Comparing Sentiment Analysis Algorithms

Michael Luke Ireland

5th June 2020

Contents

1 Introduction

This project focuses on the comparison of Sentiment Analysis algorithms (SAA) and various other forms of Natural Language Processing(NLP), with a chosen dataset of current US President and businessman Donald Trump's tweets.

Sentiment Analysis (SA) is classifying text into various classifications. For the purposes of this project, it is into 3 classes: negative, neutral and positive. Natural Language Processing is the subfield of linguistics and computer science that looks at how computers process and analyze speech and text. Sentiment Analysis is a form of NLP.

Example analysis methods include:

- Frequency Distribution
- Word Clouds
- Sentiment Analysis
- Tweet Generation

Frequency Distribution is another form of NLP that looks at the frequency of words and phrases within a given corpus. It is useful to be able to remove stopwords (words that don't have any sentiment) from a corpus, and to highlight which words are more important in deciding the general sentiment of a given corpus, or the specific sentiment of a given sentence or subsubsection.

Word Clouds provide a visual representation of the frequency of words and phrases within a given corpus.

Tweet Generation typically involves using Markov chains, to replicate the prose within a corpora, and use that understanding to generate tweets that have the same prose as the original corpora.

Markov chains are models that describe the sequence of possible events, and their possibilities are dependent on the state set in the previous event.

1.1 Background

I was interested analyzing Trump's tweets for sentiment due to his controversial nature. It would be interesting to see, given what the media often say about him, if he truly is a bad (negative) person via unbiased (relative to humans, who are emotional, have political and personal beliefs, and are often irrational) sentiment analysis, and letting a machine look just at the words used, rather than the character of the person saying them.

I had also imagined it would be entertaining to analyse his tweets in other was, such as word clouds, frequency distribution and tweet generation, to see if he is overly repetitive of particular words or phrases, such as "MAGA" or "Make America Great Again", and to see if I could make tweets that are very similar to his style, in order to see if it's possible to capture (part of) his cognitive function in a Python function.

While looking at the various algorithms available to perform this sort of task, I thought it would be worthwhile to evaluate which of them works best on tweets, which are typically informal, 280 character long posts submitted to Twitter, as opposed to other forms as text you would expect from someone in his position, such as a speech.

1.2 Aim

The aim of this project is to perform various types of analysis on the language used in Trump's tweet to see if any interesting trends arise. Mainly, this project seeks to inform and entertain people interested in Trump, for either good or bad reasons.

I will also be evaluating the performance of sentiment analysis algorithms on their speed, as well as their accuracy, precision and recall scores.

1.3 Objectives

- Perform frequency distribution on variable length phrases.
- Render word clouds of phrases.
- Analyse whole tweets for sentiment.
- Create unbiased classifer to initally label dataset.
- Find sentiment classifer algorithms.
- Compare classification algorithms to see which is the most accurate, most precise, yields the highest recall score, and the fastest.
- Create tweets using data from Trump's Twitter account.

2 Literature Review

I decided to use Python as it's my strongest language, plus its flexibility across platforms and level of API support makes it an obvious choice.

I originally planned to use Twitter's API via Twitter Search,[1] but I couldn't use it due to being unable to apply for a Twitter Developer Account. I instead opted for someone else's collected tweets at Trump Twitter Archive.[2] The export format wasn't great, as you had to wait a while for the page to compile all the tweets into the correct format (when it would be useful to have it precompiled) and the page doesn't actually give you a JSON file, just a text field with text in JSON format, that you have to slowly copy and paste into a file and use programs to format the JSON into readable format. This wasn't ideal, but was the best option available to me.

I saw guides such as Basic Binary Sentiment Analysis using NLTK,[3] Text Classification using NLTK[4] and Creating a Twitter Sentiment Analysis program using NLTK's Naive Bayes Classifier[5] using NLTK's (Natural Language Toolkit - A Python library for NLP) Naive Bayes Classifier, but they used pre-processed data meaning I could't use them for my tweets, as that would mean using an classifier trained on film reviews, on tweets, specifically do. This guide[6] used Google's Natural Language API to perform Sentiment Analysis, but this method required constant communication with Google's Cloud server, so was fairly slow overall.

Eventually, I fell upon this article[7] which used TextBlob to perform sentiment analysis instead. TextBlob is a simplified text processing library for Python, and provides a simple API for performing Natural Language Processing tasks, such as speech tagging, noun extraction, various forms of classification, translation and, most importantly, sentiment analysis. I used TextBlob briefly to get an understanding on how I could label tweets in preparation for the SAA comparisons.

I needed more methods of sentiment analysis, so I decided on Random

[1] Ckoepp. *ckoepp/TwitterSearch.* 2018. URL: `https://github.com/ckoepp/TwitterSearch`, p. 3.

[2] *Trump Twitter Archive.* URL: `http://www.trumptwitterarchive.com/archive`, p. 3.

[3] Samira Munir. *Basic Binary Sentiment Analysis using NLTK.* 2019. URL: `https://towardsdatascience.com/basic-binary-sentiment-analysis-using-nltk-c94ba17ae386`, p. 3.

[4] *Text Classification NLTK Tutorial.* URL: `https://pythonprogramming.net/text-classification-nltk-tutorial/`, p. 3.

[5] Anas Al-Masri. *Creating The Twitter Sentiment Analysis Program in Python with Naive Bayes Classification.* 2019. URL: `https://towardsdatascience.com/creating-the-twitter-sentiment-analysis-program-in-python-with-naive-bayes-classification-672e5589a7ed`, p. 3.

[6] Dzaky Widya Putra. *How to make your own sentiment analyzer using Python and Google's Natural Language API.* 2019. URL: `https://www.freecodecamp.org/news/how-to-make-your-own-sentiment-analyzer-using-python-and-googles-natural-language-api-9e91e1c493e/`, p. 3.

[7] *Twitter Sentiment Analysis using Python.* 2020. URL: `https://www.geeksforgeeks.org/twitter-sentiment-analysis-using-python/`, p. 3.

Forests, XGBoost,[8] Logistic Regression and Multilayer Perceptron models to compare to the Naive Bayes classifier.

Naive Bayes classifiers are simple probabilistic classifers which apply Bayes' theorem of naive assumptions between the features. I will be using a Multinomial Naive Bayes classifier, which uses feature vectors in the form of histograms, which count the number of times an event (in this case - sentiment) was observed in a particular instance (in this case - tweet).[9]

```
sentiments = {
    word:{
        "neg":0,
        "neut":0,
        "pos":0
    } for word in getwords() # Dictionary tracking sentiment history of words, where getwords() gets all words in given document
}

for tweet in tweets:
    sentiment = getlabel(tweet) # where getlabel() is a function that gets the sentiment of the labelled tweet.
    for word in tweet:
        if sentiment == -1:
            sentiments[word]["neg"]+=1
        elif sentiment == 0:
            sentiments[word]["neut"]+=1
        elif sentiment == 1:
            sentiments[word]["pos"]+=1
    return max(sentiments)
```

Listing 1: Multinomial Naive Bayes classifier psuedopythoncode

Random Forest classifiers operate by constructing multiple decision trees at training time, and use the mode of each individual tree's classification of the input vector (in this case - array of sentiments) to decide upon a class. A decision tree is a tree that consists of parent nodes that contain decisions or clauses, and leaf nodes with a classification.[10]

```
for tweet in tweets:
    tree = DTree(tweet)
    sentiment = tree.mode()
    print(tweet, sentiment)

class DTree:
    def __init__(self, text):
```

[8] *XGBoost in Python.* URL: `https://www.datacamp.com/community/tutorials/xgboost-in-python`, p. 4.

[9] *Naive Bayes classifier.* 2020. URL: `https://en.wikipedia.org/wiki/Naive_Bayes_classifier`, p. 4.

