MANGO SOLUTIONS

Workshop: Text Analysis in R

Thriving on Data Science

LEVEL: Intermediate (3)
DURATION: 1/2 Day Workshop

training@mango-solutions.com
+44 (0)1249 705 450
mango-solutions.com

Chapter 1
Introduction to the Workshop

1.1 Course Aims

This course gives an introduction to text analysis in R.

Many companies have a large amount of data stored as text that is not being used effectively. In this introductory workshop we will show how you can get started with analysing text data, from simple manipulation through to sentiment analysis. By the end of the course attendees will have a good understanding of the techniques as well as how to implement them in R.

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

1.3 Course Script and Exercise Answers

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

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 usable 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 Basic Text Manipulation with stringr and Regular Expressions

In R, text data is stored as strings. The stringr package provides simple and easy to use functions for manipulating strings, and combined with regular expression provides a powerful set of tools to start processing text data.

The stringr package provides a collection of very useful functions for manipulating strings. These can be useful in cleaning, combining, or extracting certain text fields in data. There are also two very useful functions from the tidyr package that will be used: `separate` and `unite`.

Function	Usage
`str_length`	Gives the number of characters in a string
`str_to_lower`	Converts all characters in a string to lower case
`str_to_upper`	Converts all characters in a string to upper case
`str_sub`	Selects part of a string by position
`str_c`	Combines several strings into one
`str_split`	Splits a string into multiple pieces
`str_view`	Views the matching pattern of a string
`str_remove`	Removes the matching pattern of a string
`str_detect`	Boolean matching to a pattern in a string
`str_replace`	Replaces the matching pattern of a string
`str_locate`	Locates the matching pattern of a string
`str_extract`	Extracts the matching pattern of a string
`separate`	Splits a column of strings by a separator
`unite`	Joins several columns of strings into one

Several of these functions have a `str_*_all` equivalent such as `str_replace_all`, and `str_remove_all`. Here these replace/removes all matching characters rather than just

the first match.

The `str_c` function combines several strings, into one. You can use the `sep` or `collapse` arguments to control how this happens.

```
library(stringr)

str_c("Mango", "the", "cat")
```

```
## [1] "Mangothecat"
```

```
str_c("Mango", "the", "cat", sep = " ")
```

```
## [1] "Mango the cat"
```

```
str_vec <- c("Mango", "the", "cat")
str_c(str_vec)
```

```
## [1] "Mango" "the"   "cat"
```

```
str_c(str_vec, collapse = " ")
```

```
## [1] "Mango the cat"
```

```
str_c(str_vec, 1:3, sep = "-", collapse = " ")
```

```
## [1] "Mango-1 the-2 cat-3"
```

The `str_sub` function extracts parts of a string based on start and end positions. Negative values can be used to count backwards from the last character.

```
str <- "Mango the cat"

str_sub(str, start = 1, end = 5)
```

```
## [1] "Mango"
```

```
str_sub(str, start = -3)
```

```
## [1] "cat"
```

2.1.1 Regular Expressions

The patterns mentioned in the above table can just be a simple string of characters, or can be regular expressions. Regular expressions provide a powerful language for matching patterns in strings. They are the subject of many long and detailed books, but here we will just introduce a few useful examples.

Regular expressions, or 'regex', are a way of encoding more general patterns in text data, such as 'a character followed by three numbers'. They can be built up from a few simple rules, into long and powerful expressions that can be used to match emails, phone numbers, specific ID codes, and so on.

Below is a table of some useful patters;

Pattern	Usage
`A`	Any character matches to itself
`.`	The `.` wildcard matches to anything
`|`	Logical 'or' for matching patterns
`[...]`	One of the characters in the brackets
`[^...]`	Not one of the characters in the brackets
`[A-Z]`	An uppercase letter
`[a-z]`	A lowercase letter
`[0-9]`	A number 0 to 9
`A$`	'$' anchors match 'A' to end character
`^A`	'^' anchors match 'A' to start character
`A*`	Matches 'A' zero or more times
`A+`	Matches 'A' one or more times
`A?`	Optionally matches 'A'
`A{n}`	Matches 'A' *n* times
`A{n1, n2}`	Matches 'A' *n1* to *n2* times

For this next section we will use the `people_data.rds` dataset.

This data is a set of dummy names, emails, ids, and purchase amounts.

```
people_data <- read_rds("../data/people_data.rds")

people_data
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B    £12
## 4        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA   null
## 5       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
## 6       Fran       Evans      fevans@kiwi-solutions.com RN382A    34r
## 7       Gary      Thomas     gthomas@kiwi-solutions.com ZT720B     26
```

You may notice the `amount` column contains some unwanted characters, as is common in real datasets. We can use some of the stringr functions with regular expressions to help us clean this data.

`str_remove_all` can be used to remove the offending characters.

```
# could remove £
people_data %>%
  mutate(amount = str_remove_all(
    amount, pattern = "£"
  ))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B     12
## 4        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA   null
## 5       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
## 6       Fran       Evans      fevans@kiwi-solutions.com RN382A    34r
## 7       Gary      Thomas     gthomas@kiwi-solutions.com ZT720B     26
```

```
# £ or r
people_data %>%
  mutate(amount = str_remove_all(
    amount, pattern = "£|r"
  ))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B     12
## 4        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA   null
## 5       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
## 6       Fran       Evans      fevans@kiwi-solutions.com RN382A     34
## 7       Gary      Thomas     gthomas@kiwi-solutions.com ZT720B     26
```

```
# [] - means any of these
# could remove £ or r
people_data %>%
  mutate(amount = str_remove_all(
    amount, pattern = "[£r]"
  ))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B     12
## 4        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA   null
## 5       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
## 6       Fran       Evans      fevans@kiwi-solutions.com RN382A     34
## 7       Gary      Thomas     gthomas@kiwi-solutions.com ZT720B     26
```

If we later found a different offending character we would have to update our code. Instead, we can remove everything that is *not* a number;

```
people_data %>%
  mutate(amount = str_remove_all(
    amount, pattern = "[^0-9]"
    ))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B     12
## 4        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA
## 5       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
## 6       Fran       Evans      fevans@kiwi-solutions.com RN382A     34
## 7       Gary      Thomas     gthomas@kiwi-solutions.com ZT720B     26
```

This is effective but we do have to be careful - we may not know what the characters we removed were for. For example, they could be a % sign meaning that we should divide by 100, or could be anything!

