Chapter 1
Text Analysis in R

1.1 Introduction to the Course

1.1.1 Course Aims

This course gives an introduction to text analysis in R.

The course starts off with basic text analysis, where you'll learn how to import, clean and process text data and employ techniques such as sentiment analysis to produce analysis and insights.

1.1.2 Course Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here's how they look:

```
This is a section of code               # This is a comment
```

A warning, typically describing non-intuitive aspects of the R language

A tip: additional features of R or "shortcuts" based on user experience

Exercises to be performed during (or after) the training course

1.1.3 Course Script and Exercise Answers

The course examples and exercises are available from the Scripts folder in the distributed course files.

1.1.4 Teaching Approach

This will be a hands on course taught using the RStudio IDE with exercises throughout. All attendees will need access to a computer and will need to have pre-installed a recent version of R and RStudio and will need to be able to install R packages. The course will be taught by Mango Solutions consultants.

Chapter 2
Analysing Text Data in R

As the majority of data is unstructured, and most analysis requires structured data to produce any meaningful results, we first need to manipulate this data into a useable format i.e. into a data frame. Thankfully, an excellent package available to us is the **tidytext** package, and is the package we will be focusing on here.

There are several other packages that in recent years have proven popular within the community; of which **tm**, **RWeka** and **openNLP** are the forerunners. However, with the rapidly increasing interest in the ability to analyse large quantities of qualitative data, new and powerful packages are emerging.

2.1 Tidying and Summarising Text Data

2.1.1 The tidytext Package

The **tidytext** package was released in October 2016 and has since proven very popular among the R community. The key feature of the tidytext package is the ability to process text data into a usable format and then easily apply common analysis approaches such as **sentiment analysis**. tidytext integrates well with other packages in the tidyverse, such as **dplyr** and **tidyr** so the infrastructure to build tidy data is already available within the package.

The definition of tidy text within this package is described as one-term-per-row and within this chapter we will be taking the raw string format and coercing it to this more readily useable format.

Function	Usage
`unnest_tokens`	Converts unstructured data to one-token-per-document-per-row structure
`stop_words`	English stop words to remove for tidied data
`get_sentiments`	Retrieves sentiment lexicons in a tidy format
`inner_join`	Useful for joining sentiment to our tidied data
`tidy`	Tidying model objects into a data frame
`%>%`	The pipe operator from the **magrritr** package is invaluable in making our text analysis code workflow readable

2.1.2 The gutenbergr Package

To begin we need some data we are interested in analysing. The **gutenbergr** package provides the functionality to download and process public domain books from the Gutenberg project collection. This project contains over 5000 different datasets which we can use to analyse and find words of interest.

The function `gutenberg_works` gives us all the different texts available to us and the function `gutenberg_download` downloads, when supplied with one or more ID numbers, a complete text. For the following examples we will be using “The Wonderful Wizard of Oz” text which has the Gutenberg ID of **55**.

```
library(gutenbergr)

# view The Wizard of OZ
gutenberg_works(title == "The Wonderful Wizard of Oz")
#> # A tibble: 1 x 8
#>   gutenberg_id title author gutenberg_autho~ language gutenberg_books~
#>          <int> <chr> <chr>             <int> <chr>    <chr>
#> 1           55 The ~ Baum,~               42 en       Children's Lite~
#> # ... with 2 more variables: rights <chr>, has_text <lgl>

# download text
wizardOfOz <- gutenberg_download(55)

head(wizardOfOz)
#> # A tibble: 6 x 2
#>   gutenberg_id text
#>          <int> <chr>
#> 1           55 "[Illustration]"
#> 2           55 ""
#> 3           55 ""
#> 4           55 ""
#> 5           55 ""
#> 6           55 "The Wonderful Wizard of Oz"
```

2.1.3 Tokenising Data

A token can be described at a low level as a meaningful unit of text, usually a single word that we intend to use in our analysis. We tokenise data by breaking out a string of text into individual tokens.

