Exam PA Study Manual

Sam Castillo

2020-01-03

Contents

Chapter 1

What's in this book

- Explanations of the statistical concepts
- All data sets needed packaged in an R library
- `R code` examples

1.1 About the author

Sam Castillo is a predictive modeler at Milliman in Chicago, maintains a blog about the future of risk, and won the 2019 SOA Predictive Analytics and Fururism's Jupyter contest.

Contact:

Support: sam@exampa.net

Chapter 2

The exam

The main challenge of this exam is in communication: both understanding what they want you to do as well as telling the grader what it is that you did.

You will have 5 hours and 15 minutes to use RStudio and Excel to fill out a report in Word on a Prometric computer. The syllabus uses fancy language to describe the topics covered on the exam, making it sound more difficult than it should be. A good analogy is a job description that has many complex-sounding tasks, when in reality the day-to-day operations of the employee are far simpler.

A non-technical translation is as follows:

Writing in Microsoft Word (30-40%)

- Write in professional language
- Type more than 50 words-per-minute

Manipulating Data in R (15-25%)

- Quickly clean data sets
- Find data errors planted by the SOA
- Perform queries (aggregations, summaries, transformations)

Machine learning and statistics (40-50%)

- Interpret results within a business context
- Change model parameters

Chapter 3

Prometric Demo

The following video from Prometric shows what the computer set up will look like. In addition to the files shown in the video, they will give you a printed out project statement (If they don't give this to you right away, ask for it.)

SOAFinalCut from Prometric on Vimeo.

Chapter 4

You already know what learning is

All of use are already familiar with how to learn - by improving from our mistakes. By repeating what is successful and avoiding what results in failure, we learn by doing, by experience, or trial-and-error. Machines learn in a similar way.

Take for example the process of studying for an exam. Some study methods work well, but other methods do not. The "data" are the practice problems, and the "label" is the answer (A,B,C,D,E). We want to build a mental "model" that reads the question and predicts the answer.

We all know that memorizing answers without understanding concepts is ineffective, and statistics calls this "overfitting". Conversely, not learning enough of the details and only learning the high-level concepts is "underfitting".

The more practice problems that we do, the larger the training data set, and the better the prediction. When we see new problems, ones which have not appeared in the practice exams, we often have a difficult time. Quizing ourselves on realistic questions estimates our preparedness, and this is identical to a process known as "holdout testing" or "cross-validation".

We can clearly state our objective: get as many correct answers as possible! We want to correctly predict the solution to every problem. Said another way, we are trying to minimize the error, known as the "loss function".

Different study methods work well for different people. Some cover material quickly and others slowly absorb every detail. A model has many "parameters" such as the "learning rate". The only way to know which parameters are best is to test them on real data, known as "training".

Chapter 5

Getting started

5.1 Download the data

For your convenience, all data in this book, including data from prior exams and sample solutions, has been put into a library called `ExamPAData` by the author. To access, simplly run the below lines of code to download this data.

```
# Install remotes if it's not yet installed
# install.packages("remotes")
remotes::install_github("sdcastillo/ExamPAData")
```

Once this has run, you can access the data using `library(ExamPAData)`. To check that this is installed correctly see if the `insurance` data set has loaded. If this returns "object not found", then the library was not installed.

```
library(ExamPAData)
summary(insurance)
```

```
##     district       group                age                holders      
##  Min.   :1.00   Length:64          Length:64          Min.   :   3.00  
##  1st Qu.:1.75   Class :character   Class :character   1st Qu.:  46.75  
##  Median :2.50   Mode  :character   Mode  :character   Median : 136.00  
##  Mean   :2.50                                         Mean   : 364.98  
##  3rd Qu.:3.25                                         3rd Qu.: 327.50  
##  Max.   :4.00                                         Max.   :3582.00  
##      claims        
##  Min.   :  0.00  
##  1st Qu.:  9.50  
##  Median : 22.00
```

```
##  Mean   : 49.23
##  3rd Qu.: 55.50
##  Max.   :400.00
```

5.2 Download ISLR

This book references the publically-avialable textbook "An Introduction to Statistical Learning", which can be downloaded for free

http://faculty.marshall.usc.edu/gareth-james/ISL/

If you already have R and RStudio installed then skip to "Download the data".

5.3 New users

Install R:

This is the engine that *runs* the code. https://cran.r-project.org/mirrors.html

Install RStudio

This is the tool that helps you to *write* the code. Just as MS Word creates documents, RStudio creates R scripts and other documents. Download RStudio Desktop (the free edition) and choose a place on your computer to install it.

https://rstudio.com/products/rstudio/download/

Set the R library

R code is organized into libraries. You want to use the exact same code that will be on the Prometric Computers. This requires installing older versions of libraries. Change your R library to the one which was included within the SOA's modules.

```
.libPaths("PATH_TO_SOAS_LIBRARY/PAlibrary")
```

Chapter 6

R programming

This chapter covers the bare minimum of R programming needed for Exam PA. The book "R for Data Science" provides more detail.

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

6.1 Notebook chunks

On the Exam, you will start with an .Rmd (R Markdown) template, which organize code into R Notebooks. Within each notebook, code is organized into chunks.

```
# This is a chunk
```

Your time is valuable. Throughout this book, I will include useful keyboard shortcuts.

> **Shortcut:** To run everything in a chunk quickly, press `CTRL + SHIFT + ENTER`. To create a new chunk, use `CTRL + ALT + I`.

6.2 Basic operations

The usual math operations apply.

```
# Addition
1 + 2
```

```
## [1] 3
```

```
3 - 2
```

```
## [1] 1
```

```
# Multiplication
2 * 2
```

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

```
# Division
4 / 2
```

```
## [1] 2
```

```
# Exponentiation
2^3
```

```
## [1] 8
```

There are two assignment operators: `=` and `<-`. The latter is preferred because it is specific to assigning a variable to a value. The `=` operator is also used for specifying arguments in functions (see the functions section).

> **Shortcut:** `ALT + -` creates a `<-`.

```
# Variable assignment
y <- 2

# Equality
4 == 2
```

```
## [1] FALSE
```

```
5 == 5
```

```
## [1] TRUE
```

```
3.14 > 3
```

```
## [1] TRUE
```

```
3.14 >= 3
```

```
## [1] TRUE
```

Vectors can be added just like numbers. The `c` stands for "concatenate", which creates vectors.

```
x <- c(1, 2)
y <- c(3, 4)
x + y
```

```
## [1] 4 6
```

```
x * y
```

```
## [1] 3 8
```

```
z <- x + y
z^2
```

```
## [1] 16 36
```

```
z / 2
```

```
## [1] 2 3
```

```
z + 3
```

```
## [1] 7 9
```

I already mentioned `numeric` types. There are also `character` (string) types, `factor` types, and `boolean` types.

```
character <- "The"
character_vector <- c("The", "Quick")
```

Character vectors can be combined with the `paste()` function.

```
a <- "The"
b <- "Quick"
c <- "Brown"
d <- "Fox"
paste(a, b, c, d)
```

```
## [1] "The Quick Brown Fox"
```

Factors look like character vectors but can only contain a finite number of predefined values.

The below factor has only one "level", which is the list of assigned values.

```
factor <- as.factor(character)
levels(factor)
```

```
## [1] "The"
```

The levels of a factor are by default in R in alphabetical order (Q comes alphabetically before T).

```
factor_vector <- as.factor(character_vector)
levels(factor_vector)
```

```
## [1] "Quick" "The"
```

In building linear models, the order of the factors matters. In GLMs, the "reference level" or "base level" should always be the level which has the most observations. This will be covered in the section on linear models.