Instead of re-formatting the column, we could just filter the rows where we have the correctly formatted data, or view only the data where it is not.

```
# This gives us the rows with those incorrect characters in
people_data %>%
  filter(str_detect(amount,
                    pattern = "[^0-9]"))
```

```
##   first_name second_name                          email     id amount
## 1        Cam    Williams cwilliams@banana-solutions.com AG978B    £12
## 2        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA   null
## 3       Fran       Evans      fevans@kiwi-solutions.com RN382A    34r
```

```
# The rows without can be found using the negate argument;
people_data %>%
  filter(str_detect(amount,
                    pattern = "[^0-9]", negate = TRUE))
```

```
##   first_name second_name                        email     id amount
## 1       Abby       Jones ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith bsmith@mango-solutions.com EX981B     14
## 3       Ella      Davies edavies@kiwi-solutions.com DE700B     18
## 4       Gary      Thomas gthomas@kiwi-solutions.com ZT720B     26
```

Once we have decided it makes sense to keep and clean the column, it still needs to be converted to the correct data type;

```
# Currently the column amount is treated as a character;
# we can convert it;
people_data %>%
  mutate(amount = as.numeric(
    str_remove_all(
      amount, pattern = "[^0-9]")
    ))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B     12
## 4        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA     NA
## 5       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
## 6       Fran       Evans      fevans@kiwi-solutions.com RN382A     34
## 7       Gary      Thomas     gthomas@kiwi-solutions.com ZT720B     26
```

```
# we still get the valid NA for 'null'
```

Next, lets draw our attention to the `id` column.

Using `str_detect` we can pull out specific IDs that match a certain pattern using regular expressions.

```
# can pull out the ID's that end in A
people_data %>%
  filter(str_detect(id, pattern = "A$"))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones   ajones@mango-solutions.com QO509A     23
## 2        Dan      Taylor dtaylor@banana-solutions.com JDJWLA   null
## 3       Fran       Evans    fevans@kiwi-solutions.com RN382A    34r
```

```
# can pull out the ID's that start in letters in the first half of the
  alphabet
people_data %>%
  filter(str_detect(id, pattern = "^[A-M]"))
```

```
##   first_name second_name                          email     id amount
## 1        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 2        Cam    Williams cwilliams@banana-solutions.com AG978B    £12
## 3        Dan      Taylor   dtaylor@banana-solutions.com JDJWLA   null
## 4       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
```

```
# can pull out the ID's whose's second character is a vowel
people_data %>% filter(str_detect(id, pattern = "^.[AEIOU]"))
```

```
##   first_name second_name                      email     id amount
## 1       Abby       Jones ajones@mango-solutions.com QO509A     23
## 2       Ella      Davies edavies@kiwi-solutions.com DE700B     18
```

```
# can pull out the ID's where one of the first two characters is a vowel
people_data %>% filter(str_detect(id, pattern = "^.?[AEIOU]"))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B    £12
## 4       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
```

Watch out when trying to match to "." as a character. Since this has a special meaning within regular expressions we need to escape it using a backslash. However the `\` character is an escape character in R strings too and so you need to use two: `\\.`.

2.1.2 Checking for Other Valid Values

We can also check for validity within our columns. Say we know the ID column has 2 letters, 3 numbers, and then ends in either A or B. We can see Dan actually has a bad id.

We can use the `str_view` function to see the regular expression match as we build up a more complex pattern.

```
# to create a regex like this - it can help to view it
# we know it ends in A or B
str_view(people_data$id, pattern = "[AB]$")
```

QO509A
EX981B
AG978B
JDJWLA
DE700B
RN382A
ZT720B

```
# we know it starts with two letters;
str_view(people_data$id, pattern = "^[A-Z][A-Z]")
```

QO509A
EX981B
AG978B
JDJWLA
DE700B
RN382A
ZT720B

```
# we can also write this like so;
str_view(people_data$id, pattern = "^[A-Z]{2}")
```

QO509A
EX981B
AG978B
JDJWLA
DE700B
RN382A
ZT720B

```
# we also know there are 3 numbers in the middle;
str_view(people_data$id, pattern = "[0-9]{3}")
```

QO509A
EX981B
AG978B
JDJWLA
DE700B
RN382A
ZT720B

```
# lets put this all together;
str_view(people_data$id, pattern = "^[A-Z]{2}[0-9]{3}[AB]$")
```

QO509A
EX981B
AG978B

JDJWLA

DE700B

RN382A

ZT720B

```
# so we can use this to filter out the bad row of data;
people_data %>%
  filter(str_detect(id,
                    pattern = "^[A-Z]{2}[0-9]{3}[AB]$"))
```

```
##   first_name second_name                          email     id amount
## 1       Abby       Jones     ajones@mango-solutions.com QO509A     23
## 2        Ben       Smith     bsmith@mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams@banana-solutions.com AG978B    £12
## 4       Ella      Davies     edavies@kiwi-solutions.com DE700B     18
## 5       Fran       Evans      fevans@kiwi-solutions.com RN382A    34r
## 6       Gary      Thomas     gthomas@kiwi-solutions.com ZT720B     26
```

```
# we've lost Dan!
```

