**tapply**

is similar to GROUP BY with an aggregate function in SQL

First argument: the numerical variable you want to **apply** the aggregate function

Second argument: the categorical variable you want to aggregate **by**

Third argument: the aggregate function you want to **apply**

tapply(mtcars$mpg,mtcars$carb,mean)

1 2 3 4 6 8

25.34286 22.40000 16.30000 15.79000 19.70000 15.00000

**by multiple categorical variable**

you can supply multiple categorical variables to the sapply function in the form of a *list*:

translation: average mpg BY cylinder(cyl) and gears(gear):

tapply(mtcars$mpg,list(mtcars$cyl,mtcars$gear),mean,na.nm=TRUE)

3 4 5

4 21.50 26.925 28.2

**Gear**

6 19.75 19.750 19.7

8 15.05 NA 15.4

**Cylinders**

#merge the CPS dataset with the MetroAreaMap

CPS = merge(CPS, MetroAreaMap, by.x="MetroAreaCode", by.y="Code", all.x=TRUE)

First argument: first data frame to combine (implied alias of “x”)

Second argument: second data frame to combine (implied alias of “y”)

Third argument: variable in first data frame to merge by. **This is the common field to both data frames**

Fourth argument: variable in second data frame to merge by. **This is the common field to both data frames**

Fifth argument (all.x=TRUE) – analogous to a JOIN type in SQL (inner, outer)

If TRUE will return ALL rows in the x alias, regardless of a match (same a LEFT OUTER JOIN in SQL)

Analyics edge

use abline to create

plot(CocaCola$Date[301:432], CocaCola$StockPrice[301:432], type="l", col="red", ylim=c(0,210))

lines(ProcterGamble$Date, ProcterGamble$StockPrice,col="Blue")

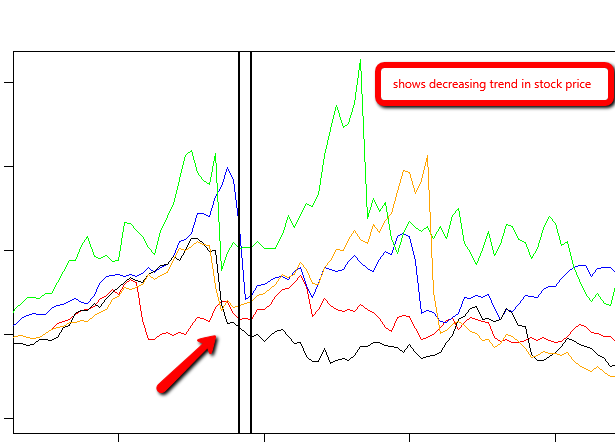
lines(IBM$Date, IBM$StockPrice,col="Green")

lines(Boeing$Date, Boeing$StockPrice,col="Black")

lines(GE$Date, GE$StockPrice,col="Orange")

abline(v=as.Date(c("1997-09-01")), lwd=2)

abline(v=as.Date(c("1997-11-01")), lwd=2)



**week 1 R**

**Command FAQ**

|  |  |
| --- | --- |
| Get working directory | getwd()  > getwd()  [1] "D:/win7docs" |
| list objects in workspace | > ls()  [1] "complx" "x" "y" |
| list all files in working directory | dir() OR  list.files() |
| set working directory | setwd(“path”) |
| get information about a file | file.info("mytest.R")  size isdir mode mtime ctime atime exe  mytest.R 0 FALSE 666 2016-02-28 21:11:56 2016-02-28 21:11:56 2016-02-28 21:11:56 no |
|  |  |

**todo**

* read through some in the “Some R resources” slide in Overview and HIstory of R
* and also the “Some Useful Books on S/R” slide
* read “how to ask questions the smart way”
* load best R manuals into dropbox so you can read and annotate in goodreader
* **ORDER A STYLUS!!!!**

S language developed at Bell Labs.

R is an implementation of the S language..

about 4,000 packages developed by users around the world

CRAN

* has a number a quality standards: documentation, passes certain number of tests

Bioconductor project - genomic and biological data analysis

**Help**

discussion boards

email

answer sources:

manual

FAQ

goodle

**checklist when asking questions:**

* say what you did (FAQ, read manual, google, forum)
* reproduce problem
* version of R running
* which OS
* additional information

**Places to turn**

* class discussion forum
* [r-help@r-project.org](mailto:r-help@r-project.org)

**R OBJECTS**

**“Atomic” classes – the lowest level**

* character
* numeric (Real numbers)
* integer
* complex
* logical (true/false)

most basic object is a **vector**

**Vector**

* a vector can only contain objects of the **same class**
* EXCEPTION - a **list** contain objects of different classes

**List**

Any element of a list can be anything

* A list within a list

create an empty vector: use the **vector function**

**Example**

Creates an empty vector z or numeric objects, with a length of 100:

z<-vector(mode="numeric", length=100)

|  |
| --- |
| > ##creates empty vector  > z<-vector(mode="numeric", length=10)  > str(z)  num [1:10] 0 0 0 0 0 0 0 0 0 0  > z  [1] 0 0 0 0 0 0 0 0 0 0 |
|  |
| |  | | --- | | > | |

**Numbers**

* Number default to numeric objects (real numbers) as opposed to integers
* Explicit integer: use the L suffix
* Inf – represents *infinity*
* NaN – undefined value “non a number”

**Attributes**

* R objects can have an attribute
* Not every object have them
* names, dimnames
* dimensions (e.g. matrices, arrays)
* class
* length
* user-defined

Attributes can be accessed using the attributes() function

**console:**

> x<-5

> str(x)

num 5

> y<-5L

> str(y)

int 5

The output type is determined from the highest type of the components in the hierarchy

NULL < raw < logical < integer < double < complex < character < list < expression

^^^^^^^^^^ ^^^^^^^^^^^

**lowest on tree highest on tree**

**SO,**

list trumps logical

character trumps logical

integer trumps logical

logical trumps null,

and so on

> y<-c(1.7,'x') ##character vector

> str(y)

chr [1:2] "1.7" "x"

> y<-c(TRUE,1) ##numeric vector

> str(y)

num [1:2] 1 1

> y<-c("x",TRUE) ##character vector

> str(y)

chr [1:2] "x" "TRUE"

# **Coercion**

When you call a function with an argument of the wrong type, R will try to coerce values to a different type so that the function will work. There are two types of coercion that occur automatically in R: coercion with formal objects and coercion with built-in types.

With generic functions, R will look for a suitable method. If no exact match exists, then R will search for a coercion method that converts the object to a type for which a suitable method does exist. (The method for creating coercion functions is described in [Creating Coercion Methods](https://www.safaribooksonline.com/library/view/r-in-a/9781449358204/ch10s02.html#coercion-methods).)

