Putting the "R" in Roger Ebert:
An Exercise in Data Mining with R

Welcome to this little tutorial on mining data from the web in R. It grew out of preparations for a workshop on automatizing stuff in R that I am going to teach in a couple of weeks, but I took this part out because it's way too advanced for the target audience of that workshop. But I figured that some people might still find it interesting.
To fully understand this tutorial, you need two things (apart from the document you're currently reading): The R script films.R and the spreadsheet films.csv. It should be in the same .zip folder as this document (which is available here).
The tutorial is divided into three parts, entitled *Why?*, *How?*, and *Some cautionary remarks* (sorry for being a killjoy and ending the tutorial with a caveat).

1. Why?

The web is an enormously rich source of data. This is obviously a great thing, as it opens up new avenues of research. But it also entails new challenges. You have to know how to handle the data (see Section 3), but you also have to know how to get the data in the first place. In some cases, this is easy. If you work with corpora, for example, you can often use nice export functions. If you're working with the World Atlas of Language Structures, you can download the data you're interested in quite conveniently in different formats. You can even get the entire FrameNet data upon request.
But things aren't always that easy. Sometimes you might want to access data that can't be easily exported. Unlike WALS or FrameNet, most data available on the web was never intended for scientific studies - which makes it all the more interesting. So it's good to know how to access data from the web directly.

Excursus 1: My first corpus study

Believe it or not, but when I did my first corpus study, I copy&pasted (sic!) every single token I found into a word document (sic!!!). The corpus I was working with wasn't actually a corpus, and I didn't know much about corpus linguistics back then anyway. So I just searched the Middle High German texts with my browser. Life would have been so much easier for me back then if I had already known at least some of the methods I'll describe below...
In fact, there is even a rather embarrassing section in my first journal paper in which I point out that I can't access the total token frequency of the texts I worked with because the texts have row numbers, which would be included in the count. Needless to say, removing them would have been easy as pie with the adequate methods in R. (Irrespective of that, there's this incredibly complicated and advanced mathematical operation called "subtraction" that I could have used...)

Suppose you want to know if there's a correlation between the commercial success of a movie and the rating it received from the late Roger Ebert, who was probably the most famous film critic in the world (apart from Slavoj Žižek, of course ;-)).[1]
In the file films.csv, you find a list of 100 movies from the "New Hollywood" era to the early 2000s, which I compiled by wildly choosing films from the "1980s in film" etc. Wikipedia articles. A real study would have to choose its objects of investigation much more carefully, of course. But for exploring data mining, this dataset is good enough.

[1] Disclaimer: Note that we're only interested in a possible correlation for the sake of the example - there's no really good hypothesis supporting it.

For conducting our correlation study, we need two things: Ebert's ratings and the box office data for each film. How do we get these?

2. How?

2.1 X marks the spot: Getting acquainted with the sources

Let's start with the box office data. An obvious choice to get this information is the website boxofficemojo.com. Of course it would be very tedious to search every single film there and then copy&paste the domestic gross data (especially if we were working with, say, 1000 or 10,000 films rather than just 100).

For the sake of simplicity, we'll only take the US domestic gross into account (which is what you get in Boxofficemojo's preview right after using the search interface). We want R to do this for us. But mining data is a bit like searching for a hidden treasure: If you just start digging a hole in your garden, you're bound to fail. You have better chances when you study a map (remember, X marks the spot!). In our case, the map is the website itself. You have to get acquainted with it and understand how it works.

Luckily, this is quite straightforward in the case of Boxofficemojo. It uses a very data miner-friendly search engine. Try, for example, searching for "Jurassic Park". You will encounter the first problem we have to deal with - and its solution.

TASK 1
What's the problem, what's the solution?

The problem is, of course, that we get more than one result (because some people thought it would be a good idea to do a couple of sequels). However, the good news is that the hit that fits our query most accurately is highlighted in some yellow-ish color. This color has the html code #FFFF99, which we can search in the source text. (In all common browsers, you can access the source text by right clicking and selection "View source text" or the like.) In other words, #FFFF99 is the X that marks the spot on our map. We'll keep this in mind for later.