Booleans are just `TRUE` and `FALSE` values. R understands `T` or `TRUE` in the same way, but the latter is preferred. When doing math, bools are converted to 0/1 values where 1 is equivalent to TRUE and 0 FALSE.

```
bool_true <- TRUE
bool_false <- FALSE
bool_true * bool_false
```

```
## [1] 0
```

Booleans are automatically converted into 0/1 values when there is a math operation.

```
bool_true + 1
```

```
## [1] 2
```

Vectors work in the same way.

```
bool_vect <- c(TRUE, TRUE, FALSE)
sum(bool_vect)
```

```
## [1] 2
```

Vectors are indexed using `[`. If you are only extracting a single element, you should use `[[` for clarity.

```
abc <- c("a", "b", "c")
abc[[1]]
```

```
## [1] "a"
```

```
abc[[2]]
```

```
## [1] "b"
```

```
abc[c(1, 3)]
```

```
## [1] "a" "c"
```

```
abc[c(1, 2)]
```

```
## [1] "a" "b"
```

```
abc[-2]
```

```
## [1] "a" "c"
```

```
abc[-c(2, 3)]
```

```
## [1] "a"
```

6.3 Lists

Lists are vectors that can hold mixed object types.

```r
my_list <- list(TRUE, "Character", 3.14)
my_list
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] "Character"
##
## [[3]]
## [1] 3.14
```

Lists can be named.

```r
my_list <- list(bool = TRUE, character = "character", numeric = 3.14)
my_list
```

```
## $bool
## [1] TRUE
##
## $character
## [1] "character"
##
## $numeric
## [1] 3.14
```

The `$` operator indexes lists.

```r
my_list$numeric
```

```
## [1] 3.14
```

```r
my_list$numeric + 5
```

```
## [1] 8.14
```

Lists can also be indexed using `[[`.

```r
my_list[[1]]
```

```
## [1] TRUE
```

```
my_list[[2]]
```

```
## [1] "character"
```

Lists can contain vectors, other lists, and any other object.

```
everything <- list(vector = c(1, 2, 3),
                   character = c("a", "b", "c"),
                   list = my_list)
everything
```

```
## $vector
## [1] 1 2 3
##
## $character
## [1] "a" "b" "c"
##
## $list
## $list$bool
## [1] TRUE
##
## $list$character
## [1] "character"
##
## $list$numeric
## [1] 3.14
```

To find out the type of an object, use `class` or `str` or `summary`.

```
class(x)
```

```
## [1] "numeric"
```

```
class(everything)
```

```
## [1] "list"
```

```
str(everything)
```

```
## List of 3
##  $ vector   : num [1:3] 1 2 3
```

```
##  $ character: chr [1:3] "a" "b" "c"
##  $ list     :List of 3
##   ..$ bool     : logi TRUE
##   ..$ character: chr "character"
##   ..$ numeric  : num 3.14
```

```
summary(everything)
```

```
##           Length Class  Mode
## vector    3      -none- numeric
## character 3      -none- character
## list      3      -none- list
```

6.4 Functions

You only need to understand the very basics of functions. The big picture, though, is that understanding functions helps you to understand *everything* in R, since R is a functional programming language, unlike Python, C, VBA, Java which are all object-oriented, or SQL which isn't really a language but a series of set-operations.

Functions do things. The convention is to name a function as a verb. The function `make_rainbows()` would create a rainbow. The function `summarise_vectors()` would summarise vectors. Functions may or may not have an input and output.

If you need to do something in R, there is a high probability that someone has already written a function to do it. That being said, creating simple functions is quite useful.

Here is an example that has a side effect of printing the input:

```
greet_me <- function(my_name){
  print(paste0("Hello, ", my_name))
}

greet_me("Future Actuary")
```

```
## [1] "Hello, Future Actuary"
```

A function that returns something

When returning the last evaluated expression, the `return` statement is optional. In fact, it is discouraged by convention.

```
add_together <- function(x, y) {
  x + y
}

add_together(2, 5)
```

```
## [1] 7
```

```
add_together <- function(x, y) {
  # Works, but bad practice
  return(x + y)
}

add_together(2, 5)
```

```
## [1] 7
```

Binary operations in R are vectorized. In other words, they are applied element-wise.

```
x_vector <- c(1, 2, 3)
y_vector <- c(4, 5, 6)
add_together(x_vector, y_vector)
```

```
## [1] 5 7 9
```

Many functions in R actually return lists! This is why R objects can be indexed with dollar sign.

```
library(ExamPAData)
model <- lm(charges ~ age, data = health_insurance)
model$coefficients
```

```
## (Intercept)         age
##   3165.8850    257.7226
```

Here's a function that returns a list.

```
sum_multiply <- function(x,y) {
  sum <- x + y
  product <- x * y
```

```
  list("Sum" = sum, "Product" = product)
}

result <- sum_multiply(2, 3)
result$Sum
```

```
## [1] 5
```

```
result$Product
```

```
## [1] 6
```

6.5 Data frames

You can think of a data frame as a table that is implemented as a list of vectors.

```
df <- data.frame(
  age = c(25, 35),
  has_fsa = c(FALSE, TRUE)
)
df
```

```
##   age has_fsa
## 1  25   FALSE
## 2  35    TRUE
```

You can also work with tibbles, which are data frames but have nicer printing:

```
# The tidyverse library has functions for making tibbles
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------
## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
df <- tibble(
  age = c(25, 35), has_fsa = c(FALSE, TRUE)
)
df
```

```
## # A tibble: 2 x 2
##     age has_fsa
##   <dbl> <lgl>
## 1    25 FALSE
## 2    35 TRUE
```

To index columns in a tibble, the same "$" is used as indexing a list.

```
df$age
```

```
## [1] 25 35
```

To find the number of rows and columns, use `dim`.

```
dim(df)
```

```
## [1] 2 2
```

To find a summary, use `summary`

```
summary(df)
```

```
##       age        has_fsa
##  Min.   :25.0   Mode :logical
##  1st Qu.:27.5   FALSE:1
##  Median :30.0   TRUE :1
##  Mean   :30.0
##  3rd Qu.:32.5
##  Max.   :35.0
```

6.6 Pipes

The pipe operator `%>%` is a way of making code *modular*, meaning that it can be written and executed in incremental steps. Those familiar with Python's Pandas will be see that `%>%` is quite similar to ".". This also makes code easier to read.

In five seconds, tell me what the below code is doing.

```
log(sqrt(exp(log2(sqrt((max(c(3, 4, 16))))))))
```

```
## [1] 1
```

Getting to the answer of 1 requires starting from the inner-most nested brackets and moving outwards from right to left.

The math notation would be slightly easier to read, but still painful.

$$log(\sqrt{e^{log_2(\sqrt{max(3,4,16)})}})$$

Here is the same algebra using the pipe. To read this, replace the `%>%` with the word `THEN`.

```
max(c(3, 4, 16)) %>%
  sqrt() %>%
  log2() %>%
  exp() %>%
  sqrt() %>%
  log()
```

```
## [1] 1
```

```
# max(c(3, 4, 16) THEN  # The max of 3, 4, and 16 is 16
#  sqrt() THEN          # The square root of 16 is 4
#  log2() THEN          # The log in base 2 of 4 is 2
#  exp() THEN           # The exponent of 2 is e^2
#  sqrt() THEN          # The square root of e^2 is e
#  log()                # The natural logarithm of e is 1
```

Pipes are exceptionally useful for data manipulations, which is covered in the next chapter.

> **Tip:** To quickly produce pipes, use `CTRL + SHIFT + M`.

By highlighting only certain sections, we can run the code in steps as if we were using a debugger. This makes testing out code much faster.

```
max(c(3, 4, 16))
```

```
## [1] 16
```

```
max(c(3, 4, 16)) %>%
  sqrt()
```

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

```
max(c(3, 4, 16)) %>%
  sqrt() %>%
  log2()
```

```
## [1] 2
```

```
max(c(3, 4, 16)) %>%
  sqrt() %>%
  log2() %>%
  exp()
```

```
## [1] 7.389056
```

```
max(c(3, 4, 16)) %>%
  sqrt() %>%
  log2() %>%
  exp() %>%
  sqrt()
```

```
## [1] 2.718282
```

```
max(c(3, 4, 16)) %>%
  sqrt() %>%
  log2() %>%
  exp() %>%
  sqrt() %>%
  log()
```

```
## [1] 1
```

6.7 The SOA's code doesn't use pipes or dplyr, so can I skip learning this?

Yes, if you really want to.

The advantages to learning pipes, and the reason why this manual uses them are

1) It saves you time.
2) It will help you in real life data science projects.
3) The majority of the R community uses this style.
4) The SOA actuaries who create the Exam PA content will eventually catch on.
5) Most modern R software is designed around them. The overall trend is towards greater adoption, as can bee seen from the CRAN download statistics here after filtering to "magrittr" which is the library where the pipe comes from.

Chapter 7

Data manipulation

About two hours in this exam will be spent just on data manipulation. Putting in extra practice in this area is garanteed to give you a better score because it will free up time that you can use elsewhere. In addition, a common saying when building models is "garbage in means garbage out", on this exam, mistakes on the data manipulation can lead to lost points on the modeling sections.

Suggested reading of *R for Data Science* (https://r4ds.had.co.nz/index.html):

Chapter	Topic
9	Introduction
10	Tibbles
12	Tidy data
15	Factors
17	Introduction
18	Pipes
19	Functions
20	Vectors

All data for this book can be accessed from the package `ExamPAData`. In the real exam, you will read the file from the Prometric computer. To read files into R, the readr package has several tools, one for each data format. For instance, the most common format, comma separated values (csv) are read with the `read_csv()` function.