Additionally, R will automatically convert between built-in object types when appropriate. R will convert from more specific types to more general types. For example, suppose that you define a vector x as follows:

> **x <- c(1, 2, 3, 4, 5)**  
> **x**  
[1] 1 2 3 4 5  
> **typeof(x)**  
[1] "double"  
> **class(x)**  
[1] "numeric"

Let’s change the second element of the vector to the word “hat.” R will change the object class to character and change all the elements in the vector tochar:

> **x[2] <- "hat"**  
> **x**  
[1] "1" "hat" "3" "4" "5"   
> **typeof(x)**  
[1] "character"  
> **class(x)**  
[1] "character"

Here is an overview of the coercion rules:

* Logical values are converted to numbers: TRUE is converted to 1 and FALSEto 0.
* Values are converted to the simplest type required to represent all information.
* The ordering is roughly logical < integer < numeric < complex < character < list.

**nonsensical coercion**

You cannot convert certain object from one to another

* a character cannot convert to an integer
* a character cannot convert to a logical
* will result in error message “NAs introduced by coercion”

> ##explicit coerciion

> ##nonsensical results

> x<-c("a","b","c")

> as.numeric(x)

[1] NA NA NA

Warning message:

NAs introduced by coercion

> as.logical(x)

[1] NA NA NA

> as.complex(x)

[1] NA NA NA

Warning message:

NAs introduced by coercion

**Lists**

> x<-list(1,"a",TRUE,1+4i)

> x

[[1]] <<<<<<< **elements of a list will have double brackets around them**

[1] 1

[[2]]

[1] "a"

[[3]]

[1] TRUE

[[4]]

[1] 1+4i

**Matrices**

* special type of vector in R
* vectors that have an attribute called a *dimension*

Create matrix by using the **matrix** function:

- Creates an empty matrix of 2 rows and 3 columns

- dim(m) will return the dimension attribute where the

* first number – number of rows
* second number – number of columns

-attributes() function will return a ***list*** where the first element is the dim element, which has the vector 2,3

|  |
| --- |
| > ##matrices  > m<-matrix(nrow=2,ncol=3)  > m  [,1] [,2] [,3]  [1,] NA NA NA  [2,] NA NA NA  > dim(m)  [1] 2 3  > attributes(m)  $dim  [1] 2 3 |
|  |
| |  | | --- | | > | |

**Matrices**

* are a vector with a dimension attribute

**Creating and populating a matrix**

* is populated *column-wise*
* this means *down* then *across*
* the below code shows the creation of a matrix, filling with the integer sequence 1-6, with 2 rows and 3 columns

|  |
| --- |
| > ##creating and populating a matrix  > m<-matrix(1:6,nrow=2,ncol=3) <<<<< shows a new argument to the matrix function  > m  [,1] [,2] [,3]  [1,] 1 3 5  [2,] 2 4 6 |
| **Creating a matrix by CONVERTING a vector into a matrix:** |
| |  | | --- | |  | |
| > ##another way of creating a matrix  > m<-1:10  > ##this creates a VECTOR of numbers 1-10:  > m  [1] 1 2 3 4 5 6 7 8 9 10  > ##call the dim function to the vector  > ##dim(m)<-c(2,5) will CONVERT the vector into  > ##a matrix of 2 rows and 5 columns  > dim(m)<-c(2,5)  > m  [,1] [,2] [,3] [,4] [,5]  [1,] 1 3 5 7 9  [2,] 2 4 6 8 10  **Creating a matrix by using cbind or rbind**   * cbind will fill the matrix columns-first * rbind will fill the matrix rows-first   > ##yet another way of creating a matrix  > x<-1:3  > y<-10:12  > cbind(x,y)  x y  [1,] 1 10  [2,] 2 11  [3,] 3 12  > rbind(x,y)  [,1] [,2] [,3]  x 1 2 3  y 10 11 12 |
|  |
| |  | | --- | | > | |

**Factors**

1. used to represent categorical data
2. can be ordered an unordered
   1. ordered (also called *ordinal* in statistics). Examples:
      1. levels of education : high school, some college, 4 years college, ect
      2. ratings of a restaurant 1-10
      3. income levels 0-$10,000,>$10,000-20,000 ect
   2. unordered: states, sex (M/F)
3. **IMPORTANT –** used for modeling functions in R, graphing as well

> ##factors

> x<-factor(c("yes","no","no","yes","no"))

> x

[1] yes no no yes no

Levels: no yes

> table(x) ##gives frequency count

x

no yes

3 2

>

> unclass(x) ##strips out the class of a vector

[1] 2 1 1 2 1

attr(,"levels")

[1] "no" "yes"

> - ##brings down to an integer vector

> ##represented INTERNALLY by R as 1's and 2's

> x

[1] yes no no yes no

Levels: no yes

> str(x)

Factor w/ 2 levels "no","yes": 2 1 1 2 1

**Setting levels explicitly with the levels argument**

|  |
| --- |
| > ##orders of levels can be set using the levels argument of the factor() function  >  > x<-factor(c("yes","yes","no","yes","no"),  + levels=c("yes","no"))  > ##in x, "yes" is the baseline due to the explicit declaration  > ## using the levels argument  > x  [1] yes yes no yes no  Levels: yes no  > nolevel<-factor(c("yes","yes","no","yes","no"))  > ##without the levels argument, "no" is the baseline  > ##due to alphabetical order  > nolevel  [1] yes yes no yes no  Levels: no yes |
|  |
| |  | | --- | |  | |

**Missing values**

* Is

**Data frames**

* Special type of list – seems similar to database tables
* Data frames can store different objects with different classes, **unlike a matrix**
* Usually created with read.table(), read.csn

|  |
| --- |
| > ##Data frame - used to store tabular data  > ##a special type of list  > ##each element - column - each  > x<-data.frame(foo=1:4, bar=c(T,T,F,F))  > ##prints out frame, with now row names, as none were specified  > x  foo bar  1 1 TRUE  2 2 TRUE  3 3 FALSE  4 4 FALSE  > nrow(x)  [1] 4  > ncol(x)  [1] 2 |
|  |
| |  | | --- | | > | |

**Names**

* R objects can have names – useful for writing reusable code and self-describing objects

> ##lists can also have names

> ##create list - 1st element called 1

> ##second element called b

> ##third element is called c

> x<-list(a=1,b=3,c=3)

> ##printout shows names of each element and the

> ##values associated with those names

> x

$a

[1] 1

$b

[1] 3

$c

[1] 3

**Reading data**

|  |  |  |
| --- | --- | --- |
| **Read** | **Write** | **Description** |
| read.table read.csv | write.table | reading tabular data |
| readLines | writeLines | reading lines of a text file |
| source | dump | reading R code files |
| Dget | dput | reading R code files? |
| load | save | reading in saved workspaces |
| unserialize | serialize | reading R objects in serial form |

**read.table**

<insert full description from Peng’s presentation>

* file – the name of a file
* header – does first line have variable names
* *sep*, delimiter – comma, tab
* colClasses – character vector – specify the class of each object in the data **NOT REQUIRED**
* nrows – number of rows in dataset **NOT REQUIRED**
* comment.char – character string indicating comment character – default is # character – you can specify other characters
* skip - # of lines to skip – if there is non-data information you can skip
* stringsAsFactors, defaults as true.. if a value is character – it defaults as character

R automatically

* skips lines that begin with a #
* figures out how many rows
* determine type of variable . **telling R directly is more efficient and faster**
* read.csv is identical to read.table, except default separator is a comma
* default separator in read.table is space
* defaults to header=true

