Introduction to Data Extraction: Lab 4 Solutions

Simon Caton

Saving and loading R objects

It can often be helpful to store R objects. When preparing each lab pdf I essentially have a clean environment, as the code is executed as the PDF is generated. This is something called **literate programming** and it allows me to combine the lab exercise with the code that is needed to run the lab. This not only saves a lot of time! There is no need to take screenshots or worry about how to format the code, output etc. it also means that the labs are reproducible, i.e. I know that the code runs, and can check the output, adding a level of testing and quality control to the lab worksheets. The labs are created using something called R Markdown see https://jabranham.com/categories/rmarkdown/ for a brief intro to this.

Now a consequence of doing this is that each lab is a seperate environment, so I cannot pass R objects between the PDFs that are produced. In lab 4 we have some complex objects being produced, and obviously I don't want to embed all the code into this answers PDF to reproduce them. Instead there is a small amount of hidden code (you can't see it – it's executed, but the results are not displayed, echoed in RMarkdown terminology, in the produced PDF) that saves the key R objects to disk that are needed for the exercises:

```r
save(GoTWikipedia, file="GoTWikipedia.RData")
save(GoTJsonSeason1, file="GoTJsonSeason1.RData")
save(GoTJsonSeason2, file="GoTJsonSeason2.RData")
save(GoTJsonSeason3, file="GoTJsonSeason3.RData")
save(GoTJsonSeason4, file="GoTJsonSeason4.RData")
save(GoTJsonSeason5, file="GoTJsonSeason5.RData")
save(GoTJsonSeason6, file="GoTJsonSeason6.RData")
save(GoTJsonSeason7, file="GoTJsonSeason7.RData")
```

Now in order to prepare the answers to the exercises, all that is needed is to read these objects back in:

```r
load("GoTWikipedia.RData")
load("GoTJsonSeason1.RData")
load("GoTJsonSeason2.RData")
load("GoTJsonSeason3.RData")
load("GoTJsonSeason4.RData")
load("GoTJsonSeason5.RData")
load("GoTJsonSeason6.RData")
load("GoTJsonSeason7.RData")

str(GoTWikipedia)
```

```
## 'data.frame':    7 obs. of  16 variables:
##  $ Season                              : Factor w/ 7 levels "1","2","3","4",..: 1 2 3 4 5 6 7
##  $ Ordered                             : chr  "March 2, 2010" "April 19, 2011" "April 10, 2012" "Apri
##  $ Filming                             : chr  "Second half of 2010" "Second half of 2011" "July - Nov
##  $ First.aired                         : chr  "April 17, 2011" "April 1, 2012" "March 31, 2013" "Apri
##  $ Last.aired                          : chr  "June 19, 2011" "June 3, 2012" "June 9, 2013" "June 15
##  $ Novel.s..adapted                    : chr  "A Game of Thrones" "A Clash of Kings and some early ch
##  $ Critical.response....Rotten.Tomatoes: chr  "89% (33 reviews)" "96% (33 reviews)" "97% (44 reviews)
##  $ Critical.response....Metacritic     : chr  "80 (28 reviews)" "90 (26 reviews)" "91 (25 reviews)"
##  $ averages                            : num  78 80 80 82 81 79 86
##  $ movieDBRatings                      : num  78 80 80 82 81 79 86
##  $ Critic                              : num  80 90 91 94 91 73 77
##  $ RT                                  : num  89 96 97 97 95 96 96
##  $ maleExtras                          : int  122 118 132 141 128 107 39
##  $ femaleExtras                        : int  34 42 38 37 38 40 8
##  $ unknownGenderExtras                 : int  104 87 74 88 107 84 14
##  $ totalExtras                         : int  260 247 244 266 273 231 61
```

Note that we don't need to decalre a variable name for our objects. This, as well as a lot of other metadata, was retained when we saved them. Why is this useful? Well if ever you need to backup parts of your environment, or take a snapshot of an object this is a really useful way of doing that.

Question 1: Plot gender distributions by season

The last task sort of did this, but it did so to the extent of plotting totals. Here we want more granularity, i.e. non-aggregate values.

Let's get the genders of extras according to season, and episodes. For this, we need to combine our getGender function with the code that computes the totals, to get an episode by episode break down of the gender of extras.

```r
getGender <- function(seasonJSON) {

  gender <- list()
  #this time we're making a list, not a vector

  for (i in 1:length(seasonJSON)) {
    gender[[i]] <- seasonJSON[[i]]$gender
  }

  gender.male <- unlist(lapply(gender, FUN = function(x) {
    length(x[x == 2])
  }))

  gender.female <- unlist(lapply(gender, FUN = function(x) {
    length(x[x == 1])
  }))

  gender.unknown <- unlist(lapply(gender, FUN = function(x) {
    length(x[x == 0])
  }))

  gender.total <- unlist(lapply(gender, FUN = function(x) {
    length(x)
  }))

  gender <- data.frame(gender.male, gender.female, gender.unknown, gender.total)
  #and we want to return a dataframe

  return(gender)
}
```

