MET CS688 C1

WEB ANALYTICS AND MINING

ZLATKO VASILKOSKI

R SUPPLEMENT

Getting Started with R

Reference:

"R in Action" Second Edition

By

ROBERT I. KABACOFF

History of R

- R is a dialect of S (Fortran statistical library developed in 1976 by Bell Labs)
- Created in 1991 in New Zealand
- In 1998 rewritten in C (not much changed since then)
- Runs on any system (including Play Station)
- Syntax very similar to S
- It is modular and compact so you can add components (packages) as you need
- Interactive, powerful to create new features

Positive Features

- Free Software
- Freedom to redistribute
- Free to improve and release the improvements

Negative Features

- Based on 40 year technology
- Little support for dynamic 3D graphics
- R objects stored in memory

Why use R?

R has many features to recommend it:

■ Most commercial statistical software platforms cost thousands, if not tens of thousands, of dollars. R is free! If you're a teacher or a student, the benefits are obvious.

■ R is a comprehensive statistical platform, offering all manner of data-analytic techniques. Just about any type of data analysis can be done in R.

■ R contains advanced statistical routines not yet available in other packages. In fact, new methods become available for download on a weekly basis.

■ R has state-of-the-art graphics capabilities. If you want to visualize complex data, R has the most comprehensive and powerful feature set available.

■ R is a powerful platform for interactive data analysis and exploration. For example, the results of any analytic step can easily be saved, manipulated, and used as input for additional analyses.

Why use R?

■ Getting data into a usable form from multiple sources can be a challenging proposition. R can easily import data from a wide variety of sources, including text files, database-management systems, statistical packages, and specialized data stores. It can write data out to these systems as well. R can also access data directly from web pages, social media sites, and a wide range of online data services.

■ R provides an unparalleled platform for programming new statistical methods in an easy, straightforward manner. It's easily extensible and provides a natural language for quickly programming recently published methods.

■ R functionality can be integrated into applications written in other languages, including C++, Java, Python, PHP, Pentaho, SAS, and SPSS. This allows you to continue working in a language that you may be familiar with, while adding R's capabilities to your applications.

■ R runs on a wide array of platforms, including Windows, Unix, and Mac OS X. It's likely to run on any computer you may have. (even on an iPhone, which is impressive but probably not a good idea.)

■ If you don't want to learn a new language, a variety of graphic user interfaces (GUIs) are available, offering the power of R through menus and dialogs.

R Basics needed for this class

Some of these R aspects will be explained on some level but it is good to review them if necessary.

- Data types in R such as
  - Matrices
  - Lists (vectors where the elements can be of different class)
  - Factors (categorical data)
  - Data Frames (storing tabular data)

- Sub-setting (extracting subsets of R objects)
  - Useful to analyze and display subset of R objects created by the text mining packages which are typically Data Frames, Lists, or Factors.
- Reading/writing data from/to a file
- How to create and define functions
- Familiarity with loop functions
  - We'll use lapply() – loops over a List and evaluates a function on each List element.

Some code examples

- Some code examples we'll encounter

```
dtm # Is a Document Term Matrix

freq <- colSums(as.matrix(dtm)) # Column wise summation to get the term frequencies in the document

ord <- order(freq) # Ordering the frequencies

freq[tail(ord)] # Sub-setting most frequent terms
```

Obtaining and installing R and R Studio

- Download R (http://www.inside-r.org/download)

- R studio is an open source IDE for R
- R Studio can be downloaded from: http://www.rstudio.com/
- You would probably need R to be installed before installing R studio.

- Part of the reason R has become so popular is the vast array of packages available at the CRAN and Bioconductor repositories.
  http://cran.r-project.org/web/views/
  https://www.bioconductor.org/

- The data structures and algorithms can be extended to fit custom demands, since the package is designed in a modular way to enable easy integration of new file formats, readers, transformations and filter operations.

R studio features

Syntax-highlighting editor that supports direct code execution

Workspace management (write/save your scripts)

