Data Structuring and Mining

Dr. Shahid Mahmood Awan
Assistant Professor
School of Systems and Technology, University of Management and Technology
shahid.awan@umt.edu.pk

Umer Saeed(MS Data Science, BSc Telecommunication Engineering)
Sr. RF Optimization & Planning Engineer
f2017313017@umt.edu.pk

Data Structuring and Mining

Raw and Processed Data

Definition of Data

- “Data are values of qualitative or quantitative variables, belonging to a set of items.”

- **Set of items:** Sometimes called the population; the set of objects you are interest in.
- **Variables:** A measurement or characteristics of an item.
- **Qualitative:** Country of origin, Sex, Treatment
- **Quantitative:** Height, Weight, Blood Pressure

Raw-Vs-Processed Data

- **Raw Data:**
- The original source of the data.
- Often hard to use for data analyses.
- Data analysis includes processing
- Raw data may only need to be processed once.

- **Processed Data:**
- Data that is ready for analysis.
- Processing can include merging, subsetting, transforming, etc.
- There may be standards for processing.
- All steps should be recorded.

Data Structuring and Mining

The Components of tidy data

What you should deliver to the statistician

- To facilitate the most efficient and timely analysis this is the information you should pass to a statistician:
- 1- The raw data.
- 2- A tidy data set.
- 3- A code book describing each variable and its values in the tidy data set.
- 4- An explicit and exact recipe you used to go from 1 -> 2,3

Raw Data

- It is critical that you include the rawest form of the data that you have access to. This ensures that data provenance can be maintained throughout the workflow. Here are some examples of the raw form of data:
- 1- The strange binary file your measurement machine spits out
- 2- The unformatted Excel file with 10 worksheets the company you contracted with sent you
- 3- The complicated JSON data you got from scraping the Twitter API
- 4- The hand-entered numbers you collected looking through a microscope

1- The raw data

- You know the raw data are in the right format if you:
- 1- Ran no software on the data
- 2- Did not modify any of the data values
- 3- You did not remove any data from the data set
- 4- You did not summarize the data in any way

- If you made any modifications of the raw data it is not the raw form of the data.
- Reporting modified data as raw data is a very common way to slow down the analysis process, since the analyst will often have to do a forensic study of your data to figure out why the raw data looks weird.

Tidy data.

- The principles are more generally applicable:
- 1- Each variable you measure should be in one column
- 2- Each different observation of that variable should be in a different row
- 3- There should be one table for each "kind" of variable
- 4- If you have multiple tables, they should include a column in the table that allows them to be joined or merged
- While these are the hard and fast rule; Some other important tips;
- 1- Include a row at the top of each file with variable names
- 2- Make variable names human readable AgeAtDiagnosis instead of AgeDx.
- 3- In general data should be saved in one file per table

Code Book

- For almost any data set, the measurements you calculate will need to be described in more detail than you can or should sneak into the spreadsheet. The code book contains this information. At minimum it should contain:
- 1- Information about the variables (including units!) in the data set not contained in the tidy data
- 2- Information about the summary choices you made
- 3- Information about the experimental study design you used
- Some other important tips are;
- 1- A common format for this document is a Word file/text file.
- 2- There should be a section called "Study design" that has a thorough description of how you collected the data.
- 3- There is a section called "Code book" that describes each variable and its units.

The Instruction list

- You may have heard this before, but reproducibility is a big deal in computational science. That means, when you submit your paper, the reviewers and the rest of the world should be able to exactly replicate the analyses from raw data all the way to final results. If you are trying to be efficient, you will likely perform some summarization/data analysis steps before the data can be considered tidy.
- The ideal thing for you to do when performing summarization is to create a computer script
- The input for the script is the raw data
- The output is the processed, tidy data
- There are not parameters to the script
- In some cases it will not be possible to script every step. In that case you should provide instruction like;
- 1- Step-1: take the raw file, run version 3.1.2 of summarize software with parameter a=1,b=2, c=3
- 2- Step-2: run he software separately for each sample
- 3- Step-3: take column tree of output file. Out of each sample and that is the corresponding row in the output data set.

Data Structuring and Mining

Downloading Files

Get/Set Your Working Directory

- The basic component of working with data is knowing your working directory.
- The tow main commands are getwd() and swtwd()
- Be aware of relative-vs-absolute paths;
- **Relative-** setwd(“./data”), setwd(“../”)
- **Absolute-** setwd("D:/MS DS/Others/3_Getting and Cleaning Data")
- Important difference in Windows setwd (D:\MS DS\Others\1_Data Scientist's Toolbox)