Using our Wizard of Oz text from the Gutenberg package, we can use the tidytext package to tokenize out data. The `unnest_tokens` function requires 3 main arguments to be supplied. First, the data frame we wish to perform tokenization upon, secondly the name of the output column we are creating and finally the existing column we wish to tokenize. As shown below, and as previously mentioned,

the tidytext package integrates with dplyr functionality, so we can use the pipe operator to pass our wizard of Oz data into the `unnest_tokens` function. Further formatting arguments can be passed such as the units to tokenize by e.g. “words” or “sentences”.

```r
library(tidytext)
library(dplyr)

# convert text into tokenised format
tidyWizard <- wizardOfOz %>% unnest_tokens(word, text)
head(tidyWizard)
#> # A tibble: 6 x 2
#>   gutenberg_id word
#>          <int> <chr>
#> 1           55 illustration
#> 2           55 the
#> 3           55 wonderful
#> 4           55 wizard
#> 5           55 of
#> 6           55 oz
```

2.1.4 Counting Word Frequencies

We now have our data in a format we can work with. If we perform a simple count of the most common words, we find that the most frequently used words are normally rather uninteresting, such as 'the' and 'and'. These types of words are referred to as "stop words".

```r
tidyWizard %>% count(word, sort = TRUE) %>% head()
#> # A tibble: 6 x 2
#>   word      n
#>   <chr> <int>
#> 1 the    3004
#> 2 and    1671
#> 3 to     1118
#> 4 of      847
#> 5 a       807
#> 6 i       598
```

2.1.5 Removing Stop Words

We can filter out these uninteresting words by performing an anti-join using the handy `stop_words` included in the tidytext package. Any words that appear in the `stop_words` object will be removed from our data. We can now see the most common words which relate specifically to the Wizard of Oz text.

```
head(stop_words)
#> # A tibble: 6 x 2
#>   word      lexicon
#>   <chr>     <chr>
#> 1 a         SMART
#> 2 a's       SMART
#> 3 able      SMART
#> 4 about     SMART
#> 5 above     SMART
#> 6 according SMART

tidyWizard  %>%
  anti_join(stop_words, by = "word") %>%
  count(word, sort = TRUE) %>%
  head()
#> # A tibble: 6 x 2
#>   word          n
#>   <chr>     <int>
#> 1 dorothy     347
#> 2 scarecrow   219
#> 3 woodman     176
#> 4 lion        173
#> 5 oz          164
#> 6 tin         140
```

2.1.6 Removing Other Unwanted Content

We then may have some data specific words that we are simply not interested in analysing or not needed for our analysis. Below is an example whereby a vector containing the character names in the book is created and used in an anti-join to further refine the data we wish to analyse.

```
characters <- data.frame(word = c("dorothy", "scarecrow", "woodman",
                                  "lion", "tin", "witch", "toto"))

tidyWizardNoCharacters <-  tidyWizard %>%
  anti_join(stop_words, by = "word") %>%
  anti_join(characters, by = "word") %>%
  count(word, sort = TRUE)

head(tidyWizardNoCharacters)
#> # A tibble: 6 x 2
#>   word         n
#>   <chr>    <int>
#> 1 oz         164
#> 2 green      104
#> 3 girl        93
#> 4 head        90
#> 5 city        82
#> 6 answered    78
```

2.1.7 Stemming

It is important to be able to identify words with a common meaning when analysing text data. For example, we would like to consider the words “fearing” and “fearsome” as the same as they are both derived from the word “fear”. This process is called stemming, whereby removing suffixes from words retrieves their common origin.

The **SnowballC** package provides an R interface for collapsing words to a common root using the Porter’s word stemming algorithm.

```
library(SnowballC)

wordStem(c("fear", "fearing", "fearful", "play", "played", "playing"))
#> [1] "fear" "fear" "fear" "plai" "plai" "plai"


tidyWizardStemmed <-  tidyWizard %>%
  anti_join(stop_words) %>%
  mutate(word = wordStem(word)) %>%
  count(word, sort = TRUE)
#> Joining, by = "word"

head(tidyWizardStemmed)
#> # A tibble: 6 x 2
#>   word          n
#>   <chr>     <int>
#> 1 dorothi     347
#> 2 scarecrow   221
#> 3 lion        176
#> 4 woodman     176
#> 5 oz          164
#> 6 tin         140
```

Exercise

1. Extract the “Treasure Island” text from the **gutenbergr** public library and tidy the book into the tidy one-token-per-row structure
2. Find the most common words in the Treasure Island text
3. What are the most common words after removing stop words
4. Stem the Treasure Island text. Does this change the order of the most common words within the text?

