Introduction to R and RStudio

Course Taught at SUAD

Tanujit Chakraborty & Madhurima Panja (TA)

MDA Course @ Sorbonne

Code Link: https://github.com/tanujit123/MATH-260

What we quest to achieve through the upcoming session

- Introduction and installation of R and R Studio
- Basics of R Studio
- Understand different data types in R
- Data structures in R
- Creating vectors, matrices, lists, data frames
- Manipulating data structures in R
- Functions in R (loops and conditional statements)
- Descriptive Statistics
- Data Visualization

History of R…

- R is an open source programming language and software environment for statistical computing and graphics.

- The R language is widely used among statisticians and data scientists for developing statistical and data analytics tools.

- Modelled after S & S plus, developed at AT&T labs in late 1980s.

- R project was started by Robert Gentleman and Ross Ihaka Department of Statistics, University of Auckland (1995).

- Currently maintained by R core development team – an international team of volunteer developers (since 1997).

ROBERT GENTLEMAN

ROSS IHAKA

Download R & R Studio

- http://www.rproject.org/
- http://cran.rproject.org/doc/contrib/VerzaniSimpleR.pdf

- Download R: http://cran.r project.org/bin/
- Download RStudio: http://www.rstudio.com/ide/download/desktop
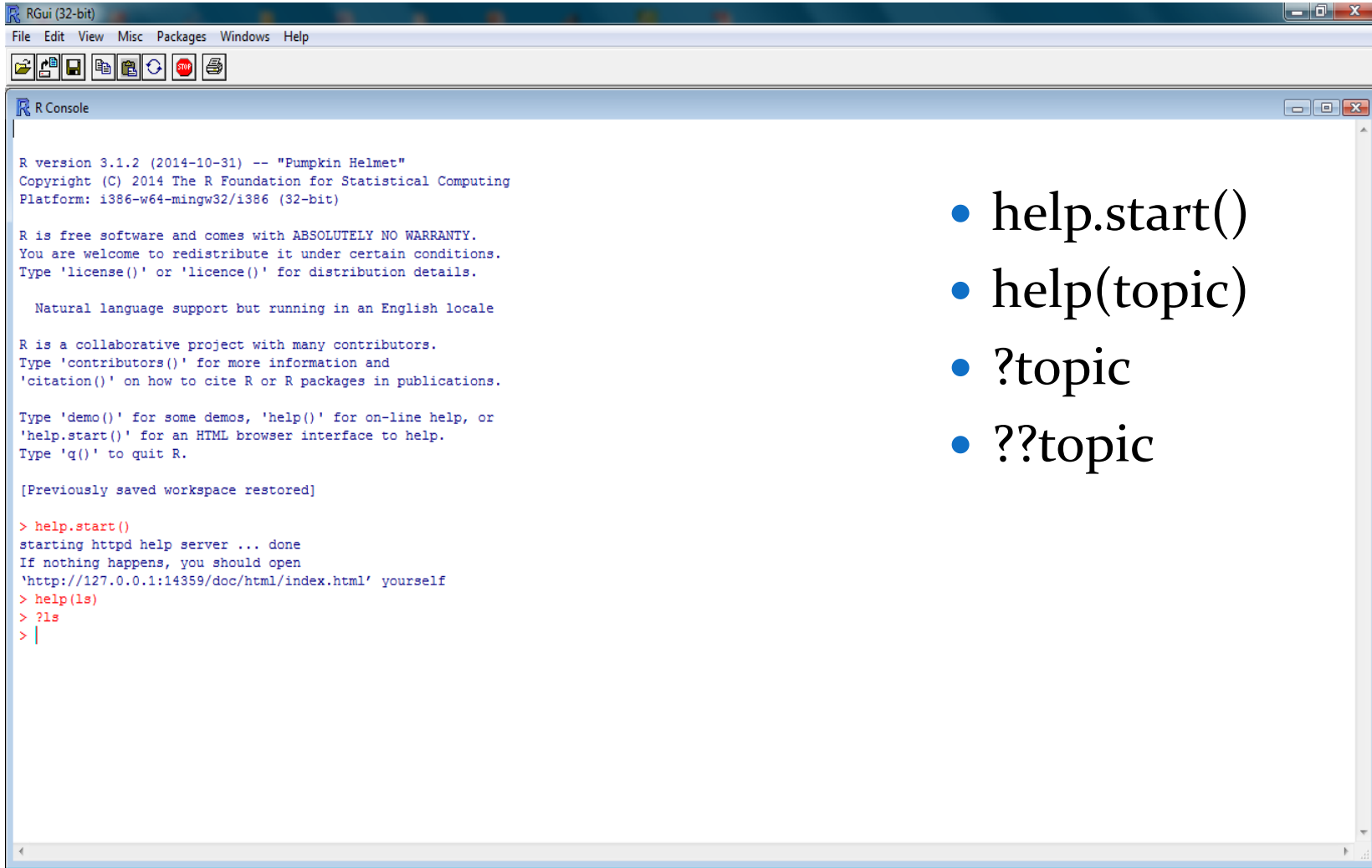
INSTALLATION

1. Download R (Windows) software from **http://cran.r project.org/bin/windows/base/**
2. Run the R set up (exe) file and follow instructions
3. Double click on the R icon in the desktop and R window will open
4. Download RStudio from **http://www.rstudio.com/**
5. Run R studio set up file and follow instructions
6. Click on R studio icon, R Studio IDE Studio will load
7. Tools – Global Options – Appearances – Change Colour Size Theme
   (if you wish to change the background, not a mandatory step)
4. Go to R Script (Ctrl + Shift + N)
5. Write 'Hello World !'
6. Save & Run (Ctrl + Enter)

Congrats ! You have written your very first R Program

Getting help from R console

- help.start()
- help(topic)
- ?topic
- ??topic

R command in integrated environment

How to use R for simple Mathematics

- > 3+5
- > 12 + 3 / 4 – 5 + 3*8
- > (12 + 3 / 4 – 5) + 3*8
- > pi * 2^3 – sqrt(4)
- >factorial(4)
- >log(2,10)
- >log(2, base=10)
- >log10(2)
- >log(2)

Note

- **R ignores spaces**

How to store results of calculations for future use

- > x = 3+5
- > x
- > y = 12 + 3 / 4 − 5 + 3*8
- > y
- > z = (12 + 3 / 4 − 5) + 3*8
- > z
- > A < 6 + 8   ## no space should be between < &
- > a   ## Note: R is case sensitive
- >A

Using C command

- > data1 = c(3, 6, 9, 12, 78, 34, 5, 7, 7)   ## numerical data
- > data1.text = c('Mon', 'Tue', "Wed") ## Text data
-   ## Single or double quote both ok
-                            ##copy/paste into R console may not work
- > data1.text = c(data1.text, 'Thu', 'Fri')

Scan command for making data

- > **d3 = scan(what = 'character')**
- 1: mon
- 2: tue
- 3: wed thu
- 5:

- > **d3**
- [1] "mon" "tue" "wed" "thu"
- > **d3[2]**
- [1] "tue"
-
- > **d3[2]='mon'**
-
- > **d3**
- [1] "mon" "mon" "wed" "thu"

- > d3[6]='sat'
-
- > d3
- [1] "mon" "mon" "wed" "thu" NA "sat"
-
- > d3[2]='tue'
-
- > d3[5] = 'fri'
-
- > d3
- [1] "mon" "tue" "wed" "thu" "fri" "sat"

Concept of working directory

- >getwd()

- [1] "C:\Users\DA\R\Database"

- > setwd('D:\Data Analytics\Project\Database)

- > dir()   ## working directory listing

- >ls()   ## Workspace listing of objects

- >rm('object')   ## Remove an element "object", if exist

- > rm(list = ls())   ## Cleaning

Reading data from a data file

- > setwd("D:/Desktop/work") #Set the working directory to file location
- > getwd()
- [1] " D:/Desktop/work"
- > dir()
- rm(list=ls(all=TRUE)) # Refresh session
- > data=read.csv('iris.csv', header = T, sep=",")
- (data = read.table('iris.csv', header = T, sep = ','))
- > ls()
- [1] "data"
- > str(data)
- 'data.frame':   149 obs. of 5 variables:
-   X5.1   : num 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 5.4 ...
-   X3.5   : num 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 ...
-   X1.4   : num 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 ...
-   X0.2   : num 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 0.2 ...
-   Iris.setosa: Factor w/ 3 levels "Iris setosa",..: 1 1 1 1 1 1 1 1 1 1 ...

Accessing elements from a file

- > dataX5.1
- [1] 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7
- > dataX5.1[7]=5.2
- > dataX5.1
- [1] 4.9 4.7 4.6 5.0 5.4 4.6 5.2 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7

#Note: This change has happened in workspace only not in the file.

- How to make it permanent?
- write.csv / write.table
- >write.table(data, file ='iris_mod.csv', row.names = FALSE, sep = ',')
- If row.names is TRUE, R adds one ID column in the beginning of file.
- So its suggested to use row.names = FALSE option
- >write.csv(data, file =='iris_mod.csv', row.names = TRUE)   ## to test

Different data items in R

- **Vector**

- **Matrix**

- **Data Frame**

- **List**

Vectors in R

- The most important family of data types in base R: **vectors.**
- Vectors come in two flavors: atomic vectors and lists.
- They differ in terms of their elements' types:
  - ❑ for atomic vectors, all elements must have the same type
  - ❑ for lists, elements can have different types.

Atomic vectors

- There are four primary types of atomic vectors:
  - **logical**
  - **integer**
  - **double**
  - **character** (which contains strings).
- Collectively integer and double vectors are known as **numeric vectors**
- Each of the four primary types has a special syntax to create an individual value, also known as, a **scalar**:
  - Logicals can be written in full (TRUE or FALSE), or abbreviated (T or F).
  - Doubles can be specified in decimal (0.1234), scientific (1.23 e^4) form. There are three special values unique to doubles: Inf, Inf, and NaN (not a number). These are special values defined by the floating point standard.
- Strings are surrounded by " ("hi") or ' ('bye'). Special characters are escaped with |.
- Integers are written similarly to doubles but must be followed by `L` (`1234L`, `1e4L`, or `0xcafeL`), and can not contain fractional values

Handling numeric type variables

What we will cover:

- Manage the numeric type (integer vs. double)
- Generating non random numbers
- Generating random numbers
- Setting seed values for reproducible random number generation
- Comparing numbers
- Rounding numeric values
- Two most common numeric classes used in R: integer and double (for double precision floating point numbers).
- R automatically converts between these two classes when needed for mathematical purposes.
- Creating a numeric vector using the "*combine*" c() function will produce a vector of double precision numeric values.
- To create a vector of integers using c() we must specify explicitly by placing an L directly after each number.

Vectors in R

```
x = c(1,2,3,4,56)
x
```
[1] 1 2 3 4 56

 # Creating double and integer vectors

```
dbl_var <  c(1, 2.5, 4.5)  # double precision values
dbl_var
```
[1] 1.0 2.5 4.5

```
int_var <  c(1L, 6L, 10L)  # Integer vectors
int_var
```
[1] 1 6 10

 # Checking for numeric type

```
typeof(dbl_var)
```
[1] "double"

```
typeof(int_var)
```
[1] "integer"

Converting between types in R

- By default, if we read in data that has no decimal points or you create numeric values using the x=1: 10 method the numeric values will be coded as integer.
- If you want to change a double to an integer or vice versa you can specify one of the following:

   as.double or as.numeric or as.integer

```
# converts integers to double precision values
as.double(int_var)
## [1]  1  6 10


# identical to as.double()
as.numeric(int_var)
## [1]  1  6 10


# converts doubles to integers
as.integer(dbl_var)
## [1] 1 2 4
```

More on Vectors in R

- >y = c(x,c(1,5),x)
- >length(x)
- >length(y)

There are useful methods to create long vectors whose elements are in arithmetic progression:

- > x=1:20

- > x

If the common difference is not 1 or 1 then we can use the seq function

- > y=seq(2,5,0.3)
- > y
- [1] 2.0 2.3 2.6 2.9 3.2 3.5 3.8 4.1 4.4 4.7 5.0
- > length(y)
- [1] 11

More on Vectors in R

- > x=1:5
- > mean(x)
- [1] 3
- > x
- [1] 1 2 3 4 5
- > x^2
- [1] 1 4 9 16 25
- > x+1
- [1] 2 3 4 5 6
- > 2*x
- [1] 2 4 6 8 10
- > exp(sqrt(x))
- [1] 2.718282 4.113250 5.652234 7.389056 9.356469

- It is very easy to add/subtract/multiply/divide two vectors entry by entry.
- > y=c(0,3,4,0)
- > x+y
- [1] 1 5 7 4 5
- > y=c(0,3,4,0,9)
- > x+y
- [1] 1 5 7 4 14
- Warning message:
- In x + y : longer object length is not a multiple of shorter object length
- > x=1:6
- > y=c(9,8)
- > x+y
- [1] 10 10 12 12 14 14

Logical values in R

- R allows manipulation of logical quantities.
- The elements of a logical vector can have the values TRUE, FALSE, and NA (for "not available").
- The first two are often abbreviated as T and F, respectively.

! : Logical NOT # & : Element-wise logical AND # && : Logical AND # | : Element-wise logical OR # || : Logical OR

Operators & and | perform element-wise operation producing result having length of the longer operand.

But && and || examines only the first element of the operands resulting into a single length logical vector.

Zero is considered FALSE and non-zero numbers are taken as TRUE. An example run.

```
x2 <- c(TRUE,FALSE,TRUE)              y2 <- c(FALSE,TRUE,TRUE)
print(!x2)      # [1] FALSE  TRUE FALSE      print(x2&y2)   # [1] FALSE FALSE  TRUE
print(x2&&y2) # [1] FALSE                    print(x2|y2)    # [1] TRUE TRUE TRUE
print(x2||y2)  # [1] TRUE
```

Relational Operators / Comparing numbers in R

- **x<y** is x less than y
- **x>y** is x greater than y
- **x<=y** is x less than or equal to y
- **x>=y** is x greater than or equal to y
- **x==y** is x equal to y
- **x !=y** is x not equal to

Note: = operator is used for assignment and == for comparison

```
x <- 9
y <- 10
x == y
## [1] FALSE


x <- c(1, 4, 9, 12)
y <- c(4, 4, 9, 13)
x == y
## [1] FALSE  TRUE  TRUE FALSE


x <- c(1, 4, 9, 12)
y <- c(4, 4, 9, 13)
# How many pairwise equal values are in vectors x and y
sum(x == y)
## [1] 2


# Where are the pairwise equal values located in vectors x and y
which(x == y)
## [1] 2 3
```

Generating random numbers in R

- Simulation is often required in data analysis.

- Sometimes our analysis requires the implementation of a statistical procedure that requires random number generation or sampling (i.e. Monte Carlo simulation, bootstrap sampling, etc).

- R comes with a set of pseudo-random number generators that allow you to simulate the most common probability distributions such as Binomial, Poisson, Normal, Uniform, Exponential, etc.

Generating random numbers in R

 # simple random sampling with replacement

x <- c(1:10)

sample(x, size = 5, replace = TRUE)

[1] 1 1 8 4 3

 # simple random sampling without replacement

x <- c(1:10)

sample(x, size = 5, replace = FALSE)

[1] 4 2 10 3 9

Generating Random numbers: Discrete and continuous distributions in R

 # random sample of size 10 from a Bin(4, 0.8) distribution

rbinom(n = 10, size = 4, prob = 0.8)

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

 # random sample of size 10 from a Poisson(2) distribution

rpois(n = 10, lambda = 2)

[1] 3 3 5 3 0 3 2 1 2 0

 # random sample of size 5 from a Uniform(2,4) distribution

runif(n = 5, min = 2, max = 4)

[1] 2.147157 3.766859 3.293724 3.071439 3.903576

 # random sample of size 10 from a Normal(2,4) distribution

rnorm(10, mean = 2, sd = 2) # note the parameter spec. here

[1] 2.0745291 0.1719257 2.3249074 3.2447822 1.4329557 0.1173869 0.5231311 1.4754708
1.7591520 -1.8763400

Random numbers are changing?
Let's set the seed!

```
rnorm(n = 5, mean = 1, sd = 2)
## [1]  1.7727385 -3.6486144  0.9951339  2.5762444 -0.8415272


rnorm(n = 5, mean = 1, sd = 2)
## [1]  1.22932369 -1.98687014  0.59244070  2.38830593 -0.01179321


set.seed(100)
rnorm(n = 5, mean = 1, sd = 2)
## [1] -0.004384701  1.263062331  0.842165820  2.773569619  1.233942541


set.seed(100)
rnorm(n = 5, mean = 1, sd = 2)
## [1] -0.004384701  1.263062331  0.842165820  2.773569619  1.233942541
```

Rounding in R

```
x <- runif(5, 2, 4)
x
## [1] 3.249993 3.764331 2.560708 2.796976 3.525102

# round to nearest integer
round(x)
## [1] 3 4 3 3 4

# round to 3 places of decimal
round(x,3)
## [1] 3.250 3.764 2.561 2.797 3.525

x
## [1] 3.249993 3.764331 2.560708 2.796976 3.525102

floor(x) # round down
## [1] 3 3 2 2 3

ceiling(x) # round up
## [1] 4 4 3 3 4
```

Character string in R

Use quotation marks and assign a string to an object similar to creating number sequences.

a <- "This course is on"   **# string a**

b <- "Multivariate Data Analysis"   **# string b**

 # paste together string a & b

paste(a, b)

[1] "This course is on Multivariate Data Analysis"

 # paste character and number strings (converts numbers to character class)

paste("I want to eat a", pi)

[1] "I want to eat a 3.14159265358979"

 # paste multiple strings

paste("Multivariate", " Data Analysis", "with", "R")

[1] "Multivariate Data Analysis with R"

 # paste multiple strings with a separating character

paste("Data analysis", "applications","with", "R", sep = "-")

[1] "Data analysis-applications-with-R"

 # use pasteo() to paste without spaces btwn characters

pasteo("Data analysis", "applications", "with", "R")

[1] "Data analysisapplicationswithR"

 # paste objects with different lengths

paste("R", 1:5, sep = " v1.")

[1] "R v1.1" "R v1.2" "R v1.3" "R v1.4" "R v1.5"

Coercion in atomic vectors in R

- For atomic vectors, type is a property of the entire vector:
  - ❑ all elements must be the same type.
- When one attempts to combine different types they will be coerced in a fixed order:
  - ❑ character -> double -> integer -> logical.

For example, combining a character and an integer yields a character:

```
c("A",10)
## [1] "A"  "10"
```

```
typeof(c("A",10))
## [1] "character"
```

```
# Combining logical and numeric creates numeric
x <- c(10, 57, 3, 90)
x > 20
## [1] FALSE  TRUE FALSE  TRUE
```

```
sum(x > 20) # numeric function applied on logical
## [1] 2
```

Subsetting vectors in R

There are four primary ways to subset a vector. All of them include combining square brackets [ ] with the following:
- Positive integers
- Negative integers
- Logical values
- Names

```
x_1 <- c(1:10)
x_2 <- 4
x_3 <- c(101:110)
x_4 <- c(2,3)
x_5 <- c(1,2,3)


x_1
## [1] 1 2 3 4 5 6 7 8 9 10


x_1[4]
#get the fourth element of vector x
## [1] 4
```

```
x_1[c(2,7)]
# get elements in position 2 and 7
## [1] 2 7


x_1[-3]
# everything except element in position 3
## [1] 1 2 4 5 6 7 8 9 10


x_1[c(-1,-3)]
# everything except elements in pos. 1 & 3
## [1] 2 4 5 6 7 8 9 10
```

Subsetting vectors in R

```
names(x_1) <- letters[1: length(x_1)] # assign
names to x
x_1
## a b c d e f g h i j
## 1 2 3 4 5 6 7 8 9 10

x_1[c("b","j")] # get elements named "b" and "j"
## b j
## 2 10

x_1[x_1 > 6]
## g h i j
## 7 8 9 10
```

```
x_1[4]
#preserves (here, keeps the name)
## d
## 4

x_1[[4]]
# simplifies (here, name is removed)
## [1] 4
```

Matrices in R

- A matrix is a collection of data elements arranged in a two-dimensional rectangular layout.
- A matrix is of homogeneous type, that is, all elements must be of the same type. Hence, a matrix may be thought of as an atomic vector.
- All rows of a matrix must have the same length.

Creating matrices

- Matrices are constructed column-wise.
- We can create a matrix using the matrix() function and specifying the values to fill in the matrix and the number of rows and columns to make the matrix.

Matrices in R

- Same data type/mode – number , character, logical
- a.matrix < matrix(vector, nrow = r, ncol = c, byrow = FALSE, dimnames = list(char vector rownames, char vector col names))

dimnames is optional argument, provides labels for rows & columns.
y < matrix(1:20, nrow = 4, ncol = 5)
A = matrix(c(1,2,3,4),nrow=2,byrow=T)
A
A = matrix(c(1,2,3,4),ncol=2)
B = matrix(2:7,nrow=2)
C = matrix(5:2,ncol=2)
mr < matrix(1:20, nrow = 5, ncol = 4, byrow = T)
mc < matrix(1:20, nrow = 5, ncol = 4)
mr
mc

Arithmetic Operations on matrices in R

- Recall that matrices behave like atomic vectors, and hence arithmetical operations are similar.

- Consider two matrices A and B.

```
A <- matrix(c(1:4), 2, 2)
B <- matrix(c(11:14), 2, 2)

A + B
# elementwise addition
##        [,1] [,2]
## [1,]   12   16
## [2,]   14   18
```

```
A * B
# elementwise multiplication
##        [,1] [,2]
## [1,]   11   39
## [2,]   24   56

A%*%B
#Matrix multiplication.
##        [,1] [,2]
##[1,]   47   55
##[2,]   70   82
```

More on matrices in R

dim(B)

#Dimension

[1] 2 2

nrow(B)

[1] 2

ncol(B)

[1] 2

t(A)

#Transpose

[,1] [,2]

##[1,]   1   2

##[2,]   3   4

A[1,2]

 # element in row 1, column 2

[1] 3

A[2,]

 # row 2

[1] 2 4

A[,1]

 # column 1

[1] 1 2

B[,-1]

 # all except column 1

[1] 13 14

rownames(A) = c("a","b")
colnames(A) = c("d", "e")

A

d e

a 1 3

b 2 4

Lists in R

- Vectors and matrices in R are two ways to work with a collection of objects.