Checking For and Creating Directories

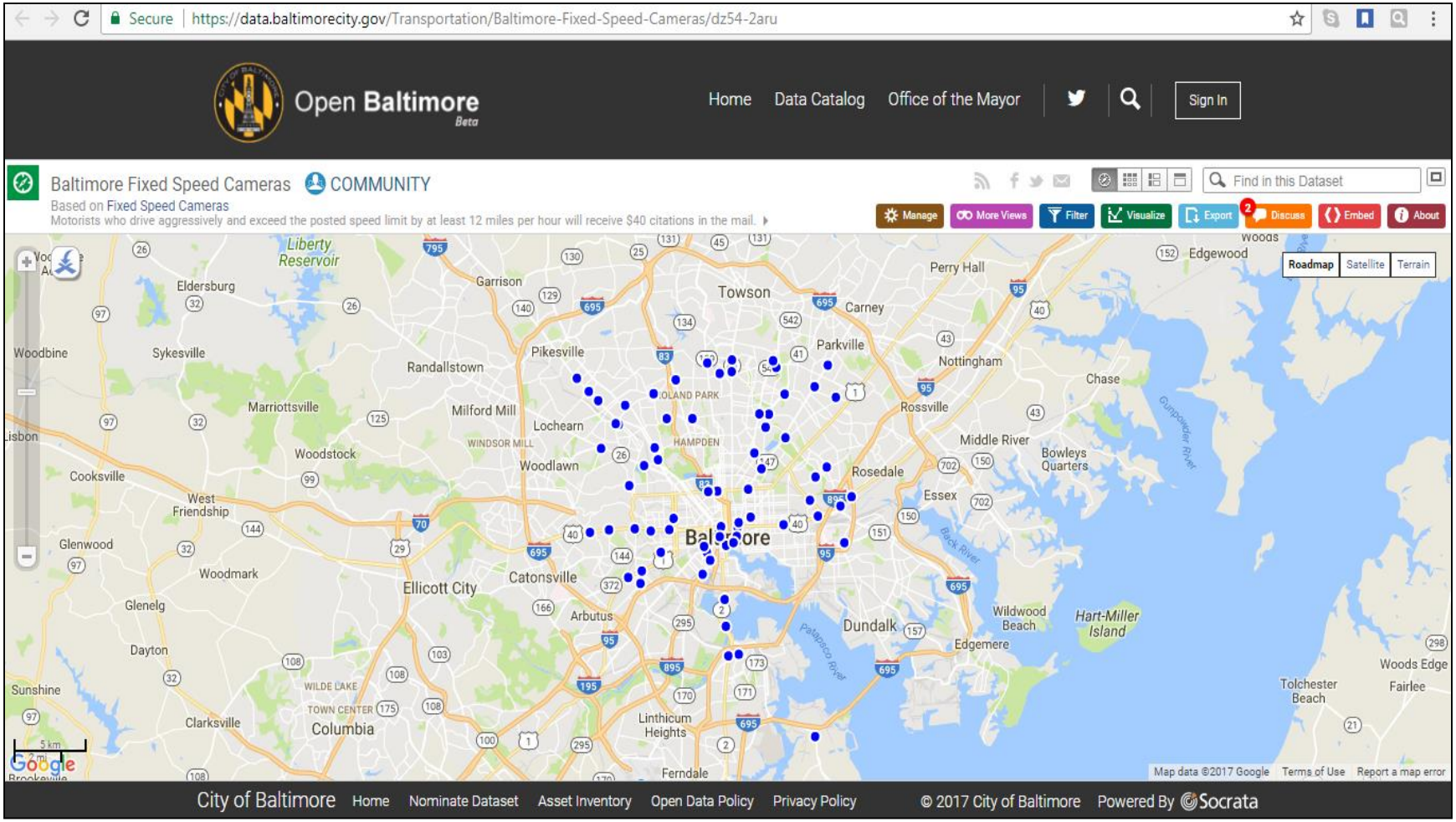
- *file.exists ("directoryName")* will check to see if the directory exists.
- *dir.create("directoryName")* will create a directory if it doesn't exists.
- Here is an example checking for "data" directory and creating it if it doesn't exist.

```
if(!file.exists("data")){
      dir.create("data")
}
```

Getting data from the internet-download.file()

- Downloads a file from the internet.
- Even if you could do this by hand, help with reproducibility
- Import parameters are url, destfile, method
- Useful for downloading tab-delimited, csv and other files