2.2 “ngrams” and Relationships Between Words

We can use the argument token = “ngrams” to tokenize pairs of adjacent words rather than individually tokenizing the words. This allows us to see how often one word is directly followed by another. For example, if we set n = 2 we will be pairing by 2 consecutive words.

```
# tokenise into bigrams
tidyWizardNgram <- wizardOfOz %>%
  unnest_tokens(word, text, token = "ngrams", n = 2)

tidyWizardNgram  %>%
  count(word, sort = TRUE) %>%
  head()
#> # A tibble: 6 x 2
#>   word              n
#>   <chr>         <int>
#> 1 <NA>           1352
#> 2 of the          316
#> 3 the scarecrow   183
#> 4 to the          176
#> 5 and the         152
#> 6 said the        150
```

It is important to point out that our data remains in the same tidy format, however each token is now represented in what we call a bigram (a pairing of words). Each bigram also overlaps, i.e. the second word in the first bigram is used as the first word of the next.

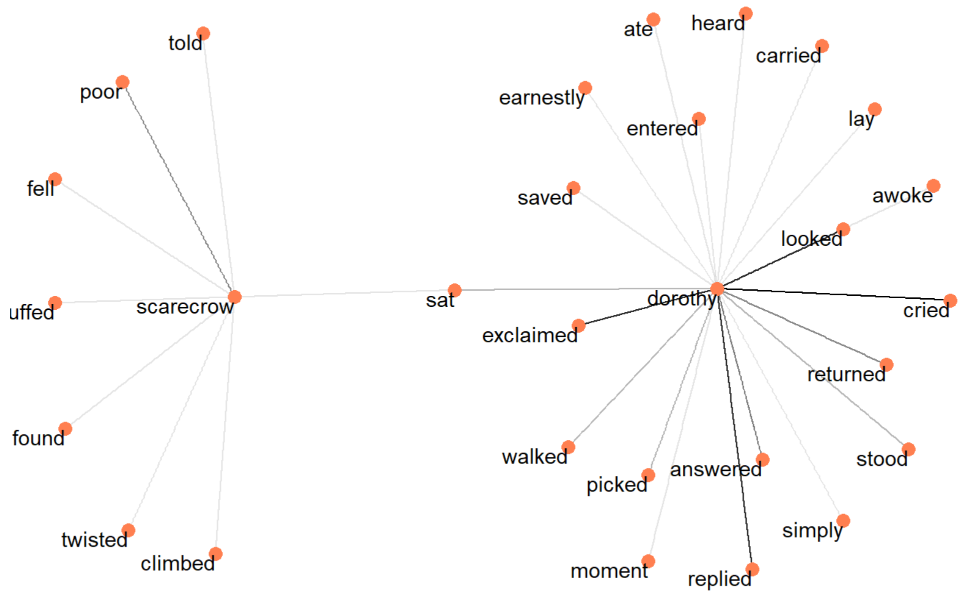
From here it is helpful to split the words into two columns so we can easily filter out stop words or similar.

```
# make tidy ngram data
tidyWizardNgram  <- wizardOfOz %>%
   unnest_tokens(word, text, token = "ngrams", n = 2) %>%
   count(word, sort = TRUE) %>%
   separate(word, c("firstWord", "secondWord"), sep = " ")

head(tidyWizardNgram)
#> # A tibble: 6 x 3
#>   firstWord secondWord     n
#>   <chr>     <chr>      <int>
#> 1 <NA>      <NA>        1352
#> 2 of        the          316
#> 3 the       scarecrow    183
#> 4 to        the          176
#> 5 and       the          152
#> 6 said      the          150

# remove stop words
tidyWizardNgram <- tidyWizardNgram %>%
   anti_join(stop_words, by = c("firstWord" = "word")) %>%
   anti_join(stop_words, by = c("secondWord" = "word"))

head(tidyWizardNgram)
#> # A tibble: 6 x 3
#>   firstWord secondWord     n
#>   <chr>     <chr>      <int>
#> 1 <NA>      <NA>        1352
#> 2 tin       woodman      107
#> 3 wicked    witch         56
#> 4 emerald   city          53
#> 5 winged    monkeys       28
#> 6 aunt      em            22
```