Because the data is already loaded, simply use the below code to access the data.

```
library(ExamPAData)
```

7.1 Look at the data

The data that we are using is `health_insurance`, which has information on patients and their health care costs.

The descriptions of the columns are below.

- `age`: Age of the individual
- `sex`: Sex
- `bmi`: Body Mass Index
- `children`: Number of children
- `smoker`: Is this person a smoker?
- `region`: Region
- `charges`: Annual health care costs.

`head()` shows the top n rows. `head(20)` shows the top 20 rows.

```
library(tidyverse)
head(health_insurance)
```

```
## # A tibble: 6 x 7
##     age sex      bmi children smoker region    charges
##   <dbl> <chr>  <dbl>    <dbl> <chr>  <chr>       <dbl>
## 1    19 female  27.9        0 yes    southwest  16885.
## 2    18 male    33.8        1 no     southeast   1726.
## 3    28 male    33          3 no     southeast   4449.
## 4    33 male    22.7        0 no     northwest  21984.
## 5    32 male    28.9        0 no     northwest   3867.
## 6    31 female  25.7        0 no     southeast   3757.
```

Using a pipe is an alternative way of doing this.

```
health_insurance %>% head()
```

> **Shortcut**: Use `CTRL` + `SHFT` + `M` to create pipes `%>%`

The `glimpse` function is a transpose of the `head()` function, which can be more spatially efficient. This also gives you the dimension (1,338 rows, 7 columns).

```
health_insurance %>% glimpse()
```

```
## Observations: 1,338
## Variables: 7
## $ age      <dbl> 19, 18, 28, 33, 32, 31, 46, 37, 37, 60, 25, 62, 23, 5...
## $ sex      <chr> "female", "male", "male", "male", "male", "female", "...
## $ bmi      <dbl> 27.900, 33.770, 33.000, 22.705, 28.880, 25.740, 33.44...
## $ children <dbl> 0, 1, 3, 0, 0, 0, 1, 3, 2, 0, 0, 0, 0, 0, 0, 1, 1, 0,...
## $ smoker   <chr> "yes", "no", "no", "no", "no", "no", "no", "no", "no"...
## $ region   <chr> "southwest", "southeast", "southeast", "northwest", "...
## $ charges  <dbl> 16884.924, 1725.552, 4449.462, 21984.471, 3866.855, 3...
```

One of the most useful data science tools is counting things. The function `count()` gives the number of records by a categorical feature.

```
health_insurance %>% count(children)
```

```
## # A tibble: 6 x 2
##   children     n
##      <dbl> <int>
## 1        0   574
## 2        1   324
## 3        2   240
## 4        3   157
## 5        4    25
## 6        5    18
```

Two categories can be counted at once. This creates a table with all combinations of `region` and `sex` and shows the number of records in each category.

```
health_insurance %>% count(region, sex)
```

```
## # A tibble: 8 x 3
##   region    sex        n
##   <chr>     <chr>  <int>
## 1 northeast female   161
## 2 northeast male     163
## 3 northwest female   164
## 4 northwest male     161
## 5 southeast female   175
## 6 southeast male     189
## 7 southwest female   162
## 8 southwest male     163
```

The `summary()` function is shows a statistical summary. One caveat is that each column needs to be in it's appropriate type. For example, `smoker`, `region`, and `sex` are all listed as characters when if they were factors, `summary` would give you count info.

With incorrect data types

```
health_insurance %>% summary()
```

```
##       age            sex                 bmi           children
##  Min.   :18.00   Length:1338        Min.   :15.96   Min.   :0.000
##  1st Qu.:27.00   Class :character   1st Qu.:26.30   1st Qu.:0.000
##  Median :39.00   Mode  :character   Median :30.40   Median :1.000
##  Mean   :39.21                      Mean   :30.66   Mean   :1.095
##  3rd Qu.:51.00                      3rd Qu.:34.69   3rd Qu.:2.000
##  Max.   :64.00                      Max.   :53.13   Max.   :5.000
##     smoker             region            charges
##  Length:1338        Length:1338        Min.   : 1122
##  Class :character   Class :character   1st Qu.: 4740
##  Mode  :character   Mode  :character   Median : 9382
##                                        Mean   :13270
##                                        3rd Qu.:16640
##                                        Max.   :63770
```

With correct data types

This tells you that there are 324 patients in the northeast, 325 in the northwest, 364 in the southeast, and so fourth.

```
health_insurance <- health_insurance %>%
  modify_if(is.character, as.factor)

health_insurance %>%
  summary()
```

```
##       age            sex            bmi           children     smoker
##  Min.   :18.00   female:662   Min.   :15.96   Min.   :0.000   no :1064
##  1st Qu.:27.00   male  :676   1st Qu.:26.30   1st Qu.:0.000   yes: 274
##  Median :39.00                Median :30.40   Median :1.000
##  Mean   :39.21                Mean   :30.66   Mean   :1.095
##  3rd Qu.:51.00                3rd Qu.:34.69   3rd Qu.:2.000
##  Max.   :64.00                Max.   :53.13   Max.   :5.000
##        region       charges
##  northeast:324   Min.   : 1122
##  northwest:325   1st Qu.: 4740
```

```
##  southeast:364   Median : 9382
##  southwest:325   Mean   :13270
##                  3rd Qu.:16640
##                  Max.   :63770
```

7.2 Transform the data

Transforming, manipulating, querying, and wrangling are synonyms in data terminology.

R syntax is designed to be similar to SQL. They begin with a `SELECT`, use `GROUP BY` to aggregate, and have a `WHERE` to remove records. Unlike SQL, the ordering of these does not matter. `SELECT` can come after a `WHERE`.

R to SQL translation

```
select() -> SELECT
mutate() -> user-defined columns
summarize() -> aggregated columns
left_join() -> LEFT JOIN
filter() -> WHERE
group_by() -> GROUP BY
filter() -> HAVING
arrange() -> ORDER BY
```

```
health_insurance %>%
  select(age, region) %>%
  head()
```

```
## # A tibble: 6 x 2
##     age region
##   <dbl> <fct>
## 1    19 southwest
## 2    18 southeast
## 3    28 southeast
## 4    33 northwest
## 5    32 northwest
## 6    31 southeast
```

Tip: use `CTRL + SHIFT + M` to create pipes `%>%`.

Let's look at only those in the southeast region. Instead of `WHERE`, use `filter`.

```
health_insurance %>%
  filter(region == "southeast") %>%
  select(age, region) %>%
  head()
```

```
## # A tibble: 6 x 2
##     age region
##   <dbl> <fct>
## 1    18 southeast
## 2    28 southeast
## 3    31 southeast
## 4    46 southeast
## 5    62 southeast
## 6    56 southeast
```

The SQL translation is

```
SELECT age, region
FROM health_insurance
WHERE region = 'southeast'
```

Instead of `ORDER BY`, use `arrange`. Unlike SQL, the order does not matter and `ORDER BY` doesn't need to be last.

```
health_insurance %>%
  arrange(age) %>%
  select(age, region) %>%
  head()
```

```
## # A tibble: 6 x 2
##     age region
##   <dbl> <fct>
## 1    18 southeast
## 2    18 southeast
## 3    18 northeast
## 4    18 northeast
## 5    18 northeast
## 6    18 southeast
```

The `group_by` comes before the aggregation, unlike in SQL where the `GROUP BY` comes last.

```
health_insurance %>%
  group_by(region) %>%
  summarise(avg_age = mean(age))
```

```
## # A tibble: 4 x 2
##   region    avg_age
##   <fct>       <dbl>
## 1 northeast    39.3
## 2 northwest    39.2
## 3 southeast    38.9
## 4 southwest    39.5
```

In SQL, this would be

```
SELECT region,
       AVG(age) as avg_age
FROM health_insurance
GROUP BY region
```

Just like in SQL, many different aggregate functions can be used such as `SUM`, `MEAN`, `MIN`, `MAX`, and so forth.

```
health_insurance %>%
  group_by(region) %>%
  summarise(avg_age = mean(age),
            max_age = max(age),
            median_charges = median(charges),
            bmi_std_dev = sd(bmi))
```

```
## # A tibble: 4 x 5
##   region    avg_age max_age median_charges bmi_std_dev
##   <fct>       <dbl>   <dbl>          <dbl>       <dbl>
## 1 northeast    39.3      64         10058.        5.94
## 2 northwest    39.2      64          8966.        5.14
## 3 southeast    38.9      64          9294.        6.48
## 4 southwest    39.5      64          8799.        5.69
```