Example-Baltimore Camera Data

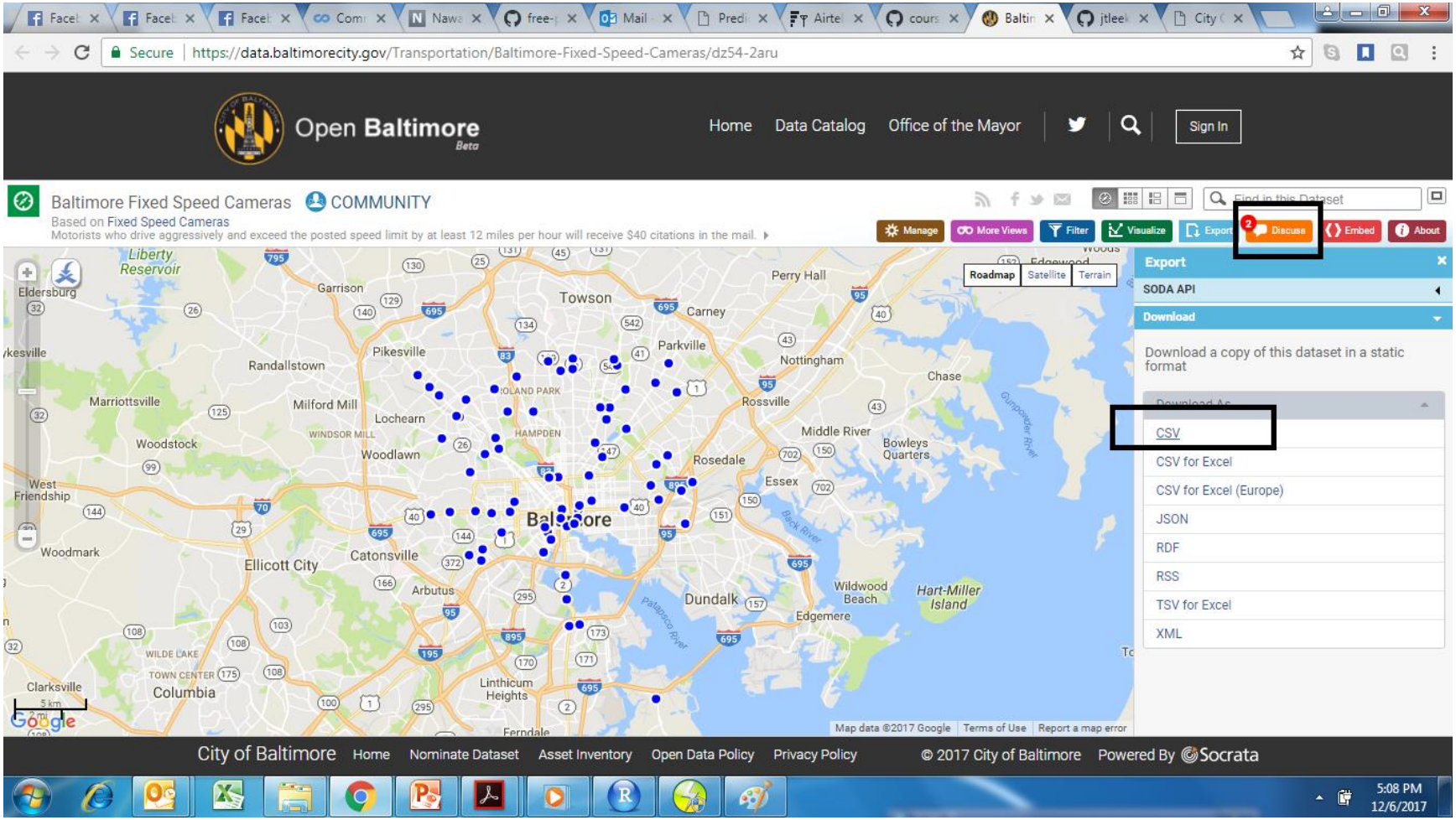


https://data.baltimorecity.gov/Transportation/Baltimore-Fixed-Speed-Cameras/dz54-2aru