Let's now have a look at rogerebert.com. Unfortunately, this website is not as data miner-friendly as Boxofficemojo. Try to search for your favorite movie and then take a look at the source text of the page with the search results. You probably won't find the title of your favorite movie here because the search results are drawn from a database and only embedded into the page. In other words, the source text will always be the same, no matter which movie you've searched for. Luckily, however, the URLs of the reviews always have the same format:

```
http://www.rogerebert.com/reviews/jurassic-park-1993
http://www.rogerebert.com/reviews/the-godfather-1972
```

So we also need the year in which each movie was released. Luckily, we don't have to look far: Apart from the box office results, Boxofficemojo also lists the release years. However, things are a bit more complicated if a movie has multiple release years. Take another look at the Boxofficemojo search result page for "Jurassic Park".

TASK 2
Which problem becomes apparent if you compare the search result for "Jurassic Park" with that of "Jurassic Park III"? Do you have an idea how we could approach it?

"Jurassic Park" was released in 1993 - and later, it was released again, in 3D. This causes a problem because the release date is not shown in the preview box that is displayed in the search results. Also, this problem is not limited to "Jurassic Park". Nowadays it's fairly uncommon that a movie sees multiple releases, but this used to be quite different. So we have to take the extra step and access the movie page itself (i.e. the page you get on when you click "Jurassic Park" in the search results[2]). If we take a look at the source code again, we see that the link to the movie page is in the same segment as the box office gross (which makes sense, because they're both in the yellow-ish area):

```
<tr bgcolor=#FFFF99>
      <td>
            <b><font size="2"  face="verdana"><a
href="/movies/?id=jurassicpark.htm">Jurassic Park</a></font></b></td>
      <td><font size="2"  face="verdana">Uni.</font></td>
      <td align="right"><font size="2"  face="verdana">$402,453,882</font></td>
      <td align="right"><font size="2"  face="verdana">2,566</font></td>
      <td align="right"><font size="2"  face="verdana">$47,026,828</font></td>
      <td align="right"><font size="2"  face="verdana">2,404</font></td>
      <td align="right"><font size="2"  face="verdana"><a
href="/movies/?page=releases&id=jurassicpark.htm">multiple</a></font></td>
      <td align="center"><font face="Verdana" size="2"><a
href="/movies/?page=showtimes&id=jurassicpark.htm"><b>Showtimes</b></a><br><a
href="/movies/?page=media&id=jurassicpark.htm">Images</a></font></td>

</tr>
```

2.2 Writing the code

2.2.1 Box office results

Methods introduced in this section:
- access web pages with R by using the URL
- automatically access many web pages by putting together the individual elements of the URL
- using basic Perl expressions to extract data

Functions used/introduced in this section: *readLines(), gsub(), paste(), strsplit()*

To get the box office results and the years, we want to automatically access the Boxofficemojo search results for each movie as well as the movie page itself, to which the

[2] A note on terminology: I will refer to Boxofficemojo's page about a movie as the "movie page", while "search result page" refers - quite straightforwardly - to the page you get when searching for a movie title. If your search was successful, you see a box with at least one link to a movie page in the search results. I refer to this box as the "preview box" because you can see a preview of the most important data in this box.

link in the search results leads. Before doing that, let's read in the spreadsheet with the movie titles:

```
films <- read.csv("films.csv", head=T, quote="'")
```

When coding a complex function, I always recommend that you start in the core and work your way to the "periphery". In our case, the core is of course the line of code that extracts the data from the website. So let's just start by trying out the function we have in mind with the first movie on our list (>>> means that the previous line continues):

```
temp <- readLines(con=paste("http://www.boxofficemojo.com/search/?q=", gsub(" ",
>>>"%20", films$Film[1]), sep=""))
```

What does it do? Let's walk through it bit by bit:
The function name *readLines()* is self-explanatory - it reads in lines from a connection (con). This connection can be a file on your hard drive or, as in our case, a web page. As we plan to automatize the function and to build a loop around this "core", we break up the URL in order to make sure that we don't have to enter the URL for each movie individually. The first part, "http://www.boxofficemojo.com/search/?q=", is the same for every search results page. It is followed by the title of the movie. In the address bar of your browser, it is probably displayed like this:

```
http://www.boxofficemojo.com/search/?q=jurassic park
```

However, as soon as you copy and paste the URL into a text document, you get:

```
http://www.boxofficemojo.com/search/?q=jurassic%20park
```

