

Scratch Buffer 05 PCA and MDS Visualisation

Ryan Greenup

April 26, 2020

Contents

| | |
|---|----------|
| Visualisation and MDS | 1 |
| Load Packages | 1 |
| Get Tweets | 2 |
| .1 Create a Corpus | 2 |
| .2 Clean the Corpus | 3 |
| .3 Use TF-IDF Weighting | 3 |
| .4 Plot and Post the Words | 4 |
| Use PCA to Visualise the Tweets | 4 |
| .1 Bi Plot | 6 |
| .2 Scree Plot | 6 |
| .3 Plot the First two PCA's | 7 |
| Use Multi-Dimensional Scaling to Visualise the tweets | 10 |
| .1 Finding the Distance Matrix of Tweets | 11 |
| .2 Binary Distance | 11 |
| .3 Cosine Distance | 13 |

Visualisation and MDS

Load Packages

First Load all the packages:

```
# Load Packages -----  
  
if(require("pacman")){  
  library(pacman)  
}else{  
  install.packages("pacman")  
  library(pacman)  
}  
pacman::p_load(xts, sp, gstat, ggplot2, rmarkdown, reshape2, ggmap,  
               parallel, dplyr, plotly, tidyverse, reticulate, UsingR, Rmpfr,
```

```

        swirl, corrplot, gridExtra, mise, latex2exp, tree, rpart, lattice,
        coin, primes, epitools, maps, clipr, ggmap, twitterR, ROAuth,
        tm, rtweet, base64enc, httpuv, SnowballC, RColorBrewer, wordcloud, ggwordcloud,
mise()

```

The tokens are located in KeePass:

```

# Set up Tokens =====

options(RCurlOptions = list(verbose = FALSE, capath = system.file("CurlSSL", "cacert.pem")))

setup_twitter_oauth(
  consumer_key = "dE7...",
  consumer_secret = "7B...",
  access_token = "1240821...",
  access_secret = "HLBWzHce...")

# rtweet =====
tk <- rtweet::create_token(
  app = "SWA",
  consumer_key = "dE7H...",
  consumer_secret = "7By...",
  access_token = "1240...",
  access_secret = "HLBWzH...",
  set_renv = FALSE)

```

Get Tweets

In order to practice visualisation using PCA, it may be helpful to have data that appears to cluster, so political tweets would be ideal, we can use the handles for the party leaders:

```

# Political Tweets -----
n <- 100

# twitterR::userTimeline("billshortemp", n = n)
altw <- rtweet::get_timeline("AlboMP", n = n, token = tk)
smtw <- rtweet::get_timeline("ScottMorrisonMP", n = n, token = tk)
abtw <- rtweet::get_timeline("AdamBandt", n = n, token = tk)

```

Create a Corpus

The tm package requires tweets to be stored in a corpus object, there are three steps to creating this: 1. Extract the Text 2. use `tm::VectorSource(text)` to create a tm source file. 3. use `tm::VCorpus()` to create a *Volatile Corpus* 1. Volatile meaning stored in memory.

```

# Create a Corpus =====
tweets <- c(altw$text, smtw$text, abtw$text) # Get all the text
tweet_source <- tm::VectorSource(tweets) # Create the

```

```
tweet_corpus <- tm::VCorpus(x = tweet_source)
tweet_corpus[[1]]$content
strwrap(tweet_corpus[[1]])
```

Clean the Corpus

This is the same method as shown in [Prac 04](#) I've simply copied the function over, in [DataCamp Work](#) the qdap package was used as well.

```
# Create a WordCloud #####

## Clean the Corpus
clean_corp <- function(corpus) {
  corpus <- tm_map(corpus, FUN = removeNumbers)
  corpus <- tm_map(corpus, FUN = removePunctuation)
  corpus <- tm_map(corpus, FUN = stripWhitespace)
  corpus <- tm_map(corpus, FUN = removeWords, stopwords())
  ## stopwords() returns characters and is feed as second argument
  corpus <- tm_map(corpus, FUN = stemDocument)
}
tweet_corpus <- clean_corp(tweet_corpus)

wordcloud(tweet_corpus)
```

