Udacity Machine Learning Nanodegree 2019

Capstone Report:

Understanding Emotion in 280 Characters

Daniel Blignaut

September 2019

Definition

Project Overview

Since the advent and explosion of social media as a form of communication, an entire new touch point and data set has been created, one that is bigger in size than others and generates an astonishing amount of content daily. So much so in fact that over 500 million tweets and potential data points are created every day (Cooper, 2019).

Understanding this pool of data and using it to make inferences and commentary on what is happening in the world is a powerful tool. This realm of understanding, known as computational linguistics in the 1990's and natural language processing in todays terms has skyrocketed into a key field in artificial intelligence. The main reason for this has been a series of major breakthroughs but even more importantly is the large increase in text available online for processing. A statistic describing this is the fact that over 99% of research papers on this topic were published after 20014 (Mika V. Mäntylä)

Sentiment analysis, a particular subsection of natural language processing attempts to distill emotion and attitude out of a body of text using various methods. The applications of this are endless. Sentiment analysis is used in fields such as:

- **Customer satisfaction**: tracking customer happiness over the user's lifecycle. As businesses scale and the number of customers increase, it becomes difficult to manually interpret the satisfaction of users (Reptation.com editors, 2018)
- **Customer service prioritization**: By analyzing inbound requests, ticketing systems can make decisions on which tickets to prioritize based on the customers emotional discontent (Reptation.com editors, 2018)
- **Politics**: Using social media information to understand public opinion on specific political groups and recent decisions (Bermingham)s

Problem Statement

The objective of this project is to try and distil a suitable Machine Learning algorithm to classify tweets into positive or negative sentiment categories. A "sentiment" category is in it's simplest form, an opinionated classification of the body of text based on the "emotion" the body of text contains. Our project will focus specifically on "positive" or "negative" emotion categories.

For example,

- I love dogs (positive)
- I hate dogs (negative)
- I don't hate dogs (positive)
- I didn't say I hate dogs, I love them (positive)
- I don't love dogs (negative)

Our goal is to correctly classify tweets such as the above into the correct categories (specified in the brackets). However, it's easy to see the complexity of the classification problem itself

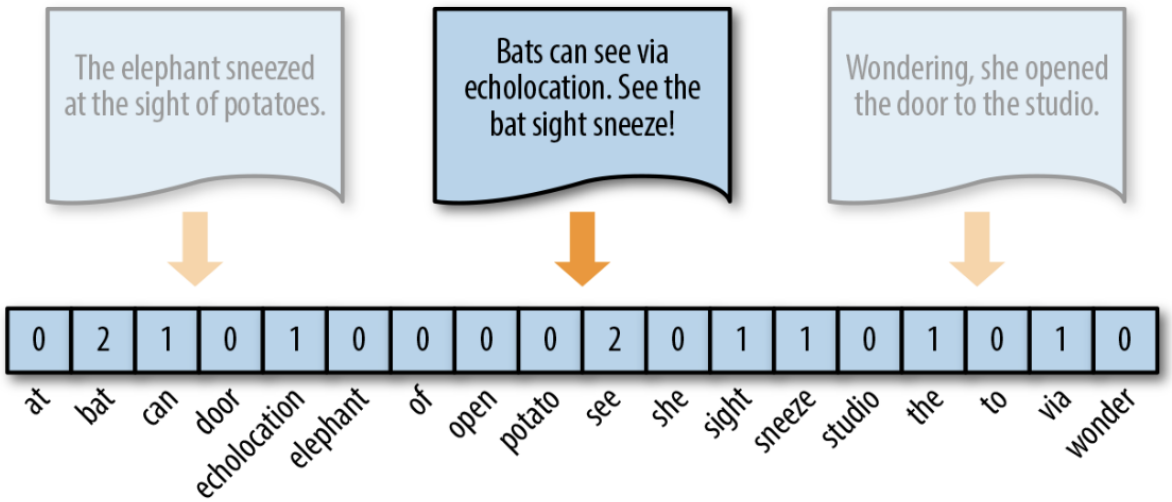
by seeing how many variations of the same sentence can exist across millions of topics. It's clear that due to the **complexity** of the English language itself, the nature of this problem is highly complex and therefore well-suited to machine learning techniques.

In order to solve this problem, we will analyse and compare various supervised and unsupervised machine learning techniques and data pre-processing techniques to narrow down the best solution. The process we will be as follows:

Data analysis: To start, we will analyse our existing dataset to understand what pre-processing will be necessary in the following step and to identify any future issues that may arise.

Common data pre-processing: Because tweets can vary so much in content due to the flexibility of what they can contain (usernames, links, emojis, numbers text) we will need to try and clean this data as much as possible to try and standardise and remove any potential accidental relationships that may exist as not to train our algorithms on false relationships that we know logically do not exist. For example, we may want to expand all contractions as to help standardise format so that our algorithms don't have to interpret / create associations to negative tweets if they contain "don't" and "do not". Instead, they will just have to worry about "do not" and in return we may end up with a stronger correlation between "do not" and our desired label.

Word vectorisation / embedding: Although humans speak in a human language, sentences and their structure are difficult inputs for machine learning algorithms to process. Instead we must go through the process of "vectorising" our tweets so that they can be fed as inputs to our algorithms. In this section, we will perform multiple types of word embedding. In it's simplest form, word embedding is the process of indexing all of our words so that we get a large, sparse array that is of a fixed length (so that all of our inputs are now standardised in size and easily consumable by a machine elarning algorithm) and each value in the matrix corresponds to some sort of relationship between that word and a given tweet. For example, bag of words embedding is simply a count of how many times a given word appears in our tweet (see the image below)



However, more complex models may instead contain different values or more complicated values than simply the amount of times a word is featured, giving our algorithms more "features" or complex input relations so that we can try to better analyse the nuances of these

relationships to determine the tweet's sentiment. For this reason, we will investigate the following word embedding models:

- Unigrams using Count vectorizer
- Bigrams using Count vectorizer
- GloVe (trained on our dataset) using TF-IDF vectorizer
- GloVe (trained on Wikipedia dataset - pretrained) using TF-IDF vectorizer

Supervised Learning Methods & Optimisation: Once our data has been correctly converted to an embedding and vectorised, we will apply a selection of supervised learning models to try and determine the best baseline supervised method for determining sentiment. Once we have found the best baseline model, we will then perform hyper parameter optimisation of the model to try and get the best results possible. This is the process in which we cut our test data into a third set, the validation set and we subsequently use this set to "finetune" the parameters that we pass into our model. The models we will test are:

- Logistic regression
- SVM
- Multinomial Naïve Bayes classifier

Unsupervised Learning Methods: Finally, we will create one unsupervised model known as a recurrent network LTSM model. We will then compare the output of this model against our supervised learning methods to identify the best natural language processing model for analysing general "sentiment" in tweets.

Metrics

In all of our comparison, we will be looking at the following metrics:

- Accuracy
- F1 score
- Training time
- Testing time

To try and determine which model is the best. The reason for analysing training and test times is simply because we are looking at this project in the context of a real-world example and as such, we would like to have an algorithm that can be used in "real-time" or "near real-time" for it's predictive powers and can similarly be trained easily by most users.

We will also be primarily focussing on the F1 score, as although accuracy is a key metric, F1 score will give us a more detailed and realistic measure due to its relative weighting of precision and recall. See the below image and mathematical formulation to highlight this difference.

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall}$$

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP}$$

	Predicted Positives	Predicted Negatives
Positives	True Positives	False Negatives
Negatives	False Positives	True Negatives

Pre-requisites

I specifically used Python 3.7 in Anaconda. One particular library however was at the time of writing, not ported over to Python3. This was the "glove-python" library. In order to port the library, you will also need Cython installed in your releveant Conda environment. I have included the glove-python source files in my repo. You will need to execute the following commands:

cd glove-python/glove

cython glove_cython.pyx
cythonize glove_cython.pyx

cython metrics/accuracy_cython.pyx
cythonize metrics/accuracy_cython.pyx

cython --cplus corpus_cython.pyx
cythonize corpus_cython.pyx

cd ..

python3 setup.py cythonize

pip install -e .

Analysis

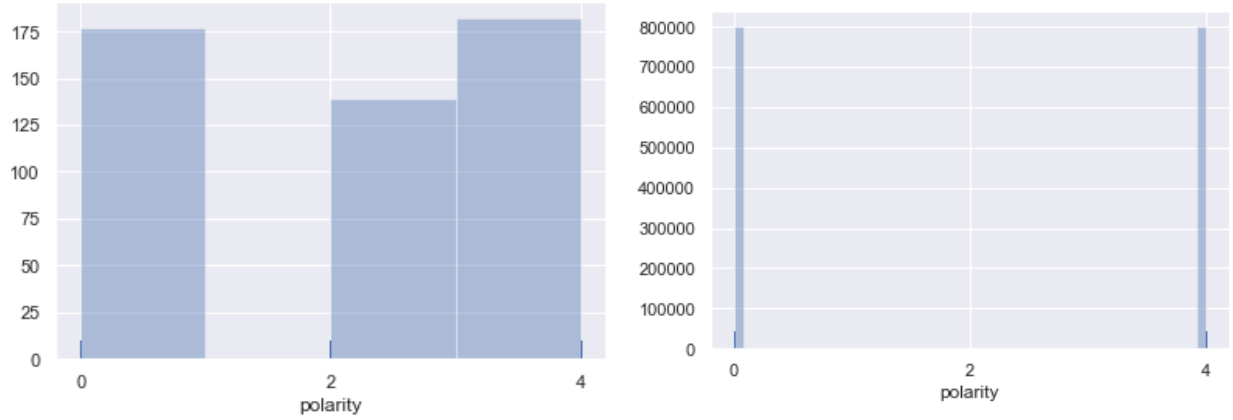
Data exploration & visualisation

For this project, I chose to use the Sentiment140 dataset. This is a dataset that originates from Stanford University and contains over 1.6 million data entries. Each entry has the following columns for the corresponding tweet: (University, n.d.)