[10] *Random forest.* 2020. URL: `https://en.wikipedia.org/wiki/Random_forest`, p. 4.

```
        self.children = []
        self.scores = []
        self.sentiment = None
        self.text = text
        calculate()

    def calculate(self, weight=1):
        length = len(text.split())
        if length == 1:
            self.sentiment = get_sentiment(text) * weight #
    Sentiwordnet
            self.scores = [self.sentiment]
        else:
            for word in text:
                self.children.append(DTree(word))
            for child in self.children:
                self.scores += child.scores

    def mode(self):
        neutrals = 0
        negatives = 0
        positives = 0
        for child in self.children:
            if child.sentiment = -1:
                negatives += 1
            elif child.sentiment = 0:
                neutrals += 1
            else:
                positives += 1
        return max(neutrals, negatives, positives)

    def adjust(self, loss):
        new_children = []
        for child in self.children:
            new_children.append(child.calculate(weight=loss))
            self.children = new_children
```

Listing 2: Random Forest classifier psuedopythoncode

XGBoost classifier is a gradient boosting algorithm that uses proportional shrinking of leaf nodes, smart tree penalization and differentiation to improve it's understanding of each of the classes used in training. It uses a variety of parameters requiring optimisation to improve accuracy. Gradient boosting algorithms produce prediction models in the form of an ensemble weak prediction model, typically decision trees (similar to random forest). The models are then built in stages, and generalised by allowing optimisation of an arbitrary differentiable loss function, often softmax for multiclass classifiers.[11]

[11] *XGBoost*. 2020. URL: https://en.wikipedia.org/wiki/XGBoost, p. 5.

```
def softmax(tree):
    scores = []
    for child in tree.children:
        scores += child.scores
    return (e^scores)/sum(scores)

for tweet in tweets:
    ...
    loss = softmax(tree)
    while loss > desired:
        tree.adjust(loss)
        loss = softmax(tree)
```

Listing 3: XGBoost classifier psuedopythoncode

A Multilayer Perceptron is a type of feedforward artificial neural network, consisting of an input layer, a hidden layer and an output layer. Each non-input node is a neuron that uses a nonlinear activation function (to calculate the neuron's output from the input nodes), and uses a supervised learning technique called backpropagation (hence feedforward), similar to the least mean squares algorithm, for training. Backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output(tweet-sentiment) example. Learning is performed by changing the connection weights between the layers, based on the amount of error between the class prediction and actual class, calculated through backpropagation.[12]

```
class MLP:
    def __init__(self, text):
        self.input = [InputNode(word) for word in text]
        self.hidden = [HiddenNode(self.input) for node in self
    .input]
        self.output = OutputNode(self.hidden)
        for node in self.hidden
            node.parent = self.output
        for epochs in range(10):
            for node in self.hidden:
                node.calculate()
                print(self.output.prediction)

class Node:
    def __init__(self):
    pass

class InputNode(Node):
    def __init__(self, text):
    self.text = text

```

[12] *Multilayer perceptron.* 2020. URL: https://en.wikipedia.org/wiki/Multilayer_perceptron, p. 6.

```
class HiddenNode(Node):
    def __init__(self, children, weight=1):
        self.scores = None
        self.weight = weight
        self.children = children

    def calculate(self)
        scores = []
        for child in self.children:
            text = child.text
            scores.append(get_sentiment(text) * self.weight)
            self.loss = loss_func(scores) # where loss_func is
 any given loss func, such as least mean squares.
            self.scores = scores
            self.parent.result()

class OutputNode(Node):
    def __init__(self, children):
        self.prediction = None
        self.children = children
        self.weights = [1 for child in self.children]

    def result(self):
        scores = []
        for i in range(len(self.children)):
            scores.append(self.children[i].score * self.
    weights[i])
            self.prediction = mean(scores)
            self.weights = [loss_func_deriv(score) for score
    in scores] # Where loss_func_deriv is the inverse of
    loss_func.
```

Listing 4: Multilayer Perceptron classifier psuedopythoncode

Logistic Regression is a classifier model that uses a logistic function to model a dependent variable. It measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function, which is the cumulative distribution function of logistic regression.

I will specifically be using multinomial logistic regression, as I have 3 classes. The score vector for a given document (tweet) is in the format

$$score(X_i, k) = \beta_k \cdot X_i \tag{1}$$

Figure 1: MLR Equation

where X is a vector of the words that make up the tweet, and β is a vector of weights corresponding to outcome (sentiment) k. This score vector is then used by the softmax function to calculate a probability of the tweet belonging to that sentiment.[13]

Softmax is simply: $\frac{e^{scores}}{sum(scores)}$, where scores is a vector, where each element corresponds to a word and sentiment prediction.

```
def get_scores(words, weights):
    neg_weight = weights[0]
    neut_weight = weights[1]
    pos_weight = weights[2]

    neg_scores = [get_neg_score(word) for word in words]
    neut_scores = [get_neut_score(word) for word in words]
    pos_scores = [get_pos_score(word) for word in words]

    neg_scores = [score * neg_weight for score in neg_words]
    neut_scores = [score * neut_weight for score in neut_words
    ]
    pos_scores = [score * pos_weight for score in pos_words]

    return [neg_scores, neut_scores, pos_scores]

def softmax(scores):
    return math.E**scores/sum(scores)

weights = [1, 1, 1]

for tweet in tweets:
    probabilities = [softmax(score) for score in get_scores(
    tweet.split(), weights)]
    prediction = probabilities.index(max(probabilities))
    weights[prediction] *= 1.1
    for i in range(len(weights)):
        if i != prediction:
            weights[i] *= 0.9
    return prediction - 1
```

Listing 5: Multinomial Logistic Regression classifier psuedopythoncode

Stochastic Gradient Descent is an iterative method for optimizing an objective function with suitable smoothness properties. The objective function (loss relative to actual sentiment) to be minimised is:

$$Q(w) = \frac{1}{n}\sum_{i-1}^{n} Q_i(w) \qquad (2)$$

Figure 2: SGD Equation

[13] *Logistic regression.* 2020. URL: `https://en.wikipedia.org/wiki/Logistic_regression`, p. 8.

where the parameter w (sentiment polarity) is to be estimated. Each summand function Q<sub>i</sub> is typically associated with the i-th observation (i-th word) in the data set (tweet) of length n.[14] The algorithm works as follows:

1. Choose an initial vector of sentiments w, and learning rate r. 2. Repeat until an appropriate minimum is obtained: 1. Randomly shuffle examples in the training set 2. $Foriinrange(n)$: 1. $w = w - r * loss(w)$

Due to their simple effectiveness, I predict Naive Bayes, Random Forest, and Logistic Regression will perform the best to begin with, certainly in terms of speed, but with tweaking, Multilayer Perceptron and XGB should provide comparable or better accuracy.

After coming across this article[15] on algorithm comparisons, I found that creating a tf-idf transformer to use on the initial bag of words model, prior to training, massively boosts accuracy. The article also demonstrates a fair, reproducible method of algorithm comparison, using Scikit Learn's Pipelines, so I began altering my code to minimise the difference between how classifiers are ran, to isolate the performance of the classifier down to the algorithm itself and not any slight differences in pre-processing. Scikit Learn is another Python NLP library that builds on the NLP available in NLTK. Pipelines are flexible data types that contain steps which define how a particular algorithm should be prepared, and then trained.

The function that would be used to evaluate the pipelines: `cross_val_score`, gave a `scoring` parameter with many options,[16] I chose accuracy score, which simply compares prediction labels with actual labels, but there were a range of alternatives available, such as average precision, balanced accuracy score, Brier score, F1 score and Normalized Discounted Cumulative Gain. I later went back to this and decided to add multiple metrics, so I had to instead use `cross_validate` and chose Precision score, and Recall score. These two score types required an averaging function to be set, with the options being: Micro - Perform the scoring function, but don't average; Macro - Perform the scoring function, then find the unweighted mean; Weighted - Perform the scoring function, then find the weighted mean. Weighted was the only scoring function that accounted for label imbalance, so as I already had accuracy as a metric, I thought weighted made the most sense.

For tweet generation, I used Markovify,[17] which I found from this article[18] attempting the same thing. Markovify is a Python library that uses Markov

[14] *Stochastic gradient descent.* 2020. URL: https://en.wikipedia.org/wiki/Stochastic_gradient_descent, p. 9.

[15] Avishek Nag. *Text Classification by XGBoost & Others: A Case Study Using BBC News Articles.* 2019. URL: https://medium.com/towards-artificial-intelligence/text-classification-by-xgboost-others-a-case-study-using-bbc-news-articles-5d88e94a9f8, p. 9.