Use TF-IDF Weighting

Now apply TF-IDF weighting from [before](#):

```
# Create a WordCloud Using TF-IDF Weighting #####

## Create a DTM
tweet_matrix <- as.matrix(DocumentTermMatrix(tweet_corpus))
colnames(tweet_matrix)[1:3]

## Use Term-Frequency and Inter-Document Frequency
N <- nrow(tweet_matrix) # Number of Documents
ft <- apply(tweet_matrix, 2, sum)

TF <- log(tweet_matrix + 1)
IDF <- log(N/ft)

# Because each term in TF needs to be multiplied through
# each column of IDF there would be two ways to do it,
# a for loop which will be really slow
# Diagonalise the matrix then use Matrix multiplication

tweet_weighted <- TF %*% diag(IDF)
colnames(tweet_weighted) <- colnames(tweet_matrix)
```

Plot and Post the Words

```
## Filter Relevant words
relevant <- sort(apply(tweet_weighted, 2, mean), decreasing = TRUE)[1:30]

## Make a wordcloud
p <- brewer.pal(n = 5, name = "Set3")
wordcloud(words = names(relevant),
          freq = relevant,
          colors = p, random.color = FALSE)
# Posting Tweets using R =====
tw1 <- rtweet::post_tweet(status = "My first tweet from R, #WSU300958", token = tk)
tw2 <- rtweet::post_tweet(status = "Political Wordcloud Using TF-IDF #WSU300958", media = "~")
```

Use PCA to Visualise the Tweets

The next step is using [PCA](#) to visualise the higher dimensional data.

<!-- Sometimes you need a dot,
 ruins the spacing Important -->

<details>

<p>

:warning: Scaling bug :warning:

</p>

<p>

If you scale the data first, whether by using `scale = TRUE`, using `scale()` or by simply doing $\text{myscale}(x) = \frac{\bar{x}-x}{s}$ the PCA plot will come out with large outliers, like this:

</p>

<!--Newline Important-->

</details>

<!--Newline Important-->

PCA operates by taking a matrix of values, whereby rows are observations and columns are features, this means we need rows as Documents and Columns as features in order to perform PCA, this would be a `DocumentTermMatrix`^[1].

Geometrically a plot of the first two *Principal Components* (Z_1, Z_2) is a projection of data onto the subspace spanned by the first two loading vectors (ϕ_1, ϕ_2)

To perform the PCA use the `prcomp` function:

```
pol.pca <- prcomp(tweet_weighted, scale = FALSE)
```

This returns a `prcomp` object that is essentially a list with

- a rotation matrix
 - (which is a matrix of the loading vectors)
- a matrix of the co-ordinates in the Principle Component frame of reference
 - denoted by `x`



Figure 1: Principle Comonents with Scaling

- `sdev`; The standard deviation attributable to each principle component.
- `centre` which is a logical indicating whether centring was used
 - If centring was used it will be a vector of the `centre` values
- `scale` which is a logical indicating whether scaling was used
 - If scaling was used it will be a vector of the `scale` values

Bi Plot

a [Biplot](#) is a plot of the first two principle components and the corresponding loading vectors. Roughly speaking, the loading vectors help show the explanation each variable contributes to a principle component.^[2]

<p style="font-family: Courier New, Courier, monospace, serif; font-size: 22px; font-style: italic; " align="right" color="blue">

#biplot

</p>

A biplot can be produced by simply using the `biplot` function:

```
biplot(pol.pca, cex = 0.5)
```

See also [the ggbiplot package](#)

Scree Plot

A scree plot is a plot of the variance explained by each principle component, Ideally we would find a vertex where we could cut it off and say that the data can be explained by said principle components, for large data sets like this it isn't the case however.