Finally, it is worth mentioning the two tidyr functions `separate` and `unite`. These can be very useful in a range of text processing tasks.

```
library(tidyr)

people_data %>%
  separate(email, into = c("username", "domain"), sep = "@")
```

```
##   first_name second_name  username              domain     id amount
## 1       Abby       Jones    ajones  mango-solutions.com QO509A     23
## 2        Ben       Smith    bsmith  mango-solutions.com EX981B     14
## 3        Cam    Williams cwilliams banana-solutions.com AG978B    £12
## 4        Dan      Taylor   dtaylor banana-solutions.com JDJWLA   null
## 5       Ella      Davies   edavies   kiwi-solutions.com DE700B     18
## 6       Fran       Evans    fevans   kiwi-solutions.com RN382A    34r
## 7       Gary      Thomas   gthomas   kiwi-solutions.com ZT720B     26
```

```
people_data %>%
  select(first_name, second_name, id) %>%
  unite("full_name", first_name, second_name, sep = " ", remove = FALSE)
```

```
##      full_name first_name second_name     id
## 1   Abby Jones       Abby       Jones QO509A
## 2    Ben Smith        Ben       Smith EX981B
## 3 Cam Williams        Cam    Williams AG978B
## 4   Dan Taylor        Dan      Taylor JDJWLA
## 5  Ella Davies       Ella      Davies DE700B
## 6   Fran Evans       Fran       Evans RN382A
## 7  Gary Thomas       Gary      Thomas ZT720B
```

This is only scratches the surface of what regular expressions can do. There is far more to dive into, and we recommend you read further if you are interested.

1. Using the `first_name` column in people data set, select names that start with 'A' or 'B'
2. Similarly, select names that have exactly four letters?
3. Extract only the numeric characters from the string `"abcd2ef7gh9ij2klmn98op"`

Extension: 4. Think about how you could apply this to search a text for phone numbers, emails, or similar coded text? For example, consider: c("example.email@gmail.com", "07719 377 390", "User:123ABC", "SN14 0GB")

2.2 Tidying and Summarising Text Data

2.2.1 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 usable 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

2.2.2 Exploring our Data

Throughout this course, we will be using data which covers the star wars scripts, it has been provided in an `.rds` file and can be loaded as so;

```
star_wars_script <- readRDS("../data/star_wars_scripts.rds")

str(star_wars_script)
```

```
## 'data.frame':    2523 obs. of  10 variables:
##  $ line     : chr  "1" "2" "3" "4" ...
##  $ movie    : chr  "IV" "IV" "IV" "IV" ...
##  $ title    : chr  "A New Hope" "A New Hope" "A New Hope" "A New Hope"
...
##  $ character: chr  "THREEPIO" "THREEPIO" "THREEPIO" "THREEPIO" ...
##  $ dialogue : chr  "Did you hear that?  They've shut down the main
reactor.  We'll be destroyed for sure.  This is madness!" "We're doomed!"
"There'll be no escape for the Princess this time." "What's that?" ...
##  $ length   : int  103 13 49 12 104 66 34 30 135 35 ...
##  $ ncap     : int  4 1 2 1 1 4 3 2 4 1 ...
##  $ nexcl    : int  1 1 0 0 0 4 1 1 1 0 ...
##  $ nquest   : int  1 0 0 1 0 1 1 1 1 1 ...
##  $ nword    : int  20 3 10 3 17 13 6 6 28 7 ...
```

This contains lines from the scripts of 3 star wars movies, and 129 of the main characters;

```
star_wars_script %>%
  distinct(movie, title)
```

```
##   movie                  title
## 1    IV             A New Hope
## 2     V The Empire Strikes Back
## 3    VI      Return of the Jedi
```

```
star_wars_script %>%
  distinct(character) %>%
  head()
```

```
##          character
## 1         THREEPIO
## 2             LUKE
## 3 IMPERIAL OFFICER
## 4            VADER
## 5    REBEL OFFICER
## 6          TROOPER
```

The data also contains useful information surrounding the `dialogue` column of data, including the `length` number of characters in the string, the `ncap` number of capitalised letters, `nexcl`/`nquest` number of `!` or `?` and `nword` number of words.

Let's begin by just examining the first movie, and only keeping the key columns - line, character and dialogue;

```
episode_iv <- star_wars_script %>%
  filter(movie == "IV") %>%
  select(line, character, dialogue)
head(episode_iv)
```

```
##   line character
## 1    1  THREEPIO
## 2    2  THREEPIO
## 3    3  THREEPIO
## 4    4  THREEPIO
## 5    5  THREEPIO
## 6    6      LUKE
##
dialogue
## 1  Did you hear that?  They've shut down the main reactor.  We'll be
destroyed for sure.  This is madness!
## 2
We're doomed!
## 3                                                          There'll be no
escape for the Princess this time.
## 4
What's that?
## 5 I should have known better than to trust the logic of a half-sized
thermocapsulary dehousing assister...
## 6                                        Hurry up!  Come with me!  What
are you waiting for?!  Get in gear!
```

2.2.3 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 tokenize data by breaking out a string of text into individual tokens.

Using our episode iv data from the star wars scripts, 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 data into the `unnest_tokens` function. Further formatting arguments can be passed such as the units to tokenize by e.g. "words" or "sentences".

```
library(tidytext)
library(dplyr)

# convert text into tokenized format
tidy_episode_iv <- episode_iv %>% unnest_tokens(word, dialogue)
head(tidy_episode_iv)
```

```
##     line character    word
## 1      1  THREEPIO     did
## 1.1    1  THREEPIO     you
## 1.2    1  THREEPIO    hear
## 1.3    1  THREEPIO    that
## 1.4    1  THREEPIO they've
## 1.5    1  THREEPIO    shut
```

2.2.4 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".

```
tidy_episode_iv %>%
  count(word, sort = TRUE) %>%
  head()
```

```
## # A tibble: 6 x 2
##   word      n
##   <chr> <int>
## 1 the     413
## 2 you     359
## 3 to      289
## 4 i       287
## 5 a       219
## 6 of      157
```

2.2.5 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 Episode IV 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
```

```
tidy_episode_iv  %>%
  anti_join(stop_words, by = "word") %>%
  count(word, sort = TRUE) %>%
  head()
```

```
## # A tibble: 6 x 2
##   word      n
##   <chr> <int>
## 1 luke     63
## 2 sir      46
## 3 ship     32
## 4 red      26
## 5 time     26
## 6 artoo    22
```

2.2.6 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. For example we may wish to remove the character names, to do this, we can created a dataframe of these names and used in an anti-join to further refine the data we wish to analyse.

