South African National Address: A Natural Language Proccessing & Neural Networks Application

Predicting President using SONA Speeches

Pavan Singh

11 January 2023

Contents

Introduction

We shall apply several key learnings from text mining and neural network applications to create our very own neural network to predict the president of South Africa given a sentence of text - more details shall be further outlined below.

Problem and Task

We are to build a neural network that, given a sentence of text, predicts which president was the source of that sentence. We then, shall use this configured neural network and assess the out-of-sample performance for this classifier.

The problem task is essentially one of classification. Classification is one of the two major problem spaces in supervised learning. It essentially entails an approach for predicting qualitative responses. Here, we regard the action of predicting a qualitative response for an observation (sentence of text) as classifying that observation (sentence) since it involves assigning the observation (sentence) to a category, or class (a president). So, as mentioned the context of this project is around the implementation of classification algorithms, specifically (but not limited to) artificial neural networks - the details for which shall be discussed later on. Using our neural network results we can compare and benchmark the accuracy and classifications with other widely-used classifiers; namely, multinomial logistic regression, decision trees, random forests and gradient boosted machines.

Project Overview

We begin the analysis by quickly loading up the packages, data and providing a brief description of it. Essentially this is the set-up phase, before we get to playing around with the data. We then shall be cleaning and pre-processing the data, before conducting a brief data exploration. Here, our goal is to familiarise ourselves with the data at hand, so as to help consolidate our approach in later stages of analysis and to better inform the splitting of the data into training and testing sets. We also briefly look at the aggregated sentiment of each president across their addresses given. We then look to turn the data into a bag-of-words model. At this stage we look at further processing the text to look for removing stop works or generating TF-IDF weighted counts. Next, we introduce neural networks. We start by giving some brief, high level theoretical background to our application, before discussing our modelling approach. We also look to discuss the data imbalance problem faced, training and testing splits, hyper-parameter tuning and methods of assessing performance. Finally, we get to applying and building our neural network structures - the results of which are briefly discussed. To provide greater context of the results, we also look at several different other classifiers, as such, the sections which follow the neural networks look at the results of conducting several other classification algorithms. In the penultimate section of this project, we compare the results of the optimal neural network classifier and other classification algorithms generated. Finally, we conclude this project with a brief discussion highlighting the topics covered in this project and a brief summary of the analysis and results, as well as some limitations encountered and perhaps, further improvements that can be made to the analysis.

Setup and Data

In this stage of our report, we load up all the packages we shall be using for this project, as well as provide a brief data description. The main package that is used would be the Tidyverse library, which in fact is a collection of R packages designed for data manipulation/wrangling. We also make use of Keras and TensorFlow extensively to build, compile and fit our neural network models.

The data is made available in the zip file `sona-addresses-1994-2022.zip`. It contains text from State of the Nation Address speeches from 1994 to 2022. The State of the Nation Address of the President of South Africa (SONA) is an annual event in which the President of South Africa reports on the status of the nation, normally to the resumption of a joint sitting of Parliament. The data is sources from the SONA website. We note that we don't simply have 28 text files (speeches) as in the years that elections took place, a SONA happens twice, once before and again after the election. As such we have 35 documents (speeches) in total. To read in the several data `.txt` files to R we used the `readtext()` function from the *readtext* package. We then converted the read in data to a tibble, so as to prepare for cleaning and tidying of the data using *dplyr* functionality in R.

Data Cleaning and Pre-Processing

At this stage in the project we look to start some initial data cleaning and pre-processing with respects to turning our text data into tidy data.

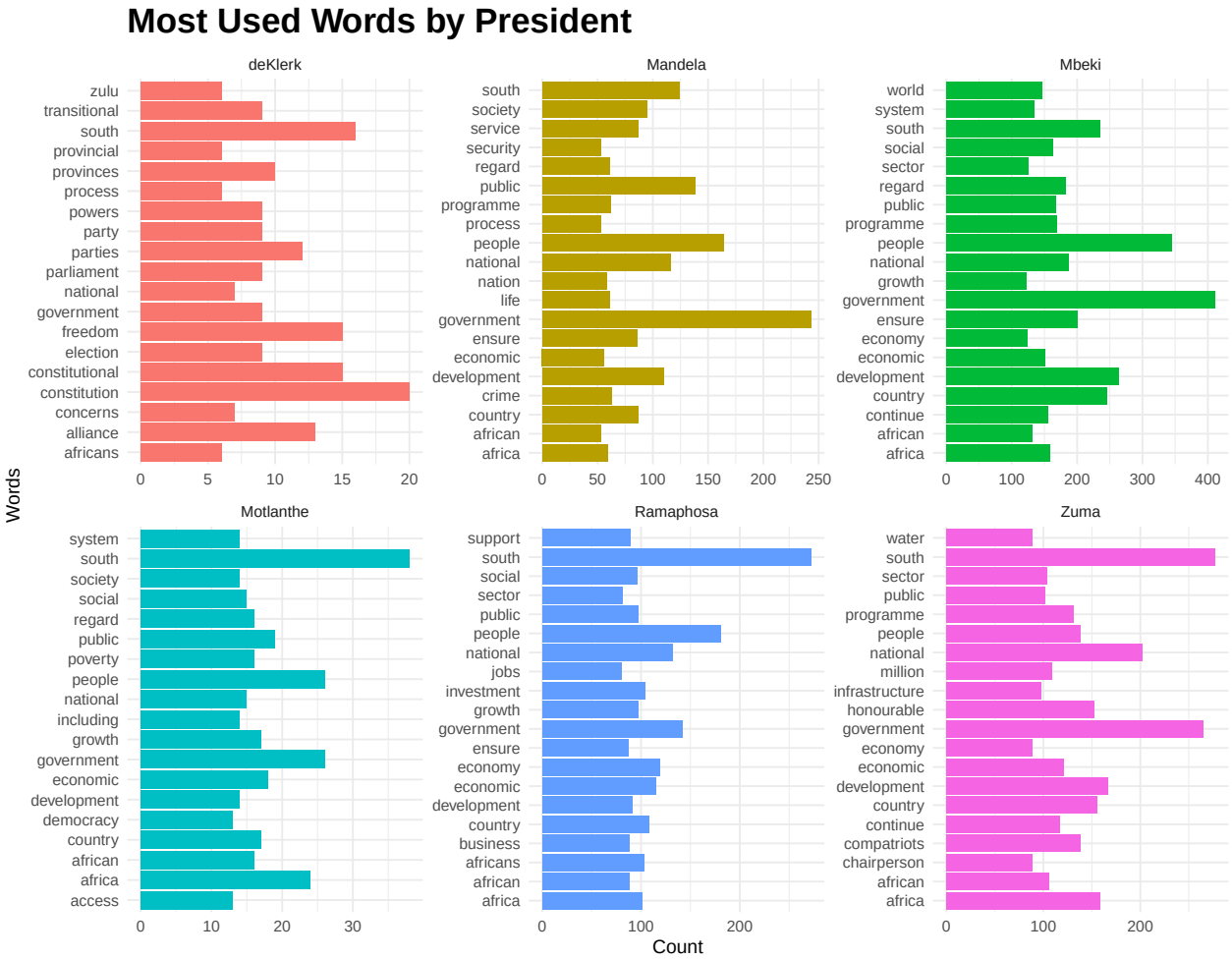
We start by cleaning the date variable. Turning the date into a format that will be easier to work with (if we want to). We use the `parse_datetime()` function from the *lubridate* package. We also add a new column, called *president* using `mutate()`, which essentially just indicates the president who gave each respective speech. This will become more important later on when we generate tokens (extract individual sentences from the text). We also convert all the text in the tibble to lower case. Next we look at turning the text into tidy format. Tidy data is essentially just a way of consistently organizing your data that often makes subsequent analysis easier, particularly since we shall be using tidyverse packages for cleaning and wrangling. Whilst turning our data into tidy format, we simultaneously conduct tokenisation. A token is a whatever unit of text is meaningful for your analysis: it could be a word, a word pair, a sentence, a paragraph, or even chapter of text. For our problem, we shall tokenize by sentences - as this is our desired input for our classifiers. We do this now, as to get our data in tidy format requires us to decide what the level of the analysis is going to be - what the "token" is. We use the `unnest_tokens()` function to perform tokenization by splitting text up into the required tokens (sentences), and creating a new data frame with one token per row i.e. tidy text data. Whilst tokenising our text, we also conduct some cleaning of the text. Specifically, we look to remove some of the dates that occur at the start of the speeches, as well as the backslashes and escape sequences within the text data. Once we have processed this data (cleaning the text, tokenising the data, and transforming to tidy format), we also ensure to remove stop words from our sentences (tokens). Note, we also created a separate tibble, which consists of text that has been tokenized by words, this is simply for exploratory purposes that we can use to better understand the language of the presidents. As such, we shall use both tibbles containing sentence tokens and word tokens in the exploratory section next.