So now, when we do this for each Season:

```r
s1genderByEpisode <- getGender(GoTJsonSeason1$episodes$guest_stars)
s2genderByEpisode <- getGender(GoTJsonSeason2$episodes$guest_stars)
s3genderByEpisode <- getGender(GoTJsonSeason3$episodes$guest_stars)
```

```r
s4genderByEpisode <- getGender(GoTJsonSeason4$episodes$guest_stars)
s5genderByEpisode <- getGender(GoTJsonSeason5$episodes$guest_stars)
s6genderByEpisode <- getGender(GoTJsonSeason6$episodes$guest_stars)
s7genderByEpisode <- getGender(GoTJsonSeason7$episodes$guest_stars)
```

To give an indication of what we have:

```r
s1genderByEpisode
```

```
##    gender.male gender.female gender.unknown gender.total
## 1           12             2              5           19
## 2            6             3              8           17
## 3           13             2              9           24
## 4           13             5             12           30
## 5           10             2             13           25
## 6            9             5             13           27
## 7           13             4             11           28
## 8           17             4             12           33
## 9           12             3             12           27
## 10          17             4              9           30
```

We can combine each of our gender data.frames, but we will need to capture the Season they belong to. In this instance, we don't really need to record the episode number, but we'll add it anyway, just to show how we could do it – we know that the episodes are added in order, so we can just add a counter to capture them from 1 to however many there are, i.e. how many rows (nrow) our data.frame has.

```r
s1genderByEpisode$Season <- 1
s1genderByEpisode$Episode <- c(1:nrow(s1genderByEpisode))

s2genderByEpisode$Season <- 2
s2genderByEpisode$Episode <- c(1:nrow(s2genderByEpisode))

s3genderByEpisode$Season <- 3
s3genderByEpisode$Episode <- c(1:nrow(s3genderByEpisode))

s4genderByEpisode$Season <- 4
s4genderByEpisode$Episode <- c(1:nrow(s4genderByEpisode))

s5genderByEpisode$Season <- 5
s5genderByEpisode$Episode <- c(1:nrow(s5genderByEpisode))

s6genderByEpisode$Season <- 6
s6genderByEpisode$Episode <- c(1:nrow(s6genderByEpisode))

s7genderByEpisode$Season <- 7
s7genderByEpisode$Episode <- c(1:nrow(s7genderByEpisode))
```

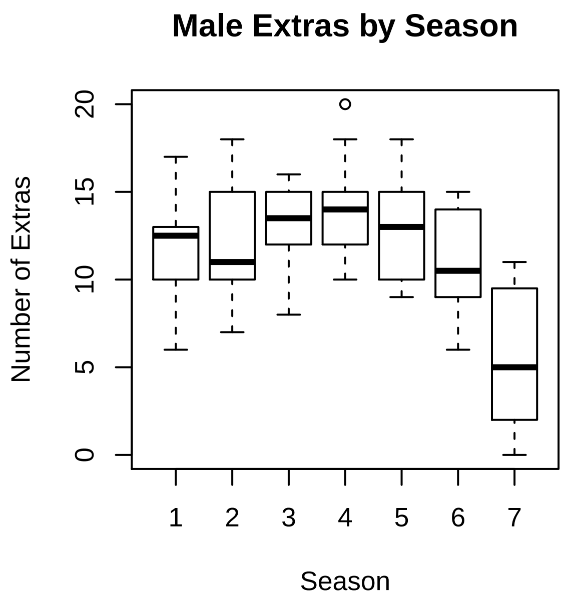
Now we combine our seasons together.

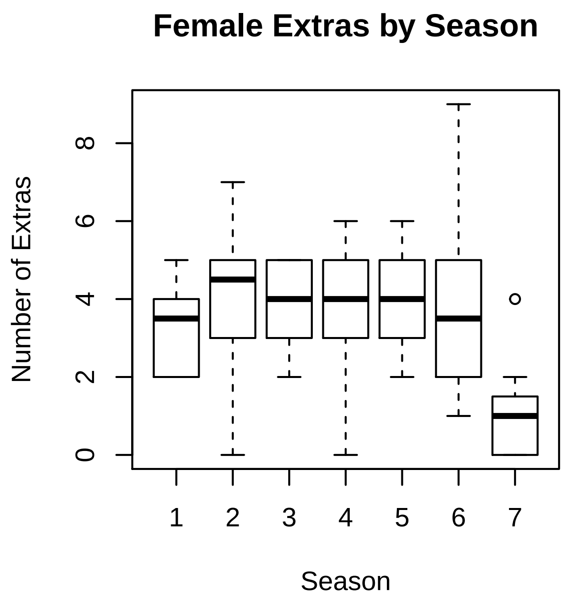
```r
gender <- rbind(s1genderByEpisode, s2genderByEpisode, s3genderByEpisode,
                     s4genderByEpisode, s5genderByEpisode, s6genderByEpisode,
                     s7genderByEpisode)
```

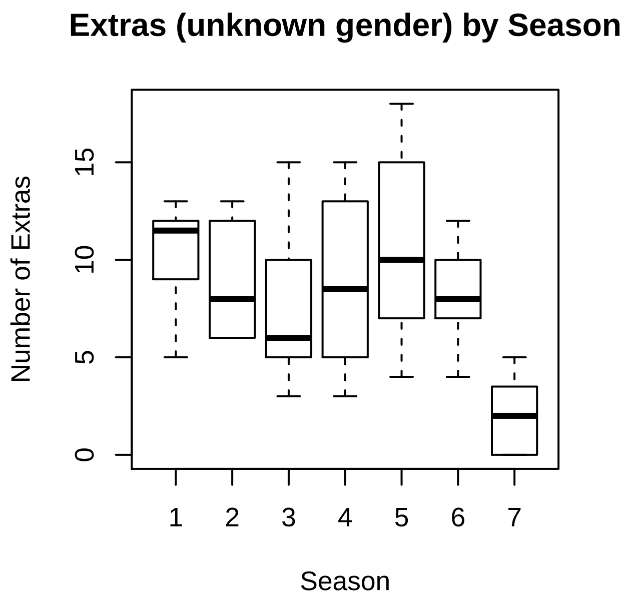
Any make a simple boxplot for the gender(s) we are interested in.