**Larger datasets**

* read help page for read.table
* calculate how large the input file is – if larger than memory in computer, stop
* use the colClasses argument. – MUCH FASTER
* know system
  + OS 32 or 64 – 64 access more memory

**Calculating memory**

* will need more physical memory than this calculation requires
* **double** the memory is a rule of thumb

data frame 1,500,000 rows, 120 columns, all numeric

**formula:**

rows \* columns \* storage per field

1,500,000 x 120 x 8 bytes/numeric

=1,440,000,000 bytes

=1,440,000,000 / 220  bytes/MB

=1,373.29 MB

=1.34 GB

> initial<-read.csv("mtcars.csv",nrows=10)

> #sapply - loops over the columns

> # while calling the class function

> # the class function will tell you what class

> # of data is in each column

> classes<-sapply(initial,class)

> tabAll<-read.csv("mtcars.csv",

+ colClasses=classes)

> initial

X mpg cyl disp hp drat wt qsec vs am gear carb

1 Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

2 Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

3 Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

4 Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

5 Hornet Sportabout 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2

6 Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

7 Duster 360 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4

8 Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

9 Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2

10 Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

> classes

X mpg cyl disp hp drat wt

"factor" "numeric" "integer" "numeric" "integer" "numeric" "numeric"

qsec vs am gear carb

"numeric" "integer" "integer" "integer" "integer"

> tabAll

X mpg cyl disp hp drat wt qsec vs am gear carb

1 Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

2 Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

3 Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

4 Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

5 Hornet Sportabout 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2

6 Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

7 Duster 360 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4

8 Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

9 Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2

10 Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

11 Merc 280C 17.8 6 167.6 123 3.92 3.440 18.90 1 0 4 4

**Textual formats**

* dumping and dputing
* preserve the *metadata* of the data
* better for version control – like Subversion or Git
* can be longer-lived
* adhere to the “Unix” philosophy
* Downside: not space efficient

**dput-ting R Objects**

> ##dput-ting writes out data AND metadata to a file

> y<-data.frame(a=1,b="a")

> dput(y)

structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), .Names = c("a",

"b"), row.names = c(NA, -1L), class = "data.frame")

> dput(y,file="y.R") ##saves structure to a file - writes R code which can

> ##reconstruct an R object

> new.y<-dget("y.R")

> new.y

a b

1 1 a

**Dumping R Objects**

Multiple objects can be deparsed using the dump function and read back in using source

|  |
| --- |
| > #dumping R objects - can save MULTIPLE R objects  > # create two objects, x and y, dump to file, remove, then  > # read back in  > x<-"foo"  > y<-data.frame(a=1,b="a")  > dump(c("x","y"),file="datadump.R")  > rm(x,y) #removes objects from memory  > source("datadump.R")  > y  a b  1 1 a  > x  [1] "foo" |
|  |
| |  | | --- | | > | |

**Interfaces to outside world:**

connections *abstract out*the specifics of connecting to the specific data sources

* file, opens a connection to a file
* gzfile, opens a connection to a file compressed with gzip
* bzfile, opens a connection to a file compressed with bzip2
* url, opens a connection to a webpage

**File Connections**

> str(file)

function (description = "", open = "", blocking = TRUE, encoding = getOption("encoding"),

raw = FALSE)

* description – name of file
* open is a code indicating
* “r” read only
* “w” writing and initializing a new file
* “a” appending
* “rb”,”wb”,”ab” reading, writing, or appending in binary mode
* **see help for details**

**Connections**

* in many cases you do not need the connection
* example:

#connections

con<-file("mtcars.csv")

data=read.csv(con)

close(con)

#is the same as

data<-read.csv("mtcars.csv")

**Reading from a URL**

##using readlines to read from web page

> con<-url("http://www.jhsph.edu","r")

> x<-readLines(con)

> head(x)

[1] "<!DOCTYPE html>"

[2] "<html lang=\"en\">"

[3] ""

[4] "<head>"

[5] "<meta charset=\"utf-8\" />"

[6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"

**Subsetting**

* single bracket: [ always returns an object as the same class as the original; can be used for more than one element
* double bracket: [[ - to extract elements of a list or a data frame; **it can only be used to extract a single element and the class**
* **s of the returned object will not necessarily be a list or data frame**
* $ is used to extract elements of a list or data frame **by name;** semantics are similar to that of [[

> ##subsetting in R

> #subsetting with a numeric index

> x<-c("a","b","c","c","d","a")

> x[1] #returns character vector with the element a

[1] "a"

> x[2] #return character vector with the element b

[1] "b"

> x[1:4]

[1] "a" "b" "c" "c"

>

> #subsetting with logical index

> x[x>"a"] #

show me only letters greater than a

[1] "b" "c" "c" "d"

> #returns a character vector with values greater than a

>

> #create a logical vector

> u<-x>"a"

> u

[1] FALSE TRUE TRUE TRUE TRUE FALSE

> x[u] ##subset with a LOGICAL index

[1] "b" "c" "c" "d"

**Subsetting lists**

The [[ operator can be using with *computer indices; $*  can only be used with literal names

> x<-list(foo=1:4,bar=0.6, baz="hello")

> name<-"foo"

>

> x[[name]] ##computed index for 'foo'

[1] 1 2 3 4

> ##looking for element call 'foo'

>

> x$name ##element 'name' does not exist

NULL

> x$foo ##element 'foo' does exist

[1] 1 2 3 4

> x<-matrix(1:6, 2,3)

> ##default behavior is a vector is returned

> ##in this case a vector of length 1

> x[1,2]

[1] 3

> ##1,

> x[1,,drop=FALSE]

[,1] [,2] [,3]

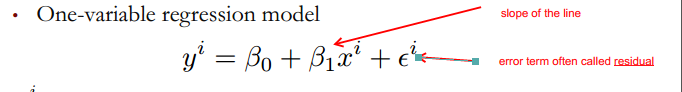
[1,] 1 3 5

|  |
| --- |
| > ##partial matching  > ##good for working on the command line  > x<-list(aardvark=1:5)  > x$a  [1] 1 2 3 4 5  >  > ##double bracket does not use partial matching by default  > ##but you can change this with the exact argument  > ##double bracket alone:  > x[["a"]]  NULL  > ##with exact argument  > x[["a",exact=FALSE]]  [1] 1 2 3 4 5  **Vectorized operations**   * can do parallel mathematical operations on vectors   > ##vectorized operations  > ##languages like matlab have these features as well  > x<-1:4;y<-6:9  > x  [1] 1 2 3 4  > y  [1] 6 7 8 9  > ##returns logical vector is the element >2?  > x>2  [1] FALSE FALSE TRUE TRUE  > x>=2  [1] FALSE TRUE TRUE TRUE  > y==8  [1] FALSE FALSE TRUE FALSE  > x\*y  [1] 6 14 24 36  > x/y  [1] 0.1666667 0.2857143 0.3750000 0.4444444 |
|  |
| |  | | --- | | > | |

|  |
| --- |
| > ##vectorizing matrix operations  > x<-matrix(1:4,2,2); y<-matrix(rep(10,4),2,2)  > x;y  [,1] [,2]  [1,] 1 3  [2,] 2 4  [,1] [,2]  [1,] 10 10  [2,] 10 10  > ##element wise multiplication  > x\*y  [,1] [,2]  [1,] 10 30  [2,] 20 40  > x/y  [,1] [,2]  [1,] 0.1 0.3  [2,] 0.2 0.4  >  > ##true matrix multiplication  > x;y  [,1] [,2]  [1,] 1 3  [2,] 2 4  [,1] [,2]  [1,] 10 10  [2,] 10 10  > x %\*% y  [,1] [,2]  [1,] 40 40  [2,] 60 60 |
|  |
| |  | | --- | | > | |