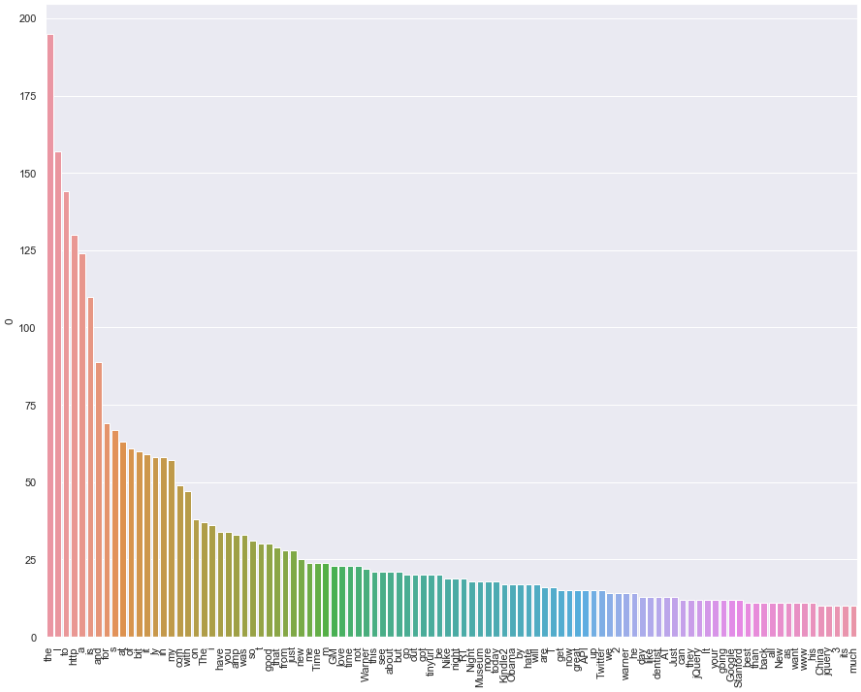
- Sentiment
  - 0 – negative
  - 2 – neutral
  - 4 – positive
- Id
- Date
- The query (lyx). If there is no query then the value is NO_QUERY

- The user that sent the tweets (their handle)
- The text

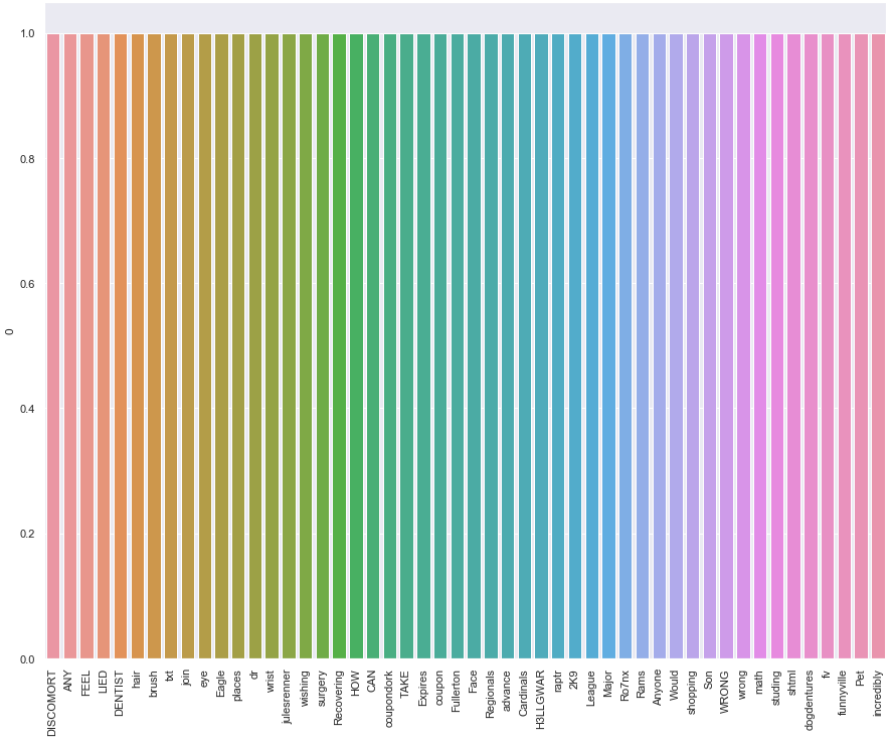
Test data (left) and training data (right) polarity distribution



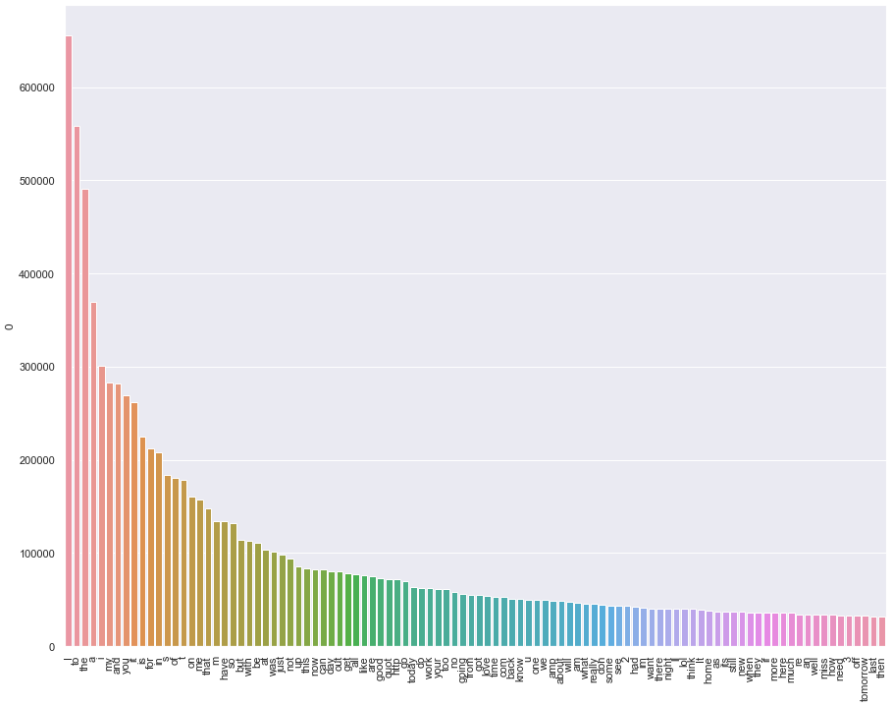
Test data set most used 100 words



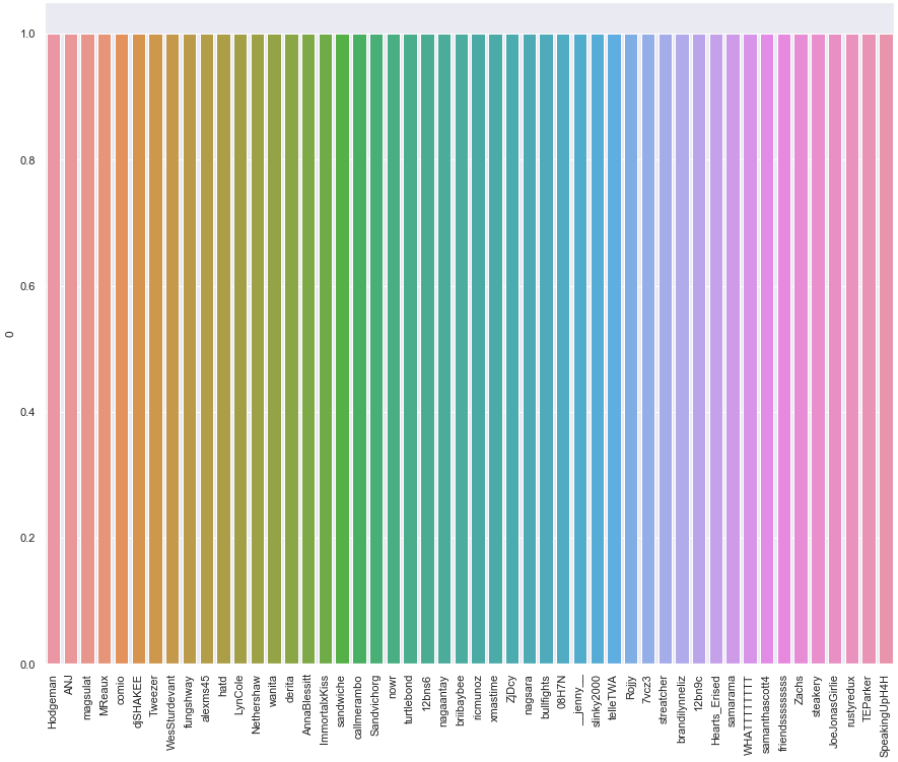
Test data set least used 100 words



Training data set most used 100 words



Training set least used 100 words



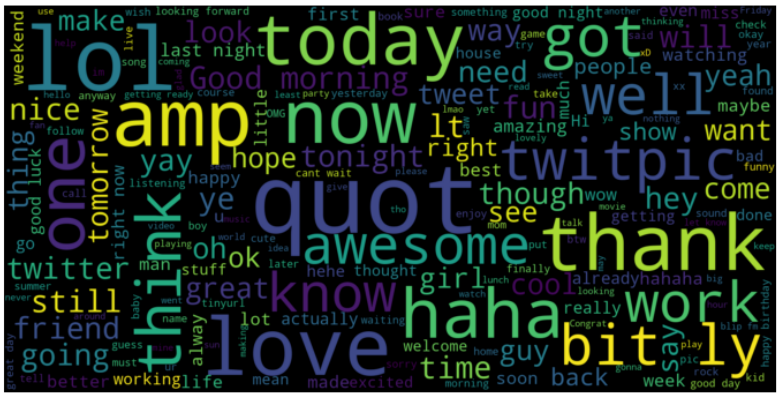
As can be seen from above, the training data set is evenly distributed between positive and negative rated answers. The test dataset however contains a third category, neutral which contains roughly 40 less data points than the positive and negative labels (which have the same distribution).

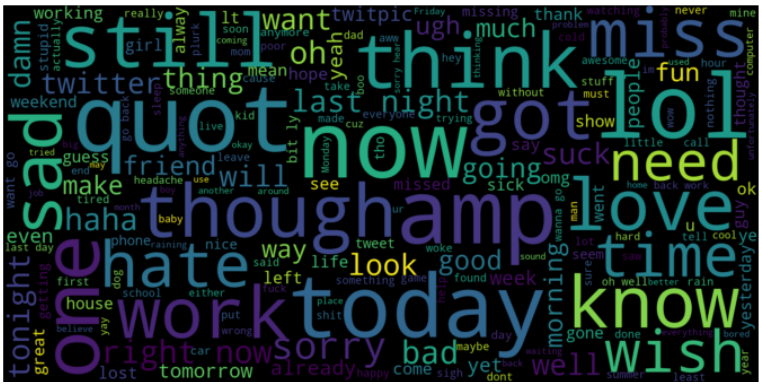
Upon further analysis, the importance of data preprocessing is extremely relevant. It is clear in both the training and test datasets that the most highly distributed words are those which occur most often (by definition) but also imply the least in terms of sentiment (I, the, to, and my). This may confuse any algorithms that we use that, in their simplest forms, are pattern recognition algorithms. In layman's terms, commonly used words such as I, the, to can be though of as "noise", they are not adding any value to the classification of the tweets however they may be ranked fairly importantly because they commonly occur which would be a mistake.

Another issue is that in the least common set of words, user handles and numbers occur frequently. In the case of these words, as described above it is often the case that the most important words are inversely proportional to their frequency – Zipf's law. However, we know that in our use case, usernames, numbers and URL's have no impact on the sentiment of the tweet and we'd therefore like to exclude them from our corpus as they are not important and we wouldn't want any classifier, particularly if we use a TF-IDF (implementation of Zipf's law when doing a word embedding) vectorizer to mistakenly highly rank these words.

And finally, we can also see a quite a few misspelled words and acronyms in the least common words. In the case of misspelled words, we will try correct these and for acronyms, expand them to help ensure that instead of having large variability / variations of patterns, we can have fewer, more common patterns to assist our algorithms in identifying tweets. In our

data pre-processing section we will focus on these 3 topics highlighted above for anything we do to the corpus.





Training data – positive (top) and negative (bottom) word cloud

Finally, I've also attached 2 word clouds of most common positive and negative words from the training set.