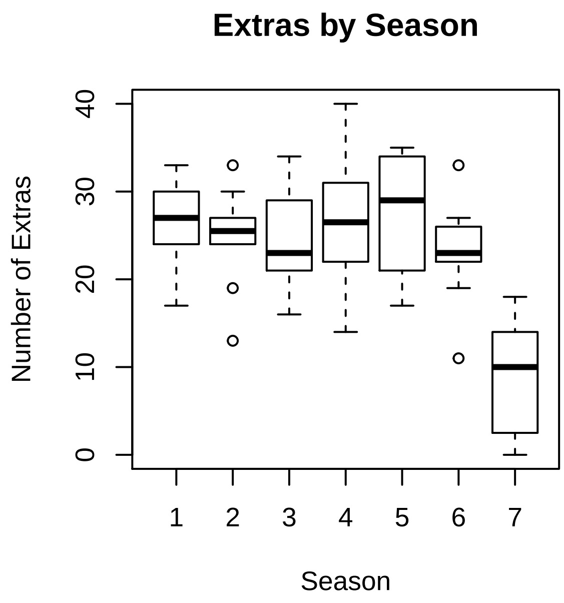
```r
 # allows us to put plots into a 2 x 2 grid -- makes it easier to compare
par(mfrow=c(2,2))
```

```r
boxplot(gender$gender.male ~ gender$Season, xlab="Season", ylab="Number of Extras",
        main="Male Extras by Season")
boxplot(gender$gender.female ~ gender$Season, xlab="Season", ylab="Number of Extras",
        main="Female Extras by Season")
boxplot(gender$gender.unknown ~ gender$Season, xlab="Season", ylab="Number of Extras",
        main="Extras (unknown gender) by Season")
boxplot(gender$gender.total ~ gender$Season, xlab="Season", ylab="Number of Extras",
        main="Extras by Season")
```





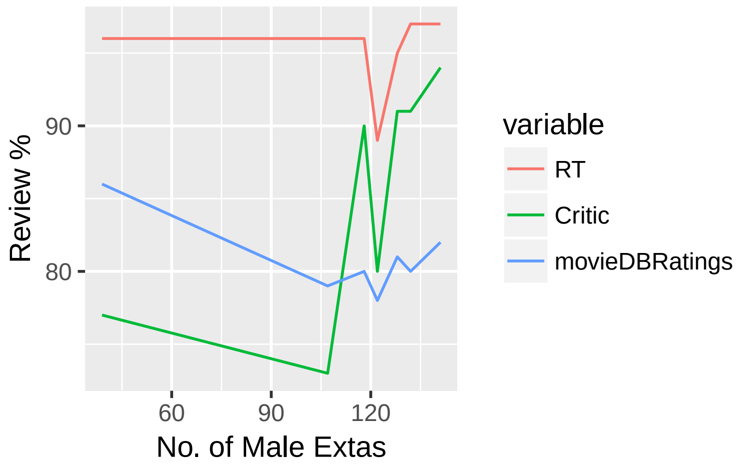




Note: the box plots capture data at the episode level.

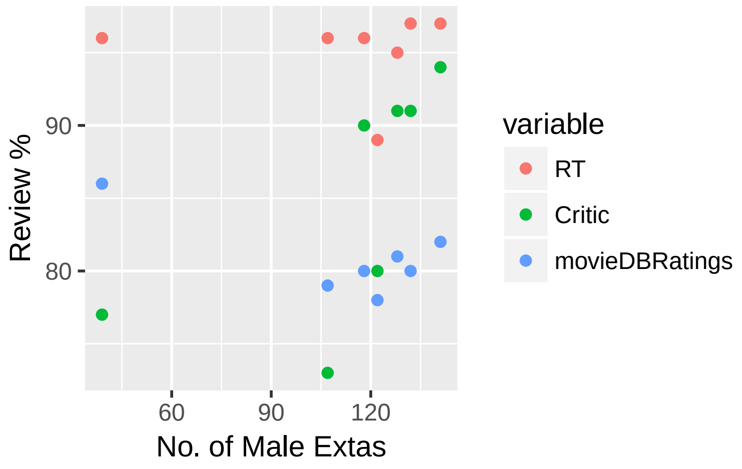
Question 2: Is there a relationship between the number of (fe)male extras and the Season ratings?

```r
library(ggplot2)
library(ggthemes)
library(reshape)
data <- data.frame(GoTWikipedia[, c("RT", "Critic", "movieDBRatings", "maleExtras")])
Molten <- melt(data, id.vars = "maleExtras")
ggplot(Molten, aes(x = maleExtras, y = value, colour = variable)) + geom_line() +
  ylab("Review %") + xlab("No. of Male Extas")
```



We have to be really really careful with a plot like this! The lines in the plot deceive us somewhat. Here is the same plot, without connecting the points.

