# Trump Tweet Semantics Report

Luke Ireland

## Table of Contents



## 1. Introduction

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

Sentiment Analysis is classifying text into various classifications. In this case, 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.

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 paragraph.

Word Clouds provide visual representations 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 and 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 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.

## 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 and accuracy.

## 1.3. Objectives

1. Perform frequency distribution on variable length phrases.
2. Render word clouds of phrases.
3. Analyse whole tweets for sentiment.
4. Create unbiased classifer to initally label dataset
5. Find sentiment classifer algorithms
6. Compare classification algorithms to see which is the most accurate and the fastest.
7. Create tweets using data from Trump's Twitter account.

# 2. Literature Review

I decided to use Python as it's my strongest language, plus it's 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 output in JSON format, that you have to slowly copy and paste into a file and use programs to format the JSON into readable format.

I saw guides such as Basic Binary Sentiment Analysis using NLTK[3], Text Classification using NLTK[4] and Creating a Twitter Sentiment

[1] https://github.com/ckoepp/TwitterSearch

[2] http://www.trumptwitterarchive.com/archive

[3] https://towardsdatascience.com/basic-binary-sentiment-analysis-using-nltk-c94ba17ae386

[4] https://pythonprogramming.net/text-classification-nltk-tutorial/

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 can'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, classification, translation and, most importantly, sentiment analysis.

I needed more methods of sentiment analysis, so I decided on using Latent Sentiment Analysis(LSA), Random Forests (used as both part of LSA, and independently), XGBoost[8], Logistic Regression and Multilayer Perceptron models to compare to my Naive Bayes classifier.

Latent Sentiment Analysis(LSA) is one SA (Sentiment Analysis) technique that scores a corpus based on an assumption meaning similar meaning words will appear in similar pieces of text[9].

Naive Bayes classifiers are simple probabilistic classifers which applies 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 - document)[10].

Random Forest classifiers operate by constructing multiple decision trees at training time, and uses 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[11].

XGBoost classifier is a gradient boosting algorithm that uses proportional shrinking of leaf nodes, smart tree penalization and differen-

[5] https://towardsdatascience.com/creating-the-twitter-sentiment-analysis-program-in-python-with-naive-bayes-classification-672e5589a7ed

[6] https://www.freecodecamp.org/news/how-to-make-your-own-sentiment-analyzer-using-python-and-googles-natural-language-api-9e91e1c493e/

[7] https://www.geeksforgeeks.org/twitter-sentiment-analysis-using-python/

[8] https://medium.com/@mc7968/whatwouldtrumptweet-topic-clustering-and-tweet-generation-from-donald-trumps-tweets-b191fccaffb2

[9] https://en.wikipedia.org/wiki/Latent_semantic_analysis

[10] https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[11] https://en.wikipedia.org/wiki/Random_forest

tiation 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 abitrary differentiable loss function, often softmax for multiclass classifiers[12].

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, and uses a supervised learning technique called backpropagation, similar to the least mean squares algorithm, for training. Learning is performed by changing the connection weights between the layers, based on the amount of error between the prediction and actual class[13].

Logisitic 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 estamating 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: scores(X,k) = βk*X, where X is a vector of the words that make up the tweet, and β is a vector of weights corresponding to outcome (class) k. This score vector is then used by the softmax function to calculate a probability of the tweet belonging to that sentiment[14].

Softmax is simply: e^scores/sum(scores), where scores a vector where each element corresponds to a word and sentiment prediction.

Stochastic Gradient Descent is an iterative method for optimizing an objective function with suitable smoothness properties. The objective function to be minimised is:

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

Figure 1: "SGD Equation"

where the parameter w (sentiment polarity) is to be estimated. Each

[12]https://en.wikipedia.org/wiki/XGBoost
[13]https://en.wikipedia.org/wiki/Multilayer_perceptron
[14]https://en.wikipedia.org/wiki/Logistic_regression

summand function Qi is typically associated with the i-th observation in the data set.[15]

Due to their simple effectiveness, I assume 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 on algorithm comparison[16], I found that creating a tf-idf transformer to use on the initial bag of words model massively boosts accuracy.

When implementing my models, I discovered that the fairest, most reproducible method of comparison was using Scikit Learn's Pipelines[17], and 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 pre-processing. I had to cut Latent Sentiment Analysis as it didn't fit this streamlined format, due to the way it retrospectively trains itself. Scikit Learn is another Python NLP library that builds on the NLP available in NLTK.

For tweet generation, I used Markovify[18], which I found from this[19] article attempting the same thing. 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. Data Pre-processing

## 3.1 Dataset

The dataset 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.

[15] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[16] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[17] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[18] https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation
[19] https://medium.com/@mc7968/whatwouldtrumptweet-topic-clustering-and-tweet-generation-from-donald-trumps-tweets-b191fccaffb2

Here is a word cloud I created using the Python library wordcloud.