```
characters <- data.frame(word = c("leia", "darth", "vader", "obi", "wan",
  "kenobi",
                                  "han", "solo", "luke", "skywalker",
  "chewbacca",
                                  "r2", "d2", "artoo", "detoo"))
# they spell R2-D2 as artoo detoo too!

tidy_episode_iv_no_characters <-  tidy_episode_iv %>%
  anti_join(stop_words, by = "word") %>%
  anti_join(characters, by = "word") %>%
  count(word, sort = TRUE)
```

```
## Warning: Column `word` joining character vector and factor, coercing
into character vector
```

```
head(tidy_episode_iv_no_characters)
```

```
## # A tibble: 6 x 2
##   word         n
##   <chr>    <int>
## 1 sir         46
## 2 ship        32
## 3 red         26
## 4 time        26
## 5 force       22
## 6 alderaan    20
```

2.2.7 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"
```

```
tidy_episode_iv_stemmed <-  tidy_episode_iv %>%
  anti_join(stop_words) %>%
  mutate(word = wordStem(word)) %>%
  count(word, sort = TRUE)
```

```
## Joining, by = "word"
```

```
head(tidy_episode_iv_stemmed)
```

```
## # A tibble: 6 x 2
##   word      n
##   <chr> <int>
## 1 luke     63
## 2 sir      46
## 3 ship     38
## 4 time     30
## 5 droid    29
## 6 red      26
```

Exercise

1. Consider the `star_wars_script` text as a whole and tidy the book into the tidy one-token-per-row structure
2. Find the most common words in the dialogue of the films together
3. What are the most common words after removing stop words?
4. Stem this text - does this change the order of the most common words within the text?

2.3 Word Clouds

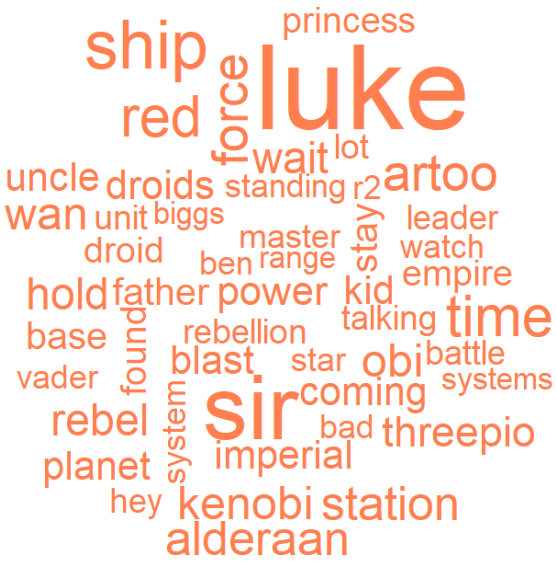
Word clouds are a very popular visualise the key words within text data. Now our data is in the correct format, a word cloud is very simple to produce. The `wordcloud` function takes two main arguments: the words to include in the image and a count of their frequency. We can then use some of the other 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

```
library(wordcloud)

cloud_episode_iv <-  episode_iv %>%
  unnest_tokens(word, dialogue)%>%
  anti_join(stop_words) %>%
  count(word, sort = TRUE)
```

```
## Joining, by = "word"
```

```
wordcloud(words = cloud_episode_iv$word,
          freq = cloud_episode_iv$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 characters 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 character counts for each group we want to compare. We will split our words into two documents, one for each character, and plot each together on a comparison cloud.

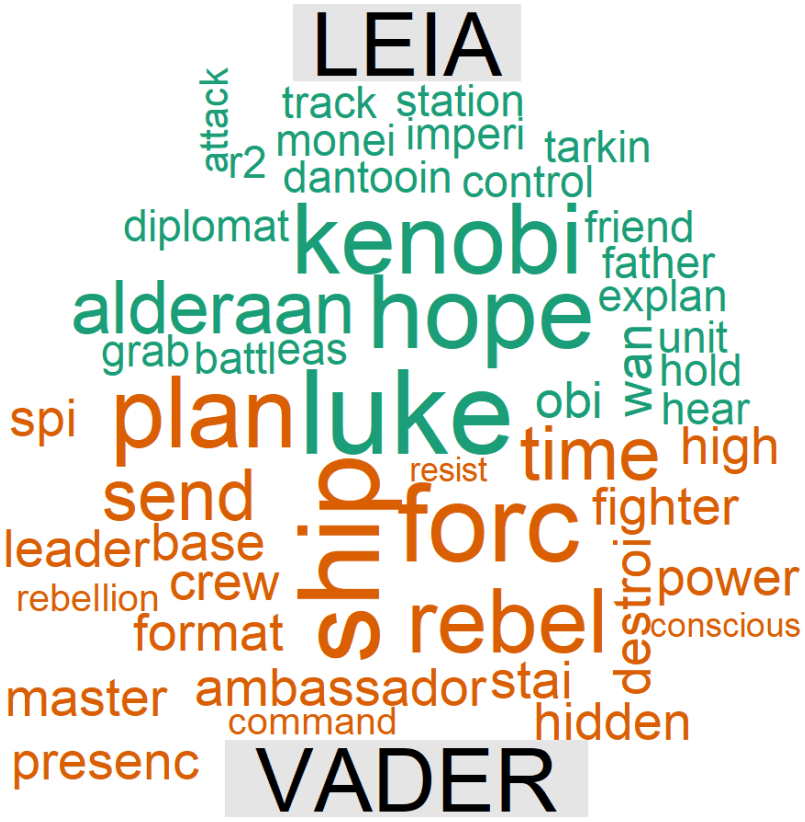
Unfortunately, the tidyverse dataframe, the tibble, does not support names rows, so we have to convert our data back into a base R dataframe.