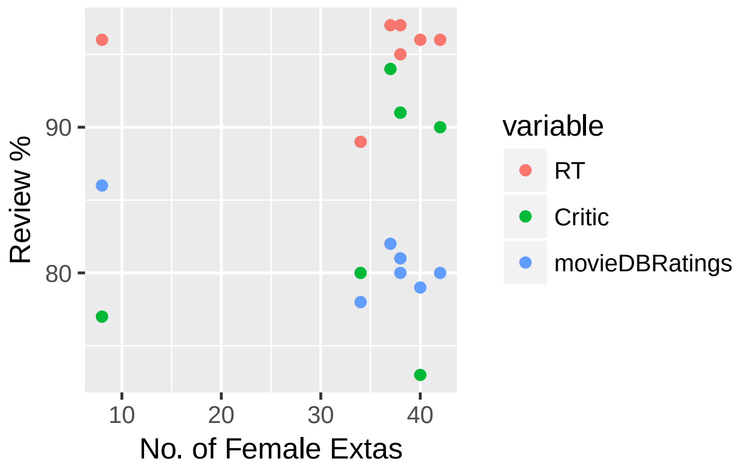
```r
ggplot(Molten, aes(x = maleExtras, y = value, colour = variable)) + geom_point() +
  ylab("Review %") + xlab("No. of Male Extas")
```



So we can see that the somewhat outlier point at the low end (left) of the scale gives us the impression that there are other points that may "sit" on the line. This essentially corresponds to a small dataset problem; so we have to be quite careful, i.e. conservative, with respect to how we interpret our data. Intutitively, we can probably say that the number of male extras doesn't seem to influence the RT scores. There appears to potentially be a negative relationship between the number of male extras and the reviewer score for the movie DB reviewers, and it appear to potentially be a positive relationship between critics scores and the number of male extras.

Repeating the same for female extras; RT and the movieDB interpretations appear unchanged, but the critics are all over the place now, thus there probably isn't a relationship between the number of female extras and the crictics review scores:

```r
data <- data.frame(GoTWikipedia[, c("RT", "Critic", "movieDBRatings", "femaleExtras")])
Molten <- melt(data, id.vars = "femaleExtras")
ggplot(Molten, aes(x = femaleExtras, y = value, colour = variable)) + geom_point() +
  ylab("Review %") + xlab("No. of Female Extas")
```



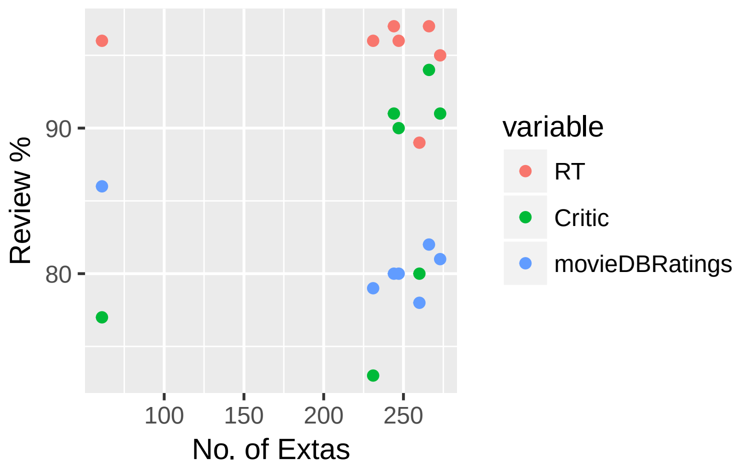
Note that we have not specified the direction of any relationships, i.e. noting a potentially postive relationship is different to saying that the number of extras positively influences the review scores; for all we know there could be some other effect that we are not observing, and we have not performed any formal statistical methods to try and substantiate our qualitative observations.

Going further than this in our interpretation would be beyond the scope of this module, as elements of statistical modelling are needed to refine our interpretation of the data.

Question 3: Do more extras tend to result in better ratings?

Ok so we are repeating to some extent the analysis from the previous question here, but looking at a more generalised view of the data.

```r
data <- data.frame(GoTWikipedia[, c("RT", "Critic", "movieDBRatings", "totalExtras")])
Molten <- melt(data, id.vars = "totalExtras")
ggplot(Molten, aes(x = totalExtras, y = value, colour = variable)) + geom_point() +
  ylab("Review %") + xlab("No. of Extas")
```



Not too surprisingly, we see a similar outcome. In the previous question, we noted that there isn't really enough data. Yet, we do actually have theMovieDB reviews (average scores) at the level of episode. So let's get them and add them to the results of our getGender function. First let's look at just getting reviews for Season one:

```
GoTJsonSeason1$episodes$vote_average
```

```
##  [1] 7.443 6.806 7.694 7.438 8.311 7.794 8.015 7.574 8.614 8.343
```

Luckily this is easier than it could have been :)

