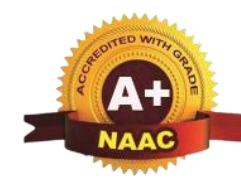


Data Analytics with R

BDS306C

Prepared By,
Dr. Anitha DB
Associate Professor & Head
Department of CSE-Data Science
ATME College of Engineering, Mysuru



Topics

- Datasets
- Importing and Exporting files
- Accessing Databases
- Data Cleaning and Transformation

I. Datasets

- R has many built in datasets.
- Any one can see all the datasets available in the loaded packages using the **data()** function.
- To **access** a particular dataset use the **data()** function with its argument as the dataset name enclosed within double quotes and the second optional argument being the package name in which the dataset is present

Example

```
> data("cancer", package="survival")  
> print(head(cancer))
```

	inst	time	status	age	sex	ph.ecog	ph.karno	pat.karno	meal.cal	wt.loss
1	3	306	2	74	1	1	90	100	1175	NA
2	3	455	2	68	1	0	90	90	1225	15
3	3	1010	1	56	1	0	90	90	NA	15
4	5	210	2	57	1	1	90	60	1150	11
5	1	883	2	60	1	0	100	90	NA	0
6	12	1022	1	74	1	1	50	80	513	0



II. Importing and Exporting Files

Types of Files

1. Text and CSV files
2. Unstructured Files
3. XML and HTML Files
4. JASON and YAML Files
5. Excel Files
6. SAS,SPSS and MATLAB Files
7. Web Data

1 Text and CSV files

- Text documents have several formats. **Comma separated values(CSV) file** is a spreadsheet like data stored with comma delimited values.
- read.table()** and **read.csv()** functions are used to read(**import**) the file.
- The **read.table()** function reads these files and stores the result in data frame. If the data has header, it is required to pass the argument **header=TRUE** to the **read.table()** function. The argument **fill=TRUE** makes the read,table() function substitute **NA** values for the **missing fields**.
- The functions **write.table()** and **write.csv()** functions are used to Write data from R into file(**Export**)



2 Unstructured Files

readLines() function is used to read from the file

writeLines() function is used to write the text to the files

Example

```
tempset<-readLines("F:/Tempset.txt")
```

```
writeLines("This book is about a analysis of data using R","F:/dar.csv")
```



3 XML and HTML Files

- To read XML files, the XML package has to be installed.
- The functions for importing XML pages are **xmlParse()** and **xmlTreeParse()**
- The functions for importin HTML pages are **htmlParse()** and **htmlTreeParse()**

4 JASON and YAML Files

- The two packages dealing with JSON data are RJSONIO and rjson.
- The function used to import the JSON file is fromJSON() and the function used to export the JSON file is toJSON().
- The functions used to import the YAML data file are yaml.load() and yaml.load_files().

5 Excel Files

- The spreadsheets can be imported with the functions **read.xlsx()** and **read.xlsx2()**
- To write to an excel file, the **write.xlsx2()** function will be used

6 SAS,SPSS and Matlab Files

- The **read.ssd()** function is used to read SAS datasets and **read.dta()** function is used to read Stata DTA files.
- The **read.spss()** function is used to import the SPSS data files.
- The Matlab binary data files can be read and written using the **readMat()** and **writeMat()** functions in the R.matlab package.
- The files in **picture formats** can be read via the **jpeg, png, tiff, rtff** and **readbitmap** packages



7 Web Data

- R has ways to import data from web sources using Application Programming Interface(API).
- The **read.table()** function can accept URL, rather than a local file.
- Accessing a large file from internet can be slow and if the file is required frequently, it is better to download the file using **download.file()** function and create a local copy and then import.



III. Accessing Databases

R can connect to all database management systems(DBMS) like SQLite, MySQL, MarioDB,PostgreSQL and Oracle using the DBI package.

The function **dbReadTable()** reads a table from the connected database and the function **dbListTables()** can list all the tables in the database.

The function **dbDisconnect()** is used for disconnectiong and function **dbUnloadDriver()** is used to unload the defined database driver.