To create a scree plot basically just plot the `sdev` in as a line or barchart, just be mindful to encode the names of the *Principal Components* as ordered factors because otherwise the order may not be preserved:

```
pcaVar <- pol.pca$sdev ^ 2
pcaVar <- pcaVar[1:10]
pcaVarpr <- pcaVar / sum(pcaVar)
pcaVarpr <- enframe(pcaVarpr)

names(pcaVarpr) <- c("Principal.Component", "Proportion.Variance")
## This throws a warning
for (i in 1:nrow(pcaVarpr)) {
  pcaVarpr[["Principal.Component"]][i] <- paste("PC", i)
}

pcaVarpr$Principal.Component <-
  factor(
    pcaVarpr$Principal.Component,
    ordered = TRUE,
    levels = pcaVarpr$Principal.Component
  )
```

```

ggplot(data = pcaVarpr,
       aes(x = Principal.Component,
           y = Proportion.Variance,
           group = 1)) +
  geom_point(size = 3, alpha = 0.7, col = "RoyalBlue") +
  geom_line(col = "IndianRed") +
  labs(x = "Principal Component",
       y = "Proportion of Variance",
       title = "Variance Explained by PC, TF-IDF Weigthing Tweets by Party Leaders") +
  theme_classic() +
  geom_vline(xintercept = 4,
            col = "purple",
            lty = 2)

```

Plot the First two PCA's

In order to plot the first two PC's, create a data frame:

```

pca_data$Party <-
  factor(c(rep("Greens", n), rep("Labor", n), rep("Liberal", n)))
pol.km <-
  kmeans(tweet_weighted, centers = 3, nstart = 200) %>% as.vector()
pca_data$kmeans <- as.factor(pol.km$cluster)
head(pca_data)

```

Then it's just a matter of calling ggplot:

```

ggplot(data = pca_data,
       aes(x = PC1, y = PC2, size = PC3, col = Party)) +
  geom_point(alpha = 0.3) +
  ## geom_point(aes(shape = kmeans)) +
  theme_classic() +
  ## Colours are applied in order of appearance of the factor
  ## Introduce factors alphabetically where possible
  scale_color_manual(values = c("Palegreen3", "DodgerBlue", "Palevioletred3")) +
  scale_size(range = c(0.1, 3)) +
  labs(main = "Tweets by Part Leaders using TF-IDF Weighting",
       subtitle = "First Two Principle Components without scaling") +
  stat_ellipse() ## +
  ## Scaling Fix
  ## scale_x_continuous(limits = c(-quantile(pca_data$PC1, c(0.99)), quantile(pca_data$PC1,
  ## scale_y_continuous(limits = c(-quantile(pca_data$PC2, c(0.99)), quantile(pca_data$PC2,

```

in this example I encoded the third principle component as the size of the points:



Figure 2: Scree Plot of Tweets



Figure 3: First Two Principle Components

Use Multi-Dimensional Scaling to Visualise the tweets

The whole idea of Multi-Dimensional Scaling is to ### Using the `dist` function in **R** The `dist` function takes each row as a vector and each column as a variable (i.e. axis) of that vector, it then compares every point and produces a table summarising every combination of possible differences, so for example the matrix:

$$\$ \$ \begin{bmatrix} 0 & 0 & 5 \\ 0 & 4 & 0 \\ 3 & 0 & 0 \end{bmatrix} \$ \$$$

that matrix in **R** would really represent the three vectors $\vec{v}_1 = \langle 0, 0, 3 \rangle$, $\vec{v}_2 = \langle 0, 4, 0 \rangle$, $\vec{v}_3 = \langle 5, 0, 0 \rangle$ and so there would be three possible distances we could be possibly be interested in:

- $\|\vec{v}_1 - \vec{v}_2\| = \|\vec{v}_1 - \vec{v}_2\| = d(\vec{v}_1, \vec{v}_2) = d(\vec{v}_2, \vec{v}_1)$
- $\|\vec{v}_1 - \vec{v}_3\| = \|\vec{v}_3 - \vec{v}_1\| = d(\vec{v}_1, \vec{v}_3) = d(\vec{v}_3, \vec{v}_1)$
- $\|\vec{v}_2 - \vec{v}_3\| = \|\vec{v}_3 - \vec{v}_2\| = d(\vec{v}_2, \vec{v}_3) = d(\vec{v}_3, \vec{v}_2)$