2.3 Visualising Related Words

Now we have our ngrams in a nice format, we can visualise these relationships. Here, we show a network plot where the most common words associated with two of the characters are visualised.

```
library(ggraph)
library(igraph)
library(tidyr)

# Make tidy ngram data
tidyWizardNgram  <- wizardOfOz %>%
   unnest_tokens(word, text, token = "ngrams", n = 2) %>%
   count(word, sort = TRUE) %>%
   separate(word, c("firstWord", "secondWord"), sep = " ")

# Remove stop words
tidyWizardNgram <- tidyWizardNgram %>%
   anti_join(stop_words, by = c("firstWord" = "word")) %>%
   anti_join(stop_words, by = c("secondWord" = "word"))

# Filter so only common pairs about Dorothy or the scarecrow remain
tidyWizardNgram <- tidyWizardNgram %>%
  filter((firstWord  %in% c("dorothy", "scarecrow")) |
           secondWord %in% c("dorothy", "scarecrow"),
         n > 1)


head(tidyWizardNgram)
#> # A tibble: 6 x 3
#>   firstWord secondWord     n
#>   <chr>     <chr>      <int>
#> 1 cried     dorothy        7
#> 2 dorothy   looked         6
#> 3 exclaimed dorothy        6
#> 4 replied   dorothy        6
#> 5 answered  dorothy        4
#> 6 poor      scarecrow      4

# Create the igraph object
igraph_wizard <-  graph_from_data_frame(tidyWizardNgram)

# Plot the ggraph
ggraph(igraph_wizard, layout = 'stress') +
  geom_edge_link(aes(edge_alpha = n), show.legend = FALSE) +
  geom_node_point(color = "coral", size = 3) +
  geom_node_text(aes(label = name), vjust = 1, hjust = 1) +
  theme_void()
```



Exercise

1. Tokenize the Treasure Island data into word triplets (trigrams)
2. Which words co-occur most frequently?

Chapter 3

Sentiment Analysis Using tidytext

Gathering the sentiment from a section of text is very useful to gain an automated understanding of the general feel of a document. It is a commonly used technique and can be useful when analysing social media content, or survey responses from customers, as it allows quick classification and filtering to get a feel for the popularity or unpopularity of a certain trend in the form of easily consumable metrics.

There are a few packages available in R that are designed to make sentiment analysis quick and easy, including the *sentiments* package. Here however, we are going to use the `get_sentiments` function from the tidytext package. This function returns words with a pre-categorised sentiment score allowing us to join this onto our data and calculate the sentiments.

3.1 The Sentiments Data Frame

First, we have to start with importing and tidying our text data. We will use the Wizard of Oz in our example.

```
library(gutenbergr)

# download text
wizardOfOz <- gutenberg_download(55)

# Make tidy data
tidyWizard <- wizardOfOz %>%
   unnest_tokens(word, text) %>%
   anti_join(stop_words) %>%
   count(word, sort = TRUE)
#> Joining, by = "word"


head(tidyWizard)
#> # A tibble: 6 x 2
#>   word          n
#>   <chr>     <int>
#> 1 dorothy     347
#> 2 scarecrow   219
#> 3 woodman     176
#> 4 lion        173
#> 5 oz          164
#> 6 tin         140
```

`get_sentiments` returns sentiment lexicons in an already tidy format which we can use to join with our one-word-per-row data set. There are four different lexicons sets we can use; the **afinn**, **bing**, **nrc** and **loughran**. “nrc” and “bing” are both character classifications, by which words are categorised into a positive or negative sentiment.

