**STAT 19000 Projects from 2018-19**

**Fall 2018 website:** [**https://www.stat.purdue.edu/datamine/19000/oldindex.html**](https://www.stat.purdue.edu/datamine/19000/oldindex.html)

**Spring 2019 website:** [**https://www.stat.purdue.edu/datamine/19000/**](https://www.stat.purdue.edu/datamine/19000/)

**Project 1:**

1.  Use the airline data stored in this directory:  
  
/depot/statclass/data/dataexpo2009  
  
In the year 2005, find:

1a.  the number of flights that occurred, on every day of the year, and

1b.  find the day of the year on which the most flights occur.  
  
2.  Again considering the year 2005, did United or Delta have more flights?  
  
3.  Consider the June 2017 taxi cab data, which is located in this folder:  
  
/depot/statclass/data/taxi2018  
  
What is the distribution of the number of passengers in the taxi cab rides?  In other words, make a list of the number of rides that have 1 passenger; that have 2 passengers; etc.

**Project 1 solution:**

# 1. We switch to the directory for the airline data

cd /depot/statclass/data/dataexpo2009

# 1a. The number of flights that occurred, on every day of the year,

# can be obtained by extracting the 1st, 2nd, and 3rd fields,

# sorting the data, and then

# summarizing the data using the uniq command with the -c flag

sort 2005.csv | cut -d, -f1-3 | sort | uniq -c

# The first few lines of the output are:

# 16477 2005,10,1

# 19885 2005,10,10

# 19515 2005,10,11

# 19701 2005,10,12

# 19883 2005,10,13

# and the last few lines of the output are:

# 20051 2005,9,6

# 19629 2005,9,7

# 19968 2005,9,8

# 19938 2005,9,9

# 1 Year,Month,DayofMonth

# 1b. The day of the year on which the most flights occur can be found by

# sorting the results above, in numerical order, using sort -n

# and then (if desired, although it is optional) we can

# extract the last line of the output using tail -n1

sort 2005.csv | cut -d, -f1-3 | sort | uniq -c | sort -n | tail -n1

# and we conclude that the most flights occur on August 5:

21041 2005,8,5

# 2. We can extract the 9th field, which is the carrier (i.e., the airline company)

# and then, in the same way as above, we can sort the data, and then we can

# summarize the data using uniq -c

# This yields the number of flights for each carrier.

# We can either read the number of United or Delta flights with our eyeballs,

# or we can use the grep command, searching for both the pattern UA and DL

# to isolate (only) the number of flights for United and Delta, respectively.

sort 2005.csv | cut -d, -f9 | sort | uniq -c | grep "UA\|DL"

# The output is

# 658302 DL

# 485918 UA

# so Delta has more flights than United in 2005.

# 3. Now we change directories to consider the taxi cab data

cd ../taxi2018

# The ".." in the previous command just indicates that we want to go up one level to

# /depot/statclass/data

# and then, from that point, we want to go into the taxi cab directory.

# If this sounds complicated, then (instead) it is safe to use the longer version:

cd /depot/statclass/data/taxi2018

# The number of passengers is given in the 4th column, passenger\_count

# We use a method that is similar to the one from the first three questions,

# We extract the 4th column, sort the data, and then

# summarizing the data using the uniq command with the -c flag

sort yellow\_tripdata\_2017-06.csv | cut -d, -f4 | sort | uniq -c

# and the distribution of the number of passengers is:

# 1

# 548 0

# 6933189 1

# 1385066 2

# 406162 3

# 187979 4

# 455753 5

# 288220 6

# 26 7

# 30 8

# 20 9

# 1 passenger\_count

# Notice that we have some extraneous information, i.e., there is

# one blank line and also one line for the passenger\_count (from the header)

**Project 2:**

awk examples: <https://www.stat.purdue.edu/datamine/19000/day3.html>

1.  Use the airline data stored in this directory:  
  
/depot/statclass/data/dataexpo2009

1a. What was the average arrival delay (in minutes) for flights in 2005?  
  
1b. What was the average departure delay (in minutes) for flights in 2005?  
  
1cd. Now revise your solution to 1ab, to account for the delays (of both types) in the full set of data, across all years.  
  
  
2. Revise your solutions to 1abcd to only include flights that took place on the weekends.

3.  Consider the June 2017 taxi cab data, which is located in this folder:  
  
/depot/statclass/data/taxi2018  
  
What is the average distance of a taxi cab ride in New York City in June 2017?

**Project 3:**

Use R to revisit these questions.  They can each be accomplished with 1 line of code.

[[It is OK to submit this project as a ".R" file in GitHub.  In the near future, I hope to move us into RMarkdown, but I don't want to overwhelm students with this at the start.  So just submit your ".R" files into GitHub.  \*As always\* please include your solution and also comments about your method of solution, so that the graders can quickly check your work!]]

1.  As in Project 1, question 2:  In the year 2005, did United or Delta have more flights?

2.  As in Project 2, question 2a:  Restricting attention to weekends (only), what was the average arrival delay (in minutes) for flights in 2005?

3.  As in Project 1, question 3:  In June 2017, what is the distribution of the number of passengers in the taxi cab rides?

4.  As in Project 2, question 3:   What is the average distance of a taxi cab ride in New York City in June 2017?

**Project 4:**

Revisit the map code on the STAT 19000 webpage: <http://www.stat.purdue.edu/datamine/19000/>

Goal:  Make a map of the State of Indiana, which shows all of Indiana's airports.

Notes:

* You will need to install the ggmap package, which takes a few minutes to install.
* You can read in the data about the airports from the Data Expo 2009 Supplementary Data: <http://stat-computing.org/dataexpo/2009/supplemental-data.html>
* It will be necessary to extract (only) the airports with "state" equal to "IN"
* It is possible to either dynamically load the longitude and latitude of Indianapolis from Google, or to manually specify the longitude and latitude (e.g., by looking them up yourself in Google and entering them).
* After you plot the State of Indiana with all of the airports shown, you can print the resulting plot to a pdf file as follows: dev.print(pdf, "filename.pdf")
* Please submit your GitHub code in a ".R" file and also the resulting ".pdf" file. It is not (yet) necessary to submit your work in RMarkdown.

**Project 5:**

1a.  Compute the average distance for the flights on each airline in 2005.   
  
1b.  Sort the result from 1a, and make a dotchart to display the results in sorted order.  (Please display all of the values in the dotchart.)  
  
Hint:  You can use: ?dotchart if you want to read more about how to make a dotchart about the data.   
  
2a.  Compute the average total amount of the cost of taxi rides in June 2017, for each pickup location ID.   
  
You can see which variables have the total amount of the cost of the ride, as well as the pickup location ID, if you look at the data dictionary for the yellow taxi cab rides, which you can download here: <http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml>   
  
2b.  Sort the result from 2a, and make a dotchart to display the results in sorted order.  (Please ONLY display the results with value bigger than 80.)

3.  Put the two questions above -- including your comments -- into an RMarkdown file.  Submit the .Rmd file itself and either the html or pdf output, when you submit your project in GitHub.

Note: For Project 5, it is enough to only analyze the airline flights from the year 2005 (only) and the taxi cabs from the month July 2017 (only)

**Project 6:**

Consider the election donation data: <https://www.fec.gov/data/advanced/?tab=bulk-data> from "Contributions by individuals" for 2017-18. Download this data. Unzip the file (in the terminal).

Use the cat command to concatenate all of the files in the by\_date folder into one large file (in the terminal).

Read the data dictionary: <https://www.fec.gov/campaign-finance-data/contributions-individuals-file-description/>

Hint: When working with a file that is not comma separated, you can use the read.delim command in R, and \*be sure to specify\* the character that separates the various pieces of data on a row. To do this, you can read the help file for read.delim by typing: ?read.delim  
(Look for the "field separator character".) Also there is no header, so also use header=F  
  
Question 1:  Rank the states according to how many times that their citizens contributed (i.e., total number of donations). Which 5 states made the largest numbers of contributions?  
  
Question 2:  Use awk in the terminal to verify your solution to question 1.  
  
Question 3:  Now (instead) rank the states according to how much money their citizens contributed (i.e., total amount of donations).  Which 5 states contributed the largest amount of money?  
  
(Optional!!) challenge question:  Use awk in the terminal to verify your solution to question 3.  
This can be done with 1 line of awk code, but you need to use arrays in awk, as demonstrated (for instance) on Andrey's solution on this page: <https://unix.stackexchange.com/questions/242946/using-awk-to-sum-the-values-of-a-column-based-on-the-values-of-another-column/242949>   
  
Submit your solutions in RMarkdown.  
For question 2 (and for the optional challenge question), it is OK to just put your code into your comments in RMarkdown, so that the TA's can see how you solved question 2, but (of course) the awk code does not run in RMarkdown! You are just showing the awk code to the TA's in this way!

**Project 7:**

Consider the Lahman baseball database available at: <http://www.seanlahman.com/baseball-archive/statistics/>   
  
Download the 2017 comma-delimited version and unzip it. Inside the "core" folder of the unzipped file, you will find many csv files.  
  
If you want to better understand the contents of the files, there is a helpful readme file available here: <http://www.seanlahman.com/files/database/readme2017.txt>   
  
1.  Use the Batting.csv file (inside the "core" folder) to discover who is a member of the 40-40 club, namely, who has hit 40 home runs and also has (simultaneously) stolen 40 bases in the same season.  Hint: There are multiple ways to solve this question.  It is not necessary to use a tapply function. This can be done with one line of code.  
  
2.  Make a plot that depicts the total number of home runs per year (across all players on all teams).  The plot should have the years as the labels for the x-axis, and should have the number of home runs as the labels for the y-axis. Hints:  Use the tapply function.  Save the results of the tapply function in a vector v.  If do this, then names(v) will have a list of the years.  The plot command has options that include xlab and ylab, so that you can put intelligent labels on the axes, for instance, you can label the x-axis as "years" and the y-axis as "HR".  
  
3a.  Try this example: Store the Batting table into a data frame called myBatting. Store the People table into a date frame called myPeople. Merge the two data frames into a new data frame, using the "merge" function:  
     myDF <- merge(myBatting, myPeople, by="playerID")  
3b.  Use the paste command to paste the first and last name columns from myDF into a new vector.  Save this new vector as a new column in the data frame myDF.  
3c.  Return to question 1, and resolve it.  Now we can see the person's full name instead of their playerID.

Fun side project (easy to do; impressive to see). Not required, but fun!  
Read Teams.csv file into a data.frame called myDF. Break the data.frame into smaller data.frames, according to the teamID, using this code:  
by(myDF, myDF$teamID, function(x) {plot(x$W)} )  
  
For each team, this draws 1 plot of the number of wins per year.  The number of wins  
will be on the y-axis of the plots. For an improved version, we can add the years on the x-axis, as follows:  
by(myDF, myDF$teamID, function(x) {plot(x$year, x$W)} )  
  
Change your working directory in R to a new folder, using the menu option:  
Session -> Set Working Directory -> Choose Directory  
We are going to make 149 new plots!  
  