The main point that I drew from this is that both word clouds contain generally "sentiment" positive words such as "love", "lol", "haha" as well as neutral words such as "time", "hey", "think". This indicates to me that whatever solution we choose, this problem won't be as simple as identifying common words in positive and negative categories and simply trying to classify them and that this problem is highly complex and dimensional.

Secondly, it also indicates the need for us to standardize spelling and expanding abbreviations that are common such as "lol" to help us reduce the number of patterns and remove any common neutral words to help assist any algorithms in identify positive and negative sentiment.

Algorithms & Techniques

Based on the above visualization and exploration, I've decided to focus on 3 main areas:

Data preprocessing: I intend to focus on the 3 major topics mentioned above, those being:

1. Removing as much noise as possible from the tweets – This means moving any common occurring patterns that logically wouldn't impact the effect of sentiment analysis.
2. Removing any unique or uncommon words that shouldn't have a direct effect on sentiment, including numbers, URLs, usernames, etc. as these (if we try to model Zipf's law) may similarly add noise by being the exact inverse of our first point.
3. Standardizing our data format to reduce the amount of variation between patterns and promote pattern commonality – if a pattern exists, we want to promote it as much as possible through standardization. For example, our algorithms may not associate "LOL" to "laugh out loud" and therefore not treat them the same. We would like to expand acronyms such that it's more noticeable that they are the same.

You can read more specifically about the steps taken within the next section.

Word embedding and vectorization: In the field of NLP, there are multiple methods for word embedding and vectorization, ranging from simple bag of words transformation to more modern techniques such as word2vec and GloVe. As previously discussed, these are all methods of trying to summarize a corpus of text into a more readable format for our algorithms and often include converting the text into a matrix (as seen below) as well as feature reduction. I will be testing 4 different word embedding techniques to try and understand which has the best outcome using 3 different vectorizers. The word embeddings I will use are the following:

1. **Unigrams**: This would convert "The quick brown fox jumped over the lazy dog" into a matrix of ["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy" , "dog"]. This is a very simple embedding and places a lot of emphasis on individual words and classifying each of those words.
2. **Bigrams**: This would convert "The quick brown fox jumped over the lazy dog" into a matrix of ["the quick", "quick brown", "brown fox", "fox jumped", "jumped over", "over the", "the lazy", "lazy dog"]. Here we are creating a matrix of unique two word phrases instead of single sentences and placing emphasis on classifying these phrases. The main idea here is that we can capture more complicated sentence structure patterns, for example: "not love", "not hate" and interpret them correctly instead of "not" and "hate" separately.
3. **GloVe trained on our own corpus**: GloVe is a modern word embedding model built by a group of Stanford NLP researchers (Pennington, Socher, & Manning, 2014). This model converts words into vectors by learning their co-occurrence matrix. A co-occurrence matrix is simply how often that word appears (a probability) next to another word or in the context of a given phrase. The GloVe model then performs dimensionality / feature reductions and factorization such that this matrix is reduced to a reasonable size which still explains most of the variance within our corpus.
4. **GloVe trained on Wikipedia's corpus**: Here we are using the same model as above, but instead of calculating the co-occurrence matrix based on our own training data, we are going to use a pre-trained co-occurrence matrix, calculated based on all of the page in Wikipedia. This is presumably a much more comprehensive and exhaustive dictionary of word vectors and may have more accurate vectors however Wikipedia does not contain much informal or slang language and therefore it may not necessarily be expected to perform as well as our own trained GloVe model.
5. **Word2Vec trained on our own corpus**: Similar to GloVe, Word2vec converts words into a vector matrix representations of normally a few hundred features. Words are given specific feature values such that they are positioned close to other words that share common context in this vector space. Word2Vec is a general grouping of a few methods / variations of calculating this vector space but all of them are shallow, 2 layer neural networks.
6. **Word2Vec trained on Google news's corpus**: Here we are using the same model as above, but instead of calculating the vector space values based on our own training data, we are going to use a pre-trained model, calculated based on over 100 billion words from corpuses extracted from Google News. This is presumably a much more comprehensive and exhaustive dictionary of word vectors and may have more accurate vectors however Google News does not contain much informal or slang

language and therefore it may not necessarily be expected to perform as well as our own trained Word2Vec model.

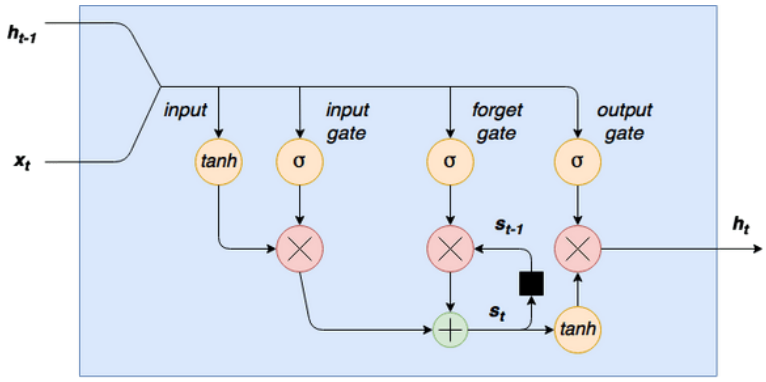
As previously mentioned, I plan on testing 3 different vectorizers:

1. **Count Vectorizer:** Converts the matrix of words to simple matrix of counts such as:

The	Quick	Brown	Fox	Jumped	Over	Lazy	Dog
2	1	1	1	1	1	1	1

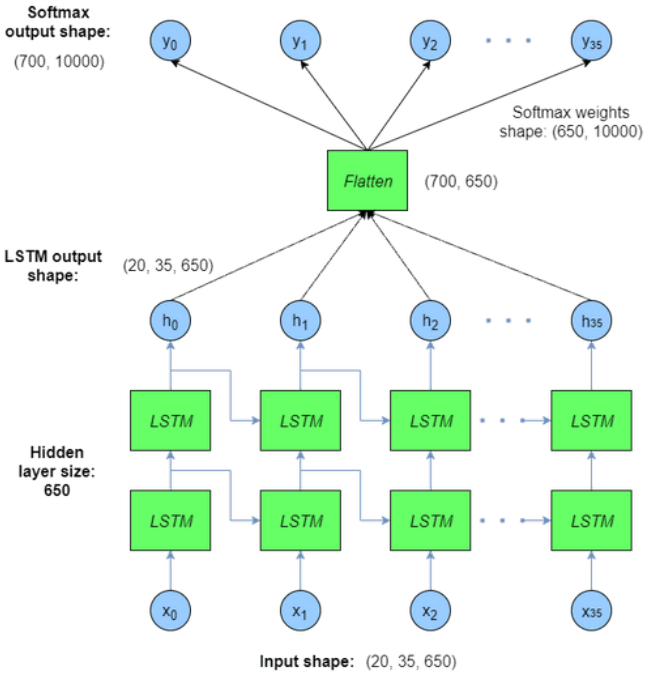
   However, this would be a sparse matrix, whose width would be that of all of the unique words in every single tweet so that each data input is the same length. I will use this for all of our n-gram embeddings where it is most commonly used.
2. **TF-IDF vectorizer:** Instead of converting our vectors into simple counts, here we first calculate a "weight" for each word. This is based on Zipf's law and we give words that occur less common, higher weights. These weights are then multiplied by the corresponding vector values to give us the final word embedding. Instead of retuning a simple single value per cell as the Count vectorizer does, we will most likely be returning an array or list, specifically in the case of our GloVe embeddings. I will use this vectorizer for all of our word embeddings
3. **Mean Embedding Vectorizer:** This will calculate the final input vector by calculating the average of all the input matrices for a given x input. For example, if input 1 has the following x value: [1, 2] our final embedding will be [1.5]. This is a common vectorizer used for modern embedding algorithms such as GloVe where each matrix is a co-occurrence matrix.

Models: I will then focus on the implementation of a few supervised models and one unsupervised models. I will initially train a variety of well known NLP supervised learning models, including Logistic Regression, SVM and Multinomial Naïve Bayes model. These models have historically yielded good performance, typically between the 75%-83% accuracy mark on sentiment analysis (Alec Go) (Yao, 2018). For speed and performance issues, where possible I will make use of as much multi-processing as possible and default to linear kernels. This is because we will be training a minimum of 3 different classifiers on 4 variations of our data set across varying sample sizes (1%, 10%, 50%, 100%) to understand which is best. This means we have over 3x4x4= 48 model variations to visualize, graph and process which may negatively impact our training – specifically on SVM which is not possible to multi-process and very slow as our number of features grows (our Wikipedia GloVe embedder results in 100 features p/ word). Once finding the best baseline model across various metrics, I will then perform GridSearch optimization to find the best parameters to fine tune our model. Finally, I will build an unsupervised LSTM model and compare the performance of that with our supervised learner.



An LSTM model is unsupervised model based on a Recurrent Neural Network.

The main reason why RNN's were introduced were to introduce the conept of time-series analysis, providing a method for neural networks to not only learn pattern recognitioins based on the current input but also based on previous outputs in the series.



However, a major flaw of them is their method of backpropagation. Backpropagation is a method of optimising the neuron weights based on the models training data output and what the expected output is. In the case of RNNs, because the output calculated by it may take as an input the outputs from it's previous calculations, it ends up with a very complicated and multiplied derivative, leading to a vanishing gradient problem where the derivative either explodes or tends to zero.

LSTMs solve this as shown in the top image which depicts a single LSTM "unit" – the bottom image depicts how and overall LSTM netword is made by grouping these units and treating outputs. LSTM's work by making use of input gates which combine the current input (xt) and the output of the previous LSTM unit by using a tanH activation function to squash our inputs into one and then performing elemnt-wise multiplication by typically some sigmoid activation functions.

This input then goes through an additive transformation whereby a new value, St-1 is added to it. This is the internal state of our LSTM cell and transforms our previous input value. Once this has happened, our input goes through the forget gate. The forget gate is typically another layer of sigmoid functions that when our input flows through it, the forget gate decides what values to drop by multiplying various parts of our input matrix by a value between 0 and 1. This is importantly how our LSTM cell "decays" it's memory.

Finally, our value flows through the output gate which is simply another multiplication by an activation function to determine what the output of our LSTM cell should be. The first image displays this entire process within one LSTM cell where as the second very clearly signifies the flexibility of an LSTM cell and how it can be chained together to process large input matrices and time-series data and how each input flows from cell to cell. (Admin, 2019) (Nguyen, 2018)

Benchmark

As previously mentioned, most of my project is centered around creating a suitable benchmark to test against the LSTM network. However, after conducting research, I would be happy with accuracy and f1 scores between 75%-83% with a less than 5 second prediction time.

Methodology

Data Preprocessing

In NLP, data preprocessing is extremely important as sentences and written language can tend to be one of the dirtiest or unstructured forms of data. This can arise due to a number of issues, including human error and the highly complex nature of a given language and the many ways we can convey ideas in sentences, leading to many variations of the same idea.

Coupled together with this, because we are specifically running sentiment analysis, we are looking to capture a particular set of features from our text and whilst it may be hard to exactly identify those features (thus the need for complex machine learning algorithms to solve this problem), we do know what features we don't need such as numbers or URLs that don't provide any emotion to our text.