- Lists provide a third method. Unlike a vector or a matrix a list can hold different kinds of objects (heterogenous).

- One entry in a list may be a number, while the next is a matrix, while a third is a character string (like "Hello R!").

- Statistical functions of R usually return the result in the form of lists. So we must know how to unpack a list using the symbol *list()*.

Examples of lists in R

```
x = list(name = 'Tanujit', nationality =
'Indian' , height =5.5 , marks
=c(95,45,80))
names(x)

## [1] "name" "nationality" "height"
"marks"

str(x)
## List of 4
## $ name      : chr "Tanujit"
## $ nationality: chr "Indian"
## $ height     : num 5.5
## $ marks      : num [1:3] 95 45 80
```

```
x$name
## [1] "Tanujit"

x$hei   #abbreviations are OK
## [1] 5.5

x$marks
## [1] 95 45 80

x$m[2]
## [1] 45

typeof(x)
## [1] "list"
```

Adding lists in R

We can add new elements to an existing list with list(). However, that will create a list of two components, component 1 will be a nested list of the original list and component 2 will be the new elements added.

```
l1 <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
l2 <- list(l1, c(1:10))
str(l2)

## List of 2
## $ :List of 4
##   ..$ : int [1:3] 1 2 3
##   ..$ : chr "a"
##   ..$ : logi [1:3] TRUE FALSE TRUE
##   ..$ : num [1:2] 2.3 5.9
## $ : int [1:10] 1 2 3 4 5 6 7 8 9 10

#To add an additional list component without
creating nested lists we use the append() function.
l3 <- append(l1, list(c(1:10)))
str(l3)
```

```
## List of 5
## $ : int [1:3] 1 2 3
## $ : chr "a"
## $ : logi [1:3] TRUE FALSE TRUE
## $ : num [1:2] 2.3 5.9
## $ : int [1:10] 1 2 3 4 5 6 7 8 9 10

# Alternatively, we can add additional elements by
using the $ sign and naming the item.

l1$x <- c(1.4,2.3)
str(l1)
## List of 5
## $  : int [1:3] 1 2 3
## $  : chr "a"
## $  : logi [1:3] TRUE FALSE TRUE
## $  : num [1:2] 2.3 5.9
## $ x: num [1:2] 1.4 2.3
```

Data frame in R

- A data frame is a list of equal-length vectors.

- Data frames are heterogeneous, hence can contain different data types.

- Data frames have a rectangular structure and explains why they share the properties of both matrices and lists.

- We can use rbind() and cbind() functions to append data to an existing data frame.

- Columns and rows can be named using the rownames and colnames functions.

- Creating data frames in R is done using data.frame() function.

Data frame in R

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.).

- >d < c(1,2,3,4)
- >e < c("red", "white", "red", NA)
- >f < c(TRUE,TRUE,TRUE,FALSE)
- >myframe < data.frame(d,e,f)
- >names(myframe) < c("ID","Color","Passed") # Variable names
- >myframe
- >myframe[1:3,]   # Rows 1 , 2, 3 of data frame
- >myframe[,1:2]   # Col 1, 2 of data frame
- >myframe[c("ID","Color")] #Columns ID and color from data frame
- >myframeID   # Variable ID in the data frame

Factors in R

- In R we can make a variable is nominal by making it a factor.

- Sometimes we need to deal with factor level variables in our data analysis, especially for categorical data.

- The factor stores the nominal values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable).

- An internal vector of character strings (the original values) mapped to these integers.

Factors in R

```
section <- rep(c("A","B"), each = 3)
section
## [1] "A" "A" "A" "B" "B" "B"

typeof(section)
## [1] "character"

section <- as.factor(section)
section
## [1] A A A B B B
## Levels: A B

class(section)
## [1] "factor"
```

```
gender <- c(rep("male",20), rep("female", 30))
gender <- factor(gender)

# Stores gender as 20 1s and 30 2s
# 1=male, 2=female internally (alphabetically)
# R now treats gender as a nominal variable

summary(gender)
## [1] female   male
##          30    20
```

BASIC TASKS

Matrix multiplication – Code

Read the matrix A and B

A = matrix(c(21,57,89,31,7,98), nrow =2, ncol=3, byrow = TRUE)

B = matrix(c(24, 35, 15, 34, 56,25), nrow = 3, ncol = 2, byrow = TRUE)

Multiplication of matrices

C = A%*%B

C

Determinant – R Code

A = matrix(c(51, 10, 23, 64), nrow = 2, ncol =2, byrow =TRUE)

det(A)

Matrix Inverse – R code

A = matrix(c(51, 10, 23, 64), nrow = 2, ncol =2, byrow =TRUE)

solve(A)

BASIC TASKS

Eigen values and Eigen vectors – R Code

A = matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2, byrow = TRUE)

eigen(A)

Generating 5 Random Numbers – R Code

x = rnorm(5, mean = 0, sd = 1)

x

Functional Help

?rnorm()

Package Installation

install.packages("ggplot")

Library Call (for use)

library(ggplot)

USEFUL FUNCTIONS IN R

FUNCTIONS AND CONDITIONAL EXECUTIONS

What we will learn in this session

- What functions are and when should we use functions
- Fundamentals of writing and executing functions in R
- Function rules
- Basics of conditional executions
- Functions in conditional executions

FUNCTIONS IN R

- R is a functional programming language, that is, everything we execute is based on functions

- For example, the *mean()* function is a pre-defined function in base R which returns the mean of a certain input

- There are plenty of pre-built functions in R and associated packages

- However, depending on our work to be done, many a time we shall need to write our own functions

- Functions allow us to reduce code duplication by automating a generalized task to be applied recursively.

FUNDAMENTALS OF FUNCTIONS

To understand functions in R you need to internalize two important ideas:

- Functions can be broken down into three components: arguments, body, and environment.
- Functions are objects, just as vectors are objects.

Functions in R


```
f = function(x) x/(1-x)
```
name of the function — f
argument — x
body of function — x/(1-x)

- >g = function(x,y) (x+2*y)/3
- >g(1,2)
- >g(2,1)

FUNCTION COMPONENTS

A function has three parts:

- The *formals ()*, the list of arguments that control how you call the function.
- The *body ()*, the code inside the function.
- The *environment()*, the data structure that determines how the function finds the values associated with the names.

○ While the *formals* and *body* are specified explicitly when you create a function, the *environment* is specified implicitly, based on where you defined the function.

○ The function environment always exists, but it is only printed when the function isn't defined in the global environment.

FUNCTION COMPONENTS: EXAMPLE

Suppose we need to write a function to calculate the total accumulated value of an investment with periodic compounding. The equation for the accumulated value is given by

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

where, A: total accumulated value, P: principal investment amount, r: nominal annual interest rate, n : compounding frequency, t: overall time period

FUNCTION COMPONENTS

In R we can define a function to calculate the accumulated sum as follows:

```
compound_amount <- function(P, r, n, t){
  A <- P*(1 + r/n)^(n*t)
  return(A)
}
```

Suppose someone invested a sum of 10000 INR at an annual interest rate of 6.5%, compounded quarterly for a period of 6 years. To calculate the accumulated sum, we can use

```
P <- 10000; r <- 0.065; n <- 4; t <- 6
compound_amount(P, r, n, t)
## [1] 14723.58
```

FUNCTION COMPONENTS

As mentioned earlier, formals() specifies the list of arguments that control how you call the function.

formals(compound_amount)

$P

$r

$n

$t

body(compound_amount)

{

A <- P * (1 + r/n)^(n * t)

return(A)

}

environment(compound_amount)

<environment: R_GlobalEnv>

FUNCTION ARGUMENTS

We can call the arguments in several ways

using argument names

compound_amount(P = 10000, r = 0.065, n = 4, t = 6)

[1] 14723.58

same as above but without using names ("positional matching")

compound_amount(10000, .065, 4, 6)

[1] 14723.58

if using names you can change the order

compound_amount(r = .065, P = 10000, n = 4, t = 6)

[1] 14723.58

if not using names you must insert arguments in proper order

compound_amount(.08, 10000, 4, 6)

[1] 2.869582e+80

FUNCTION ARGUMENTS

We can call the arguments in several ways

While writing a function we can also set default values to arguments.

compound_amount <- function(P, r = 0.065, n = 4, t){

  A <- P*(1 + r/n)^(n*t)

  return(A)

}

compound_amount(P = 10000, t = 6) # uses default values of r and n

[1] 14723.58

compound_amount(P = 10000, r = 0.07, t = 6) # uses specified value of r, default for n

[1] 15164.43

LEXICAL SCOPING

What is the output of this code?

```
x <- 10
g01 <- function() {
x <- 20
x
}
g01()
```

- R uses lexical scoping
- This means that a function will first look inside the function to identify all the variables being called.
- If all variables exist then their is no additional search required to identify variables.

LEXICAL SCOPING

P <- 1

compound_amount <- function(P, r, n, t){

  P <- 10000; r <- 0.065; n <- 4; t <- 6

  A <- P*(1 + r/n)^(n*t)

  return(A)

}

compound_amount(P, r, n, t)

[1] 14723.58

#If a variable does not exist within the function,

R will look one level up to see if the variable exists.

P <- 10000

compound_amount <- function(P, r, n, t){

  r <- 0.065; n <- 4; t <- 6

  A <- P*(1 + r/n)^(n*t)

  return(A)

}

compound_amount(P, r, n, t)

[1] 14723.58

#The same rule applies for functions within functions

P <- 100

compound_amount <- function(P){

  r <- 0.065; n <- 4; t <- 6

  interest_factor <- function(){

    out <- (1 + r/n)^(n*t)

  }

  A <- P*interest_factor()

  return(A)

}

compound_amount(P)

[1] 147.2358

LEXICAL SCOPING

This also applies for functions in which some arguments are called but not all variables used in the body are identified as arguments.

r <- 0.07

compound_amount <- function(P, t){

  n <- 4

  A <- P*(1 + r/n)^(n*t)

  return(A)

}

n is defined within the function

r is defined outside the function

compound_amount(P = 10000, t = 6)

[1] 15164.43

FUNCTIONS VS VARIABLES

- In R, functions are ordinary objects.
- This means the scoping rules described above also apply to functions.

```
cool_func <- function(x) x + 10
ubercool_func <- function() {
 cool_func <- function(x) x + 99
 cool_func(10)
}
ubercool_func()
```

[1] 109

DYNAMIC LOOKUP

- Lexical scoping determines where, but not when to look for values.
- R looks for values when the function is run, not when the function is created.
- Together, these two properties tell us that the output of a function can differ depending on the objects outside the function's environment.

foo <- function() x + 1

x <- 25

foo()

[1] 26

x <- 50

foo()

[1] 51

LAZY EVALUATION

In R, function arguments are lazily evaluated: they're only evaluated if accessed.

For example, this code doesn't generate an error because x is never used.

```
h01 <- function(x) {

  10

}
h01(stop("This is an error!"))
## [1] 10
```

This is an important feature because it allows you to do things like include potentially expensive computations in function arguments that will only be evaluated if needed.

```
# the y argument is not used so not included it causes no harm
lazy_func <- function(x, y){

  x^2

}
lazy_func(2)
## [1] 4
```

LAZY EVALUATION

if both arguments are required in the body an error will result if an argument is missing

```r
lazy_func2 <- function(x, y){
  (x + y)^2
}
lazy_func2(2)
```

Error in lazy_func2(2): argument "y" is missing, with no default

RETURNING MULTIPLE OUTPUTS

- If a function performs multiple tasks and therefore has multiple results to report then we have to include the c() function inside the function to display all the results.

- If you do not include the c() function then the function output will only return the last expression

- When we have a function which performs multiple tasks (i.e. computes multiple computations) then it is often useful to save the results in a list.

```
bad_func <- function(x, y) {
  2*x + y
  x + 2*y
  2*x + 2*y
  x/y
}
bad_func(2,3)
## [1] 0.6666667
```

RETURNING MULTIPLE OUTPUTS

```
good_func <- function(x, y) {
  output1 <- 2*x + y
  output2 <- x + 2*y
  output3 <- 2*x + 2*y
  output4 <- x/y
  c(output1, output2, output3, output4)
}
good_func(2, 3)
```

[1] 7.0000000 8.0000000 10.0000000 0.6666667

```
good_list <- function(x, y) {
  output1 <- 2*x + y
  output2 <- x + 2*y
  output3 <- 2*x + 2*y
  output4 <- x/y
  c(list(Output1 = output1, Output2 = output2,
      Output3 = output3, Output4 = output4))
}
```

RETURNING MULTIPLE OUTPUTS

good_list(1, 2)

$Output1

[1] 4

$Output2

[1] 5

$Output3

[1] 6

$Output4

[1] 0.5

CONDITIONAL EXECUTION

o Multistep processes can be organized in sequences of R expressions, using loops

o There are several loop control statements in R that allow us to perform repetitive code processes.

o Sometimes looping is a painstakingly slow process, that can be avoided by using 'vectorized' expressions (we shall discuss this later)

o But often looping turns out to be the only solution

o These include the following:

- if statement for conditional programming

- if...else statement for conditional programming

- for loop to iterate over a fixed number of iterations

- while loop to iterate until a logical statement returns FALSE

- repeat loop to execute until told to break

- break/next arguments to exit and skip iterations in a loop

IF STATEMENT

- The conditional if statement is used to test an expression.
- If the test_expression is TRUE, the statement gets executed.
- But if it's FALSE, nothing happens.

```
# syntax of if statement
if (test_expression) {
  statement
}


x <- 2
if(x < 3){
  y <- x + 2
  y
}
## [1] 4
```

```
# second example
coin <- rbinom(1,1,0.5) # tossing a fair coin once
toss_call <- "Heads"
coin # toss outcome
## [1] 1

if(coin == 1){
  print("Heads it is!")
}
## [1] "Heads it is!"
```

IF…ELSE STATEMENT

- The conditional if...else statement is used to test an expression similar to the if statement.

- However, rather than nothing happening if the test_expression is FALSE, the else part of the function will be evaluated.

```
# syntax of if...else statement
if (test_expression) {
  # statement 1
} else {
  # statement 2
}
```

```
if(coin == 1){
  print("Heads it is!")
} else {
  print("No, it's Tails!")
}
## [1] "Heads it is!"
```

```
coin <- rbinom(1,1,0.5) # tossing a fair coin once
toss_call <- "Heads"
coin # toss outcome
## [1] 1
```

```
set.seed(100)
coin <- rbinom(1,1,0.5) # tossing a fair coin once
toss_call <- "Tails"
coin # toss outcome
## [1] 1
```

IF…ELSE STATEMENT

```
if(coin == 1 & toss_call == "Heads"){
  print("Heads it is!")
} else if(coin == 1 & toss_call == "Tails"){
  print("No, it's Heads!")
} else if(coin == 0 & toss_call == "Heads"){
  print("No it's Tails!")
} else if(coin == 0 & toss_call == "Tails"){
  print("Tails it is!")
}
## [1] "Heads it is!"
```

FOR LOOP

- The for loop is used to execute repetitive code statements for a particular number of times.

- The general syntax is provided below where *i* is the counter and as *i* assumes each sequential value defined (1 through 100 in this example) the code in the body will be performed for that i^th value.

```
# syntax of for loop
for(i in 1:100) {

  <do stuff here with i>

}
```

```
# squaring elements in a vector
x <- seq(from = 0, to = 10, by = 0.5)
xsq <- 0 #initialization


for(i in 1:length(x)){
  xsq[i] <- x[i]*x[i]
}
```

```
xsq
## [1]   0.00   0.25   1.00   2.25   4.00   6.25
9.00  12.25  16.00  20.25
## [11]  25.00  30.25  36.00  42.25  49.00  56.25
64.00  72.25  81.00  90.25
## [21] 100.00
```

FOR LOOP

#Note: This was just for demonstration. We could have just done this:

```r
xsq <- x^2

xsq
```

```
## [1]   0.00   0.25   1.00   2.25   4.00   6.25
9.00  12.25  16.00  20.25
## [11]  25.00  30.25  36.00  42.25  49.00  56.25
64.00  72.25  81.00  90.25
## [21] 100.00
```

```r
for (i in 2023:2029){
  output <- paste("Good times shall come in ",
i) # the "promise"
  print(output)
}
```

```
## [1] "Good times shall come in  2023"
## [1] "Good times shall come in  2024"
## [1] "Good times shall come in  2025"
## [1] "Good times shall come in  2026"
## [1] "Good times shall come in  2027"
## [1] "Good times shall come in  2028"
## [1] "Good times shall come in  2029"
```

```r
print("Good times are yet to come!") # the "reality"
```

```
## [1] "Good times are yet to come!"
```

WHILE LOOP

- while loops begin by testing a condition.
- If it is true, then they execute the statement.
- Once the statement is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.
- It's considered a best practice to include a counter object to keep track of total iterations.

```
# syntax of while loop
counter <- 1
while(test_expression) {
  statement
  counter <- counter + 1
}
```

The primary difference between a for loop and a while loop is:
 a for loop is used when the number of iterations a code should be run is known where as a while loop is used when the number of iterations is not known.

WHILE LOOP

```
wins <- 0; games <- 0
while(wins < 3) {
  x <- rbinom(1,1,0.5)
  wins <- wins + x
  games <- games + 1
  print(c(toss = x, games = games, wins = wins))
}
```

```
##  toss games  wins
##    1    1    1
##  toss games  wins
##    0    2    1
##  toss games  wins
##    0    3    1
##  toss games  wins
##    1    4    2
##  toss games  wins
##    1    5    3
```

REPEAT STATEMENT

- A repeat loop is used to iterate over a block of code multiple number of times.
- There is test expression in a repeat loop to end or exit the loop.
- We must put a condition statement explicitly inside the body of the loop and use the break function to exit the loop. Failing to do so will result into an infinite loop.

```
# We monitor a captain's toss winning record, and count how many games are required
to win at least 3 tosses

games <- 0 # games counter
wins <- 0 # toss wins counter

repeat {
  games <- games + 1 # new game
  x <- rbinom(1,1,0.5) # toss outcome
  wins <- wins + x # wins update
  if(wins == 3){
    break
  }
}

c(wins, games) # toss wins and number of games
## [1] 3 6
```

BREAK STATEMENT

- The break argument is used to exit a loop immediately, regardless of what iteration the loop may be on.
- break arguments are typically embedded in an if statement in which a condition is assessed, if TRUE break out of the loop, if FALSE continue on with the loop.
- In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

Example: Suppose we want to create a 10 x 10 lower-triangular matrix with elements given by the product of the row and column numbers.
This means that all matrix elements for which row number <= column number must be zero.

use of breaks

m <- n <- 10 # matrix dimensions
# Create a 10 x 10 matrix with zeroes
mymat <- matrix(0, nrow = m, ncol = n)
ctr <- 0 # counter to count the assignment

BREAK STATEMENT

```
for(i in 1:m) {
  for(j in 1:n) {
    if(i == j) {
      break
    } else {
      mymat[i,j] <- i*j # assign the values only when i > j
      ctr <- ctr+1
    }
  }
}

# Print how many matrix cells were assigned
print(ctr)
## [1] 45
```

DESCRIPTIVE STATISTICS USING R

DESCRIPTIVE STATISTICS

Exercise 1: The monthly credit card expenses of an individual in 1000 rupees is given in the file Credit_Card_Expenses.csv.

a.Read the dataset to R studio

b.Compute mean, median minimum, maximum, range, variance, standard deviation, skewness, kurtosis and quantiles of Credit Card Expenses

c.Compute default summary of Credit Card Expenses

d.Draw Histogram of Credit Card Expenses

DESCRIPTIVE STATISTICS

The monthly credit card expenses of an individual in 1000 rupees is given below. Kindly summarize the data

Month	Credit Card Expenses	Month	Credit Card Expenses
1	55	11	63
2	65	12	55
3	59	13	61
4	59	14	61
5	57	15	57
6	61	16	59
7	53	17	61
8	63	18	57
9	59	19	59
10	57	20	63

Reading a csv file to R Studio

The file open dialog box will pop up

Browse to the file

Reading a csv file to R Studio

Click Import button
R studio will read the data set to a data frame with specified name

DESCRIPTIVE STATISTICS

Reading a csv file to R Studio : Source code

➢Credit_Card_Expenses < read.csv("C:/Desktop/Data/Credit_Card_Expenses.csv")

To change the name of the data set to : mydata

> mydata = Credit_Card_Expenses

To display the contents of the data set

> print(mydata)

To read a particular column or variable of data set to a ne variable Example: Read

CC_Expenses to CC

>CC = mydataCC_Expenses

DESCRIPTIVE STATISTICS

Reading data from MS Excel formats to R Studio

Format	Code
Excel	library(xlsx) mydata < read.xlsx("c:/myexcel.xlsx", "Sheet1")

Reading data from databases to R Studio

Function	Description
odbcConnect(*dsn*, uid="", pwd="")	Open a connection to an ODBC database
sqlFetch(*channel*, *sqtable*)	Read a table from an ODBC database into a data frame
sqlQuery(*channel*, *query*)	Submit a query to an ODBC database and return the results
sqlSave(*channel*, *mydf*, tablename = *sqtable*, append = *FALSE*)	Write or update (append=True) a data frame to a table in the ODBC database
sqlDrop(*channel*, *sqtable*)	Remove a table from the ODBC database
close(*channel*)	Close the connection

DESCRIPTIVE STATISTICS

Operators   Arithmetic

Operator	Description
+	addition
	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2

DESCRIPTIVE STATISTICS

Operators   Logical

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
! =	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

DESCRIPTIVE STATISTICS

Descriptive Statistics

Computation of descriptive statistics for variable CC

Function	Code	Value
Mean	> mean(CC)	59.2
Median	> median(CC)	59
Standard deviation	> sd(CC)	3.105174
Variance	> var(CC)	9.642105
Minimum	> min(CC)	53
Maximum	> max(CC)	65
Range	> range(CC)	53.65