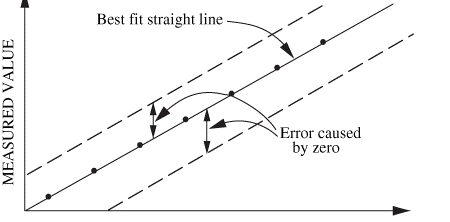
**Week 2**

**Linear regression (one variable)**

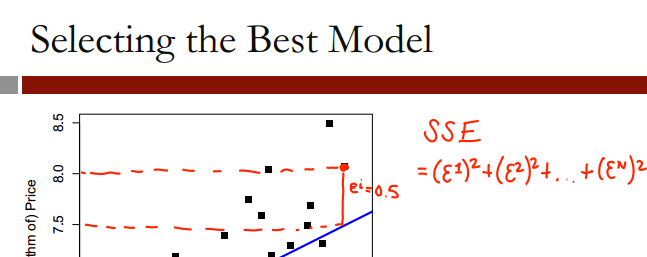


**\*\*\*\***error coefficient would be zero where the line fit the data points perfectly

see best fit straight line:



\*\* One measure of the quality of a regression line is Sum of Squared Errors, or SSE:



> model1=lm(Price ~AGST, data=wine)

> summary(model1)

Call:

lm(formula = Price ~ AGST, data = wine)

Summary of error terms

Residuals:

Min 1Q Median 3Q Max

-0.78450 -0.23882 -0.03727 0.38992 0.90318

AGST row is independent variable – avg growing season temperature

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -3.4178 2.4935 -1.371 0.183710

AGST 0.6351 0.1509 4.208 0.000335 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.4993 on 23 degrees of freedom

Multiple R-squared: 0.435, Adjusted R-squared: 0.4105

F-statistic: 17.71 on 1 and 23 DF, p-value: 0.000335

**SSE**  for model 1: 5.734875

multiple R – squared – R2 value for model

adjusted R – squared – R2 value for model

**adjusted R squared will decrease if a variable is added to model that doesn’t help**

SSE

[1] 5.734875

**model 2**

> summary(model2)

Call:

lm(formula = Price ~ AGST + HarvestRain, data = wine)

Residuals:

Min 1Q Median 3Q Max

-0.88321 -0.19600 0.06178 0.15379 0.59722

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -2.20265 1.85443 -1.188 0.247585

AGST 0.60262 0.11128 5.415 1.94e-05 \*\*\*

HarvestRain -0.00457 0.00101 -4.525 0.000167 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3674 on 22 degrees of freedom

Multiple R-squared: 0.7074, Adjusted R-squared: 0.6808

F-statistic: 26.59 on 2 and 22 DF, p-value: 1.347e-06

**Notes:**

multiple R-squared increased from .435 to .7074

Adjusted R-squared increased from .4105 to .6808

SSE decreased from 5.734875 to 2.970373

SSE

[1] 2.970373

**Adding this variable significantly helped the model**

**model 3**

summary(model3)

Call:

lm(formula = Price ~ AGST + HarvestRain + WinterRain + Age +

FrancePop, data = wine)

Residuals:

Min 1Q Median 3Q Max

-0.48179 -0.24662 -0.00726 0.22012 0.51987

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -4.504e-01 1.019e+01 -0.044 0.965202

AGST 6.012e-01 1.030e-01 5.836 1.27e-05 \*\*\*

HarvestRain -3.958e-03 8.751e-04 -4.523 0.000233 \*\*\*

WinterRain 1.043e-03 5.310e-04 1.963 0.064416 .

Age 5.847e-04 7.900e-02 0.007 0.994172

FrancePop -4.953e-05 1.667e-04 -0.297 0.769578

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3019 on 19 degrees of freedom

Multiple R-squared: 0.8294, Adjusted R-squared: 0.7845

F-statistic: 18.47 on 5 and 19 DF, p-value: 1.044e-06

**Notes model 3 compared to 2:**

multiple R-squared increased from .7074 to .8294

Adjusted R-squared increased from .6808 to .7845

SSE decreased from 2.970373 to 1.732113

SSE

[1] 1.732113

**model 4 – using significant variables**

summary(model4)

Call:

lm(formula = Price ~ AGST + HarvestRain + WinterRain + Age, data = wine)

Residuals:

Min 1Q Median 3Q Max

-0.45470 -0.24273 0.00752 0.19773 0.53637

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -3.4299802 1.7658975 -1.942 0.066311 .

AGST 0.6072093 0.0987022 6.152 5.2e-06 \*\*\*

HarvestRain -0.0039715 0.0008538 -4.652 0.000154 \*\*\*

WinterRain 0.0010755 0.0005073 2.120 0.046694 \*

Age 0.0239308 0.0080969 2.956 0.007819 \*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.295 on 20 degrees of freedom

Multiple R-squared: 0.8286, Adjusted R-squared: 0.7943

F-statistic: 24.17 on 4 and 20 DF, p-value: 2.036e-07

Moneyball – predicting wins by run differential (variable RD)

winsreg=lm(W~RD,moneyball)

> summary(winsreg)

Call:

lm(formula = W ~ RD, data = moneyball)

Residuals:

Min 1Q Median 3Q Max

-14.2662 -2.6509 0.1234 2.9364 11.6570

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 80.881375 0.131157 616.67 <2e-16 \*\*\*

RD 0.105766 0.001297 81.55 <2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.939 on 900 degrees of freedom

Multiple R-squared: 0.8808, Adjusted R-squared: 0.8807

F-statistic: 6651 on 1 and 900 DF, p-value: < 2.2e-16

**This shows:**

1. strong model due to the R squared value of .8807

**Predicting Wins by OBP, SLG, and BA**

> summary(RunsReg)

Call:

lm(formula = RS ~ OBP + SLG + BA, data = moneyball)

Residuals:

Min 1Q Median 3Q Max

-70.941 -17.247 -0.621 16.754 90.998

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -788.46 19.70 -40.029 < 2e-16 \*\*\*

OBP 2917.42 110.47 26.410 < 2e-16 \*\*\*

SLG 1637.93 45.99 35.612 < 2e-16 \*\*\*

BA -368.97 130.58 -2.826 0.00482 \*\*

---

Signif. codes:

0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 24.69 on 898 degrees of freedom

Multiple R-squared: 0.9302, Adjusted R-squared: 0.93

F-statistic: 3989 on 3 and 898 DF, p-value: < 2.2e-16

**Interpretation:**

* the BA (batting average) coefficient tells us that teams with a LOWER batting average will score more runs. **This is counter-intuitive**, and the cause is that the three predictor variables are highly correlated