The “AFINN” lexicon uses a numeric scoring system running from -5 to +5, where -5 is a negative sentiment and +5 a positive. The “loughran” sentiment imports the Loughran and McDonald dictionary of sentiment for words specific to financial reports.

```
sentiment <- get_sentiments(lexicon = "bing")

head(sentiment)
#> # A tibble: 6 x 2
#>   word        sentiment
#>   <chr>       <chr>
#> 1 2-faces     negative
#> 2 abnormal    negative
#> 3 abolish     negative
#> 4 abominable negative
#> 5 abominably negative
#> 6 abominate  negative

tidyWizard %>%
  inner_join(sentiment, by="word") %>%
  head()
#> # A tibble: 6 x 3
#>   word          n sentiment
#>   <chr>     <int> <chr>
#> 1 wicked       74 negative
#> 2 beautiful    38 positive
#> 3 afraid       29 negative
#> 4 golden       29 positive
#> 5 terrible     29 negative
#> 6 pretty       28 positive
```

From here, it is possible to analyse if the Wizard of Oz is overall positive or negative.

```
tidyWizard %>%
  inner_join(sentiment, by="word") %>%
  group_by(sentiment) %>%
  summarise(total = sum(n))
#> # A tibble: 2 x 2
#>   sentiment total
#>   <chr>     <int>
#> 1 negative   1105
#> 2 positive    840
```

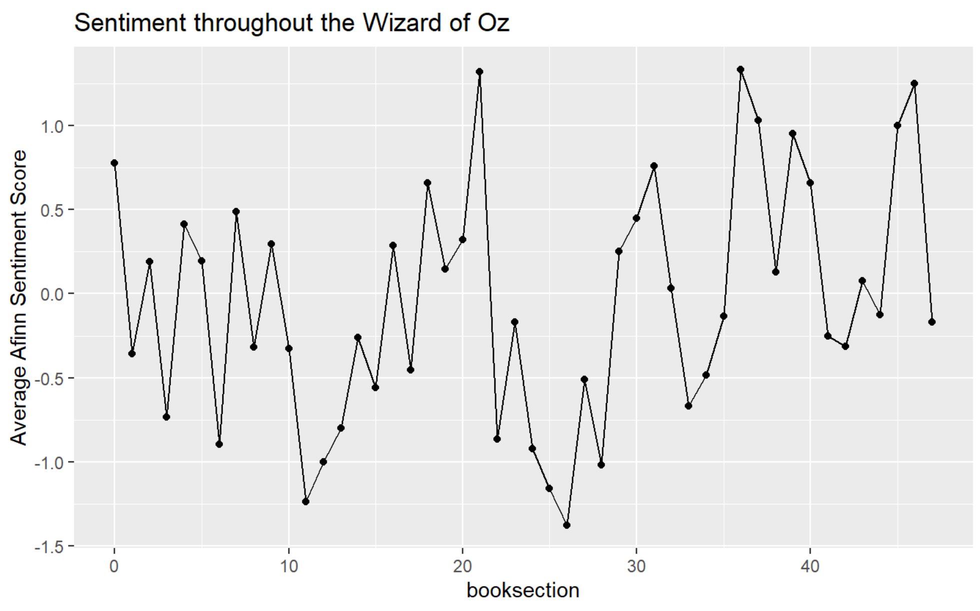
3.2 Producing Quantitative Results from Text

The “AFINN” sentiment can be used to convert our qualitative data into quantitative results that we can further analyse.

In the below example we can look at how the sentiment changes throughout the Wizard of Oz.