DESCRIPTIVE STATISTICS

Descriptive Statistics

Function	Code
Quantile	> quantile(CC)

Output					
Quantile	0%	25%	50%	75%	100%
Value	53	57	59	61	65

Function	Code
Summary	>summary(CC)

Output					
Minimum	Q1	Median	Mean	Q3	Maximum
53	57	59	59.2	61	65

DESCRIPTIVE STATISTICS

Descriptive Statistics

Function	Code
describe	> library(psych) > describe(CC)

Output	
Statistics	Values
N	20
mean	59.2
sd	3.11
median	59
Trimmed	59.25
mad	2.97
min	53
Max	65
Range	12
Skew	0.08
Kurtosis	0.85
se	0.69

DESCRIPTIVE STATISTICS

Graphs

Graph	Code
Histogram	> hist(CC)
Histogram colour ("Blue")	> hist(CC,col="blue")
Dot plot	> dotchart(CC)
Box plot	> boxplot(CC)
Box plot colour	> boxplot(CC, col="dark green")

DESCRIPTIVE STATISTICS

Histogram : Variable   CC

Histogram of CC
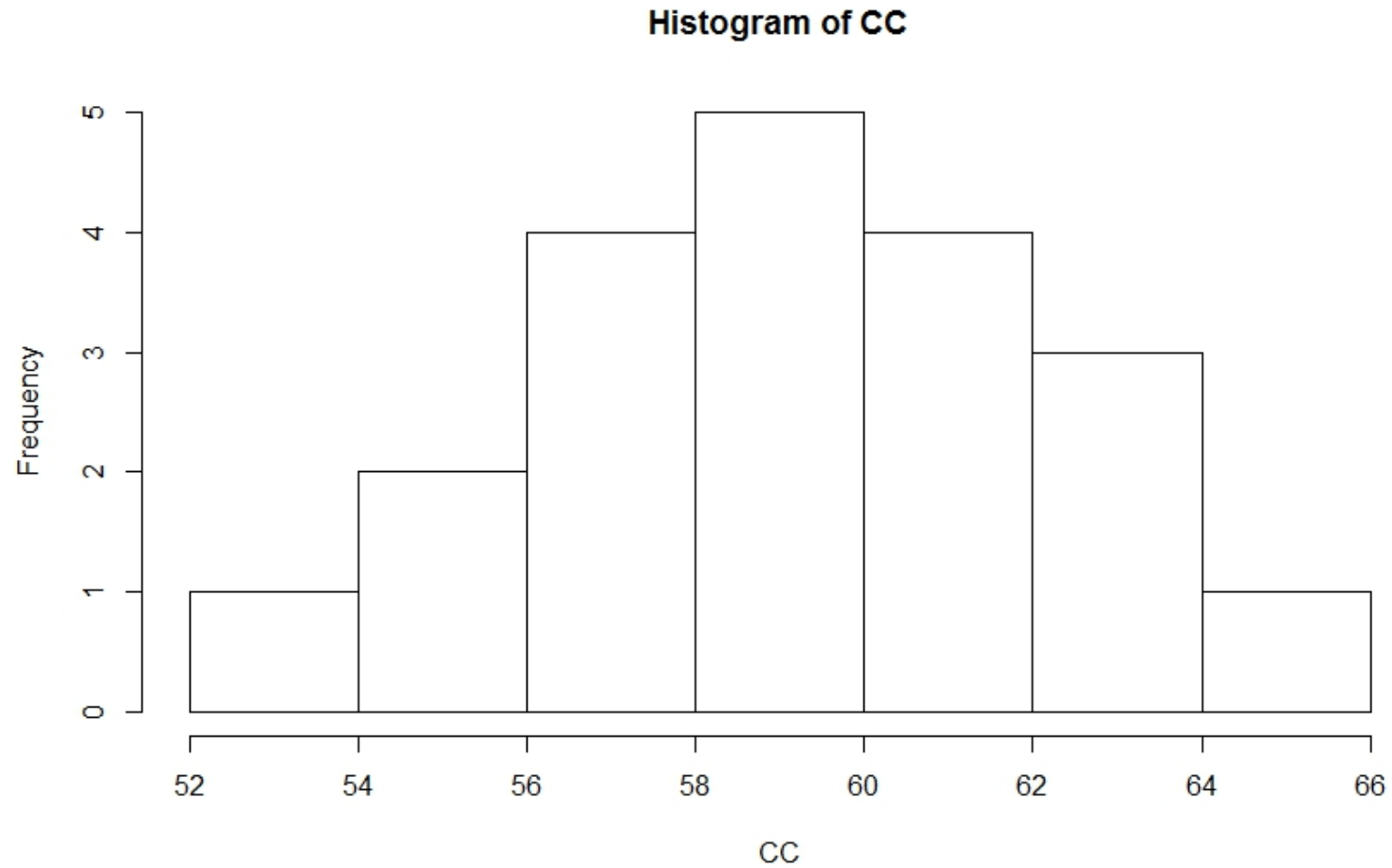
DESCRIPTIVE STATISTICS

Box plot : Variable   CC
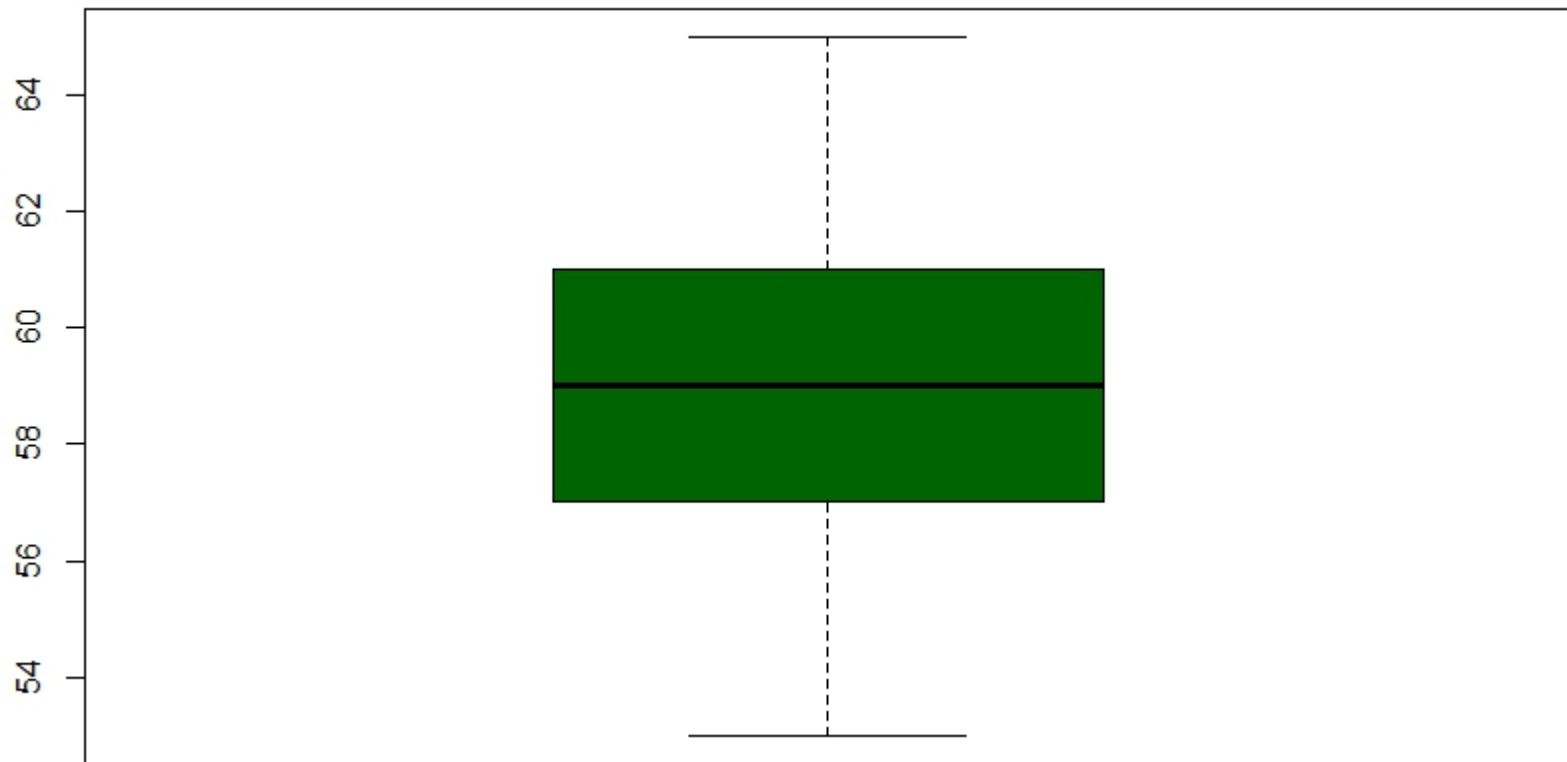
DATA VISUALIZATION USING GGPLOT2

What we will learn in this session?

- Why visualizing data is important for any analysis

- Learn how to visualize data using the ggplot2 package

- Different types of data visualization tools

DATA VISUALIZATION

- Creating visualizations or graphical representations of data is a key step in being able to communicate information and findings to others.

- But improper or bad visualizations can cause harm.

- Need to produce proper and nice visualizations.

GRAMMAR OF GRAPHICS

There are several systems for graphics in R
- One of the most elegant systems is ggplot2
- It implements the 'grammar of graphics'
- Very flexible and versatile
- Helps us to construct graphical figures out of different visual elements.
- This grammar opens up a conversation about parts of a plot: circles, lines, arrows, and words that are combined into a diagram for visualizing data.
- Helps us describing various components of a plot

GRAMMAR OF GRAPHICS

Components of a plot include
- the data!
- geometric objects (dots, circles, lines, etc.) appearing on the plot
- a set of mappings from variables in the data to the aesthetics (appearance) of the geometric objects
- statistical transformations used to calculate the data values used in the plot
- position adjustments for locating each geometric object on the plot
- scales (e.g., range of values) for each aesthetic mapping used
- coordinate system used to organize the geometric objects
- the facets or groups of data shown in different plots
- These components are further organized into layers, where each layer has a single geometric object, statistical transformation, and position adjustment.
- Following this grammar, you can think of each plot as a set of layers of images, where each image's appearance is based on some aspect of the data set.

PACKAGES

- You may install ggplot2 separately, but it is better to install the larger package 'tidyverse'
- We start with installing and loading the "tidyverse" package first

```
install.packages("tidyverse")
# Library Call (for use)
library("tidyverse")

## —— Attaching packages ————————————— tidyverse 1.3.2 ——
## ✓ ggplot2 3.4.0     ✓ purrr   0.3.4
## ✓ tibble  3.1.7     ✓ dplyr   1.0.9
## ✓ tidyr   1.2.0     ✓ stringr 1.4.0
## ✓ readr   2.1.2     ✓ forcats 0.5.1
## —— Conflicts ————————————— tidyverse_conflicts() ——
## ✗ dplyr::filter() masks stats::filter()
## ✗ dplyr::lag()    masks stats::lag()
```

BASICS OF GGPLOT2

Let us first load a dataset to explore different visualizations. Consider the *mpg* data, that contains observations collected by the US Environmental Protection Agency on 38 models of car. The *mpg* data frame has several variables, including
- *displ* : a car's engine size, in litres.
- *hwy* : a car's fuel efficiency on the highway, in miles per gallon (mpg).

A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance.

Note: The *mpg* data set has 234 rows and 11 columns, and hence we load this data frame as a tibble.

Tibble is a better way of handling large data frames and provides all features as of data frame along with some additional features e.g., printing tibble prints 10 rows and as many columns as fits the screes.

BASICS OF GGPLOT2

For a basic plot, you need three primary steps
- Create a blank canvas for your plot, using the ggplot() call
- Specify aesthetic mappings, which specifies how you want to map variables to visual aspects.
- Add layers of geometric objects

 # create the blank canvas
ggplot(mpg)
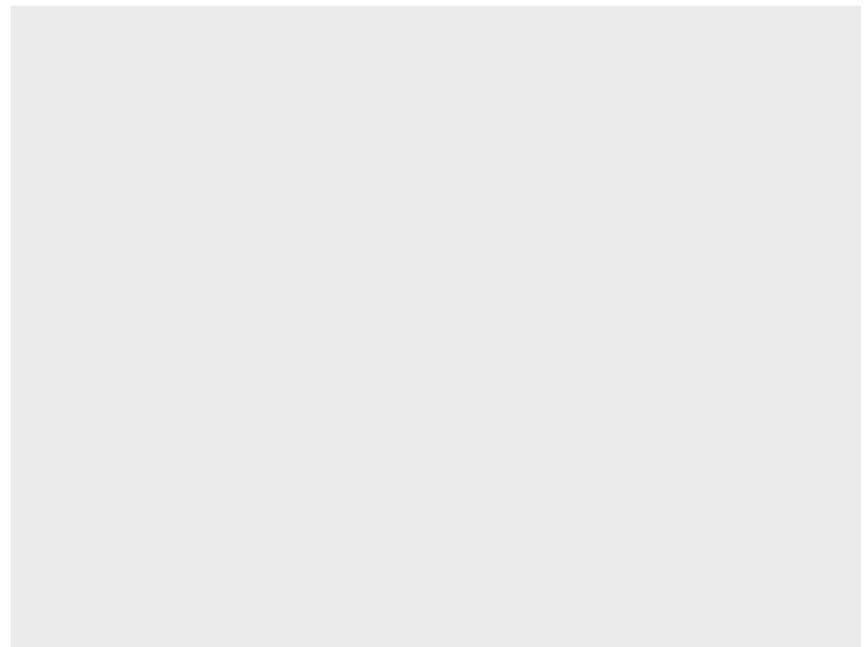
BASICS OF GGPLOT2

 # variables of interest mapped
ggplot(mpg, aes(x = displ, y = hwy))
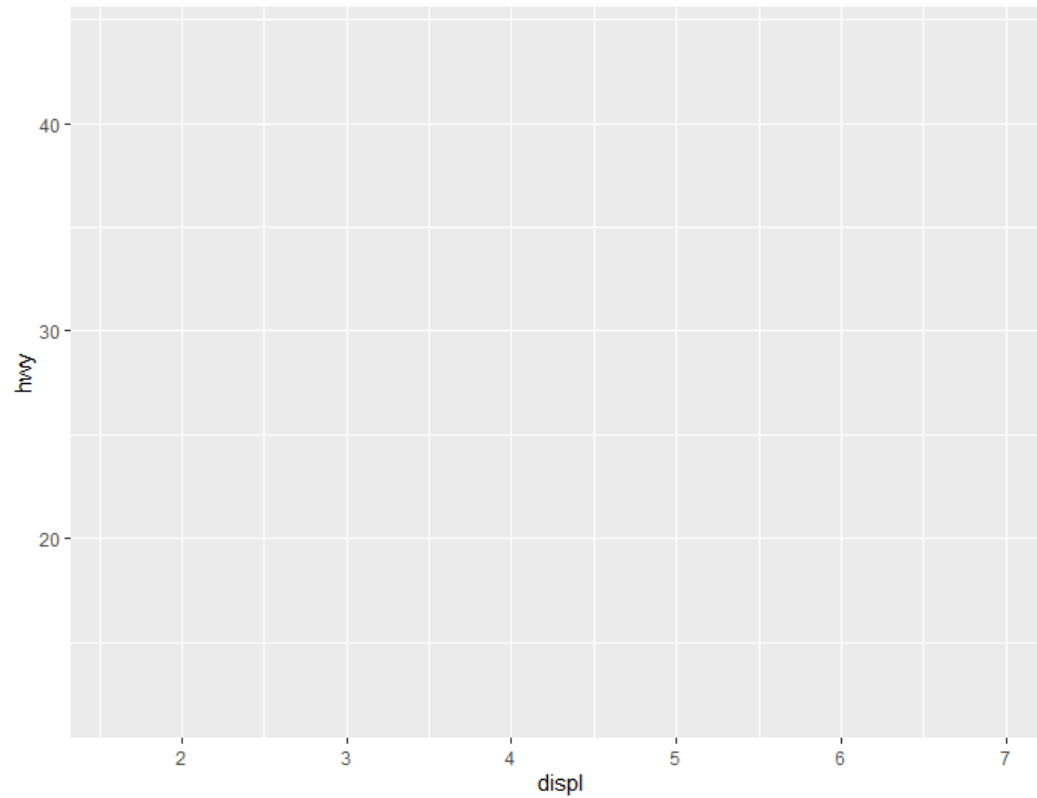
BASICS OF GGPLOT2

 # type of display
ggplot(mpg, aes(x = displ, y = hwy)) + geom_point()
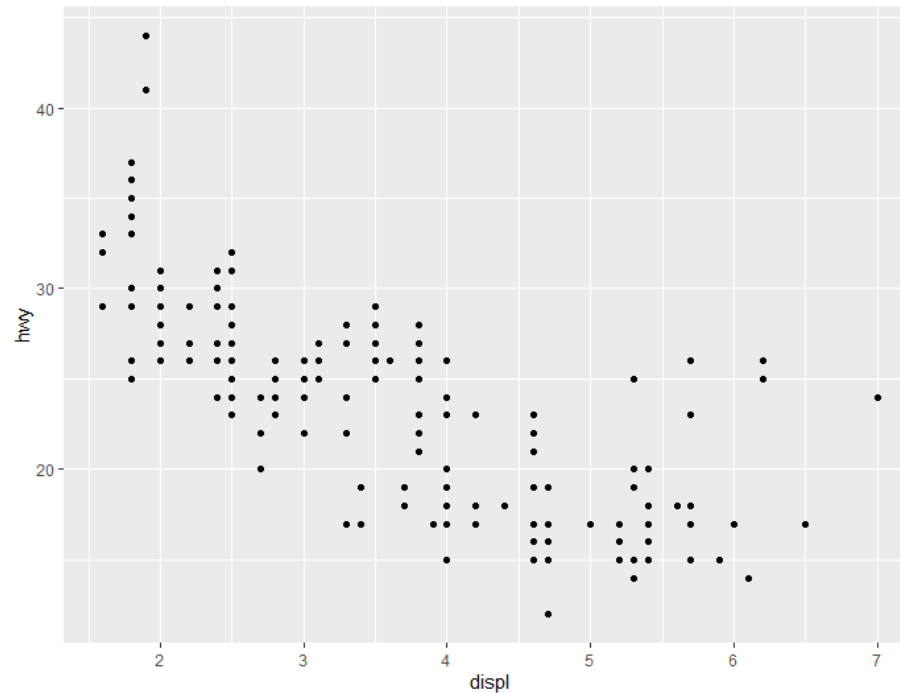
Note: We have added the geom layer you used the addition (+) operator. New layers are always added using (+)to add onto your visualization.

AESTHETICS MAPPING

- An aesthetic is a visual property of the objects in your plot.
- Aesthetics include things like the size, the shape, or the color of your points.
- You can display a point in different ways by changing the values of its aesthetic properties.
- All aesthetics for a plot are specified in the aes() function call
- Each geom layer can have its own aes specifications.

Aesthetic mappings: examples

- The class variable of the mpg dataset classifies cars into groups such as compact, midsize, and SUV.
- We can add a third variable, like class, to a two dimensional scatterplot by mapping it to an aesthetic.
- For example, we can map the colours of your points to the class variable to reveal the class of each car.

AESTHETICS MAPPING

ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, color = class))
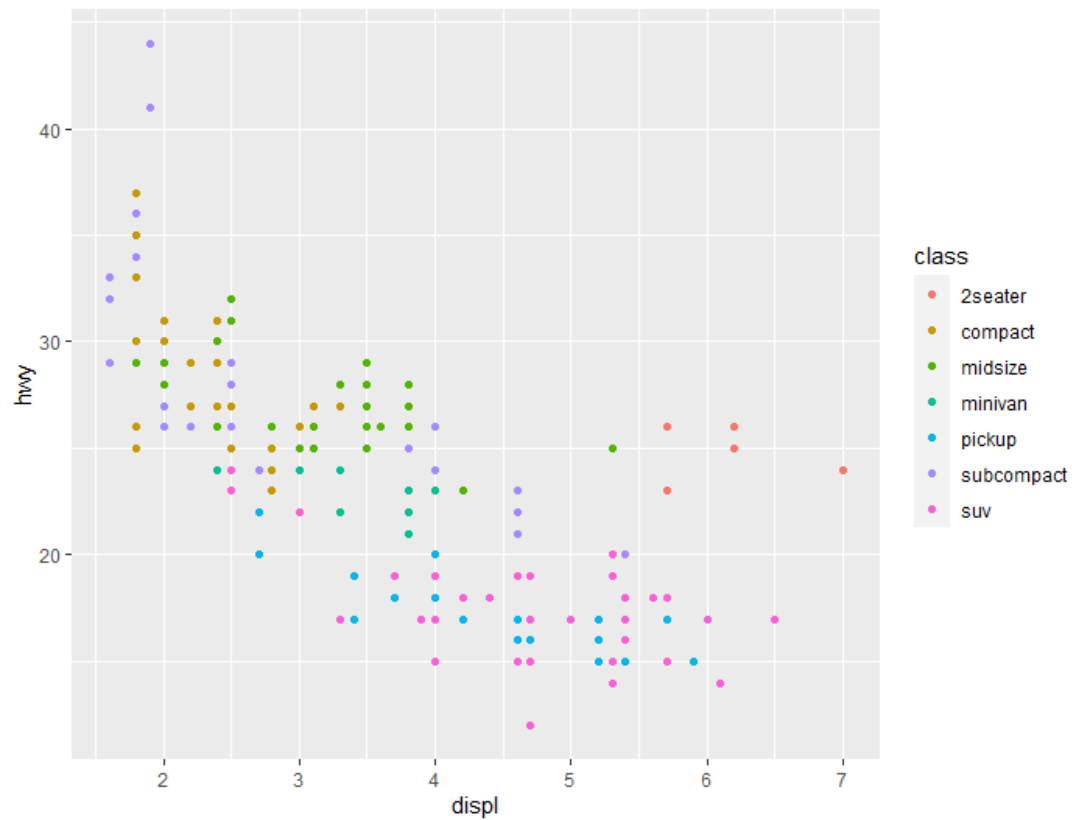
AESTHETICS MAPPING

We could have mapped class to the
- alpha aesthetic, which controls the transparency of the points, or to the
- shape aesthetic, which controls the shape of the points.

ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, alpha = class))
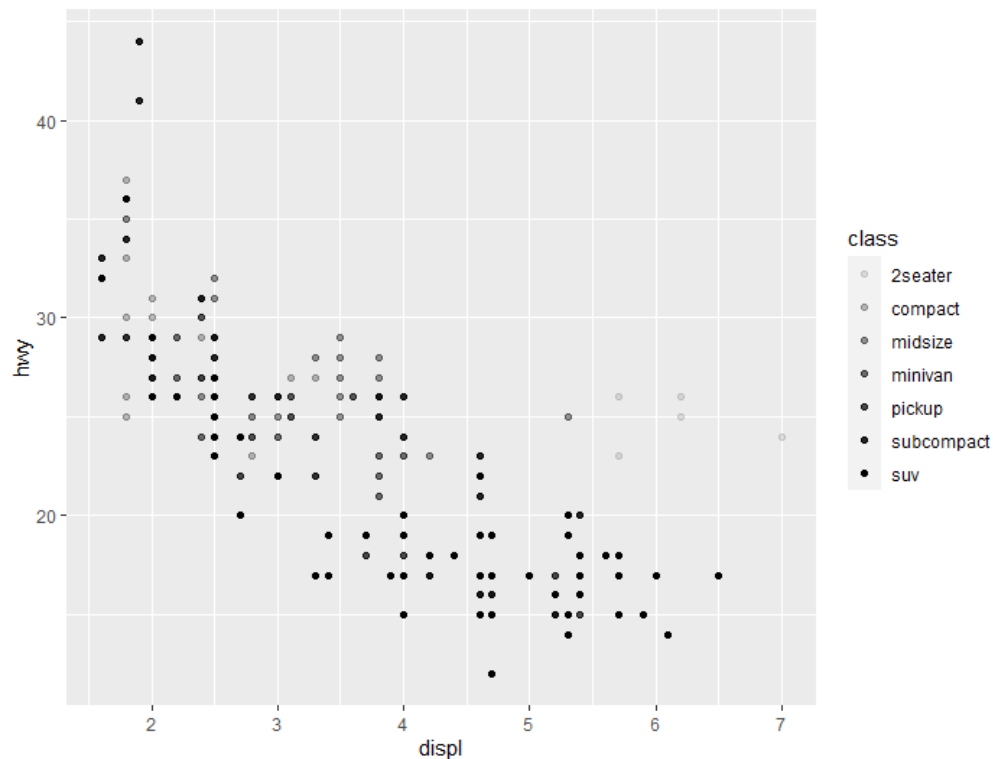
AESTHETICS MAPPING

 # shape aesthetic, which controls the shape of the points.
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy, shape = class))
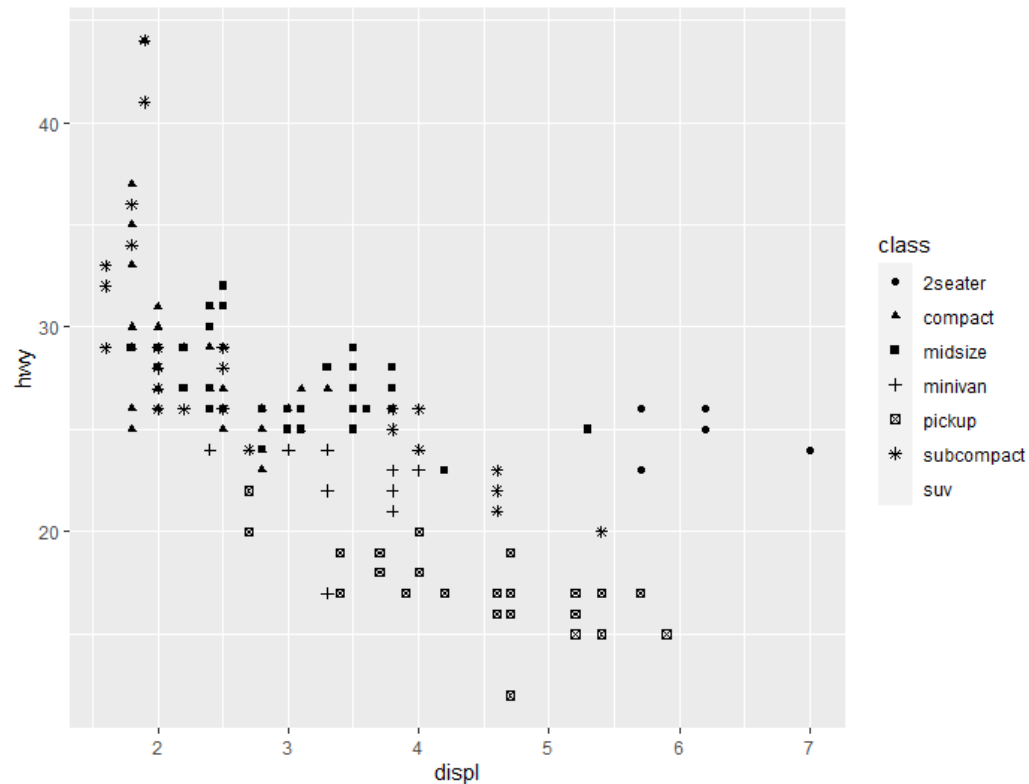
AESTHETICS MAPPING

 # You can also set the aesthetic properties of your geom manually. For example, we can make all of the points in our plot red:

ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy), color = "red")
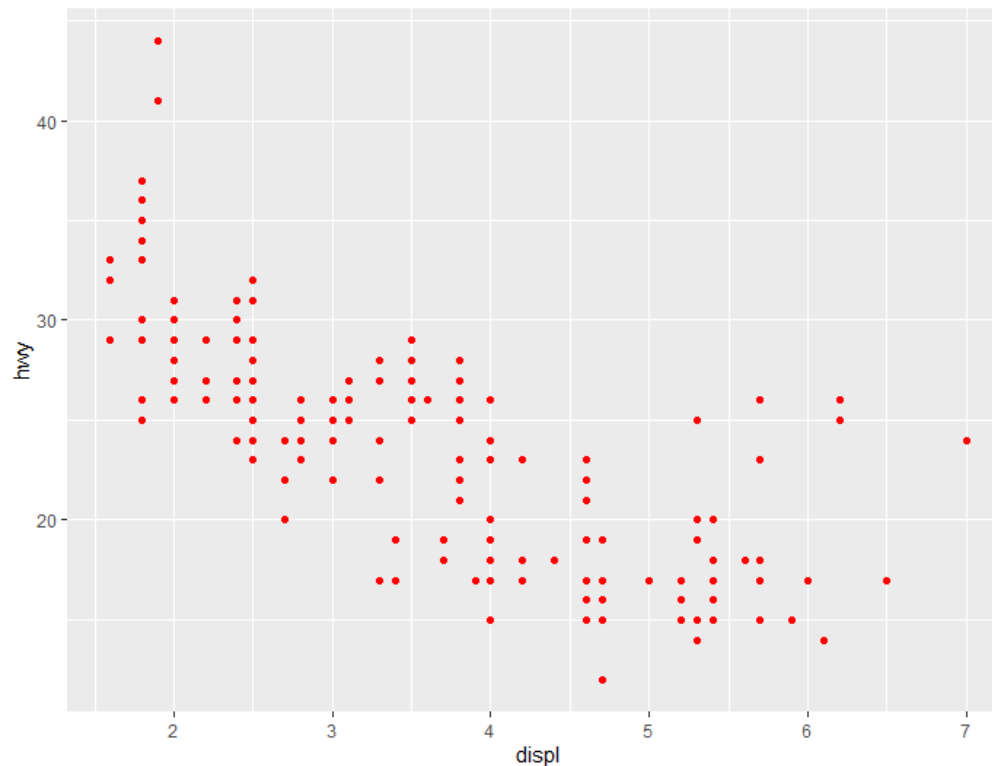
GEOMETRIC OBJECTS

ggplot2 supports different types of geometric objects, including:

- *geom_point*: drawing individual points, like a scatter plot
- *geom_line* : drawing lines
- *geom_smooth* : drawing smoothed lines, like moving averages
- *geom_bar* : drawing bars
- *geom_histogram* : drawing binned values, like a histogram
- *geom_polygon* : drawing arbitrary shapes
- *geom_map* : drawing polygons in the shape of a map! (cool feature indeed)

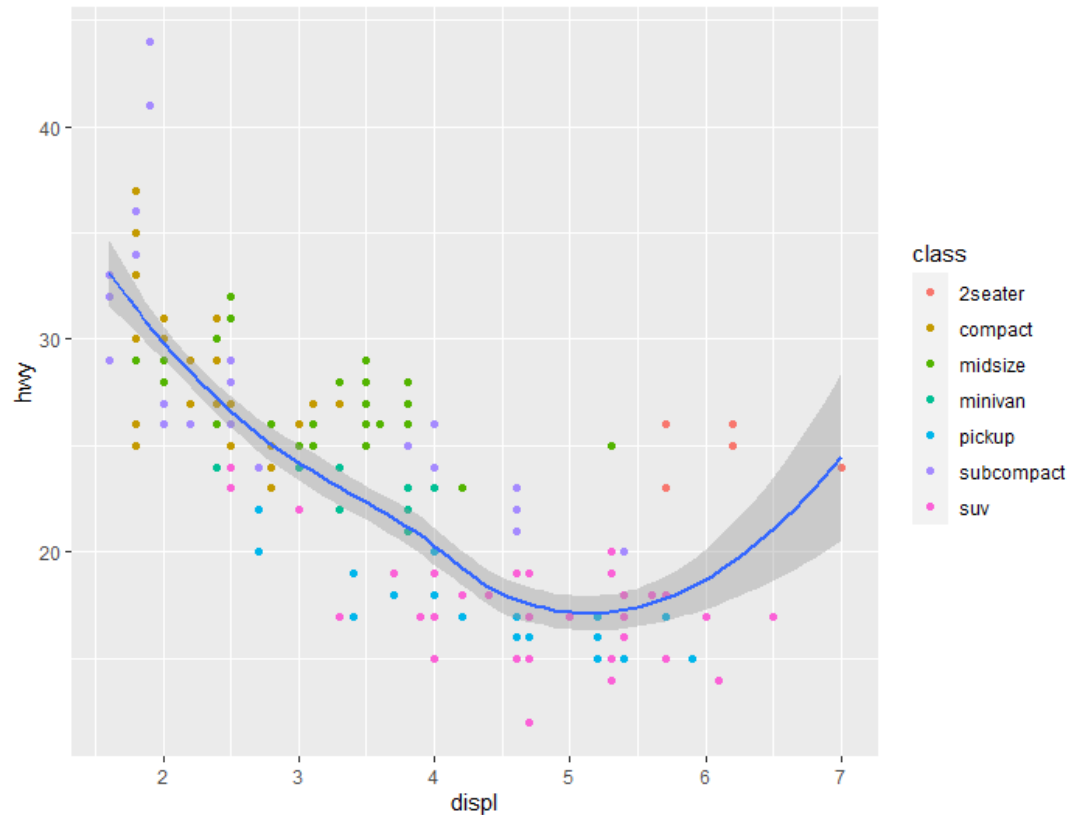# GEOMETRIC OBJECTS

ggplot(mpg, aes(x = displ, y = hwy)) + geom_point() + geom_smooth()
# `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
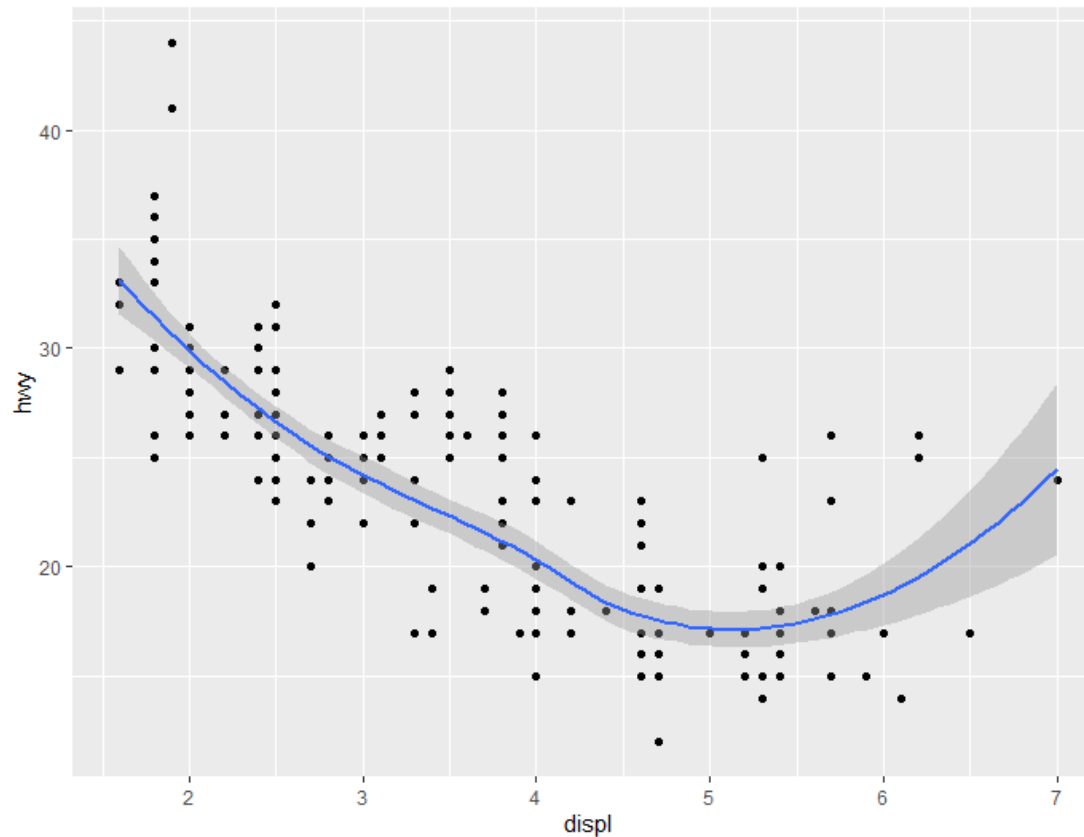
GEOMETRIC OBJECTS

ggplot(data = mpg, aes(x = class)) + geom_bar()
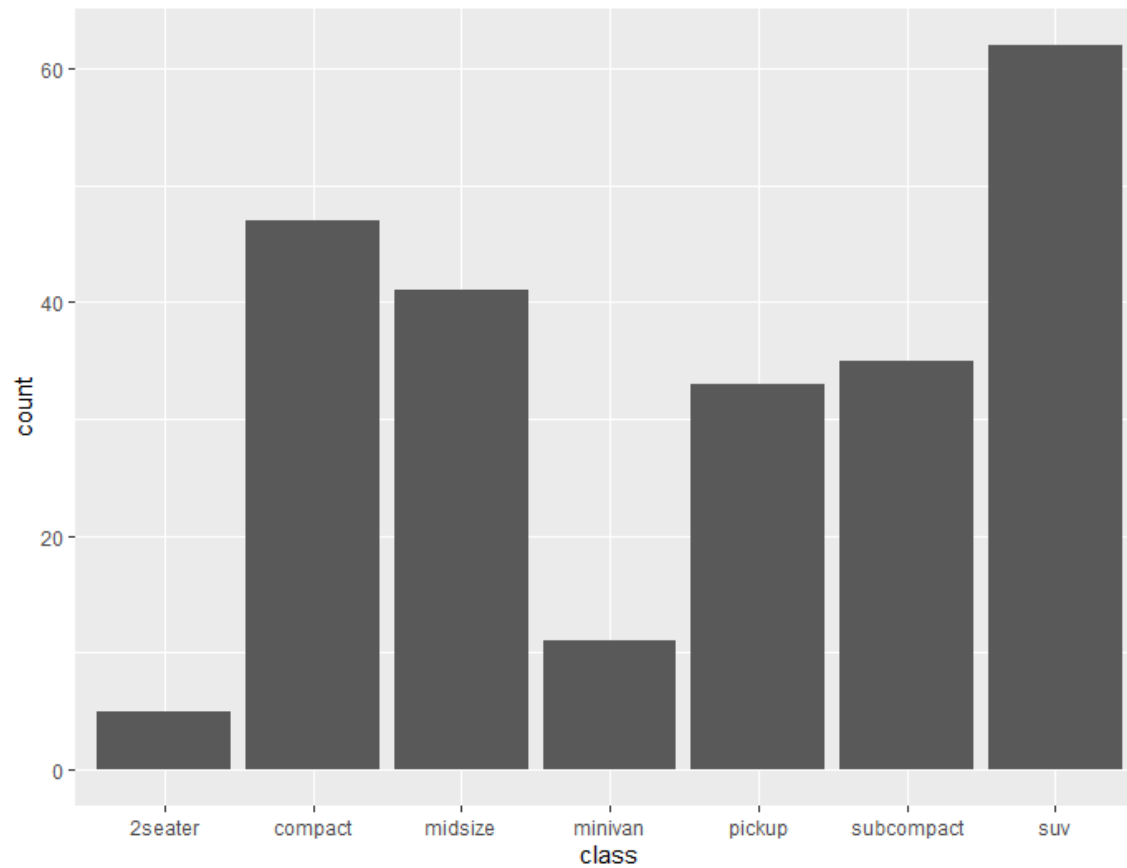
GEOMETRIC OBJECTS

ggplot(data = mpg, aes(x = hwy)) + geom_histogram()
# `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
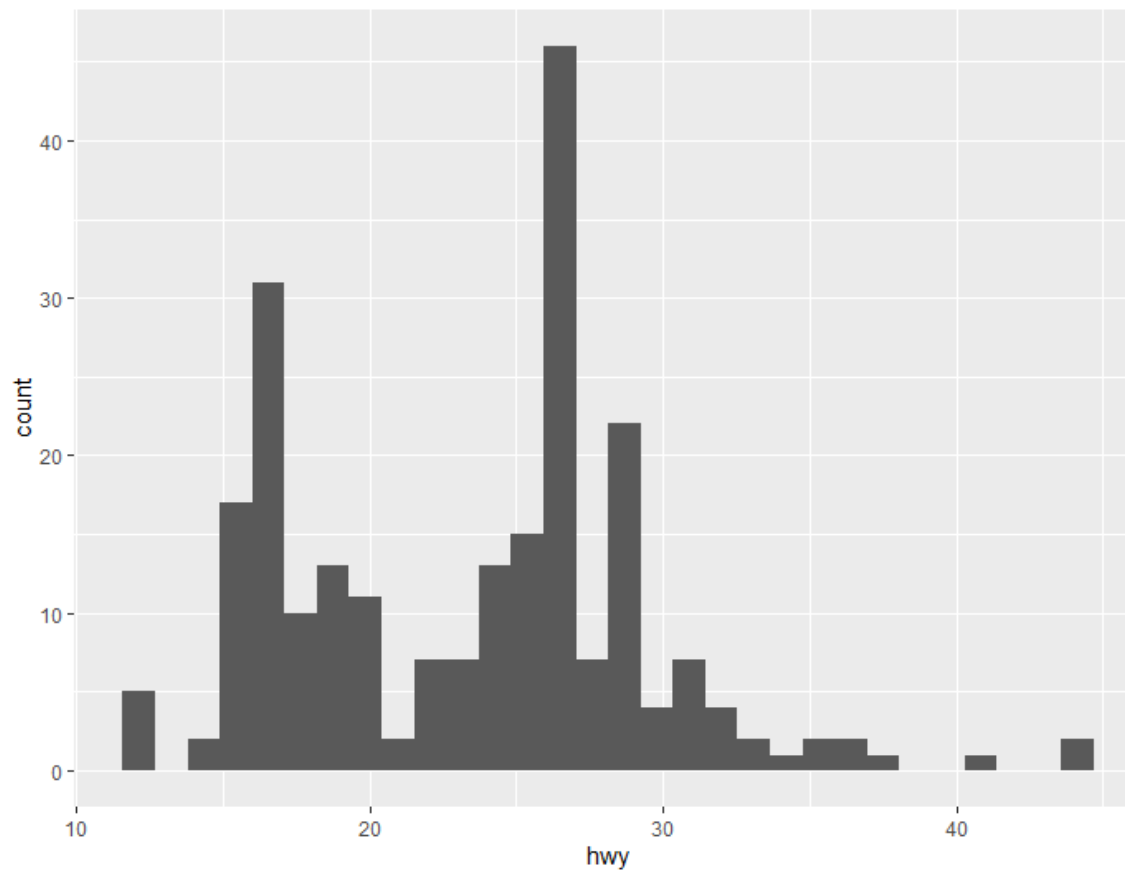
GEOMETRIC OBJECTS

- The aesthetics for each *geom* can be different, so you could show multiple lines on the same plot (or with different colors, styles, etc).
- It is also possible to give each *geom* a different data argument, so that you can show multiple data sets in the same plot.
- If you place mappings in a *geom* function, ggplot2 will treat them as local mappings for the layer.
- It will use these mappings to extend or overwrite the global mappings for that layer only. This makes it possible to display different aesthetics in different layers.