Data Exploration

With respect to our exploratory analysis, we split this up into two parts as we see this is the best way of scrutinising the data whilst providing context of the textual data. We shall look at the presidents and high level patterns that emerge between themselves and the language they use (words and sentences) in the first part. In the second part of this exploratory analysis, we shall briefly look at the sentiment of the presidents using the words from their speeches.

Words and Sentences

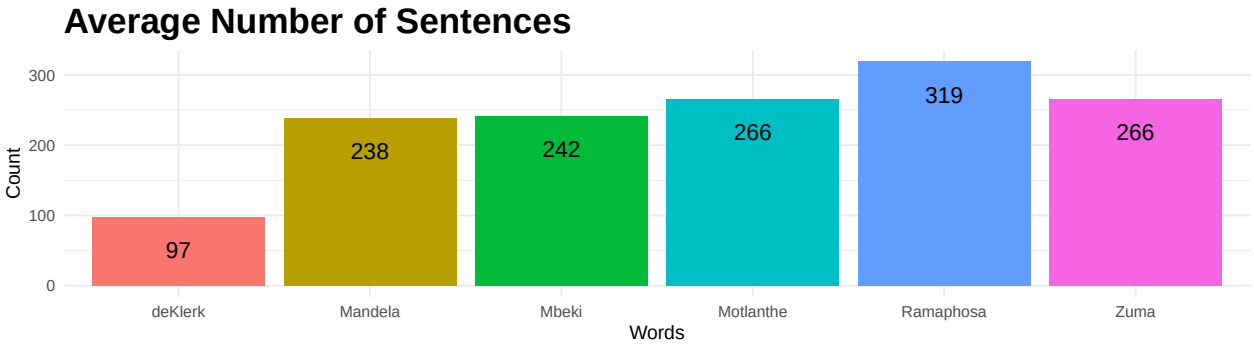
We start by looking at the most common words used by each president. The result of which is shown in the plot below.



Certain words certainly stand out for each president. For De Klerk, we only have one speech from him which was back in 1994, but from that speech we can identify that the words "constitution", "south", "freedom" and "constitutional" were some of the most frequently used words in his speech - understandably so given the political climate of his speech.

Across all presidents, the word "south" is often used, which makes sense. The impact of this on a proposed classifier might suggest that it will not hold a strong influence on predicting the president. Similarly, we can see that across the six different president's their top 20 most common words seem relatively similar. There is not any clear word used frequently (from the top 20 words) by a president, that distinguishes them from one another. By using TF-IDF, we can clearly observe this . This shall be touched on again soon.

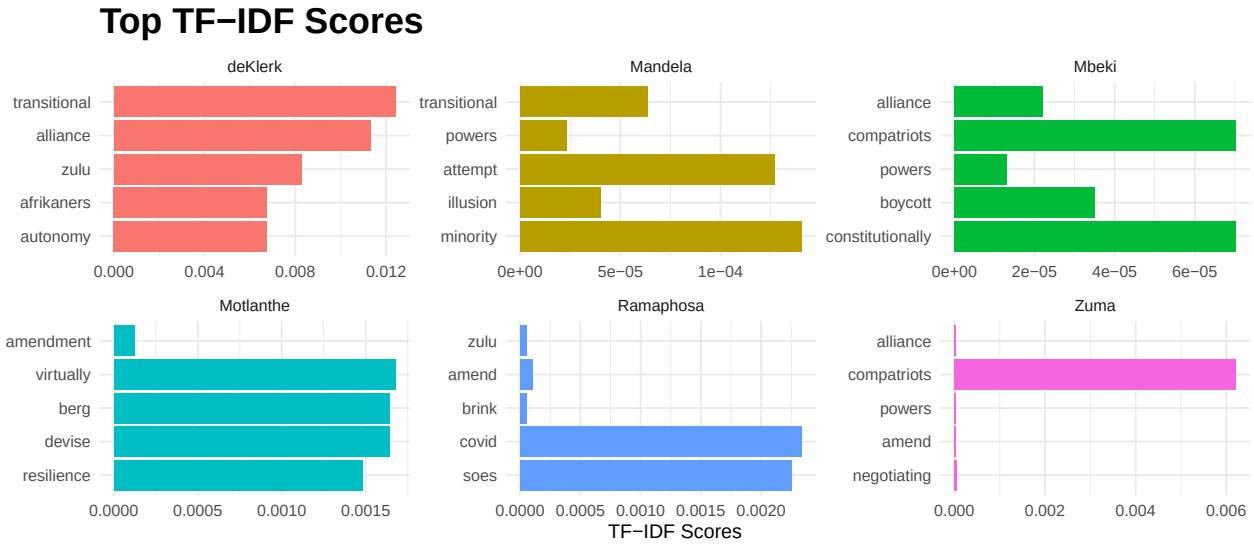
We now look at the average number of sentences given by each president. Although not particularly accurate, it does give us an idea of how long on average each speech is for each president.