This correlation is proven by the following analysis in R:

> moneynew=data.frame(moneyball$BA,moneyball$SLG,moneyball$OBP)

> cor(moneynew)

moneyball.BA moneyball.SLG moneyball.OBP

moneyball.BA 1.0000000 0.8140681 0.8540549

moneyball.SLG 0.8140681 1.0000000 0.8061539

moneyball.OBP 0.8540549 0.8061539 1.0000000

**read.csv**

* if a variable in the file starts with a number, R will put an “X” in front of it see yellow shaded area in table:

|  |  |
| --- | --- |
| **original name** | **R variable name** |
| SeasonEnd | SeasonEnd |
| Team | Team |
| Playoffs | Playoffs |
| W | W |
| PTS | PTS |
| oppPTS | oppPTS |
| FG | FG |
| FGA | FGA |
| 2P | X2P |
| 2PA | X2PA |
| 3P | X3P |
| 3PA | X3PA |
| FT | FT |
| FTA | FTA |
| ORB | ORB |
| DRB | DRB |
| AST | AST |
| STL | STL |
| BLK | BLK |
| TOV | TOV |

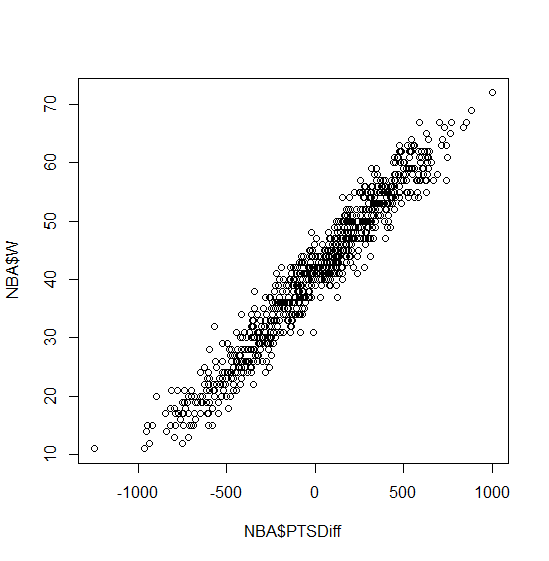
**Interpretation:**  based on the table below a logical assumption would be that if a team wins at least 42 games, they have a very good chance of making it to the playoffs

|  |  |  |
| --- | --- | --- |
|  | not in playoffs | in playoffs |
| Wins | 0 | 1 |
| 11 | 2 | 0 |
| 12 | 2 | 0 |
| 13 | 2 | 0 |
| 14 | 2 | 0 |
| 15 | 10 | 0 |
| 16 | 2 | 0 |
| 17 | 11 | 0 |
| 18 | 5 | 0 |
| 19 | 10 | 0 |
| 20 | 10 | 0 |
| 21 | 12 | 0 |
| 22 | 11 | 0 |
| 23 | 11 | 0 |
| 24 | 18 | 0 |
| 25 | 11 | 0 |
| 26 | 17 | 0 |
| 27 | 10 | 0 |
| 28 | 18 | 0 |
| 29 | 12 | 0 |
| 30 | 19 | 1 |
| 31 | 15 | 1 |
| 32 | 12 | 0 |
| 33 | 17 | 0 |
| 34 | 16 | 0 |
| 35 | 13 | 3 |
| 36 | 17 | 4 |
| 37 | 15 | 4 |
| 38 | 8 | 7 |
| 39 | 10 | 10 |
| 40 | 9 | 13 |
| 41 | 11 | 26 |
| 42 | 8 | 29 |
| 43 | 2 | 18 |
| 44 | 2 | 27 |
| 45 | 3 | 22 |
| 46 | 1 | 15 |
| 47 | 0 | 28 |
| 48 | 1 | 14 |
| 49 | 0 | 17 |
| 50 | 0 | 32 |
| 51 | 0 | 12 |
| 52 | 0 | 20 |
| 53 | 0 | 17 |
| 54 | 0 | 18 |
| 55 | 0 | 24 |
| 56 | 0 | 16 |
| 57 | 0 | 23 |
| 58 | 0 | 13 |
| 59 | 0 | 14 |
| 60 | 0 | 8 |
| 61 | 0 | 10 |
| 62 | 0 | 13 |
| 63 | 0 | 7 |
| 64 | 0 | 3 |
| 65 | 0 | 3 |
| 66 | 0 | 2 |
| 67 | 0 | 4 |
| 69 | 0 | 1 |
| 72 | 0 | 1 |

**Assumption:**

games in the NBA are won by scoring more points than the opponent. See if there is a strong correlation between wins and point differential by plotting wins against point differential. There is a very strong correlation between the two variables.

**Conclusion:** this supports that linear regression would be a good tool to predict wins based on points differential



> summary(WinsReg)

Call:

lm(formula = NBA$W ~ NBA$PTSDiff, data = NBA)

Residuals:

Min 1Q Median 3Q Max

-9.7393 -2.1018 -0.0672 2.0265 10.6026

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 4.100e+01 1.059e-01 387.0 <2e-16 \*\*\*

NBA$PTSDiff 3.259e-02 2.793e-04 116.7 <2e-16 \*\*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.061 on 833 degrees of freedom

Multiple R-squared: 0.9423, Adjusted R-squared: 0.9423

F-statistic: 1.361e+04 on 1 and 833 DF, p-value: < 2.2e-16

**This linear model tells us that:**

Wins=41 + 0.326 \* PTSDiff >= 42

PTSDIff >=

= 30.67

**Interpretation:** we need to score at least 31 more points than we allow to win at least 42 games

**Predicting points scored**

Call:

lm(formula = PTS ~ X2PA + X3PA + FTA + AST + ORB + DRB + TOV +

STL + BLK, data = NBA)

Residuals:

Min 1Q Median 3Q Max

-527.40 -119.83 7.83 120.67 564.71

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -2.051e+03 2.035e+02 -10.078 <2e-16 \*\*\*

X2PA 1.043e+00 2.957e-02 35.274 <2e-16 \*\*\*

X3PA 1.259e+00 3.843e-02 32.747 <2e-16 \*\*\*

FTA 1.128e+00 3.373e-02 33.440 <2e-16 \*\*\*

AST 8.858e-01 4.396e-02 20.150 <2e-16 \*\*\*

ORB -9.554e-01 7.792e-02 -12.261 <2e-16 \*\*\*

DRB 3.883e-02 6.157e-02 0.631 0.5285

TOV -2.475e-02 6.118e-02 -0.405 0.6859

STL -1.992e-01 9.181e-02 -2.169 0.0303 \*

BLK -5.576e-02 8.782e-02 -0.635 0.5256

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 185.5 on 825 degrees of freedom

Multiple R-squared: 0.8992, Adjusted R-squared: 0.8981

F-statistic: 817.3 on 9 and 825 DF, p-value: < 2.2e-16

**Interpretation:**

1. There is a linear relationship between points and these basketball statistics. This is a strong model, with a .8992 R-squared, but with some insignificant variables
2. Steals (STL) is not that significant – only one \*
3. Defensive rebounds (DRB) and turnovers (TOV) is not significant at all