Below is a list of text cleaning methods I've implemented based on my 3 data cleaning goals mentioned above. They are also based on the following research: (Seo, 2018) (Alec Go)

Lowercase transformation

To the human eye, we may perceive words such as "You" and "you" as being the exact same. The only significant difference, often from a readability perspective, is that we capitalize the first letter of a sentence. However, in data science, our algorithms will not see these spellings as synonymous (at least not initially) and although our data set may be big, this could lead to a lot of data that we would normally attribute as being the same but our algorithm would attribute them different due to the different capitalization, making our data set seem smaller. To counteract this, we lowercase all of the words in our dataset. This is sometimes considered dimensionality reduction. (Oppermann, 2019)

@stellargirl I loooooooovvvvvveee my Kindle2. Not that the DX is cool, but the 2 is fantastic in its own right.

After:

@stellargirl i loooooooovvvvvveee my kindle2. not that the dx is cool, but the 2 is fantastic in its own right.

Expand contractions

To further standardize our data, we will expand all contractions. Whilst this will increase word count and possibly the sizing of word vectors as it may result in more unique words than there were before, it will help standardize our text as our content may contain a mix of contracted and expanded words, for instance "You'll" or "you will".

@richardebaker no. it is too big. i'm quite happy with the kindle2.

After:

@richardebaker no. it is too big. I am quite happy with the kindle2.

Remove links

URL references don’t provide any sentiment related data and can thus be removed.

here's a case study on how to use viral marketing to add over 10,000 people to your list http://snipr.com/i50oz

after:

here's a case study on how to use viral marketing to add over 10,000 people to your list

Remove usernames

Usernames don’t provide any sentiment related data and can thus be removed

@stellargirl i loooooooovvvvvveee my kindle2. not that the dx is cool, but the 2 is fantastic in its own right.

After:

i loooooooovvvvvveee my kindle2. not that the dx is cool, but the 2 is fantastic in its own right.

Remove punctuation

Whilst we will be making use of an LSTM recurrent network model to analyse text, which does make use of “memory”, we are not focusing explicitly on the effects on sentence structure in polarity of sentiment of the text. Whilst sentence structure may be pertinent to text synthesis and text generation, for our use of text analysis, it adds an unnecessarily large amount of extra data points.

Another key reason for removing punctuation is that it doesn’t convey emotion or meaning.

An interesting paper investigates the effects of punctuation on sentiment analysis. (Gizem Gezici)

i loooooooovvvvvveee my kindle2. not that the dx is cool, but the 2 is fantastic in its own right.

After:

i loooooooovvvvvveee my kindle2 not that the dx is cool but the 2 is fantastic in its own right

Expand abbreviations & contractions

Code for this section has been taken from here:
https://github.com/rishabhverma17/sms_slang_translator

As per expanding contractions, we also try to expand abbreviations. However instead of doing this for purely word normalization means, we do this because abbreviations may contain hidden words such as “LOL” which translates to “laugh out loud” which contain emotion. We therefore would like to normalize and find patterns in our datasets easier to find for our algorithm by it just having to understand “laugh” for instance and not “LOL” and “laugh”

lebron is a hometown hero to me lol i love the lakers but let us go cavs lol

after:

lebron is a hometown hero to me lol i love the lakers but let us go cavs laugh out loud

Remove digits

As per the above definition, digits are “words” that convey no additional meaning to the context of the sentence / data point. We therefore choose to remove them to remove the unnecessary data points in our set.

heres a case study on how to use viral marketing to add over 10000 people to your list

after:

heres a case study on how to use viral marketing to add over people to your list

HTML decode characters

Because this source is pulled from the internet and twitter in general, often characters are HTML encoded in order to make them passable in your web browser or URL and not to be confused with a few key characters. For instance, “&” which is often used to separate query parameters in a URL becomes “&”. We decode these characters to standardize the data we are working with and to make it easier to work with.

house correspondents dinner was last night whoopi, barbara & sherri went, obama got a standing ovation

after:

house correspondents dinner was last night whoopi, barbara & sherri went, obama got a standing ovation

Remove hashtags

I have chosen not to remove hashtags as I believe that they may contain sentiment based information and other contextual data that our algorithms would be able to use to help classify each tweet.

Remove extra letters
Due to the nature of twitter and it's informality, often user's spell words intentionally with extra vowels for effect. For example,

> *@stellargirl I loooooooovvvvvveee my Kindle2. Not that the DX is cool, but the 2 is fantastic in its own right.*

However, when this is interpreted by our machine learning algorithm, it should be interpreted as "love" instead of the mis-spelling above. There will be many more occurences of the word "love" if we standardise all of the spellings meaning that commonalities or patterns across our training data will be better inferred.

Lemmatize words

I've opted to utilise lemmatizing instead of stemming for our dataset. Lemmatising is the process of generating / converting words from their inflected self to their root self. (Jabeen, 2018) For instance, "playing", "player", "played" to "play". Where stemming and lemmatization differ is in the fact that stemming converts to a root word that is not necessarily a real word where as lemmatization converts the inflected word to a real root word.

At this point I am unsure on whether or not language is of importance in my study and thus for the sake of precaution, I've opted to use a lemmatizer.

Remove stop words

Often sentences contain words such as "the", "a", etc. These words often make up the bulk of the corpus. These words, which then skew our data and add a lot of noise, don't imply any meaning or applicable information in the context of sentiment analysis.

However it does provide mixed results. In some cases model performance has decreased due to the removal of stop words, as can be seen in the next section.

Negations
Haven't modified any negations however the article below contains interesting insight into the relevance of managing and pre-processing negations. (Fueyo, 2018)

Data Preprocessing Analysis

For the sake of space and readability of this report, I have not included the visual analysis of our data set post the pre-processing. However – it is clear that our pre-processing has drastically affected and cleaned our data. We now have less neutral words in our word cloud such as "a", "the" that featured in both our positive and negative clouds.

When looking at the least popular / common words in our positive and negative categories, we no longer see any usernames, digits or URL links. These are irrelevant pieces of information when it comes to sentiment analysis and thus we have reduced the dimensionality of our data substantially.

With all of this in mind, I believe we have done well to clean our data set and provide more optimal and structured data for our learners.

Implementation

For any definitions or high level outlines of what we are implementing, please read the previous section for verbose explanations. In this section I will focus explicitly on the Python code implementation of this project.

Due to the many variations of data and models, I have ensured that I make use of the pickle function in python to continuously save my complex data objects. Pickle is a low level object serialiser and deserialiser. I have also made sure that in my training pipelines and model definitions, I have tried to make use of multi-processing as much as possible to speed up the training time of the various models by making use of multiple CPU cores.

Train, Test split
After pre-processing our data, I decided to re-join the original data set so that I could apply any future transformations on one variable instead of 2 variables as to prevent any human errors by forgetting to run a certain variable through a transformation.

This also meant that when it comes to doing hyper parameter optimization using GridSearch, I can also create a validation data set.

I also decided to remove the neutral category (any tweet in our test set with a y-label of 2). This was because I explicitly want to perform binary classification (classification of one set or another) and in order to explicitly try figure out the sentiment of tweets, it therefore made sense to remove this data from our set as not to skew performance or confuse our algorithms when training.

Pipelines
Because I plan on testing a variation of supervised models with many variations of our data set at varying sample sizes, it was important that I created a robust and parallel programmed pipeline to manage the training, visualization and storage of our data, models and results. This is clearly labelled under the "supervised learning method pipelines" section in my jupyter notebook that explicitlty manages the following:

1. Creating a pool of processes to run all of the following steps (besides the final saving and displaying results in parallel)
2. Sets some global variables (this is done to prevent having to parse large data objects to each individual process in memory). The main variables here are our x inputs (remember we've made many variations of our x input) and a unique table key for the current pipelines we are running.
3. We then run our supervised pipeline which:

a. Splits our data into training (90%) and testing (10%) data sets. I chose a low testing data set percentage due to the large size of our data set. A 10% split still equates to over 140 000 inputs.
b. Trains the given model that is passed in as a parameter and also tries to run predictions against both the training and testing data set. This is so that we can assess overfitting later on in our assessment and if models are performing well on our training set but poorly on our testing set because we've over-optimized it to the training data. We also record all of the various metrics we're interested in, including training time, prediction time, f1 training, f1 test, accuracy training, accuracy test. Please refer to the previous metrics section for an explanation of each of these

Word Embedding

Finally, before running our pipelines, I had to calculate the word embeddings (final x inputs) for our pipelines. For a definition of each of the algorithms below, please refer to the previous Algorithms and techniques section.

To make all of our n-gram vectors, I made use of SciKit learns built in vectorizers to manage the transformation. Scikit learn provides both a CountVectoriser and TD-IDF vectorizer implementation that takes a corpus of text and transforms it to the final word embeddings following the process and concepts mentioned earlier for each respective vectorizer.

For the GloVe word transformation, I made use of the glove-python library which is a python port of the original C GloVe source code. Please follow the steps in the beginning of this report to get this library running in python 3 and Anaconda.

For our initial GloVe model, I had to train it on our own corpus of text, being all of our tweets. Once trained, I then looped through every unique word in our corpus and create a dictionary, mapping each GloVe vector to each word.

For the pretrained GloVe model, I simply had to download the pretrained and do the same as above, barring the training. This meant once again, looping through every unique word in our corpus and create a dictionary, mapping each GloVe vector to each word. As can be seen from the Jupyter notebook, this was a 100 feature vector where as our internal GloVe vector had 300 features.

Once I had the unique dictionaries of GloVe vectors, I built a vectorizer for TD-IDF and Mean Embedding around them. This meant that these vectorizers could be initialized with either the pre-trained our dictionary and when they were passed in our text corpus, it would return an object, similar to our corpus's structure but instead of words, it would contain the corresponding GloVe vector that had a TD-IDF or mean embedding transformation applied to it.

At this point, I now the following embeddings to test across all of my models:

- Unigram with count vectorizer
- Unigram with TF-IDF vectorizer
- Bigram with count vectorizer
- Bigram with TF-IDF vectorizer
- Trigram with count vectorizer

- Trigram with TF-IDF vectorizer
- GloVe with Mean Embedding vectorizer trained on our dataset
- GloVe with TF-IDF vectorizer trained on our dataset
- GloVe with Mean Embedding vectorizer trained on Wikipedia dataset
- GloVe with TF-IDF vectorizer trained on Wikipedia dataset

Initially, based on the above data sets and the models I chose to use below, the highest f1 score I could attain on the test set was 78%. Because I was slightly unsatisfied with this, I decided to tweak my data sets further and introduced the following sets in addition to the above:

- Trigram with count vectorizer with stop words included
- Trigram with TF-IDF vectorizer with stop words included
- Word2Vec with Mean Embedding vectorizer trained on our dataset
- Word2Vec with TF-IDF vectorizer trained on our dataset
- Word2Vec with Mean Embedding vectorizer trained on Google news dataset
- Word2Vec with TF-IDF vectorizer trained on Google news dataset

Although 78% is a good baseline score, I wanted to explore the effect of data-preprocessing, embedding and vectorization had on performance versus model selection which is often highlighted as the most important decision.

The outcome, as you can see in the results section and below was that not only were some of my assumptions on data pre-processing wrong (i.e. effect of stop words on predictability) but we managed to increase performance by over 5 percentage points.