Note that the blank spaces are replaced with the ISO code for a blank space, preceded by "%". So we have to replace the blank spaces in our titles with "%20" as well. To this end, we use R's *gsub()* function. The *paste()* function is used to put the individual parts together, and sep="" specifies that the individual parts shouldn't be separated by any character.
If you use the code above, the source text of the search result page for the first movie in our list should be stored in the variable temp.
We've seen before that all the data that interest us are stored in one single segment, starting with <tr bgcolor=#FFFF99>. However, this only works for exact search results. Try entering *Lord of the Rings Fellowship*. As there's no exact hit, there's no yellow highlighting. Therefore, we can't use #FFFF99 as the X that marks the spot in these cases. On the upside, however: You only get one result, preceded by "1 Movie Matches". To make sure that our function can deal with both options, we use if-conditionals:

```
  if(length(grep("<tr bgcolor=#FFFF99>", temp))>0) {
      temp <- temp[grep("<tr bgcolor=#FFFF99>", temp):length(temp)]
  } else if(length(grep("1 Movie Matches", temp))>0) {
      temp <- temp[grep("1 Movie Matches", temp):length(temp)]
  } else {
      temp <- NA
  }
```

These conditionals are a bit cumbersome and there might be more elegant solutions. Anyway, the logic behind it is: as soon as a line specifying the color #ffff99 can be found, anything preceding this line is deleted; if "1 Movie Matches" can be found, anything preceding this line is deleted. The last condition is necessary to avoid errors: If neither #ffff99 nor "1 Movie

Matches" can be found, R won't know what to do and produce an error. In addition, this probably means that our movie hasn't been found and that the data are useless anyway.[3]
However, it should work perfectly fine with the first film in our list. So we can now try to get its box office results:

```
gsub("[^0-9]", "", unlist(strsplit(grep("<td align=\"right\"><font size=\"2\"
>>>face=\"verdana\">",temp,value=T)[1], "\\$"))[2], perl=T)
```

The *strsplit()* function is a pretty cool tool for breaking up text elements into different parts. For example, you can tear apart *Romeo and Juliet* with `strsplit("Romeo and Juliet", split=" ")`. Here, we define a blank space as the separator. *strsplit()* returns a list of text elements (in our case: "Romeo", "and", "Juliet"). The separator itself is lost in the process, i.e. there a no blank spaces any more in the list. If we want a vector instead of a list, we have to unlist the resulting list: Try `unlist(strsplit("Romeo and Juliet", split=" "))`.
What is left of the Boxofficemojo source text in our temp variable starts with `<tr bgcolor=#FFFF99>` or with `<br><b>1 Movie Matches: </b><table border="0" cellpadding="5" cellspacing="0">`, followed by a few more lines. What is important for us is that the line specifying the gross is the first in a number of lines that begins with

```
<td align="right"><font size="2" face="verdana">
```

So we can search for the first line containing this bit of code. By using the *grep()* function with `value=T`, we ensure that the function returns the matching elements themselves rather than their position. As the gross is always preceded by a dollar sign, we can use it as split argument in the strsplit function. Note that we have to escape scarequotes as well as the dollar sign with the help of backslashes, otherwise R will interpret them as operators.

```
unlist(strsplit(grep("<td align=\"right\"><font size=\"2\"
>>>face=\"verdana\">",temp,value=T)[1], "\\$"))[2]
```

returns something like:

```
 "179,800,601</font></td>"
```

To get rid of the non-numeric characters, we use *gsub()* with Perl expressions: [0-9] stands for numbers in Perl, while ^ is used as a negator. So `gsub([^0-9], "", ...)` means: replace everything that's not a number by nothing.
In the same way, we can get the link to the movie page:

```
link2 <- unlist(strsplit(grep("movies/\\?id\\=", temp, value=T)[1], "a
>>>href=\"|\\.htm"))[2]
```

... and we can use it in the same way we used the link for the search results:

```
temp002 <- readLines(con=paste("http://www.boxofficemojo.com", link2, ".htm",
>>>sep=""))
```

Again, we have to get acquainted with the structure of the page. We quickly find that the release year is either preceded by "release=theatrical&date=" or simply by "Release Date:". Again, we can use if-conditionals to take both options into account.