To create new columns, the `mutate` function is used. For example, if we wanted a column of the person's annual charges divided by their age

```
health_insurance %>%
  mutate(charges_over_age = charges/age) %>%
  select(age, charges, charges_over_age) %>%
  head(5)
```

```
## # A tibble: 5 x 3
##     age charges charges_over_age
##   <dbl>   <dbl>            <dbl>
## 1    19  16885.            889.
## 2    18   1726.             95.9
## 3    28   4449.            159.
## 4    33  21984.            666.
## 5    32   3867.            121.
```

We can create as many new columns as we want.

```
health_insurance %>%
  mutate(age_squared  = age^2,
         age_cubed = age^3,
         age_fourth = age^4) %>%
  head(5)
```

```
## # A tibble: 5 x 10
##     age sex     bmi children smoker region charges age_squared age_cubed
##   <dbl> <fct> <dbl>    <dbl> <fct>  <fct>    <dbl>       <dbl>     <dbl>
## 1    19 fema~  27.9        0 yes    south~  16885.         361      6859
## 2    18 male   33.8        1 no     south~   1726.         324      5832
## 3    28 male   33          3 no     south~   4449.         784     21952
## 4    33 male   22.7        0 no     north~  21984.        1089     35937
## 5    32 male   28.9        0 no     north~   3867.        1024     32768
## # ... with 1 more variable: age_fourth <dbl>
```

The `CASE WHEN` function is quite similar to SQL. For example, we can create a column which is `0` when `age < 50`, `1` when `50 <= age <= 70`, and `2` when `age > 70`.

```
health_insurance %>%
  mutate(age_bucket = case_when(age < 50 ~ 0,
                                age <= 70 ~ 1,
                                age > 70 ~ 2)) %>%
  select(age, age_bucket)
```

```
## # A tibble: 1,338 x 2
##      age age_bucket
##    <dbl>      <dbl>
##  1    19          0
##  2    18          0
##  3    28          0
##  4    33          0
```

```
##  5    32          0
##  6    31          0
##  7    46          0
##  8    37          0
##  9    37          0
## 10    60          1
## # ... with 1,328 more rows
```

SQL translation:

```
SELECT CASE WHEN AGE < 50 THEN 0
       ELSE WHEN AGE <= 70 THEN 1
       ELSE 2
FROM health_insurance
```

7.3 Exercises

Run this code on your computer to answer these exercises.

The data `actuary_salaries` contains the salaries of actuaries collected from the DW Simpson survey. Use this data to answer the exercises below.

```
actuary_salaries %>% glimpse()
```

```
## Observations: 138
## Variables: 6
## $ industry    <chr> "Casualty", "Casualty", "Casualty", "Casualty", "C...
## $ exams       <chr> "1 Exam", "2 Exams", "3 Exams", "4 Exams", "1 Exam...
## $ experience  <dbl> 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5,...
## $ salary      <chr> "48 - 65", "50 - 71", "54 - 77", "58 - 82", "54 - ...
## $ salary_low  <dbl> 48, 50, 54, 58, 54, 57, 62, 63, 65, 70, 72, 85, 55...
## $ salary_high <chr> "65", "71", "77", "82", "72", "81", "87", "91", "9...
```

1. How many industries are represented?
2. The `salary_high` column is a character type when it should be numeric. Change this column to numeric.
3. What are the highest and lowest salaries for an actuary in Health with 5 exams passed?
4. Create a new column called `salary_mid` which has the middle of the `salary_low` and `salary_high` columns.
5. When grouping by industry, what is the highest `salary_mid`? What about `salary_high`? What is the lowest `salary_low`?
6. There is a mistake when `salary_low == 11`. Find and fix this mistake, and then rerun the code from the previous task.

7. Create a new column, called `n_exams`, which is an integer. Use 7 for ASA/ACAS and 10 for FSA/FCAS. Use the code below as a starting point and fill in the _ spaces
8. Create a column called `social_life`, which is equal to `n_exams/experience`. What is the average (mean) `social_life` by industry? Bonus question: what is wrong with using this as a statistical measure?

```
actuary_salaries <- actuary_salaries %>%
  mutate(n_exams = case_when(exams == "FSA" ~ _,
                             exams == "ASA" ~ _,
                             exams == "FCAS" ~ _,
                             exams == "ACAS" ~ _,
                             TRUE ~ as.numeric(substr(exams,_,_))))
```

8. Create a column called `social_life`, which is equal to `n_exams/experience`. What is the average (mean) `social_life` by industry? Bonus question: what is wrong with using this as a statistical measure?

7.4 Answers to exercises

Answers to these exercises, along with a video tutorial, are available at ExamPA.net.

Chapter 8

Visualization

This sections shows how to create and interpret simple graphs. In past exams, the SOA has provided code for any technical visualizations which are needed.

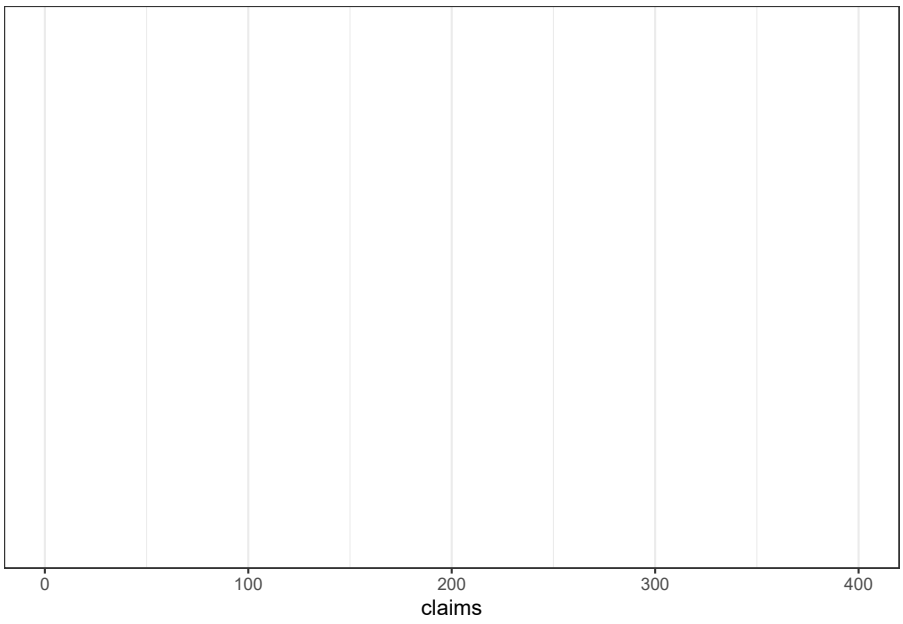
8.1 Create a plot object (ggplot)

Let's create a histogram of the claims. The first step is to create a blank canvas that holds the columns that are needed. The `aesthetic` argument, `aes`, means that the variable shown will the the claims.

```
library(ExamPAData)
p <- insurance %>% ggplot(aes(claims))
```

If we look at `p`, we see that it is nothing but white space with axis for `count` and `income`.