As a side note, had I not run short on time, I would have liked to test combining various combinations of the above data sets (i.e. combining word2vec and trigram inputs, etc.) as well as further tested my data pre-processing assumptions. This is because as you can see in the results, when I kept stop words in the data inputs, I scored the highest f1 score of 82% and therefore my assumption that improving non-important stop words would improve performance was wrong. I'd also like to test these assumptions across not only my n-gram embeddings but also on GloVe and Word2Vec embeddings.

Supervised Models

Because I knew the metrics I wanted to measure and that all of the supervised models were SciKit learn models that all contained similar methods, I created a processing pipeline that would take each model, how big I wanted the training set to be and the variation of the training set as a parameter. The output of this would be a dictionary that I could map back to the original array.

The way I structured this meant I could make use of wrapping the for loop in a Pool of multi processors and map them back to my results object, allowing me to perform the training as quickly as possible on all of the given models.

I managed to fairly easily loop through all of the data sets and get the outputs which I then proceeded to analyse and display by making use of line charts, however I struggled with the SVM model.

I believe this was due to memory issues as well as processing time issues due to the O(n) notation of the SVM model and how slow it would therefore be on large data sets with many features. (the training set contained roughly 1.4 million tweets). I therefore had to train the SVM model on much smaller sample sizes as depicted in the notebook, settling on samples sizes of 5000 & 10 000 as the most reasonable sizes.

This meant that I now had the following models, paired with samples sizes to test on all of the above data sets:

- Multinomial Naïve-Bayes trained on 1% of training data
- Multinomial Naïve-Bayes trained on 10% of training data
- Multinomial Naïve-Bayes trained on 50% of training data
- Multinomial Naïve-Bayes trained on 100% of training data
- Logistic Regression (linear kernel) trained on 1% of training data
- Logistic Regression (linear kernel) trained on 10% of training data
- Logistic Regression (linear kernel) trained on 50% of training data
- Logistic Regression (linear kernel) trained on 100% of training data
- Support Vector Machine (linear Kernel) trained on 5000 training data points
- Support Vector Machine (linear Kernel) trained on 10 000 training data points

LTSM Network

As described further below in the results, the best data set for the supervised models was the TF-IDF trigrams with stop words. However, this data is not in a "time-series" format but is rather a frequency count. Therefore I chose to rather use plain GloVe vectors and use those as the input to the LSTM network.

In this sense, the network would therefore take in a tweet where each word has been translated into a vector which is made up of 300 values.

I chose to use a GloVe model that was pretrained on the Wikipedia data set instead of my own, even though the supervised results showed better performance when GloVe was trained on my own corpus. The reason for this decision is that I have chosen to disregard earlier results as I don't believe that GloVe paired with a supervised learning method and then massively feature reduced by using a Mean embedding or TF-IDF embedding is an accurate benchmark to test this assumption. I still believe that the pre-trained model should prevail due to the fact that it has already been trained and optimised on a much bigger corpus.

Once I had my GloVe inputs for the model, I split the data into training and testing datasets.

One issue I knew I would have was the fact that tweets are of varying lengths whereas Keras models need to take inputs of a fixed length. I therefore had to pad each sequence to the maximum length of the longest tweets as not to loose any valuable information. However, padding all of our tweets in memory would lead to memory issues due to a lack of RAM on both the CPU and GPU front. I therefore had to create a generator which is a function that is called multiple time during training to convert / pad smaller "batches" of our training data, fit it and repeat this process. This means we can save memory by converting smaller portions of the training data on the fly as and when needed.

I also wrote 2 checkpoint functions that will be called at the end of every epoch.

1. Metrics – this would output some of the metrics that we are interested in by testing the model at the end of each epoch against our cross validation data set. These metrics included accuracy, f1 score, precision and recall
2. F1Evaluation – this would evaluate the f1 score of our model. If the score was greater than the max score from all the epochs, it will save the models weights. If the score was not better over 3 consecutive epochs, our model will quite the training process and we will assume we've found the best possible model.

Because we are doing binary classification, I wrote a method that I then wrapped around every prediction our model makes. This method would transform the output from our model where any value that is less than 50% will have a sentiment category of 0 (negative) otherwise the output will have a sentiment category of 1 (positive). In this sense, we can then finally compare the output of our model to the sentiment category from our data to calculate our accuracy and f1 scores accurately.

I chose to build a simple LTSM made up of the following layers:

1. Input layer: this is the start of our LTSM network. It is designed to take an input with the following shape (301,300). You can think of the "301" as the maximum length of a tweet and in a sense, this is our time series data. 300 represents the size of each GloVe vector for each word in our tweet
2. LTSM layer: See under Algorithms and techniques for the definition and how LSTM networks work. This layer adds all of the input, output and forget gates needed to implement a layer of LSTM nodes. The number of nodes in this hidden layer is calculated based on the input shape
3. GRU layer: This layer is similar to the LSTM layer. The GRU comprises of the reset gate and the update gate instead of the input, output and forget gate of the LSTM. The reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep around. If we set the reset to all 1's and update gate to all 0's we again arrive at our plain RNN model.
4. Global average pooling layer: This layer reduces each 2D filter in the filter map to one value. This value is the average of that 2D filter. It therefore converts an array of matrices into a vector. (see second image above)
5. Dense layer: this is our final output layer. Because we are doing binary classification, we specify that we simply want one output category.

In our use case, because we are doing a single class classification, binary classification, we therefore use a sigmoid activation function and binary cross entropy as our loss function.

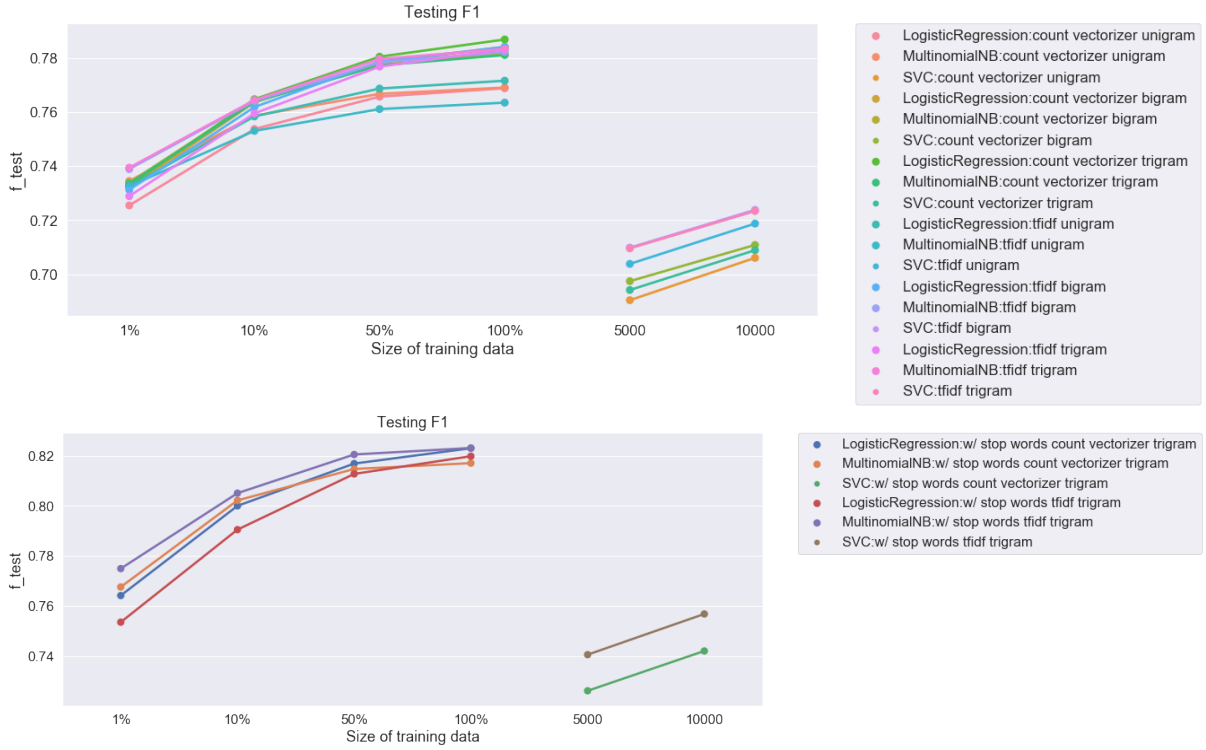
Finally, we are ready for training. Before we train our model, I've introduced K-Fold cross-validation. This allows us to break up our training data into k-folds, in our case 3, and in each fold we have a subset of training and validation data. We can then use each training subset and test each model against the validation subset and finally use the average across all of these models to find our best model. This allows us to maximise the opportunity with the limited data set to find the best optimised model weights. It also manages shuffling of our training data.

Refinements

Data Pre-processing & word embedding assumptions and algorithms
At a high level, I noticed that my supervised models seem to reach a maximum F1 and accuracy value of approximately 78%. I believe that in order to achieve better results I should test my assumptions about data pre-processing. Certain actions such as removing stop words, may have actually negatively impacted my models as there is evidence that the removal of stop words and some other pre-processing actions I performed don't lead to an increase in model performance but actually worsen it as documented in various academic literature.

In order to test this theory, I introduced a new dataset: Trigrams with stop words. I chose trigrams as it was already the input data set that was giving me the highest f1 score. The result of introducing this data set was an increase in the f1 score from 78.6% to 82.32%, a 5% improvement.

Variable	Log Reg., Trigrams (Count Vectorizer)	MultinomialNB., Trigrams w/ stop words (TF-IDF)
F1-score (training)	0.960118	0.912637
F1-score (test)	0.786853	0.823119
Accuracy (training)	0.97	0.913333
Accuracy (test)	0.790741	0.807593
Prediction Time	0.11854	3.88924
Training time	892.955	37.7834

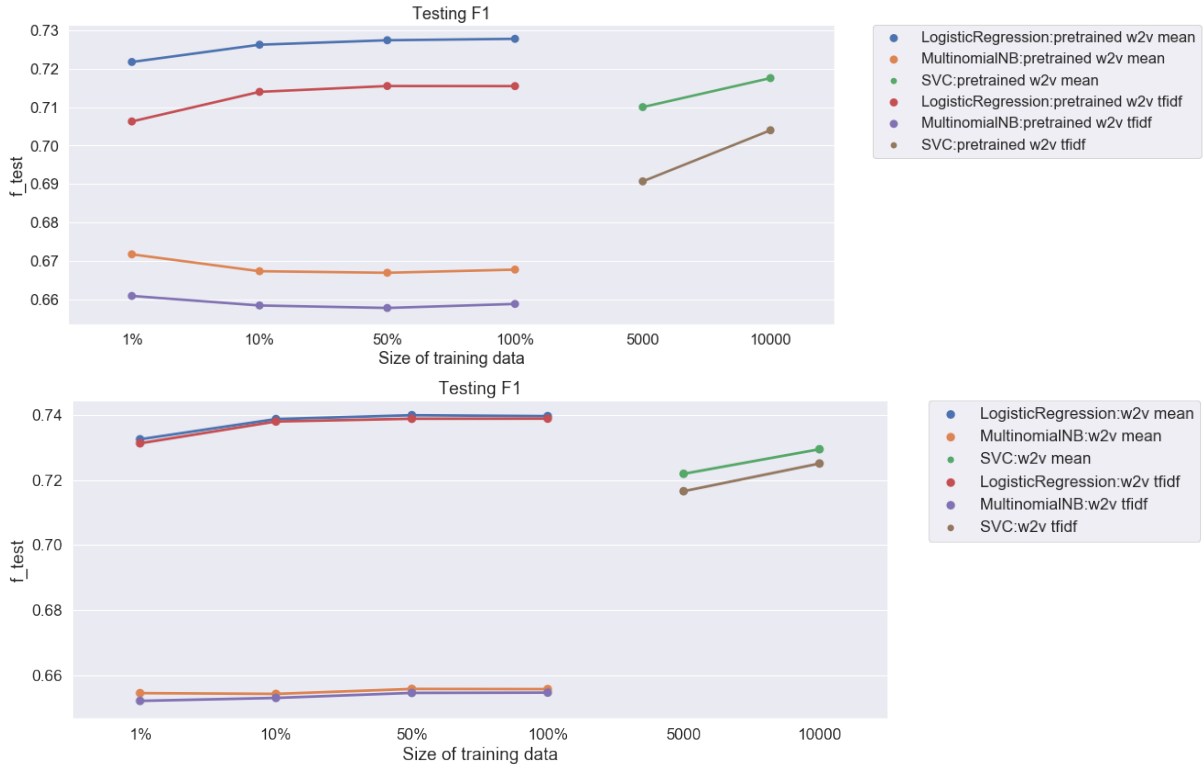


Above, I compare the best n-gram model without stop words (Logistic Regression with Trigrams using a Count Vectorizer) to the best n-gram model with stop words (MultinomialNB with trigrams and stop words using a TF-IDF vectorizer).

