baseline model, the CNN model had a higher number of true-positives under the hate-category. The offensive class accuracy was almost comparable while the neutral class true-positives were slightly lower.

Code Section 6.1 Gridsearch on overfitting hyperparameters

```
param_grid = dict(dropout_rate=[0.0,0.25,0.5],
                  kernel_reg = [0,0.01,0.001])
```

```
Best: 0.835195 using {'dropout_rate': 0.5, 'kernel_reg': 0.0}
0.818951 (0.007960) with: {'dropout_rate': 0.0, 'kernel_reg': 0.0}
0.808009 (0.014968) with: {'dropout_rate': 0.0, 'kernel_reg': 0.01}
0.818274 (0.009907) with: {'dropout_rate': 0.0, 'kernel_reg': 0.001}
0.827524 (0.005984) with: {'dropout_rate': 0.25, 'kernel_reg': 0.0}
0.824027 (0.010713) with: {'dropout_rate': 0.25, 'kernel_reg': 0.01}
0.822222 (0.010123) with: {'dropout_rate': 0.25, 'kernel_reg': 0.001}
0.835195 (0.007843) with: {'dropout_rate': 0.5, 'kernel_reg': 0.0}
0.823689 (0.008979) with: {'dropout_rate': 0.5, 'kernel_reg': 0.01}
0.830344 (0.007704) with: {'dropout_rate': 0.5, 'kernel_reg': 0.001}
```

Code Section 6.2 CNN Model fit

Model Summary

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 34, 300)           5580600
conv1d_4 (Conv1D)            (None, 31, 128)           153728
max_pooling1d_4 (MaxPooling1 (None, 15, 128)           0
dropout_4 (Dropout)          (None, 15, 128)           0
conv1d_5 (Conv1D)            (None, 12, 64)            32832
max_pooling1d_5 (MaxPooling1 (None, 6, 64)             0
conv1d_6 (Conv1D)            (None, 3, 32)             8224
max_pooling1d_6 (MaxPooling1 (None, 1, 32)             0
flatten_2 (Flatten)          (None, 32)                0
dense_3 (Dense)              (None, 10)                330
dense_4 (Dense)              (None, 3)                 33
=================================================================
Total params: 5,775,747
Trainable params: 5,775,747
Non-trainable params: 0
```

Model Fit