[16] *API Reference¶.* URL: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics, p. 5.

[17] Jsvine. *jsvine/markovify.* 2020. URL: https://github.com/jsvine/markovify, p. 9.

[18] Melissa Hall. *WhatWouldTrumpTweet: Topic Clustering and Tweet Generation from Donald Trump's Tweets.* 2017. URL: https://medium.com/@mc7968/whatwouldtrumptweet-topic-clustering-and-tweet-generation-from-donald-trumps-tweets-b191fccaffb2, p. 9.

chains to generate text based on an input corpora. The article listed multiple approaches, including using a Keras API and k-means clustering to build a Machine Learning model to feed into tweet generators, but that added a significant layer of obscurity, and made less coherent tweets.

3 Development Methodology

I decided on an Agile approach, as requirements weren't well known at the start, with sprints lasting a week, as I had weekly meetings where I would discuss my progress with the client (project supervisor). I would make clear targets at the beginning of the sprint and make sure they're done by the end of the sprint. This was mostly successful and I feel like I got a lot done by creating small tasks into tickets on JIRA after discussing them with the client, then commit to completing the most important tickets first. My biggest strength was coding (implementing logic) and refactoring (making code easier to understand). My biggest weakness was writing the report, as I found it hard to think about what is relevant or interesting within each chapter. I imagine this is probably a result of the Agile approach, as I had to create more work for myself in order to write more.

I followed the MoSCoW method of Must Have, Should Have, Could Have and Won't Have, with Must Have tickets or epics being the most time critical, and Won't Have being the least time critical. Epics were creating to link tickets together, so that I could complete related same-priority tickets at the same time, to increase logical cohesion in any work I completed.

Here is a cumulative flow diagram, demonstrating my ticket completion and creation rates:

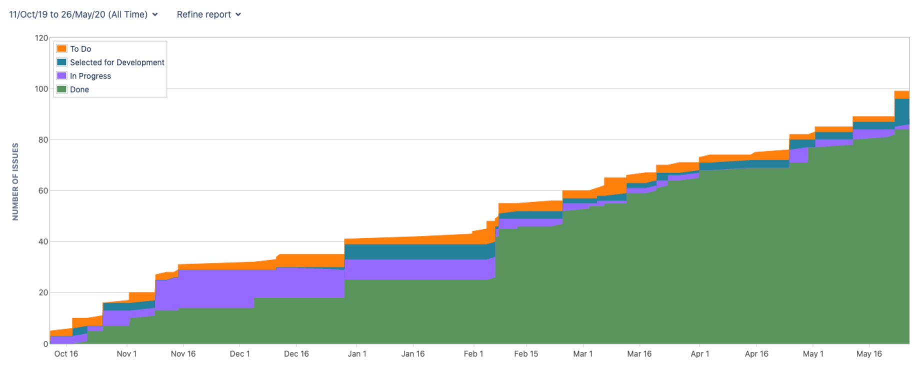


Figure 3: Cumulative Flow Diagram

4 Data Pre-processing

4.1 Dataset

The dataset I extracted from Trump Twitter Archive, was last updated 5th March 2020 and contains 46208 tweets.

I used NLTK to look at the most common words and phrases of different lengths, to explore the dataset.

Here is a word cloud I created using my custom TweetCloud class that uses Python library wordcloud, but adds important features for NLP abstraction, **without** tweet cleaning.

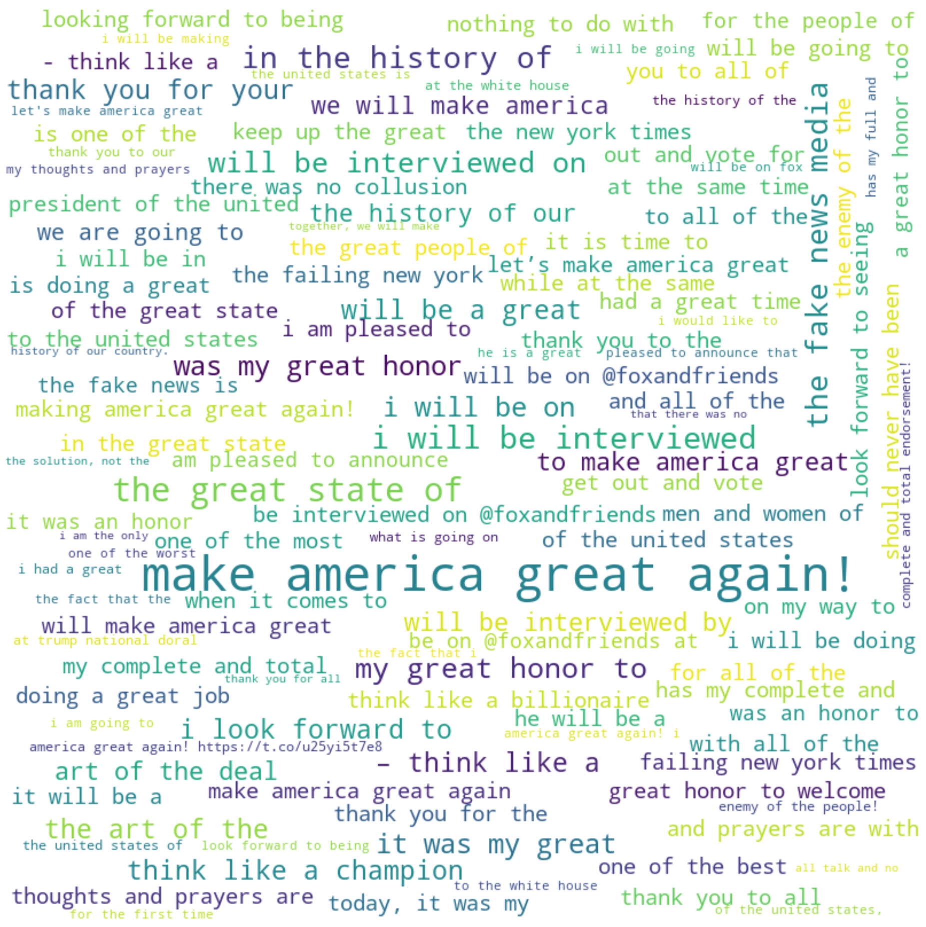
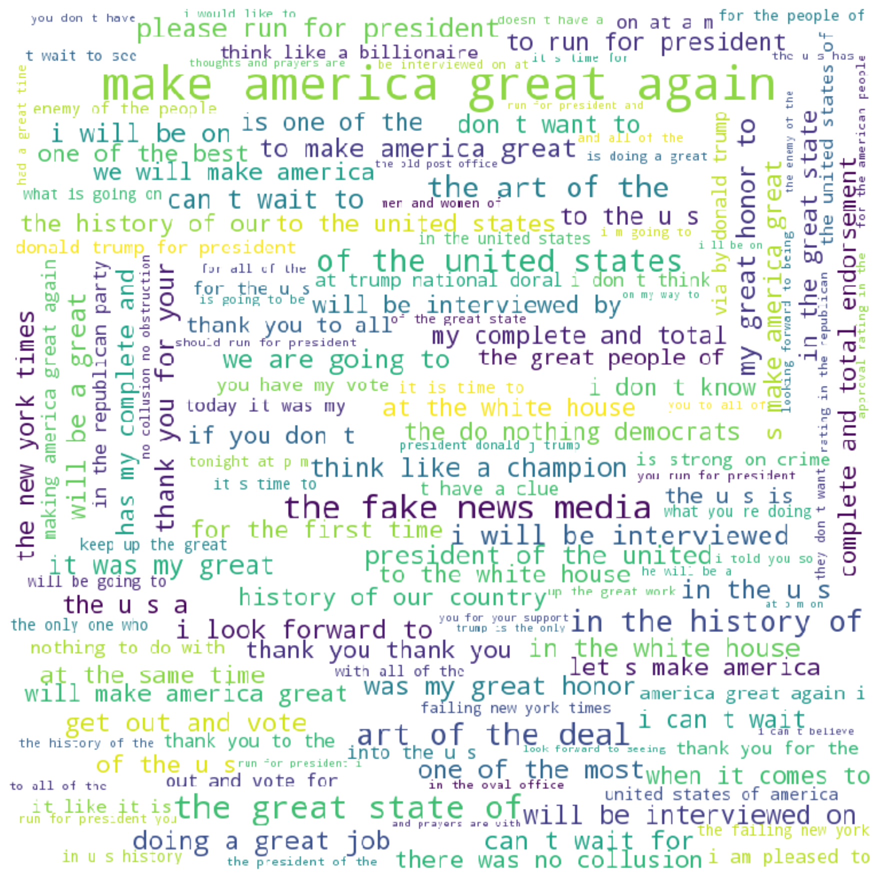


Figure 4: WordCloud of phrases of length 4 (uncleaned)

Here is one produced after tweet cleaning:



(a) WordCloud of phrases of length 4 (cleaned)

I created my own class using Markovify, that would force the library's generator to generate tweets (280 characters max) in Trump-like prose containing a given input word.