Beyond comparing only the best models above, it's clear from the diagrams, that when the stop words were used, overall model performance was better across both Multinomial and Logistic Regression models and nearly indifferent on SVM.

As another method to try and increase model performance, I introduced Word2Vec as a new embedding model. This model is similar to GloVe in the sense that it is unsupervised and creates a vector space with generally a few hundred features. To find out more about this, please refer to the Algorithms and techniques sections in the start of this report.

Variable	Log Reg., pre-trained word2vec	Log Reg., word2vec (trained on tweets corpus)
F1-score (training)	0.665295	0.685871
F1-score (test)	0.727768	0.739984
Accuracy (training)	0.706667	0.726667
Accuracy (test)	0.728111	0.740415
Prediction Time	4.55242	1.82694
Training time	2013.12	294.865

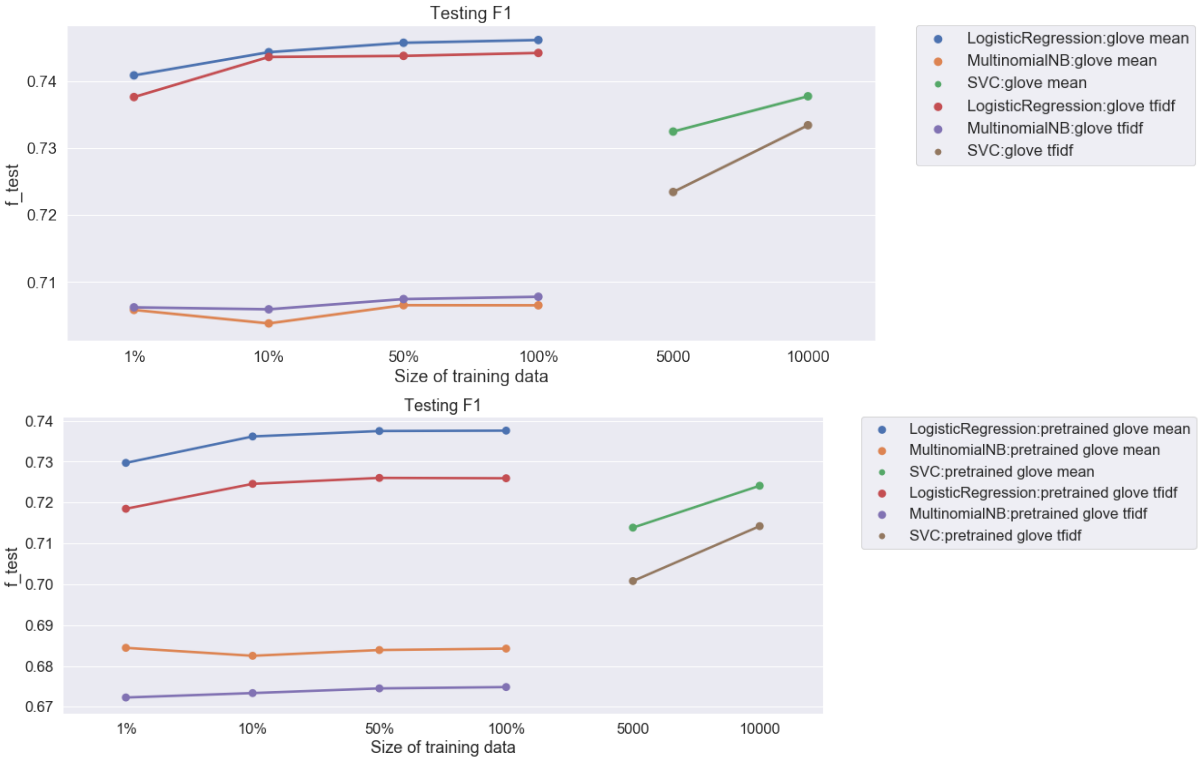


It is clear however that from the above, that just the n-gram models outperformed word2vec by over 6% without stop words and 10%+ with stop words on their f1 scores respectively. It is therefore clear that this refinement did not improve our model performance.

Finally, as mentioned in the start of this paper, I also made use of the GloVe model. For a full definition of this word embedding model and how I paired it with the selected supervised models, please read the previous section. In this scenario, the output was similar to our Word2Vec results and thus the best baseline model was the MultinomialNB model coupled

with trigrams with stop words and using a TF-IDF vectorizer. I've attached the GloVe results below further analysis.

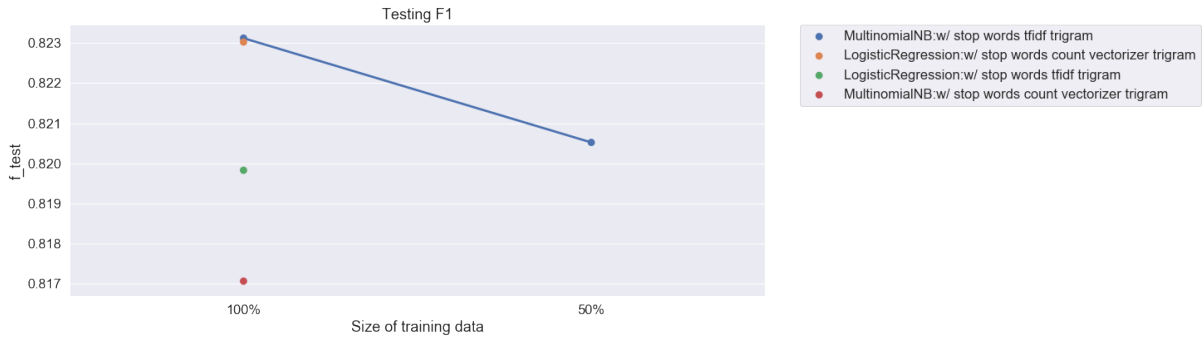
Variable	Log Reg., pre-trained GloVe w/ Mean Embedding Vector	Log Reg., GloVe (trained on tweets corpus) w/ Mean Embedding Vector
F1-score (training)	0.689405	0.666199
F1-score (test)	0.7376	0.746236
Accuracy (training)	0.726667	0.706667
Accuracy (test)	0.73729	0.746207
Prediction Time	2.63077	1.63669
Training time	4376.57	602.43



As can be seen above, GloVe also failed to out perform our n-gram models. However, an interesting observation is that in both cases, when using Word2Vec or GloVe, when the model was trained on our own corpus as opposed to using each pretrained version respectively, the models performed better.

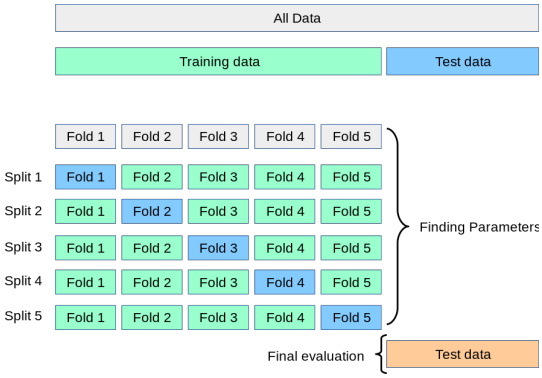
Optimisation of baseline supervised model

After working around all the various issues highlighted above, the best model appeared to be a multinomial model that made use of trigram word embeddings and included stop words. You can review this models performance in the table below. In comparison, the top 5 models' (before running Grid Search) F1 scores were as follows to give you some reference:



However, in total, before running Grid Search to optimise the best models hyper parameters, I tested over 190 model variations based on sample sizes, word embeddings and model classifiers. For comprehensive results on all of these models, please refer to the Jupyter notebook.

It is clear from the above that the Multinomial model, paired with trigram vectors, including stop words and using a TF-IDF vectorizer clearly outperformed any other model.



Upon optimisation, I tried to use Grid Search. This is the process whereby we cut our data set into multiple validation sets depending on how many hyperparameters we are trying to optimise.

Specifically I tested 20 different alpha values between 0 and our data was split into 10 different cross-validation sets, leading to over 200 different iterations to try and determine the best performing model.
In comparing the post-grid search optimised model to the baseline model, the results were the following:

Best alpha value: 0.631578947368421

Variable	Grid Search Optimised
F1-score (test)	0.8215345271261777
Accuracy (training)	0.8070217583139337
Accuracy (test)	0.8080681846584519

Based on all of my optimisations of our baselines and supervised models, I settled on the above as the best baseline model.

LSTM Model Optimisation
Due to time constraints, I wasn't able to manually play around with various hidden and LSTM layers in the LSTM model, however, I plan to spend time manipulating these layers manually to try and improve the f1 score of the model. I also chose an extremely simple model due to resource constraints.

It is important to note however that as discussed in the implementation section, I did make use of KFold cross validation to help refine and optimise the model.

Time & Resource limitations

Due to various time and resource limitations, I wasn't able to take a more scientific approach in testing my data pre-processing assumptions. Ideally, I would have liked to re-run each model training and predictions after each data pre-processing filter I applied to see the impact on f1 score that certain transformations such as removing digits, etc. have on the model.

I would have also have liked to re-tested all of my data sets on my LSTM model as opposed to simply re-using the same data set that performed the best on the supervised model. This is because it would be a very narrow minded assumption to presume that whatever data set and embeddings performed best on the supervised model would also perform best on the unsupervised model.

I had to limit the LSTM X training input size to 15 000 inputs only, only do 3 folds in our Kfold cross validation and only do 50 epochs per fold. This was to avoid getting memory issues during training and to avoid having to run Amazon EC2 instances for too long and costing me large amounts of money.

Lastly, I would have liked to have tested the results of combining various embeddings together, particularly with the LSTM network. For instance, combining GloVe with word2Vec or Glove with our best n-gram model (trigram w/ stop words using TF-IDF).