```
Train on 8865 samples, validate on 4367 samples
Epoch 1/20
8865/8865 [==============================] - 10s 1ms/step - loss: 0.8434 - acc: 0.6232 - val
_loss: 0.5723 - val_acc: 0.7715
Epoch 2/20
8865/8865 [==============================] - 9s 1ms/step - loss: 0.4718 - acc: 0.8176 - val_
loss: 0.4225 - val_acc: 0.8390
Epoch 3/20
8865/8865 [==============================] - 9s 1ms/step - loss: 0.3185 - acc: 0.8840 - val_
loss: 0.3982 - val_acc: 0.8576
Epoch 4/20
8865/8865 [==============================] - 9s 1ms/step - loss: 0.2322 - acc: 0.9192 - val_
loss: 0.3976 - val_acc: 0.8631
Epoch 5/20
8865/8865 [==============================] - 9s 1ms/step - loss: 0.1625 - acc: 0.9462 - val_
loss: 0.4227 - val_acc: 0.8587
Epoch 6/20
8865/8865 [==============================] - 9s 1ms/step - loss: 0.1169 - acc: 0.9597 - val_
loss: 0.4947 - val_acc: 0.8555
```

Classification Report

	f1-score	precision	recall	roc_auc	support
Hate	0.782998	0.863961	0.715909	0.915763	2200.0
Offensive	0.886892	0.888325	0.885465	0.967294	2174.0
Neutral	0.850171	0.785601	0.926306	0.962097	2144.0
micro avg	0.841669	0.841669	0.841669	0.948805	6518.0
macro avg	**0.840020**	0.845962	0.842560	0.948384	6518.0
weighted avg	0.839746	0.846312	0.841669	0.948191	6518.0
samples avg	0.841669	0.841669	0.841669	0.906643	6518.0

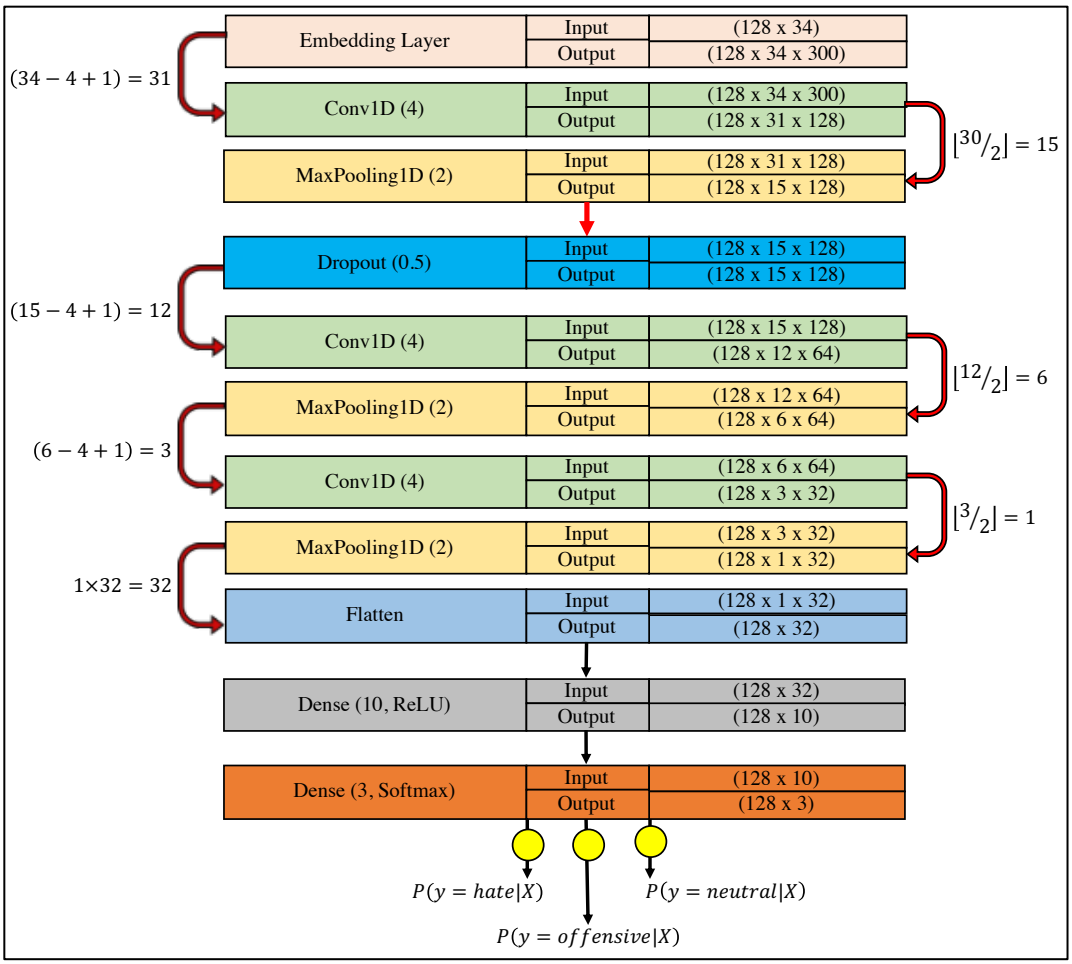

Figure 6.6 Final CNN architecture for tweet classification

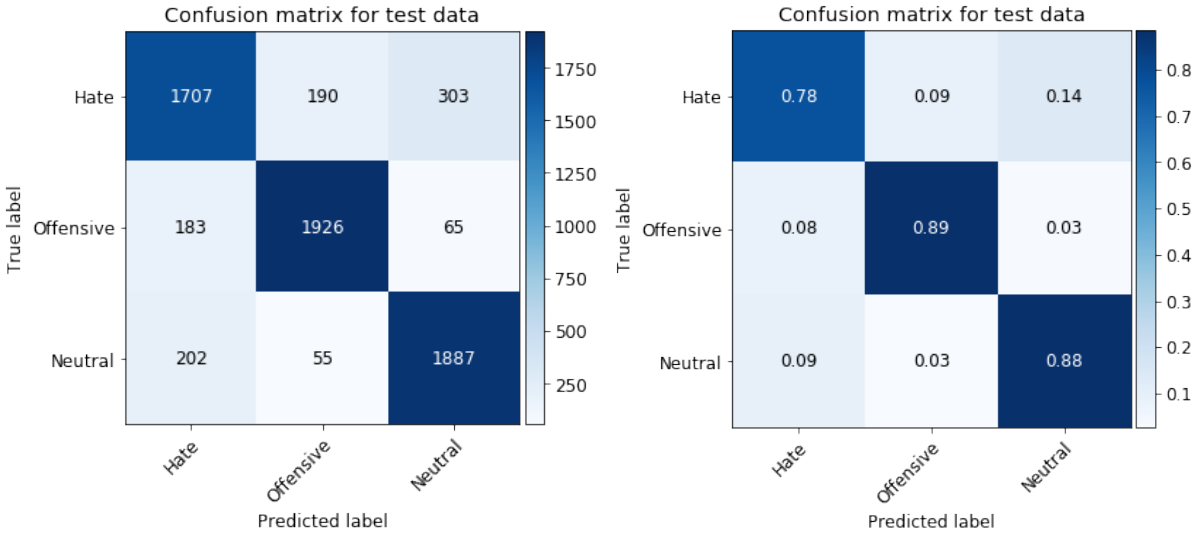

Figure 6.7 Confusion matrices (actual counts and normalized)

7 Recurrent Neural Networks

Text data is similar to a time series where information is gathered sequentially at each time step, and the order of time steps is crucial. The MLP and CNN models process the text in its entirety, i.e., the network can view an entire input text before any transformation. Two-word vectors ‘interact' only via randomly assigned weights, updated during the backpropagation. Each text is an independent input with no direct flow of information between the text tokens. MLP and CNNs do not treat the text data explicitly as a sequence. Recurrent Neural Networks (RNNs) are a special class of the neural networks developed to process the input data as a sequence and retain the information learned during each step. Figure 7.1 shows a simple recurrent neural network. At any step *t*, the activation unit receives two inputs:

1. token (x_i^t) in the text sequence, and
2. activation unit-output from the previous $(t-1)^{th}$ step (a^t).

The affine function of inputs, weights, and biases from previous steps (W_{aa}, W_{ax}, b_a) are transformed using non-linear activation functions (f). For the very first input, the activation unit (a^0) is an array of zeros. The entire process can is expressed in one equation,

$$a^t = f\left(W_{aa}a^t + W_{ax}x_i^t + b_a\right)$$

For the multi-class classification problems, the last unit usually has one more non-linear activation unit (e.g., *softmax* function), Figure 7.2.

$$a^{L_x} = f\left(W_{aa}a^{L_x-1} + W_{ax}x_i^{L_x} + b_a\right)$$

$$\hat{y}_i^k = softmax\left(W_{ya}a^{L_x} + b_a\right) = \frac{exp\left(W_{ya}a_k^{L_x} + b_a\right)}{\sum_{l=1}^{C} exp\left(W_{ya}a_l^{L_x} + b_a\right)}$$

For each data point error between the predicted and true labels is computed. The total loss (cross-entropy loss) per input sequence is minimized by updating the weights and biases using *backpropagation through time*,