Here are 5 examples of tweets with "Biden" as the input.

4.1.1 Generated Tweets

Trump said Biden 108 times Hard for Biden on his ridiculous Climate Change Plan is a heartbeat away from you! RT @GreggJarrett: Here's Why The Justice Department Must Investigate Joe Biden's *obvious* Corruption &; why Congressional Democrats' push for impeachment is… RT @America1stTiger: Just in case you forgot...I have @JoeBiden here admitting to a request for Ukraine and in China. If the V.P. of the American people care about Hunter Biden brought to the euphoria of getting Obama/Biden OUT, &; getting Trump/Pence

IN. China wants Sleepy Joe Biden, on the other hand, take out millions of American...

4.2 Labelling

I used Python's JSON library to load the tweets.json file into the program as a dictionary, implemented in Python as a dict. The dict contained lots of useful information, but I was actually only interested in the tweets themselves, so I extracted them, cleaned them up by removing anything that wasn't a word (URLs mostly, numbers, symbols). These cleaned tweets then needed to be labelled, so I created my own classifier due the size of the data, rather than labelling 46208 tweets myself.

My classifier tokenised the cleaned tweets, then applied lemmatisation on the tokens, which is an advanced form of stemming. Stemming is the process of reducing inflections or derivations to their base form. Tokenisation is another type of cleaning which throws away any useless (relative to NLP) information from a given input and lemmatisation looks at the context of each word in the given text, unlike stemming, then reduces all forms/inflections of a word into its base form, based on that context. This base form is then compatible with WordNet which is a lexical database created by psychology professor George Armitage Miller of Princeton University, that groups words into cognitive synonyms or synsets.[19]

Synsets are grouped data elements considered semantically equivalent for information retrieval purposes.[20]

These base forms can be further processed, by tagging them with the appropriate usage, meaning the most accurate synset can be retrieved. This synset is then used by SentiWordNet to retrieve a sentiment score. SentiWordNet is a lexical resource used for opinion mining created by Andrea Esuli of the Institute of the National Research Council of Italy.[21]

These sentiment scores for each individual word in a tweet, are used to build an aggregate score for the tweet, and negation is used when words were used next to modfiers like "not" and "no". There are more possible modifers, such as "very", "hardly", "extremely", "barely" or "never" - "It's never a good idea to...", but these are more complex modfiers than the ones I implemented and were fairly out of scope, given my project is more to do with comparing existing algorithms, than making a perfect one myself.

These scores are used to calculate a polarity (negative, neutral, positive) using a threshold I set through trial and error to see if the labels set on a large sample of tweets were reasonable and accurate enough. I couldn't possibly hope to verify my classifier is accurate for every single tweet, again, due to the size of the data. However, this doesn't matter, as the point of the project is to compare how other classifiers understand the link between a tweet and label, in order to

[19] *WordNet*. 2020. URL: `https://en.wikipedia.org/wiki/WordNet`, p. 14.

[20] *Synonym ring*. 2017. URL: `https://en.wikipedia.org/wiki/Synonym_ring`, p. 14.

[21] Andrea Esuli. *SentiWordNet*. URL: `https://github.com/aesuli/SentiWordNet/blob/master/papers/LREC06.pdf`, p. 14.

create their own label. If the classifier is accurate, they should be able to label a tweet with the same sentiment as my classifier, regardless of the polarity.

This resulted in rather narrow ranges for each sentiment class.

- Less than -0.125 = Negative - Between +0.125 and -0.125 = Neutral - More than +0.125 = Positive

The tweets and polarities were then fed into a pandas DataFrame, then provided to the Scikit Learn pipelines, containing implementations of the algorithms to be compared.

I evaluated my classifier's labelling accuracy based on my own interpretation of a tweet's sentiment, and if my classifier was true to my interpretation for a reasonably large sample of tweets (around 100).

pandas is a Python library for data analysis and manipulation, and provides a DataFrame class, which is very useful for organising data, and is compatible with most NLP libraries in Python.

I decided to standardise my scores using the z-score method of $newscore = \frac{rawscore - mean}{standarddeviation}$, where raw score is the original score for a given tweet calculated by my classifier. All scores are calculated by my classifier, then a mean and standard deviation for all tweets are collected to calculate the z-score for each tweet.[22]

[22] *Cross-validation (statistics).* 2020. URL: `https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation`, p. 15.

5 How I compared each Sentiment Classification algorithm to one another

5.1 Overview

I originally intended to implement the algorithms themselves, but came across Scikit-Learn's pipelines, which gave very easily reproducible and fair ways of running classification algorithms, which was immensely useful to me trying to compare these algorithms. I could have used these pipelines with my own implementations, but that would then require the further diversion of learning how to make it compatible, plus their implementations will have been well tested and scrutinised. I instead opted to go along with their implementation which slot into their pipelines very nicely. Pipelines contain function calls as steps, which are used on the input data, through the pipeline training function provided by Scikit-Learn: cross_val_score.

As a result, I only chose algorithms that were implemented by or compatible with Scikit-Learn, meaning I had to drop some algorithms I was considering for comparison during my research stage. Each algorithm was implemented as a class, with appropriate evaluation functions, and allowed for configuration through parameters. XGBoost offered the largest amount of customisation, but there were only a few options for multi-class classification, so I went with softmax for its objective parameter, as it had the highest accuracy out of the few options available.

I wanted to use k-fold cross validation on the data, in order to increase the fairness of training. This involves splitting the data into k equal sized partitions, where one of the partitions is used as validation data for testing the model, with the remaining k-1 partitions used for training the model. This split repeats k times, in a way in which every partition is used as the validation at least one, then averaged to produce a single estimation.

K-fold cross validation is built into cross_val_score, which has k-value parameter, which I passed a value of 10 for.

The data is pre-processed and formatted, as described in Chapter 3, and a pipeline is created for each algorithm inside a wrapper class. The wrappper class initially contains a name and pipeline, but will later contain accuracy and time information. All classes are initialised inside a list, then each class has its pipeline passed to the evaluation function (cross_val_score), as well as the tweets, and labels from the data. This evaluation is timed, and the function itself returns an accuracy score for each of the k validations. This time and the mean of the accuracies are added to the wrapper class.

Each algorithm had to be trained through all 46208 tweets, 10 times through k-fold cross validation, so I implemented multithreading to better use of system resources when running the program. I used Python's multithreading library concurrent, which contains a module futures, with the function ThreadPoolExecutor , which is used to create multiple future objects able to be processed on their own separate thread. Adding this made the program around 2.5x faster, and finished in 14 minutes instead of 36.

A graph is created to compare accuracy against time using matplotlib.pyplot.

I later decided I wanted to train on various sizes of tweets, and use more than one metric, so I switched to cross_validate from Scikit-Learn and used multiprocessing. cross_validate is identical to cross_val_score, but allows for multiple metrics. multiprocessing uses multiple processes, instead of threads, which in this case, resulted in better performance. Upon returning to this to measure how much faster this improvement, I found a further improvement in Ray's multiprocessing library.[23] Ray is a Python library focusing on distributed and parallel processing.[24] Originally, I had decided to compare the serial case to both Python's and Ray's multiprocessing libraries. Developing on AArch64, meant that I had to build Ray myself, which introduced many problems, and I ultimately decided to abandon this as I couldn't successfully build it.