```
# get data into the correct format
comp_cloud_episode_iv <- episode_iv %>%
  filter(character %in% c("LEIA", "VADER")) %>%
  unnest_tokens(word, dialogue) %>%
  anti_join(stop_words, by = "word") %>%
  mutate(word = wordStem(word)) %>%
  count(word, character, sort = TRUE) %>%
  pivot_wider(names_from = character,
              values_from = n,
              values_fill = list(n = 0)) %>%
  data.frame() # tibble row names are deprecated

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

head(comp_cloud_episode_iv)
```

```
##        LEIA VADER
## kenobi    7     1
## luke      7     0
## hope      6     0
## plan      2     6
## ship      1     6
## forc      0     5
```

```
# plot comparison cloud
comparison.cloud(comp_cloud_episode_iv,
                 max.words = 50)
```

```
## Warning in comparison.cloud(comp_cloud_episode_iv, max.words = 50):
transmiss could not be fit on page.
## It will not be plotted.
```



1. Make a word cloud for your favourite character from all the start wars text
2. Considering the separate movies - combine/compare the wordclouds for each one

2.4 "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.

```
# tokenize into bigrams
tidy_episode_iv_ngram <- episode_iv %>%
  unnest_tokens(word, dialogue, token = "ngrams", n = 2)

tidy_episode_iv_ngram  %>%
  count(word, sort = TRUE) %>%
  head()
```

```
## # A tibble: 6 x 2
##   word          n
##   <chr>     <int>
## 1 going to     44
## 2 <NA>         32
## 3 of the       31
## 4 all right    29
## 5 in the       29
## 6 are you      27
```

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
tidy_episode_iv_ngram <- episode_iv %>%
   unnest_tokens(word, dialogue, token = "ngrams", n = 2) %>%
   count(word, sort = TRUE) %>%
   separate(word, c("firstWord", "secondWord"), sep = " ")

head(tidy_episode_iv_ngram)
```

```
## # A tibble: 6 x 3
##   firstWord secondWord     n
##   <chr>     <chr>      <int>
## 1 going     to            44
## 2 <NA>      <NA>          32
## 3 of        the           31
## 4 all       right         29
## 5 in        the           29
## 6 are       you           27
```

```
# remove stop words - and NA's (when there's only one word in the line)
tidy_episode_iv_ngram <- tidy_episode_iv_ngram %>%
   anti_join(stop_words, by = c("firstWord" = "word")) %>%
   anti_join(stop_words, by = c("secondWord" = "word")) %>%
   drop_na()

head(tidy_episode_iv_ngram)
```

```
## # A tibble: 6 x 3
##   firstWord secondWord     n
##   <chr>     <chr>      <int>
## 1 obi       wan           19
## 2 wan       kenobi        12
## 3 r2        unit          11
## 4 rebel     base          11
## 5 battle    station       10
## 6 sand      people        10
```

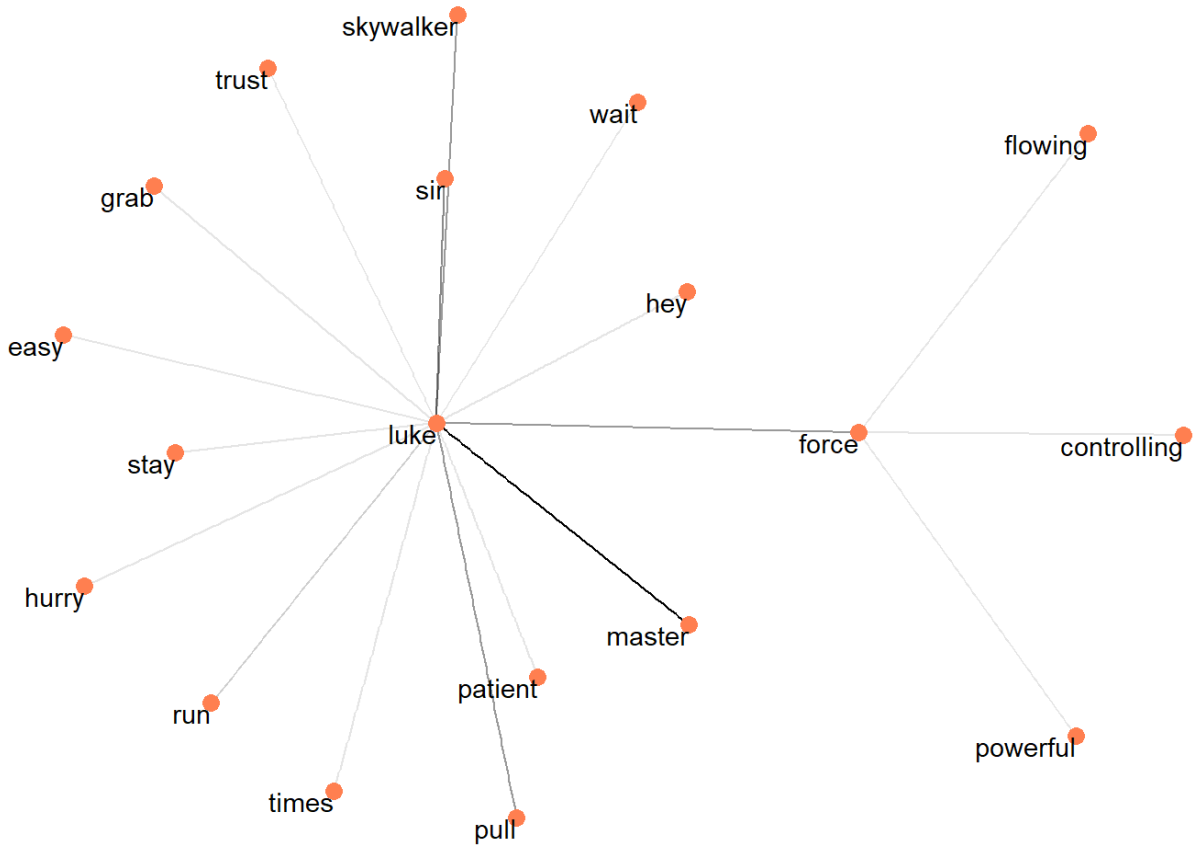
2.5 Visualising Related Words

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 one of the characters (`luke`) and a key word (`force`) is visualised.