$$\mathcal{J}(y_i, \hat{y}_i) = -\sum_{i=1}^{C} y_i \ln \hat{y}_i \rightarrow \frac{\partial \mathcal{J}(y_i, \hat{y}_i)}{\partial W}$$

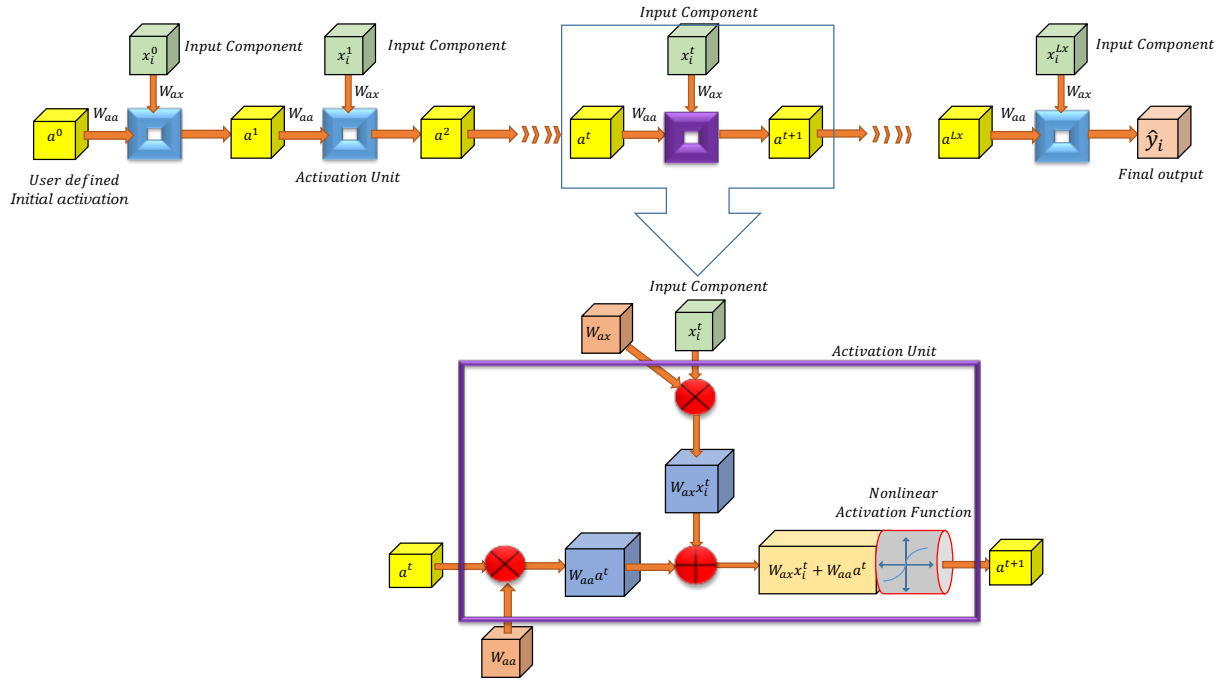

Figure 7.1 Simple Recurrent Neural Network (step *t*)

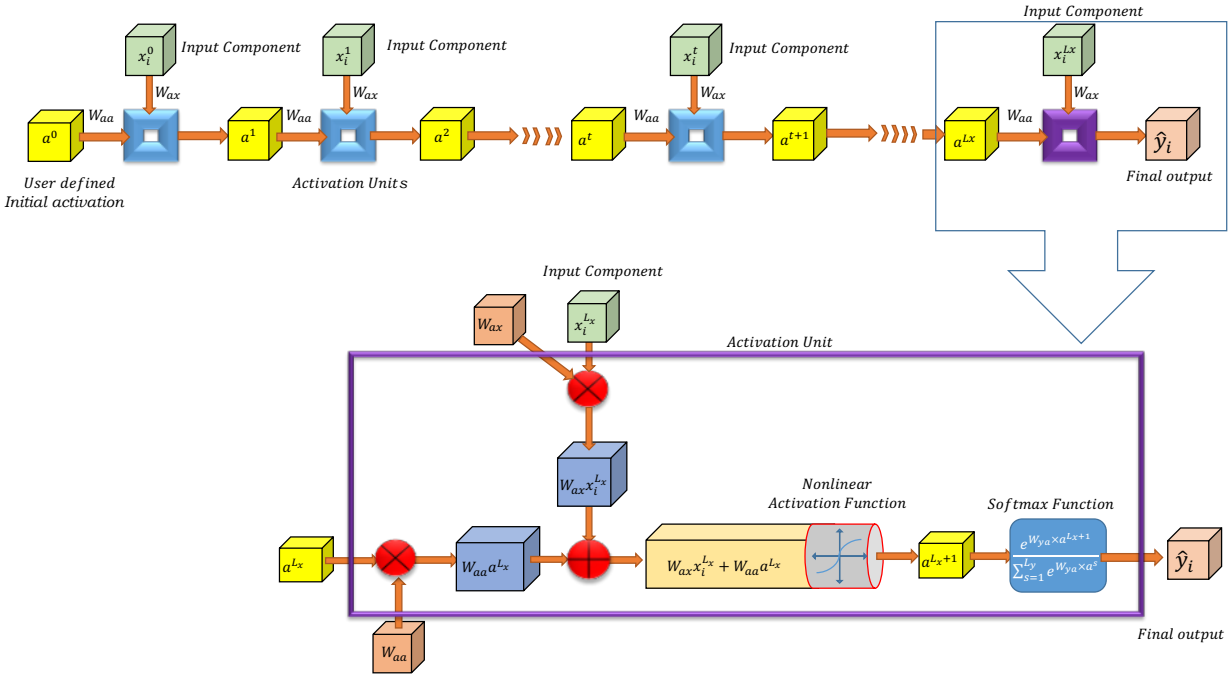

Figure 7.2 Simple Recurrent Neural Network (step L_x)

Types of RNNs

Consider two (neutral) tweets:

1. "It is a sunny morning in the woods."
2. "Sam is planning to buy the groceries today."

$$\boldsymbol{x}_1 = \begin{Bmatrix} It \\ is \\ a \\ sunny \\ morning \\ in \\ the \\ woods \end{Bmatrix} = \begin{Bmatrix} x_1^0 \\ x_1^1 \\ x_1^2 \\ x_1^3 \\ x_1^4 \\ x_1^5 \\ x_1^6 \\ x_1^7 \end{Bmatrix}, \boldsymbol{y}_1 = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}, \boldsymbol{x}_2 = \begin{Bmatrix} Sam \\ is \\ planning \\ to \\ buy \\ the \\ groceries \\ today \end{Bmatrix} = \begin{Bmatrix} x_2^0 \\ x_2^1 \\ x_2^2 \\ x_2^3 \\ x_2^4 \\ x_2^5 \\ x_2^6 \\ x_2^7 \end{Bmatrix}, \boldsymbol{y}_2 = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}$$

Each tweet, as an input vector, had eight components (x_i^t). Assume there exists a vocabulary corpus (size V) for each word in the English language. The input vector components were one-hot encoded: binary, sparse vectors (of length V) represented the words and then condensed to obtain the word embeddings. Depending on the difference in the length of input (L_x) and output (L_y), sequence information is processed differently. Figure 7.3 shows four common Figure 7.3types of RNNs. Text classification falls under the many-to-one category.

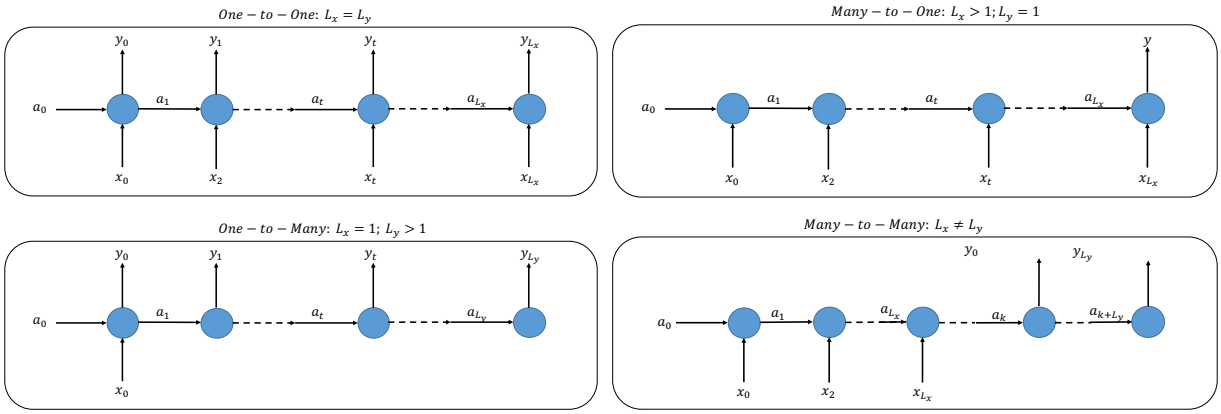

Figure 7.3 Types of Recurrent Neural Network

Gated Recurrent Unit (GRU)

A deep RNN struggles in terms of the long-term dependencies. If the input tweet sequence is long, the information between the outer most layers of a deep RNN gets lost in the process. The cell memory shared between the two units closer in the sequence is more abundant than those further apart. In the deep networks, the gradient of error could vanish before the weights near the earlier hidden layers

could be updated, defined as the vanishing gradient problem.

Long-Short Term Memory (LSTM) networks were developed to capture the long-range dependencies in the sequence data [18]. A simplified version of LSTM called Gated Recurrent Unit (GRU) addresses the issue of vanishing gradient using network *gates* [19]. Gates are the parameters that relate the information flow between the units. The gates in RNNs are based on the *sigmoid* function ($\sigma(x)$):

$$Update\ Gate \rightarrow\ \Gamma_u = \sigma\left(W_{ua}a^{t-1} + W_{ux}x_i^t + b_u\right)$$

$$Relevance\ Gate \rightarrow\ \Gamma_r = \sigma\left(W_{ra}a^{t-1} + W_{rx}x_i^t + b_r\right)$$

The *update gate* controls how much of the past information should be updated based on the current state of the unit. The *relevance gate* decides if previous memory information has any relevance to the current or the future layers of the network. In a GRU, the cell memory and the activation unit's output is the same.

Figure 7.4 shows a GRU with steps highlighted below:

1. At any step *t*, a GRU receives two inputs:
    a. Value of text sequence at the current time step t, x_i^t, and
    b. Cell memory from the previous activation unit, $c^{t-1}(= a^{t-1})$
2. The relevance gate first processes the information to computes Γr, which updates (elementwise) the activation output of the prior unit ($\Gamma_r * a^{t-1}$).