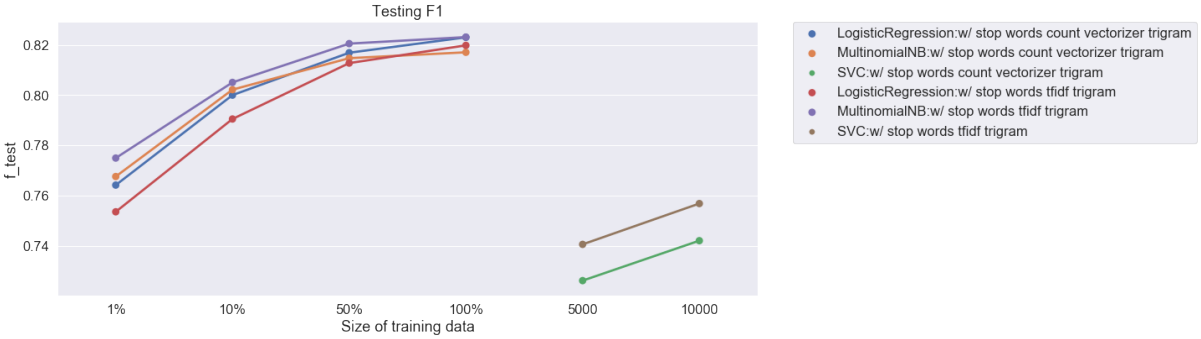
Results

Model Evaluation & Validation

Supervised / Baseline model

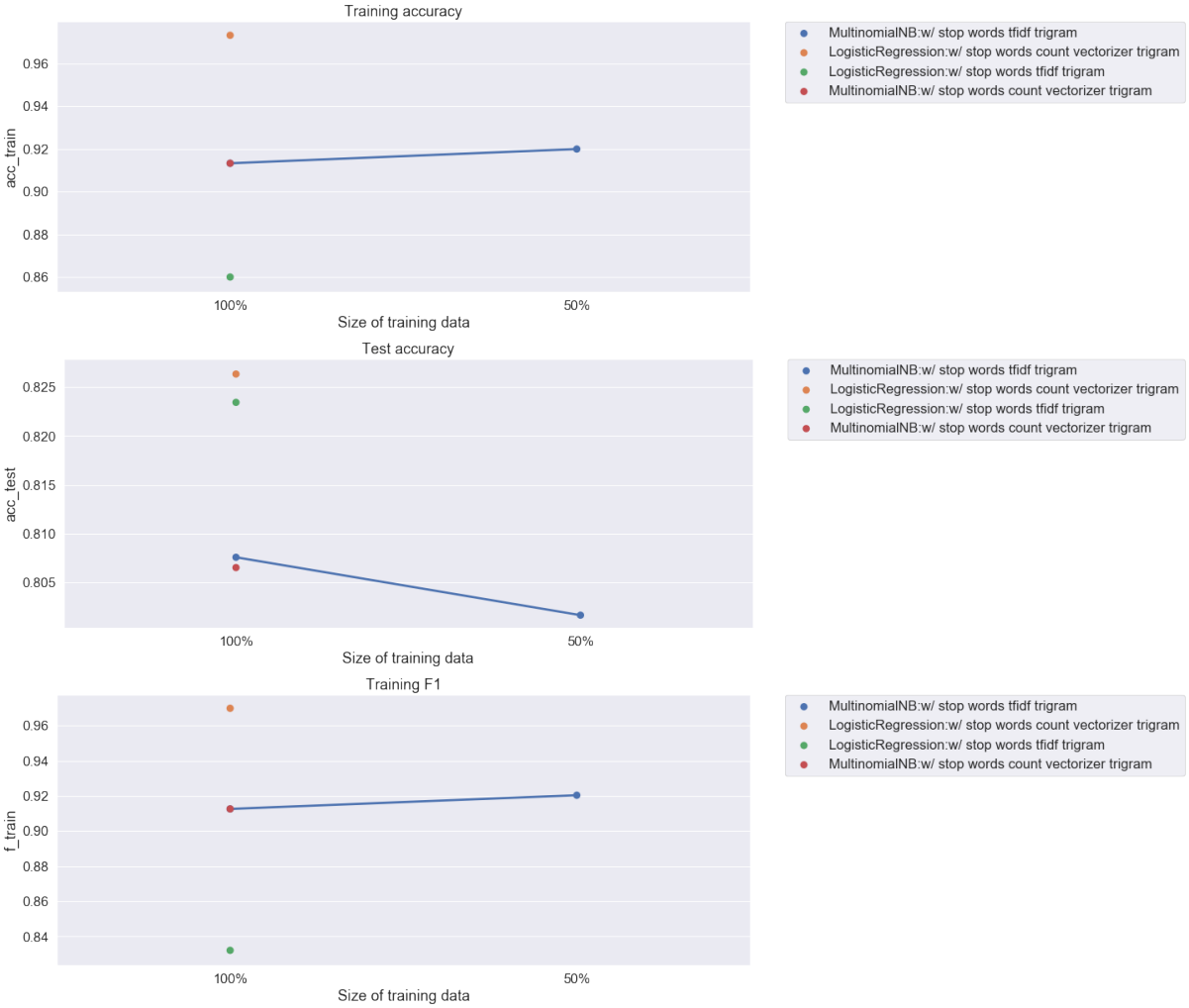
The final baseline / supervised model I settled on was a MultinomialNB model, making use of trigrams with stop words and using a TF-IDF vectorizer and an alpha value of 0.631578947368421. Please see a summary of testing results after Grid Search optimization below:

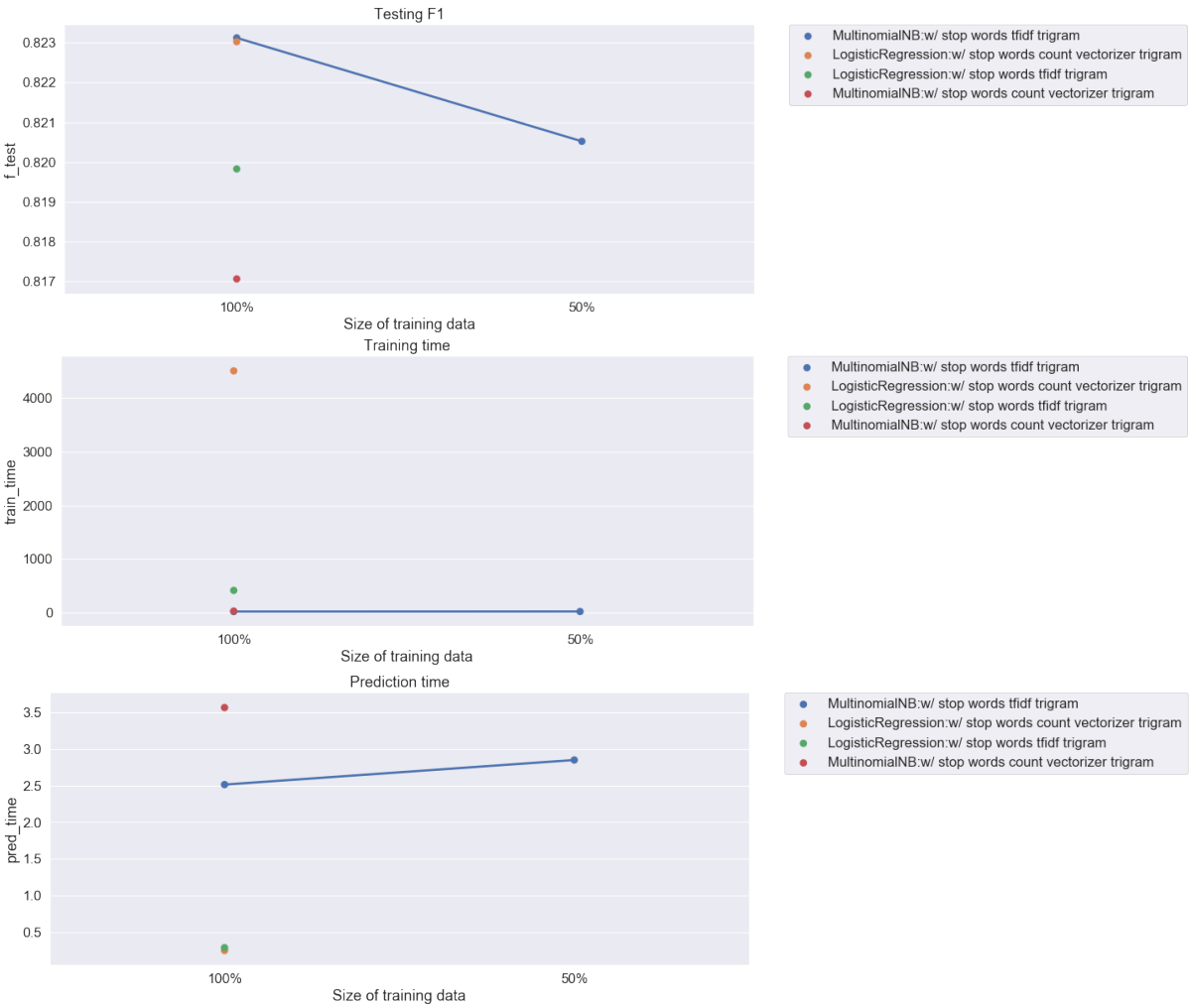
Variable	MultinomialNB., Trigrams w/ stop words (TF-IDF)
F1-score (training)	0.912637
F1-score (test)	0.823119
Accuracy (training)	0.913333
Accuracy (test)	0.807593
Prediction Time	3.88924
Training time	37.7834



Over the course of this project, I tested over 190 model variations, making it exceptionally difficult to visualise all of these models together. Please refer to the Jupyter notebook for these visualisations.

As a brief summary, I've attached a visualisation of the top 5 supervised models below:





I believe that the final supervised model chosen was very reasonable because of the following:

- Performs above average in comparison to most academic papers featuring supervised models on this data set
- The multinomial training accuracy & f1 is much lower than the logistic regression models indicating less overfitting on the training data and could explain why this model performs slightly better on the testing data
- In terms of the best performing models, it has a decent prediction time and is less than the logistic model
- Seems to generalise well to inputs that it hasn't seen before as showcased by the higher f1 score than accuracy, indicating that it
- It is a fairly robust model however because of it's simplicity in comparison to LSTM models and other unsupervised models which leverage perceptron's to rapidly increase the dimensionality of our solution functions, it probably does not adjust as well as these other algorithms would to unseen data.

Examples of correct classifications:

- Blink by malcolm gladwell amazing book and The tipping point!
- Malcolm Gladwell might be my new man crush
- F**k this economy. I hate aig and their non loan given asses.

- I was talking to this guy last night and he was telling me that he is a die hard Spurs fan. He also told me that he hates LeBron James.
- RT @shrop: Awesome JQuery reference book for Coda! http://www.macpeeps.com/coda/ #webdesign
- Math review. Im going to fail the exam.