Average number of sentences for each president

We see that on average, Ramaphosa has the most sentences at 319. This is closely followed by Motlanthae and Zuma, who have, interestingly, the same average number of sentences per speech. In general, besides De Klerk, who only has an average of 97, the presidents appear to have a similar length speeches. Note, that since De Klerk only had 1 speech, which happened to be dramatically much shorter than the other presidents, the impact of this is that we will end up having a rather imbalanced data set for our classification problem. This problem shall be discussed in greater detail later on.

As mentioned previously, we can explore using the method called term frequency – inverse document frequency (TF-IDF) to circumvent seeing words that offer no analytical value - i.e., they appear often across all presidents. We will be able to identify which words have a high analytical value. These words would or may appear for one president and not so for the others, for example. The plot below shows this.



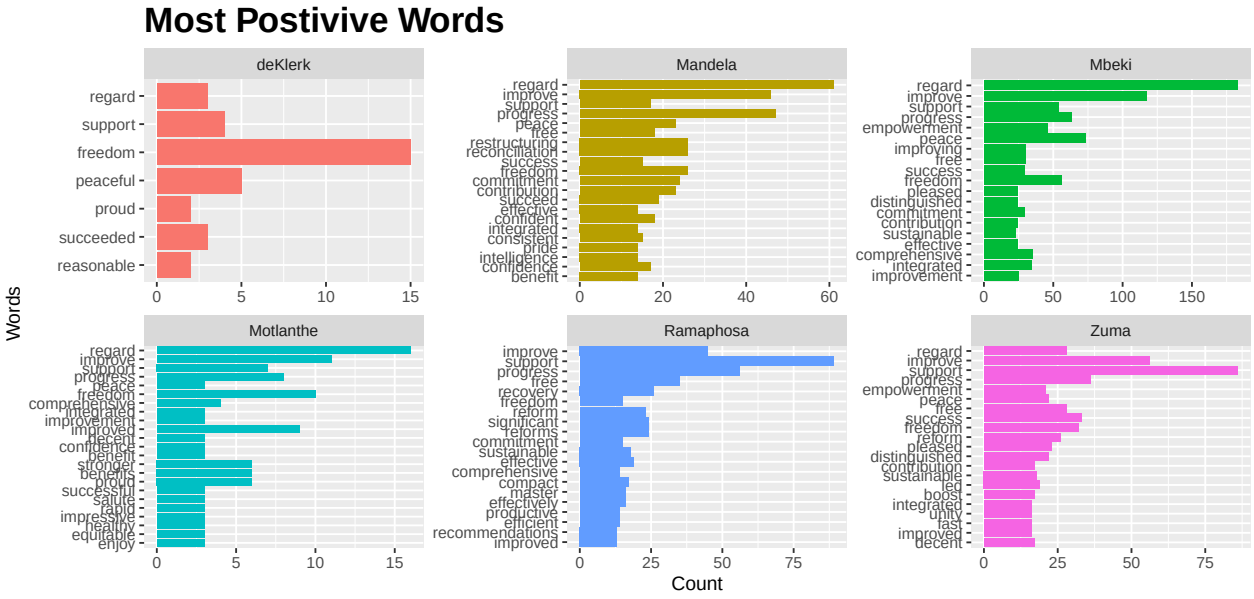
Top TF−IDF scores for words for each president

What this plot is telling us is **which are the main words that separate one president from the other**. A clear and easy example to understand this is the word "covid" used by Rampahosa in some of his speeches. This word holds a high TF-IDF score as it is not used by any other president in their speeches. Similarly, we can say the same for other words which hold high TF-IDF scores for other presidents, like the word "compatriots" mentioned by Zuma in his speeches. De Klerk, interestingly, in his one speech, uses words which clearly distinguishes him from the other presidents. In general we can see that each president, has several words which they tend to use a lot more than others. This could become important in our classification models.

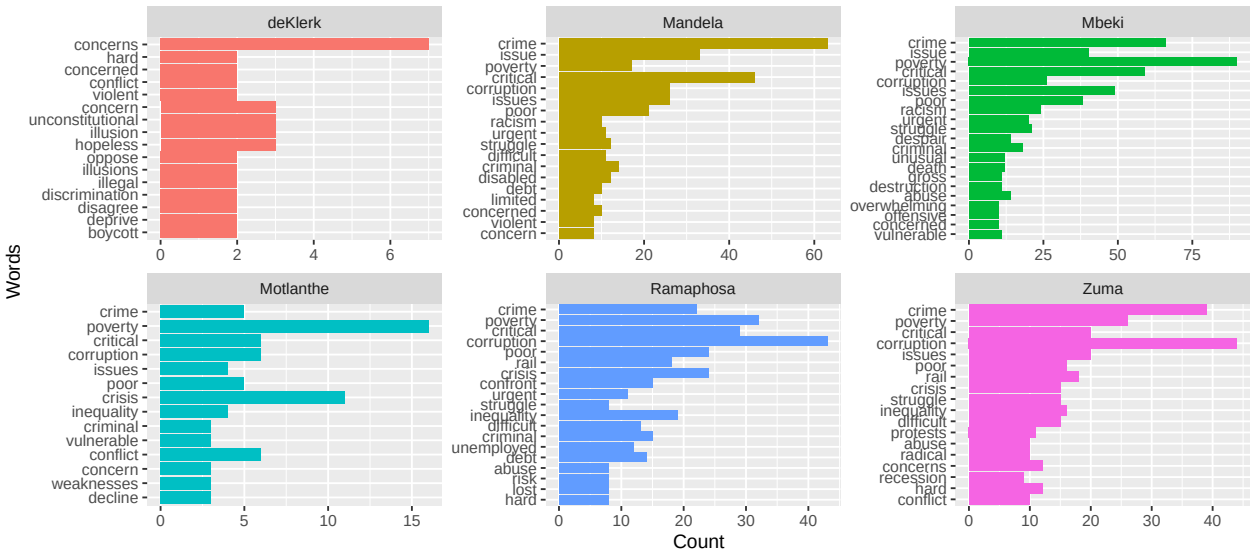
Sentiment Analysis

We briefly summarise the sentiment of the speeches, more specifically the choice of words used in the speeches, by each president. This can better help in understanding the differences between presidents and can aid in modelling our data - however this is likely to be beyond the scope of the workings for this task.

Sentiment analysis is the study of the emotional content of a body of text. When assessing the sentiment or emotional content of individual words, we make use of existing sentiment dictionaries ("lexicons") that have already done this using some kind of manual classification. For this subsection, we make use of the *bing* lexicon. This lexicon contains a list of words are labelled as "positive" or "negative". We load the lexicon into R and save the dictionary to a variable which we can merge with our words tibble (contains word tokens for presidents). As such, we can easily identify which are the most positive and negative words used by each president. This is given below in the two figures.