3. An update (suggested) to cell memory is computed, $c_{new}^t = \tanh\left(W_{ca}[\Gamma_r * a^{t-1}] + W_{cx}x_i^t + b_c\right)$.
4. Update gate (Γ_u) is computed separately. It combines the suggested cell memory (c_{new}^t) and cell memory at the previous step (c^{t-1}), to calculate the cell memory at time step *t*, $c^t = \Gamma_u * c_{new}^t + (1 - \Gamma_u) * c^{t-1}$.

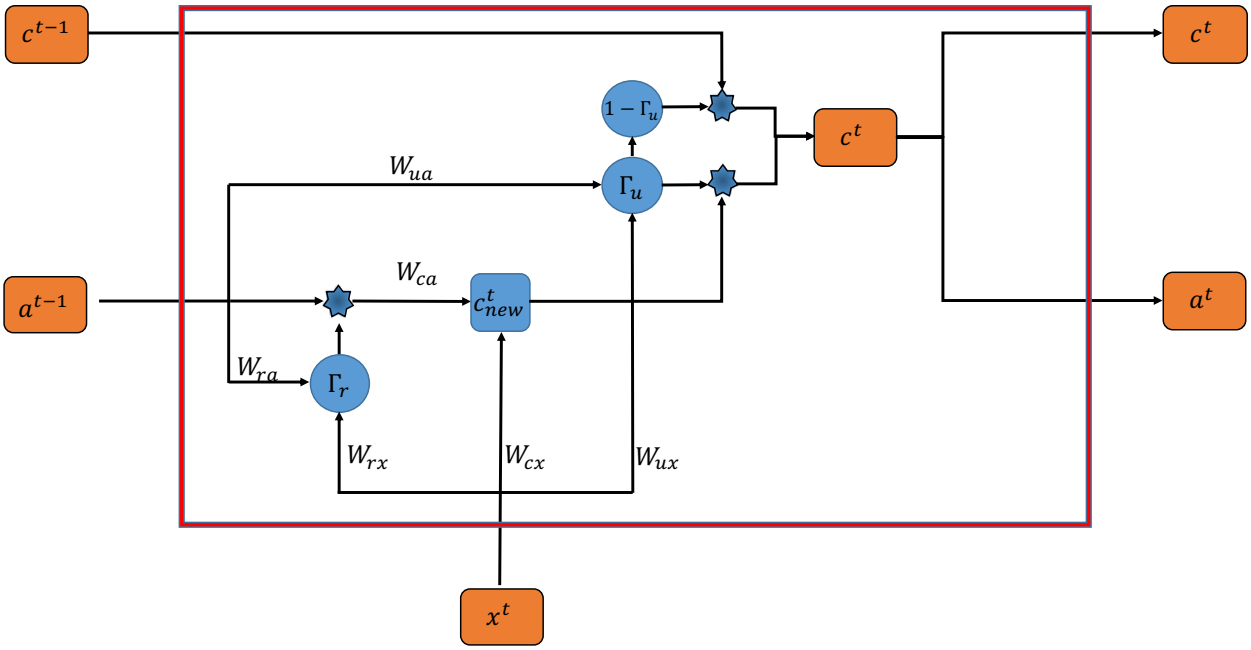

Figure 7.4 Gated Recurrent Unit

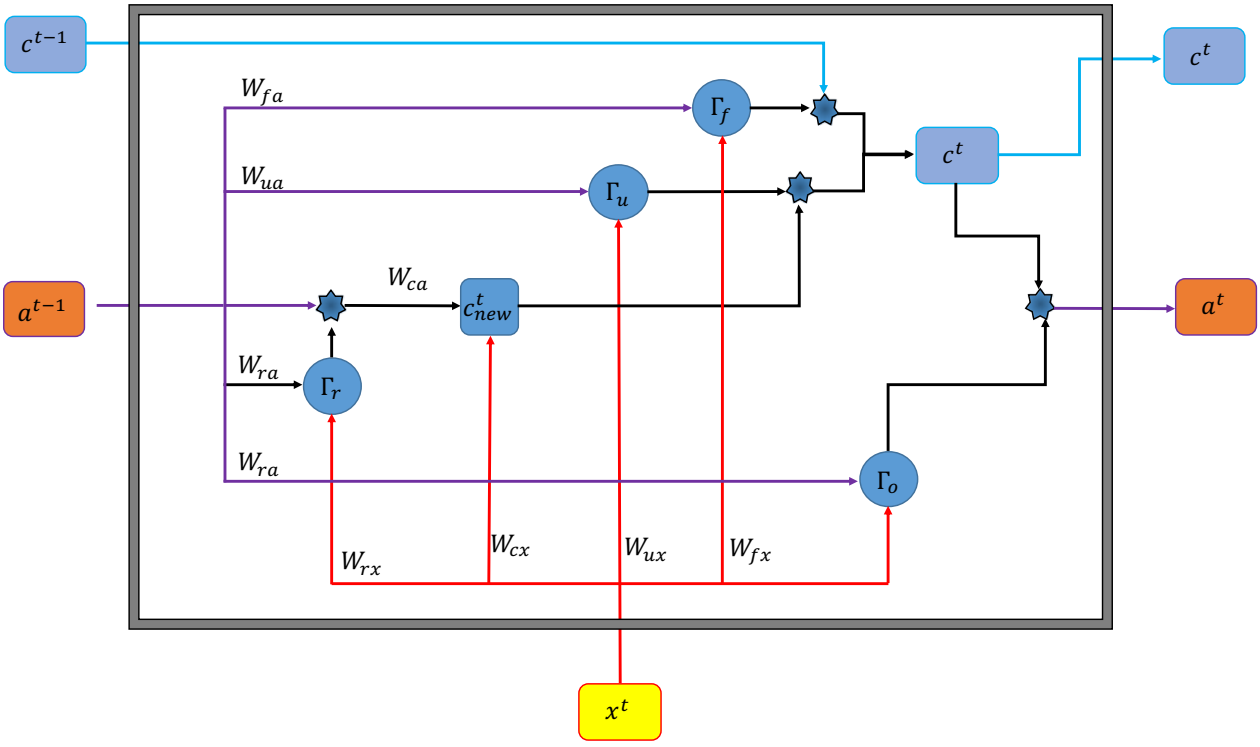

Figure 7.5 Long-Short Term Memory Unit

Long-Short Term Memory Unit (LSTM)

GRUs are simplified versions of LSTMs. LSTMs have two additional gates:

$$Forget\ Gate \rightarrow\ \Gamma_f = \sigma\left(W_{fa}a^{t-1} + W_{fx}x_i^t + b_f\right)$$

$$Output\ Gate \rightarrow\ \Gamma_o = \sigma\left(W_{oa}a^{t-1} + W_{ox}x_i^t + b_o\right)$$

The *forget gate* decides whether to delete the current cell memory. The *output gate* filters the amount of information revealed from the current memory cell to the next one. Unlike GRUs, c^t and a^t are not equal in LSTMs. Figure 7.5 shows an LSTM unit with its steps discussed below:

1. The LSTM unit receives three different inputs:
    a. Value of text sequence at the current time step t, x_i^t.
    b. Cell memory computed at the prior step, c^{t-1}.
    c. Activation unit output at the previous time step, a^{t-1}.
2. The information passes through the relevance gate that computes Γ_r, which updates (elementwise) the activation unit output from the previous step ($\Gamma_r * a^{t-1}$).
3. An update (suggested) to cell memory is computed as, $c_{new}^t = \tanh\left(W_{ca}[\Gamma_r * a^{t-1}] + W_{cx}x_i^t + b_c\right)$.
4. Update gate (Γ_u) and forget gate (Γ_f) are computed to combine the suggested cell memory (c_{new}^t) and cell memory from the previous step (c^{t-1}), and to calculate the cell memory at step *t*, $c^t = \Gamma_u * c_{new}^t + \Gamma_f * c^{t-1}$
5. The LSTM unit has two separate outputs: the cell memory (c^t) and the filtered version called the activation output, $a^t = \Gamma_o * c^t$.

LSTM for Tweet Classification

The first two layers of the neural network were LSTM unit with 300 and 100 neurons. The output from the second LSTM layer was flattened, which connected to a dense layer with 10 ReLU activation units. The final output layer had three units and the *softmax* activation function. Grid search with two hyperparameters (number of neurons in the first LSTM layer and the network and recurrent dropout rates) suggested the LSTM model shown in Figure 7.6. Spatial dropout was 0.4. Code Section 7.1 shows the grid search results with 5-fold, stratified cross-validation. The accuracy score had a low standard deviation with the final validation score of 0.85. Code Section 7.2 shows the final model summary, along with the validation test results and the value of performance metrics. The macro F1-Score of the LSTM model on the test set was 0.87 with AUC about 0.97. LSTM model was slightly better than the baseline model with performance metrics about 2% higher than the gradient boosting model. The overall performance of LSTM, CNN was almost comparable as the number of true-positives is shown in the confusion matrices are not significantly different for the two classifiers, Figure 6.7 and Figure 7.7.

Code Section 7.1 Gridsearch on overfitting hyperparameters