After changing the directory, try this code, which makes 149 separate pdf files:  
by(myDF, myDF$teamID, function(x) {pdf(as.character(x$teamID[1])); plot(x$year, x$W); dev.off()} )

**Project 8:**

1.  Modify SQL Example 2 to find the Pitcher who has the most Strikeouts in his career.

Hint:  You need to use a "Pitching p" table instead of a "Batting b" table.

Hint:  The strikeouts are in column "SO" of the Pitching table.

Hint:  This pitcher is named "Nolan Ryan"... but you need to use SQL to figure that out.

I am just trying to give you a way to know when you are correct.

Please momentarily forget that I am giving you the answer at the start!

2.  Which years was Nolan Ryan a pitcher?

For this project, to make your life easier, it is OK to just submit a regular R file, rather than an RMarkdown file.

**SQL Example 1**

# We only need to install this package 1 time.  
install.packages("RMySQL")  
# No need to run the line above, if you already ran it.  
# We need to run this library every time we load R.  
library("RMySQL")  
myconnection <- dbConnect(dbDriver("MySQL"),  
                          host="mydb.ics.purdue.edu",  
                          username="mdw\_guest",  
                          password="MDW\_csp2018",  
                          dbname="mdw")  
  
easyquery <- function(x) {  
  fetch(dbSendQuery(myconnection, x), n=-1)  
}  
###############################################  
  
# Here are the players from the Boston Red Sox in the year 2008  
myDF <- easyquery("SELECT m.playerID, b.yearID, b.teamID,  
                   m.nameFirst, m.nameLast  
                   FROM Batting b JOIN Master m  
                   ON b.playerID = m.playerID  
                   WHERE b.teamID = 'BOS'  
                   AND b.yearID = 2008;")  
myDF

# SQL Example 2

# We only need to install this package 1 time.  
install.packages("RMySQL")  
# No need to run the line above, if you already ran it.  
  
# We need to run this library every time we load R.  
library("RMySQL")  
myconnection <- dbConnect(dbDriver("MySQL"),  
                          host="mydb.ics.purdue.edu",  
                          username="mdw\_guest",  
                          password="MDW\_csp2018",  
                          dbname="mdw")  
  
easyquery <- function(x) {  
  fetch(dbSendQuery(myconnection, x), n=-1)  
}  
###############################################  
  
# Here are the total number of home runs hit by each player in their entire career  
myDF <- easyquery("SELECT m.nameFirst, m.nameLast,  
                   b.playerID, SUM(b.HR)  
                   FROM Batting b JOIN Master m  
                   ON m.playerID = b.playerID  
                   GROUP BY b.playerID;")  
myDF  
  
# Here are the players who hit more than 600 home runs in their careers  
myDF[ myDF$"SUM(b.HR)" >= 600, ]

# SQL Example 3

# We only need to install this package 1 time.  
install.packages("RMySQL")  
# No need to run the line above, if you already ran it.  
  
# We need to run this library every time we load R.  
library("RMySQL")  
myconnection <- dbConnect(dbDriver("MySQL"),  
                          host="mydb.ics.purdue.edu",  
                          username="mdw\_guest",  
                          password="MDW\_csp2018",  
                          dbname="mdw")  
  
easyquery <- function(x) {  
  fetch(dbSendQuery(myconnection, x), n=-1)  
}  
###############################################  
  
# Here is basic version for the players who have more than 60 Home Runs  
# during one season.  
myDF <- easyquery("SELECT b.playerID, b.yearID, b.HR  
                  FROM Batting b  
                  WHERE b.HR >= 60;")  
myDF  
# Here is an improved version,   
# which includes the Batting and the Master table,  
# so that we can have the players' full names.  
myDF <- easyquery("SELECT m.nameFirst, m.nameLast,  
                  b.playerID, b.yearID, b.HR  
                  FROM Master m JOIN Batting b  
                  ON m.playerID = b.playerID  
                  WHERE b.HR >= 60;")  
myDF

# SQL Example 4

# We only need to install this package 1 time.  
install.packages("RMySQL")  
# No need to run the line above, if you already ran it.  
  
# We need to run this library every time we load R.  
library("RMySQL")  
myconnection <- dbConnect(dbDriver("MySQL"),  
                          host="mydb.ics.purdue.edu",  
                          username="mdw\_guest",  
                          password="MDW\_csp2018",  
                          dbname="mdw")  
  
easyquery <- function(x) {  
  fetch(dbSendQuery(myconnection, x), n=-1)  
}  
###############################################  
  
# Here is basic version for the 40-40 club question. (Same question as last week.)  
myDF <- easyquery("SELECT b.playerID, b.yearID, b.SB, b.HR  
                   FROM Batting b  
                   WHERE b.SB >= 40 AND b.HR >= 40;")  
myDF  
  
# Here is an improved version, which includes the Batting and the Master table,  
# so that we can have the players' full names.  
myDF <- easyquery("SELECT m.nameFirst, m.nameLast,  
                   b.yearID, b.SB, b.HR  
                   FROM Master m JOIN Batting b  
                   ON m.playerID = b.playerID  
                   WHERE b.SB >= 40 AND b.HR >= 40;")  
myDF  
  
# Here is a further improved version, which includes the Batting, Master, and Teams table,  
# so that we can have the players' full names, and the teams that they played on.  
myDF <- easyquery("SELECT m.nameFirst, m.nameLast,  
                   b.yearID, b.SB, b.HR, t.name  
                   FROM Master m JOIN Batting b  
                   ON m.playerID = b.playerID  
                   JOIN Teams t  
                   ON b.yearID = t.yearID  
                   AND b.teamID = t.teamID  
                   WHERE b.SB >= 40 AND b.HR >= 40;")  
myDF

**Project 9:**

(Please remember that you have a "ReadMe" file, posted on Piazza last week, which tells you  
 about all of the tables, including the table that tells you where the students went to school.)  
  
1.  Find the first and last names of all players who attended Purdue.  
   
2.  Find all of the pitchers who have pitched 300 or more strikeouts during a single season.  
    In the output, give their first and last name and the year in which this achievement occurred.  
    (You can just modify Example 3.)  
  
3a. Modify Example 5 to find out which pitchers were able to achieve 300 or more strikeouts AND 20 or more wins during the same season.

3b. Consider the years in which this achievement occurred. Use R to find the list of distinct years in which this achievement occurred at least once.  
  
Background discussion:  
  
If you look at the example for the 40-40 club (in Example 4), it works because each time that a player achieved 40 (or more) HR's and 40 (or more) SB's during the same season, he was only playing for one team. A player never got traded to a new team, in any of those years. Some complications will arise if a player switches teams (i.e., gets traded) during the season.  For this reason, we introduce Example 5.  
  
Here are some notes about Example 5:  
If we incorporate the SUM function into a condition, for instance, WHERE SUM(b.SB) >= 40  
 the query will not work.  Instead, if the condition has a SUM inside it, we change "WHERE" to "HAVING". See Example 5 as a perfect example of this. We can also return the results in a given order, using:  ORDER BY  for instance, ORDER BY by.yearID if we want to get the results (say) in order by the year.

# SQL Example 5

# We only need to install this package 1 time.  
install.packages("RMySQL")  
# No need to run the line above, if you already ran it.  
  
# We need to run this library every time we load R.  
library("RMySQL")  
myconnection <- dbConnect(dbDriver("MySQL"),  
                          host="mydb.ics.purdue.edu",  
                          username="mdw\_guest",  
                          password="MDW\_csp2018",  
                          dbname="mdw")  
  
easyquery <- function(x) {  
  fetch(dbSendQuery(myconnection, x), n=-1)  
}  
###############################################  
# Here is basic version for the 30-30 club question.  
# (Same question as last week.)  
myDF <- easyquery("SELECT b.playerID, b.yearID, SUM(b.SB), SUM(b.HR)  
                   FROM Batting b  
                   GROUP BY b.playerID, b.yearID  
                   HAVING SUM(b.SB) >= 30 AND SUM(b.HR) >= 30  
                   ORDER BY b.yearID;")  
myDF  
  
# Here is an improved version, which includes the Batting and the Master table,  
# so that we can have the players' full names.  
myDF <- easyquery("SELECT m.nameFirst, m.nameLast,  
                   b.yearID, SUM(b.SB), SUM(b.HR)  
                   FROM Master m JOIN Batting b  
                   ON m.playerID = b.playerID  
                   GROUP BY b.playerID, b.yearID  
                   HAVING SUM(b.SB) >= 30 AND SUM(b.HR) >= 30  
                   ORDER BY b.yearID;")  
myDF

**Project 10:**

Use the results of the National Park Service scraping example to answer the following two questions:

1. Which states have at least 20 NPS properties?
2. One zip code has 13 properties in the same zip code!  What are the names of those 13 properties?

If you want to learn XPath (as demonstrated in the case study) to scrape data from a website of your choice, you can make up the grades from 1 or 2 of the previous projects. If you scrape at least 500 pieces of data from the XML of a page,you can replace the grade from 1 previous project. If you scrape at least 1000 pieces of data from the XML of a page, you can replace the grade from 2 previous projects. Your project plan will require written approval from Dr Ward, and it will require you to scrape the data from XML itself (not just download the data).