The `%/%` operator is used for integer division to split the text into sections, rather than plotting the sentiment for each line.

```
library(tidyverse)

# get the data into the correct form, keeping line number
afinnWizard <- wizardOfOz  %>%
  mutate(lineNumber = row_number()) %>%
  unnest_tokens(word, text) %>%
  anti_join(stop_words, by = "word") %>%
  inner_join(get_sentiments("afinn"), by = "word")

head(afinnWizard)
#> # A tibble: 6 x 4
#>   gutenberg_id lineNumber word      value
#>          <int>      <int> <chr>     <dbl>
#> 1           55          6 wonderful     4
#> 2           55         11 dedicated     2
#> 3           55         21 saved         2
#> 4           55         23 rescue        2
#> 5           55         24 cowardly     -2
#> 6           55         30 wicked       -2

# split into sections of 100 lines and summarise
afinnWizard <- afinnWizard %>%
  mutate(booksection = lineNumber %/% 100) %>%
  group_by(booksection) %>%
  summarise(score = mean(value))

head(afinnWizard)
#> # A tibble: 6 x 2
#>   booksection  score
#>         <dbl>  <dbl>
#> 1           0  0.778
#> 2           1 -0.357
#> 3           2  0.191
#> 4           3 -0.735
#> 5           4  0.414
#> 6           5  0.194

# plot the sentiment though the book
ggplot(afinnWizard , aes(booksection,  score)) +
  geom_line()  +
  geom_point() +
  labs(title = "Sentiment throughout the Wizard of Oz",
       y = "Average Afinn Sentiment Score")
```



Exercise

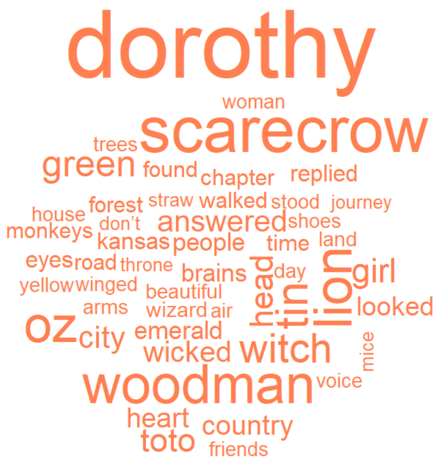
1. What are the most frequently expressed positive words in Treasure Island?
2. Is the Treasure Island text mostly positive or negative?
3. How does sentiment change throughout the treasure Island text?

3.3 Word Clouds

Word clouds are a very popular visualise the key words within text data. Now our data is in the correct format, a word cloud is very simple to produce. The `wordcloud` function takes two main arguments: the words to include in the image and a count of their frequency. We can then use some of the other arguments available to change the aesthetics of our word cloud. Below are some (not all) of the available arguments to the `wordcloud` function.

Argument	Default Value	Description
`words`		The words
`freq`		Their frequencies
`scale`	c(4,.5)	A vector of length 2 indicating the range of the size of the words
`min.freq`	3	Words with frequency below min.freq will not be plotted
`max.words`	Inf	Maximum number of words to be plotted. least frequent terms dropped
`random.order`	FALSE	Plot words in random order. If false, they will be plotted in decreasing frequency
`random.color`	FALSE	Choose colours randomly from the colours object. If false, the colour is chosen based on the frequency
`rot.per`	.1	Proportion of words with 90 degree rotation
`colors`	“black”	Colour words from least to most frequent

```r
library(wordcloud)

cloudWizard <-  wizardOfOz %>%
  unnest_tokens(word, text)%>%
  anti_join(stop_words) %>%
  count(word, sort = TRUE)
#> Joining, by = "word"

wordcloud(words = cloudWizard$word,
          freq = cloudWizard$n,
          max.words = 50, colors ="coral")
```

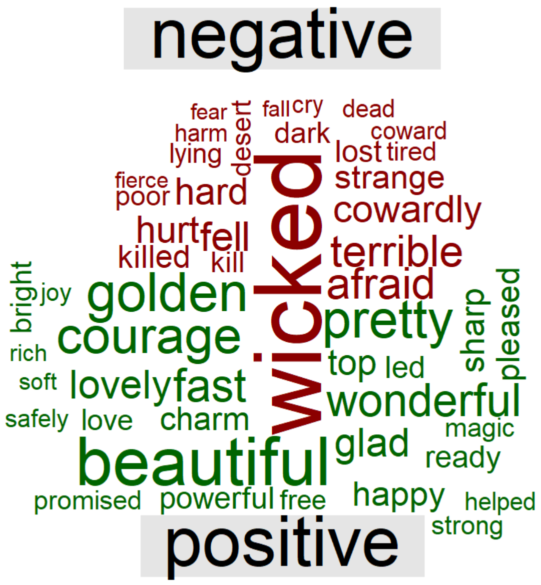


As well as basic word clouds, we can also produce comparison clouds. These types of cloud plots are particularly useful when comparing words between two different documents. Here, we will use the technique to contrast words of different sentiments within the text.