```
param_grid = dict(LSTM_out= [200,300],
                  dropout_rate=[0.2, 0.4, 0.6])
```

```
Best: 0.853920 using {'LSTM_out': 300, 'dropout_rate': 0.4}
0.848054 (0.007193) with: {'LSTM_out': 200, 'dropout_rate': 0.2}
0.849408 (0.012049) with: {'LSTM_out': 200, 'dropout_rate': 0.4}
0.852792 (0.006721) with: {'LSTM_out': 200, 'dropout_rate': 0.6}
0.853243 (0.009257) with: {'LSTM_out': 300, 'dropout_rate': 0.2}
0.853920 (0.009226) with: {'LSTM_out': 300, 'dropout_rate': 0.4}
0.851213 (0.005963) with: {'LSTM_out': 300, 'dropout_rate': 0.6}
```

Code Section 7.2 LSTM Model summary

Model Summary

```
Layer (type)                 Output Shape              Param #
=================================================================
=================================================================

embedding_11 (Embedding)     (None, 34, 300)           5580600
spatial_dropout1d_13 (Spatia (None, 34, 300)           0
lstm_21 (LSTM)               (None, 34, 300)           721200
lstm_22 (LSTM)               (None, 34, 100)           160400
flatten_7 (Flatten)          (None, 3400)              0
dense_13 (Dense)             (None, 10)                34010
dense_14 (Dense)             (None, 3)                 33
=================================================================
Total params: 6,496,243
Trainable params: 6,496,243
Non-trainable params: 0
```

Code Section 7.3 LSTM Model fit

Model Fit