The parallel version took 2.79 hours, and the serial version took 4.18 hours. This is an improvement of around 150%.

The training time per "step" (a given data length) was improved, but it wasn't by a huge amount. I added accuracy, precision_weighted and recall_weighted to the scoring array passed to cross_validate which returned results for each metric, as well as a score_time passed this information through tuples to create an Algorithm class, with attributes name, accuracy, precision, recall and time. I created a dictionary for each data length, with each value being another dictionary containing the results for each algorithm, as well as aggregated statistics, such as mean and standard deviation. I also created an individual graph for each data size.

The data lengths used were 1/8, 1/4, 3/8, 1/2, 5/8, 3/4, 7/8, 1/1 of the total 46208 tweets.

These dictionaries were passed to an export function that saved the results to a text file, and created a combined graph. I also created an import function if I wished to alter the graph, without running the 3 hour long program again.

5.2 My Code

The general structure of my program is modular, with training, pre-processing and pre-labelling all split up into seperate python files within the same module.

Pre-labelling is handled by utils/lukifier.py which mainly revolves around the function swn_polarity. swn_polarity uses WordNet tags to get the right synset from sentiwordnet and extract the corresponding sentiment score. I added a very small example of negation, but this is a naive classifier, and can't handle overly complex English sentences, which is fine, due to general prose used for tweets, and Donald Trump's in particular.

```python
def swn_polarity(self):
```

[23]Robert Nishihara. *10x Faster Parallel Python Without Python Multiprocessing.* 2020. URL: https://towardsdatascience.com/10x-faster-parallel-python-without-python-multiprocessing-e5017c93cce1, p. 17.

[24]Ray-Project. *ray-project/ray.* 2020. URL: https://github.com/ray-project/ray, p. 17.

```
    """
    Return a sentiment polarity: -1 = negative, 0 = neutral, 1
    = positive
    """

    sentiment = 0.0
    tokens_count = 0

    raw_sentences = sent_tokenize(self.text)
    for raw_sentence in raw_sentences:
        tagged_sentence = pos_tag(word_tokenize(raw_sentence))

        negated = False
        for word, tag in tagged_sentence:
            self.tag = tag

            if word.endswith("n't") or word == "not":
                negated = True

            if len(word) <= 2:
                continue

            wn_tag = self.penn_to_wn()

            if wn_tag:
                lemma = lemmatizer.lemmatize(word, pos=wn_tag)
            else:
                continue

            if not lemma:
                continue

            synsets = wn.synsets(word, pos=wn_tag)
            if not synsets:
                synsets = wn.synsets(word)
            if not synsets:
                continue

            # Take the first sense, the most common
            synset = synsets[0]
            swn_synset = swn.senti_synset(synset.name())

            pos_score = swn_synset.pos_score()
            neg_score = swn_synset.neg_score()

            if negated:
                pos_score = pos_score*-1
                neg_score = neg_score*-1

            if neg_score == pos_score:
```

```python
                pos_score = 0

            diff = pos_score - neg_score
            self.words.append((word, diff))

            sentiment += diff
            tokens_count += 1

    # judgment call ? Default to positive or negative
    if not tokens_count:
        return 0

    self.score = sentiment

    # positive sentiment
    if sentiment >= 0.125:
        return 1

    # negative sentiment
    elif sentiment <= -0.125:
        return -1

    # neutral sentiment
    return 0
```

Listing 6: Lukifier's sentiment labelling function

As you can see, the function takes any block of text and finds the sentiment score for each word within that text, using lemmas, negations and tags to correctly and fairly identify usages of words. Invalid input will simply return a neutral score of 0.

Pre-processing is handled by utils/tweet_getter.py which reads the resources/tweets.json file as a dictionary, cleans them as previously described, then uses utils/lukifier.py to get a label. It can return uncleaned tweets, cleaned tweets, and cleaned labelled tweets; both standardised and non-standardised. The classifiers to be compared are obviously interested in the latter.

```python
def get_clean_tweets_with_scores(self):
    if not self.tweets_with_scores:
        print("Getting scores")
        if not self.cleaned_tweets:
            self.cleaned_tweets = self.get_clean_tweets()
        data = {
            'tweet': [],
            'score': [],
            'polarity': []
        }
        for tweet in self.cleaned_tweets:
            classifier = Lukifier(tweet)
            data['tweet'].append(tweet)
            data['score'].append(classifier.score)
            data['polarity'].append(classifier.polarity)
        self.tweets_with_scores = pd.DataFrame(data)
    print("Got {} scores".format(len(self.cleaned_tweets)))
    return self.tweets_with_scores

def standardise(self, x, std, mean):
    return (x-mean)/std

def classify(self, sentiment):
    if sentiment >= 0.125:
        return 1

    elif sentiment <= -0.125:
        return -1

    return 0

def get_standardised_tweets(self):
    if not self.tweets_with_scores:
        self.get_clean_tweets_with_scores()
    std = self.tweets_with_scores.score.std()
    mean = self.tweets_with_scores.score.mean()
    self.tweets_with_scores.score = [self.standardise(
        x, std, mean) for x in self.tweets_with_scores.score]
    self.tweets_with_scores.polarity = [
        self.classify(x) for x in self.tweets_with_scores.score]
    return self.tweets_with_scores
```

Listing 7: Tweet Cleaner & Feature and Label setup

As you can see, the `standardise()` function is called on all labelled tweets, to adjust their corresponding sentiment score using the z-score method. The tweets are then re-classified based on their new standardised score.

The data returned from `utils/tweet_getter.py` is used in the main file `main_async.py`. The main function gives you an initial option to restart training, or continue from where the program last terminated. The program then gets the labelled tweets and calls `progressive_train` using the tweets and corresponding sentiments.

```python
def progressive_train(X, y, save_path):
    results = []

    orig_size = len(X)
    sizes = [i*0.125 for i in range(1, 9)]
    lengths = [int(size*orig_size) for size in sizes]

    print("Training on tweets of sizes: {}".format(lengths))
    print("Original size is: {}".format(orig_size))

    for length in lengths:
        X_train, y_train = get_slice(X, y, length)
        print(length, len(X), len(X_train), len(y_train))
        if length in lengths:
            algorithms = train(X_train, y_train, save_path, length)
            if algorithms:
                if not results:
                    results = algorithms
                    for result in results:
                        result.time = [result.time]
                        result.accuracy = [result.accuracy]
                        result.precision = [result.precision]
                        result.recall = [result.recall]
                else:
                    for i in range(len(results)):
                        results[i].time.append(algorithms[i].time)
                        results[i].accuracy.append(algorithms[i].accuracy)
                        results[i].precision.append(algorithms[i].precision)
                        results[i].recall.append(algorithms[i].recall)

        time_string = time.strftime("%H:%M:%S", time.localtime())
        print("Training for size {} finished at {}".format(length, time_string))

    plt.autoscale(True)
    for algorithm in algorithms:
        print(algorithm)
        plt.plot(algorithm.accuracy, algorithm.time, "{}o".format(
            algorithm.colour), label="{} (Accuracy)".format(algorithm.name))
        plt.plot(algorithm.precision, algorithm.time, "{}v".format(
            algorithm.colour), label="{} (Precision)".format(algorithm.name))
        plt.plot(algorithm.recall, algorithm.time, "{}s".format(
            algorithm.colour), label="{} (Recall)".format(algorithm.name))

    plt.xlabel("Score")
    plt.ylabel("Time (s)")
    lg = plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
    save_file = Path("graphs/combined.svg")
    plt.savefig(str(save_file),
                dpi=300,
                format='svg',
                bbox_extra_artists=(lg,),
                bbox_inches='tight')
```

Listing 8: Progressive Training function

progressive_train() is an optional setup function allowing a user to use the main train() function on varying sizes of tweets. progressive_train() finishes by plotting an aggregated graph.