History tools

Debugging tools

Console

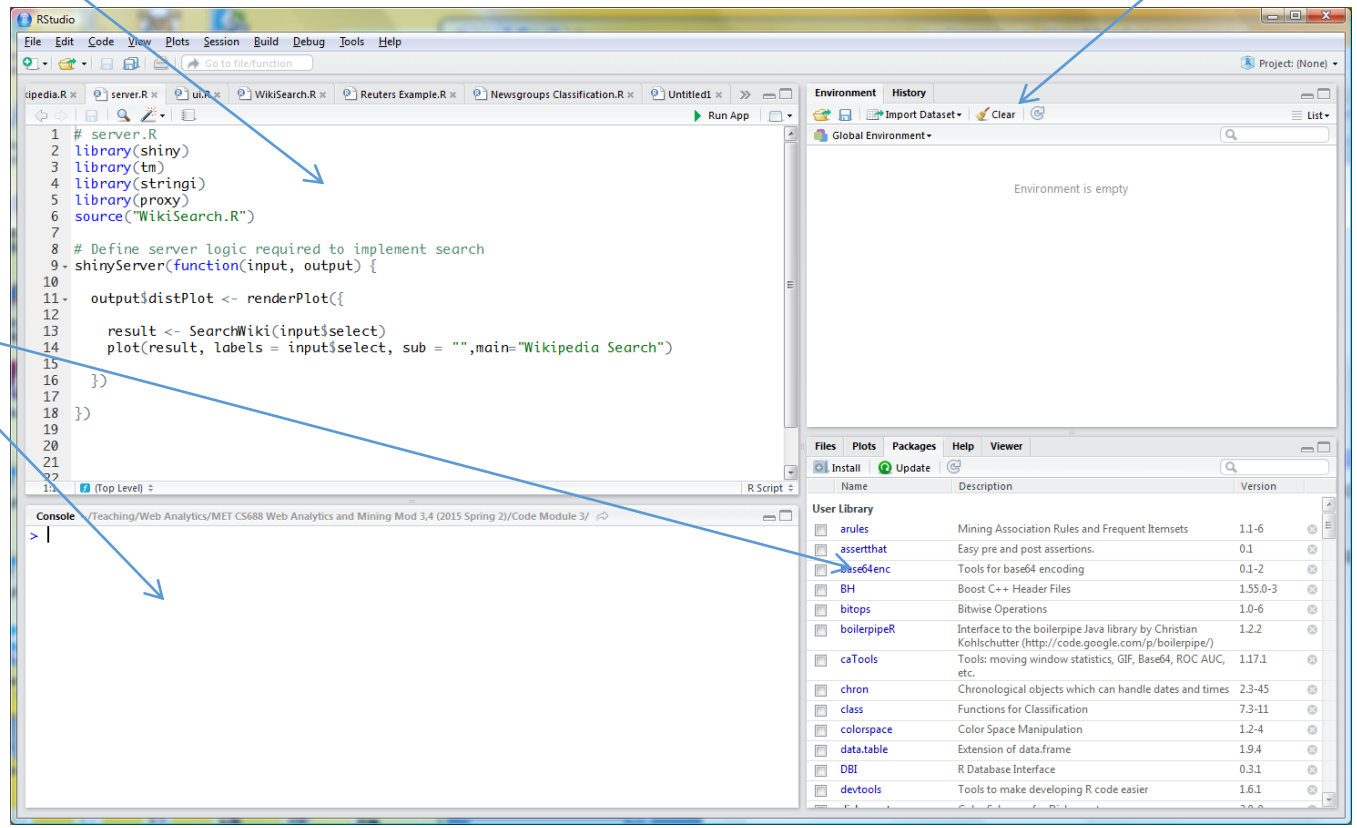
Executed code messages

Packages

Under the tab "Packages" you can see which packages are installed on your computer, add new etc.

Environment management

Analyze/Clear R objects



Basic R Overview - Libraries in R

- Part of the reason R has become so popular is the vast array of packages available at the CRAN (https://cran.r-project.org/) and Bioconductor (https://www.bioconductor.org/) repositories.
- Packages can be thought of as add-on to R that make it easier to perform your specific task.
- In the last few years, the number of packages has grown exponentially!
- To see what is available on CRAN repositories, in R you can type:

  ```
  a <- available.packages()
  ```

- this will put package names in an object a.
- Check “Global Environment” console of R studio to examine object a
- To display the names of the first 3 packages, type:

  ```
  head(rownames(a),3)
  ```

- In R-Studio, under the tab "Packages" you can see which packages are installed on your computer, install more or remove some.
  - remove(a) - removes object a from “Global Environment”

Basic R Overview - Libraries in R

- You can check if you have a particular package installed by typing:

```
> find.package("tm")
```

- In R studio you can access help files on a specific function with “?”

```
?find.package # Access Help Files
> help.search(find.package) # Search Help Files
```

- If the tm package is not installed, you can install the package by typing:

```
> install.packages("tm") # Install a package
```

- Alternatively you can click on the "Install" tab under the "Packages" tab in R-Studio
- Once installed, to use it, you would need to load the package by typing:

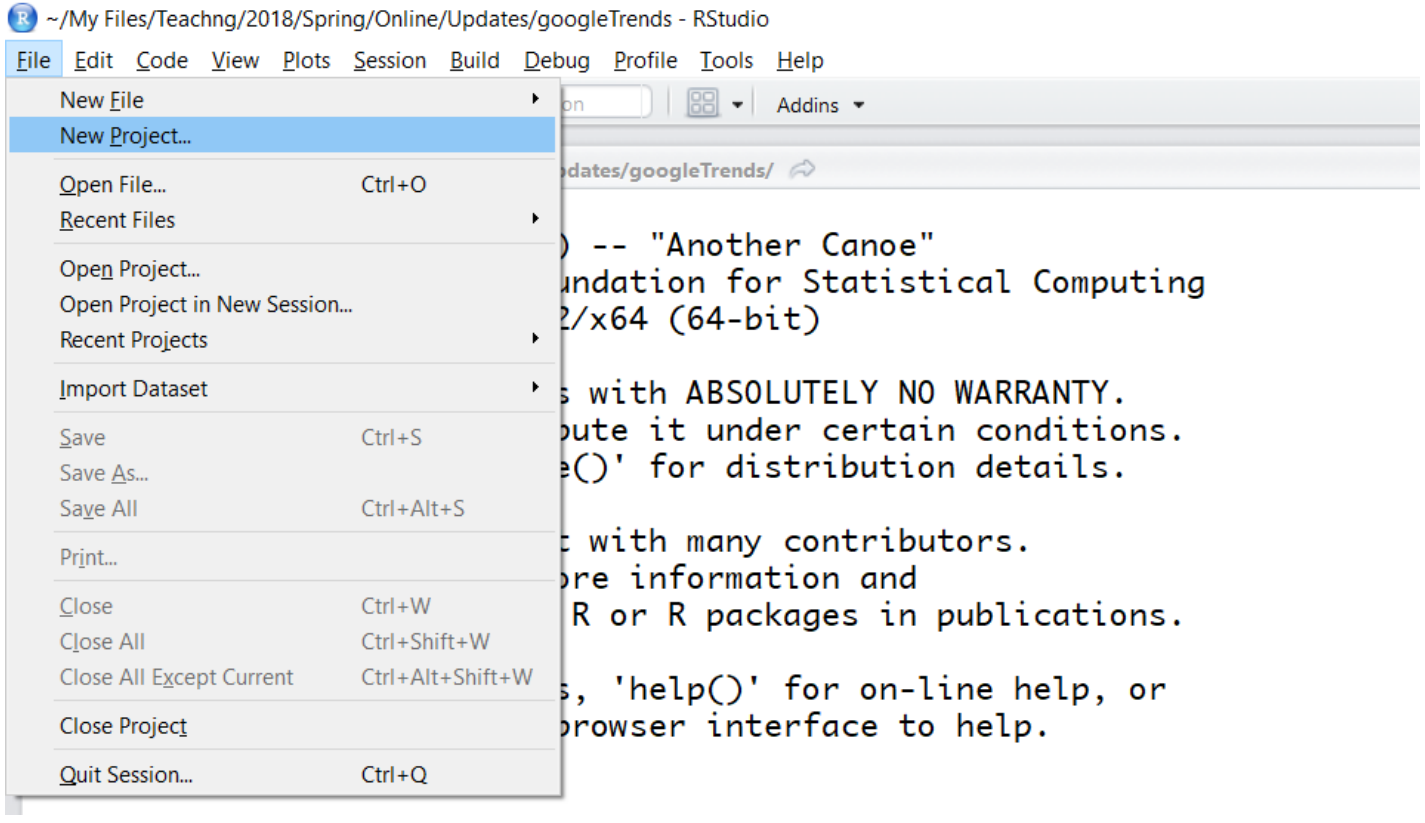
```
library(tm) # Load a package
```

- Also you can unload the package by typing:

```
detach("package:tm", unload=TRUE) # Unload a package
```

Using R Projects

- RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents.
- RStudio projects are associated with R working directories.
- To create a new project use the “New Project” from the “File” menu.



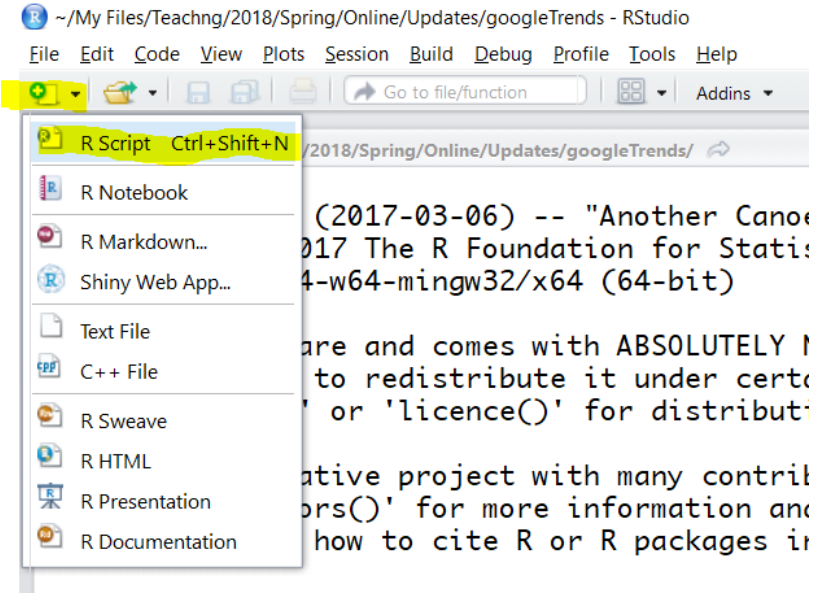
Using R Projects

When a new project is created RStudio:

- Creates a project file (with an .Rproj extension) within the project directory. This file can also be used as a shortcut for opening the project directly from the filesystem.
- Creates a hidden directory (named .Rproj.user) where project-specific temporary files (e.g. auto-saved source documents, window-state, etc.) are stored. This directory is also automatically added to .Rbuildignore, .gitignore, etc. if required.
- Loads the project into RStudio and display its name in the Projects toolbar (which is located on the far right side of the main toolbar)

Once you have created a new project, you can write your R code in a R script, by opening one

- Always use “R Script” file format for your R code.
- Be aware of OS format differences if you copy/paste your code.