R will return this as a table (well a `matrix` class) like this:

| | Vector 1 | Vector 2 | Vector 3 |
|----------|-----------|-----------|----------|
| Vector 1 | 0 | Repeated | Repeated |
| Vector 2 | $d(1, 2)$ | 0 | Repeated |
| Vector 3 | $d(1, 2)$ | $d(1, 2)$ | 0 |

`help(diff)`; This function computes and returns the distance matrix computed by using the specified distance measure to compute the distances between the rows of a data matrix.

So if we did this all in **R** with some arbitrary numbers:

```
(rbind(c("x" = 0, "y" = 0, "z" = 4),
       c("x" = 1, "y" = 1, "z" = 1),
       c("x" = 0, "y" = 3, "z" = 0))) %>% dist()
```

```
      x y z
[1,] 0 0 4
[2,] 1 1 1
[3,] 0 3 0
>
      1      2
2 3.316625
3 5.000000 2.449490
```

Then we could use these distances to visualise these vectors by plotting them in such a way that preserves the distances, in order to do that the `cmdscale` can be used:

`help(cmdscale)`; Multidimensional scaling takes a set of dissimilarities and returns a set of points such that the distances between the points are approximately equal to the dissimilarities. (It is a major part of what ecologists call 'ordination'.)

A set of Euclidean distances on n points can be represented exactly in at most $n - 1$ dimensions. `cmdscale` follows the analysis of [Mardia \(1978\)](#), and returns the best-fitting k -dimensional representation, where k may be less than the argument k . `[[[]]]`

... The configuration returned is given in principal-component axes...

Finding the Distance Matrix of Tweets

In order to find the distance matrix of the tweets first be mindful that the matrix must have rows as observations and columns as features, then it's just a matter of calling `dist` over the matrix, in order to produce the *Classical Multidimensional Scaling* representation, simply pass this distance matrix to the `cmdscale` function and it will do all the work:

```
tweet_matrix[1:3, 1:3]
tweet_dist <- dist(tweet_weighted)
tweet_mds <- cmdscale(d = tweet_dist, k = 2)
tweet_mds.df <- as.data.frame(tweet_mds)
names(tweet_mds.df) <- c("MDX", "MDY")
head(tweet_mds.df)

## Add Colours
tweet_mds.df$Party <- factor(c(rep("Greens", n), rep("Labor", n), rep("Liberal", n)))
```

which will produce a data frame like this:

| | MDX | MDY | Party |
|---|--------|--------|--------|
| | <dbl> | <dbl> | <fct> |
| 1 | -0.427 | 0.211 | Greens |
| 2 | -0.447 | 0.294 | Greens |
| 3 | -0.240 | -0.320 | Greens |
| 4 | -0.541 | 0.249 | Greens |
| 5 | -0.302 | 0.374 | Greens |
| 6 | -0.388 | -0.278 | Greens |

Then it's simply a matter of plotting it which is easy enough:

```
(ggmds <- ggplot(mds_data, aes(x = MDX, y = MDY, col = Party)) +
  geom_point() +
  scale_color_manual(values = c("Palegreen3", "DodgerBlue", "Palevioletred3")) +
  theme_classic())
```

Binary Distance

In much the same way binary distance could be plotted by specifying that particular method (without the niceties of `ggplot`):



Figure 4: MDS Plot of Party Tweets, Euclidean 2dim

```
# Binary Distance =====
## Remember to use the weighted values though,
## The weighted values are a good adjustment and is consistent with different
## measures of distance
dist_mat_bin <- dist(tweet_matrix, method = "binary")
mds_data_bin <- cmdscale(dist_mat_bin, k = 2)
plot(mds_data_bin[,1], mds_data_bin[,2], col = c("ForestGreen", "PowderBlue", "MediumVioletRed"))
```