[3] This is one option to handle errors that might potentially occur. Another one is *tryCatch()*, which I'll introduce in the next section.

```
if(length(grep("release\\=theatrical&date\\=", temp002))>0) {
      year <- unlist(strsplit(unlist(strsplit(grep("Release Date\\: ", temp002,
>>>value=T), "release\\=theatrical&date\\="))[2], "-"))[1]
    } else {
      year <- gsub(" ", "", unlist(strsplit(unlist(strsplit(grep("Release Date\\:",
>>>temp002, value=T), "Release Date\\:"))[2], ",|</nobr"))[2])
}
```

Now we have the box office results and the years - we just have to put all this into a nice for-loop. Before that, let's create three new columns for our dataframe: one for the year, one for the box office gross, and one for the Ebert stars.
We could count how many columns our data frame already has (pretty easy in our case: one!) and manually add a second, third, etc. column. But from my experience with corpus data (with lots and lots of columns), I recommend checking for the column length automatically - if you want to open your spreadsheet again in Excel, Calc, or any other program and add a column manually (e.g. with your own ratings of the films), you don't have to modify your code afterwards.

```
##add column for years
current_length <- length(films)
films[current_length+1] <- NA; colnames(films)[current_length+1] <- "Year"

##add column for box office
current_length <- length(films)
films[current_length+1] <- NA; colnames(films)[current_length+1] <- "Box_Office"

##add column for Ebert Stars
current_length <- length(films)
films[current_length+1] <- NA; colnames(films)[current_length+1] <- "Ebert_Stars"
```

We can now easily fill the second and third rows by putting what we have so far into a for loop:

```
for(i in 1:length(films$Film)) {
  ##get data from boxofficemojo.com:
  temp <- readLines(con=paste("http://www.boxofficemojo.com/search/?q=", gsub(" ",
>>>"%20", films$Film[i]), sep=""))

  if(length(grep("<tr bgcolor=#FFFF99>", temp))>0) {
    temp <- temp[grep("<tr bgcolor=#FFFF99>", temp):length(temp)]
  } else if(length(grep("1 Movie Matches", temp))>0) {
    temp <- temp[grep("1 Movie Matches", temp):length(temp)]
  } else {
    temp <- NA
  }

  if(is.na(temp)) {
    films$Year[i] = films$Box_Office[i] = NA
  } else {
    films$Box_Office[i] = gsub("[^0-9]", "", unlist(strsplit(grep("<td
>>>align=\"right\"><font size=\"2\"  face=\"verdana\">",temp,value=T)[1],
>>>"\\$"))[2], perl=T)

    ###in case of multiple releases, the release year is not given in the search
    ###overview; therefore we have to extract it from the movie page
```

```
##get link
   link2 <- unlist(strsplit(grep("movies/\\?id\\=", temp, value=T)[1], "a
>>>href=\"|\\.htm"))[2]

##use link
    temp002 <- readLines(con=paste("http://www.boxofficemojo.com", link2, ".htm",
>>>sep=""))

    ##get year
    if(length(grep("release\\=theatrical&date\\=", temp002))>0) {
      films$Year[i] <- unlist(strsplit(unlist(strsplit(grep("Release Date\\: ",
>>>temp002, value=T), "release\\=theatrical&date\\="))[2], "-"))[1]

    } else {
      films$Year[i] <- gsub(" ", "", unlist(strsplit(unlist(strsplit(grep("Release
>>>Date\\:", temp002, value=T), "Release Date\\:"))[2], ",|</nobr"))[2])
    }



  }
}
```

2.2.2 Get stars

Methods introduced in this section:
- Basic error handling

Functions used/introduced in this section: *tryCatch()*

Now let's reach for the stars! Again, we start with the "core" of the function:

```
review_page <- readLines(con=tolower(paste("http://www.rogerebert.com/reviews/",
>>>paste(gsub(" ", "-", films$Film[1]), films$Year[1], sep="-"), sep="")))
```

We've seen before that the URL of each review consists of "rogerebert.com/reviews/", the title, and the year. The *paste()* function in the line of code above puts these three elements together pretty much in the same way as we've seen in the previous examples. In contrast to Boxofficemojo, the individual words of each title are connected by a dash rather than a blank space. Also, the year is separated from the title with a dash. That's why we use a *paste()* function within a *paste()* function here: There should be a dash between the title and the year, but not between "reviews/" and the first word of the title. Importantly, all titles are lowercase, so we use *tolower()* to transform the titles.
If we tried this for every single film in our films dataframe, we would find that the function doesn't always work: In some cases, the link might be wrong, in other cases, there might be no review in the first place. In these cases, you'll get the infamous error 404. Remember that we plan to automatize our function - but when it encounters an error, the loop (which we haven't created yet) will stop. To prevent this, we can use R's *tryCatch()* function. To illustrate how it works, let's write a little function with an inbuilt nasty error (you can find it at the bottom of the *films.R* script):

```
errorpronefunction <- function(x) {
  if(x %in% c(5,7,9)) {
    stop("This is a nasty error")
  }
  return(x+pi)
}
```