Examples of incorrect classifications:

- obviously not siding with Cheney here: http://bit.ly/19j2d
- I just realized we three monkeys in the white Obama.Biden,Pelosi . Sarah Palin 2012
- Colin Powell rocked yesterday on CBS. Cheney needs to shut the hell up and go home.Powell is a man of Honor and served our country proudly
- In montreal for a long weekend of R&R

From the above examples, it's clear that when certain positive keywords appear, it is very easy for the supervised classifier to determine the sentiment however when less common words are used and presumably less featured during training as well, the classifier struggle to determine the correct sentiment as is the case with the first 2 incorrect classifications. The final classification contains both positive (i.e. rocked) and negative words (i.e. shut the hell up) and it is therefore understandable as to why our classifier would make a mistake here.

I am therefore comfortable that results from this model can therefore be trusted.

LSTM Model

On the LSTM Model, I failed to score an F1 score above 68% after training the model for upwards of 3 hours. For these reasons, I've chosen to focus more on the process and results of developing the baseline model above.

When testing the LSTM model, I also found that it failed to classify more basic tweets in comparison to the supervised learning method above, for example:

- Malcolm Gladwell might be my new man crush
- I was talking to this guy last night and he was telling me that he is a die hard Spurs fan. He also told me that he hates LeBron James.

For these reasons, I believe that:

- This model does not align with solution expectations as I've seen models listed in my citations that were LSTM RNNs, built in Keras, scoring accuracy scores in the 90th percentile. This model also vastly underperforms in comparison to my benchmark and therefore does not meet my expectations.
- I have tested this model with unseen inputs and specifically with inputs from validation and test data sets and have found, based on overall performance and some specific examples, that this model does not generalise as well as our previous model.
- This model is therefore seemingly less robust than our supervised solution as it does not seem to generalise well. Also, due to the lengthy training time and pipeline that needs to be put in place to manage feed in unseen tweets to this model (i.e. we need to first train a GloVe model, get the word embeddings, pad the tweet, train our

Keras model, feed our tweet into the Keras model, transform the output to be from a probability to a category of 0 or 1). This process is much longer than feeding simple trigram inputs into our supervised model.

- Trust is relative in this case as I'd prefer to trust the previous model than our final LSTM model due to the above reasons

Justification

As mentioned in the proposal of our project, the main metric we wanted to track was the f1 score. Below is a summary of these scores:

MultinomialNB, trigrams w/ stop words, TF-iDF vectorizer	LSTM network, GloVe pretrained on Wikipedia corpus
0.823119	0.630204

As you can see from the above, our supervised model managed to outperform the LSTM network by over 19%.

It is also worth noting that whilst I didn't manage to effectively time the training process for the LSTM network, I estimate that it took over 3 hours on a EC2 instance with 4 GPUS, and even then I had to really limit the training process which I believe fundamentally hindered this models performance (see the refinements > time & resource constraints section as well as the conclusion for more about this).

Therefore, the final results of the benchmark were the best results. I believe that overall, we can't say that we have solved the problem as our supervised model struggles to pick up nuances in the language used in tweets and therefore whilst it can generalise to most answers, it really struggles with more complicated tweets. In the case of our LSTM model, we see even more varying outputs as it struggles to classify even more basic tweets.

I believe that I have discussed in previous sections, in significant details, how both the supervised and unsupervised models were architected, how the pipelines were developed and therefore how I got to the final solution.

Conclusion

Free-Form Visualisation

I've attached the above to simply showcase the sheer volume of model variations that I tested over the course of this project.

The reason why I've also chosen the above visualisation is to showcase the clear gap (from 68% to 71% f1 score) in the graphs above. All of the models under that gap are Logistic Regression and MultinomialNB that were trained using GloVe or Word2Vec embeddings and all the graphs above the gaps were trained on n-gram embeddings.

I found this very interesting and showcasing the fact that sometimes simpler solutions can outperform more complicated solutions such as needing to train GloVe and Word2Vec embeddings and then build a specific pipeline to convert our tweets into vectors to evaluate as opposed to just creating the simpler n-grams that still performed better on our supervised models.

Reflection

Overall, my end to end solution and notebook comprised of the following:

1. Pre-processing the twitter data to reduce as many features as possible to help speed up the training process. This was predominantly done on gut feel and in reference to other academic works cited at the end of this document

2. Creating efficient, parallel programming pipelines for all of the supervised pipelines, allowing us to feed different classifiers, word embeddings and input data through the same pipelines
3. Creating Logistic regression, multinomial Naive Bayes and SVM classifiers
4. Testing n-grams (unigrams, bigrams, trigrams, trigrams with stop words) on each of the above supervised classifiers
5. Training a GloVe unsupervised model on our corpus and generating embeddings based on all of our tweets. I also downloaded a pre-trained GloVe model trained on the Wikipedia corpus and generated the embeddings based on all of the unique words in our tweets
6. Training a Word2Vec unsupervised model on our corpus and generating embeddings based on all of our tweets. I also downloaded a pre-trained Word2Vec model trained on the Google News corpus and generated the embeddings based on all of the unique words in our tweets
7. Feeding the GloVe and Word2Vec embeddings through vectorizers with our tweets we transform the tweets into their final word embeddings (either Mean Embeddings or TF-IDF embedding) and then train the above supervised models on all of the various embeddings
8. Analysed all of the baseline supervised results and chose the best model: MultinomialNB with trigrams with stop words and a TF-iDF vectorizer
9. Applied GridSeach optimisation to optimise the hyper parameters for the MultinomialNB (alpha)
10. Defined a simple LSTM RNN model
11. Put our pretrained GloVe training data through KFold cross validation and iterating through 3 folds, each doing a maximum of 50 epochs, feeding in batches of 500 tweets at a time that were padded. If the models f1 score failed to improve after 3 epochs, the current model training would stop. Every time we improved the model weights on an epoch, these weights would be saved to load for later
12. Tested our models on unseen data

What I found very interesting was learning:

- How different embedding systems work and how effective simpler embeddings can be (trigrams and n-grams) in comparison to more complex embeddings (word2vec and GloVe which make use of unsupervised models)
- How LSTM models work

What I found fairly challenging was:

- Time and resource constraints. Because of the many models I trained, tested as well as all of the variations of word embeddings, I found I really struggled to complete this project on time
- I also found I continuously was dealing with scripts taking to long to run and had to optimise by making use of multi-processing a lot, using generator functions in Keras and editing various training parameters (epochs, KFold, etc.). I feel that this led to me failing to spend enough time on the LSTM model and optimising this as I know

that this model, if trained, optimised and architected correctly has the potential to be much better than our supervised model.

I believe that the final model, the LSTM model, does not meet my expectations for the problem based on the above explanation and can be improved on based on the above explanation. However, I was very impressed by the possibilities out there and how by simply testing many variations in supervised classifiers and embeddings we managed to get an f1 score of over 80% which is a very good baseline solution for this problem.

Improvement

Some clear improvements that I would make in this project would be:

- To spend a lot more time on the data pre-processing step. I believe that there are a lot of small assumptions here that are made to help reduce the number of features in our data however as we saw when I tested keeping stop words IN the data, we can often be wrong in these assumptions. I would therefore like to generalize a pre-processing pipeline by applying the transformation, then compiling all of our models, judging the final f1 score and then deciding whether or not the transformation should be used or not.
- Spend more time on the LSTM network and optimizing this training process. I only used at maximum 15 000 tweets in the unsupervised learning process due to memory and other constraints. I would have much preferred to spend longer on this project and increase the number of training samples however I have already run out of free Amazon credits
- Combine input embeddings on the LSTM network. I think I could have really improved the LSTM networks performance had I combined various embeddings (such as adding trigrams to our GloVe inputs or combining GloVe and Word2Vec).
- Building a more robust prediction pipeline for the LSTM network as to represent a more commercially usable solution for this model. At the moment, this code would need to be substantially refactored.

Bibliography

Oppermann, A. (2019, 02 24). *Sentiment Analysis with Deep Learning of Netflix Reviews* . Retrieved from Towards Data Science: https://towardsdatascience.com/sentiment-analysis-with-deep-learning-62d4d0166ef6

Gizem Gezici, B. Y. (n.d.). *New Features for Sentiment Analysis: Do Sentences Matter?* Retrieved from http://ceur-ws.org: http://ceur-ws.org/Vol-917/SDAD2012_1_Gezici.pdf

Jabeen, H. (2018, 10 23). *Stemming and Lemmatization in Python* . Retrieved from DataCamp: https://www.datacamp.com/community/tutorials/stemming-lemmatization-python

Cooper, P. (2019, 01 16). *Twitter Statistics*. Retrieved from Hootsuite: https://blog.hootsuite.com/twitter-statistics/

Mika V. Mäntylä, D. G. (n.d.). *The Evolution of Sentiment Analysis - A Review of Research Topics, Venues, and Top Cited Papers.* Retrieved from arXiv.org: https://arxiv.org/pdf/1612.01556.pdf

Reptation.com editors. (2018, December 21). *5 Real-World Sentiment Analysis Use Cases* . Retrieved from reputation.com: https://www.reputation.com/resources/blog/5-real-world-sentiment-analysis-use-cases/

Bermingham, D. A. (n.d.). *Insight.* Retrieved from Sentiment Analysis Use Cases: https://www.isin.ie/assets/38/903B8F83-111B-4DCA-A84606B4F6E557B6_document/0915_Sentiment_Analysis_Dr._Adam_Bermingham.pdf

University, S. (n.d.). *For Academics* . Retrieved from sentiment140.com: http://help.sentiment140.com/for-students/

Pennington, J., Socher, R., & Manning, C. (2014, August). *GloVe: Global Vectors for Word Representation*. Retrieved from Stanford: https://nlp.stanford.edu/projects/glove/

Wikipedia. (n.d.). *Zipf's law*. Retrieved from Wikipedia: https://simple.wikipedia.org/wiki/Zipf%27s_law

Alec Go, R. B. (n.d.). *Twitter Sentiment Classification using Distant Supervision.* Stanford University.

Yao, M. (2018, July 24). *How to develop a hyper-personalized recommendation system* . Retrieved from freecodecamp: https://www.freecodecamp.org/news/how-to-develop-a-hyper-personalized-recommendation-system-ab9faf41b9a/

Shetty, B. (2018, November 5). *Learning to make Recommendations*. Retrieved from Towards Data Science: https://towardsdatascience.com/learning-to-make-recommendations-745d13883951

Admin. (2019, October 09). *Recurrent neural networks and LSTM tutorial in Python and TensorFlow* . Retrieved from Adventures in Machine Learning: https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/

Nguyen, M. (2018, September 24). *Illustrated Guide to LSTM's and GRU's: A step by step explanation* . Retrieved from Towards Data Science: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Seo, J. D. (2018, May 28). *[Basic Data Cleaning/Engineering Session] Twitter Sentiment Data* . Retrieved from Towards Data Science: https://towardsdatascience.com/basic-data-cleaning-engineering-session-twitter-sentiment-data-95e5bd2869ec

Fueyo, E. (2018, April 6). *Understanding what is behind Sentiment Analysis (Part II)* . Retrieved from lang.ai: https://building.lang.ai/understanding-what-is-behind-sentiment-analysis-part-ii-9307926d1435

Ghelani, S. (2019, May 16). *From Word Embeddings to Pretrained Language Models — A New Age in NLP — Part 1* . Retrieved from Towards Data Science: https://towardsdatascience.com/from-word-embeddings-to-pretrained-language-models-a-new-age-in-nlp-part-1-7ed0c7f3dfc5