Count of most positive words used by each president



Count of most negative words used by each president

Interestingly, as we found in our prior analysis into looking at the most common words used by each president, we see that a lot of the negative and positive sentiment is from words used by most of presidents. For example, the word "poverty" appears for all but one (De Klerk) presidents, and holds a relatively high negative sentiment. Similarly so, for the words "crime" and "corruption". The results are similar, when looking at the positive sentiment. The words "progress" and "improve" are common amongst all 5 presidents, De Klerk being the one exemption again. In fact, this appears to be a common theme here - De Klerk having very different words carrying his sentiment value.

Finally, we can also look at the trends in sentiment over time. The result of which is shown in the appendix. Generally, we can see that there is a larger positive sentiment in almost every speech and this seems to be apparent for every year between 1994 to 2022. The difference between positive and negative sentiment also appears to be rather stable.

Bag-of-Words Model

In this section we begin to turn our data we have tokenized into a bag of words format; essentially this is a representation of text that describes the occurrence of words within a document. Bag-of-words models are often used in NLP settings, where a text is represented as the bag of its words, disregarding grammar and even word order but keeping multiplicity. Frequency counts are provided of the words used in the document. In our case, the frequency counts of the words in each sentence are given. We can use these frequency counts for clustering or for predictive modelling, which we shall be doing. Essentially the idea is that we can use the word frequencies as features to build a model that, on the basis of frequencies of different words, predicts which president a sentence is from.

Since we are interested in sentences, what we do is we attach a sentence index to each row of data in about tibble, which represents a sentence from a president. We then can take this tibble and tokenize it further and get words from the sentences. We then can choose to sort the data by sentence index. From here to reach a bag-of-words format, we need to find all unique words used in all of the sentences, and then count how many times each of these words are used in each sentence. Note that, for this problem, we shall look at the frequency of the 1000 most popular words.

In reaching our bag-of-words format, we make use of the function `pivot_wider()`, to have that each sentence will be its own row, and each word its own column. As such, we are moving from a tidy format to an "untidy" format. The resulting bag of words format for the data is thus achieved. An extract of this data, is shown below.

Table 1: Extract of Bag-of-Words Model Format

index	pres_name	act	africa	constitution	parliament	republic
1	deKlerk	1	1	1	1	1
2	deKlerk	0	0	0	0	0
3	deKlerk	0	0	0	0	0
4	deKlerk	0	0	0	0	0
5	deKlerk	0	0	1	0	0
6	deKlerk	0	0	0	0	0

Each sentence is represented by a row and each word is represented by a column. The frequency of that word in each sentence is given as the elements. The first column (*index*) indicates the sentence index. The second column (*pres_name*) shows the president from which the sentence is taken from. The results bag-of-words model has dimensions 8719×1017. Besides the data described above, we also generate a bag-of-words model using TF-IDF method. An extract of this data is given in the appendix.

Imbalanced Data

A key component and problem for this project is the issue of imbalanced data. Imbalanced data typically refers to a problem with classification problems where the classes are not represented equally. For example, given our scenario, we have the following number of sentences for each president:

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
Count	94	1621	2369	258	1851	2526

De Klerk has 94 sentences, whilst Mandela and Ramaphosa have over 1000, and Mbeki and Zuma have well over 2000 sentences. Clearly there is a vast imbalance across all classes. This is not a unique problem. Most classification data sets do not have exactly equal number of instances in each class, however usually a small difference often does not matter. However, given the drastic imbalance we have, it is likely that we will have some difficulty generating good classifiers. The problem of imbalanced data also lends itself to asking us how should we assess the performance of our classifiers.

The *accuracy* paradox is the name for the situation where the accuracy measures may say have excellent accuracy, but this accuracy measure is only reflecting the underlying class distribution. The actual reason that we may be achieving this great accuracy score on an imbalanced data set, is because the models look at the data and decide that the best thing to do is to predict the majority class and achieve high accuracy. Given this, it is important for us to not look at accuracy alone when assessing our classifier algorithms but also other metrics, which we shall discuss further later on.

Back to the imbalanced data problem. To overcome this we can look at under-sampling and over-sampling methods. Under-sampling involves removing samples from over-represented classes. Over-sampling involves adding more samples from under-represented classes. For our problem we look at three different sampling methods. The first of which, is under-sampling. Here we simply randomly select sentences from the majority classes equal to the number in the minority class. Although simple, it can be a problem since we are greatly diminishing the data size, and dropping the number of sentences for each president to 94.

We also look at applying Synthetic Minority Over-sampling Technique (SMOTE). Without going into great detail, SMOTE is an oversampling method and works by creating synthetic samples from the minority class instead of creating copies. To apply SMOTE, we make use of the `smote()` function in the *performanceEstimation* package in R.

The final method we look at is Random Over-Sampling Examples (ROSE), which is another over-sampling technique. It is primarily used in binary classification problems, however we can tweak it to generate a new more balanced data set for our multi-class problem. We make use of the *ROSE* package in R, and the function `ovun.sample()` to conduct the oversampling and under-sampling in one go. We specify the size of the resulting data set we want. We choose arbitrarily to have around 500 observations for each president (sentences for each president).

Neural Networks

In machine learning, often we are simply trying to predict for a target or class. Loosely speaking, **the goal of machine learning is to create a function to map features to some class** - for classification tasks. The workflow for neural networks, and machine learning in general, boils down to a straightforward formula. Essentially, once we have our data we will need to split it — into training, testing and validation data sets. We will then train some model — a neural network, for example — on the training data, validate it and tweak parameters on validation data and then evaluate this model on the testing data. This is ultimately, the crux of what we shall do.

The inherent power of neural networks comes from their ability to learn the representation in your training data and how best to relate it to the output variable you want to predict. In this sense, neural networks learn the mapping.

Background

We shall be concerned with only multi-layer perceptron's or artificial neural networks for the scope of this project. A perceptron is a single neuron model that is a precursor to larger neural networks. A neuron is a simple computational unit that has weighted input signals and produces an output signal using an activation function. Weights on the inputs are very much like the coefficients used in a regression equation; in the sense that these are optimised to achieve the best mapping. The weighted inputs are summed and passed through an activation function. An activation function is a simple mapping of summed weighted input to the output of the neuron. Traditionally, non-linear activation functions are used. This allows the network to combine the inputs in more complex ways and, in turn, provide a richer capability in the functions they can model. Popular activations are sigmoid, softmax and ReLu — each having their own respective properties which make them better or more appropriate than others in a given problem.

Neurons are arranged into networks of neurons. A row of neurons is called a layer, and one network can have multiple layers. The input layer takes an input from your data set and provides it to network. Layers after the input layer are called hidden layers because they are not directly exposed to the input. The final hidden layer is called the output layer, and it is responsible for outputting a value or vector of values that correspond to the format required for the problem. The choice of activation function in the output layer is strongly constrained by the type of problem that you are modelling.