# case study: scraping National Park Service data

# This is a short project to download the data about the

# properties in the National Park Service (NPS).

# They are all online through the office NPS webpage:

# https://www.nps.gov/findapark/index.htm

# (Please note that some parks extend into more than one state.)

# At the end of the project, when we export the data,

# we do not want to use comma-separated values (i.e., a csv file)

# because there are also some commas in our data.

# So we will use tabs as our delimiter at the end of this process.

# We will use the RCurl package to download the NPS files.

# Normally we could just parse the XML (or html) content

# on-the-fly, without downloading the files, but in this case,

# it wasn't working on about 10 of the files, and somehow

# when I downloaded the files, it worked completely.

# I tried this several times, and just going ahead and downloading

# the files seems to be the most consistent solution.

install.packages("RCurl")

library(RCurl)

# We will use the XML package to parse the html (or XML) data

install.packages("XML")

library(XML)

# We will use the xlsx package to export the results at the end,

# into an xlsx file, for viewing in Microsoft Excel, if desired.

install.packages("xlsx")

library(xlsx)

# To see the list of the parks, we can go here:

# https://www.nps.gov/findapark/index.htm

# in any browser.

# In most browsers, if you navigate to a page and then type:

# Control-U (i.e., the Control Key and the letter U Key at once)

# on a Windows or UNIX machine,

# or if you type Command-U (i.e., the Command Key and the letter U Key at once)

# on an Apple Macintosh machine,

# then you can see the code for the way that the webpage is created.

# This webpage that I mentioned:

# https://www.nps.gov/findapark/index.htm

# has 1489 lines of code. Wow.

# From (roughly) lines 206 through 756, we see that the

# data for the parks are wrapped in a "div" (on line 206)

# and then in a "select" (on line 208)

# and then in an "optgroup" and then an "option".

# We want to extract the "value" of each "option".

# (We skip the "label" on line 205 because it ends on line 205 too.)

# So we do the following:

myparks <- xpathSApply(htmlParse(getURL("https://www.nps.gov/findapark/index.htm")), "//\*/div/select/optgroup/option", xmlGetAttr, "value")

myparks

# If the line of code (above) doesn't work,

# then perhaps you forgot to actually run the three "library" commands

# near the start of the file.

# We did a lot of things with 1 line of code.

# The "getURL" temporarily downloads all of the code from this webpage.

# We do not save the webpage, but rather, we send it to the htmlParse command.

# Once the page is parsed, we send the parsed results to the xpathSApply command.

# The pattern we want to look for is:

# "//\*/div/select/optgroup/option"

# The star means that anything is OK before this chunk of the pattern,

# but we definitely want our pattern to end with /div/select/optgroup/option

# and then we get the xmlGetAttr attribute called "value"

# which is one of the parks.

# When we check the results, we got 498 results:

length(myparks)

# For the Abraham Lincoln Birthplace, we want to run the following command,

# so that we are prepared to download the webpage.

# After downloading it, we will extract information from the parsed page:

system("mkdir ~/Desktop/myparks/")

download.file("https://www.nps.gov/abli/index.htm", "~/Desktop/myparks/abli.htm")

htmlParse("~/Desktop/myparks/abli.htm")

# but we want to do that for each park.

# So we build the following function:

myparser <- function(x) {

download.file(paste("https://www.nps.gov/", x, "/index.htm", sep=""), paste("~/Desktop/myparks/", x, ".htm", sep=""))

htmlParse(paste("~/Desktop/myparks/", x, ".htm", sep=""))

}

# Now, we apply this function to each element of "myparks"

# and we save the results in a variable called "mydocs":

mydocs <- sapply(myparks, myparser)

# The webpage for the Abraham Lincoln Birthplace is now parsed and stored here:

mydocs[[1]]

# The webpage for Zion National Park is now parsed and stored here:

mydocs[[498]]

# Next we look at the source for the Abraham Lincoln Birthplace:

# https://www.nps.gov/abli/index.htm

# We load that webpage in any browser and then type:

# Control-U if we are on a Windows or UNIX machine, or

# Command-U if we are on a Mac.

# Then we can search in this page (using Control-F on Windows or UNIX,

# or using Command-F on a Mac) for any pattern we want.

# If we search for "itemprop"

# we find the information about the address:

# They are all within a "span" tag, with different "itemprop" attributes:

# The street address has attribute: "streetAddress"

# The city has attribute: "addressLocality"

# The state has attribute: "addressRegion"

# The zip code has attribute: "postalCode"

# The telephone has attribute: "telephone"

# So, for instance, we can find all of these as follows:

xpathSApply(mydocs[[1]], "//\*/span[@itemprop='streetAddress']", xmlValue)

xpathSApply(mydocs[[1]], "//\*/span[@itemprop='addressLocality']", xmlValue)

xpathSApply(mydocs[[1]], "//\*/span[@itemprop='addressRegion']", xmlValue)

xpathSApply(mydocs[[1]], "//\*/span[@itemprop='postalCode']", xmlValue)

xpathSApply(mydocs[[1]], "//\*/span[@itemprop='telephone']", xmlValue)

# Then the title stuff:

xpathSApply(mydocs[[1]], "//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)

xpathSApply(mydocs[[1]], "//\*/span[@class='Hero-designation']", xmlValue)

xpathSApply(mydocs[[1]], "//\*/span[@class='Hero-location']", xmlValue)

# and, finally, the social media links:

paste(xpathSApply(mydocs[[1]], "//\*/div/ul/li[@class='col-xs-6 col-sm-12 col-md-6']/a", xmlGetAttr, "href"),collapse=",")

# Here are the versions for the entire data set:

streets <- sapply(mydocs, function(x) xpathSApply(x, "//\*/span[@itemprop='streetAddress']", xmlValue))

cities <- sapply(mydocs, function(x) xpathSApply(x, "//\*/span[@itemprop='addressLocality']", xmlValue))

states <- sapply(mydocs, function(x) xpathSApply(x, "//\*/span[@itemprop='addressRegion']", xmlValue))

zips <- sapply(mydocs, function(x) xpathSApply(x, "//\*/span[@itemprop='postalCode']", xmlValue))

phones <- sapply(mydocs, function(x) xpathSApply(x, "//\*/span[@itemprop='telephone']", xmlValue))

mynames <- sapply(mydocs, function(x) xpathSApply(x, "//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue))

mytypes <- sapply(mydocs, function(x) xpathSApply(x, "//\*/span[@class='Hero-designation']", xmlValue))

mylocations <- sapply(mydocs, function(x) xpathSApply(x, "//\*/span[@class='Hero-location']", xmlValue))

mylinks <- sapply(mydocs, function(x) paste(xpathSApply(x, "//\*/div/ul/li[@class='col-xs-6 col-sm-12 col-md-6']/a", xmlGetAttr, "href"),collapse=","))

# with some cleaning up:

streets <- sapply(streets, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

cities <- sapply(cities, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

states <- sapply(states, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

zips <- sapply(zips, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

phones <- sapply(phones, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

mynames <- sapply(mynames, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

mytypes <- sapply(mytypes, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

mylocations <- sapply(mylocations, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

mylinks <- sapply(mylinks, function(x) ifelse(length(x)==0,NA,sub("^\\s+","",sub("\\s+$","",x))), simplify=FALSE)

myDF <- data.frame(

streets=do.call(rbind,streets),

cities=do.call(rbind,cities),

states=do.call(rbind,states),

zips=do.call(rbind,zips),

phones=do.call(rbind,phones),

mynames=do.call(rbind,mynames),

mytypes=do.call(rbind,mytypes),

mylocations=do.call(rbind,mylocations),

mylinks=do.call(rbind,mylinks)

)

**Project 11:**

The names in the election data are in CAPITAL LETTERS!

When asking about names in the questions, we assume that you are using the names from the election data, available on Scholar.

You might want to practice on a smaller data set: /depot/statclass/data/election2018/itsmall.txt

The full data is available here: /depot/statclass/data/election2018/itcont.txt

We are assuming that you are using unique names from column 8, i.e., that you have already removed duplicates of any names of the donors.

Hint: Save column 8 (which contains the donor names) into a new variable. Then extract the unique values from the column using the "unique" command.

Answer these questions using the full data given above. BUT, for convenience, you might want to \*start\* by using the smaller data set to practice.

Please note that we can read the data into R using the command:

myDF <- read.csv("/depot/statclass/data/election2018/itsmall.txt", header=F, sep="|")

or, for the full data set:

myDF <- read.csv("/depot/statclass/data/election2018/itcont.txt", header=F, sep="|")

1. Find the number of (unique) donor names who have your first name,

embedded somewhere in the donor's name (not necessarily as the

first or last name--any location is OK).

2a. How many donors have a consecutive repeated letter in their name?

2b. How many donors have a consecutive repeated vowel in their name?

2c. How many donors have a consecutive repeated consonant in their name?

3. Just for fun: Come up with an interesting question about text patterns, and answer it yourself, using regular expressions. Of course you can compare questions and answers with another member of The Data Mine. Have fun!

Regular expressions enable us to find patterns in text.

Here are a handful of examples of regular expressions.

The best way to learn them in earnest is to just read some documentation about regular expressions and then try them!

Here is an example:

v <- c("me", "you", "mark", "laura", "kale", "emma", "err", "eat", "queue", "kangaroo", "kangarooooo", "kangarooooooooo")

The elements of v that contain the letter "m":

v[grep("m", v)]

containing the phrase "me":

v[grep("me", v)]

containing the letter "a":

v[grep("a", v)]

containing the letter "e":

v[grep("e", v)]

containing the letter "k":

v[grep("k", v)]

containing the letter "k" at the start of the word:

v[grep("^k", v)]

containing the letter "k" at the end of the word:

v[grep("k$", v)]

containing the letter "a" at the end of the word:

v[grep("a$", v)]

containing the letter "o" at the end of the word:

v[grep("o$", v)]

containing the letter "o" anywhere in the word:

v[grep("o", v)]

containing the letter "o" two times in a row, anywhere in the word:

v[grep("o{2}", v)]

containing the letter "o" three times in a row, anywhere in the word:

v[grep("o{3}", v)]

containing the letter "o" two to five times in a row, anywhere in the word:

v[grep("o{2,5}", v)]

containing the letter "q" followed by "ue":

v[grep("q(ue){1}", v)]

containing the letter "q" followed by "ue" two times:

v[grep("q(ue){2}", v)]

containing the letter "q" followed by "ue" three times:

v[grep("q(ue){3}", v)]

containing the letter "e" followed by "m" or "r":

v[grep("e(m|r)", v)]

again, same idea, but different way, to find words

containing the letter "e" followed by "m" or "r":

v[grep("e[mr]", v)]

containing the letter "e" followed by "ma" or "rr":

v[grep("e(ma|rr)", v)]

containing a repeated letter:

v[grep("([a-z])\\1", v)]

In this example, the \\1 refers to whatever was found in the first match

(which is just given in parentheses for convenience)

Here is a summary of regular expressions:

<https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285>

You are welcome to use any source or reference for regular expressions that you like.

We need to use double backslash for back-references, in R.

We gave a demonstration of this, in the last example given above.

In general, in R, when writing a backslash in a regular expression, a double backslash is usually needed.

**Project 14:**

Remind ourselves how to use bash and awk tools (previously we did this in the terminal).  
We will do it in Jupyter Notebooks this semester: <http://notebook.scholar.rcac.purdue.edu/>

1a.  Start a new Jupyter Notebook with type "bash" (instead of "R"). We are going to put bash code directly inside the Jupyter Notebook. (In the past, we only wrote bash code directly inside the terminal.)  
  
1b.  Look at the first 10 lines of the 2007 flight data, which is found at: /depot/statclass/data/dataexpo2009/2007.csv   
All of the flights in those first 10 lines are on the same carrier.  Which carrier is it? Remember that you can check: <http://stat-computing.org/dataexpo/2009/the-data.html>   
  
Now we are going to put awk code directly inside the Jupyter Notebook. (In the past, we only wrote awk code directly inside the terminal.)