Let's try to execute the function without *tryCatch()*:

```
for(i in 1:10) {
  print(errorpronefunction(i))
}
```

As expected, we encounter an error as soon as the fifth iteration is reached (because we specified in the errorpronefunction itself that it doesn't work for 5, 7, and 9). Let's now use the error handling function:

```
for(i in 1:10) {
  tryCatch({
    print(errorpronefunction(i))
  },
  error=function(e) {cat(conditionMessage(e), "\n")})
}
```

As you can see, it works fine now.[4]

Excursus 2: A more recent corpus study

By now you might be wondering if what we're doing here is of any use if you're not interested in box office results or the subjective ratings of one individual. However, I actually got acquainted with most of the methods I'm introducing here when doing a corpus search. I used the German Text Archive, which is a wonderful resource for anyone studying language change in German. However, it has one major shortcoming (in its present version): While you can export metadata like text type or year in your concordance, you can't export data like lemma or part of speech, although they are displayed in the html version of your search results. So I wrote a code that tries to automatically extract the data from the web. This saved me a lot of work in lemmatizing about 700,000 tokens, although it still required many days of manual correction.

I have to confess that I initially had some difficulties understanding the tryCatch() syntax, but on closer look, it's pretty simple. We first enter the expression we want to use in curved brackets, and then we define an error handling function. In our example, it is a function that simply returns the error message. We could also define an empty function:

```
for(i in 1:10) {
  tryCatch({
    print(errorpronefunction(i))
  },
  error=function(e) {})
}
```

That's what I've done in the code that you find in the *films.R* script: The entire loop is "wrapped" in a *tryCatch()* function. Before I reproduce the entire code here, let's first have a look at the other elements we need in the loop apart from the *readLines()* function given above.
After executing the line of code given at the very beginning of this section, we should have the review page of the first movie on our list stored in the variable *review_page*.[5]

[4] I am indebted to this "Stackoverflow" answer: http://stackoverflow.com/questions/14748557/skipping-error-in-for-loop, by which the example is inspired.

If you take a look at Ebert's reviews, you will see that the star ratings are given at the top of each review. He uses a four-star scale with full starts and half stars. For example, *Pulp Fiction* has four stars, the almost-forgotten *Even Cowgirls get the Blues* has only 0.5. The pictures of the full and half stars can be found in the source code by searching for *icon-star-full* and *icon-star-half*, respectively. However, a closer look at the source code reveals that there are many more stars on the page than just four (because there are previews of other reviews etc. on the page), so it wouldn't suffice to just search the page for "icon-star-full" and "icon-star-half".

TASK 3
Given what you already know about analyzing text data in R - what can we do to find the instances of "icon-star-full" and "icon-star-half" that are relevant for us? Take a look at the source text of a review page to find out.

If you take a close look at the source text, you'll find that the relevant line starts out with "reviewRating" - as you've probably already expected, that's our new X that marks the spot. We cut off everything before "reviewRating" and everything after the first instance of "", which indicates the end of the segment relevant to us.

```
review_page <- review_page[grep("reviewRating", review_page):length(review_page)]
review_page <- review_page[1:grep("</span>", review_page)[1]]
```

Now we just have to count how often "icon-star-full" and "icon-star-half" occur in the remaining piece of code:

```
rating <- length(grep("icon-star-full", unlist(strsplit(review_page[2], "<|>")))) +
>>>0.5*length(grep("icon-star-half", unlist(strsplit(review_page[2], "<|>"))))
```

Last but not least, we have to put these elements together in one single loop and we have to wrap the *tryCatch()* function around it.

```
####get Ebert Stars
for(i in 1:length(films$Film)) {
  ##get the review page
  tryCatch({
    review_page <-
>>>readLines(con=tolower(paste("http://www.rogerebert.com/reviews/", paste(gsub("
>>>", "-", films$Film[i]), films$Year[i], sep="-"), sep="")))


    ##cut it down to the relevant <span> element:
    if(length(grep("reviewRating", review_page))>0) {
      review_page <- review_page[grep("reviewRating",
>>>review_page):length(review_page)]
      review_page <- review_page[1:grep("</span>", review_page)[1]]

      ##get rating
      rating <- 0
      rating <- rating + length(grep("icon-star-full",
>>>unlist(strsplit(review_page[2], "<|>")))) +
        0.5*length(grep("icon-star-half", unlist(strsplit(review_page[2], "<|>"))))
      films$Ebert_Stars[i] <- rating
    } else {
      films$Ebert_Stars[i] <- NA
```