Modelling, Hyperparameters and Network Architecture

For our problem scenario, we are wanting to input a sentence and get a prediction as to who the president is (who said that sentence). We shall use our processed bag-of-words formatted data as input. The architecture of our neural network is of major interest in this section. We shall start with a very basic neural network, with one input layer, one hidden layer and one output layer. The hidden layer shall have a Relu activation function and the output layer will use a softmax activation with 6 units since we have 6 classes (presidents). This will be our baseline neural network for which we shall get initial results for. From here we can look to explore tuning our model. That is, we will also conduct some form of tuning for the hyperparameters to best optimise the neural network.

Hyperparameters are set by the user before training and are independent of the training process. These hyperparameters affect the performance the network, hence they need to be optimised. Hyperparameter tuning involves choosing the optimal set of hyperparameters for a learning algorithm. A grid search is an exhaustive search through a pre-specified subset of the hyperparameter space of a learning algorithm. Essentially, grid search is a technique that generates evenly spaced values for each hyperparameters and then uses cross validation to find the optimum values. We shall implore this method of optimising our neural network. The *caret* R package was designed to make finding optimal parameters for an algorithm. It provides a grid search method for searching parameters, combined with various methods for estimating the performance of a given model. As such, we use the `trainControl()` and `expand.grid()` functions in the *caret* package in R. Essentially, it will trial all combinations and locate the one combination that gives the best results. There are several methods we can specify in the `train()` function for Keras neural networks. We choose to use:

- `method = 'mlpKerasDropout'` for a Multilayer Perceptron Network with Dropout. There are quite a few tuning parameters here, like number of hidden units (`size`), dropout rate (`dropout`), batch size (`batch_size`) and more. We can even look at different activation functions. We decided to use an automatic grid, that is, allowing the system to do it automatically - we set the `tuneLength` to indicate the number of different values to try for each algorithm parameter.

The tuning from the caret package provided suboptimal results and poor validation accuracy. As such, we used some of the suggestions from this grid search using automatically generated pre-specified valued for hyperparameters and manually adjusted the neural network models. That is, we ended up tweaking the parameters of our neural networks manually using the results observed from our validation analysis when training our model and the suggestions from the grid search. Specifically, we were interested in adjusting the number of hidden layers, the number of units in hidden layers and the optimiser. A summary description of the initial neural network architecture and the optimised tuned neural networks are given below.

Baseline Neural Network: Simple neural network with 1024 units in input layer, with ReLu activation function, 512 units in hidden layer with ReLu activation function and 6 units in output layer with softmax function.

Tuned Neural Network 1: From our baseline network, we add an additional hidden layer with 512 units and a Relu activation function. We also add another dropout layer between the two hidden layers, with a rate of 0.1.

Tuned Neural Network 2: Same as Tuned Neural Network 1, however we look to change the optimisers. We try RMSprop optimiser.

Tuned Neural Network 3: Same as Tuned Neural Network 2, however we remove a hidden layer. Now we have only one hidden layer with 812 units and Relu activation function.

Tuned Neural Network 4: We take our Tuned Neural Network 3 and scale the data for training and testing.

To start constructing a model, we first initialize a sequential model with the help of the `keras_model_sequential()` function. Then, we are ready to start modelling. We add layers, specify units in each layer, and activation functions according to the above specification.

We shall apply our baseline neural network to all five different data sets we have available to us. This is to assess the impact of class imbalance and see which classifier best performs with which data set. The data sets are:

- original bag of words data
- TF-IDF bag of words data
- under sampled bag of words data
- SMOTE bag of words data
- ROSE bag of words data

Note, that to assist our neural network with the imbalanced original data, we made use of *class weights* provided by the Keras API. Essentially what this does is modify the current training algorithm to take into account the skewed distribution of the classes. This is achieved by giving different weights to both the majority and minority classes. The difference in weights will influence the classification of the classes during the training phase. Ultimately, what this means is that we are penalizing the misclassification made for the minority class by setting a higher class weight and at the same time reducing weight for the majority class. The formula used to calculate the weight is:

$$w_j = \frac{n\_samples}{(n\_classes \times n\_samples_j)}$$

where w_j is the weight for each class (j signifies the class), $n\_samples$ is the total number of samples or rows in the dataset, $n\_classes$ is the total number of unique classes in the target and $n\_samples_j$ is the total number of rows of the respective class. Note, that we only explore using class weights for the original bag of words data and the TF-IDF bag of words data, since these are imbalanced.

We shall also look at whether additional pre-processing is required, such as whether scaling the data is beneficial or not. Since it is a classification problem and we have multi-class scenario, we also use one hot encoding. That is, we transform our target attribute from a vector that contains values for each class value to a matrix with a Boolean for each class value and whether or not a given instance has that class value or not. This is done by using the `to_categorical()` function in Keras for one-hot encoding.

When deciding on our choice of optimisers, we initially used the Adam optimiser in our baseline neural network as mentioned. For our tuned neural network we experimented with other optimiser's like RMSprop as well as Adam with different learning rates. The choice of loss function was categorical cross entropy and accuracy was the metric of choice. A batch size of 5 was applied and 30 epochs were chosen, given that no real in accuracy was noticed with greater iterations.

Measure of Performance

As mentioned, due to the accuracy paradox, it would be unwise to simply use accuracy as a measure for the classification performance of our neural network (or any classifier for that matter). We measure the effectiveness of our model by viewing the confusion matrix. From the confusion matrix we can derive several metrics which better help reveal the classification performance of our algorithm. More on these metrics shall be discussed in the results section.

Training and Test Splits

Our first step in the modelling of the neural networks is to split our data into training and testing sets. We will still use our training set to build models and save the testing set for a final estimate of how our model will perform on new data. The split chosen was a 70/30 split, where 70% of the observations were used to train our model and 30 were held out and kept for testing. From the 70%, we used 20% for validation. Tweaking of parameters was done using the validation accuracy and loss observed in training our neural networks.

The results from training one of our neural networks is given in the appendix. We can easily identify that we are most likely overfitting, since the training data accuracy keeps improving while the validation data accuracy appears to not improve and gets worse. The model has is simply memorising the data instead of learning from it.

Results

As mentioned we are not interested in accuracy alone to measure the classification performance of our models. We use a confusion matrix to provide additional information, that we can dissect and investigate the actual classification from each model. A confusion matrix simply provides a breakdown of predictions into a table showing correct predictions (the diagonal) and the types of incorrect predictions made (what classes incorrect predictions were assigned). An important metric which we can retrieve is the recall or sensitivity - which is a measure of a classifiers completeness. That is, recall measures the model's ability to detect positive samples. So in the context of the problem, it is our model's ability to correctly predict a president from a sentence input when it is in fact that president as the source. We shall also be looking at Kappa (or Cohen's Kappa), which is defined as the classification accuracy normalized by the imbalance of the classes in the data.

Accuracy is simply calculated by looking at the proportion of true positive and true negative over the sum of the matrix. I.e., accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$, where:

- TP is true positive, cases model correctly predicts the positive class
- TN is true negative, cases model correctly predicts the negative class
- FP is false positive, cases model incorrectly predicts the positive class
- FN is false negative, cases model incorrectly predicts the negative class

The results from all our neural networks configured is given in the table below.