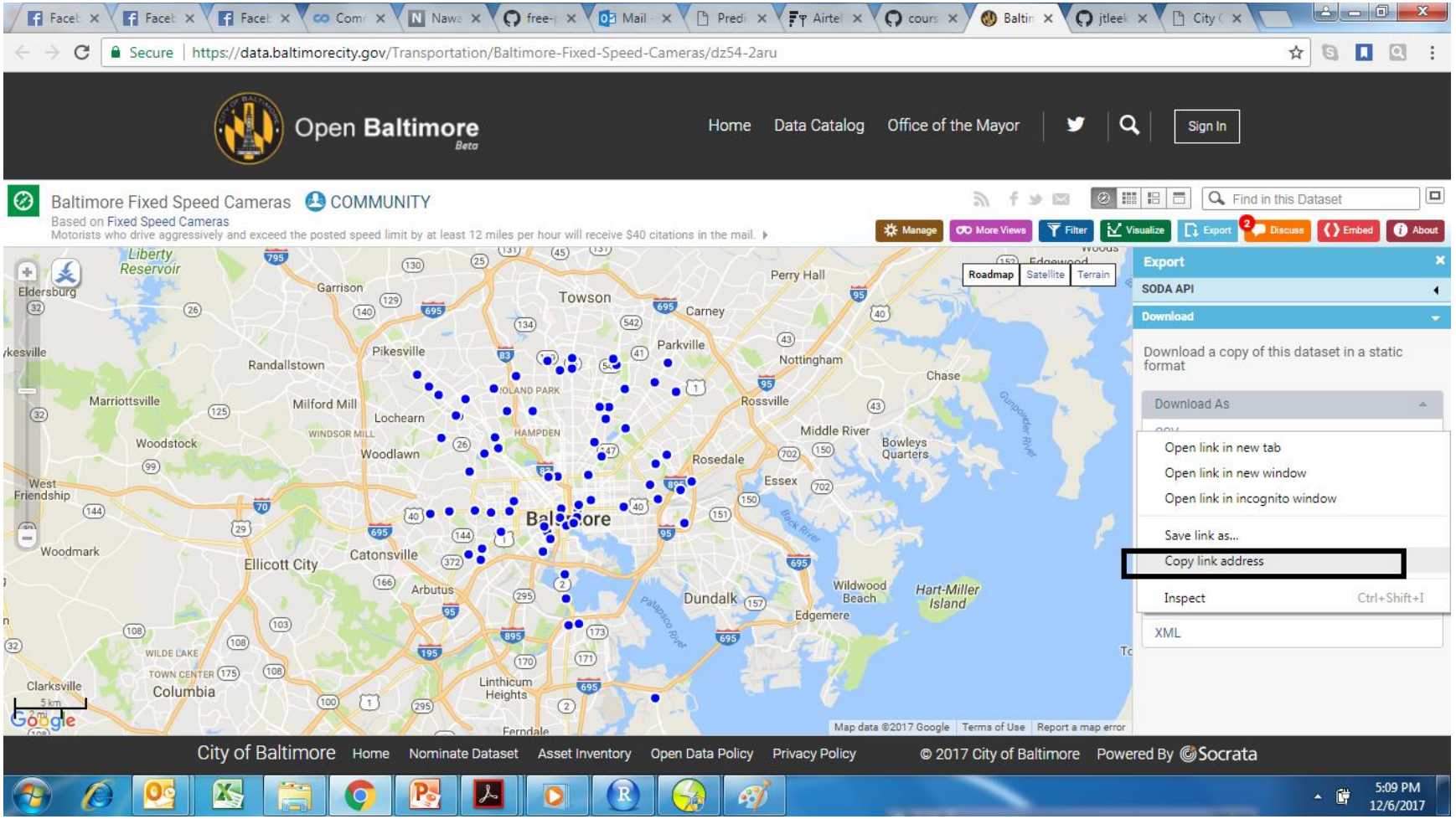
Dr. Shahid Awan

Umer Saeed

Example-Baltimore Camera Data



Example-Baltimore Camera Data



Dr. Shahid Awan

Umer Saeed

Example-Baltimore Camera Data

- >urlfile<”http://data.baltimorecity.gov/api/views/dz542aru/rows.csv?accessType=DOWNLOAD”
- > download.file(urlfile,destfile = "umer.csv“)
- > list.files()
- > dateDownloaded<-date()
- > dateDownloaded

Some notes about download.file()

- If the url starts with http you can use download.file()
- If the url starts with https on Windows you may be ok.
- If the url starts with https on Mac you may need to set method="curl"
- If the file is big, this might take a while.
- Be sure to record when you downloaded.

Data Structuring and Mining

Reading Local Flat Files

Loading Flat Files-read.table()

- This is the main function for reading data into R.
- Flexible and robust but requires more parameters.
- Reads the data into RAM-bigh data can cause problems.
- Important parameters file, header, sep, row.names, nrows
- **Related:** read.csv(),read.csv2(), read.delim(), read.delim2()

Loading Flat Files-read.table()

- **read.table (for csv files)**
- Example-1
- > data1<-read.table("WHO.csv",sep = ",",header = TRUE, quote = "\"")
- Example-2
- > data2<-read.table("mvtWeek1.csv",sep = ",",header = TRUE, quote = "\"")
- **read.table (for txt files)**
- Example-3
- > data3<-read.table("WHO.txt",sep = "\t",quote = "\"",header = TRUE)
- Example-4
- > data4<-read.table("mvtWeek1.txt",sep = "\t",quote = "\"",header = TRUE)
- Some More Imp

Loading Flat Files-read.table()

- **Some more important parameters**
- **quote:** You can tell R whether there are any quouted values quote=“” means no quotes.
- **na.string:** set the character that represents a missing value.
- **nrows:** how may rows to read of the file (e.g nrows-10 reads 10 lines)
- **skip:** number of lines to skip before starting to read.
- The biggest trouble with reading flat files are quotation marks ‘ or “ in data values, setting quote=“” often resolves these.

Loading Flat Files-read.csv()

- **read.csv (for csv file)**
- Exampple-1
- > data5<-read.csv("mvtWeek1.csv")
- Example-2
- > data6<-read.csv("WHO.csv")
- **read.csv (for txt file)**
- Example-3
- > data7_text<-read.csv("WHO.txt",sep = "\t")
- Example-4
- > data8<-read.csv("mvtWeek1.txt",sep = "\t")

Loading Flat Files-read.csv2()

- **read.csv2(for csv files)**
- Example-1
- > data9<-read.csv2("mvtWeek1.csv",sep = ",")
- Example-2
- > data10<-read.csv2("WHO.csv",sep = ",")
- **read.csv2(for txt files)**
- Example-3
- > data11<-read.csv2("WHO.txt",sep = "\t")
- Example-4
- > data12<-read.csv2("mvtWeek1.txt",sep = "\t")

Loading Flat Files-read.delim ()

- **read.delim (for csv files)**
- Example-1
- > data13<-read.delim("mvtWeek1.csv",sep = ",")
- Example-2
- > data14<-read.delim("WHO.csv",sep = ",")
- **read.delim (for txt files)**
- Example-3
- > data15<-read.delim("mvtWeek1.txt",sep = "\t")
- Example-4
- > data16<-read.delim("WHO.txt",sep = "\t")

Loading Flat Files-read.delim2 ()

- **read.delim2 (for csv files)**
- Example-1
- > data17<-read.delim2("mvtWeek1.csv",sep = ",")
- Example-2
- > data18<-read.delim2("WHO.csv",sep = ",")
- **read.delim2 (for txt files)**
- Example-3
- > data19<-read.delim2("mvtWeek1.txt",sep = "\t")
- Example-4
- > data20<-read.delim2("WHO.txt",sep = "\t")

Using the readr Package

- The readr package is devloped by Hadly Wickham to deal with reading in large flat files quickly.
- The package provides replacement for functions like read.table() and read.csv. (utils; verbose, slower)
- The analogous functions in readr are read_table() ,read_delim(), read_tsv and read_csv(). This functions are oven much faster than their base R analogues and provide a few other nice features such as progress meters.
- **Step-1:** >install.packages("reader")
- **Step-2:** >library(readr)
- **Step-3:** >getwd()
- **Step-4:** >setwd()
- **Step-5:** >dir()
- **Step-6:** Import required data using read_table(),read_delim(),read_tsv() and read_csv().