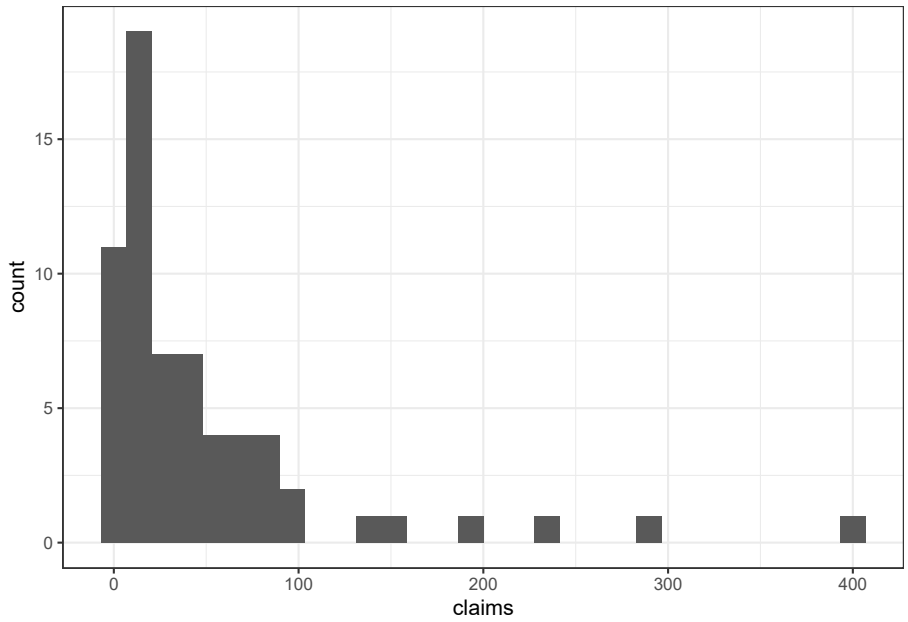
```
p
```



8.2 Add a plot

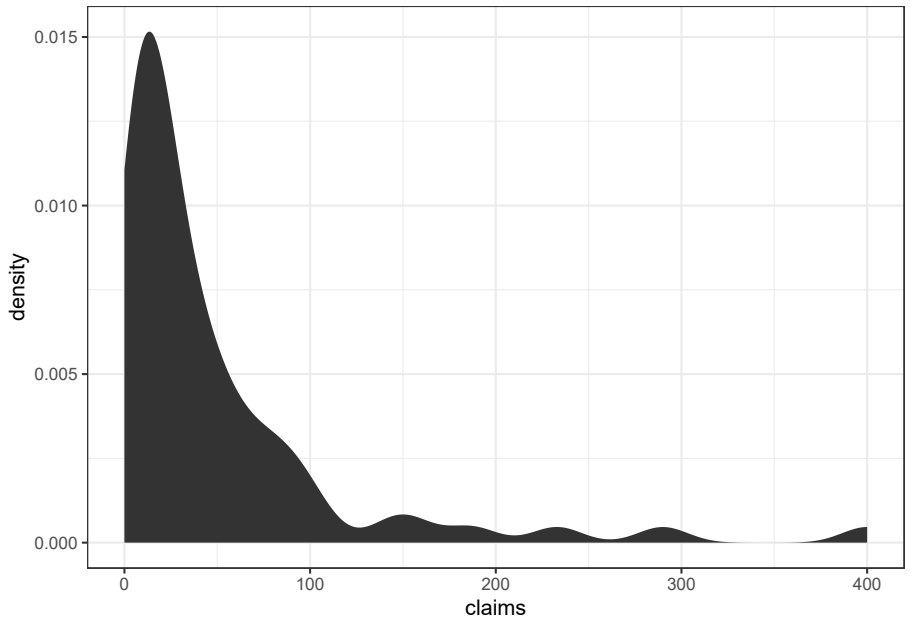
We add a histogram

```
p + geom_histogram()
```



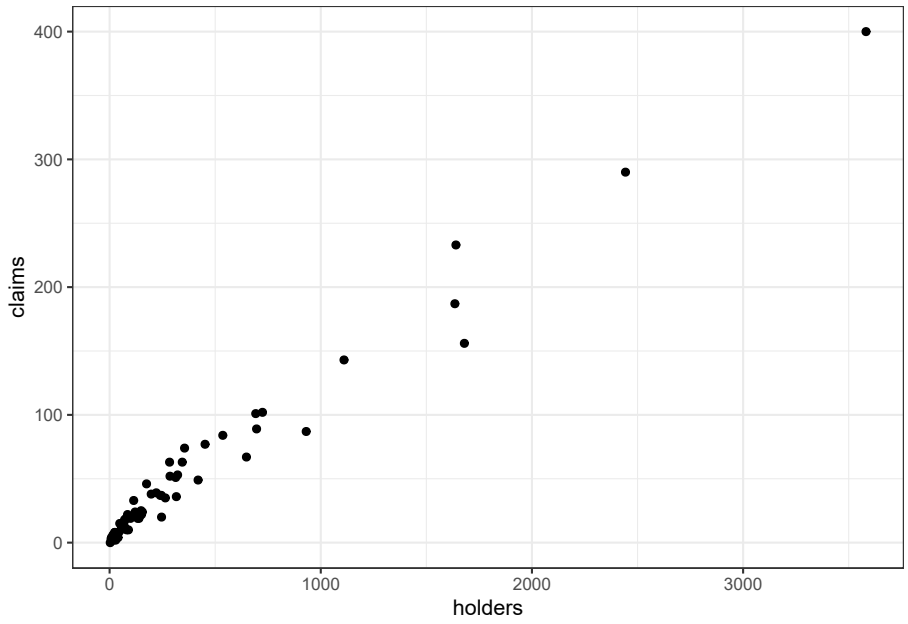
Different plots are called "geoms" for "geometric objects". Geometry = Geo (space) + metre (measure), and graphs measure data. For instance, instead of creating a histogram, we can draw a gamma distribution with `stat_density`.

```
p + stat_density()
```



Create an xy plot by adding and `x` and a `y` argument to `aesthetic`.

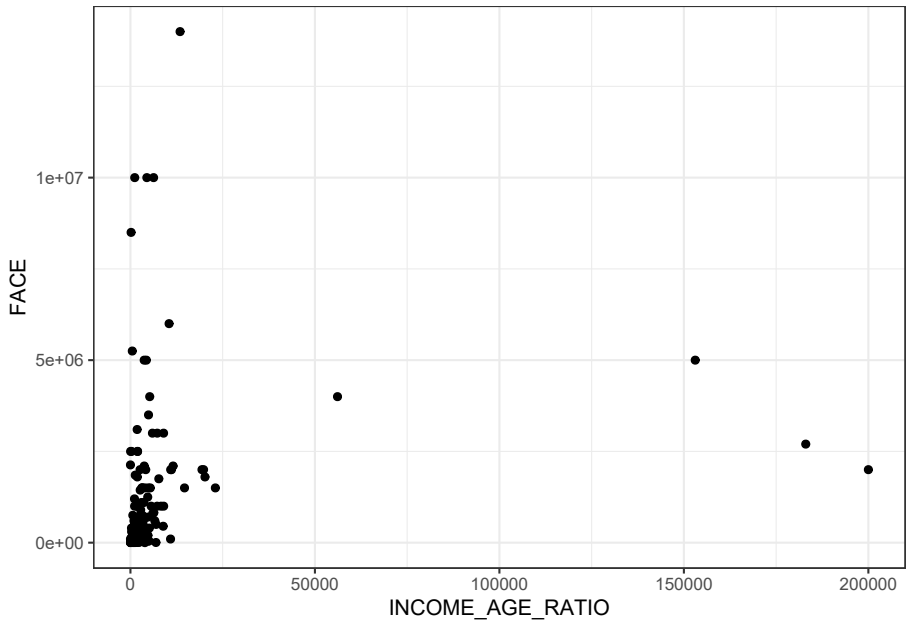
```
insurance %>%
  ggplot(aes(x = holders, y = claims)) +
  geom_point()
```



8.3 Data manipulation chaining

Pipes allow for data manipulations to be chained with visualizations.

```
termlife %>%
  filter(FACE > 0) %>%
  mutate(INCOME_AGE_RATIO = INCOME/AGE) %>%
  ggplot(aes(INCOME_AGE_RATIO, FACE)) +
  geom_point() +
  theme_bw()
```




```
set.seed(1)
library(ggplot2)
theme_set(theme_bw())
```

Chapter 9

Introduction to Modeling

About 40-50% of the exam grade is based on modeling.

9.1 Model Notation

The number of observations will be denoted by n. When we refer to the size of a data set, we are referring to n. We use p to refer the number of input variables used. The word "variables" is synonymous with "features". For example, in the `health_insurance` data, the variables are `age`, `sex`, `bmi`, `children`, `smoker` and `region`. These 7 variables mean that $p = 7$. The data is collected from 1,338 patients, which means that $n = 1,338$.

Scalar numbers are denoted by ordinary variables (i.e., $x = 2$, $z = 4$), and vectors are denoted by bold-faced letters

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

We use `y` to denote the target variable. This is the variable which we are trying to predict. This can be either a whole number, in which case we are performing *regression*, or a category, in which case we are performing *classification*. In the health insurance example, `y = charges`, which are the annual health care costs for a patient.

Both n and p are important because they tell us what types of models are likely to work well, and which methods are likely to fail. For the PA exam, we will be dealing with small n (<100,000) due to the limitations of the Prometric computers. We will use a small p (< 20) in order to make the data sets easier to interpret.

We organize these variables into matrices. Take an example with $p = 2$ columns and 3 observations. The matrix is said to be 3×2 (read as "2-by-3") matrix.

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{21} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix}$$

The target is

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

This represents the *unknown* quantity that we want to be able to predict. In the health care costs example, y_1 would be the costs of the first patient, y_2 the costs of the second patient, and so forth. The variables x_{11} and x_{12} might represent the first patient's age and sex respectively, where x_{i1} is the patient's age, and $x_{i2} = 1$ if the ith patient is male and 0 if female.

Machine learning is about using X to predict Y. We call this "y-hat", or simply the prediction. This is based on a function of the data X.

$$\hat{Y} = f(X)$$

This is almost never going to happen perfectly, and so there is always an error term, ϵ. This can be made smaller, but is never exactly zero.

$$\hat{Y} + \epsilon = f(X) + \epsilon$$

In other words, $\epsilon = y - \hat{y}$. We call this the *residual*. When we predict a person's health care costs, this is the difference between the predicted costs (which we had created the year before) and the actual costs that the patient experienced (of that current year).