Classifier (Model)	Data	Accuracy	NIR	P-Value	Kappa Score
Base Neural Network	Original Data (Non-Scaled)	0.317	0.287	0.001	0.077
Base Neural Network	Original Data (Scaled)	0.010	0.287	1.000	0.000
Base Neural Network	TF-IDF (Non-Scaled)	0.257	0.287	1.000	0.103
Base Neural Network	ROSE Data (Non-Scaled)	0.512	0.183	$<$ 2e-16	0.413
Base Neural Network	SMOTE (Non-Scaled)	0.056	0.281	1.000	-0.031
Base Neural Network	Undersampled (Non-Scaled)	0.261	0.188	0.013	0.120
Tuned Neural Network 1	ROSE Data (Non-Scaled)	0.489	0.183	$<$ 2e-16	0.384
Tuned Neural Network 2	ROSE Data (Non-Scaled)	0.497	0.183	$<$ 2e-16	0.395
Tuned Neural Network 3	ROSE Data (Non-Scaled)	0.508	0.183	$<$ 2e-16	0.408
Tuned Neural Network 4	ROSE Data (Scaled)	0.518	0.183	$<$ 2e-16	0.419

- No Information Rate (NIR): is the naive classifier which needs to be exceeded in order to prove that model we created is significant. We calculate accuracy and then compare it with naive classifier. That is, we can use this metric to run a hypothesis test to evaluate whether the classification model created is significant.

Looking at the table we see that all the models using the ROSE data are significant, having very low p-values; that is, these model are significant at the 5% significant level. They also tend to have higher Kappa scores. We have already touched on the Kappa score, which essentially just measures the agreement between classification and truth values. A kappa value of 1 represents perfect agreement, while a value of 0 represents no agreement. Generally, the literature suggests that a kappa of less than 0.35 is considered poor (a Kappa of 0 means there is no difference between the observers and chance alone). Kappa values of 0.35 to 0.75 are considered moderate to good and a kappa of > 0.75 represents excellent agreement.

The Tuned Neural Network 4 with scaled data, one hidden layer with 812 units and ReLu activation, using an RMSprop optimiser appeared to be the best performing classification neural network. It achieved an accuracy of 51.8% and was significant. The mode's confusion matrix given below. Note that, each row in a confusion matrix represents an actual target, while each column represents a predicted target.

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
De Klerk	134	4	5	2	4	4
Mandela	0	81	36	6	32	28
Mbeki	0	49	93	10	56	39
Motlanthe	4	18	12	137	17	16
Ramaphosa	0	14	25	6	40	65
Zuma	0	0	0	0	0	0

We can immediately identify the fatal issue with the network classification performance by examining the confusion matrix. Although it is able to perform reasonably well given the context of the problem, it suffers

drastically with respect to recall for a certain class. This problem actually appeared to be prevalent for almost every neural network we modelled. However, when using the ROSE generated data, we are able to improve recall scores for the other classes. Here, we see that class "Zuma" suffers dramatically, having no predicted results from the network. Although, the network was able to achieve a recall score of 0.971 for "De Klerk" class. This implies that from all the positive classes, we predicted almost all of them correctly for that class. The recall or sensitivity scores are given below from the Tuned Neural Network 4.

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
Sensitivity	0.971	0.488	0.544	0.851	0.268	0

Other Classification Methods

For multi-level classification, there are many machine learning and *traditional* statistical algorithms which can model the data. We explore the following alternatives:

- Multinomial Logistic Regression (MLR)
- Classification Decision Tree (CT)
- Random Forest (RF)
- Gradient Boosted Machines (GBM)

We give the initial accuracy results from each algorithm in each respective subsection. We also provide confusion matrices and specificities for the best performing models for each algorithm in the appendix. We also summarise all the results of the most optimal algorithms at the end together with the neural network results with various different metrics included; since, as already established, we cannot look at accuracy as a useful measure alone for classification.

Three of the four classification algorithms we look at fall under tree based methods. As such, these methods tend to perform well on unprocessed data (i.e. without normalizing, centering, scaling features), so we do not change the data in terms of conducting additional scaling etc. We shall test all four of these classification algorithms using three different data sets; the original bag-of-words data, the TF-IDF bag-of-words data, and the oversampled ROSE generated data.

Similar to our approach for neural networks, we split the data into training and testing sets - using a 70/30 split. We train the algorithms using the training data and test them on the unseen, testing data. No hyperparameter tuning is conducted for any of the algorithms, as we are only interested in benchmarking our neural network results.

Multinomial Logistic Regression

Multinomial Logistic Regression (MLR) is a form of linear regression analysis conducted when the dependent variable is nominal (no intrinsic ordering) with more than two levels. It can be considered simply as an extension of the standard Logistic Regression framework, where we simply explain the relationship between one dependent nominal variable and one or more continuous-level independent variables - logistic regression simply analyses dichotomous (binary) dependents. Here, the choice of using MLR is simply for predictive purposes.

To conduct multinomial logistic regression, we train the model by making use of the `mutlinom()` function, which is part of the *nnet* package in R.

Results

The results from our multinomial logistic regression analysis is given in the table below, for the three classifiers trained on the three different training data sets. The accuracy and other metrics mentioned in the prior neural networks section are also given.

Classifier (Model)	Data	Accuracy	NIR	P-Value	Kappa Score
Multinomial Logistic Regression	Original Data	0.454	0.293	$<$ 2e-16	0.290
Multinomial Logistic Regression	TF-IDF Data	0.440	0.286	$<$ 2e-16	0.273
Multinomial Logistic Regression	ROSE Data	0.573	0.203	$<$ 2e-16	0.487

Each of the three classifiers trained on different data sets appear to be significant, looking at the p-values. The multinomial logistic regression model using the ROSE data appeared to be the best model generating the highest accuracy, as well having a much greater Kappa score, of 0.487. The confusion matrix and recall results shown in the appendix for this model using ROSE data, also shows that it achieves relatively good sensitivities for each class, particularly for De Klerk class.

Decision Trees

Decision trees are a simple yet effective tool for classification and regression tasks, capable of fitting complex datasets. Decision trees are also the fundamental components to random forests and gradient boosted machines, both of which we shall look at in the sections to come. One of the many benefits of decision trees is that they require very little data preparation. In particular, they don't require feature scaling or centering.

We make use of the `rpart()` function to fit our decision tree. This function is from the *rpart* package in R. We specify the formula of the decision tree, in our case that is the class *president*. We specify `method='class'` which invokes a classification tree problem. By default, `rpart()` function uses the Gini impurity measure to split the note.

Results

The results for our decision tree classifiers using the three different data sets are shown in table below.