2.  Save the information about every flight departing from Indianapolis since January 1, 2000 into a common file, named MyIndyFlights.csv

Hint 1:  You only need the files 2000.csv, 2001.csv, ..., 2008.csv You can work on all of those files at once, using 2\*.csv because the "\*" is like a wildcard, that matches any pattern.  
  
Hint 2:  You can use awk to do this. For comparison, ONLY as an example, we can extract all flights  
that are on Delta airlines in 1998 as follows:  
cat /depot/statclass/data/dataexpo2009/1998.csv | awk -F, '{ if($9 == "DL") {print $0} }' >MyDeltaFlights.csv

Project 14 solutions   
  
# 1.  The head of the file with the 2007 flights is:   
head /depot/statclass/data/dataexpo2009/2007.csv   
  
#     We see that the UniqueCarrier is found in column 9.   
#     One way to extract the UniqueCarrier is with the cut command   
#     using a comma as the delimiter and retrieving (cut out) the 9th column:   
cut -d, -f9 /depot/statclass/data/dataexpo2009/2007.csv | head -n11   
#     We only displayed the head, because we only want the first 10 flights.   
#     We specified -n11 because this prints the first 11 lines of the file,   
#     namely, the header itself, and the first 10 flights.   
#     We can check the data dictionary, available at: <http://stat-computing.org/dataexpo/2009/>   
#     The information about the carrier codes is found there,   
#     by clicking on the link for supplemental data sources: <http://stat-computing.org/dataexpo/2009/supplemental-data.html>   
and then choosing the carriers file: <http://stat-computing.org/dataexpo/2009/carriers.csv>   
#     The carrier code "WN" for each of these first ten flights is Southwest.   
  
# 2.  We save the information about the Indianapolis flights by using awk.   
#     First we recall how to see the information about all such flights.   
#     Here are the first 10 lines of that data.   
cat /depot/statclass/data/dataexpo2009/2\*.csv | head   
#     Then we change the "head" to the "awk" command.   
#     We use comma as the field separator   
#     (this is the same as the role of the delimiter from cut)   
#     We modify the example from the project assignment,   
#     so that we focus on the 17th field (which are the Origin airports)   
#     and we save the resulting data into a file called MyIndyFlights.csv   
  
cat /depot/statclass/data/dataexpo2009/2\*.csv | awk -F, '{ if($17 == "IND") {print $0} }' >MyIndyFlights.csv   
#     Some of you were not working in your home directory when you ran this commmand.   
#     If you want to be sure to save the file into your home directory,   
#     remember that you can explicitly specify your home directory using a tilde, as follows:   
cat /depot/statclass/data/dataexpo2009/2\*.csv | awk -F, '{ if($17 == "IND") {print $0} }' >~/MyIndyFlights.csv   
#     It is not required that you check things,   
#     but if you want to check that things worked properly, you can use the wc command   
#     which gives the number of lines, words, and bytes in the resulting file:   
wc MyIndyFlights.csv   
#     or, even more explicitly,   
wc ~/MyIndyFlights.csv   
#     An alternative is to check the head and the tail:   
head MyIndyFlights.csv   
tail MyIndyFlights.csv   
#     or, even more explicitly,   
head ~/MyIndyFlights.csv   
tail ~/MyIndyFlights.csv

**Project 15:**

Remind ourselves how to use R tools (previously we did this in the terminal). We will do it in Jupyter Notebooks this semester.  
  
1a.  Start a new Jupyter Notebook with type "R"  
  
1b.  Import the flight data from the file MyIndyFlights.csv in a data frame.  You just created this file in Project 14. It contains all of the flights that departed from Indianapolis since January 1, 2000. (There should be 356561 flights altogether, and there is no header.) Hint:  When you import the data, if you use the read.csv command, there is no header, so be sure to use header=FALSE.  
  
1c.  What are the five most popular destinations for travelers who depart Indianapolis since January 1, 2000? List each of these 5 destinations, and the number of flights to each one.  
  
2a.  Consider the year 2005 (only).  Tabulate the number of flights per day.  
  
2b.  On each of the most popular five days, how many flights are there?  
  
2c.  On each of the least popular five days, how many flights are there?  
  
Hint:  You might be surprised to see the wide range of the number of flights per day!

Project 15 solutions   
  
# 1.  We first import the flight data from the file MyIndyFlights.csv   
  
myDF <- read.csv("MyIndyFlights.csv", header=F)   
  
# or, if you prefer to explicitly state that the file   
# is in your home directory, you can add the tilde for your home:   
  
myDF <- read.csv("~/MyIndyFlights.csv", header=F)   
  
# We check that there are 356561 flights altogether:   
  
dim(myDF)   
  
# The five most popular destinations for travelers   
# who depart Indianapolis since January 1, 2000 are:   
tail(sort(table(myDF[[18]])),n=5)   
  
# We used the 18th column, which has the Destination airports.   
# We tabulated the results, using the table command,   
# and then we sorted the results.   
# Finally, at the end, we took the tail of the results,   
# using n=5, since we wanted to see the largest 5 values.   
  
# 2a.  We load the 2005 data:   
  
myDF <- read.csv("/depot/statclass/data/dataexpo2009/2005.csv")   
  
# To get the number of flights per day,   
# we can first paste together the Month and Day columns.   
# We check the head, to make sure that this worked:   
  
head(paste(myDF$Month, myDF$DayofMonth))   
  
# It is also possible, for instance, to separate the   
# month and the day by separators, such as a slash:   
  
head(paste(myDF$Month, myDF$DayofMonth, sep="/"))   
  
# or a dash:   
  
head(paste(myDF$Month, myDF$DayofMonth, sep="-"))   
  
# Now we can tabulate the number of flights per day,   
# using the table command:   
  
table(paste(myDF$Month, myDF$DayofMonth, sep="/"))   
  
# 2b. To find the most popular five days,   
#     we can sort the table, and then consider the tail,   
#     using the n=5 option,   
#     since we only want the 5 most popular dates.   
  
tail(sort(table(paste(myDF$Month, myDF$DayofMonth, sep="/"))),n=5)   
  
# 2c. We just change tail to head,   
#     to find the 5 least popular dates:   
  
head(sort(table(paste(myDF$Month, myDF$DayofMonth, sep="/"))),n=5)

**Project 16:**

Project 16 needs to be saved as a .ipynb file. This is different from the previous two assignments where the file was uploaded directly from each students Github page. Students need to download it from this link. Thanks!

<https://raw.githubusercontent.com/TheDataMine/STAT-19000/master/Assignments/hw16.ipynb>

1.  Consider the flights from 2005 in the Data Expo 2009 data set.  The actual departure times, as you know, are given in the DepTime column.

In this question, we want to categorize the departure times according to the hour of departure.  For instance, any time in the 4 o'clock in the (very early morning) hour should be classified together.  These are the times between 0400 and 0459 (because the times are given in military time).

One way to do this is to divide each of the times by 100, and then to take the "floor" of the results, and then make a "table" of the results.  For practice (just to understand things), give this a try with the head of the DepTime, one step at a time, to make sure that you understand what is happening.  Then:

1a.  Classify all of the 2005 departure times, according to the hour of departure, using this method.

1b.  During which hour of the day did the most flights depart?

2a.  Here is another way to solve the question above.  Read the documentation for the "cut" command.  For the "breaks" parameter, use:

seq(0, 2900, by=100)

and be sure to set the parameter "right" to be FALSE.

2b.  Check that you get the same result as in question 1, using this method.

2c.  Why did we choose to use 2900 instead of (say) 2400 in this method?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Project 16 Solutions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*   
  
# 1a.  We read the data from the 2005 flights into a data frame   
  
myDF <- read.csv("/depot/statclass/data/dataexpo2009/2005.csv")   
  
#      Then we divide each time by 100 and take the floor:   
  
table(floor(myDF$DepTime/100))   
  
#      and we get:   
  
#      0      1      2      3      4      5      6      7 8      9     10   
#  21747   7092   2027    458   1610 114469 430723 440532 469386 447705 432526   
#     11     12     13     14     15     16     17     18     19 20     21   
# 446432 443252 440903 416661 441021 424299 457678 431613 390398 321680 235810   
#     22     23     24     25     26     27     28   
# 128382  58386   1711    301     56      7      1   
  
# 1b.  The most flights departed during 8 AM to 9 AM;   
  
sort(table(floor(myDF$DepTime/100)))   
  
#     28     27     26     25      3      4     24      2 1      0     23   
#      1      7     56    301    458   1610   1711   2027   7092 21747  58386   
#      5     22     21     20     19     14     16      6     18 10      7   
# 114469 128382 235810 321680 390398 416661 424299 430723 431613 432526 440532   
#     13     15     12     11      9     17      8   
# 440903 441021 443252 446432 447705 457678 469386   
  
# 2a.  We cut the DepTime column, using the breaks of 0000 through 2900   
  
table(cut(myDF$DepTime, breaks=seq(0000,2900,by=100), right=FALSE))   
  
#      and we get:   
  
#           [0,100)         [100,200)         [200,300) [300,400)   
#             21747              7092 2027               458   
#         [400,500)         [500,600)         [600,700) [700,800)   
#              1610            114469            430723 440532   
#         [800,900)       [900,1e+03)   [1e+03,1.1e+03) [1.1e+03,1.2e+03)   
#            469386            447705            432526 446432   
# [1.2e+03,1.3e+03) [1.3e+03,1.4e+03) [1.4e+03,1.5e+03) [1.5e+03,1.6e+03)   
#            443252            440903            416661 441021   
# [1.6e+03,1.7e+03) [1.7e+03,1.8e+03) [1.8e+03,1.9e+03) [1.9e+03,2e+03)   
#            424299            457678            431613 390398   
#   [2e+03,2.1e+03) [2.1e+03,2.2e+03) [2.2e+03,2.3e+03) [2.3e+03,2.4e+03)   
#            321680            235810            128382 58386   
# [2.4e+03,2.5e+03) [2.5e+03,2.6e+03) [2.6e+03,2.7e+03) [2.7e+03,2.8e+03)   
#              1711               301 56                 7   
# [2.8e+03,2.9e+03)   
#                 1   
  
#      or if you want to re-format the output, you can write, for instance:   
  
table(cut(myDF$DepTime, breaks=seq(0000,2900,by=100), dig.lab=4, right=FALSE))   
  
#     [0,100)   [100,200)   [200,300)   [300,400)   [400,500) [500,600)   
#       21747        7092        2027         458        1610 114469   
#   [600,700)   [700,800)   [800,900)  [900,1000) [1000,1100) [1100,1200)   
#      430723      440532      469386      447705      432526 446432   
# [1200,1300) [1300,1400) [1400,1500) [1500,1600) [1600,1700) [1700,1800)   
#      443252      440903      416661      441021      424299 457678   
# [1800,1900) [1900,2000) [2000,2100) [2100,2200) [2200,2300) [2300,2400)   
#      431613      390398      321680      235810      128382 58386   
# [2400,2500) [2500,2600) [2600,2700) [2700,2800) [2800,2900)   
#        1711         301          56           7           1   
  