9.2 Ordinary least squares (OLS)

The type of model used refers to the class of function of f. If f is linear, then we are using a linear model. Linear models are linear in the parameters, β.

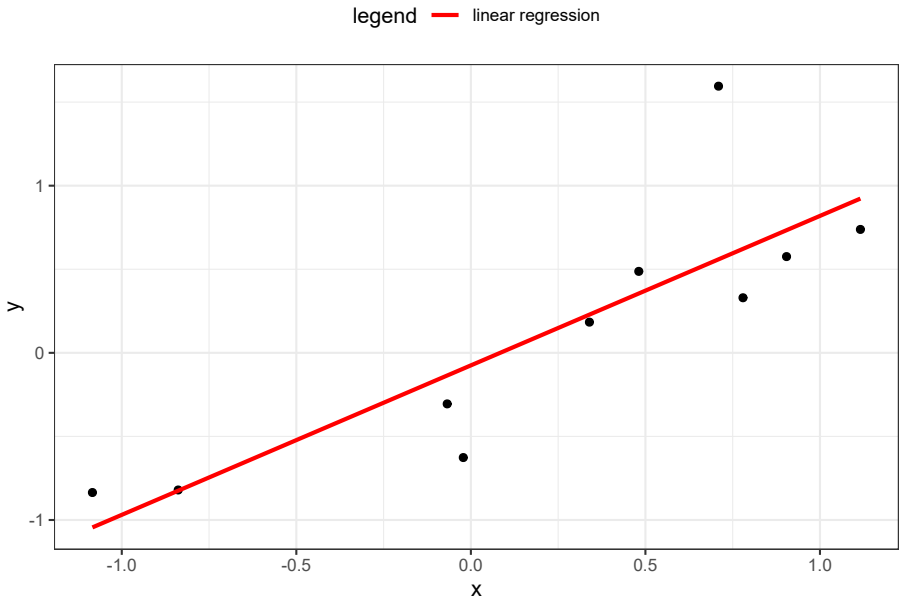
We observe the data X and the want to predict the target Y.

We find a β so that

$$\hat{Y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p$$

Which means that each y_i is a linear combination of the variables $x_1, ..., x_p$, plus a constant β_0 which is called the *intercept* term.

In the one-dimensional case, this creates a line connecting the points. In higher dimensions, this creates a hyperplane.



The question then is **how can we choose the best values of** β? First of all, we need to define what we mean by "best". Ideally, we will choose these values which will create close predictions of $\mathbf{y}$ on new, unseen data.

To solve for β, we first need to define a *loss function*. This allows us to compare how well a model is fitting the data. The most commonly used loss function is the residual sum of squares (RSS), also called the *squared error loss* or the L2 norm. When RSS is small, then the predictions are close to the actual values and the model is a good fit. When RSS is large, the model is a poor fit.

$$\text{RSS} = \sum_i (y_i - \hat{y})^2$$

When you replace $\hat{y}_i$ in the above equation with $\beta_0 + \beta_1 x_1 + ... + \beta_p x_p$, take the derivative with respect to β, set equal to zero, and solve, we can find the optimal values. This turns the problem of statistics into a problem of numeric optimization, which computers can do quickly.

You might be asking: why does this need to be the squared error? Why not the absolute error, or the cubed error? Technically, these could be used as well. In

fact, the absolute error (L1 norm) is useful in other models. Taking the square has a number of advantages.

- It provides the same solution if we assume that the distribution of $Y|X$ is guassian and maximize the likelihood function. This method is used for GLMs, in the next chapter.
- Empirically it has been shown to be less likely to overfit as compared to other loss functions

9.3 Example

In our health, we can create a linear model using `bmi`, `age`, and `sex` as an inputs.

The `formula` controls which variables are included. There are a few shortcuts for using R formulas.

Formula	Meaning
`charges` ~ `bmi` + `age`	Use `age` and `bmi` to predict `charges`
`charges` ~ `bmi` + `age` + `bmi`*`age`	Use `age`,`bmi` as well as an interaction to predict `charges`
`charges` ~ (`bmi` > 20) + `age`	Use an indicator variable for `bmi` > 20 `age` to predict `charges`
log(`charges`) ~ log(`bmi`) + log(`age`)	Use the logs of `age` and `bmi` to predict log(`charges`)
`charges` ~ .	Use all variables to predict `charges`

You can use formulas to create new variables (aka feature engineering). This can save you from needing to re-run code to create data.

Below we fit a simple linear model to predict charges.

```
library(ExamPAData)
library(tidyverse)

model <- lm(data = health_insurance, formula = charges ~ bmi + age)
```

The `summary` function gives details about the model. First, the `Estimate`, gives you the coefficients. The `Std. Error` is the error of the estimate for the coefficient. Higher standard error means greater uncertainty. This is relative to the average value of that variable. The `t value` tells you how "big" this error really is based on standard deviations. A larger `t value` implies a low probability of the null hypothesis being accepted saying that the coefficient is zero. This is the same as having a p-value (`Pr (>|t|)`) being close to zero.

The little *, **, *** indicate that the variable is either somewhat significant, significant, or highly significant. "significance" here means that there is a low probability of the coefficient being that size (or larger) if there were *no actual casual relationship*, or if the data was random noise.

```
summary(model)
```

```
##
## Call:
## lm(formula = charges ~ bmi + age, data = health_insurance)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -14457  -7045  -5136   7211  48022
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6424.80    1744.09  -3.684 0.000239 ***
## bmi           332.97      51.37   6.481 1.28e-10 ***
## age           241.93      22.30  10.850  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11390 on 1335 degrees of freedom
## Multiple R-squared:  0.1172, Adjusted R-squared:  0.1159
## F-statistic:  88.6 on 2 and 1335 DF,  p-value: < 2.2e-16
```

When evaluating model performance, you should not rely on the `summary` alone as this is based on the training data. To look at performance, test the model on validation data. This can be done by either using a hold out set, or using cross-validation, which is even better.

Let's create an 80% training set and 20% testing set. You don't need to worry about understanding this code as the exam will always give this to you.

```
set.seed(1)
library(caret)
#create a train/test split
index <- createDataPartition(y = health_insurance$charges,
                             p = 0.8, list = F) %>% as.numeric()
train <-  health_insurance %>% slice(index)
test <- health_insurance %>% slice(-index)
```

Train the model on the `train` and test on `test`.

```
model <- lm(data = train, formula = charges ~ bmi + age)
pred = predict(model, test)
```

Let's look at the Root Mean Squared Error (RMSE).

```
get_rmse <- function(y, y_hat){
  sqrt(mean((y - y_hat)^2))
}

get_rmse(pred, test$charges)
```

```
## [1] 11421.96
```

The above number does not tell us if this is a good model or not by itself. We need a comparison. The fastest check is to compare against a prediction of the mean. In other words, all values of the `y_hat` are the average of `charges`

```
get_rmse(mean(test$charges), test$charges)
```

```
## [1] 12574.97
```

The RMSE is **higher** (worse) when using just the mean, which is what we expect. **If you ever fit a model and get an error which is worse than the average prediction, something must be wrong.**

The next test is to see if any assumptions have been violated.

First, is there a pattern in the residuals? If there is, this means that the model is missing key information. For the model below, this is a **yes**, which means that this is a bad model. Because this is just for illustration, I'm going to continue using it, however.

```
plot(model, which = 1)
```

The normal QQ shows how well the quantiles of the predictions fit to a theoretical normal distribution. If this is true, then the graph is a straight 45-degree line. In this model, you can definitely see that this is not the case. If this were a good model, this distribution would be closer to normal.

```
plot(model, which = 2)
```

Once you have chosen your model, you should re-train over the entire data set. This is to make the coefficients more stable because `n` is larger. Below you can see that the standard error is lower after training over the entire data set.