Using R Projects

When a project is opened within RStudio the following actions are taken:

- A new R session (process) is started
- The .Rprofile file in the project's main directory (if any) is sourced by R
- The .RData file in the project's main directory is loaded (if project options indicate that it should be loaded).
- The .Rhistory file in the project's main directory is loaded into the RStudio History pane (and used for Console Up/Down arrow command history).
- The current working directory is set to the project directory.
- Previously edited source documents are restored into editor tabs
- Other RStudio settings (e.g. active tabs, splitter positions, etc.) are restored to where they were the last time the project was closed.

Working with R

- R is a case-sensitive, interpreted language.
- You can enter commands one at a time at the command prompt (>) or run a set of commands from a source file.

- Most functionality is provided through built-in and user-created functions and the creation and manipulation of objects. An object is basically anything that can be assigned a value. For R, that is just about everything (data, functions, graphs, analytic results, and more). Every object has a class attribute telling R how to handle it.

- All objects are kept in memory during an interactive session. Basic functions are available by default. Other functions are contained in packages that can be attached

- Statements consist of functions and assignments. R uses the symbol <- for assignments, rather than the typical "=" sign.
- NOTE R allows the = sign to be used for object assignments. But you won't find many programs written that way, because it's not a standard syntax.

R Basics

- In R variables are called R Objects and they can be a combination of the following data types
  - Vectors
  - Matrices
  - Lists (vectors where the elements can be of different class)
  - Factors (categorical data)
  - Data Frames (storing tabular data)
- Getting help on function “apply” – simply type: ?apply
- R used to create and define functions, make packages, web apps.
- Subsetting (extracting subsets of R objects)
  - Useful to analyze and display subset of R objects typically differs for Data Frames, Lists, or Factors.
- Reading writing data – many packages and functions.
- Loop functions – standard. In addition there are few R specific
  - apply – loops over
  - lapply() – loops over a List and evaluates a function on each List element.

Vectors and Matrices Math Overview

$\vec{a}$ - **Vectors** with an arrow over their variable name. Vector is $[m \times 1]$ matrix

Matrices are represented by an uppercase letter.

A is $[m \times n]$ matrix, m-**rows**, n-**columns**.

Matrix multiplication $[m \times p] * [p \times n = [m \times n]$

Non commutative - order maters $A * B \neq B * A$

I **Identity matrix** 1's on the diagonal

Inverse matrix (square A only) $A * A^{-1} = I$, similar to 1/3 being inverse to 3

Singular if A does not have inverse (too close to zero matrix)

Transpose Flipped along the diagonal (from $[m \times n]$ to $[n \times m]$)

Transpose of a matrix: $A = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}$ is $A^T = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$

Magnitude of a vector:

$$\|a\| = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

meaning: length of a vector.

Unit Vector: $\vec{n} = \frac{\vec{a}}{\|\vec{a}\|}$

meaning: Keeps only vector's directionality. Many times just the vector's directionality is needed.

Dot (scalar) product:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b\_n$$

$$\vec{a} \cdot \vec{b} = \|a\| \|b\| \cos \alpha$$

meaning: A dot product of the two vectors $\vec{a} \cdot \vec{b}$ is a measure to what degree two vectors are aligned.

Cosine Similarity is a measure based on the dot product of two vectors, since the angle between the vectors is given as $\cos \alpha = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} = \frac{\vec{a}}{\|\vec{a}\|} \cdot \frac{\vec{b}}{\|\vec{b}\|}$. It compares alignment of two vectors. For example:

Perpendicular $\cos \alpha = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} = \cos 90° = 0$

Collinear $\cos \alpha = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} = \cos 0° = 1$ (exactly opposite $\cos \alpha = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} = \cos 180° = -1$)

Implementation of Vector and Matrix operations in R

A matrix of 3 columns and 2 rows in R can be specified with:

```
> A <- matrix(1:9, nrow=3,ncol=3) # Matrix A (3x3)
> A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> B <- matrix(rep(10,6), nrow=3,ncol=2) # Matrix B (3x2)
> B
     [,1] [,2]
[1,]   10   10
[2,]   10   10
[3,]   10   10
```

Element wise multiplication is obtained using "*"

```
> B*B # Element wise multiplication
     [,1] [,2]
[1,]  100  100
[2,]  100  100
[3,]  100  100
```

Matrix multiplication is obtained using "%*%"

```
> A%*% B # Matrix multiplication
     [,1] [,2]
[1,]  120  120
[2,]  150  150
[3,]  180  180
```

Datatypes in R & Examples

Objects we operate on in R

- Character
- Numeric (in R – all double precision)
- Integer
- Complex
- Logical
- Vector (cannot be mixed datatype)
- Matrices
- Lists (vectors where the elements can be of different class)
- Factors (categorical data)
- Data Frames (storing tabular data)

Each object in R can have an attribute such as

- Names
- Dimensions
- Class
- Length
- And other user defined attributes.

library(dataset) – load package called “dataset”.

x <- 1 – Assign value 1 to object x

x <- 5:20 – Assign values 5 to 20 to object x

print(x) – Display object x

– Used to insert comments

attributes() – Access attributes of object in ()

vector() – Creates vector. Two arguments (class type, length)

y <- c(0.5,6) – Creates vector using function “c” to object x

class(y) – displays object’s attributes

Coercing:

as.numeric()
as.logical()
as.character()

Datatypes in R & Examples

Matrices in R are not special objects (just vectors)

m <- matrix(nrow=2,ncol=3) – define a 2(rows) × 3(columns)

matrix(1:6,nrow=2,ncol=3) – pass values 1:6 to a matrix

x <- 1:3; y <- 1:6;

cbind(x,y) – pass values 1:6 to a 3(rows) × 2(columns) matrix

rbind(x,y) – pass values 1:6 to a 2(rows) × 3(columns) matrix

Lists in R – special type of vector objects (contain elements of different classes).

x <- list(2,"a",TRUE) – define a list of 3 mixed class elements

x[[1]] – subset the first element

names() in R – all R objects have names (or can be assigned)

Following the idea that is easier to follow the variable names than the variable values R has objects called Factors.