train() is given any length of data, then calls all 6 algorithm implementations using the data, gathering the accuracy, precision, recall and time scores for each. The mean and standard deviation for each of these scores is also calculated, then all data produced is saved to a file using save_dict() and a template TEMPLATE. A graph for an invididual run of train() is also produced using save_graph_individual().

```
def train(X, y, save_path, length):
    size = len(X)
    if size != length:
        return

    save_file = save_path.joinpath("size{}.svg".format(size))
    print("Searching for results at {}".format(save_file))
    if save_file.exists():
        return
    print("Training on {} tweets".format(size))

    algorithms = [XGBoost(), LogisticRegression(),
RandomForest(),
                  MultilayerPerceptron(), NaiveBayes(),
StochasticGD()]

    start = timer()
    args = [(algorithm, X, y) for algorithm in algorithms]
    with multiprocessing.Pool() as pool_inner:
        algorithms = pool_inner.starmap(evaluate, args)
    print("Total training time for size {}: {}s".format(size,
timer() - start))

    algorithms.sort(key=lambda x: x.accuracy, reverse=True)

    accuracies = [algorithm.accuracy for algorithm in
algorithms]
    mean_acc = np.mean(accuracies)
    std_acc = np.std(accuracies)

    precisions = [algorithm.precision for algorithm in algorithms]
    mean_pre = np.mean(precisions)
    std_pre = np.std(precisions)

    recalls = [algorithm.recall for algorithm in algorithms]
    mean_rec = np.mean(recalls)
    std_rec = np.std(recalls)

    times = [algorithm.time for algorithm in algorithms]
    mean_time = np.mean(times)
    std_time = np.std(times)

    data = {
        "Name": [algorithm.name for algorithm in algorithms],
        "Accuracy": accuracies,
        "Accuracy (Mean)": mean_acc,
        "Accuracy (std)": std_acc,
        "Precision": precisions,
        "Precision (Mean)": mean_pre,
        "Precision (std)": std_pre,
        "Recall": recalls,
        "Recall (Mean)": mean_rec,
        "Recall (std)": std_rec,
        "Time (s)": times,
        "Time (s) (mean)": mean_time,
        "Time (s) (std)": std_time
    }

    save_dict(data, size, save_path)

    save_graph_individual(algorithms, size)

    return algorithms
```

Listing 9: Main training/comparison function

As you can see, the data length is verified before proceeding with training. The algorithm wrapper classes are setup, before being fed into a tuple with the data, then processed asynchronously. The processing function in this case is `evaluate()`, which calls the `cross_validate()` function from Scikit-Learn, along with the parameters previously described.

```
def evaluate(algorithm, X, y):
    scoring = ['accuracy', 'precision_weighted', 'recall_weighted']
    function = wrapper(
        cross_validate, algorithm.pipeline, X, y, cv=10, scoring=scoring)
    scores, time = run_algorithm(function)
    algorithm.accuracy, algorithm.precision, algorithm.recall
 = scores
    algorithm.time = time
    return algorithm
```

Listing 10: Function for getting scores for a single algorithm

The use of wrappers for both functions and classes is very useful for customisability and comprehension, and it leaves each function reasonably abstract from other functions, meaning it's both understandable, but easy for someone to build on the existing setup, and add different metrics, algorithms or use different evaluation functions.

An example algorithm wrapper class may look like this:

```
class LogisticRegression:
def __init__(self):
    self.name = "Logistic Regression"
    self.pipeline = Pipeline(
        steps=[
            ('tfidf', TfidfTransformer()),
            ('log_reg', lp(
                multi_class='multinomial', solver='saga',
max_iter=100))
        ]
    )
    self.colour = 'g'
```

Listing 11: Example classification algorithm wrapper class

All wrapper classes have a name, a Scikit-Learn compatible pipeline and a colour for use in graphs (for consistency reasons). All wrapper classes have a TfIdf transformer applied before the algorithm is called, to boost scores.

6 Evaluation

In this chapter, I will evaluate the comparisons using their accuracy and time.

6.1 Results

6.1.1 Aggregated Statistics

Accuracy (Mean): 0.738 Accuracy (std): 0.0412 Precision (Mean): 0.734 Precision (std): 0.0444 Recall (Mean): 0.738 Recall (std): 0.0412 Time (s) (mean): 0.347 Time (s) (std): 0.264

The accuracy, precision and recall's mean and standard deviation are all quite similar, with recall being identical to accuracy in this case.

6.1.2 By Accuracy (All Tweets)

Table 1: By Accuracy

Name	**Accuracy**	**Precision**	**Recall**	**Time (s)**
Logistic Regression	0.801	0.781	0.801	0.2
Multilayer Perceptron	0.763	0.782	0.763	0.208
Random Forest	0.76	0.765	0.76	0.927
XGBoost	0.716	0.715	0.716	0.336
Naive Bayes	0.714	0.678	0.714	0.21
Stochastic Gradient Descent	0.674	0.681	0.674	0.203

Logistic Regression was the most accurate algorithm by a fair margin, and was significantly faster than the closest competition.

6.1.3 By Precision (All Tweets)

Table 2: By Precision

Name	**Accuracy**	**Precision**	**Recall**	**Time (s)**
Multilayer Perceptron	0.763	0.782	0.763	0.208
Logistic Regression	0.801	0.781	0.801	0.2
Random Forest	0.76	0.765	0.76	0.927
XGBoost	0.716	0.715	0.716	0.336
Stochastic Gradient Descent	0.674	0.681	0.674	0.203
Naive Bayes	0.714	0.678	0.714	0.21

Multilayer Perceptron was the most precise, but only by 0.001 more than Logistic Regression.

6.1.4 By Time (All Tweets)

Table 3: By Time

Name	Accuracy	Precision	Recall	Time (s)
Logistic Regression	0.801	0.781	0.801	0.2
Stochastic Gradient Descent	0.674	0.681	0.674	0.203
Multilayer Perceptron	0.763	0.782	0.763	0.208
Naive Bayes	0.714	0.678	0.714	0.21
XGBoost	0.716	0.715	0.716	0.336
Random Forest	0.76	0.765	0.76	0.927

Logistic Regression was the fastest, closely followed by SGD and Multilayer Perceptron.

6.1.5 Initial Graph (Score against Time)

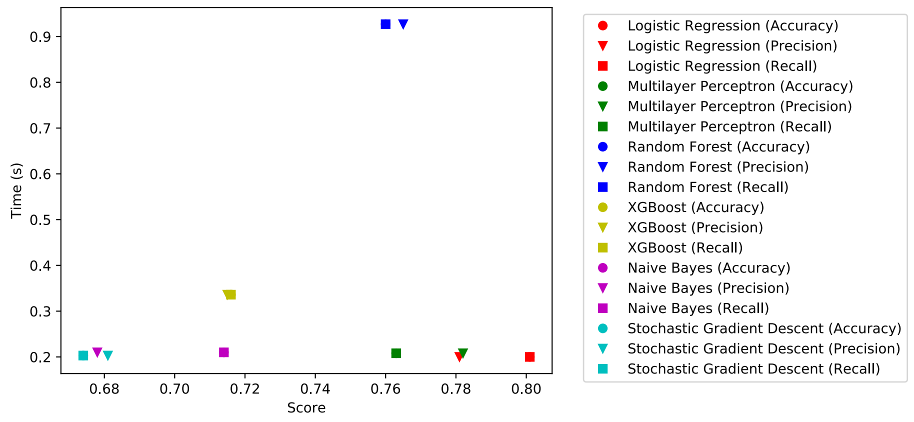


Figure 6: Initial Graph (Score against Time)

6.1.6 Findings

Generally, the fastest algorithms were the least accurate, and vice versa. Logistic Regression appears to be the exception as it was the 3rd fastest and the most accurate.

The discrepancy in time between certain algorithm evaluations was odd, so I decided to experiment further by seeing what happens when I train with varying data sizes.