GEOMETRIC OBJECTS

ggplot(mpg, aes(x = displ, y = hwy)) + geom_point(color = "blue") + geom_smooth(color = "red")

`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
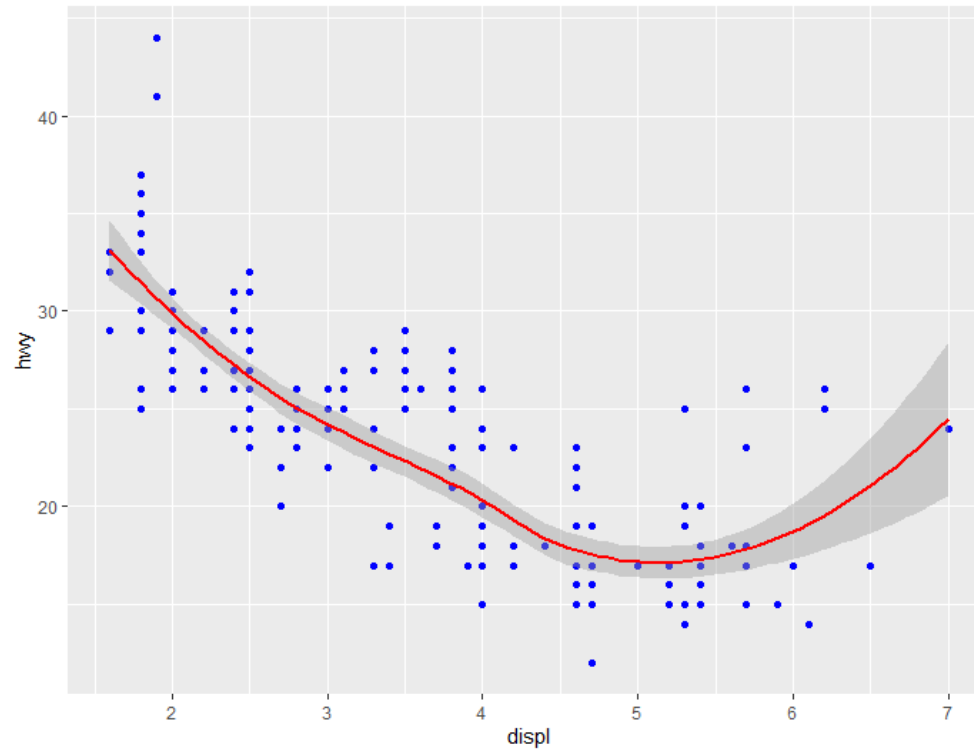
GEOMETRIC OBJECTS

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom_point(mapping = aes(color = class)) + geom_smooth()
# `geom_smooth()` using method = 'loess' and formula = 'y ~ x'

GEOMETRIC OBJECTS

- We can use the same idea to specify different data for each layer.
- Let us say that we shall display the smooth line for just a subset of the mpg dataset, the subcompact cars.
- The local data argument in geom_smooth() overrides the global data argument in ggplot() for that layer only.

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom_point(mapping = aes(color = class)) + geom_smooth(data = filter(mpg, class == "subcompact"), se = FALSE)
# `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
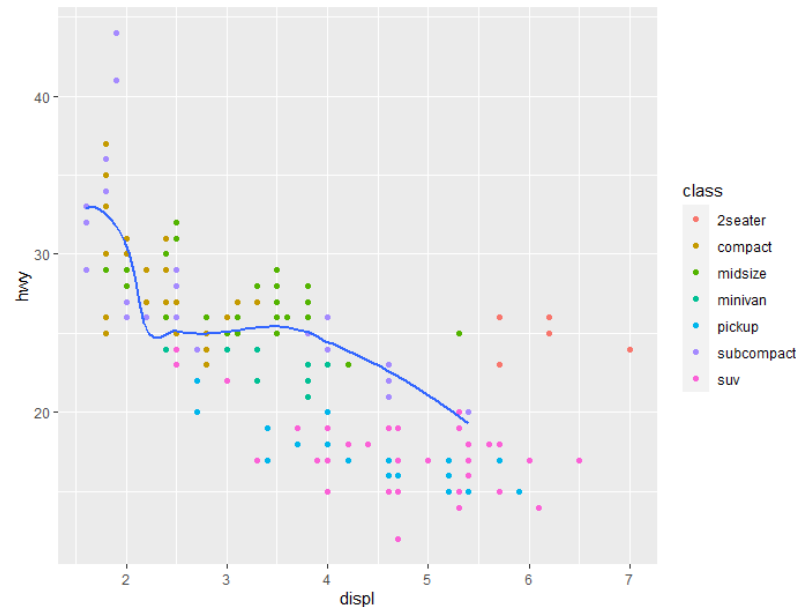
STATISTICAL TRANSFORMATIONS

Let us revisit the default bar chart we had shown before.

ggplot(data = mpg, aes(x = class)) + geom_bar()
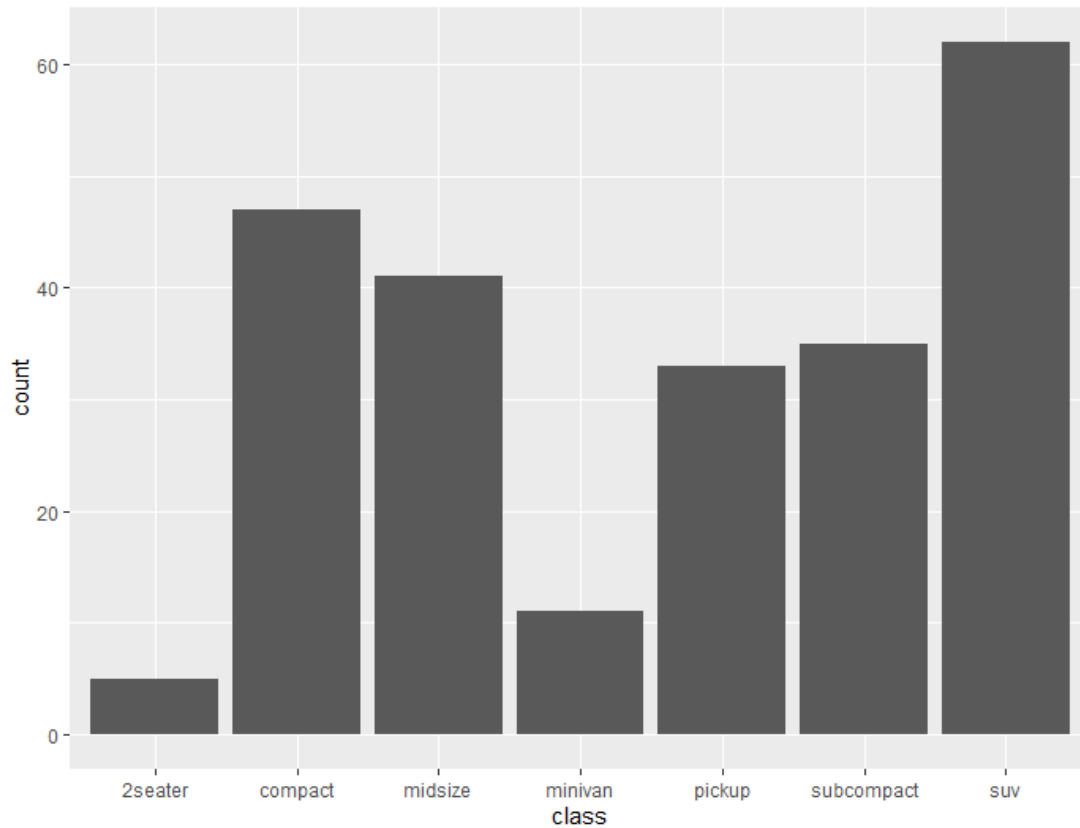
STATISTICAL TRANSFORMATIONS

- Notice that the y axis was defined for us as the count of elements that have the particular type.
- This count isn't part of the data set (it's not a column in mpg), but is instead a statistical transformation that the *geom_bar* automatically applies to the data.
- In particular, it applies the *stat_count* transformation.
- You might want to override the default mapping from transformed variables to aesthetics. For example, you might want to display a bar chart of proportion, rather than count.
- ggplot2 provides over 20 stats for you to use. Each stat is a function, so you can get help in the usual way, e.g. ?stat_bin.
- To see a complete list of stats, try the qqplot2 cheatsheet.

STATISTICAL TRANSFORMATIONS

ggplot(data = mpg, aes(x = class)) + geom_bar(mapping = aes(x = class, y = stat(prop), group = 1))
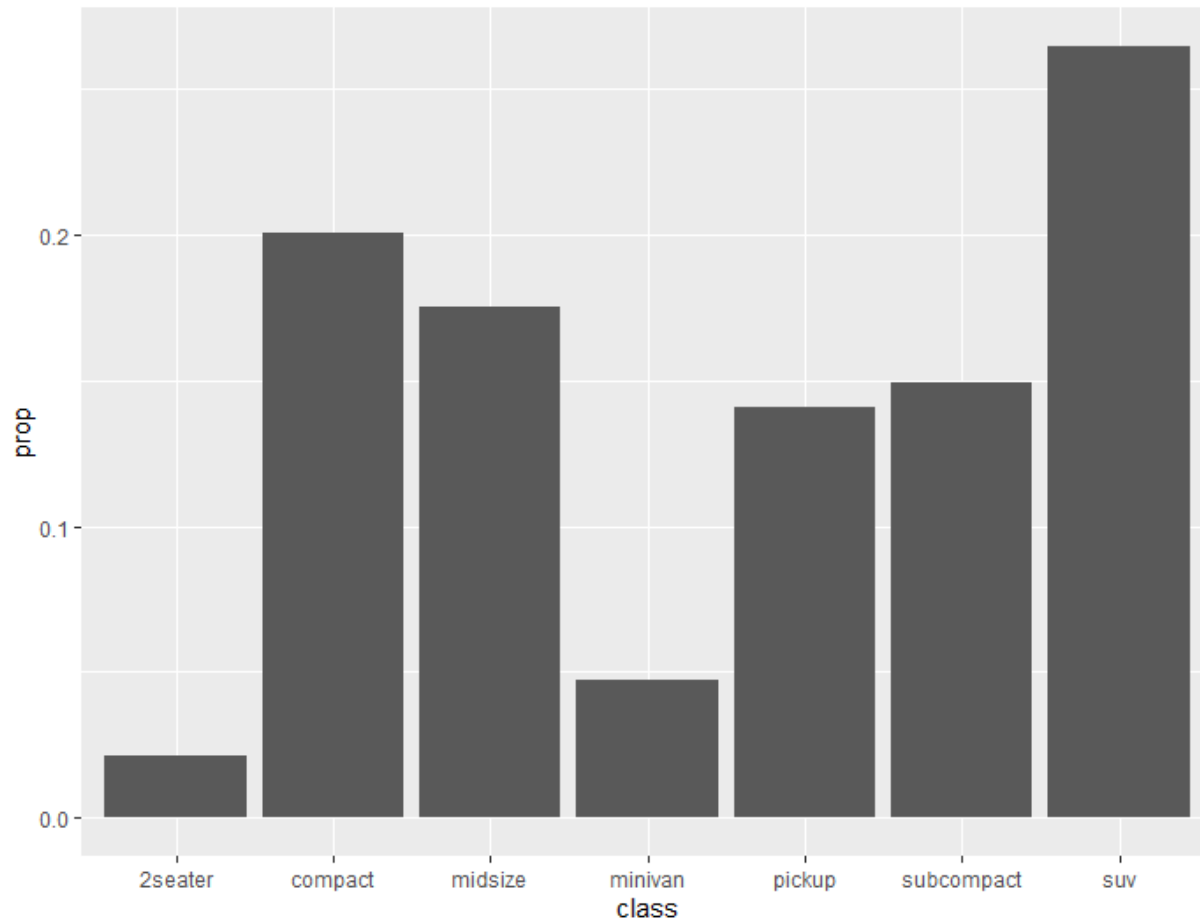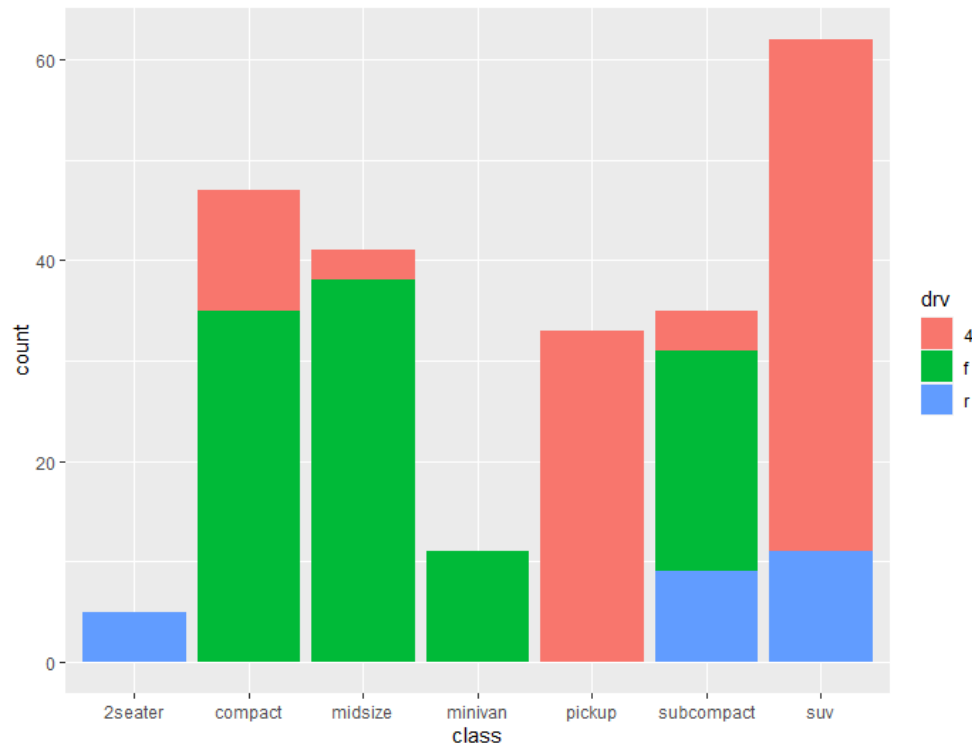
POSITION ADJUSTMENTS

- Each geom also has a default position adjustment which specifies a set of "rules" as to how different components should be positioned relative to each other.
- This position is noticeable in a geom_bar if you map a different variable to the color visual characteristic.

 # bar chart of class, colored by drive (front, rear, 4-wheel)
ggplot(mpg, aes(x = class, fill = drv)) + geom_bar()

POSITION ADJUSTMENTS

- The geom_bar by default uses a position adjustment of "stack", which makes each rectangle's height proprotional to its value and stacks them on top of each other.
- We can use the position argument to specify what position adjustment rules to follow..

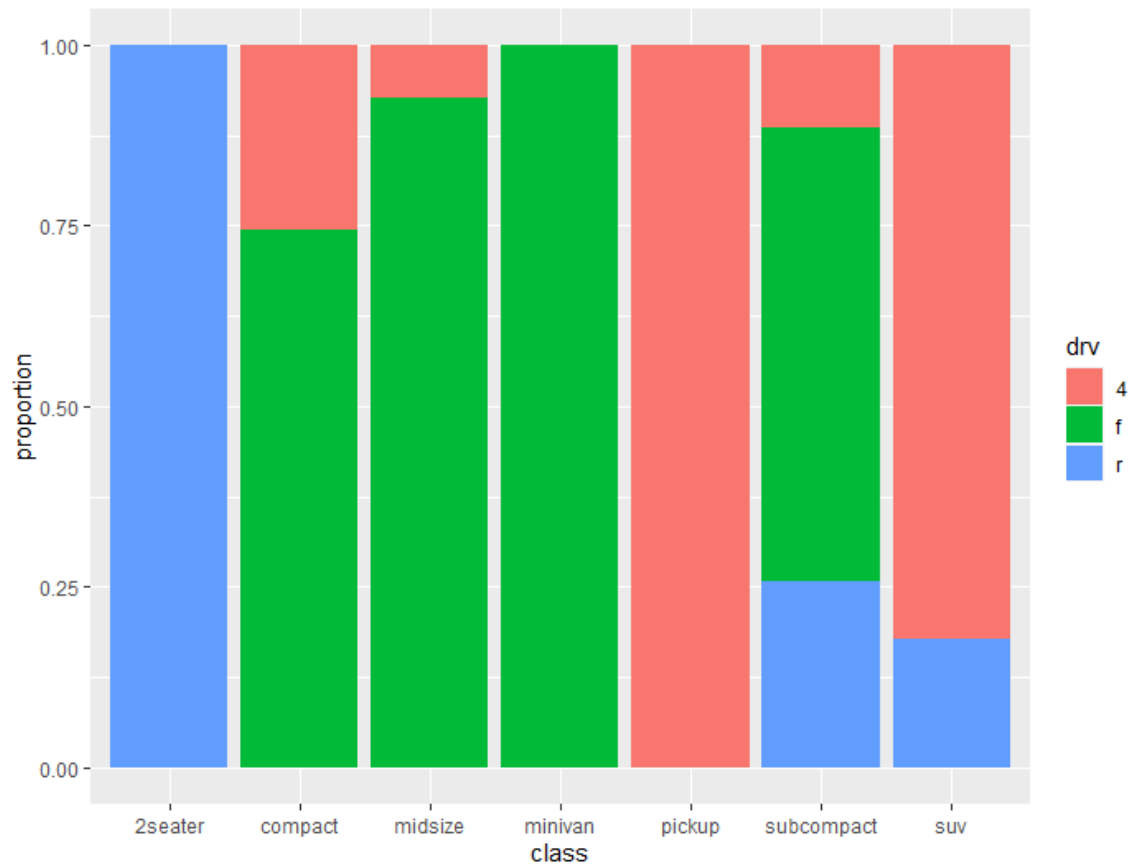**# position = "dodge": values next to each other**
ggplot(mpg, aes(x = class, fill = drv)) + geom_bar(position = "dodge")
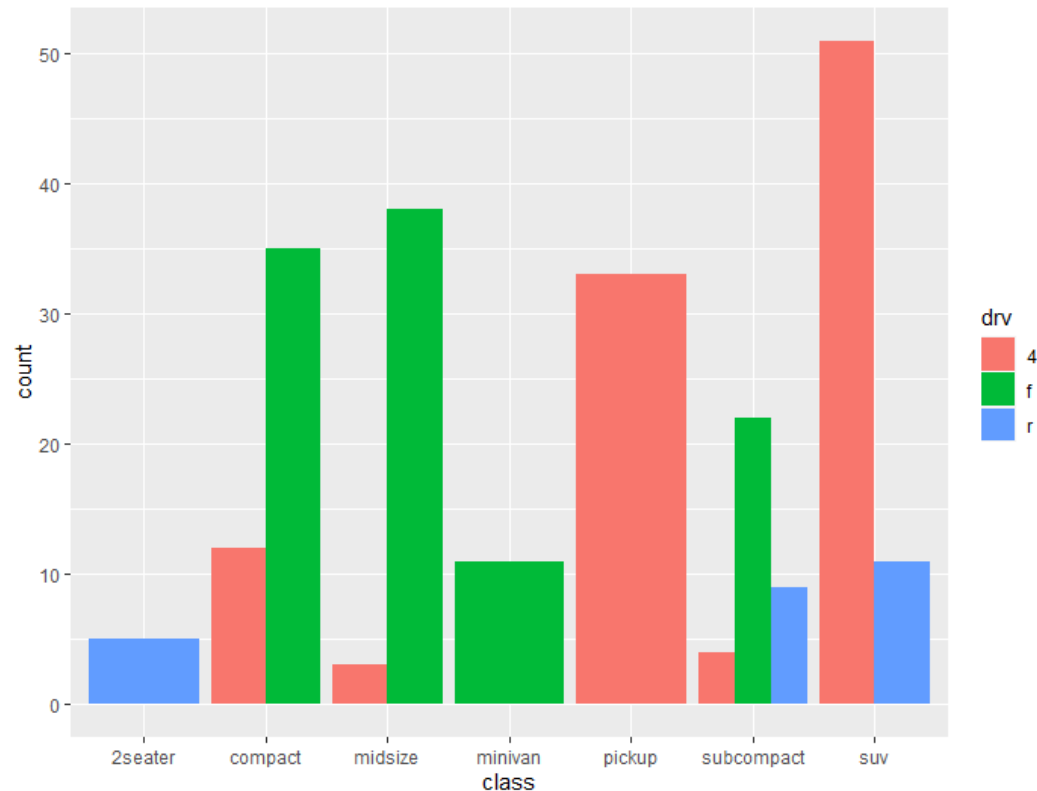
POSITION ADJUSTMENTS

# position = "fill": percentage/proportions chart
ggplot(mpg, aes(x = class, fill = drv)) + geom_bar(position = "fill") + ylab("proportion")

CO-ORDINATE SYSTEMS

The default coordinate system is the Cartesian coordinate system where the x and y positions act independently to determine the location of each point.

There are a number of other coordinate systems that are occasionally helpful.
- ❑ coord_flip() switches the x and y axes. This is useful (for example), if you want horizontal boxplots. It's also useful for long labels: it's hard to get them to fit without overlapping on the x-axis.
- ❑ coord_fixed a cartesian system with a "fixed" aspect ratio (e.g., 1.78 for a "widescreen" plot)
- ❑ coord_polar a plot using polar coordinates
- ❑ coord_quickmap a coordinate system that approximates a good aspect ratio for maps.