#     We just sort the command above, and we see that   
#     the most flights departed during 8 AM to 9 AM   
  
sort(table(cut(myDF$DepTime, breaks=seq(0000,2900,by=100), dig.lab=4, right=FALSE)))   
  
# [2800,2900) [2700,2800) [2600,2700) [2500,2600)   [300,400) [400,500)   
#           1           7          56         301         458 1610   
# [2400,2500)   [200,300)   [100,200)     [0,100) [2300,2400) [500,600)   
#        1711        2027        7092       21747       58386 114469   
# [2200,2300) [2100,2200) [2000,2100) [1900,2000) [1400,1500) [1600,1700)   
#      128382      235810      321680      390398      416661 424299   
#   [600,700) [1800,1900) [1000,1100)   [700,800) [1300,1400) [1500,1600)   
#      430723      431613      432526      440532      440903 441021   
# [1200,1300) [1100,1200)  [900,1000) [1700,1800)   [800,900)   
#      443252      446432      447705      457678      469386   
  
#   2b. We do get the same results as in question 1.   
  
#   2c.  We choose to use 2900 instead of (say) 2400 in this method   
#        because some flights departed after midnight.   
#        The time stamps are between 0000 and 2400   
#        (this is like military time, between 00:00 and 24:00).   
#        Some flights have delays until after midnight,   
#        and they are recorded in a surprising way,   
#        e.g., 24:30 for 30 minutes past midnight,   
#        or 26:10 for 2 hours and 10 minutes past midnight.   
#        In our data set, it happens that all of the ranges of the times   
#        are between 0000 and 2900.  I just checked the max to find that out.   
#        So that's why we use 2900 as an upper boundary, instead of 2400.   
  
max(myDF$DepTime, na.rm=T)

Project 17:

Please download this template and use it to submit your solutions to GitHub:

<https://raw.githubusercontent.com/TheDataMine/STAT-19000/master/Assignments/hw17.ipynb>

Recall the 2018 election data, available here: /depot/statclass/data/election2018/itcont.txt

and the data dictionary for this data, which is available here:

<https://www.fec.gov/campaign-finance-data/contributions-individuals-file-description>

1a. Use the system command in R to read the data for the first 100,000 donations and store this data into a file called: shortfile.txt (We use .txt instead of .csv because the file is not comma delimited.)

1b. Use the read.csv command to read this data into a data frame in R, called: myDF

(Hint: check the help for read.csv: ?read.csv to remind yourself about the "sep" and

the "header" parameters for read.csv. In particular, this data has "|" as the separator between

the data elements, and it does not have a header.)

1c. Check the dimension of the resulting data frame. It should be 100,000 rows and 21 columns.

2a. Split the data for these 100,000 donations according to the State from which the donation was given. Store the resulting data in a list called: myresult

(Hint: Check the data dictionary for the meanings of the columns, since we do not have column headers.)

(Another hint: Remember that we can refer to a column of data in a data frame by its number, for instance, myDF[[8]] is the name of the donor.)

2b. Check the names of myresult: names(myresult)

We see the the first element of the list does not have a name. This is a pain! To solve this,

you can give it a name, for instance, by writing: names(myresult)[1] <- "unknown"

(or any other kind of name that you want, to indicate that the name is unknown)

3a. Find the mean donation amount, according to each state.

3b. What is the mean donation from Hoosiers (i.e., for people from Indiana)?

3c. Find the standard deviation of the donation amount, according to each state.

3d. Find the number of donations, according to each state.

3e. For a sanity check, make sure that the number of donations in 3d adds up to 100,000 altogether.

Example:

# Remember that we can make system calls from R.

# For instance, we can take the first 50000 lines of a file

# and store them into a new file called shortfile.csv

# To do this, we use the "system" command in R.

# it basically enables us to run terminal commands

# while we are still working in R.

# This is an especially handy technique,

# because the operating system itself is much faster than R.

system("head -n50000 /depot/statclass/data/dataexpo2009/2005.csv >shortfile.csv")

# Now we can read this (much shorter!) file into R.

myDF <- read.csv("shortfile.csv")

# It has data about only 49,999 flights because the header itself

# counts as one of the 50,000 lines that we extracted.

dim(myDF)

# We can check to make sure that the read.csv worked,

# by examining the head of myDF:

head(myDF)

# Within myDF, we can break the data into pieces,

# according to (say) the Origin airport.

# The split command can easily do that for us.

# We give the split command 2 pieces of data:

# 1. The data that should be split, and

# 2. The way that the data is classified into pieces.

# So, for instance, we can split the DepDelays

# into pieces, based on the Origin.

myresult <- split(myDF$DepDelay, myDF$Origin)

# If we check the length of the result, it is 93:

length(myresult)

# because there are DepDelays from 93 airports.

# The type of data is a "list".

class(myresult)

# We have not (yet) worked with lists,

# but they are a lot like data frames.

# The difference is that each column can have a different length.

# For example, here are the first six columns

# of the list:

head(myresult)

# The flights to Albuquerque are found in the second column:

myresult$ABQ

# or we can get this data by just asking directly for the second column,

# without knowing the name of the column:

myresult[[2]]

# Now we can use the power of the apply functions that R provides.

# You are already familiar with the tapply function.

# Another very commonly used apply function is called "sapply".

# We use sapply to apply a function to each part of a collection of data.

# For example, remember that myresult has 93 parts:

length(myresult)

# We can take the mean of the data in each element of myresult

# by applying the function "mean" to each element, as follows:

sapply(myresult, mean)

# Unfortunately, many of the results are NA's, so we can use na.rm=T

sapply(myresult, mean, na.rm=T)

# We can apply many functions to myresult in this way.

# For instance, here is the variance of each part of the data in myresult:

sapply(myresult, var, na.rm=T)

# or the standard deviation:

sapply(myresult, sd, na.rm=T)

# Here is the number of flights from each Origin airport:

sapply(myresult, length)

# If we add up the number of flights, we better get 49,999:

sum(sapply(myresult, length))

# It is worthwhile to experience with sapply.

# For instance, for something fun to try,

# you can (simultaneously) make a plot of the DepDelays

# from each of the 93 airports, as follows:

sapply(myresult, plot)

# This runs the "plot" function on each piece of data,

# in other words, on the data from each Origin airport.

# You can see the first 6 DepDelays from each Origin airport, as follows:

sapply(myresult, head)

# This is taking the "head" of each part of the data.

Project 17 Solutions

# 1a.  We first store the first 100,000 donations into a file

#      called shortfile.txt

#      using the system command

system("head -n100000 /depot/statclass/data/election2018/itcont.txt >~/shortfile.txt")

# 1b.  Now we import this data into the read.csv file

myDF <- read.csv("~/shortfile.txt", header=F, sep="|")

# 1c.  The resulting data frame has 100000 rows and 21 columns, as it should!

dim(myDF)

# 2a.  Now we split the data for the donations according to the State

#      from which the donation was given

myresult <- split(myDF$V15, myDF$V10)

# 2b.  We check the names of myresult:

names(myresult)

# and the first element of the list does not have a name.

# so we give it a name, for instance, by writing:

names(myresult)[1] <- "unknown"

# 3a.  The mean donation amount, from each state,

#      can be found using the sapply command:

sapply(myresult, mean, na.rm=T)

# 3b.  The mean donation from Indiana can be found

#      by extracting the entry with the name "IN"

sapply(myresult, mean, na.rm=T)["IN"]

# and we get:    IN: 367.914678899083

# 3c.  The standard deviation of the donation amount for each state is:

sapply(myresult, sd, na.rm=T)

# 3d.  The number of donations per state can be found by

#      checking the length of the vector of donations from each state:

sapply(myresult, length)

# 3e.  For our sanity check, we see that yes, indeed,

#      the total number of donations is 100,000:

sum(sapply(myresult, length))

**Project 18:**

Here is the Project 18 template:

<https://raw.githubusercontent.com/TheDataMine/STAT-19000/master/Assignments/hw18.ipynb>

Consider the election data stored at:  /depot/statclass/data/election2018/itcont.txt  
The data set is very large.  You might choose to analyze a smaller portion of the data initially, and then to run your code on the full data set, once you have the code working correctly.  
  
Sometimes there will be warnings in Jupyter Notebooks, and you need to scroll past the warnings, to see the results of your analysis.  This is a known issue with Jupyter Notebooks, and other people are experiencing it too: <https://github.com/IRkernel/IRkernel/issues/590>   
  
Recall that the data dictionary for the data is found here:  
<https://www.fec.gov/campaign-finance-data/contributions-individuals-file-description>   
  
1a.  The first column contains the "Filer identification number" for various committees.  
     Which of these committees received the largest monetary amount of donations?  
  
1b.  Use the tapply function to make a matrix whose rows correspond to states, whose columns correspond to the "filer identification numbers" of committees, and whose entries contain the total amount of the donations given to the committees, by donors from each individual state.  
Hint: Wrap the states and the filer identification numbers into a list.  
Print the block of first 10 rows and 10 columns, so that the TA's can see the results of your work.  
  
2.   For this question, be sure to take into account the city and state (together).  
  
2a.  Identify the six cities that made the largest number of donations.  
  
2b.  Identify the six cities that made the largest monetary amount of funding donated.  
  
3a.  Split the data (using the split command) about the donations, according to the day when the transaction was made.  
  
     Once this split is accomplished, use the sapply function to find the following:  
3b.  On which day was the total monetary amount of donations the largest?  
3c.  On which day was the largest number of donations made?