```
Train on 8865 samples, validate on 4367 samples
Epoch 1/20
8865/8865 [==============================] - 34s 4ms/step - loss: 0.8710 - acc: 0.5912 - val
_loss: 0.5523 - val_acc: 0.7914
Epoch 2/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.6284 - acc: 0.7483 - val
_loss: 0.4537 - val_acc: 0.8200
Epoch 3/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.5204 - acc: 0.7938 - val
_loss: 0.4174 - val_acc: 0.8411
Epoch 4/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.4549 - acc: 0.8184 - val
_loss: 0.3814 - val_acc: 0.8512
Epoch 5/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.4044 - acc: 0.8386 - val
_loss: 0.3683 - val_acc: 0.8589
Epoch 6/20
8865/8865 [==============================] - 30s 3ms/step - loss: 0.3860 - acc: 0.8510 - val
_loss: 0.3510 - val_acc: 0.8633
Epoch 7/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.3580 - acc: 0.8597 - val
_loss: 0.3479 - val_acc: 0.8626
Epoch 8/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.3208 - acc: 0.8796 - val
_loss: 0.3425 - val_acc: 0.8720
Epoch 9/20
8865/8865 [==============================] - 30s 3ms/step - loss: 0.3018 - acc: 0.8820 - val
_loss: 0.3336 - val_acc: 0.8727
Epoch 10/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.2754 - acc: 0.8958 - val
_loss: 0.3325 - val_acc: 0.8738
Epoch 11/20
8865/8865 [==============================] - 29s 3ms/step - loss: 0.2558 - acc: 0.8985 - val
_loss: 0.3305 - val_acc: 0.8747
Epoch 12/20
8865/8865 [==============================] - 30s 3ms/step - loss: 0.2395 - acc: 0.9090 - val
_loss: 0.3601 - val_acc: 0.8635
Epoch 13/20
8865/8865 [==============================] - 31s 3ms/step - loss: 0.2216 - acc: 0.9162 - val
_loss: 0.3633 - val_acc: 0.8715
```

Classification Report

	f1-score	precision	recall	roc_auc	support
Hate	0.826606	0.834259	0.819091	0.943593	2200.0
Offensive	0.897735	0.893036	0.902484	0.977857	2174.0
Neutral	0.884553	0.881074	0.888060	0.972236	2144.0
micro avg	0.869592	0.869592	0.869592	0.966827	6518.0
macro avg	**0.869631**	0.869456	0.869878	0.964562	6518.0
weighted avg	0.869391	0.869262	0.869592	0.964443	6518.0
samples avg	0.869592	0.869592	0.869592	0.925207	6518.0

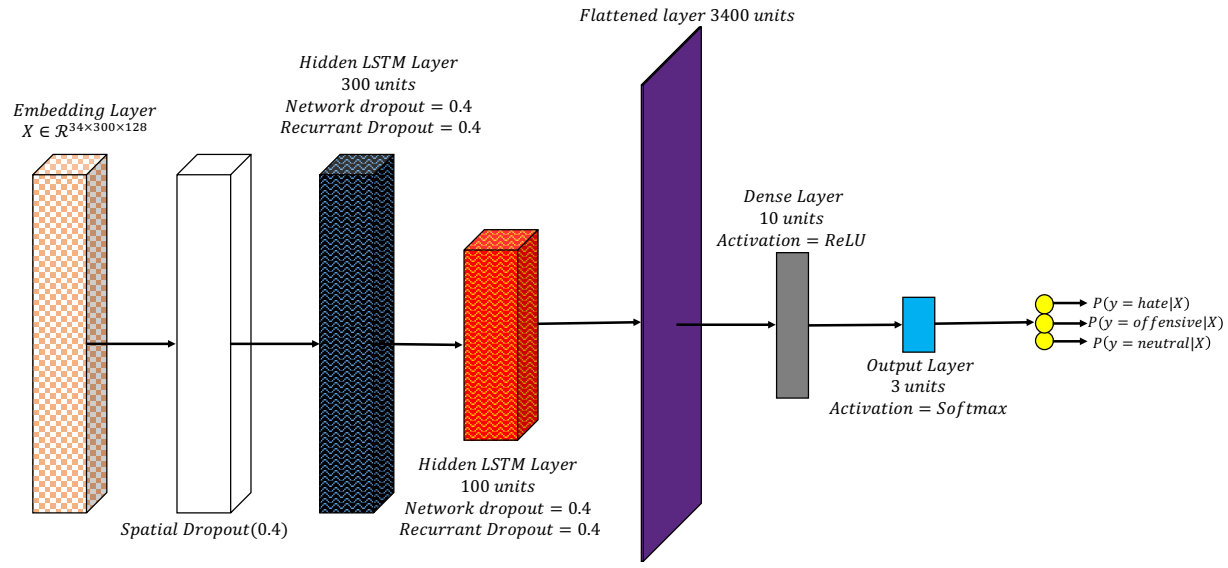

Figure 7.6 Final LSTM model architecture for tweet classification

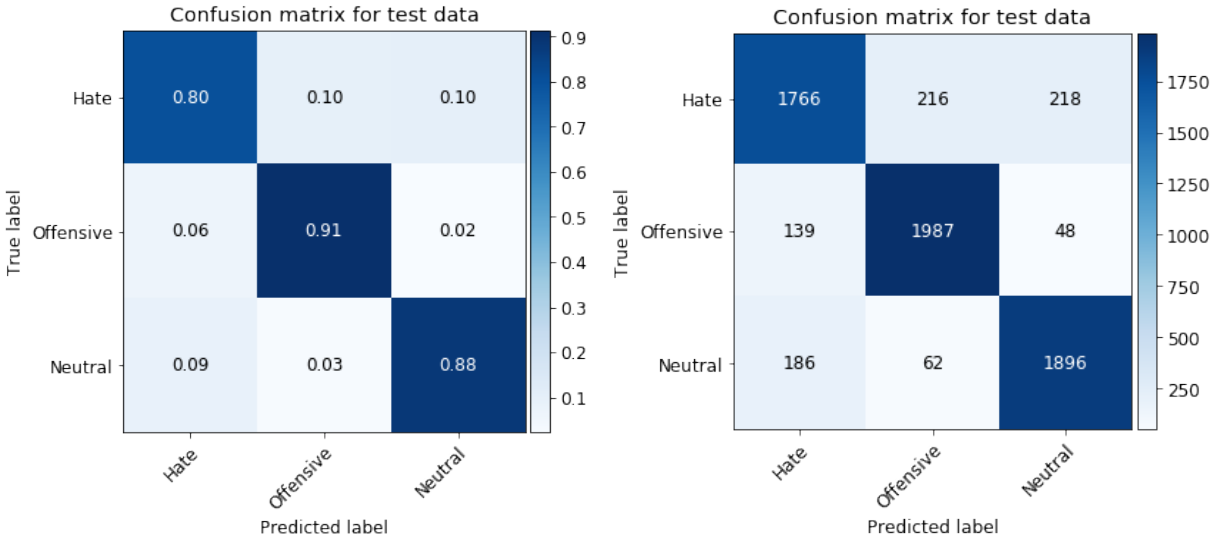

Figure 7.7 Confusion matrices (normalized and non-normalized)

1D Covnet + LSTM

1D covnets and LSTM classifiers were combined to develop a hybrid model. First hidden layer was 1D covnets with no dropouts, and its output served as a modified embedding to the LSTM model. Figure 7.8. Code Section 7.4 shows the model summary and performance on the validation and test sets. The hybrid model had macro F1- score of about 0.85. Figure 7.9 shows the confusion matrices of the hybrid model. In terms of false-negatives, the hybrid model had better performance than the MLP classifier.

Code Section 7.4 Hybrid model fit

Model Summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 34, 300)           5580600
conv1d_2 (Conv1D)            (None, 31, 256)           307456
max_pooling1d_2 (MaxPooling1 (None, 15, 256)           0
spatial_dropout1d_2 (Spatial (None, 15, 256)           0
lstm_3 (LSTM)                (None, 15, 256)           525312
flatten_2 (Flatten)          (None, 3840)              0
dense_1 (Dense)              (None, 10)                38410
dense_2 (Dense)              (None, 3)                 33
=================================================================
Total params: 6,451,811
Trainable params: 6,451,811
Non-trainable params: 0
```

Model Fit

```
Epoch 1/20
8865/8865 [==============================] - 20s 2ms/step - loss: 0.7496 - acc: 0.6556 - val
_loss: 0.4811 - val_acc: 0.8200
Epoch 2/20
8865/8865 [==============================] - 18s 2ms/step - loss: 0.4351 - acc: 0.8369 - val
_loss: 0.3994 - val_acc: 0.8498
Epoch 3/20
8865/8865 [==============================] - 18s 2ms/step - loss: 0.3213 - acc: 0.8839 - val
_loss: 0.3697 - val_acc: 0.8656
Epoch 4/20
8865/8865 [==============================] - 17s 2ms/step - loss: 0.2378 - acc: 0.9155 - val
_loss: 0.4186 - val_acc: 0.8505
Epoch 5/20
8865/8865 [==============================] - 18s 2ms/step - loss: 0.1861 - acc: 0.9351 - val
_loss: 0.4298 - val_acc: 0.8663
```