Factors in R – categorical data, typically labeled so it is easy to order the data based on labels:

two levels:

x <- factor(c("yes","no","yes","yes"),levels=c("yes","no"))

table(x) – gives content count of each level (3 for "yes")

Order is specified by the order in levels=c("yes","no")

Data Frames in R – special type of list, used to store tabular data. Each column is a list (with its own mixed classes).

x <- data.frame(myInt=1:4,myLogic=c(T,T,T,F)) – creates data frame with column names "myInt" and "myLogic".

Data frames have its own attribute called row.names:

row.names(x)=c("1a","1b","1c","1d") – give rows a name

Datatypes in R - Exercize

- As test create an R object and convert it to different data types

```
x <- 1
x <- 1:20
x <- 'Hello'
x <- c(0.5,0.6) # Function "c" creates a 2-element vector
x <- c(TRUE,FALSE) # You can also use "T" and "F" instead
x <- c('a', 'b', 'c') # characters
x <- c(1+0i, 2+4i) # Complexs
```

- Create a sequence from -10 to 3 with step 2

```
z <- seq(-10, 3, by = 2)
```

Vector() function, no mixing of data types

```
x <- vector('numeric', length=10) # Automatically it is
    initialized with zeros
```

- Try the following:

```
y <- c(1.7, 'a'); # Character, string "1.7"
y <- c(TRUE, 2); # numeric
y <- c('a',TRUE); # Character
```

- Data type coercions are implemented with:

```
as.numeric()
as.character()
as.logical()
as.integer()
as.complex()
```

- Try the above on `x <- 0:6`

Examples of sub-setting R objects

Extracting sub sets of R objects. Few operators can achieve this:

[] – Always returns an object of same class

[[]] – Used to extract a single element from a **List** or **Data Frame**.

$ – Used to extract a **List** or **Data Frame** elements by name.

---

m[1,] – subsets the first row of **matrix** object "m"

x <- list(myInt=1:4,myLogic=c(T,F)) – create a list object "x"

x[1] – returns a **List** "1 2 3 4"

x[[1]] – returns integers "1 2 3 4"

x$myLogic – returns "TRUE FALSE"

x$myInt[3:4] – returns "3 4" from the list name "myInt"

x[c(1,2)] – returns 2 **List** objects "myInt and myLogic"

Subsetting nested elements of a list (note the difference from the previous example)

x1 <- list(a=list(1,2,3,4),b=c(T,F)) – create a list object "x1"

x1[1] – returns a **List** of 4 lists "[1] [2] [3] [4]"

Note:

x1[2] – returns **List** "2" since [] is used

x1[[1]][2] – returns **element** "2" from **List** "a", since [] is used

x1$a[[2]] – returns **element** "2" from **List** "a" since [[]] is used

Partial Matching with [[]] – useful at command line.

x <- list(myList=1:5) – create a list object "x"

x$m – returns the **List** named "myList" by matching "m" to it from the available objects in the workspace.

x[["m",exact="FALSE"]] – another way to achieve the same with [[]]

Removing Missing NA from an object

Common case in Data Analytics to have missing values

x <- c(1, 2, NA, 4, NA, 5) – create data with missing values (NA)

bad <- is.na(x) – find the appearance of the NA as logical

x[!bad] – subset only the non missing data

What if there are multiple objects and how to find the subset of missing values.

y <- c("a", "b", NA, NA, "e", "f") – vector data containing NA

Using function complete.cases(y).

y[complete.cases(y)] – subset only the non missing data

It can be combined complete.cases(x,y) but note x and y have to be of same size.

Also note that it finds the intersect of NA in x and y

Also function complete.cases(y) can be used on dataframes.

Vectorized operations

Examples:

x <-1:4; y <-6:9 – create 2 objects

x + y – elementwise addition (7 9 11 13)

x > 2 – logical (FFTT)

x ≥ 2 – logical (FTTT)

y == 8 – logical (FFTF)

x * y – elementwise multiplication

x / y – elementwise division

Also true not just for vectors but for matrices too.