# Examples that might help with Project 18 (but are using the airline data set)  
# We can read in the 2005 flight data:  
myDF <- read.csv("/depot/statclass/data/dataexpo2009/2005.csv")  
# and verify that we got it read in properly, using the head:  
head(myDF)  
# We can find the mean DepDelays, according to the Origin and Destination (simultaneously).  
# This puts the Origins on the rows and the Destinations on the columns.  
tapply(myDF$DepDelay, list(myDF$Origin, myDF$Dest), mean, na.rm=T)  
# If you just want to see the first 10 rows and columns,  
# you can save the results to a variable:  
myresult <- tapply(myDF$DepDelay, list(myDF$Origin, myDF$Dest), mean, na.rm=T)  
# and then load the rows and columns that you want to see:  
myresult[1:10,1:10]  
# Many are NA because you can't always get from one city to another.  
# You can lookup specify Origins and Destinations as follows:  
myresult[c("DEN","ORD","JFK"),c("BOS","IAD","ATL")]  
# Those are flights from Origin "DEN" or "ORD" or "JFK" to Destinations "BOS" or "IAD" or "ATL"  
# Here is another example:  
# We can split all of the data about the DepDelays, according to the date.  
# To do this, I first need to make a column that contains the dates,  
# since the airport data doesn't have such a column (yet):  
myDF$completedates <- paste(myDF$Month, myDF$DayofMonth, myDF$Year, sep="/")  
# Then we split the DepDelays, according to the dates:  
mydelays <- split(myDF$DepDelay, myDF$completedates)  
# This gives us a list:  
class(mydelays)  
# Of course the length is 365, because there are 365 days per year:  
length(mydelays)  
# Here are the delays from Christmas Day:  
mydelays["12/25/2005"]  
# Now we can easily use the sapply function on  
# the DepDelay data, which has already been grouped according to the days.  
# Here is the mean DepDelay on each day:  
sapply(mydelays, mean, na.rm=T)  
# Here is the standard deviation of the DepDelay, on each day:  
sapply(mydelays, sd, na.rm=T)  
# Here is the length of each piece of the data,  
# i.e., the number of pieces of data per day.  
# (This is obviously equal to the number of flights per day too,  
#  because each flights has \*some kind\* of delay!)  
sapply(mydelays, length)

# Project 18 Solutions:

# 1a.  The committee C00401224 received $565007473 in donations altogether.

tail(sort(tapply(myDF$V15, myDF$V1, sum, na.rm=T)))

# Here are the top six committees, according to the total monetary donations:

# C00000935   109336606

# C00571703   114336858

# C00003418   116712977

# C00484642   130390881

# C00504530   133582635

# C00401224   565007473

# 1b.  We first build a matrix with the data from the states (column 10) on the rows

#      and the data from the committees (column 1) on the columns.

#      Each entry have the analogous sum of the sum of the donations.

myresult <- tapply(myDF$V15, list(myDF$V10, myDF$V1), sum, na.rm=T)

# Now we display the results of the first 10 rows and 10 columns:

myresult[1:10,1:10]

#    C00000059 C00000422 C00000638 C00000729 C00000885 C00000901 C00000935 C00000984 C00001016 C00001180

#           NA        NA        NA        NA        NA        NA    174182        NA        NA        NA

# AA        NA        NA        NA        NA        NA        NA     15336        NA        NA        NA

# AE        NA        NA        NA        NA        NA        NA     13122        NA        NA        NA

# AK        NA      5148        NA      4384      1985     23135    175850        NA      8674        NA

# AL        NA      7152        NA      9722      1868    103106    407595        NA     13518        NA

# AP        NA        NA        NA        NA        NA        NA      4705        NA        NA        NA

# AR       420      9994        NA      5750      1406     13910    183457      5000     12730        NA

# AS        NA        NA        NA        NA        NA        NA        NA        NA        NA        NA

# AZ        NA     10074        NA     26778       615     17040   1223310        NA     12488        NA

# CA        NA     89498        NA     41752     31705    108253  28039517      5000    256676        NA

# 2a. We paste together the city and state data using the paste function.

#     Then we tabulate the number of such donations, according to these city-state pairs.

#     Finally, we sort these counts and print the six largest ones, using the tail function.

tail(sort(table(paste(myDF$V9,myDF$V10))))

# 2b. We paste together the city and state data using the paste function.

#     Then we add the monetary amount of the donations (from column 15),

#     according to these city-state pairs.

#     Finally, we sort these total monetary amounts and

#     print the six largest ones, using the tail function.

tail(sort(tapply(myDF$V15,paste(myDF$V9,myDF$V10),sum,na.rm=T)))

# 3a. We split the data about the donation amounts (from column 15),

#     according to the day on which the donations were made.

myresult <- split(myDF$V15, myDF$V14)

# 3b. Now we sum the monetary amount of the donations, for each day:

tail(sort(sapply(myresult, sum, na.rm=T)))

# 3c. Alternatively, we see how many donations were made on each day,

#     by finding the length of the vector that has the donations for that day,

#     i.e., by finding how many donations there were for each day.

tail(sort(sapply(myresult, length)))

Project 20:

\*\*\*\* Please submit your answers, when you are finished, using GitHub.  We put an RMarkdown file into your individual GitHub accounts, for this purpose.  
  
Notes about scraping data:  
  
As a gentle reminder about how to access RStudio:  
  
Log on to Scholar:  
  
https://desktop.scholar.rcac.purdue.edu  
  
(or use the ThinLinc client on your computer if you installed it!)  
  
open the terminal on Scholar and type:  
  
module load gcc/5.2.0  
module load rstudio  
rstudio &

Please remember to install and load the XML and the RCurl libraries.  
  
Using RStudio, we start to learn how to extract data from the web.  
  
Use the data from the Billboard Hot 100 for question 1.  
Please use the data from the week you were born.  For instance, if I solve question 1, I would use the data located here:  
https://www.billboard.com/charts/hot-100/1976-10-13  
  
On the Hot 100 chart, from the day of your birth:  
  
1a.  Extract the titles of the songs ranked #2 through #100.  
  
1b.  Extract the artists for those 99 songs.  
  
1c.  Extract the title of the number 1 song for that day.  
  
1d.  Extract the artist for the number 1 song for that day.  
  
2a.  Extract the city where the National Park property for Catoctin Mountain is located.  This data is found at:  
https://www.nps.gov/cato/index.htm  
or in the file:  
/depot/statclass/data/parks/cato.htm  
  
2b.  Extract the state where Catoctin Mountain is located.  
  
2c.  Extract the zip code where Catoctin Mountain is located.  
  
3a.  Identify three potential websites that you are interested to try to scrape yourself, during the upcoming seminars.  
Look for websites with data that is (relatively) easy to scrape, for instance:  
Systematic URL’s that are easy to understand;  
(relative) consistency in how the data is stored; and  
make sure that the data is embedded in the page, rather than in csv files that are already prepared for download.  
(We want to actually scrape some data.)  
  
1.   
2.   
3.   
  
3b.  For each of the three websites that you identified, give a very brief description of the kind of data that you want to scrape.  
  
1.   
2.   
3.

# Project 20 Billboard Example

install.packages("XML")

library(XML)

# Considering the songs and artists who sang popular songs at the time of my birthday in 1976, we can scrape some data from Billboard Hot 100 chart

# Here are the songs titles #2 through #100 from my birthday

# Please notice the double underscore before title:

xpathSApply(htmlParse(getURL("https://www.billboard.com/charts/hot-100/1976-10-13")),

"//\*/div[@class='chart-list-item\_\_title']", xmlValue)

# Here are the artists of the songs #2 through #100 from my birthday

# Please notice the double underscore before artist:

xpathSApply(htmlParse(getURL("https://www.billboard.com/charts/hot-100/1976-10-13")),

"//\*/div[@class='chart-list-item\_\_artist']", xmlValue)

# Project 20 National Park Service Example

# We will use the XML package to parse html (or XML) data

install.packages("XML")

library(XML)

# and the RCurl package if you want to pull the data directly from the web:

install.packages("RCurl")

library(RCurl)

# To see the list of the parks, we can go here:

#    https://www.nps.gov/findapark/index.htm

# if you use Control-U

# (i.e., the Control Key and the letter U Key at once)

# then you can see the code

# for the way that the webpage is created.

# You can use Firefox to open any of the files

# with the data from the state parks;

# they are all found inside this directory:

#   /depot/statclass/data/parks/

########################################

# To study a specific park,

# we look at the source for the Abraham Lincoln Birthplace:

#   https://www.nps.gov/abli/index.htm

# We load that webpage in a browser and then type Control-U

# You search the code in a page with Control-F in Firefox

# Here is the name of the Abraham Lincoln Birthplace:

xpathSApply(htmlParse(getURL("https://www.nps.gov/abli/index.htm")), "//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)

# Here is the street address:

xpathSApply(htmlParse(getURL("https://www.nps.gov/abli/index.htm")), "//\*/span[@itemprop='streetAddress']", xmlValue)

# Alternatively, we can also do this with the file itself,

# instead of pulling the data from the web:

xpathSApply(htmlParse("/depot/statclass/data/parks/abli.htm"), "//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)

xpathSApply(htmlParse("/depot/statclass/data/parks/abli.htm"), "//\*/span[@itemprop='streetAddress']", xmlValue)

# Project 20 Answers

install.packages("XML")

library(XML)

# 1a.  here are the song titles, 2 through 100, from (for instance) January 20, 1990:

#      but students should use their OWN BIRTHDAYS for this question.

xpathSApply(htmlParse(getURL("https://www.billboard.com/charts/hot-100/2000-01-20")),

"//\*/div[@class='chart-list-item\_\_title']", xmlValue)

# 1b.  here are the artists of the songs 2 through 100:

xpathSApply(htmlParse(getURL("https://www.billboard.com/charts/hot-100/2000-01-20")),

            "//\*/div[@class='chart-list-item\_\_artist']", xmlValue)

# 1c.  here is the title of the number 1 song from that week:

xpathSApply(htmlParse(getURL("https://www.billboard.com/charts/hot-100/2000-01-20")),

            "//\*/div[@class='chart-number-one\_\_title']", xmlValue)

# 1d.  here is the artist for the number 1 song from that week:

xpathSApply(htmlParse(getURL("https://www.billboard.com/charts/hot-100/2000-01-20")),

            "//\*/div[@class='chart-number-one\_\_artist']", xmlValue)

# 2a. Here is the city:

xpathSApply(htmlParse(getURL("https://www.nps.gov/cato/index.htm")),

            "//\*/span[@itemprop='addressLocality']", xmlValue)

# alternatively:

xpathSApply(htmlParse("/depot/statclass/data/parks/cato.htm"),

            "//\*/span[@itemprop='addressLocality']", xmlValue)

# 2b. Here is the state:

xpathSApply(htmlParse(getURL("https://www.nps.gov/cato/index.htm")),

            "//\*/span[@itemprop='addressRegion']", xmlValue)

# alternatively:

xpathSApply(htmlParse("/depot/statclass/data/parks/cato.htm"),

            "//\*/span[@itemprop='addressRegion']", xmlValue)

# 2c. Here is the zip:

xpathSApply(htmlParse(getURL("https://www.nps.gov/cato/index.htm")),

            "//\*/span[@itemprop='postalCode']", xmlValue)

# alternatively:

xpathSApply(htmlParse("/depot/statclass/data/parks/cato.htm"),

            "//\*/span[@itemprop='postalCode']", xmlValue)

# 3a, 3b answers will vary

Project 21:

Please use this template to submit Project 21: <https://raw.githubusercontent.com/TheDataMine/STAT-19000/master/Assignments/hw21.Rmd>

This project is supposed to be an easy modification of the project example,   
since it is almost time for Spring Break!   
  