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

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

# Remove stop words and NA's
tidy_episode_iv_ngram <- tidy_episode_iv_ngram %>%
   anti_join(stop_words, by = c("firstWord" = "word")) %>%
   anti_join(stop_words, by = c("secondWord" = "word")) %>%
   drop_na()

# Filter so only common pairs about luke or force remain
tidy_episode_iv_ngram <- tidy_episode_iv_ngram %>%
  filter((firstWord  %in% c("luke", "force")) |
           secondWord %in% c("luke", "force"))


head(tidy_episode_iv_ngram)
```

```
## # A tibble: 6 x 3
##   firstWord secondWord     n
##   <chr>     <chr>      <int>
## 1 luke      luke           9
## 2 master    luke           4
## 3 force     luke           2
## 4 luke      pull           2
## 5 luke      skywalker      2
## 6 sir       luke           2
```

```
# Create the igraph object
igraph_episode_iv <-  graph_from_data_frame(tidy_episode_iv_ngram)

# Plot the ggraph
ggraph(igraph_episode_iv, 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()
```



1. Tokenize the whole star wars data into word pairs (bigrams)
2. Which words co-occur most frequently?
3. Re-create the ggraph for `luke` and `force` - and examine the differences

Extension:

1. Re-tokenize the whole star wars data into word triplets (trigrams) - now 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 - again we will use episode iv from the star wars data;

```
# import the data - and filter
star_wars_script <- readRDS("../data/star_wars_scripts.rds")

episode_iv <- star_wars_script %>%
  filter(movie == "IV") %>%
  select(line, character, dialogue)

# Make tidy data
tidy_episode_iv <- episode_iv %>%
   unnest_tokens(word, dialogue) %>%
   anti_join(stop_words) %>%
   count(word, sort = TRUE)
```

```
## Joining, by = "word"
```

```
head(tidy_episode_iv)
```

```
## # A tibble: 6 x 2
##   word      n
##   <chr> <int>
## 1 luke     63
## 2 sir      46
## 3 ship     32
## 4 red      26
## 5 time     26
## 6 artoo    22
```

`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
```

```
tidy_episode_iv %>%
  inner_join(sentiment, by="word") %>%
  head()
```

```
## # A tibble: 6 x 3
##   word        n sentiment
##   <chr>   <int> <chr>
## 1 bad        11 negative
## 2 master     11 positive
## 3 worry      10 negative
## 4 attack      9 negative
## 5 trouble     8 negative
## 6 afraid      7 negative
```

From here, it is possible to analyse if star wars episode iv is overall positive or negative.

```
tidy_episode_iv %>%
  inner_join(sentiment, by="word") %>%
  group_by(sentiment) %>%
  summarise(total = sum(n))
```

```
## # A tibble: 2 x 2
##   sentiment total
##   <chr>     <int>
## 1 negative    314
## 2 positive    183
```

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.

We can compare the AFINN sentiment score for each of the main characters;

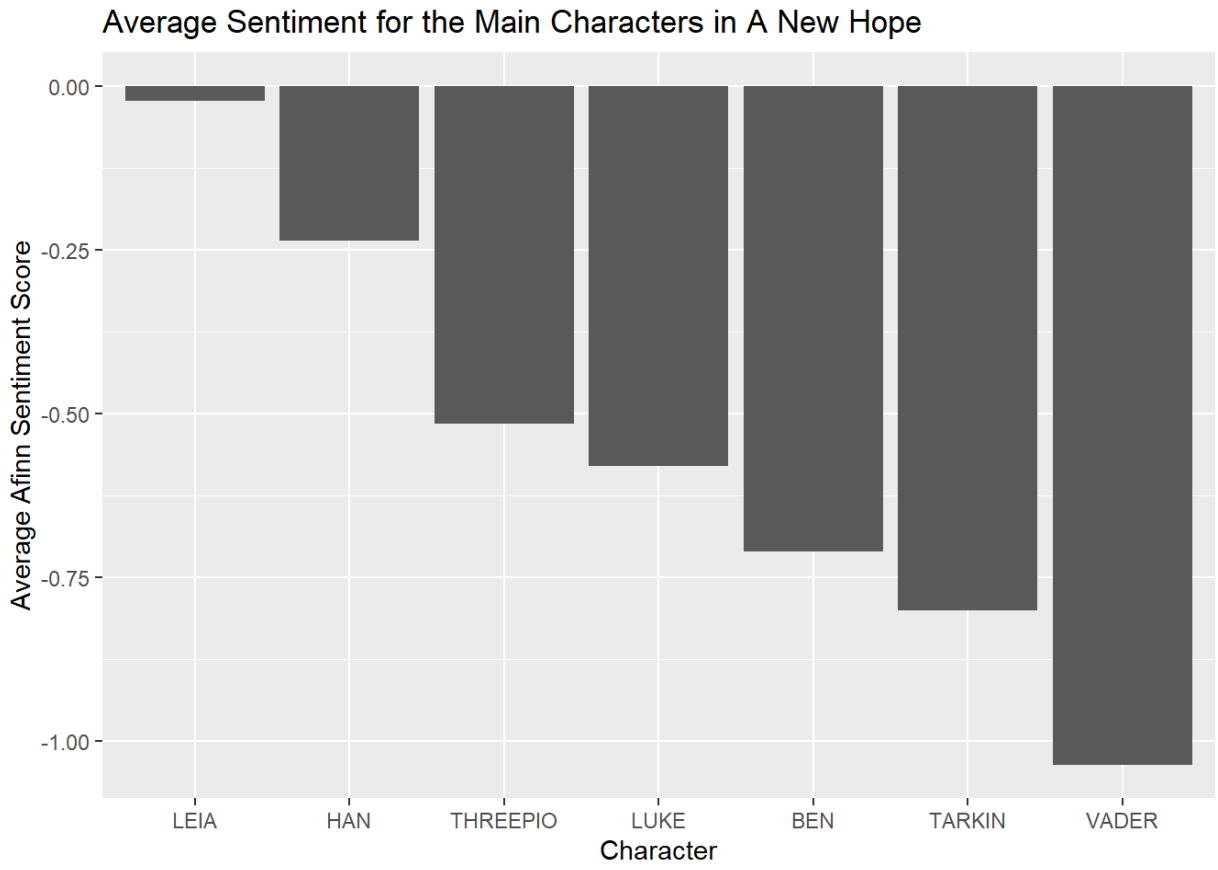
```
library(tidyverse)

# get the data into the correct form
afinn_episode_iv <- episode_iv  %>%
  unnest_tokens(word, dialogue) %>%
  anti_join(stop_words, by = "word") %>%
  inner_join(get_sentiments("afinn"), by = "word")

head(afinn_episode_iv)
```

```
##   line        character      word value
## 1    1         THREEPIO destroyed    -3
## 2    1         THREEPIO   madness    -3
## 3    2         THREEPIO    doomed    -2
## 4    3         THREEPIO    escape    -1
## 5    5         THREEPIO     trust     1
## 6   11 IMPERIAL OFFICER     death    -2
```

```
afinn_episode_iv_character_summary <- afinn_episode_iv %>%
  group_by(character) %>%
  summarise(score = mean(value),
            n = n()) %>%
  filter(n > 20)

head(afinn_episode_iv_character_summary)
```

```
## # A tibble: 6 x 3
##   character   score     n
##   <chr>       <dbl> <int>
## 1 BEN       -0.711     45
## 2 HAN       -0.236     89
## 3 LEIA      -0.0233    43
## 4 LUKE      -0.581     93
## 5 TARKIN    -0.8       30
## 6 THREEPIO  -0.516     64
```

```
ggplot(afinn_episode_iv_character_summary,
       aes(x = reorder(character, -score), y = score)) +
  geom_col() +
  labs(title = "Average Sentiment for the Main Characters in A New Hope",
       x = "Character", y = "Average Afinn Sentiment Score")
```