#compute the Sum of Squared Errors (SSE)

SSE= sum(PointsReg$residuals^2)

SSE

#28,394,314 - not very interpretable quantity

#calculate the root mean squared error - more interpretable

# and is more like the average error

#equal to the squared root of the SSE divided by the total number of observations

RMSE=sqrt(SSE/nrow(NBA))

RMSE

#184.4

#on average we make an error of 184.4 points

#this is not that bad

#when compared to the average number of points in a season

#184.4 compared to 8,370.24 points in the entire season

mean(NBA$PTS)

#8370.24

**Refining our model – remove insignificant variables**

* remove the least significant predictor first
* this is because the “P value” (.6859) is the largest

Call:

lm(formula = PTS ~ X2PA + X3PA + FTA + AST + ORB + DRB + TOV +

STL + BLK, data = NBA)

Residuals:

Min 1Q Median 3Q Max

-527.40 -119.83 7.83 120.67 564.71

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -2.051e+03 2.035e+02 -10.078 <2e-16 \*\*\*

X2PA 1.043e+00 2.957e-02 35.274 <2e-16 \*\*\*

X3PA 1.259e+00 3.843e-02 32.747 <2e-16 \*\*\*

FTA 1.128e+00 3.373e-02 33.440 <2e-16 \*\*\*

AST 8.858e-01 4.396e-02 20.150 <2e-16 \*\*\*

ORB -9.554e-01 7.792e-02 -12.261 <2e-16 \*\*\*

DRB 3.883e-02 6.157e-02 0.631 0.5285

TOV -2.475e-02 6.118e-02 -0.405 0.6859

STL -1.992e-01 9.181e-02 -2.169 0.0303 \*

BLK -5.576e-02 8.782e-02 -0.635 0.5256

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 185.5 on 825 degrees of freedom

Multiple R-squared: 0.8992, Adjusted R-squared: 0.8981

F-statistic: 817.3 on 9 and 825 DF, p-value: < 2.2e-16

The new model has the following summary:

* this new model almost the same R-squared: .8991 vs .8992

summary(PointsReg2)

Call:

lm(formula = PTS ~ X2PA + X3PA + FTA + AST + ORB + DRB + STL +

BLK, data = NBA)

Residuals:

Min 1Q Median 3Q Max

-526.79 -121.09 6.37 120.74 565.94

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -2.077e+03 1.931e+02 -10.755 <2e-16 \*\*\*

X2PA 1.044e+00 2.951e-02 35.366 <2e-16 \*\*\*

X3PA 1.263e+00 3.703e-02 34.099 <2e-16 \*\*\*

FTA 1.125e+00 3.308e-02 34.023 <2e-16 \*\*\*

AST 8.861e-01 4.393e-02 20.173 <2e-16 \*\*\*

ORB -9.581e-01 7.758e-02 -12.350 <2e-16 \*\*\*

DRB 3.892e-02 6.154e-02 0.632 0.5273

STL -2.068e-01 8.984e-02 -2.301 0.0216 \*

BLK -5.863e-02 8.749e-02 -0.670 0.5029

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 185.4 on 826 degrees of freedom

Multiple R-squared: 0.8991, Adjusted R-squared: 0.8982

F-statistic: 920.4 on 8 and 826 DF, p-value: < 2.2e-16

Now lets take out the next statistically insignificant variable, defensive rebounds (DRB)

* same R-squared: .8991

summary(PointsReg3)

Call:

lm(formula = PTS ~ X2PA + X3PA + FTA + AST + ORB + STL + BLK,

data = NBA)

Residuals:

Min 1Q Median 3Q Max

-523.79 -121.64 6.07 120.81 573.64

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -2.015e+03 1.670e+02 -12.068 < 2e-16 \*\*\*

X2PA 1.048e+00 2.852e-02 36.753 < 2e-16 \*\*\*

X3PA 1.271e+00 3.475e-02 36.568 < 2e-16 \*\*\*

FTA 1.128e+00 3.270e-02 34.506 < 2e-16 \*\*\*

AST 8.909e-01 4.326e-02 20.597 < 2e-16 \*\*\*

ORB -9.702e-01 7.519e-02 -12.903 < 2e-16 \*\*\*

STL -2.276e-01 8.356e-02 -2.724 0.00659 \*\*

BLK -3.882e-02 8.165e-02 -0.475 0.63462

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 185.4 on 827 degrees of freedom

Multiple R-squared: 0.8991, Adjusted R-squared: 0.8982

F-statistic: 1053 on 7 and 827 DF, p-value: < 2.2e-16

Take out the last significant variable, blocks (BLK):

* R-squared is still the same: .8991

summary(PointsReg4)

Call:

lm(formula = PTS ~ X2PA + X3PA + FTA + AST + ORB + STL, data = NBA)

Residuals:

Min 1Q Median 3Q Max

-523.33 -122.02 6.93 120.68 568.26

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) -2.033e+03 1.629e+02 -12.475 < 2e-16 \*\*\*

X2PA 1.050e+00 2.829e-02 37.117 < 2e-16 \*\*\*

X3PA 1.273e+00 3.441e-02 37.001 < 2e-16 \*\*\*

FTA 1.127e+00 3.260e-02 34.581 < 2e-16 \*\*\*

AST 8.884e-01 4.292e-02 20.701 < 2e-16 \*\*\*

ORB -9.743e-01 7.465e-02 -13.051 < 2e-16 \*\*\*

STL -2.268e-01 8.350e-02 -2.717 0.00673 \*\*

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 185.3 on 828 degrees of freedom

Multiple R-squared: 0.8991, Adjusted R-squared: 0.8983

F-statistic: 1229 on 6 and 828 DF, p-value: < 2.2e-16

**comparing the Root Mean Squared Error of the two models:**

|  |
| --- |
| > #compute the Sum of Squared Errors (SSE) on new model  > SSE\_4= sum(PointsReg4$residuals^2)  > SSE\_4  [1] 28421465  >  > #equal to the squared root of the SSE divided by the total number of observations  > RMSE\_4=sqrt(SSE\_4/nrow(NBA))  > RMSE\_4  [1] 184.493  > #root mean squared error difference between models are very small  > #which means we kept the same amount of error  > #original model RMSE:  > RMSE  [1] 184.4049 |
|  |
| |  | | --- | |  | |

**Validation: how well does the model fit test data?**

* R-squared from model (.8991) is the IN SAMPLE R-squared
* to get a measure of the predictions goodness of fit, we need to calculate the out of sample R-squared

|  |
| --- |
| > #Let's predict how many points will be scored in the 2012-2013 season using our model  > PointsPredictions=predict(PointsReg4,newdata=NBA\_Test)  > #sum of PREDICTED amount - sum of ACTUAL points squared and summed  > SSE=sum((PointsPredictions-NBA\_Test$PTS)^2)  > # sum of the average number of points minus the test actual number of points  > SST=sum((mean(NBA$PTS)-NBA\_Test$PTS)^2)  > SSE  [1] 1079739  > SST  [1] 5765192  > R2=1-(SSE/SST)  > R2  [1] 0.8127142  >  > RMSE=sqrt(SSE/nrow(NBA\_Test))  > # a little higher than the training data, but still reasonable  > RMSE  [1] 196.3723 |
|  |
| |  | | --- | |  |   **Factors – reference levels** |