1.  Modify the NPS example to extract the city location for every National Park.   
2.  Same question, for the state location for every National Park.   
3.  Same question, for the zip code for every National Park.   
  
Note:  Do not worry if some of the results have extra spaces.  We can deal with that later!

Project 21 Example:

library(RCurl)

library(XML)

# The webpage for the National Park Service includes

# only a little information about every NPS property:

# https://www.nps.gov/findapark/index.htm

# Importantly, it has the 4-letter codes for each property.

# If you type Control-U, then you can see the source for the page.

# Scroll down, and you will see on

# lines 210 through 753 these 4-letter codes

# (It might not be exactly lines 210 through 753 because the NPS

# modifies its webpages, just like any organization does!)

# Each such NPS property has the 4-letter code as

# an attribute to one of the XML tags. They are all found inside

# of a "select" tag,

# and then inside an "optgroup" tag,

# and then inside an "option" tag.

# You extract this XML value using the xmlGetAttr, like this:

myparks <- xpathSApply(htmlParse(getURL("https://www.nps.gov/findapark/index.htm")), "//\*/div/select/optgroup/option", xmlGetAttr, "value")

# and then we see the full listing of all 497 of these 4-digit codes here:

myparks

# Last week, we already learned how to extract the street address of a park.

# For instance, this is the name of the Abraham Lincoln Birthplace:

xpathSApply(htmlParse(getURL("https://www.nps.gov/abli/index.htm")),

"//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)

# Similarly, this is the name of Catoctin Mountain:

xpathSApply(htmlParse(getURL("https://www.nps.gov/cato/index.htm")),

"//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)

# Here's the name of the Great Smoky Mountains;

# we just change "abli" or "cato" to "grsm" and we have it!

xpathSApply(htmlParse(getURL("https://www.nps.gov/grsm/index.htm")),

"//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)

# In general, we could paste in the 4-digit letter of the park, like this:

x <- "abli"

xpathSApply(htmlParse(getURL( paste0("https://www.nps.gov/", x, "/index.htm"))),

"//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)

# where the value of "x" is the park's 4-digit abbreviation.

# Let's try to get these two park names simultaneously now.

# We build a function to do so:

mynameextractor <- function(x) {xpathSApply(htmlParse(getURL( paste0("https://www.nps.gov/", x, "/index.htm"))),

"//\*/div[@id='HeroBanner']/div/div/div/a", xmlValue)}

# and then we apply it to each of these 4-letter codes:

sapply( c("abli", "cato", "grsm"), mynameextractor )

# One thing about scraping data from the web is that

# there are always "hiccups" in the process,

# i.e., there are always challenges.

# For instance, we have codes for "cbpo" and "foca"

# but those pages do not actually exist (yet).

# So we need to remove them from our list of 4-letter codes:

mygoodparks <- myparks[(myparks != "cbpo")&(myparks != "foca")]

# Now we are ready to apply our function to

# all the NPS properties. We do it first to the "head",

# just to make sure things are working:

myresults <- sapply( head(mygoodparks), mynameextractor )

myresults

# and if this worked, then we apply it to the full list of parks.

# P.S. Depending on your web connection, and how many

# students do this at one time, you might need to run

# this a few times. It did not work quite right for me

# on the first try, but that is the nature of websites,

# i.e., sometimes there are failures and/or service interruptions,

# but it should generally work in just a few minutes!

myresults <- sapply( mygoodparks, mynameextractor )

# Finally, here are the names of all the park properties:

myresults

**Project 22:**

Here is an \*optional\* Project 22.  You don't need to do it, but if you choose to do it, we will count it as a replacement for your lowest previous project grade.

In this folder on Scholar: /depot/statclass/data/examples there is a program called "challenge", so you can run it by typing in the terminal something like this: /depot/statclass/data/examples/challenge 111

Here is the goal:  You can try to make a program (in any language) that converts strings of digits to strings of letters, by substituting

1 -> a

2 -> b

3 -> c

......

26 -> z

Please notice that we do \*not\* say

01 -> a

but rather, we say

1 -> a

The program should print the number of ways to do this.

So, for instance, if you type:

/depot/statclass/data/examples/challenge 111

It will return the number 3 because there are exactly 3 ways to decode the string 111, namely:

ak

ka

aaa

Makes sense? Here is another example:

/depot/statclass/data/examples/challenge 15114

will return the number 6 because there are exactly 6 ways to decode the string 15114, namely:

aeaad

aean

aekd

oaad

oan

okd

The challenge (again, only for bonus credit) is to write a program that will produce the same results as the program that I gave you.  You are welcome to use any programming language.

**Project 23:**

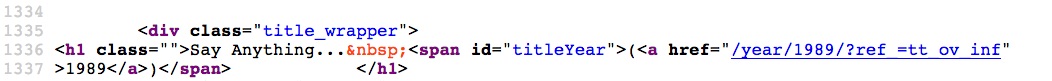
Here is the project.  We will build on this project in the upcoming work that we will do in April.  
Recall that in Project 20, question 3ab, you identified some websites that you were interested to scrape. Pick only 1 of the websites that is of interest to you, and scrape at least 5 pieces of information from a few pages within that website. (I am being a little nebulous here, because I want you to have the freedom to explore!) For instance, you could pick IMDB as the website and scrape 3 pieces of information from 5 different movies. BUT you can pick any website.  It does \*NOT\* need to be the IMDB website.  You can do any website you like. That's the entire assignment for this week! If you are not able to do it for the sites that you mentioned in Project 20, then you can (instead) identify a different website to scrape.  
  
# We recall that we can scrape information (which is stored in XML format) from the internet, using XPath.  
# Remember that we load Scholar in the web interface and open a browser and use Control-U to see the XML code.  
# Inside R, we first load the XML library and the RCurl library:  
  
library(XML)  
library(RCurl)  
  
# Then we just download the webpage and we put the path to the desired web content into the notation of XPath.  
  
# We already gave some examples of how to scrape XML data from the web,

# back in Project 20 and Project 21.  Please feel welcome to read those again and remind yourself.  
  
# Here are a few more examples to inspire you, about how to scrape and parse some XML code from the internet.  
  
#####################################################  
# Example:  IMDB (Internet Movie Database)  
# We can scrape information about movies. For instance, IMDB is a popular movie website.  
# The information about the movie Say Anything is given here: <https://www.imdb.com/title/tt0098258/>

# This is Dr Ward's favorite movie, by the way!

# Here is the title and year, which are stored together in the same place.

xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/div[@class='title\_wrapper']/h1", xmlValue)

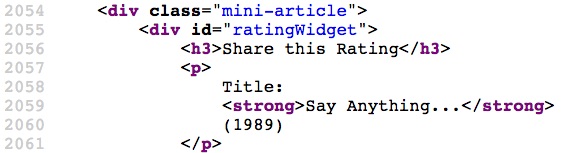


# In this XML, if you only want the year 1989 in which the movie was made,

# but do not care about the title, then just go deeper, by

# also including the "span" and "a" tags too:  
  
xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/div[@class='title\_wrapper']/h1/span/a", xmlValue)

# Here is a completely different place in the XML to find the title:  
  
xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/div/div[@id='ratingWidget']/p/strong", xmlValue)



# Here is the specific release date:  
  
xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/a[@title='See more release dates']", xmlValue)

https://piazza.com/redirect/s3?bucket=uploads&prefix=attach%2Fjl6zy9pr2rv17y%2Fil4gp785rby58b%2Fjtijco91atby%2Fsayanything4.jpg

# We can try to extract the Director and the Writer.

# Cameron Crowe was both the Director and the Writer.

# If we do the following search, we get 3 results:

xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/div[@class='credit\_summary\_item']", xmlValue)

# So we could save this information in a vector, and just extract the first and second elements

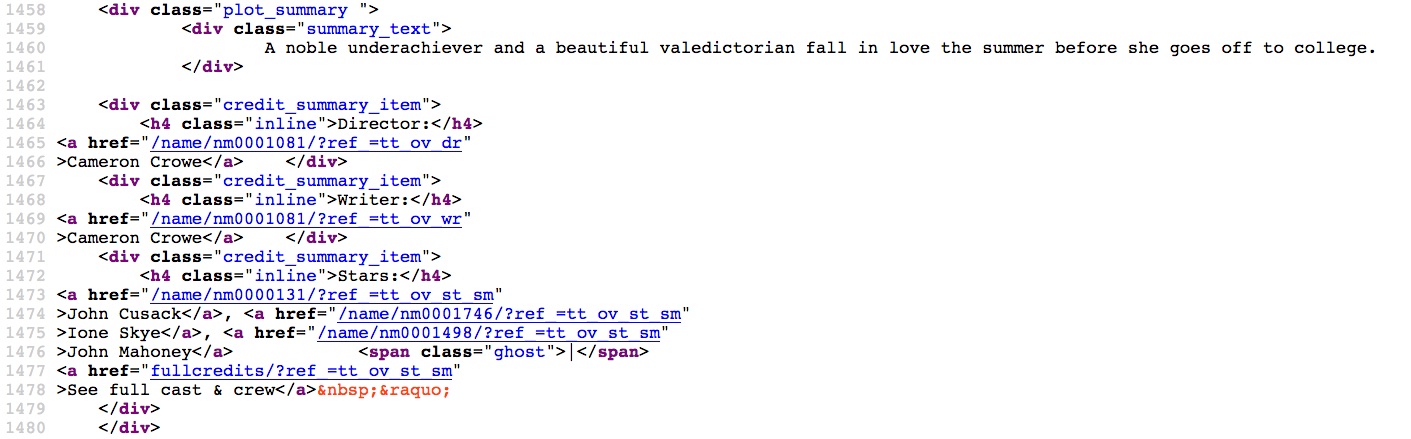
v <- xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/div[@class='credit\_summary\_item']", xmlValue)

# Now we have the DIrector:  
v[1]

# and the Writer:  
v[2]  
# in separate elements.

# There are other ways to do this, once we get more comfortable with XML

# but this is a good start!



# The title is stored lots and lots of places in the webpage.  
# It is also sometimes stored in an XML tag itself, rather than in the content of the page.