So if we modify our getGender function a little (we need now to pass it more of the data.frame to get both guest_stars and votes)

```
getGenderAndReviews <- function(seasonJSON) {

  gender <- list()
  #this time we're making a list, not a vector

  ####################
  #      added       #
  ####################
  gueststars <- seasonJSON$guest_stars

  for (i in 1:length(gueststars)) {
    gender[[i]] <- gueststars[[i]]$gender
  }

  gender.male <- unlist(lapply(gender, FUN = function(x) {
    length(x[x == 2])
  }))

  gender.female <- unlist(lapply(gender, FUN = function(x) {
    length(x[x == 1])
  }))

  gender.unknown <- unlist(lapply(gender, FUN = function(x) {
    length(x[x == 0])
  }))

  gender.total <- unlist(lapply(gender, FUN = function(x) {
    length(x)
  }))

  ####################
  #      added       #
  ####################
  votes <- seasonJSON$vote_average

  genderAndReviews <- data.frame(gender.male, gender.female, gender.unknown,
                                 gender.total, votes)
  #and we want to return a dataframe

  return(genderAndReviews)
}
```

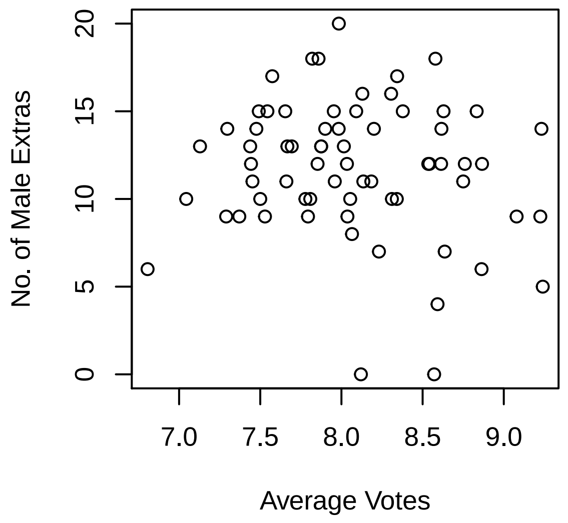
Ok, so now we can use our function in a similar way to before.

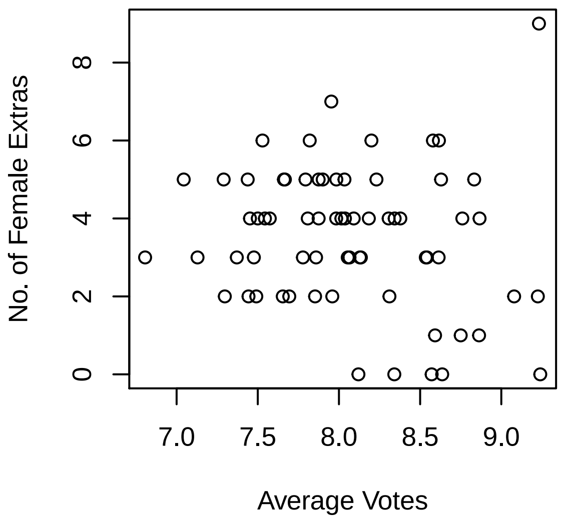
```
s1genderAndReviews <- getGenderAndReviews(GoTJsonSeason1$episodes)
s1genderAndReviews$Season <- 1
s1genderAndReviews$Episode <- c(1:nrow(s1genderByEpisode))

s2genderAndReviews <- getGenderAndReviews(GoTJsonSeason2$episodes)
s2genderAndReviews$Season <- 2
s2genderAndReviews$Episode <- c(1:nrow(s2genderByEpisode))

s3genderAndReviews <- getGenderAndReviews(GoTJsonSeason3$episodes)
s3genderAndReviews$Season <- 3
s3genderAndReviews$Episode <- c(1:nrow(s3genderByEpisode))

s4genderAndReviews <- getGenderAndReviews(GoTJsonSeason4$episodes)
s4genderAndReviews$Season <- 4
s4genderAndReviews$Episode <- c(1:nrow(s4genderByEpisode))

s5genderAndReviews <- getGenderAndReviews(GoTJsonSeason5$episodes)
s5genderAndReviews$Season <- 5
s5genderAndReviews$Episode <- c(1:nrow(s5genderByEpisode))

s6genderAndReviews <- getGenderAndReviews(GoTJsonSeason6$episodes)
s6genderAndReviews$Season <- 6
s6genderAndReviews$Episode <- c(1:nrow(s6genderByEpisode))

s7genderAndReviews <- getGenderAndReviews(GoTJsonSeason7$episodes)
s7genderAndReviews$Season <- 7
s7genderAndReviews$Episode <- c(1:nrow(s7genderByEpisode))

genderAndReviews <- rbind(s1genderAndReviews, s2genderAndReviews, s3genderAndReviews,
                          s4genderAndReviews, s5genderAndReviews, s6genderAndReviews,
                          s7genderAndReviews)
```

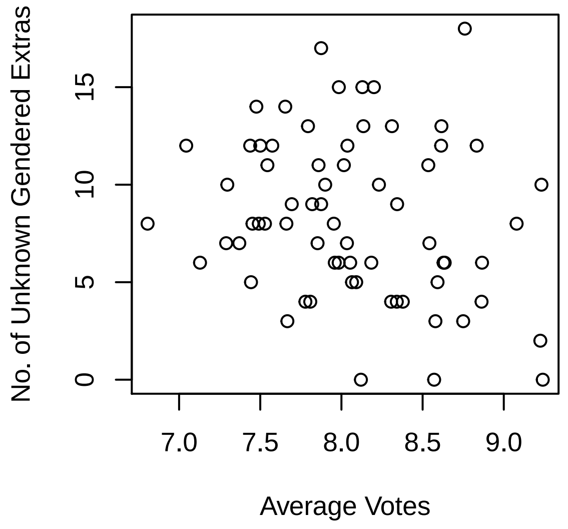
So now we can do any combination of gender plotted against theMovieDB reviews. We don't need to melt the data this time, as we already have the granularity we need. This time, let's just keep it simple and use *plot*.