Loading Flat Files-read_table ()

- **read_table()**
- > data21<-read_table("massey-rating.txt")

- **read_table2()**
- > data21<-read_table2("massey-rating.txt")

Loading Flat Files-read_delim ()

- **read_delim()**
- data22<-read_delim("mvtWeek1.csv",delim = ",")

Loading Flat Files-read_csv ()

- **read_csv()**
- > ali<-read_csv("mvtWeek1.csv")
- **read_csv2()**
- > data24<-read_csv2("forcsv2.txt“)

Loading Flat Files-read_tsv ()

- **read_tsv()**
- > data25<-read_tsv("mvtWeek1.txt")

Data Structuring and Mining

Reading Excel Files

Excel Files

- Still probably the most widely used format for data.

Loading Excel Files-read.xlsx()

- **Install /load xlsx Package**
- > install.packages("xlsx")
- > library(xlsx)
- **read.xlsx()**
- > data26<-read.xlsx("u1111.xlsx",sheetIndex = 1,header = TRUE)
- **read.xlsx2()**
- > data27<-read.xlsx2("u1111.xlsx",sheetIndex = 1,header = TRUE)
- read.xlsx2 is much faster than read.xlsx but for reading subsets of rows may be slightly unstable.
- > data27<-read.xlsx("u1111.xlsx",sheetIndex = 1,header = TRUE,,colIndex = 2:3,rowIndex = 1:4)
- > data28<-read.xlsx2("u1111.xlsx",sheetIndex = 1,header = TRUE,,colIndex = 2:3,rowIndex = 1:4)
- In general it is advised to store your data in either a database or in comma separated file(.csv) or tab separated files(.tab/.txt) as they are easier to distribute.

Data Structuring and Mining

Reading XML Files

XML

- XML stands for “Extensible markup language”
- Frequently used to sore structured data.
- Particularly widely used in internet applications
- Extracting XML is the basis for most web scraping.
- Components
- 1- Markup- Labels that give the text structure
- 2- Content- The actual text of the document

Tags, elements and attributes

- Tags correspond to general labels;
- Start tags<section>
- End tags<section>
- Empty tags<section>
- Elements are specific examples of tags;
- <Greeting> Hello, world</Greeting>
- Attributes are components of the label;
- <img src="jeff.jpg" alt="instructor"/>
- <step number="3"> Connect A to B.</step>

Example XML file

https://www.w3schools.com/xml/simple.xml

Reading the XML File into R

- **Install /load XML Package**
- > install.packages("XML")
- > library(XML)
- **Download XML File**
- > url<-" http://www.w3schools.com/xml/simple.xml"
- > download.file(url,destfile = "school.xml")
- **Read XML File into R**
- > doc<-xmlTreeParse("school.xml",useInternalNodes=TRUE)
- > rootNode<-xmlRoot(doc)
- > xmlName(rootNode)
- > names(rootNode)
- **XML to Data Frame**
- > umer<-xmlToDataFrame("school.xml")

Data Structuring and Mining

Reading JSON Files

JSON

- JSON stands for “Java Script Object Notation”
- Light weight data
- Common format for data from application programming interfaces (APIs)
- Similar structure to XML but different syntax/format
- Data stored as;
- Number (double)
- Strings (double quoted)
- Boolean (true or false)
- Array (ordered, comma separated enclosed in square brackets[]).
- Object (un-order, comma separated collection of key: values pairs in curly brackets{})

Example JSON File

```
[
  {
    "id": 101394164,
    "name": "advdatasci",
    "full_name": "jtleek/advdatasci",
    "owner": {
      "login": "jtleek",
      "id": 1571674,
      "avatar_url": "https://avatars2.githubusercontent.com/u/1571674?v=4",
      "gravatar_id": "",
      "url": "https://api.github.com/users/jtleek",
      "html_url": "https://github.com/jtleek",
      "followers_url": "https://api.github.com/users/jtleek/followers",
      "following_url": "https://api.github.com/users/jtleek/following{/other_user}",
      "gists_url": "https://api.github.com/users/jtleek/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/jtleek/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/jtleek/subscriptions",
      "organizations_url": "https://api.github.com/users/jtleek/orgs",
      "repos_url": "https://api.github.com/users/jtleek/repos",
      "events_url": "https://api.github.com/users/jtleek/events{/privacy}",
      "received_events_url": "https://api.github.com/users/jtleek/received_events",
      "type": "User",
      "site_admin": false
    },
```

Reading data from JSON

- **Install /load jsonlite Package**
- > install.packages("jsonlite")
- > library(jsonlite)
- **Download XML File**
- > download.file(url,destfile = "tep.json")
- **Read JSON File into R**
- > doc<-fromJSON("tep.json")
- **JSON to Data Frame**
- > ali<-as.data.frame(doc)