Classification Report

	f1-score	precision	recall	roc_auc	support
Hate	0.790911	0.844605	0.743636	0.923419	2200.0
Offensive	0.889713	0.896359	0.883165	0.970048	2174.0
Neutral	0.862754	0.810578	0.922108	0.967681	2144.0
micro avg	0.848880	0.848880	0.848880	0.954759	6518.0
macro avg	**0.847793**	0.850514	0.849636	0.953716	6518.0
weighted avg	0.847497	0.850674	0.848880	0.953531	6518.0
samples avg	0.848880	0.848880	0.848880	0.913931	6518.0

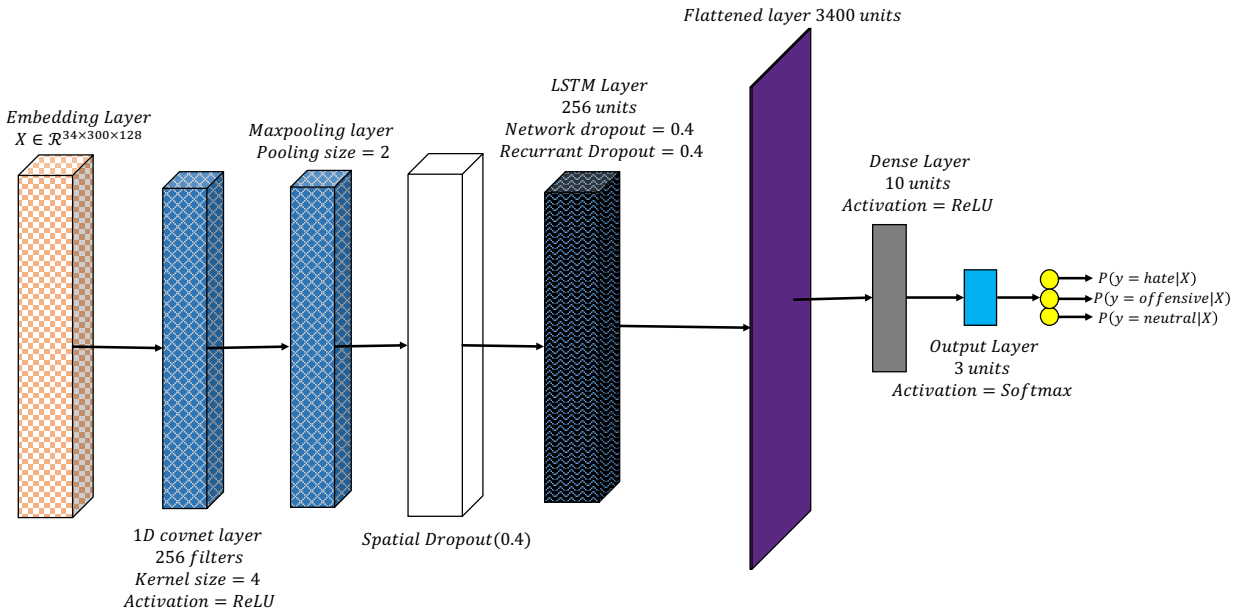

Figure 7.8 Hybrid model architecture for tweet classification

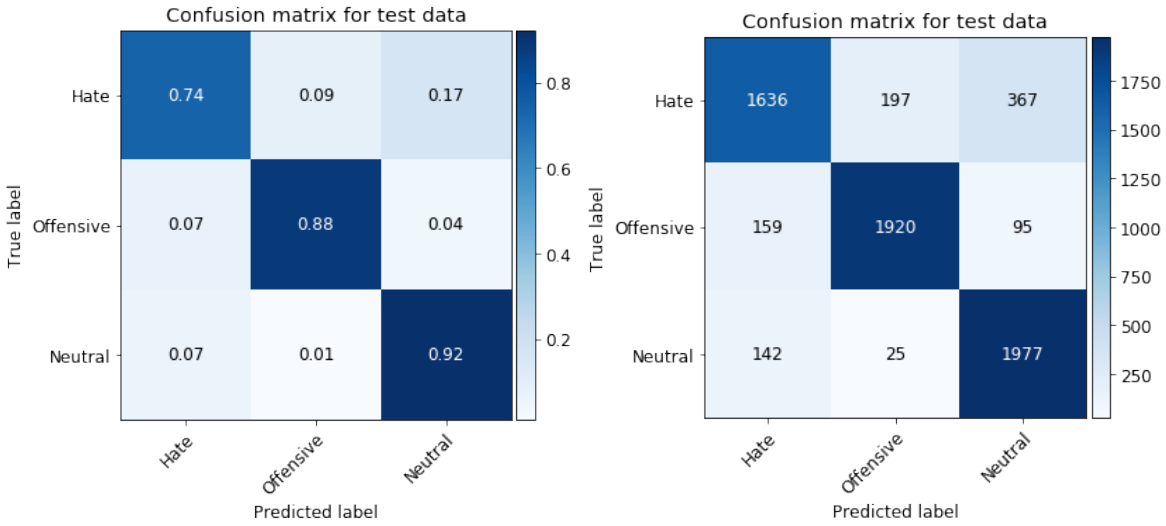

Figure 7.9 Confusion matrices (normalized and non-normalized)

8 Conclusion

This report presented an experimental study to compares the performance of well-known machine learning algorithms on text classification (tweets) into hate, offensive, or neutral categories. The input data were obtained from three different sources, and a balanced dataset (equal number of labeled tweets per class) was selected. The dataset was split into train-validation-test set. Performance of random forest and gradient boosting algorithm on different input features, document vectors, character, and word N-grams was compared. Based on the macro F1-score, gradient boosting model had better performance and selected as the baseline model. Macro F1-score was chosen among the set of different performance metrics to compare the performance of classifiers.

Multilayer neural network was trained on the twitter dataset and had relatively good performance only about 2% less than the macro F1-score of the baseline model. 1D covnets performed better than MLPs and at the level of the baseline model. LSTM neural networks outperformed all other classifiers by a slight margin with an accuracy score of 87%. A hybrid model combining the 1D *covnets* and LSTM model also performed in the average range of other neural network classifiers. Table 8.1 compares the performance metrics for the four classifiers. In terms of computational power, 1D *covnets* performed well and did not require long computation time. Figure 8.1 shows the accuracy score, training, and test time for each classifier. In terms of accuracy score, all classifiers had almost similar performance. CNN and gradient boosting methods were similar in terms of performance but had three times more speed. Same was true when comparing LSTM with the hybrid model. When considering both the performance metrics and the speed of execution, the CNN's had an advantage over other classifiers.