The word data we input to `comparison.cloud` needs to be in a special format known as a *term-frequency matrix*. Here, the words become row names and our columns are sentiment counts for each group we want to compare. We will split our words into two documents, one for each sentiment, and plot each together on a comparrison cloud.

Unfortunately, the tidyverse dataframe, the tibble, does not support names rows, so we have to convert our data back into a base R dataframe.

```r
# get data into the correct format
compCloud <- wizardOfOz %>%
        unnest_tokens(word, text)%>%
        anti_join(stop_words, by = "word") %>%
        inner_join(get_sentiments("bing"), by = "word") %>%
        count(word, sentiment, sort = TRUE) %>%
        pivot_wider(names_from = sentiment,
                    values_from = n,
                    values_fill = list(n = 0)) %>%
        data.frame() # tibble rownames are deprecated

# set words to row names
rownames(compCloud) <- compCloud$word
compCloud <- select(compCloud, - word)

head(compCloud)
#>             negative positive
#> wicked            74        0
#> beautiful          0       38
#> afraid            29        0
#> golden             0       29
#> terrible          29        0
#> pretty             0       28

# plot comparison cloud
comparison.cloud(compCloud,
                 colors = c("darkred", "darkgreen"),
                 max.words = 50)
```



Exercise

1. Make two word clouds, one using the positive and the other using the negative sentiment words from the Treasure Island text
2. Combine/compare the two clouds

Chapter 4
Word Document Frequency

4.1 Term Frequency - Inverse Document Frequency (TF-IDF)

Finding the most important words in a text is very useful in gaining a quick summary of a document. Finding the most common words can be an effective technique, but is limited by the quality of the stop words and doesn't provide a comparison to other documents.

Term Frequency - Inverse Document Frequency (TF-IDF) is a technique that can pick out the words that are key to one document specifically out of a set of documents. For example, if you had a series of financial reports they are all likely to contain many similar words, such as "to", "a", "the", "budget", "stock", ect. Stop words can be used to filter out the first half of these words, but would leave in the later half, resulting in just of list of generic financial words. TF-IDF would allow us to find words that are repeated often in the document of interest, but not in the others.

The TF-IDF of a given token (word) is the product of two statistics - the term frequency, TF, and the inverse document frequency, IDF. There are various ways to calculate these, but in its simplest from they are described below;

$$TF = \frac{n_t}{n_{dt}}$$

$$IDF = ln\left(\frac{N_d}{N_{td}}\right)$$

Here, n_t denotes the number of times a term appears in a given document, n_{dt} is the total number of terms in a given docuemnt, N_{td} denotes the number of documents that contain the term, and N_d is the total number of documents. Dividing by the total length of the document when calculating the term frequency avoids bias from large douments.

4.2 Applying TF-IDF

Here, we will continue to use the wizard of Oz as our example, but to generate several documents we will split it into several sections. The `%/%` operator is used for integer division - this allows us to split the text into sections of 1000 words.

To apply the TF-IDF, we will use the `bind_tf_idf` function from the **tidytext** package. This requires the data to be in "tidy" form, with a column for the token's count for each document as shown below;