CO-ORDINATE SYSTEMS

 # flip x and y axis with coord_flip
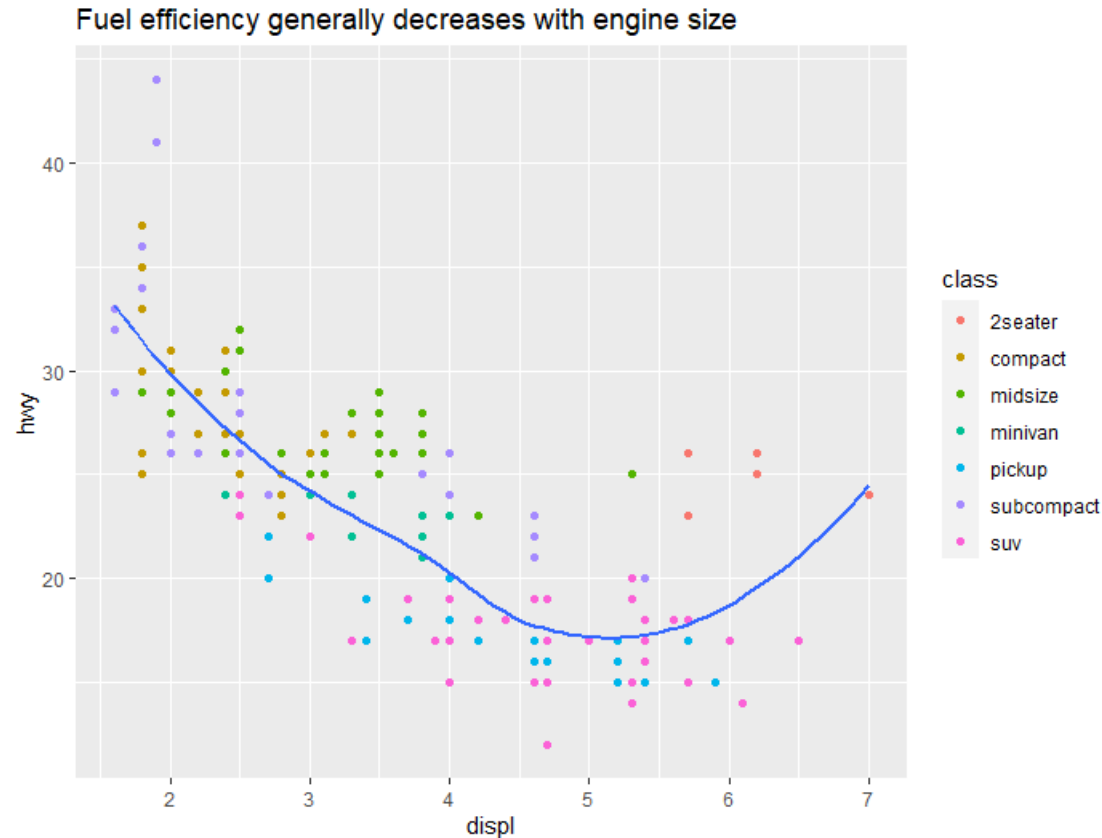ggplot(mpg, aes(x = class)) + geom_bar() + coord_flip()
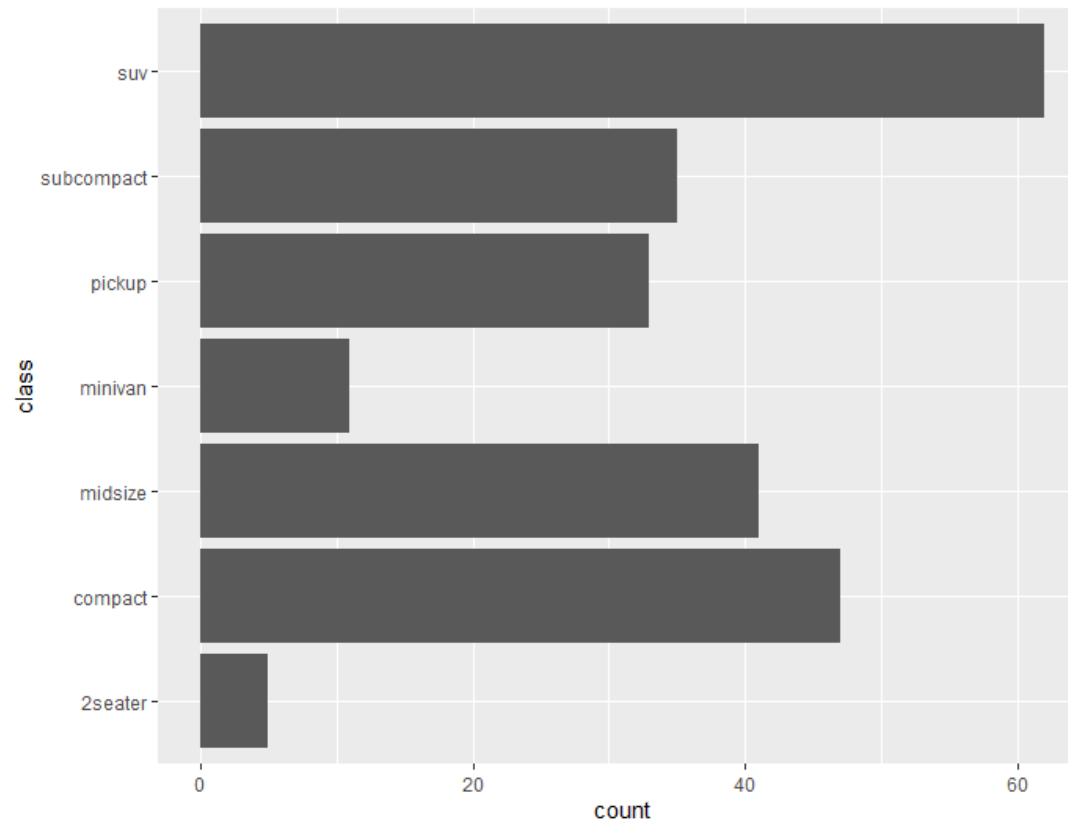
We add labels with the *labs( )* function. This example adds a plot title

Adding title to the plot
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(color = class)) + geom_smooth(se = FALSE) + labs(title = "Fuel efficiency generally decreases with engine size")

 # We can also use labs() to replace the axis and legend titles.
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(color = class)) + geom_smooth(se = FALSE) + labs(title = "Fuel efficiency generally decreases with engine size", x = "Engine displacement (L)", y = "Highway fuel economy (mog)", color = "Car types")
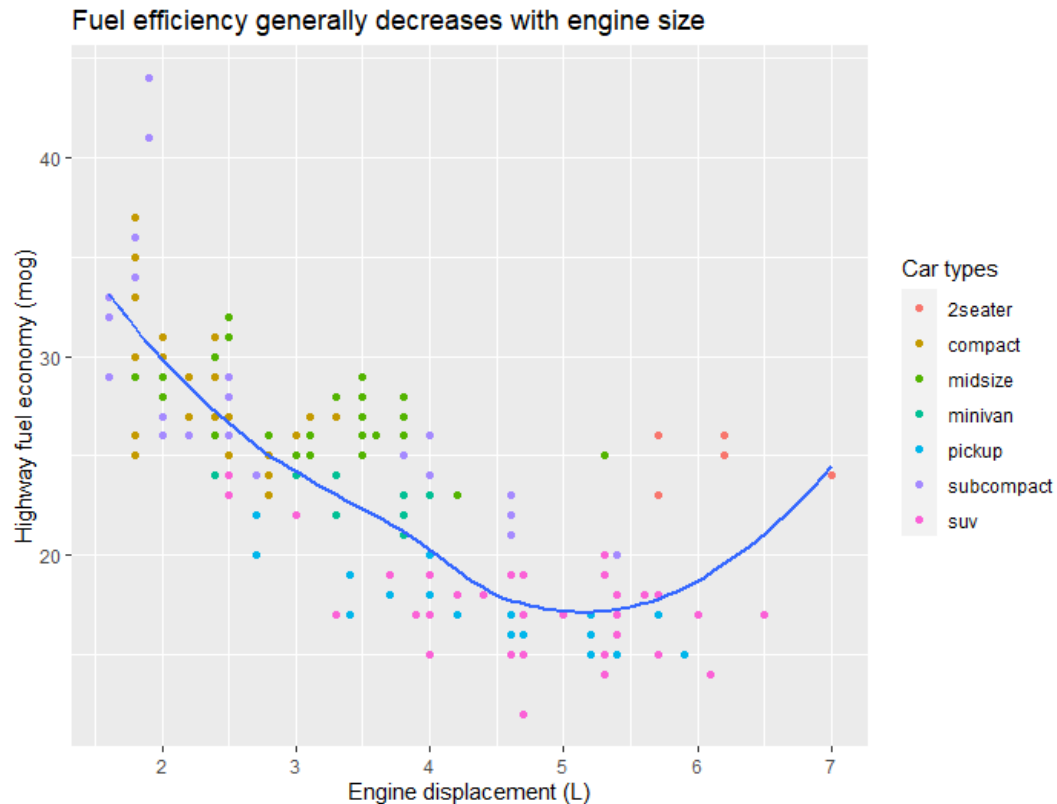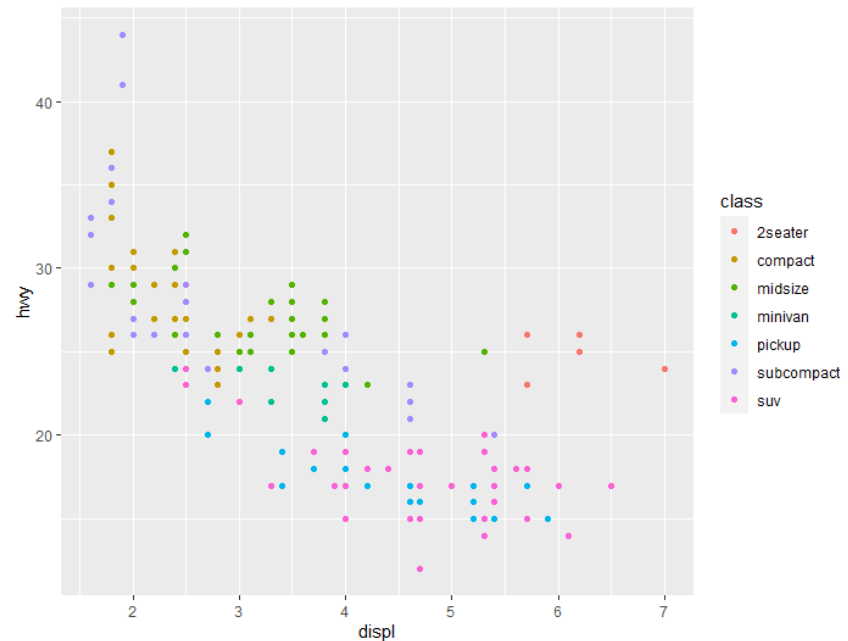
SCALES

Scales control the mapping from data values to things that you can perceive. *ggplot2* automatically adds scales for you. For example, when you type

ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = class))

ggplot adds the scales automatically behind the scenes.

ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  scale_x_continuous() +
 scale_y_continuous() +
  scale_colour_discrete()

SCALES: AXIS TICKS

- There are two primary arguments that affect the appearance of the ticks on the axes and the keys on the legend: breaks and labels.
- Breaks controls the position of the ticks, or the values associated with the keys.
- Labels controls the text label associated with each tick/key. The most common use of breaks is to override the default choice:

Change the scale of y axis
ggplot(mpg, aes(displ, hwy)) +
geom_point() + scale_y_continuous(breaks =
seq(15, 40, by = 5))
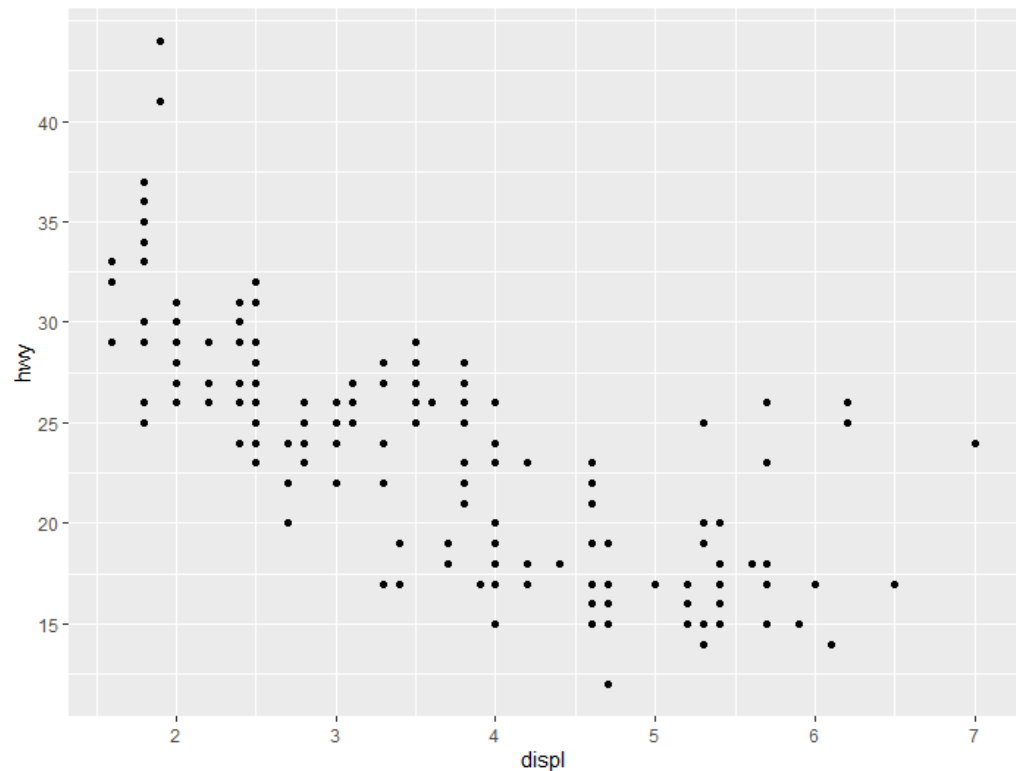
SCALES: LEGEND KEYS

- We will most often use breaks and labels to tweak the axes.
- While they both also work for legends, there are a few other techniques we are more likely to use.
- To control the overall position of the legend, we need to use a theme() setting.
- The theme setting legend.position controls where the legend is drawn:

 # Change the legend key
base <- ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class))
base + theme(legend.position = "left")
 # 'right' is default
**# may use 'top' or 'bottom' as well**
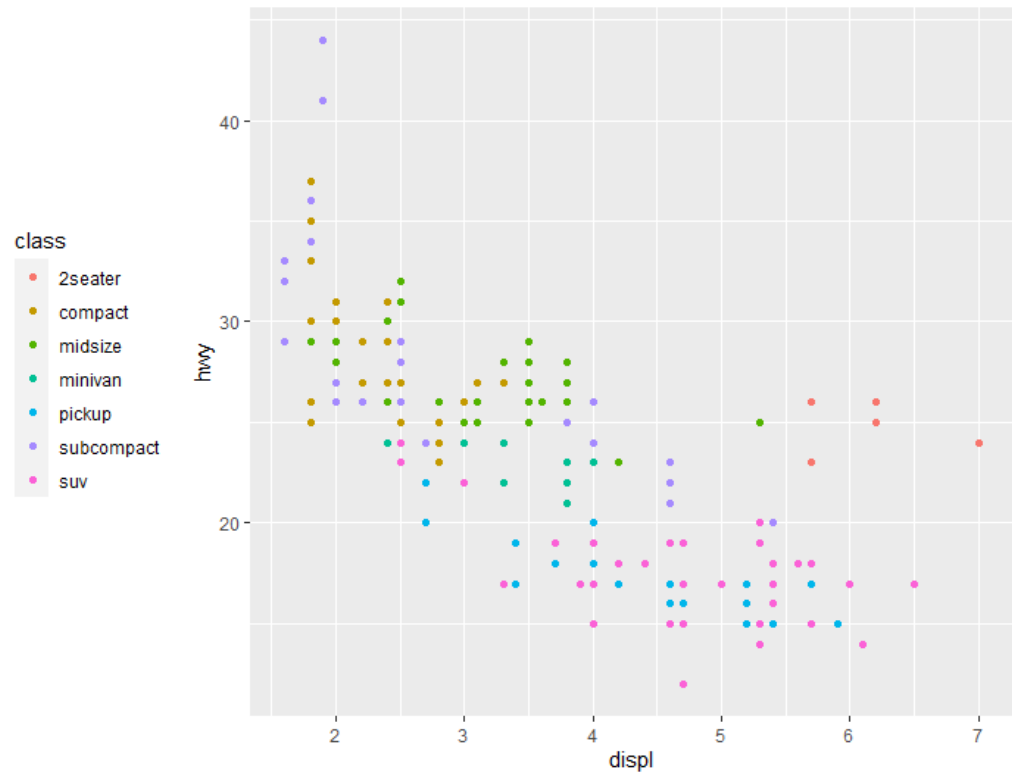
FACETS

- One way to add additional variables is with aesthetics.
- Another way, particularly useful for categorical variables, is to split your plot into facets, subplots that each display one subset of the data.
- To facet your plot by a single variable, use facet_wrap( ).
- The first argument of facet_wrap () should be a formula, which you create with followed by a variable name (here "formula" is the name of a data structure in R, not a synonym for "equation").
- The variable that you pass to facet_wrap() should be discrete.

FACETS

Adding another varaible in the plot
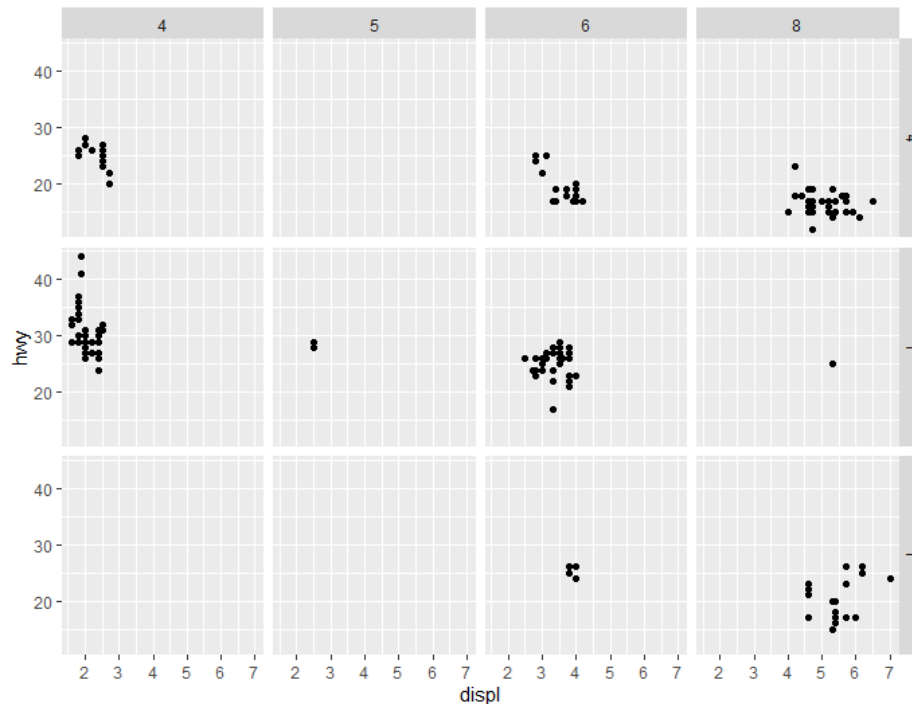ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) +
facet_wrap(~ class, nrow = 2)
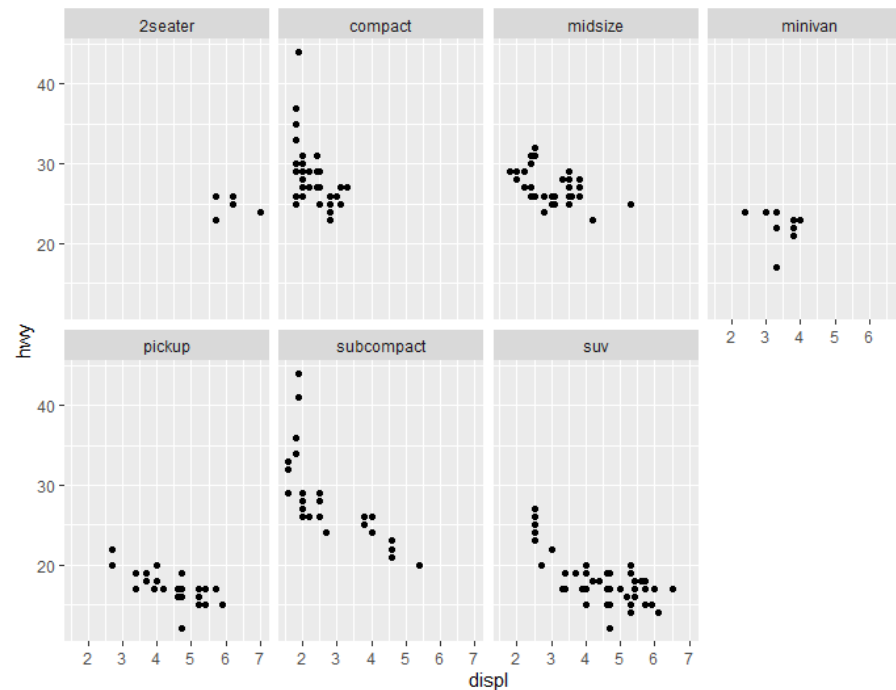
FACETS

- To facet your plot on the combination of two variables, add *facet_grid()* to your plot call.
- The first argument of *facet_grid()* is also a formula. This time the formula should contain two variable names separated by a ~.

ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) + facet_grid(drv ~ cyl)

SAVING PLOTS

- Saving your plots is an important aspect, specially if you are going to use them in your analysis reports.
- For plots generated using ggplot2, we can use *ggsave()* to save plots.
- Look into the help function for ggsave using ?ggsave for finer aspects.
- *ggsave()* will save the most recent plot to the disk.

```
ggplot(mpg, aes(displ, hwy)) + geom_point()
ggsave("plot1.png")
# Saving 7.29 x 5.56 in image
```

DATA VISUALIZATION

With ever increasing volume of data, it is impossible to tell stories without visualizations. Data visualization is an art of how to turn numbers into useful knowledge.

Popular Data Visualization Techniques:

1. Scatter Plot

2. Histogram

3. Bar & Stack Bar Chart

4. Box Plot

5. Area Chart

6. HeatMap

7. Correlogram

DATA VISUALIZATION

We'll use 'Big_Mart_Dataset.csv' example as shown below to understand how to create visualizations.

| Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type
FDA15	9.300	Low Fat	0.016047301	Dairy	249.8092	OUT049	1999	Medium	Tier 1	Supermarket Typ
DRC01	5.920	Regular	0.019278216	Soft Drinks	48.2692	OUT018	2009	Medium	Tier 3	Supermarket Typ
FDN15	17.500	Low Fat	0.016760075	Meat	141.6180	OUT049	1999	Medium	Tier 1	Supermarket Typ
FDX07	19.200	Regular	0.000000000	Fruits and Vegetables	182.0950	OUT010	1998		Tier 3	Grocery Store
NCD19	8.930	Low Fat	0.000000000	Household	53.8614	OUT013	1987	High	Tier 3	Supermarket Typ
FDP36	10.395	Regular	0.000000000	Baking Goods	51.4008	OUT018	2009	Medium	Tier 3	Supermarket Typ
FDO10	13.650	Regular	0.012741089	Snack Foods	57.6588	OUT013	1987	High	Tier 3	Supermarket Typ
FDP10	NA	Low Fat	0.127469857	Snack Foods	107.7622	OUT027	1985	Medium	Tier 3	Supermarket Typ
FDH17	16.200	Regular	0.016687114	Frozen Foods	96.9726	OUT045	2002		Tier 2	Supermarket Typ
FDU28	19.200	Regular	0.094449590	Frozen Foods	187.8214	OUT017	2007		Tier 2	Supermarket Typ
FDY07	11.800	Low Fat	0.000000000	Fruits and Vegetables	45.5402	OUT049	1999	Medium	Tier 1	Supermarket Typ
FDA03	18.500	Regular	0.045463773	Dairy	144.1102	OUT046	1997	Small	Tier 1	Supermarket Typ
FDX32	15.100	Regular	0.100013500	Fruits and Vegetables	145.4786	OUT049	1999	Medium	Tier 1	Supermarket Typ
FDS46	17.600	Regular	0.047257328	Snack Foods	119.6782	OUT046	1997	Small	Tier 1	Supermarket Typ
FDF32	16.350	Low Fat	0.068024300	Fruits and Vegetables	196.4426	OUT013	1987	High	Tier 3	Supermarket Typ

DATA VISUALIZATION

1. Scatter Plot: It is used to see the relationship between two continuous variables. In our above mart dataset, if we want to visualize the items as per their cost data, then we can use scatter plot chart using two continuous variables, namely Item_Visibility & Item_MRP as shown.