Data Structuring and Mining

Using data.table

data.table

- Inherits from data.frame (All functions that accept data.frame work on data.table)
- Written in C so it is much faster
- Much, much faster at sub setting, group and updating.
- Goal1: Reduce Programming time (fewer function calls, less variable name repetition)
- Gole2: Reduce compute time (fast aggregation, update by reference)
- Currently in-memory: 64bit and 100GB in routine
- Ordered joins: useful in finance (time series) but also in other fields e.g. genomics

General Form

```
mtcarsDT[
   mpg > 20,
   .(AvgHP = mean(hp),
     "MinWT(kg)" = min(wt*453.6)),   # wt lbs
   by = .(cyl, under5gears = gear < 5)
]
```

R:	i	j	by
SQL:	WHERE	SELECT	GROUP BY

Create data tables just like data frames

- > install.packages("data.table")
- > library(data.table")
- **data.frame**
- > ali <- data.frame(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
- > ali
- **data.table**
- > umer<-data.table(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
- > umer
- **Identical**
- > identical(ali,umer)
- > identical(dim(ali),dim(umer))
- > identical(ali$x,umer$x)
- > identical(ali$a,umer$a)

Create data tables just like data frames

- > is.list(umer)
- > is.list(ali)
- > is.data.frame(umer)
- > is.data.frame(ali)
- > is.data.table(umer)
- > is.data.table(ali)
- > is.vector(umer)
- > is.vector(ali)

See all the data tables in memory

- > set.seed(9)
- > umer<-data.table(m=rnorm(9),y=rep(c("a","b","c"),each=3),z=rnorm(9))
- > set.seed(9)
- > ali<-data.table(m=rnorm(9),y=rep(c("a","b","c"),each=3),z=rnorm(9))
- > set.seed(9)
- > ahmed<-data.table(m=rnorm(9),y=rep(c("a","b","c"),each=3),z=rnorm(9))
- > tables()

Row Subset operation

- > # basic row subset operations
- > umer[2] ; > umer[2,] # 2nd row
- > umer[3:2]; > umer[3:2,] # 3rd and 2nd row
- > umer[order(x)]; > umer[order(x),] # no need for order(umer$x)
- > umer[y>2]; > umer[y>2,] # all rows where DT$y > 2
- > umer[y>2 & v>5]; > umer[y>2 & v>5,] # compound logical expressions
- > umer[!2:4]; > umer[!2:4,] # all rows other than 2:4
- > umer[-(2:4)] > umer[-(2:4),] # all rows other than 2:4 (same)
- > # subset rows and select and compute data.table way
- > umer[2:3, sum(v)] # sum(v) over rows 2 and 3, return vector
- > umer[2:3, .(sum(v))] # same, but return data.table with column V1
- > umer[2:3, list(sum(v))] # same, but return data.table with column V1
- > umer[2:3, .(sv=sum(v))] # same, but return data.table with column sv

Row Subset operation

```
# fast ad hoc row subsets (subsets as joins)
> umer["a",on="x"]                  # same as x == "a" but uses binary search (fast)
> umer["a", on=.(x)]                # same, for convenience, no need to quote every column
> umer[.("a"), on="x"]              # same
> umer[x=="a"]                      # single "==" internally optimized to use binary search (fast)
> umer[x!="b" | y!=3]               # not yet optimized, currently vector scan subset
> umer[x!="b" & y!=3]               # not yet optimized, currently vector scan subset
> umer[.("b", 3), on=c("x", "y")] # join on columns x,y of DT; uses binary search (fast)
> umer[.("b", 4), on=c("x", "y")] # join on columns x,y of DT; uses binary search (fast)
> umer[.("b", 3), on=.(x, y)      # same, but using on=.()
> umer[.("b", 1:2), on=c("x", "y")] # no match returns NA
> umer[.("b", 1:2), on=.(x, y), nomatch=0]                    # no match returns NA
> umer[.("b", 5), on=c("x", "y"), roll=Inf]  #locf, nomatch row gets rolled by previous row
> umer[.("b", 5), on=c("x", "y"), roll=-Inf] # nocb, nomatch row gets rolled by next row
> umer["b", sum(v*y), on="x"]               # on rows where DT$x=="b", calculate sum(v*y)
```

Columns Subset operation

- # select and compute columns data.table way;
- > umer[, v] # v column (as vector)
- > umer[, list(v)] # v column (as data.table)
- > umer[, .(v)] # same as above, .() is a shorthand alias to list()
- > umer[, sum(v)] # sum of column v, returned as vector
- > umer[, list(sum(v))] # sum of column v, returned as list
- > umer[, .(sum(v))] # same as above
- > umer[, .(sv=sum(v))] # same, but column named "sv“
- > umer[, .(v, v*2)] # return two column data.table, v and v*2
- **Grouping Operations**
- > umer[, sum(v), by=x] # ad hoc by, order of groups preserved in result
- > umer[, sum(v), keyby=x] # same, but order the result on by cols
- > umer[, sum(v), by=x][order(x)] # same but by chaining expressions together

Subset operation (Columns /Rows)

- # all together now
- > umer[x!="a", sum(v), by=x] # get sum(v) by "x" for each i != "a“
- > umer[!"a", sum(v), by=.EACHI, on="x"] # same, but using subsets-as-joins
- > umer[c("b","c"), sum(v), by=.EACHI, on="x"] # same
- > umer[c("b","c"), sum(v), by=.EACHI, on=.(x)] # same, using on=.()