However, by default R selects the first level alphabetically ("American Indian/Alaska Native") as the reference level of our factor instead of the most common level ("White"). Set the reference level of the factor by typing the following two lines in your R console:

pisaTrain$raceeth = relevel(pisaTrain$raceeth, "White")

pisaTest$raceeth = relevel(pisaTest$raceeth, "White")

**LM function tips**

It would be time-consuming to type all the variables, but R provides the shorthand notation "readingScore ~ ." to mean "predict readingScore using all the other variables in the data frame." The period is used to replace listing out all of the independent variables. As an example, if your dependent variable is called "Y", your independent variables are called "X1", "X2", and "X3", and your training data set is called "Train", instead of the regular notation:

LinReg = lm(Y ~ X1 + X2 + X3, data = Train)

You would use the following command to build your model:

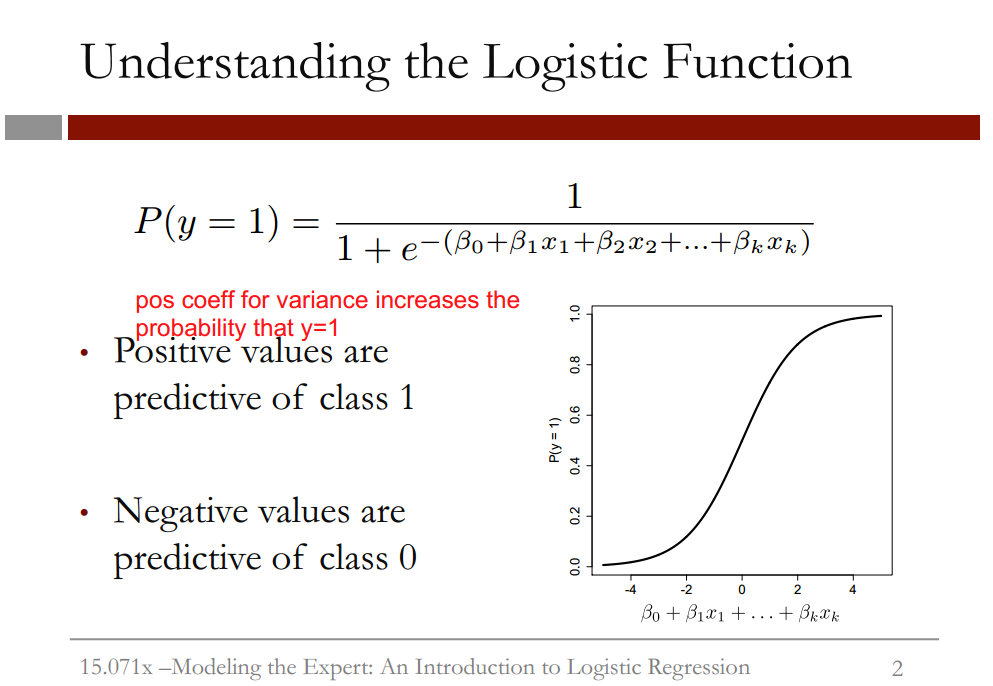
LinReg = lm(Y ~ ., data = Train)

**Logorithmic dependent variables**

However, the dependent variable in our model is log(ILI), so PredTest1 would contain predictions of the log(ILI) value. We are instead interested in obtaining predictions of the ILI value. We can convert from predictions of log(ILI) to predictions of ILI via exponentiation, or the exp() function. The new code, which predicts the ILI value, is

PredTest1 = exp(predict(FluTrend1, newdata=FluTest))

Logistical Regression



**Explanation of slide:**

* this graph shows the logistic response function for different values of the linear regression piece
* **NOTE:** the x axis always takes values between 0 and 1, as it’s a probability

**Relationship of coefficients to probabilities:**

|  |  |  |
| --- | --- | --- |
| **Coefficient (** | **Logit** | **Relation to probability** |
| positive | increases | INCREASES the probability of 1 |
| negative | decreases | DECREASES the linear regression piece, which DECREASES the probability that y=1  OR INCREASES the probability of good care |

## Threshold values

## 

* **sensitivity** is also called the “true positive rate”
* **specificity** is also called the “true negative rate”

**Relationship of sensitivity and specificity to threshold:**

|  |  |  |
| --- | --- | --- |
| **Threshold** | **Sensitivity** | **specificity** |
| **higher** | lower | higher |
| **lower** | higher | lower |

**Creating a confusion matrix in R**

#creating a confusion matrix where threshold=0.5

> #actual outcomes by predicted outcomes

> table(qualityTrain$PoorCare,predictTrain>0.5)

safasdf

**rows:** true outcome

**columns:** predicted outcome

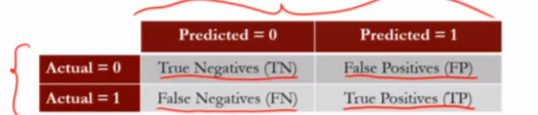
FALSE TRUE

0 70 4

1 15 10

**Interpretation:**

* 70 cases we predicted good care, and they actually received good care
* 10 cases we predicted poor care and they actually received poor care
* model makes 4 mistakes where we say poor care (1) and it’s actually good care (0)
* model make 15 mistakes where we say good care (0) and it’s actually poor care