Examples:

x <- matrix(1:4,nrow=2,ncol=2) – create 2x2 matrix

y <- matrix(rep(10,4),2,2) – create 2x2 matrix with element “10”

x * y – elementwise multiplication of 2 matrices

x / y – elementwise division of 2 matrices

x %*% y – regular matrix multiplication

Parsing Data with R

- **grep()** - takes regex as the first argument, and the input vector as the second argument. grep returns indices.
- **grepl()** - takes the same arguments as the grep function, except for the value argument, which is not supported. grepl returns a logical vector with the same length as the input vector.
- **regexpr()** - returns an integer vector with the same length as the input vector.
- **gregexpr()** - is the same as regexpr, except that it finds all matches (indices) in each string.

```
# Example: Parsing
> grep("F+", c("FCX", "B2Q", "BASF F", "FF"), perl=TRUE, value=FALSE)
[1] 1 3 4
> grepl("F+", c("FCX", "B2Q", "BASF F", "FF"), perl=TRUE)
[1] TRUE FALSE TRUE TRUE
> regexpr("F+", c("FCX", "B2Q", "BASF F", "FFF"), perl=TRUE)
[1] 1 -1 4 1 # The first index of appearance of “F” in the string
[1] 1 -1 1 3 # The number of times “F” appears  in each string
```

Parsing Data with R

- **which()** – function gives the TRUE indices of a logical object, allowing for array indices.
- **match()** – returns a vector of the positions of (first) matches of its first argument in its second.
- **%in%** – logical vector indicating if there is a match or not for its left operand.

```
# Example: Parsing
> which(c("FCX", "B2Q", "BASF F", "FF") != “B2Q”)
[1] 1 3 4

> c("FCX","B2Q", "BASF")  %in% c("FCX", "B2Q", "BASF F", "FFF")
[1] TRUE TRUE FALSE

> c("FCX", "B2Q", "BASF F", "FFF") %in% c("FCX","B2Q", "BASF")
[1] TRUE TRUE FALSE FALSE
```

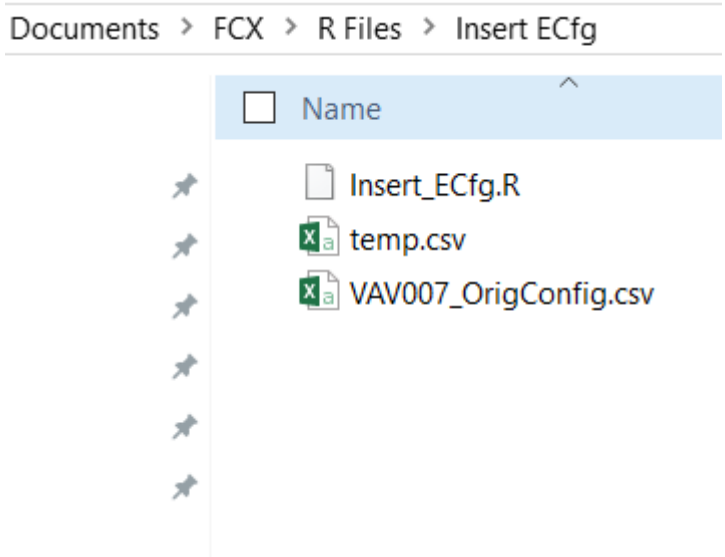
Data Import in R

Reference:

Some practical work-related examples

Paths in R - Example

- Consider the following folder
- Note how the code relates to the path
- and to one of the CSV files that is being read.




```
4  # ---- Paths in R - Examples -------
5  ###--- File Import ----
6  pth <- file.path("c:", "Users", "ZlatkoFCX","Documents","FCX","R Files","Insert ECfg")
7  CSV.files <- list.files(pth,".csv") # List of files with pattern ".csv"
8  R.files <- list.files(pth,".R") # List of files with pattern ".R"
9  Import.Files <- file.path(pth,"VAV007_OrigConfig.csv") # Specific File Name to Import
10 Import.Files <- file.path(pth,CSV.files[2]) # Another Way
11
```

Paths in R - Example

- Some code examples
  pth <- file.path("c:", "Users", "ZlatkoFCX","Documents","FCX","R Files","VAV Data Files") – specify the path to the files
  setwd(pth) – set the path to the files (if needed)

- Get all the files in the folder with
  myPDFfiles <- list.files(pth,"*.pdf") # Get All PDF files in the folder specified by the "pth"

- Using lapply() several pdf files that are contained in a single folder (specified by the R object "myPDFfiles") can be converted to text with an in line function such as this,
  lapply(myPDFfiles, function(i) system(paste(Sys.which("pdftotext") , paste0('"', i, '"')), wait = FALSE))
- Note how each PDF file converted into a text file is indexed by "i"

Some reading/writing functions

Reading data

```
> read.table()  # Reads tabulated data
> read.csv()  # Reads comma separated text file
> read.Lines()  # Reads lines of a text file
> scan()  # Reads the entire contents of a text file into an R object
> source()  # Reads in R code files (R functions etc.)
> dget()  # Reads in R objects
> load()  # Reads in binary objects or objects saved in the workspace.
> unserialize() # Reads in binary objects or objects saved in the workspace.
```

```
> scan()
> scan("file.txt", character(0)) # separate each word
> scan("file.txt", character(0), quote = NULL) # get rid of quotes
> scan("file.txt", character(0), sep = ".") # separate each sentence
> scan("file.txt", character(0), sep = "\n") # separate each line
```

Writing data

```
> write.table() - # Writes tabulated data
> write.Lines()  # Writes lines of a text file
> dump()  # Writes in R code files (R functions etc.)
> dput()  # Writes in R objects
> save()  # Writes in binary objects or objects saved in the workspace.
> serialize()  # Writes in binary objects or objects saved in the workspace.
```