Joins as subsets

- > umer<-data.table(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
- > ali <- data.table(x=c("c","b"), v=8:7, foo=c(4,2))
- > umer[ali, on="x"] # right join
- > ali[umer, on="x"] # left join
- > umer[ali, on="x", nomatch=0] # Inner join
- > umer[!ali, on="x"] # not join
- > umer[ali, on=.(y<=foo)] # new non-equi join (v1.9.8+)
- > umer[ali, on="y<=foo"] # same as above
- > umer[ali, on=c("y<=foo")] # same as above
- > umer[ali, on=.(x, y<=foo)] # new non-equi join (v1.9.8+)
- > umer[ali, .(x,y,x.y,v), on=.(x, y>=foo)] # Select x's join columns as well
- > umer[ali, on="x", mult="first"] # first row of each group
- > umer[ali, on="x", mult="last"] # last row of each group
- > umer[ali, sum(v), by=.EACHI, on="x"] # join and eval j for each row in I
- > umer[ali, sum(v)*foo, by=.EACHI, on="x"] # join inherited scope
- > umer[ali, sum(v)*i.v, by=.EACHI, on="x"] # 'i,v' refers to X's v column
- > umer[ali, on=.(x, v>=v), sum(y)*foo, by=.EACHI] # newnon-equi join with by=.EACHI

Key

```
# setting keys
> umer<-data.table(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
> umer1<-copy(umer)                    # copy umer to umer1 to work with it.
> setkey(umer1,x)                      # set a 1-column key. No quotes, for convenience.
> setkeyv(umer1,"x")                   # same (v in setkeyv stands for vector)
> umer['a']                            # Subset using key
> key(umer)
> setkey(umer,NULL)
> umer["a"]                            #subset-as-join on *key* column 'x‘
> umer["a",on="x"]                     # same, being explicit using 'on=' (preferred)
> umer[!"a",sum(v),by=.EACHI]          # get sum(v) for each i != "a“
```

Key

```
# multi-column key
> setkey(umer,x,y)                    # multi-column key
> setkeyv(umer,c("x","y"))            # same, multi-column key
# subsets on multi-column key
> umer["a"]                           # join to 1st column of key
> umer["a",on="x"]                    # on= is optional, but is preferred
> umer[.("a")]                        # same, .() is an alias for list()
> umer[list("a")]                     # same
> umer[.("a", 3)]                     # join to 2 columns
> umer[.("a", 3:6)]                   # join 4 rows (2 missing)
> umer[.("a", 3:6), nomatch=0]        # remove missing
> umer[.("a", 3:6), roll=TRUE]        # locf rolling join
> umer[.("a", 3:6), roll=Inf]         # same as above
> umer[.("a", 3:6), roll=-Inf]        # nocb rolling join
> umer[!("a")]                        # not join
> umer[!"a"]                          #same as above
```

Key

- #merge
- > g1<-data.table(x=c('a','a','b'),y=1:3)
- > g2<-data.table(x=c('a','b'),y=4:5)
- > setkey(g1,x)
- > setkey(g2,x)
- > merge(g1,g2)

Special Symbols (.N,.SD,.I,.GRP)

- **.**N An integer, length 1, containing the number.
- # more on special symbols, see also ?"special-symbols"
- > umer<-data.table(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
- > umer[.N] #Last row
- > umer[,.N] # total number of rows in DT
- > umer[,.N,by=x] # number of rows in each group
- > umer[, .SD, .SDcols=x:y] # select columns 'x' and 'y‘
- > umer[, .SD, .SDcols=c('x','v')] # select columns 'x' and ‘v’
- > umer[, .SD[1]] # first row of all columns
- > umer[, .SD[1],by=x] # first row of 'y' and 'v' for each group in 'x‘
- > umer[,tail(.SD,2),by=x] # last 2 rows of each group
- > umer[, c(.N, lapply(.SD, sum)), by=x] # get rows *and* sum columns 'v' and 'y' by group
- > umer[, .I[1], by=x] # row number in DT corresponding to each group
- > umer[, grp := .GRP, by=x] # add a group counter column

Add/Update/Delete

```
> umer<-data.table(x=rep(c("b","a","c"),each=3), y=c(1,3,6), v=1:9)
# add/update/delete by reference (see ?assign)
> umer[,z:=42L]                    # add new column by reference
> umer[,g:=y^2]                    # add new column by reference
> umer[,t:={tem=z-1;log(tem)}]   # add new column by reference (Multiple Operations)
> umer[,cc:=y>=3]                  # add new column by reference (Other Operations)
> umer[,bb:=mean(y+v),by=x] # add new column by reference (group)
> umer[,z:=NULL]                    # remove column by reference
> umer["a",v:=42L,on="x"]      # sub-assign to existing v column by reference
> umer["b",v2:=84L,on="x"]     # sub-assign to new column by reference (NA padded)
> umer[,m:=mean(v),by=x][ ] # add new column by reference by group
                                   # NB: postfix [] is shortcut to print()
> umer[2, d := 10L][]              #shorthand for update and print
> umer[, d := 10L][]
> umer[y > 4, b := d * 2L][]# sub-assign to b with d*2L on those rows where b > 4 is TRUE
```

Add/Update/Delete

- > umer[y > 4][,b := d * 2L][] # different from above. [, :=] is performed on the subset
- # which is an new (ephemeral) data.table. Result needs to be
  # assigned to a variable (using `<-`).