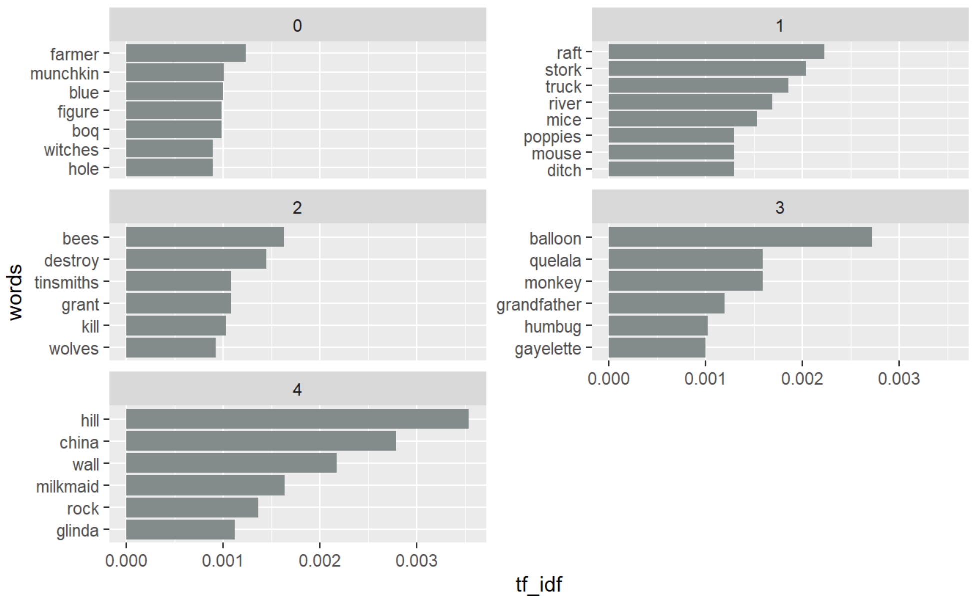
```
# import the text
wizardOfOz <- gutenberg_download(55)

# make into tidy form and add line numbers
tidyWizard <- wizardOfOz %>%
  mutate(lineNumber = row_number()) %>%
  unnest_tokens(word, text)

# label into sections by line number
tidySplitWizard <- tidyWizard %>%
  mutate(booksection = lineNumber %/% 1000) # split every 1000 lines

# add document-term count
tidySplitWizard <- tidySplitWizard %>%
  group_by(word, booksection) %>%
  summarise(count = n()) %>%
  ungroup() %>%
  arrange(-count)
#> `summarise()` has grouped output by 'word'. You can override using the
         `.groups` argument.

head(tidySplitWizard, 8)
#> # A tibble: 8 x 3
#>   word  booksection count
#>   <chr>       <dbl> <int>
#> 1 the             1   679
#> 2 the             2   672
#> 3 the             0   606
#> 4 the             3   589
#> 5 the             4   458
#> 6 and             2   424
#> 7 and             1   354
#> 8 and             0   340


# apply TF-IDF
tidySplitWizard <- tidySplitWizard %>%
  bind_tf_idf(term = word, document = booksection, n = count)

head(tidySplitWizard, 8)
#> # A tibble: 8 x 6
#>   word  booksection count     tf   idf tf_idf
#>   <chr>       <dbl> <int>  <dbl> <dbl>  <dbl>
#> 1 the             1   679 0.0781     0      0
#> 2 the             2   672 0.0755     0      0
#> 3 the             0   606 0.0741     0      0
#> 4 the             3   589 0.0729     0      0
#> 5 the             4   458 0.0775     0      0
#> 6 and              2    424 0.0476     0       0
#> 7 and              1    354 0.0407     0       0
#> 8 and              0    340 0.0416     0       0
```

You can now see that the TF-IDF for words like “the” and “and” is very small, eliminating the need for stop words or similar.

4.2.1 Visualising the TF-IDF results

Now we have applied the TF-IDF function to our text we can see which words are most indicative of each section. One way to do this is to plot the words with the highest TF-IDF for each book section;

```
# Visualise top word by section?
tidySplitWizard %>%
  group_by(booksection) %>%
  top_n(6, tf_idf) %>%
  ggplot(aes(reorder_within(x = word,
                             by = tf_idf,
                             within = booksection),
             y = tf_idf)) +
    geom_col(show.legend = FALSE, fill = "azure4") +
    facet_wrap(vars(booksection), ncol = 2, scales = "free_y") +
    coord_flip() +
    scale_x_reordered("words")
```



Exercise

1. Split the Treasure Island text into several sections, similar to above, and then manipulate it into the form required for the TF-IDF analysis
2. Find the TF-IDF for the words in Treasure Island, by book section
3. Visualise your results