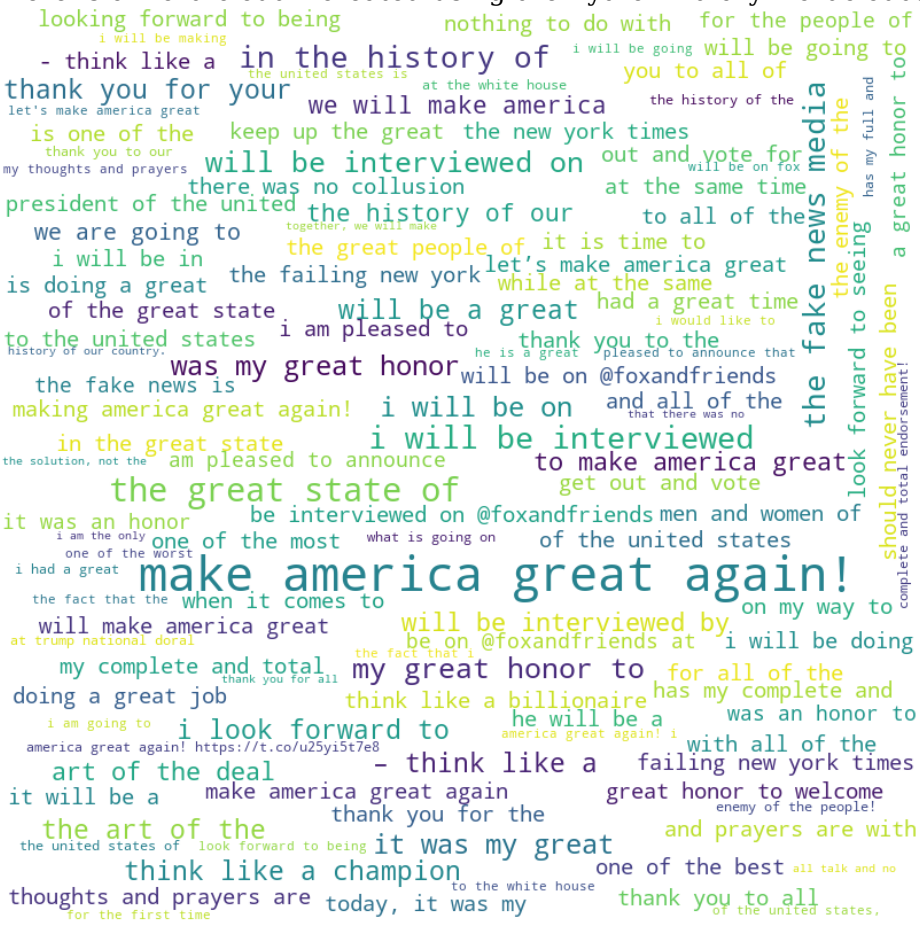


*Figure 2* - WordCloud of phrases of length 4.

I created my own class using Markovify[20], that would force the generator to generate tweets in Trump-like prose containing a given input word.

```
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 *obviou
RT @America1stTiger: Just in case you forgot...I have @JoeBiden here admitting to a reque
If the V.P. of the American people care about Hunter Biden brought to the euphoria of gett
China wants Sleepy Joe Biden, on the other hand, take out millions of American…
```

[20]https://en.wikipedia.org/wiki/WordNet

## 3.2 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 which required further processing.

My classifier tokenised the cleaned tweets, then applied lemmatisation on the tokens, which is an advanced form of stemming. 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. 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.[21]

Synsets are grouped data elements considered semantically equivalent for information retrieval purposes.[22]

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.[^23]

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.

This resulted in rather narrow ranges for each sentiment class.

- Less than -0.125 = Negative

[21] https://en.wikipedia.org/wiki/Synonym_ring
[22] https://github.com/aesuli/SentiWordNet/blob/master/papers/LREC06.pdf

- 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.

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 `new score = (raw_score - mean)/standard deviation`, where raw score is the original score calculated by my classifier. All scores are calculated by my classifier, then a mean and standard deviation are collected to calculate the z-score for each tweet.

# 4. Comparison Implementation

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. 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 Scikit-Learn, meaning I had to drop some algorithms I was considering for comparison during my research stage.

I wanted to use k-fold cross validation on the data, which involves splitting the data into k equal sized partitions, where one of the partitions is used as validation data for testing 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[23].

K-fold cross validation is built into `cross_val_score`, which is given a value of 10 for the k-value parameter.

Each algorithm had to be trained through all 46208 tweets, so I implemented multithreading to better use of system resources when

[23] https://medium.com/towards-artificial-intelligence/text-classification-by-xgboost-others-a-case-study-using-bbc-news-articles-5d88e94a9f8

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 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.

# 5. Evaluation

## Results

### By Accuracy

| Name | Accuracy | Time (s) |
|---|---|---|
| Logistic Regression | 0.800922 | 37.240018 |
| Random Forest | 0.759241 | 842.787394 |
| Multilayer Perceptron | 0.757835 | 804.682227 |
| XGBoost | 0.715872 | 405.671944 |
| Naive Bayes | 0.713579 | 32.224075 |
| Stochastic Gradient Descent | 0.674234 | 32.702595 |

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

### By Time

| Name | Accuracy | Time (s) |
|---|---|---|
| Naive Bayes | 0.713579 | 32.224075 |
| Stochastic Gradient Descent | 0.674234 | 32.702595 |
| Logistic Regression | 0.800922 | 37.240018 |
| XGBoost | 0.715872 | 405.671944 |
| Multilayer Perceptron | 0.757835 | 804.682227 |
| Random Forest | 0.759241 | 842.787394 |

Naive Bayes was the fastest, but was very closely followed by the less accurate SGD, and the more accurate Logistic Regression.

### 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.

## 9. Conclusion

## 10. References