Examples of reading CSV files

- Some code examples
  pth <- file.path("c:", "Users", "ZlatkoFCX","Documents","FCX","R Files","VAV Data Files") – specify the path to the
  setwd(pth) – set the path to the files (if needed)

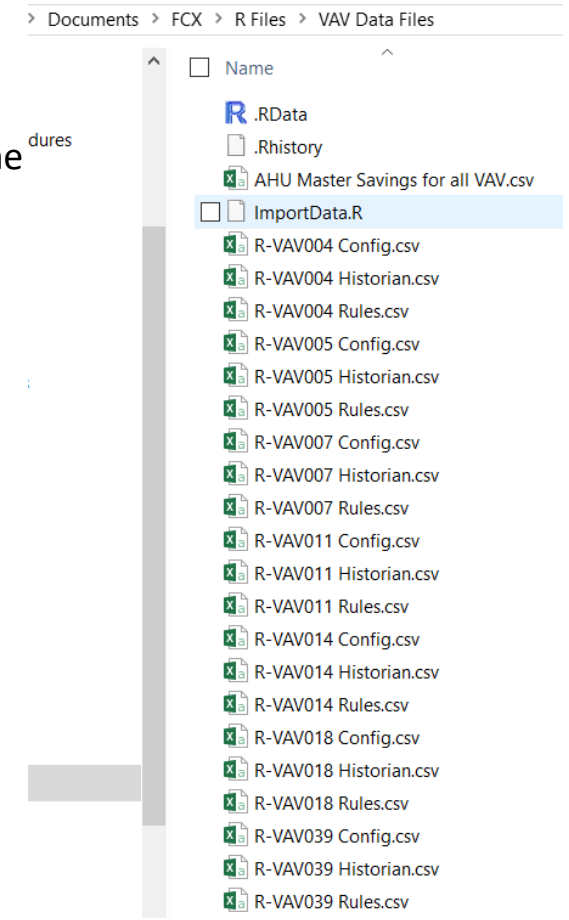
- Consider accessing the files in the folder shown in the image
  pth <- file.path("c:", "Users", "ZlatkoFCX","Documents","FCX","R Files","VAV Data Files")

- Get the files in the folder with
  CSV.Rules.files <- list.files(pth,"Rules.csv") # Get All CSV “Rules” files in the folder specified by the “pth”

- Read the CSV files using for(),

```
CSV.Content <- list(); # Initialize a list holding the CSV file content
for (ff in 1:length(CSV.Rules.files)) {
  CSV.Content.Rules[[ff]] <- read.csv(CSV.Rules.files[ff],stringsAsFactors = FALSE,blank.lines.skip = TRUE)
}
```

Note: Sometimes parameters to read.csv() needed to be specified to capture the desired format.



Basic R Overview – Editing Strings

By default cat() concatenates vectors when writing to the text file. You can change it by adding options sep="\n" or fill=TRUE. The default encoding depends on your computer.

```
cat(text,file="file.txt",sep="\n")
writeLines(text, con = "file.txt", sep = "\n", useBytes = FALSE)
```

paste() - Strings can be also concatenated using the paste function. The function takes a variable number of arguments of any type and returns a string using space as the default separator. The separator may also be explicitly specified using the sep argument.

paste0() - Concatenates the argument strings without any separator.

substr() - The substr function is used for extracting portions of the specified string. The start and stop arguments are required. The first character is at position number 1.

sub() - The sub function is used for substituting the first matching string or regular expression (the first argument) with the specified string (the second argument). The new string is returned. The original string is not modified.

Read/write a text file - Exercise

- How to read and write a text file with R
- Data is read in via **connection()** interfaces. It interfaces connection to files or compressed files.

```
file() # Opens connection to a file
gzfile() # Opens connection to a compressed file with gzip algorithm
bzfile() # Opens connection to a compressed file with bzip2
url() # Opens connection to a web page
```

- You can write the content of an R object into a text file using

```
cat()
writeLines()
```

- For practice, test some of these factions using corresponding files.

Basic R Overview – Exercise

Select a text file to work with

- Before reading a text file, you can look at its properties with

```
nlines() # From parser package
countLines() # From R.utils package
count.chars() # From parser package (counts the number of bytes and characters in each line of a file)
```

Moving text data between R and the clipboard

- R has a function readClipboard() that does reads **text** data to R from the clipboard.

  y <- readClipboard() # will assign the contents of the clipboard to the vector y.

- The function writeClipboard() does the opposite.
- Consider the following example of comparing these 2 sets and finding the intersect.
  - Change first "-" into "_" in: "ECfg_AHU_Design_Supply-Airflow" into "ECfg_AHU_Design_Supply_Airflow"
  - Add "Ecfg_" to Set 2.
  - Find Set 1 and Set 2 intersect.

```
24  # ---- Moving data between R and the clipboard  -------
25  # --- Clipboard Import & Edditing
26  y1 <- readClipboard() # Set 1 from "Two_Sets_Example.xlsx"
27  y1 <-  gsub("-","_",y1) # Replace "-" with "_"
28
29  y2 <- readClipboard() # Set 2 from "Two_Sets_Example.xlsx"
30  y2 <- paste0("ECfg_",y2) # Paste & Insert Names
31
32  #--- Set Operations
33  Difference1 <- setdiff(y1,y2) # What is Different in y1
34  Difference2 <- setdiff(y2,y1) # What is Different in y2
35  Common <- intersect(y1,y2) # Common
36
37  #--- Export to Clipboard
38  writeClipboard(as.character(Difference)) # Place it on Clipboard
39  writeClipboard(as.character(Common)) # Place it on Clipboard
```