# For instance, search for the phrase:  
#      og:title  
# in the code in your browser to see this.  
  
xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/meta[@property='og:title']", xmlGetAttr, "content")

https://piazza.com/redirect/s3?bucket=uploads&prefix=attach%2Fjl6zy9pr2rv17y%2Fil4gp785rby58b%2Fjtijej2cgyc7%2Fsayanything5.jpg  
  
# Here's another way, which is just 2 lines later, in the source code for the page.

# We just change "property" to "name"

# and we change "og:title" to "title" and we get the title and year again:  
  
xpathSApply(htmlParse(getURL("https://www.imdb.com/title/tt0098258/")),  
            "//\*/meta[@name='title']", xmlGetAttr, "content")  
  
# These are just meant to be illustrative examples to try to help! Have fun!  Explore!

**Project 24**:

# In Project 23, we scraped a few elements of data from a website.

##################################################  
# Wrap your code from Project 23 into a function, and then  
# scrape at least 100,000 pieces of data from any website:  your choice!  
##################################################

# Here is an example of how to get started:  
# First we load the needed libraries:  
library(XML)  
library(RCurl)  
# Then we wrap our code into a function.  
mytitlefunc <- function(x) {  
   xpathSApply(htmlParse(getURL(paste0("https://www.imdb.com/title/tt", x, "/"))),  
            "//\*/div[@class='title\_wrapper']/h1", xmlValue)  
}  
  
# Notice that we replaced the website:  
#   https://www.imdb.com/title/tt0098258/  
# with (instead) some code to build the website as we go:  
#   paste0("https://www.imdb.com/title/tt", x, "/"  
# This uses the value x as the number of the movie.  
# Now we can run our function and extract the results for a movie:  
mytitlefunc("0098258")  
# Our function is vectorized, i.e., we can run it on a vector,  
# and it will return the results for each individual movie,  
# for instance:  
mytitlefunc(c("0110000", "0110001", "0110002", "0110003"))  
# We could try to run it on a sequence of numbers, but  
# this will not quite work at first.  
# For instance, if we try to run it on this sequence:  
110000:110003  
# We see that these numbers are only 6 digits, but the URL expects to  
# have a total of 7 digits to work.  
# So we can use the string print function,  
# which is also available in other languages too:  
sprintf("%07d", 110000:110003)  
# Here we have the "%" which means we are printing a variable,  
# and the "0" means we should pad things with leading zeroes if needed,  
# and the "7" means that we want 7 digits, and the "d" means digits.  
# Now it will work on this input  
mytitlefunc(sprintf("%07d", 110000:110003))  
# and we can even change this to (say) 100 pages at a time:  
mytitlefunc(sprintf("%07d", 110000:110100))

Hint from Luke Francisco:

Based on my experience with students during my office hours last week, I thought I would share some things to consider when you are looking for data to use on project 24 if you have not done so already.  I should have posted this earlier, but it just now dawned on me that this would make a good Piazza post.

When you are scraping large amounts of data from the web, you want to focus on replicability.  If you look at Dr. Ward's previous two examples with the national parks data and the Billboard music data, you will see what I am talking about.  There are several websites in each case (one webpage for each national park and one webpage for the Billboard top songs for each week).  If you want to scrape all of this data you will need to give R all of the URL's in order to go find the data.  What makes these two examples easy are that the webpages all have the exact same URL's except for one part.  For example, the Billboard top songs all have the same URL except for a different date inserted.  This allows you to make a vector of dates and insert each date into the URL.  If this were not the case and the URL's were totally different for each week, you would have to find the URL for every week since 1980, which would be extremely laborious!!!!!

You also want to make sure that each website is formatted similarly.  Consider the addresses of the national parks - they were all found at the bottom of the page for the corresponding national park with the same HTML formatting.  In the case of the Billboard data, the website for every week has the top songs entered in the exact same format - the only things that change are the song titles and artists, which are the data we are interested in.  This means your code that pulls the songs on the Billboard charts from this week will also pull the songs on the Blackboard charts from 1980!!!!!!

Consider this example I used in my office hours last week.  Ken Pomeroy publishes statistics for all 353 men's division 1 college basketball teams.  The data for the most recent season can be found at this link:

<https://kenpom.com/index.php?y=2019>

If you look at the HTML code of the website and CTRL+F search for Purdue, you will notice that the data for each team, which is the data in each row of the table on the website, is entered with the exact same format.  Better yet, change the last four digits of the URL to 2018 and CTRL+F for Purdue again in the HTML code.  The data for the 2018 season for every team is also entered in the exact same format.  Even better is that you can change the year at the end of the URL to any year after 2002 and you will find a wealth of similarly formatted data.  This meets the two criteria: 1.) URL's containing data have extremely similar formats and 2.) Each webpage has identical HTML formatting.  This is the type of data you should look for when finding data for your project as it will make your life a whole lot easier!

Also keep in mind that you want to get data that you can analyze in some way for project 25!

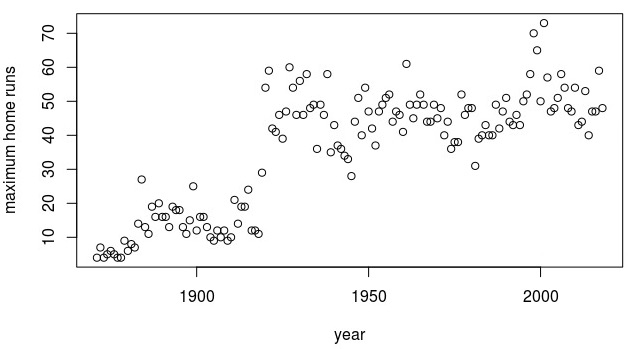
Sorry for being long, but I hope this was helpful and inspiring.  Remember to work smarter, not harder!

**Project 25:**

1a.  Store the 100,000 pieces of data that you scraped in Project 24 into a data frame.  
1b.  Save that data frame in an xlsx file, for instance, using the write.xlsx function from the library "xlsx".  
  
2a,2b,2c.  Make 3 questions about the data that you assembled in Project 24.  
  
3a,3b,3c.  Answer the 3 questions from 2a,2b,2c by making 3 visualizations from the data that you assembled.  Be sure to use best practices for data visualization.  
  
Refer to the selections from the texts:  
The Elements of Graphing Data by William S. Cleveland  
and Creating More Effective Graphs by Naomi B. Robbins  
  
These selections are archived online here:  
<http://llc.stat.purdue.edu/ElementsOfGraphingData.pdf>   
<http://llc.stat.purdue.edu/CreatingMoreEffectiveGraphs.pdf>   
  
Submit your project in RMarkdown.  Please be sure to submit the .Rmd file and also the .xlsx file created in 1b.  Of course the graders will be unable to run your code for 1a, because they do not want to scrape all of the data that you scraped.  Instead, the graders want to use the data from question 1b, so be sure to submit the .Rmd file and the .xlsx file too.

**Optional Project 1:**

Remind yourself how to run SQL queries in R, for instance, using the examples from Project 8.   
  
1.  Find the largest number of home runs (by an individual batter) each year.   
For instance:   
in 2014 a player hit 40 HR's,   
in 2015 a player hit 47 HR's,   
in 2016 a player hit 47 HR's,   
in 2017 a player hit 59 HR's, and   
in 2018 a player hit 48 HR's.   
(Yes, I have updated the data to include 2018!!)   
  
2.  Make a plot that shows this largest number of home runs per year (not just these 5 years, but the annual records back to 1871). The plot should look something like the plot in the picture (attached).

  
  
3.  Create a question about baseball that you are interested in, and use a SQL query in R to answer the question. Put all of your R code into an RMarkdown file, and give some comments about your code, to explain your method of solution. Submit the RMarkdown (.Rmd) file, and also a pdf file the shows the output (including the code, your explanation, the picture from question 2 that displays the plot, etc.).

**Optional Project 2:**

Recall how we can work with very large data sets (which are too large to import into R), by using UNIX.  We did this in some of the earliest problem sets in STAT 19000, during the fall semester.   
  
1a.  How many taxi cab rides occurred (altogether) during 2015?  Do not give a breakdown by month.  Give the total number of taxi cab rides for the full year 2015.  (Hint: Remember to be careful about the headers at the top of each file.)   
  
1b.  Give the distribution of the number of passengers in the taxi cab rides throughout (all months of) the year 2015.  Do not give a breakdown by month.  Give the distribution across the full year 2015.   
  
2a.  Across all years of the airline data, how many flights occurred on each airline?  Which airline is the most popular overall, in terms of the number of flights?   
  
2b.  Across all years of the airline data, which flight path is the most popular?  How many airplane trips occurred on that flight path?   
  
3.  Create a question about taxi cab rides or airline flights that you are interested in, and use UNIX to answer the question.   
  
Put all of your UNIX code into plain text file, and give some comments about your code, to explain your method of solution. Submit the plain text (.txt) file with your code (including your explanations).

**Optional Project 3:**

Use R to analyze the election data from the 2018 election.  Remember to use read.csv to read in the data, and use header=F (since there is no header) and use sep="|" since this symbol separates the data.   
  
1a.  Identify the top 20 employers that donated the most amount of money (altogether).  Some of these entries will be strange, e.g., blank entries, NA, self employed, etc.  That is OK!   
  
1b.  Plot the largest 20 total amounts (from the 20 employers) on a dotchart, in order from largest (at the top) to smallest (at the bottom).   
  
2a.  In which city/state is the average donation amount the largest?  (Treat the city and state data together as a pair.)   
  
2b.  How many donations were given from this city/state pair?  How large were the total amount of donations from this city/state pair?   
  
3.  Create a question about the 2018 election data that you are interested in, and use R to answer the question.   
  
Put all of your R code into an RMarkdown file, and give some comments about your code, to explain your method of solution. Submit the RMarkdown (.Rmd) file, and also a pdf file the shows the output (including the code, your explanation, the picture from question 1b that displays the plot, etc.).

**Optional Project 4:**

Please submit your project in RMarkdown.   
  
Read the selection of The Elements of Graphing Data by William Cleveland, and the selection of Creating More Effective Graphs by Naomi Robbins.   
  
Also read the classic article "How to Display Data Badly" by Howard Wainer: <http://www.jstor.org.ezproxy.lib.purdue.edu/stable/2683253>   
  
We referred to both of these in Project 25.   
  
1a. Find 3 visualizations from the Information Is Beautiful website (<http://www.informationisbeautiful.net/>) that do a BAD job of portraying data, according to the best practices in the selections mentioned above. Write 1/3 of a page (for each such visualization) about what is done poorly, i.e., write 1 single-spaced page total.   
  
1b. Identify 3 excellent visualizations of data from the Information Is Beautiful website. Write 1/3 of a page (for each such visualization) about what is done well, i.e., write 1 single-spaced page total.   
  
2. Consider the poster winner "Congestion in the Sky", from the 2009 Data Expo: <http://stat-computing.org/dataexpo/2009/posters/>   
  
2a.  Describe at least 3 significant ways that this poster could be improved. For each of these 3 ways, write a 1/3 of a page constructive criticism, specifying what could be improved and how that aspect of the visualization could be done better, i.e., write 1 single-spaced page total.   
  
2b. Which of the posters in the Data Expo 2009 do you think should be the winner? Why? (It is OK if you choose the poster that actually won, or any of the other posters.) Thoroughly justify your answer, using the techniques of effective data visualization, to justify your answer (write 1 single-spaced page total).   
  
(This entire assignment is 4 single-spaced pages.)