[5] If it didn't work, perhaps try another movie by changing the [1]s to a different number between 2 and 100 - as pointed out before, the function doesn't work with every title for various reasons.

```
    }


  }, error = function(e){})
}
```

That's it! We get a warning message, but we can safely ignore it. If we take a look at the *films* data frame now, we see that our code was relatively successful. Of course there are still some missing values - in some cases, we might avoid those by tweeking the original csv file a bit (perhaps we would have to replace "The Godfather Part 2" by "The Godfather Part II"), but that's of course always a trade-off between how much effort you can afford to put into the "fine-tuning" of your initial data to optimize the results of your automatization and how much missing data you want to live with.
Anyway, we can now try to look for a correlation. As Figure 1 reveals, higher-rated films tend to be more successful indeed, but not significantly so (R^2=0.02, F=2.42, p=0.12).

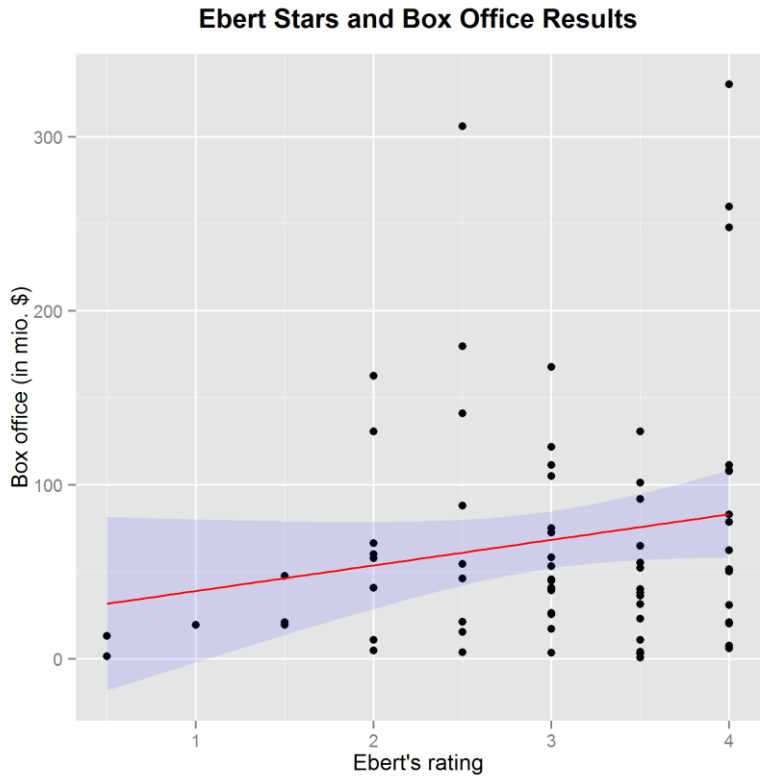

Fig. 1: Mission accomplished: Here's the result of our data mining adventure!

3. Some cautionary remarks

As I said right at the beginning, the almost unlimited possibilities of data mining open up new avenues of research. However, they also entail new problems and risks. One of those risks is the temptation to collect all kinds of big data and arriving at a variety of spurious correlations: The more data you have, the more likely it is that you will find some correlation. I therefore strongly recommend googling "spurious correlations" and/or visiting the spurious correlations section on replicatedtypo.com before publishing your paper on the correlation between Roger

Ebert's ratings of foreign movies and the suicide rate in the countries where these movies were produced. Always remember to make sure that the correlation you explore is backed by a good hypothesis and try to take potential confounding factors into account.

In addition, there are some nasty legal issues that we have to keep in mind. While it is probably fairly unproblematic to automatically access the openly available source texts of websites in order to automatically extract box office results or movie ratings for non-commercial purposes, it might be problematic to automatically extract and store large portions of text from the web (depending on the copyright laws in your country). So you should probably check what's legally possible in your country before publishing your paper on the automatic extraction of Valyrian passages from your illegally downloaded *Game of Thrones* e-books...

One last caveat: Automatization is great, and for big data, it's unavoidable. But you should never blindly rely on the functions you wrote. Always check at least a sample of the data you gathered automatically. There might still be something you overlooked or you didn't think about. For example, there's at least one thing that we could improve in the code introduced here. Can you guess what it is, and can you try to improve the code? - Hint: It has to do with apostrophes...