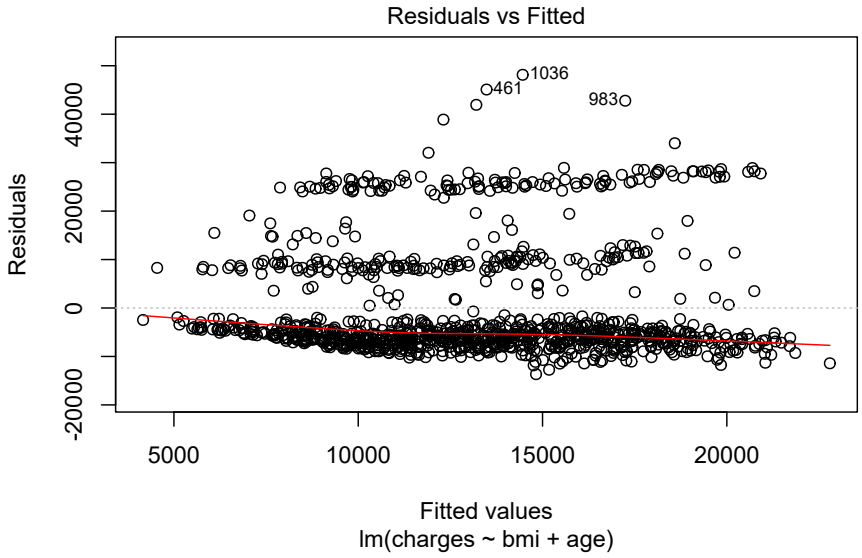


Figure 9.1: Residuals vs. Fitted

```
all_data <- lm(data = health_insurance,
               formula = charges ~ bmi + age)
testing <- lm(data = test,
              formula = charges ~ bmi + age)
```

term	full_data_std_error	test_data_std_error
(Intercept)	1744.1	3824.2
bmi	51.4	111.1
age	22.3	47.8

All interpretations should be based on the model which was trained on the entire data set. Obviously, this only makes a difference if you are interpreting the precise values of the coefficients. If you are just looking at which variables are included, or at the size and sign of the coefficients, then this would not change.

```
coefficients(model)
```

```
## (Intercept)         bmi         age
##  -4526.5284    286.8283    228.4372
```

Translating the above into an equation we have

$$\hat{y}_i = -4,526 + 287\text{bmi} + 228\text{age}$$

For example, if a patient has `bmi = 27.9` and `age = 19` then predicted value is

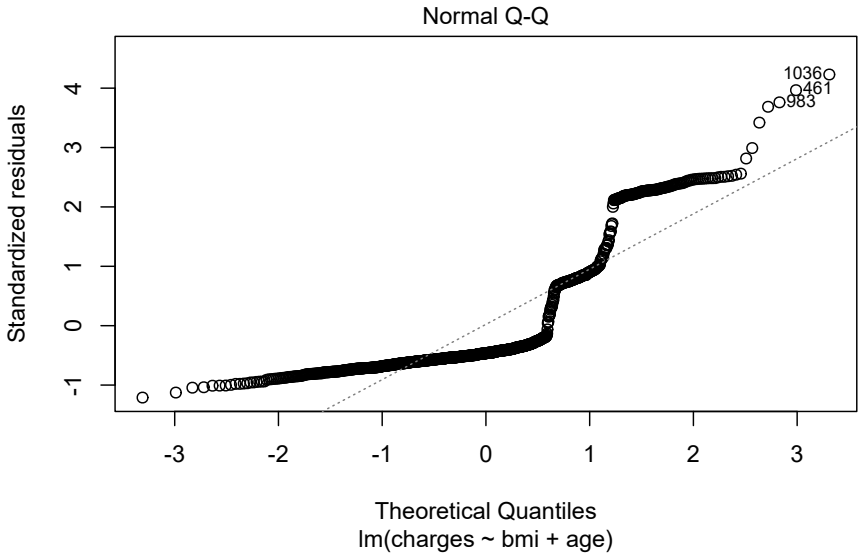


Figure 9.2: Normal Q-Q

Chapter 10

Generalized linear models (GLMs)

The linear model that we have considered up to this point, what we called "OLS", assumes that the response is a linear combination of the predictor variables. For an error term $\epsilon_i \sim N(0, \sigma^2)$, this is assumes that

$$Y = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p + \epsilon$$

In matrix notation, if X is the matrix made up of columns $X_1, ..., X_p$, then

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon$$

Another way of saying this is that "after we adjust for the data, the error is normally distributed and the variance is constant." If I is an n-by-in identity matrix, and $\sigma^2 I$ is the covariance matrix, then

$$\mathbf{Y}|\mathbf{X} \sim N(\mathbf{X}\beta, \sigma^{\mathbf{2}} I)$$

Because this notation is getting too cumbersome, we're going to stop using bold letters to denote matrices and just use non-bold characters. From now on, $\mathbf{X}$ is the same as X.

These assumptions can be expressed in two parts:

1. A *random component*: The response variable $Y|X$ is normally distributed with mean $\mu = \mu(X) = E(Y|X)$
2. A link between the response and the covariates (also known as the systemic component) $\mu(X) = X\beta$

In words, this is saying that each observation follows a normal distribution which has a mean that is equal to the linear predictor.

10.1 The generalized linear model

Just as the name implies, GLMs are more *general* in that they are more flexible. We relax these two assumptions by saying that the model is defined by

1. A random component: $Y|X \sim$ some exponential family distribution
2. A link: between the random component and covariates:

$$g(\mu(X)) = X\beta$$

where g is called the *link function* and $\mu = E[Y|X]$.

In words, this is saying that each observation follows *some type of exonential distrubution* (Gamma, Inverse Gaussian, Poisson, etc.) and that distribution has a mean which is related to the linear predictor through the link function. Additionally, there is a *dispersion* parameter, but that is more more info that is needed here. For an explanation, see Ch. 2.2 of CAS Monograph 5.

The possible combinations of link functions and distribution families are summarized nicely on Wikipedia.

For this exam, a common question is to ask candiates to choose the best distribution and link function. There is no all-encompasing answer, but a few suggestions are

- If Y is counting something, such as the number of claims, number of accidents, or some other discrete and positive counting sequence, use the Poisson;
- If Y contains negative values, then do not use the Exponential, Gamma, or Inverse Gaussian as these are strictly positive. Conversely, if Y is only positive, such as the price of a policy (price is always > 0), or the claim costs, then these are good choices;
- If Y is binary, the the binomial response with either a Probit or Logit link. The Logit is more common.
- If Y has more than two categories, the multinomial distribution with either the Probit or Logic link (See Logistic Regression)

Figure 10.1: Distribution-Link Function Combinations

10.2 Interpretation

The exam will always ask you to interpret the GLM. These questions can usually be answered by inverting the link function and interpreting the coefficients. In the case of the log link, simply take the exponent of the coefficients and each of these represents a "relativity" factor.

$$log(\hat{\mathbf{y}}) = \mathbf{X}\beta \Rightarrow \hat{\mathbf{y}} = e^{\mathbf{X}\beta}$$

For a single observation y_i, this is

$$\exp(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}) = e^{\beta_0} e^{\beta_1 x_{i1}} e^{\beta_2 x_{i2}} ... e^{\beta_p x_{ip}} = R_0 R_2 R_3 ... R_p$$

Where R_k is the *relativity* of the kth variable. This terminology is from insurance ratemaking, where actuaries need to be able to explain the impact of each variable in pricing insurance. The data science community does not use this language.

For binary outcomes with logit or probit link, there is no easy interpretation. This has come up in at least one past sample exam, and the solution was to create "psuedo" observations and observe how changing each x_k would change the predicted value. Due to the time requirements, this is unlikely to come up on an exam. So if you are asked to use a logit or probit link, saying that the result is not easy to interpret should suffice.

10.3 Residuals

The word "residual" by itself actually means the "raw residual" in GLM language. This is the difference in actual vs. predicted values.

$$\text{Raw Residual} = y_i - \hat{y_i}$$

This are not meaningful for GLMs with non-Gaussian response families because the distribution changes depending on the response family chosen. To adjust for this, we need the concept of *deviance residual*.