6.1.7 5776 Tweets (1/8 of total)

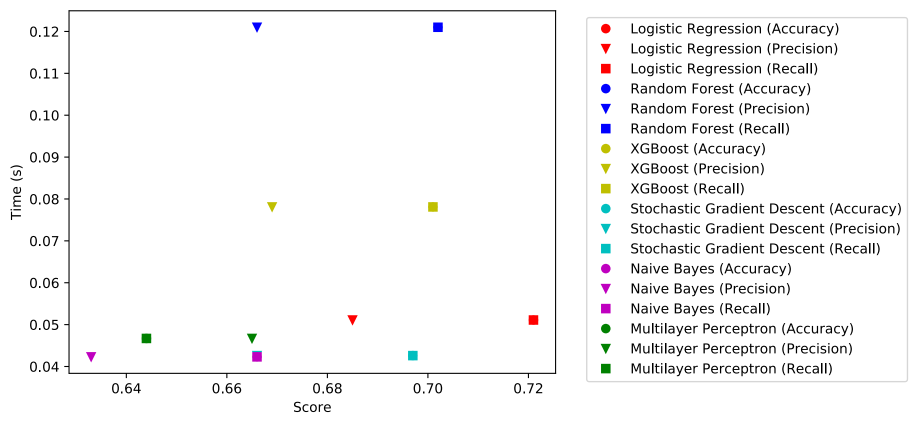


Figure 7: 5776 Tweets (1/8 of total)

6.1.8 11552 Tweets (1/4 of total)

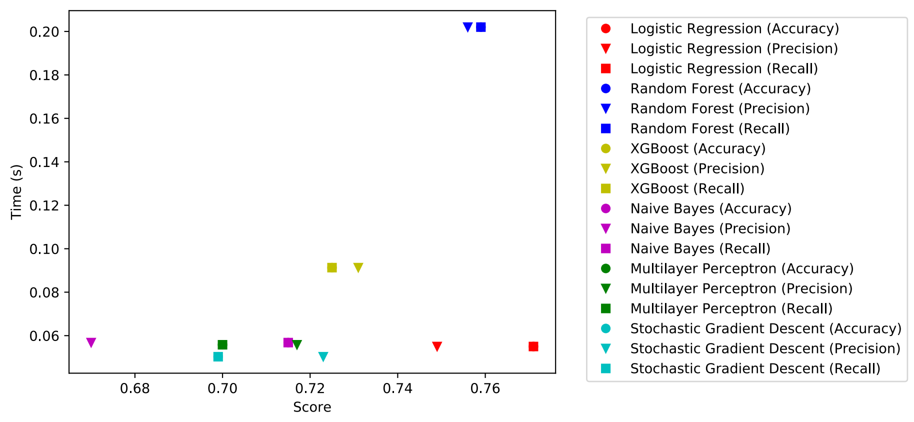


Figure 8: 11552 Tweets (1/4 of total)

6.1.9 17328 Tweets (3/8 of total)

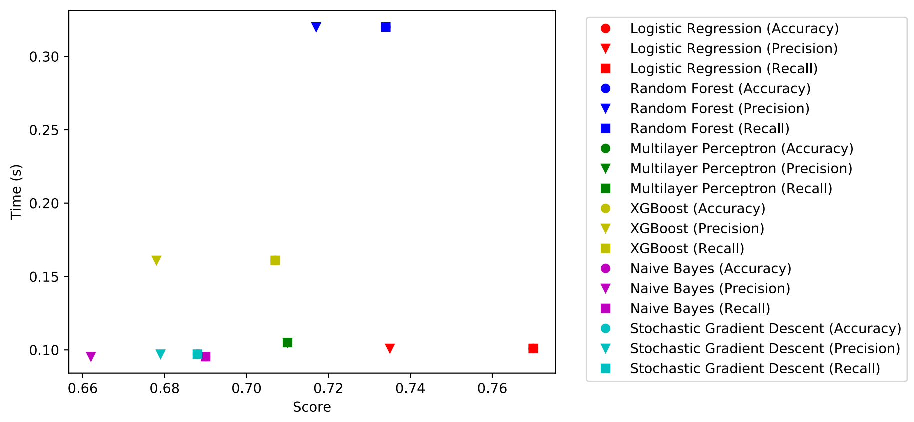


Figure 9: 17328 Tweets (3/8 of total)

6.1.10 23104 Tweets (1/2 of total)

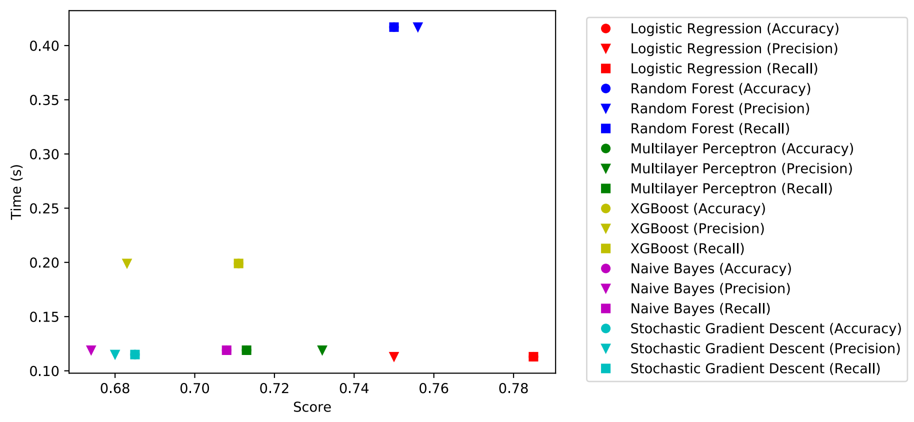


Figure 10: 23104 Tweets (1/2 of total)

6.1.11 28880 Tweets (5/8 of total)

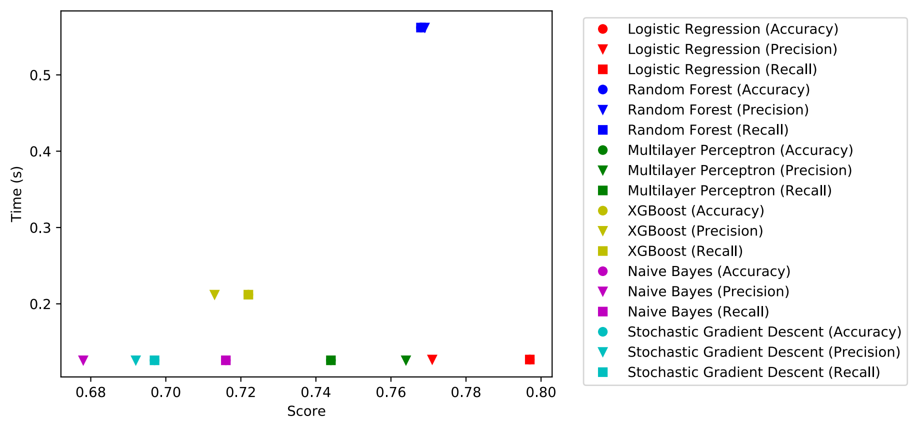


Figure 11: 28880 Tweets (5/8 of total)

6.1.12 34656 Tweets (3/4 of total)

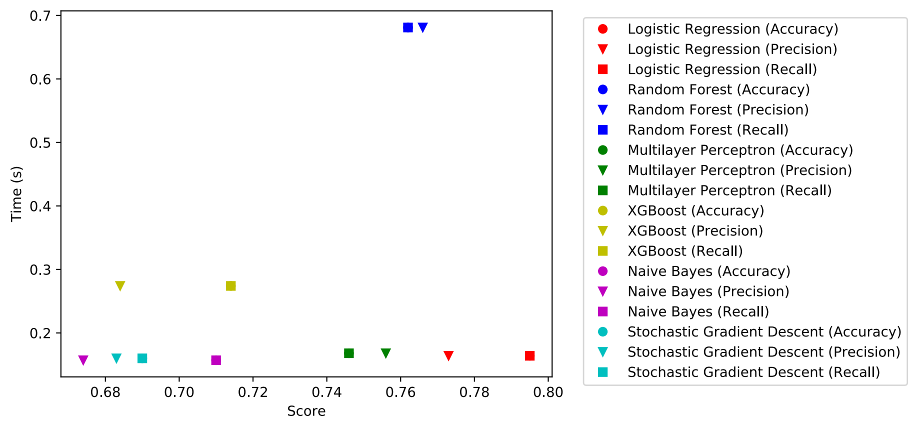


Figure 12: 34656 Tweets (3/4 of total)

6.1.13 40432 Tweets (7/8 of total)