Advanced Usage

- > umer[, sum(v), by=.(y%%2)] # expressions in by
- > umer[, sum(v), by=.(ali=y%%2)] # same, as above
- > umer[, lapply(.SD, sum), by=x] # sum of all (other) columns for each group
- > umer[, .SD[which.min(v)], by=x] # nested query by group
- > umer[,list(MySum=sum(v),MyMin=min(v),MyMax=max(v)),by=.(x,y%%2)]
- > umer[, .(y = .(y), v = .(v)), by=x] # list columns
- > umer[, sum(v), by=x][V1<20] # compound query
- > umer[, sum(v), by=x][order(V1)] # ordering results
- > umer[, sum(v), by=x][order(-V1)] # ordering results
- > umer[, c(.N, lapply(.SD,sum)), by=x] # get number of observations and sum per group

Convenience functions for range subsets

- # between
- > X <- data.table(a=1:5, b=6:10, c=c(5:1))
- > X[b %between% c(7,9)]
- > X[between(b, 7, 9)] #same, as above
- # NEW feature in v1.9.8, vectorised between
- > X[c %between% list(a,b)]
- > X[between(c, a, b)] #same, as above
- > X[between(c, a, b, incbounds=FALSE)] # Close Interval

Convenience functions for range subsets

- # inrange
- > set.seed(5)
- > Y <- data.table(a=c(8,3,10,7,-10), val=runif(5))
- > range <- data.table(start = 1:5, end = 6:10)
- > Y[a %inrange% range]
- > Y[inrange(a, range$start, range$end)] # same as above
- > Y[inrange(a, range$start, range$end, incbounds=FALSE)] # Open Interval

Determine Duplicate Rows

- # Duplicated & unique
- > DT <- data.table(A = rep(1:3, each=4), B = rep(1:4, each=3),C = rep(1:2, 6), key = "A,B")
- > duplicated(DT)
- > unique(DT)
- > duplicated(DT, by="B")
- > unique(DT, by="B")
- > duplicated(DT, by=c("A", "C"))
- > unique(DT, by=c("A", "C"))

- > DT = data.table(a=c(3.142, 4.2, 4.2, 3.142, 1.223, 1.223), b=rep(1,6))
- > unique(DT)

First/Last item of an object

- **Description:** Returns the first item of a vector or list, or the first row of a data.frame or data.table.
- > first(1:5)
- > first(x <- data.table(x=1:5, y=6:10)) # same as x[1]
- **Description:** Returns the last item of a vector or list, or the last row of a data.frame or data.table.
- > last(1:5)
- > last(x <- data.table(x=1:5, y=6:10))

Convenience function for calling regexpr

- **Description:** Intended for use in i in [.data.table.
- **Value:** Logical vector, TRUE for items that match pattern.
- **Example**
- > DT <- data.table(Name=c("UmerSaeed","AliSaeed","AhmedSaeed","Hajra"), Salary=c(2,3,4,8))
- > DT[Name %like% "Sa"]

Remove rows with missing values on columns specified

- > DT <- data.table(x=c(1,NaN,NA,3), y=c(NA_integer_, 1:3), z=c("a", NA_character_, "b", "c"))
- > na.omit(DT) # default behaviour
- > na.omit(DT, cols="x") # omit rows where 'x' has a missing value
- > na.omit(DT, cols=c("x", "y")) # omit rows where either 'x' or 'y' have missing values

Set operations for data tables

- **Description:** Similar to base's set functions, union, intersect, setdiff and setequal but for data.tables. Additional all argument controls if/how duplicate rows are returned.
- **Examples:**
- > x <- data.table(c(1,2,2,2,3,4,4))
- > y <- data.table(c(2,3,4,4,4,5))
- **Intersect**
- > fintersect(x, y) > fintersect(x, y,all=TRUE) > fintersect(x, y,all=FALSE)
- **Setdiff**
- > fsetdiff(x, y) > fsetdiff(y,x) > fsetdiff(x, y,all=TRUE)
- > fsetdiff(x, y,all=FALSE)
- **Union**
- > funion(x, y) > funion(x, y,all=TRUE) > funion(x, y,all=FALSE)
- **Set Equal**
- > fsetequal(x, y)

Fast and friendly file finagler

- **Description:** Similar to read.table but faster and more convenient. All controls such as sep, colClasses and nrows are automatically detected. bit64::integer64 types are also detected and read directly without needing to read as character before converting.
- Dates are read as character currently. They can be converted afterwards using the excellent fast time package or standard base functions.
- ‘fread‘ is for regular delimited files; i.e., where every row has the same number of columns. In future, secondary separator (sep2) may be specified within each column. Such columns will be read as type list where each cell is itself a vector.
- **Example**
- Step-1: > getwd()
- Step-2: > setwd("D:/MS DS/Others/3_Getting and Cleaning Data/Data Sets")
- Step-3: > dir()
- Step-4: > who<-fread("WHO.csv")

Reference

- https://www.coursera.org/