Table 8.1 Performance metrics for different classifiers

Performance Measures	**Random Forest**	**Gradient Boosting**	**Multi-layer perceptron**	**1D Convolution Neural Nets**	**Hybrid Model**	**Long-Short Term Memory Networks**
Macro Precision	0.83	0.85	0.82	0.85	0.85	0.87
Macro Recall	0.83	0.84	0.82	0.84	0.85	0.87
Macro F1-Score	0.82	0.84	0.82	0.84	0.85	0.87
Macro AUC	0.94	0.95	0.94	0.95	0.95	0.97

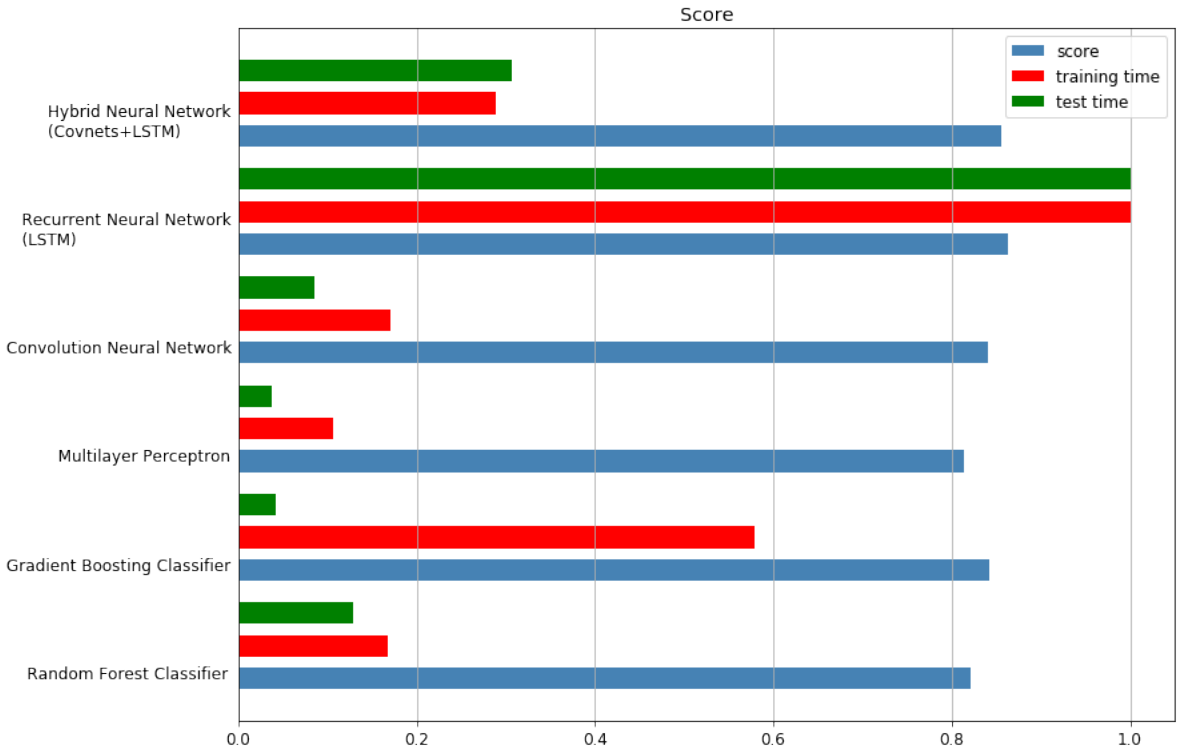

Figure 8.1 Accuracy score, training time and test time comparison

Future Work

There are few topics that can be explored to build upon the work. Rather than word inputs, the classification of neural networks of documents vectors can be included. Hybrid models combining other neural networks for tweet classification might lead to better results. Last but not least, if more computational power is available, it might be beneficial to try mush deeper neural networks for text classification.

References

[1] R. Caruana, N. Karampatziakis and A. Yessenalina, "An Empirical Evaluation of Supervised Learning in High Dimensions," in *International Conference on Machine Learning*, Helsiniki, 96-103.

[2] C. O. D. Archives, "Hate Speech Identification," 2016. [Online]. Available: https://data.world/crowdflower/hate-speech-identification.

[3] T. Davidson, D. Warmsley, M. Macy and I. Weber, "Automated Hate Speech Detection and the Problem of Offensive Language," in *Proceedings of the 11th International Conference on Web and Social Media (ICWSM)*, Montreal, 2017.

[4] R. Aggarwal, "Twitter hate speech," 2018. [Online]. Available: https://www.kaggle.com/vkrahul/twitter-hate-speech/activity#train_E6oV3lV.csv.

[5] Z. Waseem and D. Hovy, "Hateful Symbols or Hateful People? Predictive Features for Hate Speech Detection on Twitter," *Proceedings of the NAACL Student Research Workshop,* pp. 88-93, June 2016.

[6] Z. Waseem, "Hate Speech Twitter annotations," 27 December 2016. [Online]. Available: https://github.com/ZeerakW/hatespeech.

[7] T. Whitlock, "Emoji Unicode Tables," 2019.

[8] Z. Harris, "Distributional Structure," *WORD,* vol. 10, no. 23, pp. 146-162, 1954.

[9] S. Deerwester, S. Dumais, G. Furnas, T. Landauer and R. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science,* vol. 41, pp. 391-407, 1990.

[10] G. Salton, A. Wong and C.-S. Yang, "A vector space model for automatic indexing," *Communications of ACM,* vol. 18, no. 11, pp. 613-620, 1975.

[11] T. Mikolov, I. Sutskever, K. Chen, G. Corrado and J. Dean, "Distributed Representation of Words and Phrases and their Compositionality," 2013.

[12] J. Pennington, R. Socher and C. D. Manning, "GloVe: Global Vectors for Word Representation," in *Conference on Empirical Methods in Natural Language Processing*, Doha, 2014.

[13] S. Arora, Y. Li, Y. Liang, T. Ma and A. Risteski, "A Latent Variable Model Approach to PMI-based Word Embeddings," *Transactions of the Association for Computational Linguistics,* pp. 385-399, 2016.

[14] S. Malmasi and M. Zampieri, "Detecting Htae Speech in Social Media," in *Proceedings of Recent Advances in Natural Language Processing*, Varna, Bulgaria, 2017.

[15] K. &. Z. A. Simonyan, "Very Deep Convolutional Networks for Large-Scale Visual Recognition," 2014. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/research/very_deep/.

[16] K. K. S. C. A. &. A. A. Gopalakrishnan, "Deep Convolutional Neural Networks with transfer learning for computer vision-based data-driven pavement distress," *Construction and Building Material,* vol. 157, pp. 322-330, 2017.

[17] "Convolution," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Convolution.

[18] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural Computation,* pp. 1735-1780, 1997.

[19] K. Cho, B. v. Merrienboer, D. Bahdanau and Y. Bengio, "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches," *arXiv.org,* 03 Sept 2014.

[20] E. Bendersky, "The Softmax function and its derivative," 2016. [Online]. Available: https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/.

[21] T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient Estimation of Word Representations in Vector Space," 2013. [Online]. Available: https://arxiv.org/abs/1301.3781. [Accessed 18 07 2019].

[22] Wikipedia, "n-gram," 02 2011. [Online]. Available: https://en.wikipedia.org/wiki/N-gram. [Accessed July 2019].

[23] G. James, D. Witten, T. Hastie and R. Tibshirani, An Introduction to Statistical Learning with Applications in R, Springer Publishing Company, 2014.

[24] T. Pawelski, "hate-speech-detection," June 2019. [Online]. Available: https://github.com/tpawelski/hate-speech-detection/blob/master/initial%20datasets/cleaned_tweets.csv.