IV. Data Cleaning and Transforming

1. Manipulating string
2. Manipulating Data Frames
3. Data Reshaping
4. Grouping Functions

IV. Data Cleaning and Transforming

1. Manipulating string

In some datasets or data frames logical values are represented as “Y” and “N” instead of TRUE and FALSE.

In such cases it is possible to replace the string with correct logical values.

Example

```
a<-c(1,2,3)
```

```
b<-c("a","b","c")
```

```
d<-c('Y','N','Y')
```

```
df1<-data.frame(a,b,d)
```

```
Print(df1)
```

	a	b	d
1	1	a	Y
2	2	b	N
3	3	c	Y

```
convt<-function(x)
{
  Y<-rep.int(NA,length(x))
  Y[x=='Y']<-TRUE
  Y[x=='N']<-FALSE
  Y
}
df1$d<-convt(df1$d)
df1
}
```

	a	b	d
1	1	a	TRUE
2	2	b	FALSE
3	3	c	TRUE

IV. Data Cleaning and Transforming

1. Manipulating string

- The function **grep()** and **grepl()** are used to find a pattern in a given text.
- The functions **sub()** and **gsub()** are used to replace a pattern with another in a given text.
- The above four functions belong to the **base package**.
- The functions **str_detect()** in the **stringr package** does the same function of detecting the presence of a given pattern in the given text.

```
> grep('my', 'This is my pen')
[1] 1
> grepl('my', 'This is my pen')
[1] TRUE
> sub("my", "your", "This is my pen")
[1] "This is your pen"
> gsub('my', 'your', 'This is my pen')
[1] "This is your pen"
```



IV. Data Cleaning and Transforming

2. Manipulating Data Frames: Add a column to a data frame (Create a data frame with 3 column and then add one more column to the data frame)

```
> name<-c("Jhon","Peter","Mark")
> start_date<-c("1980-10-10","1999-12-12","1900-04-05")
> end_date<-c("1989-03-08","2004-04-20","2000-09-25")
> service<-data.frame(name,start_date,end_date)
> service
```

	name	start_date	end_date
1	Jhon	1980-10-10	1989-03-08
2	Peter	1999-12-12	2004-04-20
3	Mark	1900-04-05	2000-09-25

```
> service$period<-as.Date(service$end_date)-as.Date(service$start_date)
> service
```

	name	start_date	end_date	period
1	Jhon	1980-10-10	1989-03-08	3071 days
2	Peter	1999-12-12	2004-04-20	1591 days
3	Mark	1900-04-05	2000-09-25	36698 days



IV. Data Cleaning and Transforming

2. Manipulating Data Frames: Add a column to a data frame using with() function

The same can be achieved using the function with()

```
> service$period<-with(service,as.Date(end_date)-as.Date(start_date))  
> service
```

	name	start_date	end_date	period
1	Jhon	1980-10-10	1989-03-08	3071 days
2	Peter	1999-12-12	2004-04-20	1591 days
3	Mark	1900-04-05	2000-09-25	36698 days

IV. Data Cleaning and Transforming

2. Manipulating Data Frames: Add multiple columns to a data frame using within() function

within() function can be used to add multiple columns to the dataframe.

```
service<-within(service,  
  {period<-as.Date(end_date)-as.Date(start_date)  
  highperiod<-period>2000  
  })  
service
```

Output

	name	start_date	end_date	period	highperiod
1	Jhon	1980-10-10	1989-03-08	3071 days	TRUE
2	Peter	1999-12-12	2004-04-20	1591 days	FALSE
3	Mark	1900-04-05	2000-09-25	36698 days	TRUE

IV. Data Cleaning and Transforming

2. Manipulating Data Frames: The function `sort()` sorts the given vector of numbers or strings. The `order()` function is more useful than `sort()` function as it can be used to manipulate the data frames easily.