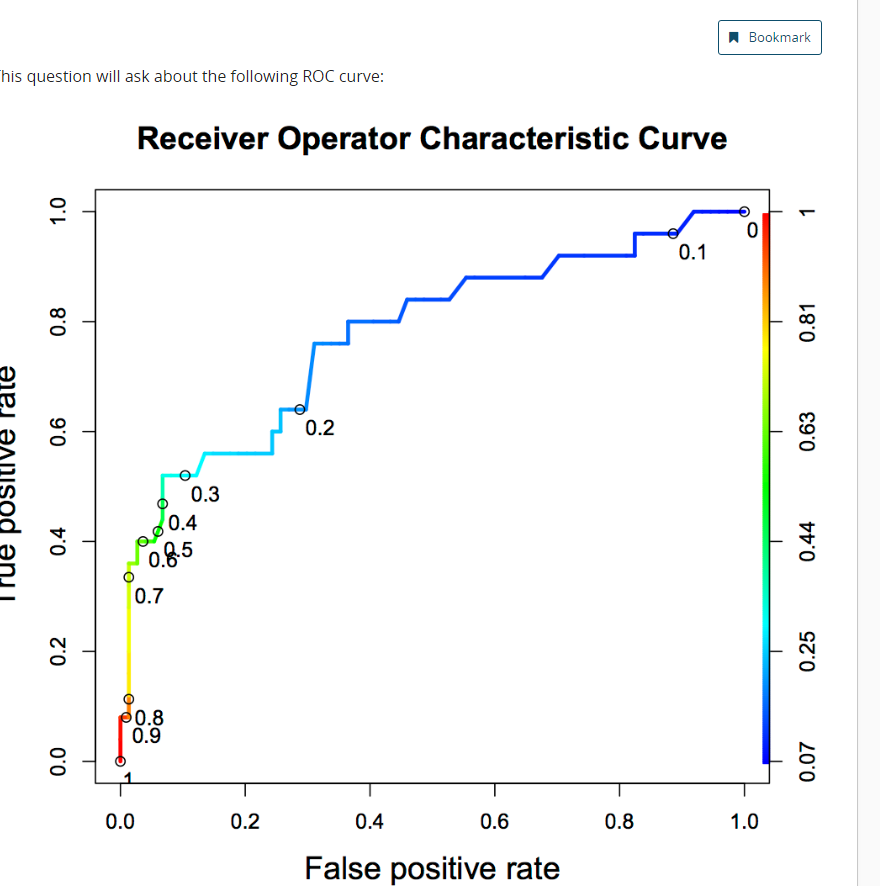
|  |
| --- |
| > table(qualityTrain$PoorCare,predictTrain>0.5)    FALSE TRUE  0 70 4  1 15 10  > #threshold= 0.5  > #sensitivity (true positive rate)= true positives/(true positives+false negatives)  > sensitivity= 10/(10+15)  > sensitivity  [1] 0.4  >  > #specificity (true negative rate)= true negatives/(true negatives+false positives)  > specificity=70/(70+4)  > specificity  [1] 0.9459459 |
|  |
| |  | | --- | | >  > #creating a confusion matrix where threshold=0.7  > #by increasing threshold, sensitivity goes down, specificity goes up  > #actual outcomes by predicted outcomes  > table(qualityTrain$PoorCare,predictTrain>0.7)    FALSE TRUE  0 73 1  1 17 8  >  > #threshold= 0.7  > #sensitivity (true positive rate)= true positives/(true positives+false negatives)  > sensitivity= 8/(8+17)  > sensitivity  [1] 0.32  >  > #specificity (true negative rate)= true negatives/(true negatives+false positives)  > specificity=73/(73+1)  > specificity  [1] 0.9864865 | |

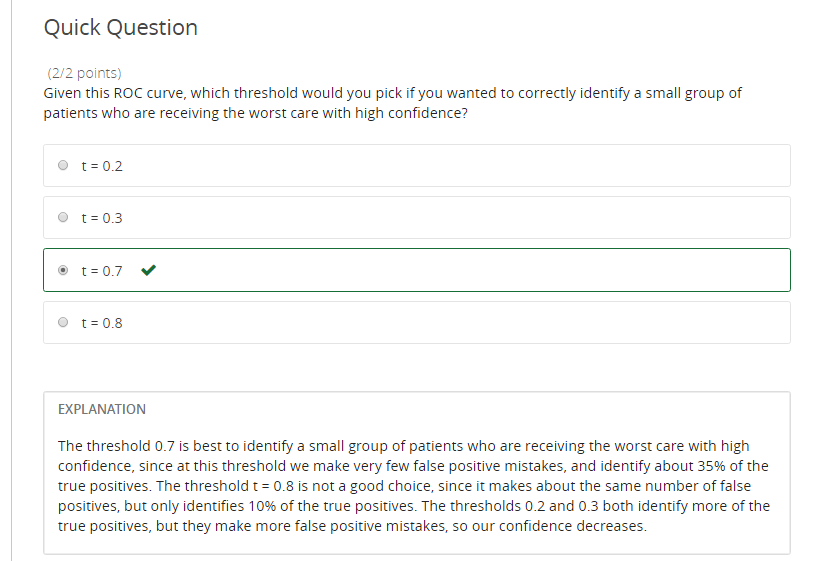
|  |
| --- |
| > #creating a confusion matrix where threshold=0.2  > #by decreasing threshold, sensitivity goes up, specificity goes down  > #actual outcomes by predicted outcomes  > table(qualityTrain$PoorCare,predictTrain>0.2)    FALSE TRUE  0 54 20  1 9 16  >  > #threshold= 0.7  > #sensitivity (true positive rate)= true positives/(true positives+false negatives)  > sensitivity= 16/(16+9)  > sensitivity  [1] 0.64  >  > #specificity (true negative rate)= true negatives/(true negatives+false positives)  > specificity=54/(54+20)  > specificity  [1] 0.7297297 |
|  |
| |  | | --- | | > | |

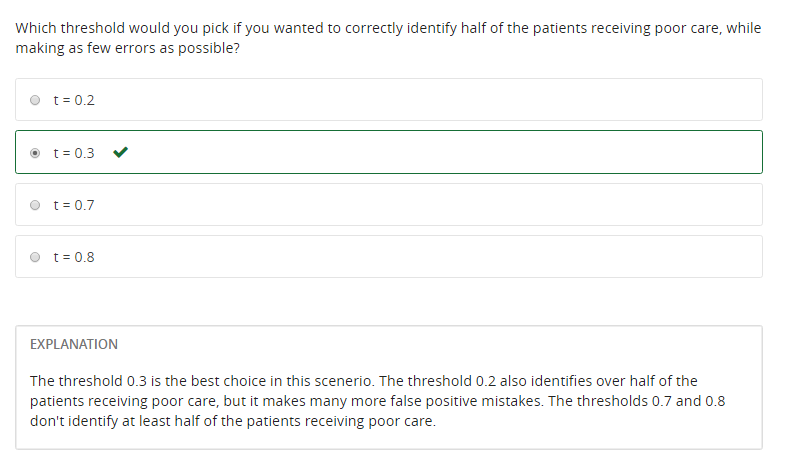
## Interpretation of ROC curve

**learnings from below**

* “high confidence” means minimizing false positives







Framingham

table(test$TenYearCHD,predictTest>0.5)

FALSE TRUE

0 1069 6

1 187 11

**Interpretation:**

accuracy of model:

total predictions we get right divided by total number of observations

= (1069+11)/(1069+6+187+11)

**84.84% accurate**

Useful packages used in analysis

caTools

**Example**

> #used to split dataset into training (75%)

> #and testing (25%)

> split=sample.split(quality$PoorCare,SplitRatio=0.75)

> split

[1] TRUE TRUE TRUE TRUE FALSE TRUE FALSE TRUE FALSE FALSE TRUE FALSE TRUE TRUE

[15] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE

[29] TRUE FALSE FALSE FALSE FALSE TRUE TRUE TRUE FALSE TRUE TRUE TRUE FALSE FALSE

[43] TRUE TRUE FALSE TRUE FALSE TRUE FALSE TRUE TRUE FALSE FALSE TRUE TRUE TRUE

[57] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE

[71] FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE

[85] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE

[99] TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE FALSE FALSE TRUE

[113] TRUE FALSE TRUE TRUE TRUE FALSE TRUE TRUE FALSE TRUE TRUE FALSE TRUE TRUE

[127] FALSE TRUE TRUE TRUE FALSE

>

>

> #create training and testing sets

> #create train

> qualityTrain=subset(quality,split==TRUE)

> qualityTest=subset(quality,split==FALSE)