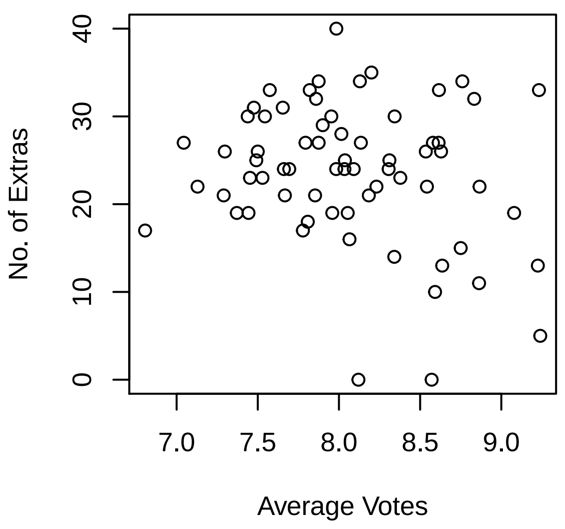
```
par(mfrow=c(2,2))
plot(genderAndReviews$votes, genderAndReviews$gender.male,
     xlab="Average Votes", ylab="No. of Male Extras")
```

```r
plot(genderAndReviews$votes, genderAndReviews$gender.female,
     xlab="Average Votes", ylab="No. of Female Extras")
plot(genderAndReviews$votes, genderAndReviews$gender.unknown,
     xlab="Average Votes", ylab="No. of Unknown Gendered Extras")
plot(genderAndReviews$votes, genderAndReviews$gender.total,
     xlab="Average Votes", ylab="No. of Extras")
```



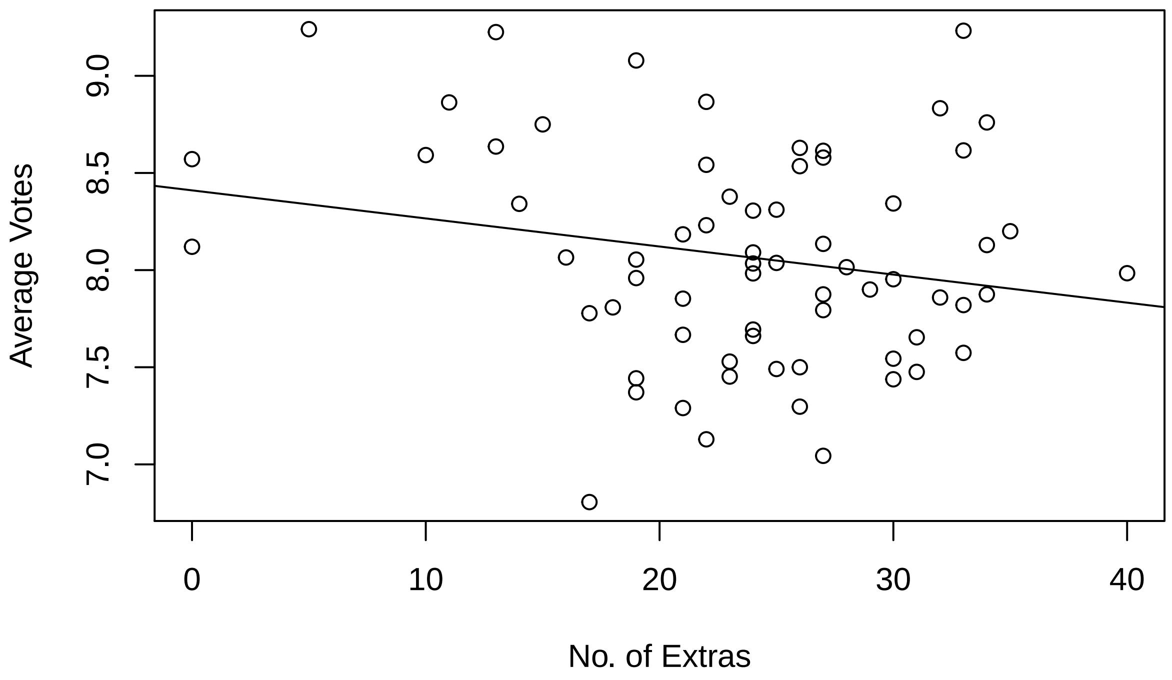






For this question we are only "interested" in the total number of extras. This dataset is a little harder to interpret. If we isolate only the last graph for now, and add a regression line to it:

```r
plot(genderAndReviews$gender.total, genderAndReviews$votes,
     xlab="No. of Extras", ylab="Average Votes")
lmModel <- lm(genderAndReviews$votes ~ genderAndReviews$gender.total)
abline(lmModel)
```



We can see a potential tendancy for a negative relationsip between the number of extras and theMovieDB average vote. It is slightly beyond the scope of DAD to interrogate this model. So simply put, the line we have added whilst showing a general tendancy is not a good fit for our data: there are not many points near the line. This can mean a few things:

- a linear model may not be a good approximation for the data we have,
- their may not really be a relationship between the number of extras per episode and the corresponding reviews received,
- we (still) don't have enough observations,
- we need to use more attibutes in order to predict review scores,
- etc.