`sort()` and `order()` function

```
x<-c(5,10,3,15,6,8)
sort(x)
sort(x,decreasing = TRUE)
y<-c("X","AB","Deer","For","Moon")
sort(y)
sort(y,decreasing = TRUE)
order(x)
x[order(x)]
identical(sort(x),x[order(x)])
```

[1] 3 5 6 8 10 1

[1] 15 10 8 6 5 3

"AB" "Deer" "For" "Moon" "X"

"X" "Moon" "For" "Deer" "AB"

3 1 5 6 2 4

3 5 6 8 10 15

TRUE

IV. Data Cleaning and Transforming

2. Manipulating Data Frame

Order() function is more useful than the **sort()** function as it can be used to **manipulate the data frame** easily.

```
name<-c("Jhon","Peter","Mark")
start_date<-c("1980-10-10","1999-12-12","1900-04-05")
end_date<-c("1989-03-08","2004-04-20","2000-09-25")
service<-data.frame(name,start_date,end_date)
service
startdt<-order(service$start_date)
service.ordered<-service[startdt,]
service.ordered
```

	name	start_date	end_date
1	Jhon	1980-10-10	1989-03-08
2	Peter	1999-12-12	2004-04-20
3	Mark	1900-04-05	2000-09-25

	name	start_date	end_date
3	Mark	1900-04-05	2000-09-25
1	Jhon	1980-10-10	1989-03-08
2	Peter	1999-12-12	2004-04-20

IV. Data Cleaning and Transforming

3. Data Reshaping

Data reshaping in R is about changing the way data is organized into rows and columns
The **cbind()** function can be used to join multiple vectors to create a data frame.
The **rbind()** function can be used to merge two data frames

```
city<-c('Mandya','Mysore','Chennai')
state<-c('KA','KA','TN')
zipcode<-c('571401','570001','600001')
address<-cbind(city,state,zipcode)
address
```

	city	state	zipcode
[1,]	"Mandya"	"KA"	"571401"
[2,]	"Mysore"	"KA"	"570001"
[3,]	"Chennai"	"TN"	"600001"

```
new.address<-data.frame(
  city=c('Banglore','Coimbatore'),
  state=c('KA','TN'),
  zipcode=c('530068','631027'),
  stringsAsFactors = FALSE)
print(new.address)
all.address<-rbind(address,new.address)
all.address
```

	city	state	zipcode
1	Banglore	KA	530068
2	Coimbatore	TN	631027

	city	state	zipcode
1	Mandya	KA	571401
2	Mysore	KA	570001
3	Chennai	TN	600001
4	Banglore	KA	530068
5	Coimbatore	TN	631027

IV. Grouping Function

i) **apply()** function is used to apply a function to **rows or columns** of a **matrix or array**.

Examples

```
> M <- matrix(seq(1,16),4,4)
```

```
> apply(M,1,min)
```

```
[1] 1 2 3 4
```

```
> apply(M,2,max)
```

```
[1] 4 8 12 16
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	5	9	13
[2,]	2	6	10	14
[3,]	3	7	11	15
[4,]	4	8	12	16

```
> M <- array(seq(32),dim=c(4,4,2))
```

```
> apply(M,1,sum)
```

```
[1] 120 128 136 144
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	5	9	13
[2,]	2	6	10	14
[3,]	3	7	11	15
[4,]	4	8	12	16
, , 2				
[1,]	17	21	25	29
[2,]	18	22	26	30
[3,]	19	23	27	31
[4,]	20	24	28	32

```
> apply(M,c(1,2),sum)
```

	[,1]	[,2]	[,3]	[,4]
[1,]	18	26	34	42
[2,]	20	28	36	44
[3,]	22	30	38	46
[4,]	24	32	40	48

Note: 1 indicates row, 2 indicates column

IV. Grouping Function

1. `apply()`
2. `lapply()`
3. `sapply()`
4. `vapply()`
5. `mapply()`
6. `rapply()`
7. `tapply()`

IV. Grouping Function

ii) **lapply()** function is used to apply the function to each elements of a **list** in turn and get a list back.

```
> x<-list(a=1,b=1:3,c=10:100)
>
> x
$a
[1] 1

$b
[1] 1 2 3

$c
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
[29] 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
[57] 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
[85] 94 95 96 97 98 99 100
```

```
> lapply(x,FUN=length)
$a
[1] 1

$b
[1] 3

$c
[1] 91
```

```
> lapply(x,FUN=sum)
$a
[1] 1

$b
[1] 6

$c
[1] 5005
```