3.2.1 Sentiment Progression

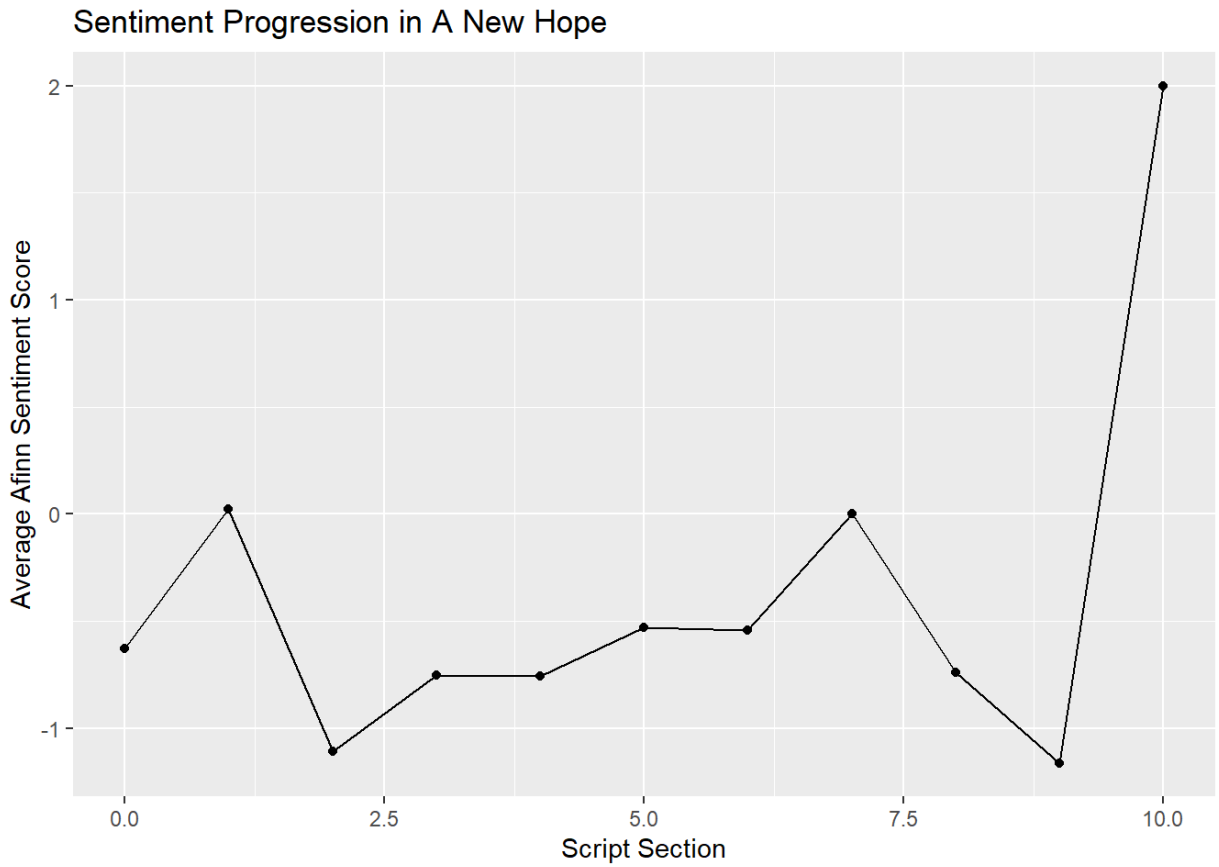
In the next example we can look at how the sentiment changes throughout the star wars film.

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

```
# split into sections of 100 lines and summarise
# make sure the line number is numeric
afinn_episode_iv_summary <- afinn_episode_iv %>%
  mutate(line = as.numeric(line)) %>%
  mutate(scriptsection = line %/% 100) %>%
  group_by(scriptsection) %>%
  summarise(score = mean(value))

head(afinn_episode_iv_summary)
```

```
## # A tibble: 6 x 2
##   scriptsection   score
##           <dbl>   <dbl>
## 1             0 -0.629
## 2             1  0.0233
## 3             2 -1.11
## 4             3 -0.75
## 5             4 -0.754
## 6             5 -0.531
```

```
# plot the sentiment though the book
ggplot(afinn_episode_iv_summary , aes(scriptsection,  score)) +
  geom_line()  +
  geom_point() +
  labs(title = "Sentiment Progression in A New Hope",
       x = "Script Section", y = "Average Afinn Sentiment Score")
```



1. What are the most frequently expressed positive words in all of the star wars films?
2. Are the star wars scripts mostly positive or negative?
3. How does sentiment change throughout the star wars movies? *hint: compare movies using group by* `movie` *and* `scriptsection` *when summarising*

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", etc. 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 document, 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 documents.