	Set 1	Set 2
2	ECfg_AHU_Design_Supply-Airflow	
3	ECfg_AHU_Design_Return-Airflow	AHU_Design_Return_Airflow
4	ECfg_AHU_SF_Nom_Power	AHU_SF_Nom_Power
5	ECfg_AHU_RF_Nom_Power	AHU_RF_Nom_Power
6	ECfg_VAV_Reheat_Capacity	VAV_Reheat_Capacity
7	ECfg_AHU_Heat_Capacity	HeatValvePSI_Open
8	ECfg_AHU_Cool_Capacity	
9		

	Difference	Intersect
12	ECfg_AHU_Design_Supply_Airflow	ECfg_AHU_Design_Return_Airflow
13	ECfg_AHU_Heat_Capacity	ECfg_AHU_SF_Nom_Power
14	ECfg_AHU_Cool_Capacity	ECfg_AHU_RF_Nom_Power
15		ECfg_VAV_Reheat_Capacity

Moving numeric data between R and the clipboard

- R's function scan() can be used to copy a column of **numbers** from Excel to R.
- Copy the column from Excel by
  1. Running in R y <- scan(),
  2. type **Ctrl-v** to paste into R,
  3. and press **Enter** to signal the end of input to scan.
- Then y will contain the numbers from Excel as numbers, not as quoted strings.

Values
75
56
65
34
12

```
> y <- scan()
1: 75
2: 56
3: 65
4: 34
5: 12
6:
Read 5 items
> y
[1] 75 56 65 34 12
>
```

Moving data between R and Excel

- Accomplished by using R functions

  read.table()

  write.table()

- Rows are combined into single entries.

  write.table(y, "clipboard", sep="\t")

- Will copy a table y to the clipboard in such a way that it can be pasted into Excel preserving the table structure.
- By default, the row and column names will come along with the table contents. To leave the row names behind, add the argument

  row.names=FALSE

- to the call to write.table.

The R Apply family

```
# ---- The R Apply family of functions  -------
Age<-c(53,32,57,23,25,38)
Weight<-c(74,62,66,47,59,99)
Height<-c(175, 166,187,197,186,197)
DF<-data.frame(Age,Weight,Height)
```

apply(x,Margin,FUN,…)

```
# 1) apply(x,Margin,FUN) Returns a vector, array or list.
apply(DF,1,sum) # Row wise sum up of DF
apply(DF,2,mean) # Column wise mean of DF
a<-5; apply(DF,2,function(x,a) mean(x+a),a)
```

```
> apply(DF,1,sum) # Row wise sum up of DF
[1] 302 260 310 267 270 334
> apply(DF,2,mean) # Column wise mean of DF
      Age    Weight    Height
 38.00000  67.83333 184.66667
> a<-5; apply(DF,2,function(x,a) mean(x+a),a)
      Age    Weight    Height
 43.00000  72.83333 189.66667
>
```

lapply(list, FUN,…)

```
# 2) lapply(x,FUN,...) Returns a list.
lapply(DF, function(DF) DF/2) # List of half the the values in entire DF
lapply(DF, mean) # Mean per field
a<-5; b<- pi; lapply(DF$Age, function(x,a,b) x + a*b, a, b) # Add "a*b" to Age field
```

```
> lapply(DF, function(DF) DF/2) # List of half the the values in entire DF
$Age
[1] 26.5 16.0 28.5 11.5 12.5 19.0

$Weight
[1] 37.0 31.0 33.0 23.5 29.5 49.5

$Height
[1] 87.5 83.0 93.5 98.5 93.0 98.5

> lapply(DF, mean) # Mean per field
$Age
[1] 38

$Weight
[1] 67.83333

$Height
[1] 184.6667
```

The R Apply family

mapply(list, FUN,…)

```
# 4) mapply(x,FUN,...) multivariate version of sapply
v1 <- 1:3; mapply(sum, v1, v1, v1) # Elementwise Sum first (1+1+1), second, third
# [1] 3 6 9
```

sapply(x,Margin,FUN,…)

```
# 3) sapply(x,FUN,...) Returns a vector.
sapply(DF, function(DF) DF/2) # Matrix of half the the values in entire DF
My.List <- list(str1=c("This","is","long","string"),str2=c("qwery"))
sapply(My.List, nchar) # Returns a list
```

```
> sapply(DF, function(DF) DF/2) # Matrix of half the the values in entire DF
      Age Weight Height
[1,] 26.5   37.0   87.5
[2,] 16.0   31.0   83.0
[3,] 28.5   33.0   93.5
[4,] 11.5   23.5   98.5
[5,] 12.5   29.5   93.0
[6,] 19.0   49.5   98.5
> My.List <- list(str1=c("This","is","long","string"),str2=c("qwery"))
> sapply(My.List, nchar) # Returns a list
$str1
[1] 4 2 4 6

$str2
[1] 5
```