Classifier (Model)	Data	Accuracy	NIR	P-Value	Kappa Score
Classification (Decision) Tree	Original Data	0.323	0.875	1.000	0.053
Classification (Decision) Tree	TF-IDF Data	0.323	0.875	1.000	0.053
Classification (Decision) Tree	ROSE Data	0.273	0.837	1.000	0.127

Interestingly, the ROSE data did not improve the accuracy of the results. However, we notice that the decision tree using the ROSE data had the highest Kappa score. In comparison, the decision trees using the TF-IDF data and original data, produced identical results. When we inspect the confusion matrices of the decision tree using the original data, the results are extremely poor. Our model only produces predictions for either class "Mbeki" or "Zuma", and neglects the other classes. As such the sensitivities for the neglected classes are reported as NA.

Random Forest

A natural downfall of decision trees, is that they suffer from high variance. We can overcome this by introducing bagging. Bagging, is a general-purpose procedure for reducing the variance of a statistical learning method. The key idea behind bagging is: averaging a set of observations reduces variance. A random forest is simply a slight extension to bagging, as such it is also a form of ensemble learning, where we have many decisions trees (a forest) whose results are aggregated into one final result. It uses bagging, however includes feature randomness (at each split) when building each individual tree to try to create an uncorrelated forest of trees. This small tweak, provides improved performance (generally) from random forests over bagged trees. So the key difference, is that in building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available predictors. Again, no tuning of the random forests were done, even though it is generally advisable.

Results

After generating these random forest classifiers without any tuning we were able to generate some results which are illustrated below.

Classifier (Model)	Data	Accuracy	NIR	P-Value	Kappa Score
Random Forest	Original Data	0.427	0.287	$<$ 2.2e-16	0.211
Random Forest	TF-IDF Data	0.428	0.287	$<$ 2.2e-16	0.220
Random Forest	ROSE Data	0.582	0.175	$<$ 2.2e-16	0.427

The results appear to be quite strong. The random forest with ROSE data achieved a high accuracy (relatively) of 0.582 and a high kappa score of 0.427. This kappa score is large in comparison to the other random forest results using the original data and TF-IDF data. Notably, all models were significant. Inspecting the confusion matrix from the random forest with ROSE data appears promising. We get high recalls for "De Klerk" class (0.958), as well as "Motlanthe" (0.757).

Gradient Boosted Trees

Gradient Boosted Machines (GBMs) are a popular machine learning algorithm that has seen coverage and application across many domains. They are often associated with decision trees, and hence random forests. Whereas random forests build an ensemble of deep independent trees, GBMs build an ensemble of shallow and weak successive trees with each tree learning and improving on the previous.

One of the reasons why GBMs are so popular is because they are very flexible - they can optimize on different loss functions and provides several hyperparameter tuning options that make the function fit very flexible. Moreover, no pre-processing required. A downside to GBMs, is that they can be computationally expensive, requiring a lot of trees. Moreover due to the flexibility, the grid search and parameter tuning stage is time consuming and can be computationally taxing.

Fundamentally, boosting is a method for improving the prediction performance from a decision tree. The method works by sequentially applying a classification algorithm to re-weighted versions of the training data, and taking a weighted majority vote of the classifiers generated. The decision trees are grown sequentially using information from previous trees. The idea is to fit the current tree to residuals of the previous tree rather than the response Y.

We do not do any hyperparameter tuning and let the default values for the hyperparameters be used. We make use of the *gbm* package in R. We train our model using the `gbm()` function, with the default values for the hyperparameters.

Results

The table below illustrates the results from running our gradient boosted machine classifiers with the three different data sets.

Classifier (Model)	Data	Accuracy	NIR	P-Value	Kappa Score
Gradient Boosted Machine	Original Data	0.432	0.349	< 2.2e-16	0.241
Gradient Boosted Machine	TF-IDF Data	0.429	0.331	< 2.2e-16	0.240
Gradient Boosted Machine	ROSE Data	0.436	0.194	< 2.2e-16	0.324

The results with respect to accuracy are very close across all three data sets. However, when inspecting the Kappa score, the distinction is clearer. The ROSE data produced a better trained GBM classifier, with a marginally better accuracy score, but a substantially larger kappa score. Again we find that all the models were significant. With respect to recall, we find that the sensitivities between "Mandela", "Mbeki", "Ramaphosa" and "Zuma" to be all very similar. Again "De Klerk" and "Motlanthe" seem to have higher recall scores.

Overall Results and Comparison

Now we can fully evaluate the performance of our neural network by comparing it to the best performing classifiers we have generated. The best performing neural network was trained on the ROSE generated data – Tuned Neural Network 4. Similarly, we found that the results of most of our classifiers appeared to be better when trained on the more balanced dataset – this in line with the literature. The table below illustrates some classification metrics we have been using in evaluating the classifiers, for each of the best classifiers generated for each algorithm.

Classifier (Model)	Data	Accuracy	NIR	P-Value	Kappa Score
Tuned Neural Network 4	ROSE Data (Scaled)	0.518	0.183	<2e-16	0.419
Multinomial Logistic Regression	ROSE Data	0.573	0.203	<2e-16	0.487
Classification (Decision) Tree	Original Data	0.323	0.875	1.000	0.053
Random Forest	ROSE Data	0.582	0.175	<2e-16	0.497
Gradient Boosted Machine	ROSE Data	0.436	0.194	<2e-16	0.324

The results were closer than expected. The Tuned Neural Network 4 had the third best accuracy score across all the classifiers generated; similarly, it had the third best Kappa score as well. The best performing model was the Random Forest trained on the ROSE data. It achieved a 0.582 accuracy score and had the highest kappa of 0.497. The model achieved high recall scores for "De Klerk" and "Motlanthe", however this appeared to be the theme across all the classifiers. The Tuned Neural Network 4 also achieved high recalls for "De Klerk" and "Mothlanthe" classes (not as high as the random forest classifier), but it also achieved higher recall scores for classes "Mbeki" and "Mandela". A large flaw of the neural network models was the fact that it attained very poor recall scores for "Zuma" class, achieving a recall score 0 for the best

model (Tuned Neural Network 4). This problem was not present when inspecting the results from the other classification algorithms.

Computationally, the random forest although yielded the best results, also took the longest time to train. The neural networks on the other hand were relatively quick to train on the ROSE data. A notable mention should be made for the multinomial logistic regression model, which achieved both a high accuracy and kappa score, as well as relatively high recall scores for all classes.

The only classification algorithm that could not generate a significant model was the decision tree, which performed woefully in general across all three of the data sets.

Discussion and Conclusion

We were tasked with constructing a neural network that given a sentence of text, could predict which president was the source of that sentence. We went about this task by cleaning the data and getting it into tidy format. From here, we briefly explored it. We then set about tokenising the data and getting it into a bag-of-words model format, which we used as the input for our neural networks. We configured both simple and relatively more complex neural networks to try best build the optimal classifier. In an attempt to offset the data imbalance problem, we used ROSE to create a balanced data set. The best performing classification neural network was the Tuned Neural Network 4, which consisted of only one hidden layer with 812 units and Relu activation function, trained on the ROSE data and using a RMSprop optimiser. The model achieved over 50% accuracy and had very high recall scores for certain classes, however failed to correctly predict any positive cases (0 recall) for one class ("Zuma"). We compared this configured neural network model to other classifiers using several classification metrics aside from accuracy which yielded more satisfactory results, having no hyperparameter tuning done for these other classifiers as well.