4.2 Applying TF-IDF

Here, we will continue to use the star wars data as our example, but to generate several documents we will split the script into several sections. The `%/%` operator is used for integer division - this allows us to split the text into sections of 200 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 data and filter
star_wars_script <- readRDS("../data/star_wars_scripts.rds")
episode_iv <- star_wars_script %>%
  filter(movie == "IV") %>%
  select(line, character, dialogue)

# make into tidy form and add line numbers
tidy_episode_iv <- episode_iv %>%
   mutate(line = as.numeric(line)) %>%
   unnest_tokens(word, dialogue)

# label into sections by line number
tidy_split_episode_iv <- tidy_episode_iv %>%
  mutate(scriptsection = line %/% 200) # split every 200 lines

# add document-term count
tidy_split_episode_iv <- tidy_split_episode_iv %>%
  group_by(word, scriptsection) %>%
  summarise(count = n()) %>%
  ungroup() %>%
  arrange(-count)

head(tidy_split_episode_iv, 8)
```

```
## # A tibble: 8 x 3
##   word  scriptsection count
##   <chr>         <dbl> <int>
## 1 the               1   119
## 2 you               1    90
## 3 you               0    89
## 4 the               2    87
## 5 i                 0    86
## 6 the               0    85
## 7 to                1    79
## 8 to                0    78
```

```
# apply TF-IDF
tidy_split_episode_iv <- tidy_split_episode_iv %>%
  bind_tf_idf(term = word, document = scriptsection, n = count)

head(tidy_split_episode_iv, 8)
```

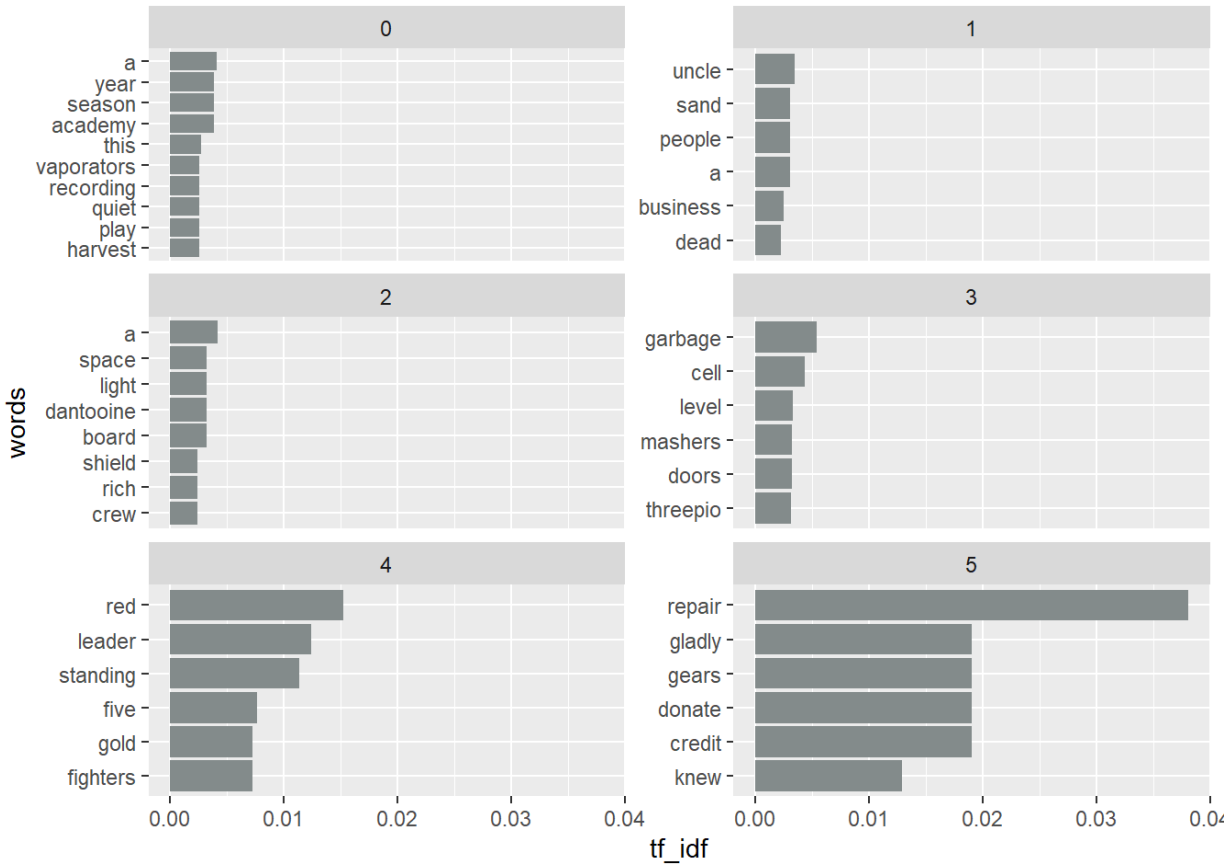
```
## # A tibble: 8 x 6
##   word  scriptsection count     tf   idf tf_idf
##   <chr>         <dbl> <int>  <dbl> <dbl>  <dbl>
## 1 the               1   119 0.0417     0      0
## 2 you               1    90 0.0316     0      0
## 3 you               0    89 0.0319     0      0
## 4 the               2    87 0.0394     0      0
## 5 i                 0    86 0.0308     0      0
## 6 the               0    85 0.0304     0      0
## 7 to                1    79 0.0277     0      0
## 8 to                0    78 0.0279     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 script section;

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



1. Split the whole star wars scripts 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 star wars scripts, by script section
3. Visualise your results

Extension:

1. After splitting the data into sections by line number, lets also consider the movie for that line, create a new column; `movie_scriptsection` which utilises `unite` to join the data in the two columns. Complete the TF-IDF analysis and visualise your results