IV. Grouping Function

iii)**apply()** function is used to apply the function to each elements of a **list** in turn and get a vector back.

```
> x<-list(a=1,b=1:3,c=10:100)
>
> x
$a
[1] 1

$b
[1] 1 2 3

$c
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
[29] 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
[57] 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
[85] 94 95 96 97 98 99 100
```

```
> sapply(x,FUN=length)
a b c
1 3 91
> sapply(x,FUN=sum)
a b c
1 6 5005
```

IV. Grouping Function

iv)vapply() function: sapply() function takes more time to perform the function on the list and to return the vector. **vapply()** function takes less time to perform the function on the list and to return the vector. So vapply() function save some time.

```
> x<-list(a=1,b=1:3,c=10:100)
> x
$a
[1] 1

$b
[1] 1 2 3

$c
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
[29] 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
[57] 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
[85] 94 95 96 97 98 99 100
```

```
> vapply(x, FUN=length,FUN.VALUE=0L)
a  b  c
1  3 91
```


IV. Grouping Function

(v)mapply() function is used to apply the function to the 1st elements of each, and second elements of each, etc., forcing the result to a vector/array.

```
> mapply(sum,1:5,1:5,1:5)
[1] 3 6 9 12 15

[1+1+1 2+2+2 3+3+3 4+4+4 5+5+5]

> mapply(rep,1:4,4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

IV. Grouping Function

(vi) **rapply()** function is used to apply a function to each element of nested list structure recursively.

```
myFun<-function(x)
{
  if(is.character(x))
  {
    return(paste(x, '!', sep=" "))
  }
  else
  {
    return(x+1)
  }
}
l<-list(a=list(a1="Boo",b1=2,c1="Eeeks"),
        b=3,c="Yikes",d=list(a2=1,b2=list(a3="Hey",b3=5)))
rapply(l,myFun)
```

```
> rapply(l,myFun,how='replace')
$a
$a$a1
[1] "Boo !"

$a$b1
[1] 3

$a$c1
[1] "Eeeks !"

$b
[1] 4

$c
[1] "Yikes !"

$d
$d$a2
[1] 2

$d$b2
$d$b2$a3
[1] "Hey !"

$d$b2$b3
[1] 6
```

a.a1	a.b1	a.c1	b	c	d.a2	d.b2.a3	d.b2.b3
"Boo !"	"3"	"Eeeks !"	"4"	"Yikes !"	"2"	"Hey !"	"6"

IV. Grouping Function

(vii) **tapply()** function is used when we want to apply a function to subsets of a vector and subsets are defined by other vector, usually a factor.

```
x<-1:20  
x  
y<-factor(rep(letters[1:5],each=4))  
y  
tapply(x,y,sum)
```

```
> x<-1:20  
> x  
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
> y<-factor(rep(letters[1:5],each=4))  
> y  
[1] a a a a b b b b c c c c d d d d e e e e  
Levels: a b c d e  
> tapply(x,y,sum)  
 a  b  c  d  e  
10 26 42 58 74
```

IV. Grouping Function

`by()` function can be thought of, as a “wrapper” for `tapply()` function

```
> cta<-tapply(iris$Sepal.width,iris$Species,summary)
> cba<-by(iris$Sepal.width,iris$Species,summary)
> cta
```

\$setosa	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	2.300	3.200	3.400	3.428	3.675	4.400

\$versicolor	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	2.000	2.525	2.800	2.770	3.000	3.400

\$virginica	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	2.200	2.800	3.000	2.974	3.175	3.800

```
> cba
iris$Species: setosa
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 2.300   3.200   3.400   3.428   3.675   4.400
-----
iris$Species: versicolor
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 2.000   2.525   2.800   2.770   3.000   3.400
-----
iris$Species: virginica
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 2.200   2.800   3.000   2.974   3.175   3.800
```

IV. Grouping Function

aggregate() function- different way using tapply() function

```
> att<-tapply(iris$Sepal.Length,iris$Species,mean)
> agt<-aggregate(iris$Sepal.Length,list(iris$Species),mean)
> att
      setosa versicolor  virginica
      5.006      5.936      6.588
> agt
  Group.1      x
1   setosa 5.006
2 versicolor 5.936
3  virginica 6.588
```