Cosine Distance

1. Finding Unit Vectors Recall that taking the matrix multiplication of a diagonal matrix is equivalent to multiplying each term of a matrix by a value, for example:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \cdot \begin{pmatrix} b_1 & 0 & 0 \\ 0 & b_2 & 0 \\ 0 & 0 & b_3 \end{pmatrix} = \begin{pmatrix} b_1 a_{1,1} & b_2 a_{1,2} & b_3 a_{1,3} \\ b_1 a_{2,1} & b_2 a_{2,2} & b_3 a_{2,3} \\ b_1 a_{3,1} & b_2 a_{3,2} & b_3 a_{3,3} \end{pmatrix}$$

Play around this in *Wolfram Mathematica*:

```
Aeg = Array[Subscript[a, #1, #2] &, {3, 3}];
Aeg // MatrixForm
Beg = {Subscript[b, 1], Subscript[b, 2], Subscript[b, 3]} //
      DiagonalMatrix
(Aeg.Beg) // MatrixForm
```

So in order to create a vector of unit vectors it is sufficient to merely perform:

```
U <- tweet_matrix %*% diag(1/sqrt(colSums(tweet_matrix^2)))
U_w <- tweet_weighted %*% diag(1/sqrt(colSums(tweet_weighted^2)))
```

2. Vector Dot Product The dot product of two vectors is the area spanned by a rectangle of length equal to the first vector and width equal to the projection (i.e. the shadow) of the first vector onto the second. The cosine similarity is the length of that projection, if the length of both vectors are independently scaled to one, the dot product will be different but the angle won't change, also the area of the dot product will be $1 \times$ the projection:

$$\begin{aligned}
 ||\mathbf{X}|| = ||\mathbf{Y}|| &\implies \frac{\mathbf{X} \cdot \mathbf{Y}}{||\mathbf{X}|| \times ||\mathbf{Y}||} \\
 &= \mathbf{X} \cdot \mathbf{Y} \\
 &= \cos(\mathbf{X}, \mathbf{Y}) \\
 &= \sum_{i=1}^n [x_i y_i]
 \end{aligned}$$

Hence the [Euclidean Distance](#) will be:

$$\begin{aligned}
\text{dist}(\mathbf{X}, \mathbf{Y}) &= \|\mathbf{X} - \mathbf{Y}\| \\
&= \sqrt{\sum_{i=1}^n [(x_i - y_i)^2]} \\
\text{dist}(\mathbf{X}, \mathbf{Y})^2 &= \sum_{i=1}^n [(x_i - y_i)^2] \\
&= \sum_{i=1}^n (x_i^2) + \sum_{i=1}^n (y_i^2) + 2 \sum_{i=1}^n (x_i y_i) \\
&= 1 + 1 + 2 \times \frac{\sum_{i=1}^n (x_i y_i)}{(1)} \\
&= 2 + 2 \times \frac{\sum_{i=1}^n (x_i y_i)}{\|\mathbf{X}\| \times \|\mathbf{Y}\|} \\
&= 2 + 2 \cos(\mathbf{X}, \mathbf{Y}) \\
\implies (1 - \cos(\mathbf{X}, \mathbf{Y})) &= \frac{\text{dist}(\mathbf{X}, \mathbf{Y})}{2}
\end{aligned}$$

The only thing to note is that the cosine similarity is 1 when two vectors are identical in direction but is 0 when they are totally different in direction (i.e. perpendicular or rather orthogonal). For a metric of distance we want it the other way around so we just subtract it from 1.