Question 4: What are the most common words in season / episode overviews?

Ok, so before we can do anything, we need to collect all of the text.

Getting the Season overviews, is quite straightforward:

```
sOverviews <- c(
  GoTJsonSeason1$overview,
  GoTJsonSeason2$overview,
  GoTJsonSeason3$overview,
  GoTJsonSeason4$overview,
  GoTJsonSeason5$overview,
  GoTJsonSeason6$overview,
  GoTJsonSeason7$overview
)
```

Now, for the episodes.

```r
eOverviews <- c(
  GoTJsonSeason1$episodes$overview,
  GoTJsonSeason2$episodes$overview,
  GoTJsonSeason3$episodes$overview,
  GoTJsonSeason4$episodes$overview,
  GoTJsonSeason5$episodes$overview,
  GoTJsonSeason6$episodes$overview,
  GoTJsonSeason7$episodes$overview
)
```

What we need to do now is covert our vector of strings into an R data type we haven't seen yet: a Corpus. A corpus is a data structure for handling text documents from the *tm* package (where tm stands for Text Mining). The VectorSource is just a means of interpreting each element in our Vector as a document.

```r
library(tm)
seasonOverviews <- Corpus(VectorSource(sOverviews))
episodeOverviews <- Corpus(VectorSource(eOverviews))
```

To inspect what we have:

```r
summary(seasonOverviews)
```

```
##   Length Class             Mode
## 1 2      PlainTextDocument list
## 2 2      PlainTextDocument list
## 3 2      PlainTextDocument list
## 4 2      PlainTextDocument list
## 5 2      PlainTextDocument list
## 6 2      PlainTextDocument list
## 7 2      PlainTextDocument list
```

```r
inspect(seasonOverviews)
```

```
## <<SimpleCorpus>>
## Metadata:  corpus specific: 1, document level (indexed): 0
## Content:  documents: 7
##
## [1] Trouble is brewing in the Seven Kingdoms of Westeros. For the driven inhabitants of this visiona
## [2] The cold winds of winter are rising in Westeros...war is coming...and five kings continue their 
## [3] Duplicity and treachery...nobility and honor...conquest and triumph...and, of course, dragons. I
## [4] The War of the Five Kings is drawing to a close, but new intrigues and plots are in motion, and 
## [5] The War of the Five Kings, once thought to be drawing to a close, is instead entering a new and 
## [6] Following the shocking developments at the conclusion of season five, survivors from all parts o
## [7]
```

Let's do just a little basic clearning and transforming of our text:

```r
# Convert the text to lower case (no need to differentiate for
# example between "Me" and "me")
seasonOverviews <- tm_map(seasonOverviews, content_transformer(tolower))
episodeOverviews <- tm_map(episodeOverviews, content_transformer(tolower))
# Remove numbers: not needed
seasonOverviews <- tm_map(seasonOverviews, removeNumbers)
episodeOverviews <- tm_map(episodeOverviews, removeNumbers)
# Remove english common stopwords
seasonOverviews <- tm_map(seasonOverviews, removeWords, stopwords("english"))
episodeOverviews <- tm_map(episodeOverviews, removeWords, stopwords("english"))
```

```r
# Remove punctuation
seasonOverviews <- tm_map(seasonOverviews, removePunctuation)
episodeOverviews <- tm_map(episodeOverviews, removePunctuation)
# Eliminate extra white spaces
seasonOverviews <- tm_map(seasonOverviews, stripWhitespace)
episodeOverviews <- tm_map(episodeOverviews, stripWhitespace)
```

For an overview why we've removed stop words (including what they are) see: https://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html.

Next we're going to build a *term-document matrix*: a document matrix containing the frequency of the words. Column names are words and row names are documents.

```r
dtmSeason <- TermDocumentMatrix(seasonOverviews)
dtmEpisodes <- TermDocumentMatrix(episodeOverviews)
```

So what do we have here:

```r
dtmSeason
```

```
## <<TermDocumentMatrix (terms: 202, documents: 7)>>
## Non-/sparse entries: 246/1168
## Sparsity           : 83%
## Maximal term length: 17
## Weighting          : term frequency (tf)
```

We have 7 documents (one overview for each of the 7 seasons). After cleaning, we have 202 unique words (terms), this means our matrix has 202 x 7 (202 *7) cells. Of which 246 have a value, and 1168 do not. Having a value means the word represented by the column is 1 or more than 1. A sparsity of 83% means that 83% of cells have no value (here 0 is the same as no value). Our longest word, is 17 charactors long.