Read data and simple scatter plot using function ggplot() with geom_point().

```
> train <  read.csv("C:/Users/Data/Big_Mart_Dataset.csv")

> view(train)

> library(ggplot2)

> ggplot(train, aes(Item_Visibility, Item_MRP)) + geom_point() +
scale_x_continuous("Item Visibility", breaks = seq(0,0.35,0.05))+
scale_y_continuous("Item MRP", breaks = seq(0,270,by = 30))+ theme_bw()
```
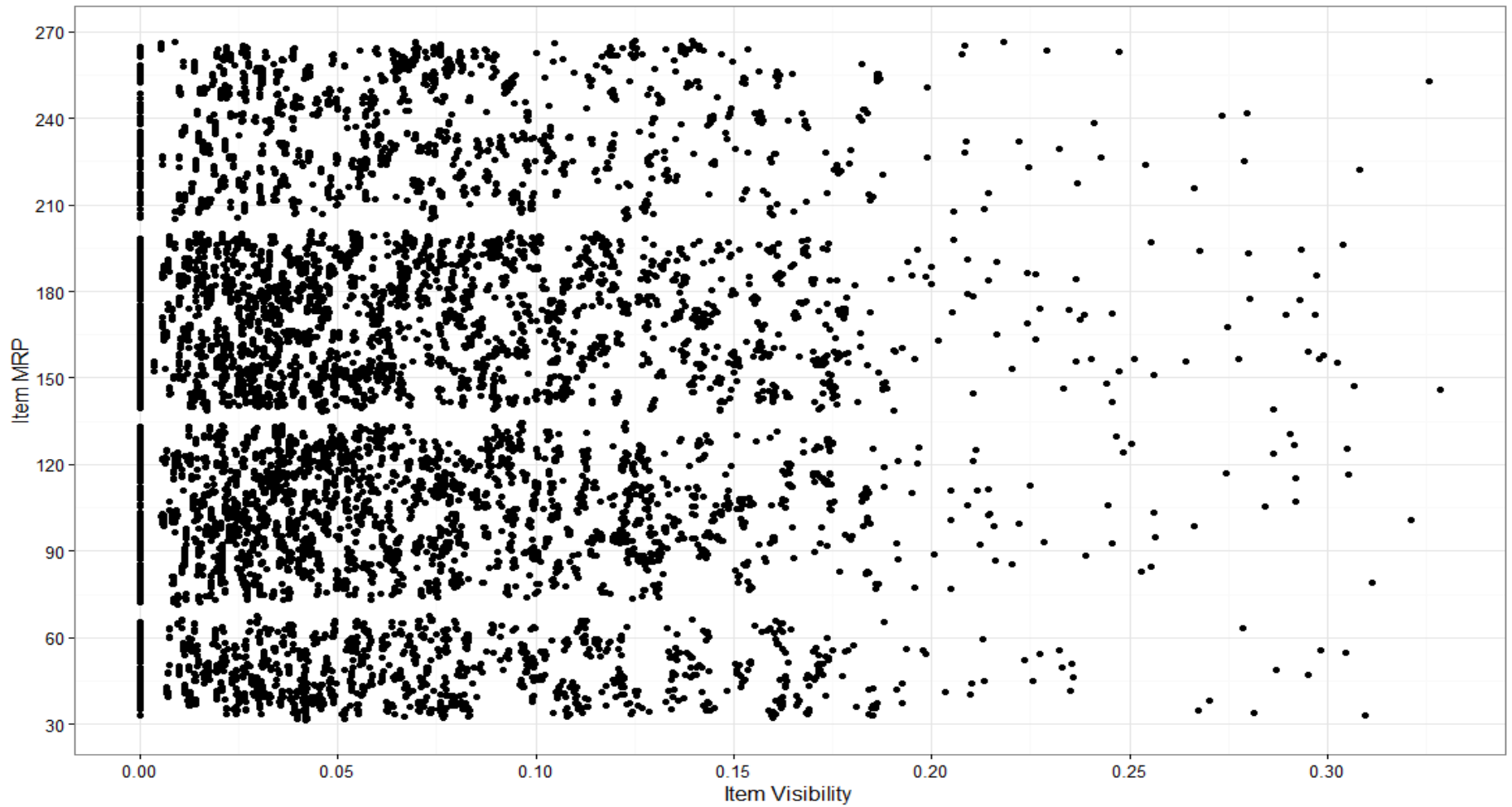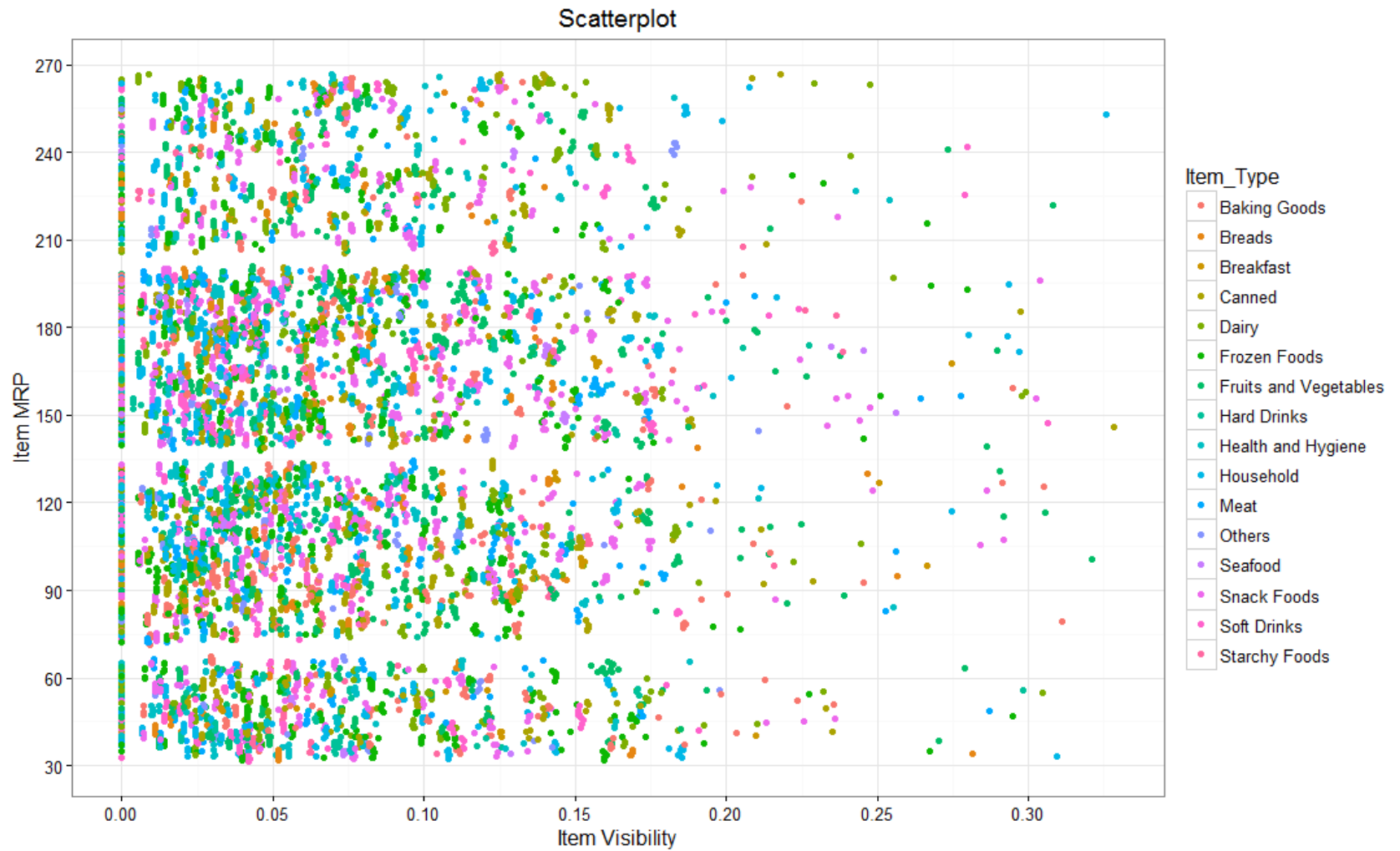
DATA VISUALIZATION

1. Scatter Plot: Now, we can view a third variable also in same chart, say a categorical variable (Item_Type) which will give the characteristic (item_type) of each data set. Different categories are depicted by way of different color for item_type in below chart.

Another scatter plot using function ggplot() with geom_point().

```
> library(ggplot2)

> ggplot(train, aes(Item_Visibility, Item_MRP)) + geom_point(aes(color =
Item_Type)) + scale_x_continuous("Item Visibility", breaks =
seq(0,0.35,0.05))+ scale_y_continuous("Item MRP", breaks = seq(0,270,by =
30))+ theme_bw() + labs(title="Scatterplot")
```

DATA VISUALIZATION
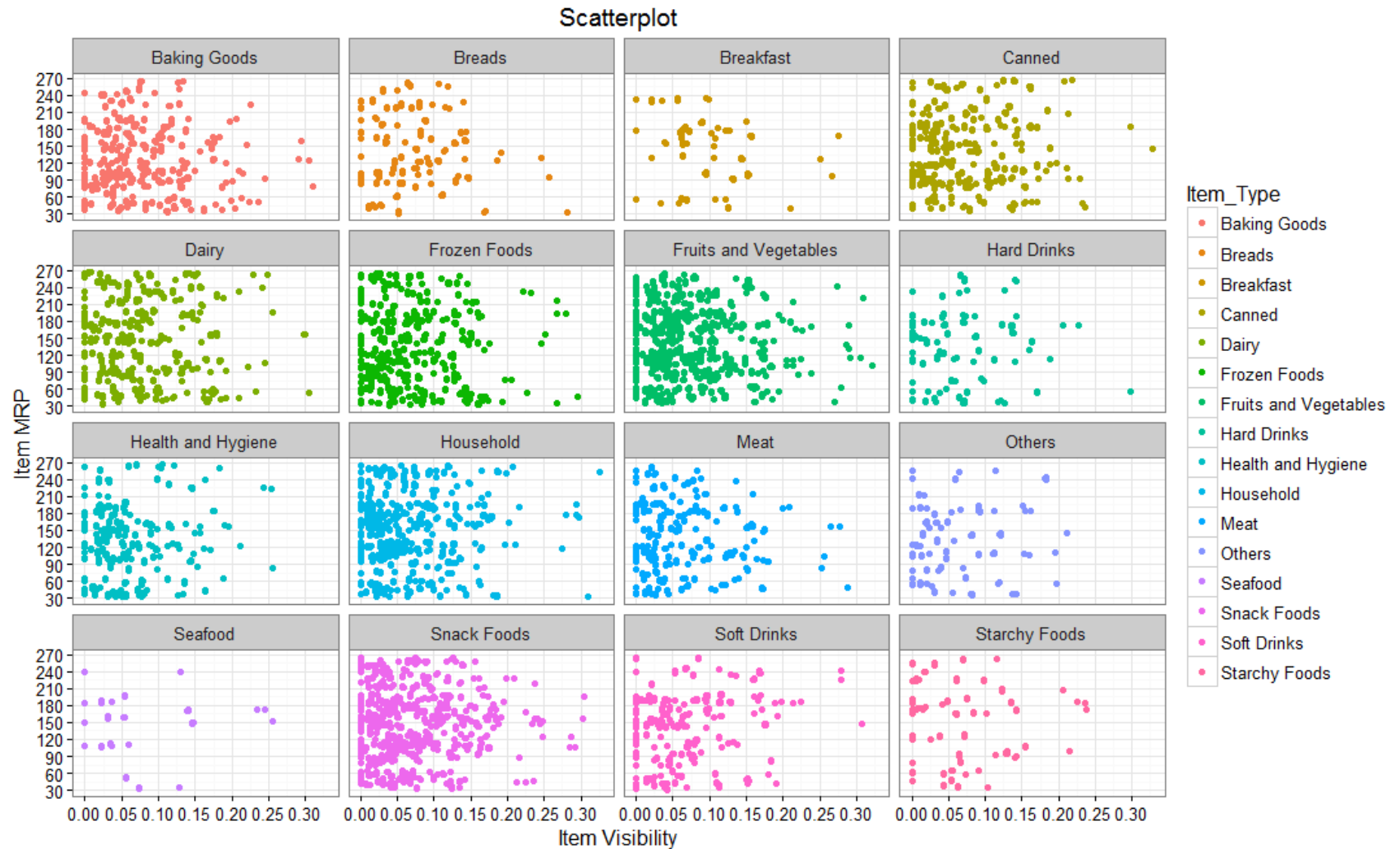
DATA VISUALIZATION

1. Scatter Plot: We can even make it more visually clear by creating separate scatter plots for each separate Item_Type as shown below.

Another scatter plot using function ggplot() with geom_point().

➢ library(ggplot2)

➢ ggplot(train, aes(Item_Visibility, Item_MRP)) + geom_point(aes(color = Item_Type)) +   scale_x_continuous("Item Visibility", breaks = seq(0,0.35,0.05))+ scale_y_continuous("Item MRP", breaks = seq(0,270,by = 30))+   theme_bw() + labs(title="Scatterplot") + facet_wrap(~ Item_Type)

Here, facet_wrap works well & wraps Item_Type in rectangular layout.

DATA VISUALIZATION


Scatterplot

DATA VISUALIZATION

2. Histogram: It is used to plot continuous variable. It breaks the data into bins and shows frequency distribution of these bins. We can always change the bin size and see the effect it has on visualization.

For Big_Mart_Dataset, if we want to know the count of items on basis of their cost, then we can plot histogram using continuous variable Item_MRP as shown below.

Histogram plot using function ggplot() with geom_ histogram()

```
> ggplot(train, aes(Item_MRP)) + geom_histogram(binwidth = 2)+
scale_x_continuous("Item MRP", breaks = seq(0,270,by = 30))+
scale_y_continuous("Count", breaks = seq(0,200,by = 20))+  labs(title =
"Histogram")
```
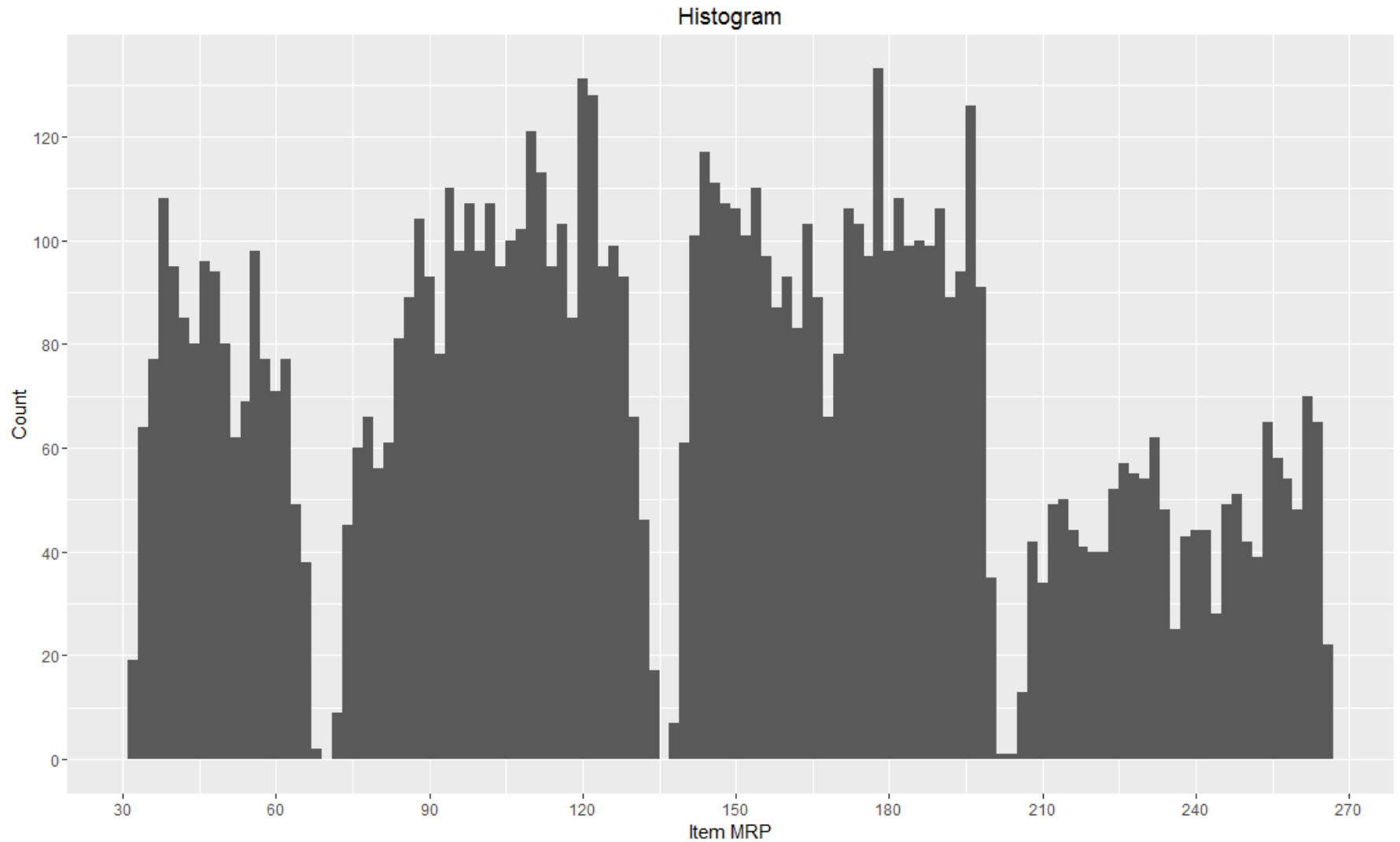
DATA VISUALIZATION

3. Bar Chart: It is used when you want to plot a categorical variable or a combination of continuous and categorical variable.
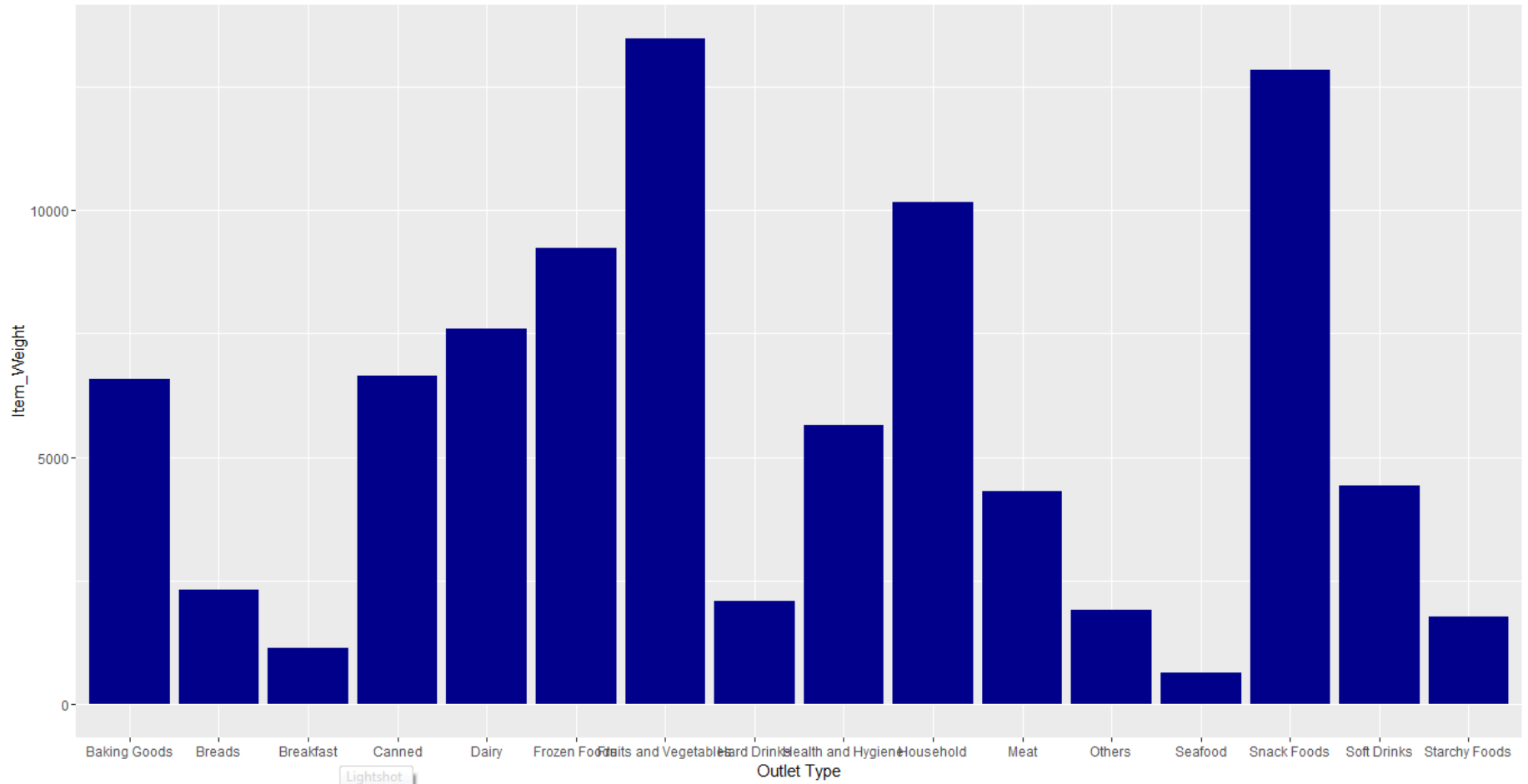
For Big_Mart_Dataset, if we want to know item weights (continuous variable) on basis of Outlet Type (categorical variable) on single bar chart as shown below.