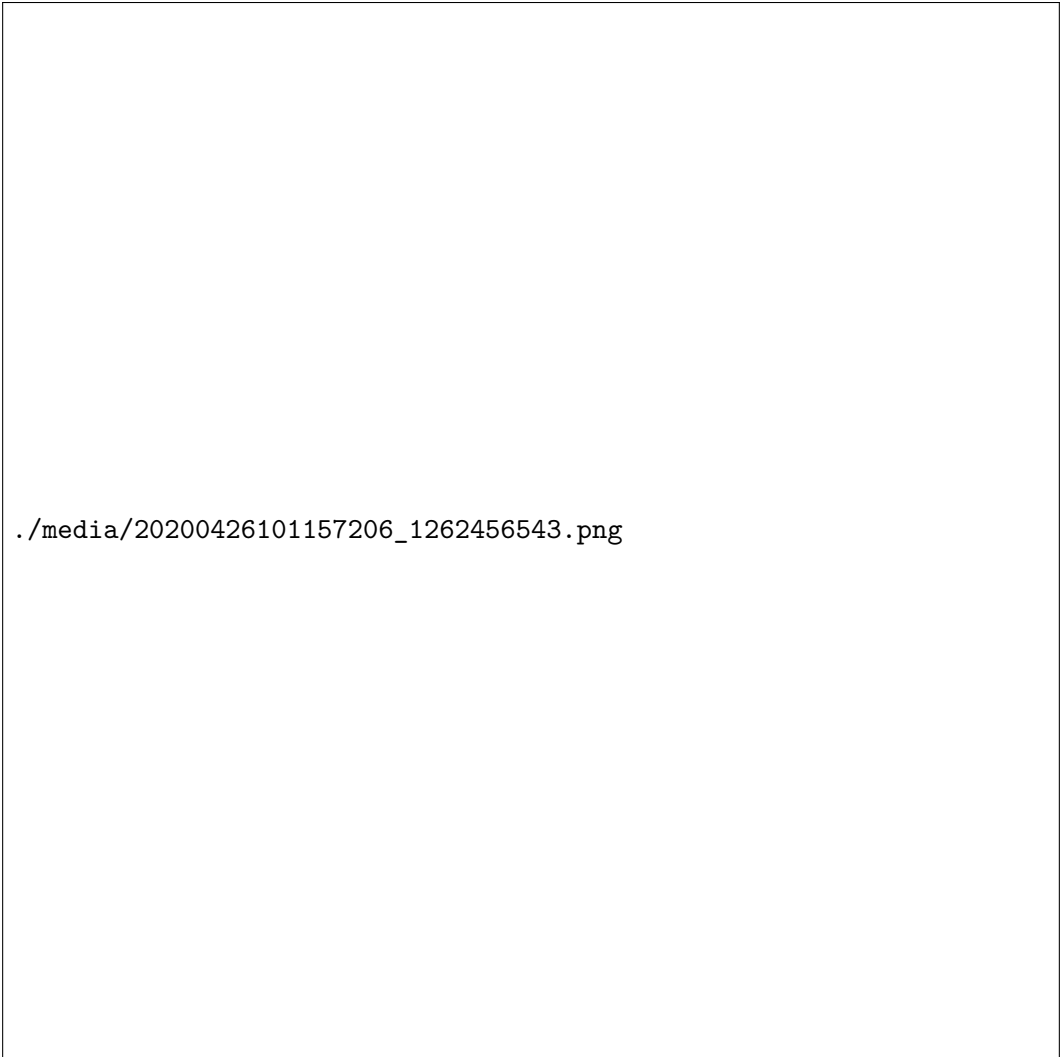
So in order to determine the cosine distance, it is sufficient to perform the following:

```

# Cosine Distance=====
# Create Unit Vectors #####
## Recall matrix multiplication of a diagonalised matrix is product of elements
U <- tweet_matrix %*% diag(1/sqrt(colSums(tweet_matrix^2)))
U_w <- tweet_weighted %*% diag(1/sqrt(colSums(tweet_weighted^2)))
# Create Distance Matrix #####
# Plot the MDS #####
## Make the Distance Matrix
dist_mat_cos <- dist(U_w, method = "euclidean")^2/2
## Make the MDS
mds_data_cos <- cmdscale(dist_mat_cos, k = 2)
names(mds_data_cos) <- c("xvals", "yvals")
plot(mds_data_cos[,1], mds_data_cos[,2], col = pca_data$Party)

```

which will produce something like this:



`./media/20200426101157206_1262456543.png`

This was converted from 'md' to 'org' using 'pandoc -f gfm' at time: 2020-04-26T01-57-51