The data used in the project was text from SONA speeches from 1994 to 2022. From this we created a bag of words model. We identified that the data was highly imbalanced. Ultimately it seemed that most classification algorithms performed better with the balanced ROSE data set. This is in line with the literature, which suggests that classification algorithms perform better with more balanced data - particularly machine learning algorithms.

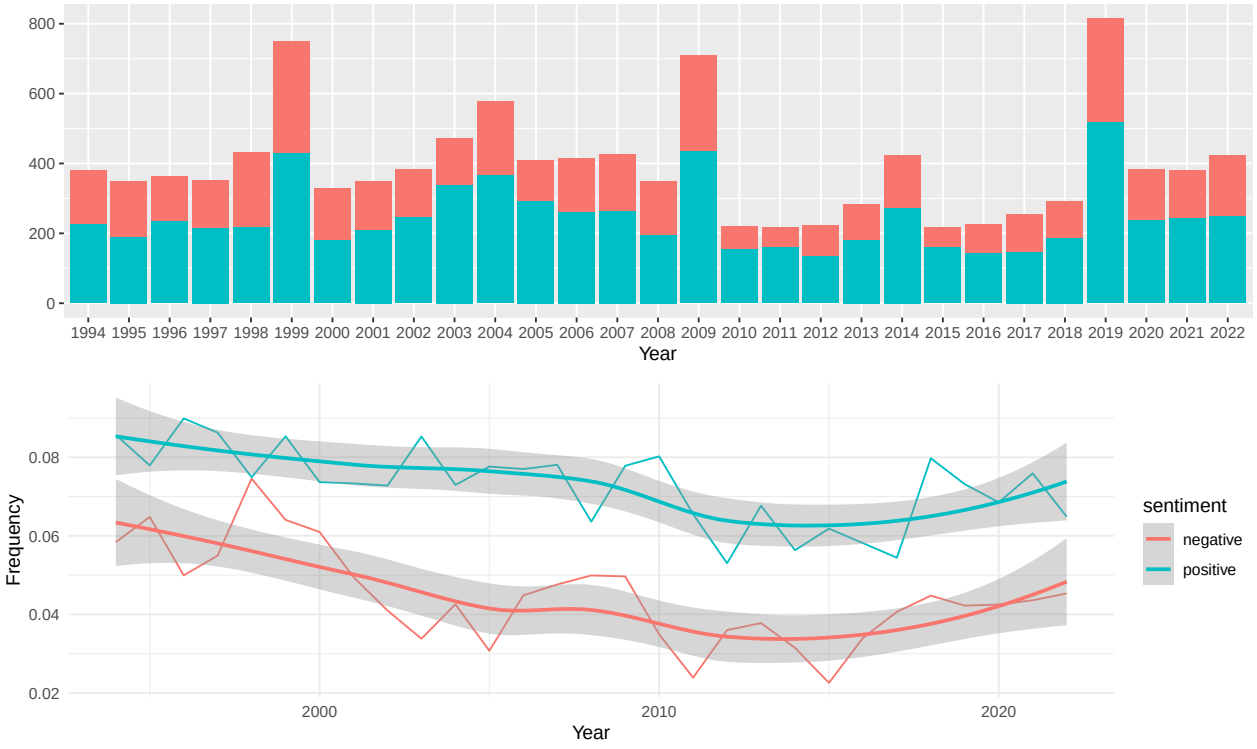
For further improvement, when constructing our bag of words model we only looks at the 1000 most popular words in sentences. We can further improve the model, by looking at a greater number of words, as such have a probably greater number of features to help our classifiers. However, this would come at a computational cost, particularly for algorithms such as the Random Forest and Gradient Boosted Machine, for example.

Appendix

Table 11: Extract of Bag-of-Words Model Format

index	pres_name	act	africa	constitution	parliament	republic
1	deKlerk	0.6273041	0.414807	0.6326956	0.5667592	0.7186012
2	deKlerk	0.0000000	0.000000	0.0000000	0.0000000	0.0000000
3	deKlerk	0.0000000	0.000000	0.0000000	0.0000000	0.0000000
4	deKlerk	0.0000000	0.000000	0.0000000	0.0000000	0.0000000
5	deKlerk	0.0000000	0.000000	1.1072172	0.0000000	0.0000000
6	deKlerk	0.0000000	0.000000	0.0000000	0.0000000	0.0000000

Here we count the number of positive, negative and neutral words used each year and plot these. Because the neutral words dominate, its difficult to see any trends with them included. We therefore remove the neutral words before plotting. We also look at plotting the proportion of all words in a speech used in that year that were positive or negative. The plot on the bottom shows the raw proportions as well as smoothed versions of these.



Looking at the plot on the top, it seems to be relatively stable throughout, with positive sentiment being generally greater than negative over the years. This is confirmed by looking at the plot on the bottom.

Results for Multinomial Logistic Regression on ROSE Data

Confusion Matrix

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
De Klerk	144	0	0	0	0	0
Mandela	2	60	40	11	18	16
Mbeki	1	27	73	16	15	27
Motlanthe	1	3	8	120	0	16
Ramaphosa	0	21	28	13	54	36
Zuma	4	17	25	24	18	68

Sensitivity

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
Sensitivity	0.947	0.469	0.420	0.652	0.514	0.417

Results for Decision Tree on Original Data

Confusion Matrix

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
De Klerk	0	0	1	0	0	28
Mandela	0	0	63	0	0	424
Mbeki	0	0	136	0	0	575
Motlanthe	0	0	6	0	0	72
Ramaphosa	0	0	55	0	0	501
Zuma	0	0	56	0	0	702

Sensitivity

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
Sensitivity	NA	NA	0.429	NA	NA	0.308

Results for Random Forest on ROSE Data

Confusion Matrix

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
De Klerk	138	28	26	13	26	22
Mandela	2	47	11	2	8	15
Mbeki	0	20	52	8	19	14
Motlanthe	0	13	32	112	14	13
Ramaphosa	0	17	18	3	72	34
Zuma	4	24	20	10	13	58

Sensitivity

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
Sensitivity	0.958	0.312	0.327	0.757	0.460	0.372

Results for Gradient Boosted Machine on ROSE Data

Confusion Matrix

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
De Klerk	117	1	14	2	3	7
Mandela	18	44	25	15	24	23
Mbeki	11	24	45	18	31	30
Motlanthe	7	10	21	72	24	14
Ramaphosa	15	17	14	16	62	28
Zuma	8	26	19	17	30	56

Sensitivity

	De Klerk	Mandela	Mbeki	Motlanthe	Ramaphosa	Zuma
Sensitivity	0.665	0.361	0.326	0.514	0.356	0.354

Training Performance of Neural Network