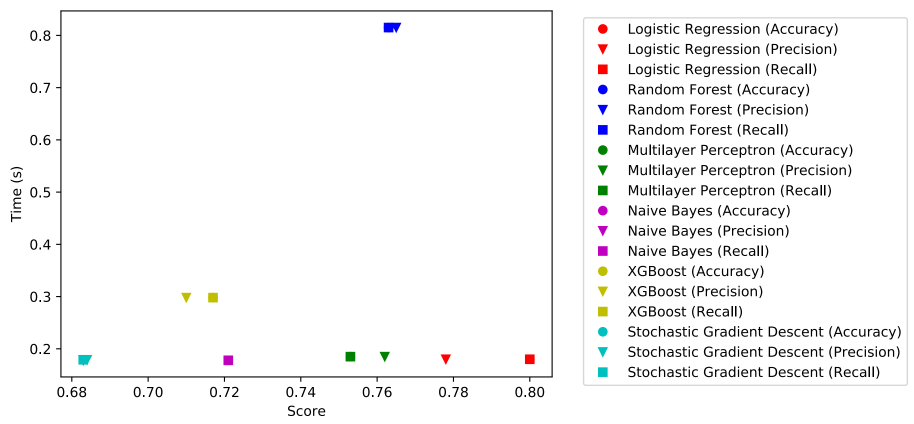


Figure 13: 40432 Tweets (7/8 of total)

6.1.14 46208 Tweets (All Tweets)

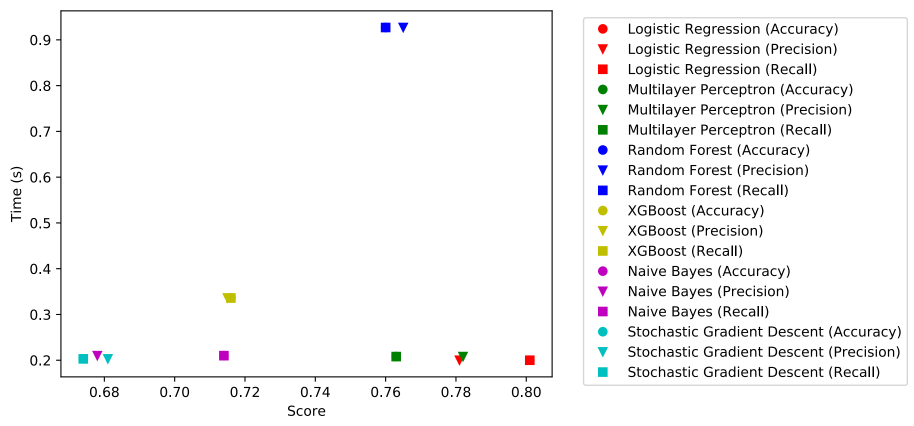


Figure 14: 46208 Tweets (All Tweets)

6.1.15 Results across size variations

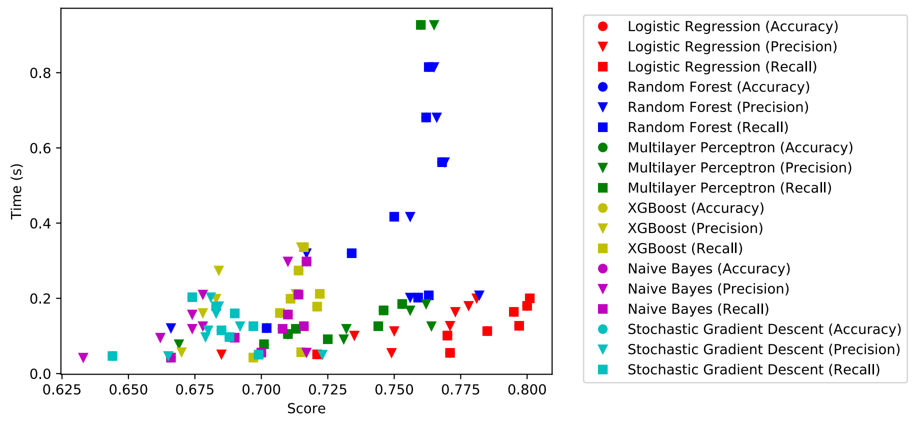


Figure 15: Combined Results

Overall, the relative positions of each algorithm with respect to each other stayed about the same, but there is some definite movement for some algorithms, meaning some algorithms perform better in comparison for smaller or larger datasets. It's clear that the biggest outlier by any measure is Random Forest due to its considerable time difference compared to other algorithms.

7 Conclusion

If I had more time, I would have tweaked all algorithms, using all available parameters and exhausting all possible combinations of parameters, to provide the highest accuracy for each. I think I got reasonably good scores for all algorithms, and I'm pretty happy with the structure and functionality of my code, in particular the pre-processing and setup for training, as well as the data handling parts. I would have also liked to have compared more algorithms, and find a clear pattern between which parameters increase precision or accuracy regardless of the algorithm, as well as improving my classifier to be able to find the sentiment of more complex sentences.

8 Bibliography

References

Al-Masri, Anas. *Creating The Twitter Sentiment Analysis Program in Python with Naive Bayes Classification.* 2019. URL: `https://towardsdatascience.`

com/creating-the-twitter-sentiment-analysis-program-in-python-with-naive-bayes-classification-672e5589a7ed.

API Reference¶. URL: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics.

Ckoepp. *ckoepp/TwitterSearch*. 2018. URL: https://github.com/ckoepp/TwitterSearch.

Cross-validation (statistics). 2020. URL: https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation.

Esuli, Andrea. *SentiWordNet*. URL: https://github.com/aesuli/SentiWordNet/blob/master/papers/LREC06.pdf.

Hall, Melissa. *WhatWouldTrumpTweet: Topic Clustering and Tweet Generation from Donald Trump's Tweets*. 2017. URL: https://medium.com/@mc7968/whatwouldtrumptweet-topic-clustering-and-tweet-generation-from-donald-trumps-tweets-b191fccaffb2.

Jsvine. *jsvine/markovify*. 2020. URL: https://github.com/jsvine/markovify.

Logistic regression. 2020. URL: https://en.wikipedia.org/wiki/Logistic_regression.

Multilayer perceptron. 2020. URL: https://en.wikipedia.org/wiki/Multilayer_perceptron.

Munir, Samira. *Basic Binary Sentiment Analysis using NLTK*. 2019. URL: https://towardsdatascience.com/basic-binary-sentiment-analysis-using-nltk-c94ba17ae386.

Nag, Avishek. *Text Classification by XGBoost & Others: A Case Study Using BBC News Articles*. 2019. URL: https://medium.com/towards-artificial-intelligence/text-classification-by-xgboost-others-a-case-study-using-bbc-news-articles-5d88e94a9f8.

Naive Bayes classifier. 2020. URL: https://en.wikipedia.org/wiki/Naive_Bayes_classifier.

Nishihara, Robert. *10x Faster Parallel Python Without Python Multiprocessing*. 2020. URL: https://towardsdatascience.com/10x-faster-parallel-python-without-python-multiprocessing-e5017c93cce1.

Putra, Dzaky Widya. *How to make your own sentiment analyzer using Python and Google's Natural Language API*. 2019. URL: https://www.freecodecamp.org/news/how-to-make-your-own-sentiment-analyzer-using-python-and-googles-natural-language-api-9e91e1c493e/.

Random forest. 2020. URL: https://en.wikipedia.org/wiki/Random_forest.

Ray-Project. *ray-project/ray*. 2020. URL: https://github.com/ray-project/ray.

Stochastic gradient descent. 2020. URL: https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

Synonym ring. 2017. URL: https://en.wikipedia.org/wiki/Synonym_ring.

Text Classification NLTK Tutorial. URL: https://pythonprogramming.net/text-classification-nltk-tutorial/.

Trump Twitter Archive. URL: http://www.trumptwitterarchive.com/archive.

Twitter Sentiment Analysis using Python. 2020. URL: https://www.geeksforgeeks.org/twitter-sentiment-analysis-using-python/.

WordNet. 2020. URL: https://en.wikipedia.org/wiki/WordNet.
XGBoost. 2020. URL: https://en.wikipedia.org/wiki/XGBoost.
XGBoost in Python. URL: https://www.datacamp.com/community/tutorials/xgboost-in-python.