We can now answer our question two ways:

- Compute which column has the highest sum, or
- Make a word cloud

To numerically find the most common word, we convert the document term matrix into a regular matrix (which will transpose it) and then compute the row sums, and then sort the row sums:

```r
m <- as.matrix(dtmSeason)
v <- sort(rowSums(m),decreasing=TRUE)
d <- data.frame(word = names(v),freq=v)
head(d, 10)
```

```
##              word freq
## throne     throne    6
## new           new    6
## great       great    4
## iron         iron    4
## wall         wall    4
## westeros westeros    4
## five         five    4
## kings       kings    4
## power       power    3
## season     season    3
```

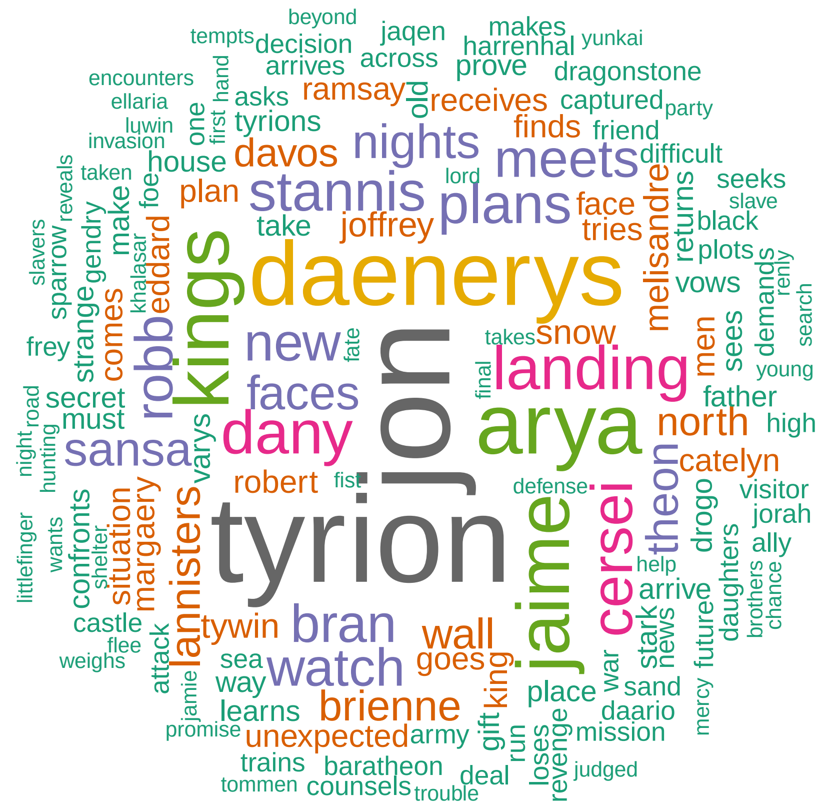
```r
m <- as.matrix(dtmEpisodes)
v <- sort(rowSums(m),decreasing=TRUE)
```

```r
d <- data.frame(word = names(v),freq=v)
head(d, 10)
```

```
##               word freq
## jon            jon   34
## tyrion      tyrion   32
## daenerys daenerys   23
## arya          arya   21
## kings        kings   19
## jaime        jaime   18
## landing    landing   14
## dany          dany   14
## cersei      cersei   13
## plans        plans   12
```

Or as a word cloud, in this case for episodes:

```r
library(wordcloud)
wordcloud(words = d$word, freq = d$freq, min.freq = 1,
          max.words=150, random.order=FALSE, rot.per=0.35,
          colors=brewer.pal(8, "Dark2"))
```



Question 5: Which season has the most positively phrased overview?

For this, we are going to consume a web API, coding this by hand is doable, but cumbersome. We're going to use AYLIEN for this lab, you will need to register and retreive an API and an App key from https://developer.aylien.com and set them as follows:

```r
key <- "<your key here>"
appID <- "<your app id here>"
```

We can use the API for example as follows. Note that AYLIEN requires the App ID and Key to be placed in the html header of the request.

```r
library(httr)
library(RCurl)

url <- "https://api.aylien.com/api/v1/sentiment"
headers <- add_headers(`X-AYLIEN-TextAPI-Application-Key` = key,
                              `X-AYLIEN-TextAPI-Application-ID` = appID)

r <- GET(url, headers, query = list(text = "i hate r"))
df <- content(r)
df$polarity
```

```
## [1] "negative"
```

```r
r <- GET(url, headers, query = list(text = "i like r"))
df <- content(r)
df$polarity
```

```
## [1] "neutral"
```

```r
r <- GET(url, headers, query = list(text = "i love r"))
df <- content(r)
df$polarity
```

```
## [1] "positive"
```

Now, let's run our Season overviews. We could do this with one of the apply functions, but it will be quite convoluted, so instead, we'll use a for loop to make the code easier to follow

```r
results <- list()

for (i in 1:length(sOverviews)) {
  r <- GET(url, headers, query = list(text = sOverviews[i]))
  results[[i]] <- httr::content(r)
}
```

So now we have a list of data.frames that correspond to the results. Let's restructure our results a little to consume them and answer our question.

```r
season <- c()
sentiment <- c()
confidence <- c()

for (i in 1:length(results)) {
  season[i] <- i
  sentiment[i] <- results[[i]]$polarity
  confidence[i] <- results[[i]]$polarity_confidence
}

df <- data.frame(season, sentiment, confidence)
df[df$sentiment == "positive", ]
```

```
##   season sentiment confidence
## 1      1  positive  0.6256582
## 3      3  positive  0.6866592
```

The confidence here is just how confident the API is in its assertion of the sentiment polarity. This is a standard measure in machine learning, it corresponds to a belief / probability of being correct.