Vertical Bar plot using function ggplot()

```
> ggplot(train, aes(Item_Type, Item_Weight)) + geom_bar(stat = "identity", fill = "darkblue") + scale_x_discrete("Outlet Type")+ scale_y_continuous("Item Weight", breaks = seq(0,15000, by = 500))+ theme(axis.text.x = element_text(angle = 90, vjust = 0.5)) + labs(title = "Bar Chart")
```

DATA VISUALIZATION

DATA VISUALIZATION

3. Stack Bar Chart: It is an advanced version of bar chart, used for visualizing a combination of categorical variables.

For Big_Mart_Dataset, if we want to know the count of outlets on basis of categorical variables like its type (Outlet Type) and location (Outlet Location Type) both, stack chart will visualize the scenario in most useful manner.

Stack Bar Chart using function ggplot()

```
> ggplot(train, aes(Outlet_Location_Type, fill = Outlet_Type)) +
geom_bar()+labs(title = "Stacked Bar Chart", x = "Outlet Location Type", y =
"Count of Outlets")
```

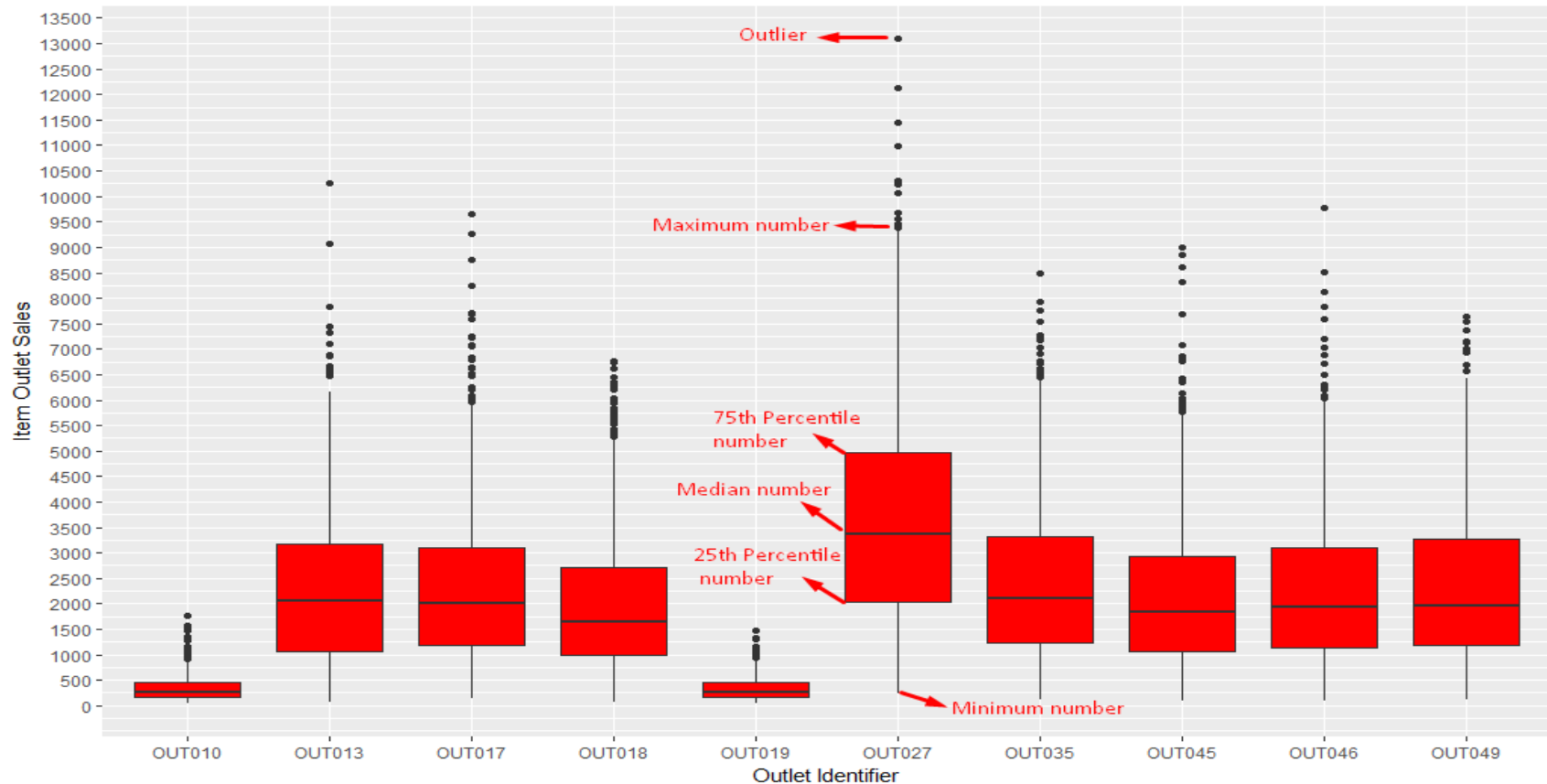# DATA VISUALIZATION
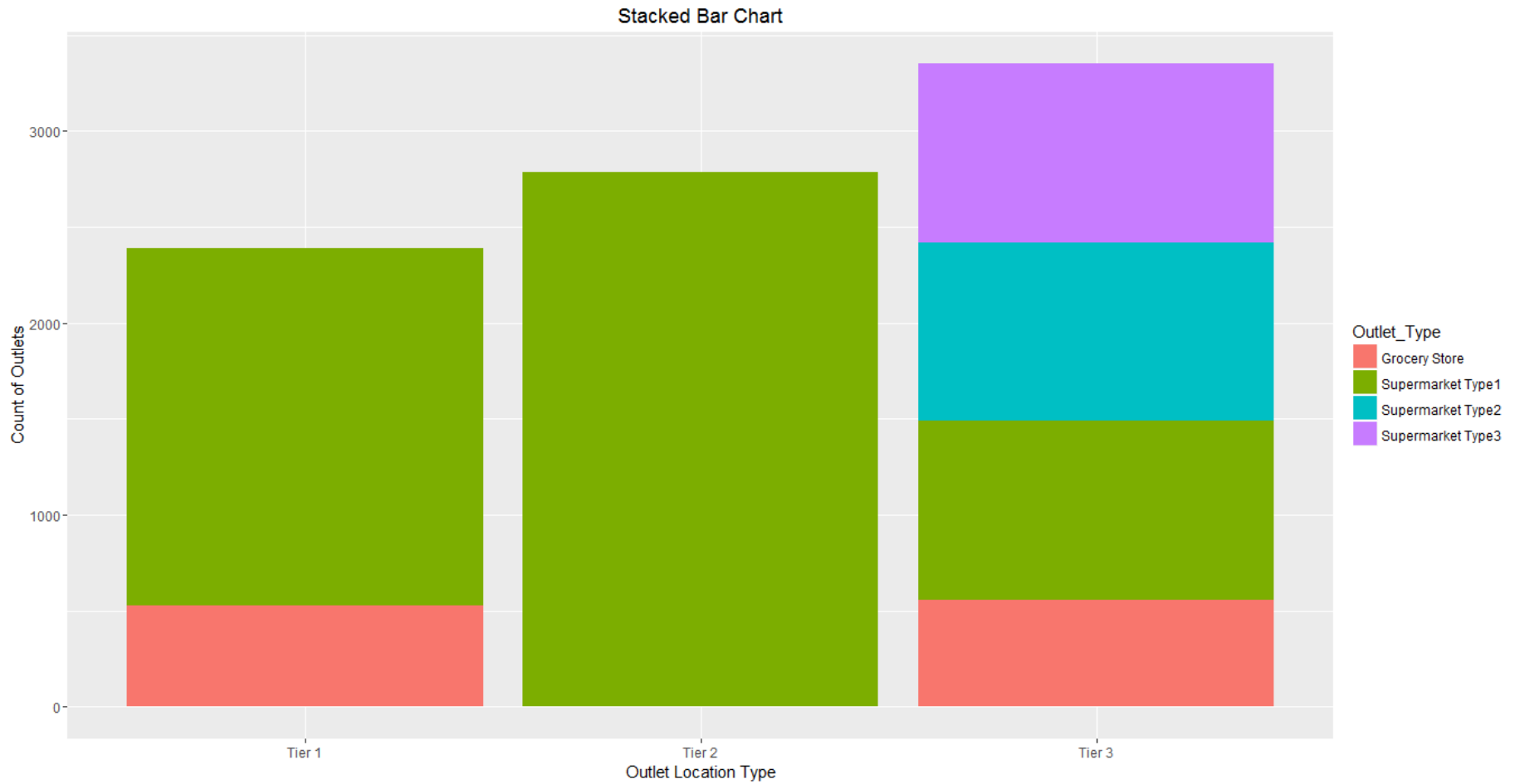
DATA VISUALIZATION

4. Box Plot: It is used to plot a combination of categorical and continuous variables. This plot is useful for visualizing the spread of the data and detect outliers. It shows five statistically significant numbers the minimum, the 25th percentile, the median, the 75th percentile and the maximum.

For Big_Mart_Dataset, if we want to identify each outlet's detailed item sales including minimum, maximum & median numbers, box plot can be helpful. In addition, it also gives values of outliers of item sales for each outlet as shown in below chart.

R Code:

```
> ggplot(train, aes(Outlet_Identifier, Item_Outlet_Sales)) + geom_boxplot(fill = "red")+scale_y_continuous("Item Outlet Sales", breaks= seq(0,15000, by=500))+labs(title = "Box Plot", x = "Outlet Identifier")
```

DATA VISUALIZATION

The black points are outliers. Outlier detection and removal is an essential step of successful data exploration.

DATA VISUALIZATION

5. Area Chart: It is used to show continuity across a variable or data set. It is very much same as line chart and is commonly used for time series plots. Alternatively, it is also used to plot continuous variables and analyse the underlying trends.
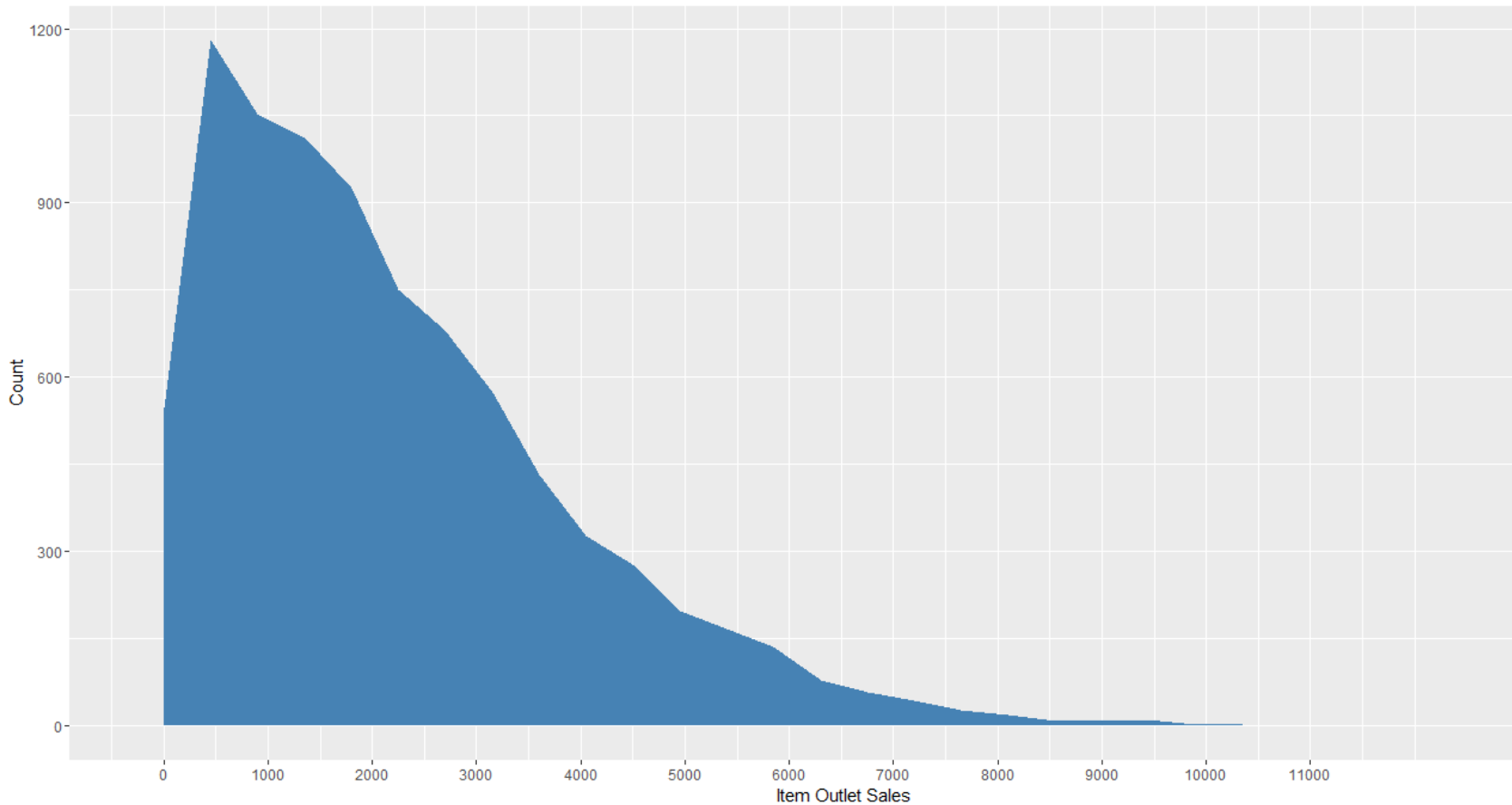
For Big_Mart_Dataset, when we want to analyse the trend of item outlet sales, area chart can be plotted as shown below. It shows count of outlets on basis of sales.

R Code:

> ggplot(train, aes(Item_Outlet_Sales)) + geom_area(stat = "bin", bins = 30, fill = "steelblue") + scale_x_continuous(breaks = seq(0,11000,1000))+ labs(title = "Area Chart", x = "Item Outlet Sales", y = "Count")

DATA VISUALIZATION

Area chart shows continuity of Item Outlet Sales using function ggplot() with geom_area.

DATA VISUALIZATION

6. Heat Map: It uses intensity (density) of colours to display relationship between two or three or many variables in a two dimensional image.

For Big_Mart_Dataset, if we want to know cost of each item on every outlet, we can plot heatmap as shown below using three variables Item MRP, Outlet Identifier & Item Type from our mart dataset.

R Code:

```
> ggplot(train, aes(Outlet_Identifier, Item_Type))+  geom_raster(aes(fill = Item_MRP))+  labs(title ="Heat Map", x = "Outlet Identifier", y = "Item Type")+ scale_fill_continuous(name = "Item MRP")
```

DATA VISUALIZATION
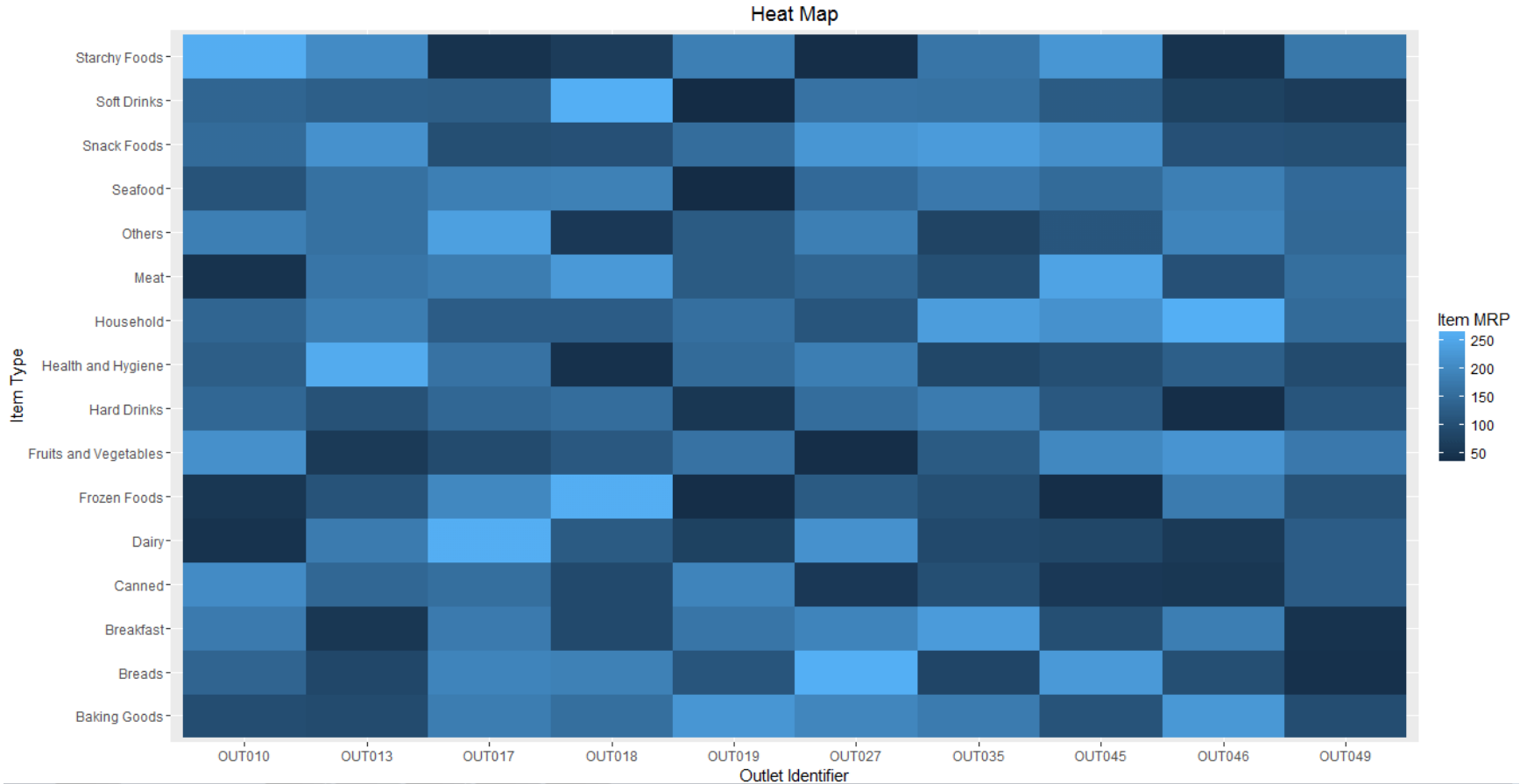
The dark portion indicates Item MRP is close 50.
The brighter portion indicates Item MRP is close to 250.

DATA VISUALIZATION

7. Correlogram: It is used to test the level of co relation among the variable available in the data set. The cells of the matrix can be shaded or coloured to show the co relation value.

For Big_Mart_Dataset, check co relation between Item cost, weight, visibility along with Outlet establishment year and Outlet sales from below plot.

R Code for simple correlogram using function corrgram():

```
> install.packages("corrgram")

> library(corrgram)

> corrgram(train, order=NULL, panel=panel.shade, text.panel=panel.txt,
main="Correlogram")
```
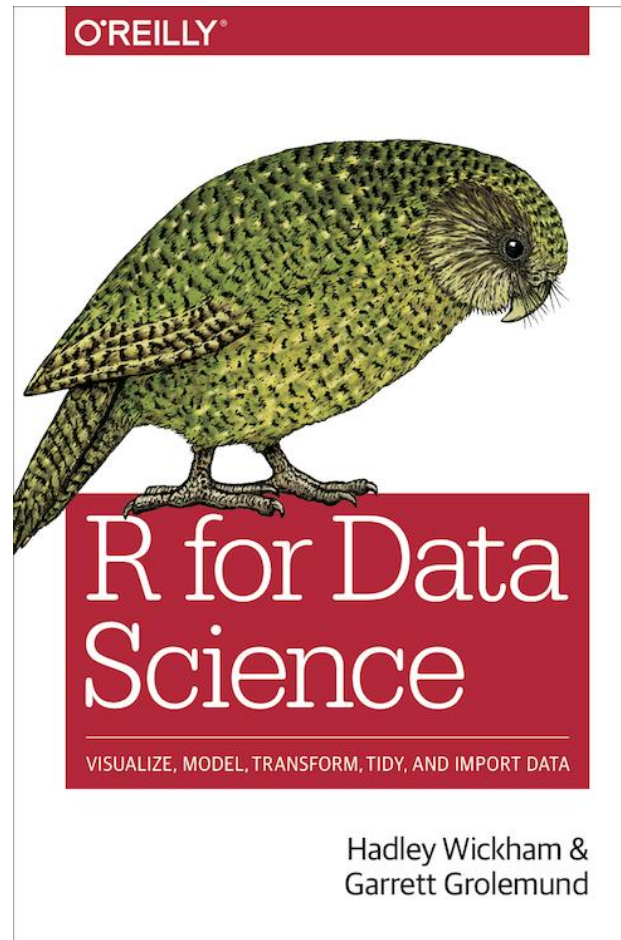
DATA VISUALIZATION

Correlogram

- Darker the colour, higher the co relation between variables. Positive co relations are displayed in blue and negative correlations in red colour. Colour intensity is proportional to the co relation value.
- We can see that Item cost & Outlet sales are positively correlated while Item weight & its visibility are negatively correlated.

Reference

https://r4ds.had.co.nz/