To paraphrase from this paper from the University of Oxford:

www.stats.ox.ac.uk/pub/bdr/IAUL/ModellingLecture5.pdf

Deviance is a way of assessing the adequacy of a model by comparing it with a more general model with the maximum number of parameters that can be estimated. It is referred to as the saturated model. In the saturated model

there is basically one parameter per observation. The deviance assesses the goodness of fit for the model by looking at the difference between the log-likelihood functions of the saturated model and the model under investigation, i.e. $l(b_{sat}, y) - l(b, y)$. Here sat b_{sat} denotes the maximum likelihood estimator of the parameter vector of the saturated model, β_{sat} , and b is the maximum likelihood estimator of the parameters of the model under investigation, β. The maximum likelihood estimator is the estimator that maximises the likelihood function. **The deviance is defined as**

$$D = 2[l(b_{sat}, y) - l(b, y)]$$

The deviance residual uses the deviance of the ith observation d_i and then takes the square root and applies the same sign (aka, the + or - part) of the raw residual.

$$\text{Deviance Residual} = \text{sign}(y_i - \hat{y}_i)\sqrt{d_i}$$

10.4 Example

Just as with OLS, there is a `formula` and `data argument`. In addition, we need to specify the response distribution and link function.

```
model = glm(formula = charges ~ age + sex + smoker,
            family = Gamma(link = "log"),
            data = health_insurance)
```

We see that `age`, `sex`, and `smoker` are all significant (p <0.01). Reading off the coefficient signs, we see that claims

- Increase as age increases
- Are higher for women
- Are higher for smokers

```
model %>% tidy()
```

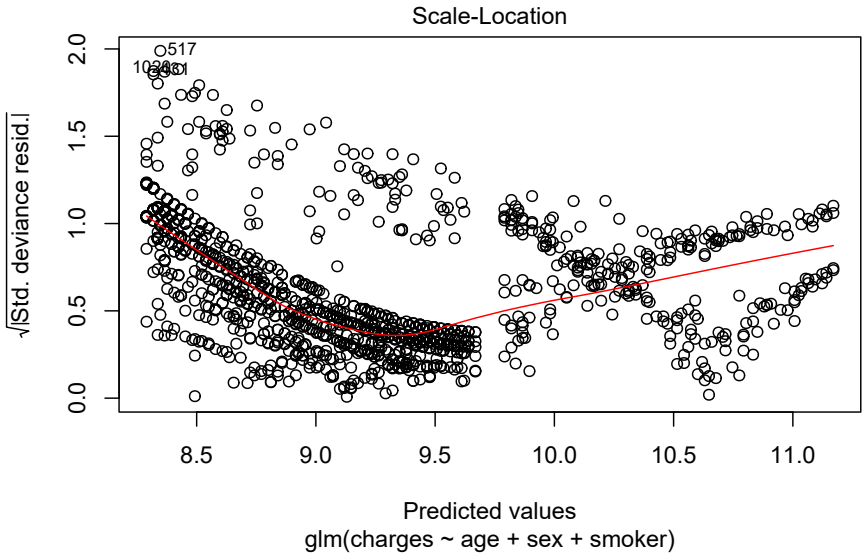
```
## # A tibble: 4 x 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)   7.82     0.0600     130.   0.
## 2 age           0.0290   0.00134     21.6  3.40e- 89
## 3 sexmale      -0.0468   0.0377      -1.24 2.15e-  1
## 4 smokeryes     1.50     0.0467      32.1  3.25e-168
```

Below you can see graph of deviance residuals vs. the predicted values.

If this were a perfect model, all of these below assumptions would be met:

- Scattered around zero?
- Constant variance?
- No obvious pattern?

```
plot(model, which = 3)
```



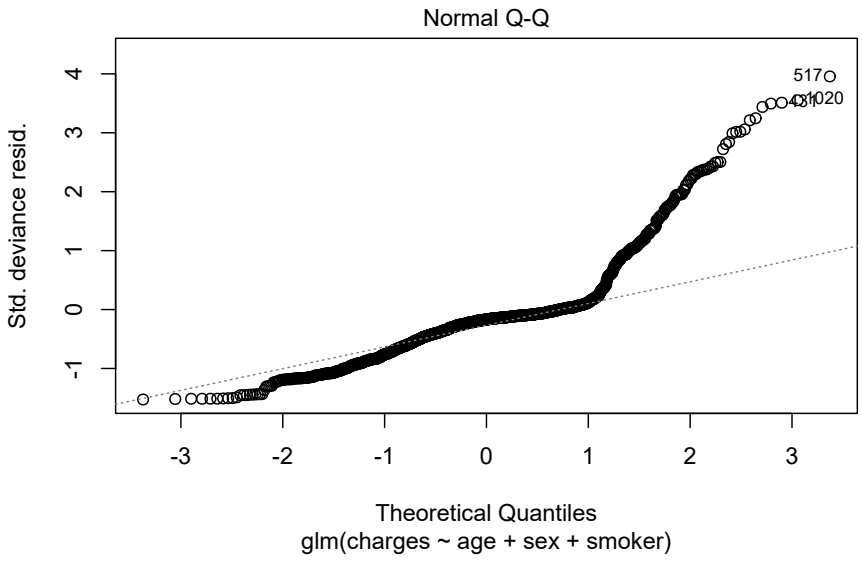
The quantile-quantile (QQ) plot shows the quantiles of the deviance residuals (i.e., after adjusting for the Gamma distribution) against theoretical Gaussian quantiles.

In a perfect model, all of these assumptions would be met:

- Points lie on a straight line?

- Tails are not significantly above or below line? Some tail deviation is ok.
- No sudden "jumps"? This indicates many Y's which have the same value, such as insurance claims which all have the exact value of $100.00 or $0.00.

```
plot(model, which = 2)
```



10.5 Combinations of Link and Response Family Examples

What is an example of when to use a log link with a guassian response? What about a Gamma family with an inverse link? What about an inverse Gaussian response and an inverse square link? As these questions illustrate, there are many combinations of link and response family. In the real world, a model rarely fits perfectly, and so often these choices come down to the judgement of the modeler - which model is the best fit and meets the business objectives?

However, there is one way that we can know for certain which link and response family is the best, and that is if we generate the data ourselves.

Recall that a GLM has two parts:

1. A **random component**: $Y|X \sim$ some exponential family distribution
2. A **link function**: between the random component and the covariates: $g(\mu(X)) = X\beta$ where $\mu = E[Y|X]$

Following this recipe, we can simulate data from any combination of link function and response family. This helps us to understand the GLM framework very clearly.

10.5.1 Gaussian Response with Log Link

We create a function that takes in data x and returns a guassian random variable that has mean equal to the inverse link, which in the case of a log link is the exponent. We add 10 to x so that the values will always be positive, as will be described later on.

```
sim_norm <- function(x) {
  rnorm(1, mean = exp(10 + x), sd = 1)
}
```

The values of X do not need to be normal. The above assumption is merely that the mean of the response Y is related to X through the link function, `mean = exp(10 + x)`, and that the distribution is normal. This has been accomplished with `rnorm` already. For illustration, here we use X's from a uniform distribution.

```
data <- tibble(x = runif(1000)) %>%
  mutate(y = x %>% map_dbl(sim_norm))
```

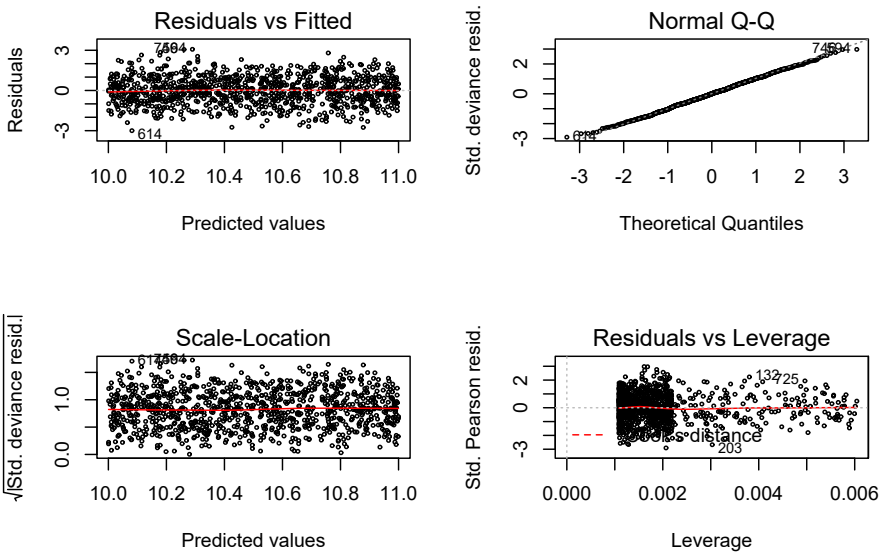
We already know what the answer is: a gaussian response with a log link. We fit a GLM and see a perfect fit.

```
glm <- glm(y ~ x, family = gaussian(link = "log"), data = data)

summary(glm)
```

```
##
## Call:
## glm(formula = y ~ x, family = gaussian(link = "log"), data = data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.0004  -0.6964   0.0005   0.7266   3.0718
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.000e+01  2.195e-06 4554982   <2e-16 ***
## x           1.000e+00  3.085e-06  324117   <2e-16 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 1.067056)
##
##     Null deviance: 1.2235e+11  on 999  degrees of freedom
## Residual deviance: 1.0649e+03  on 998  degrees of freedom
## AIC: 2906.8
##
## Number of Fisher Scoring iterations: 2
```

```
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



10.5.2 Gaussian Response with Inverse Link

The same steps are repeated except the link function is now the inverse, `mean = 1/x`. We see that some values of Y are negative, which is ok.

```
sim_norm <- function(x) {
  rnorm(1, mean = 1/x, 1)
}
```

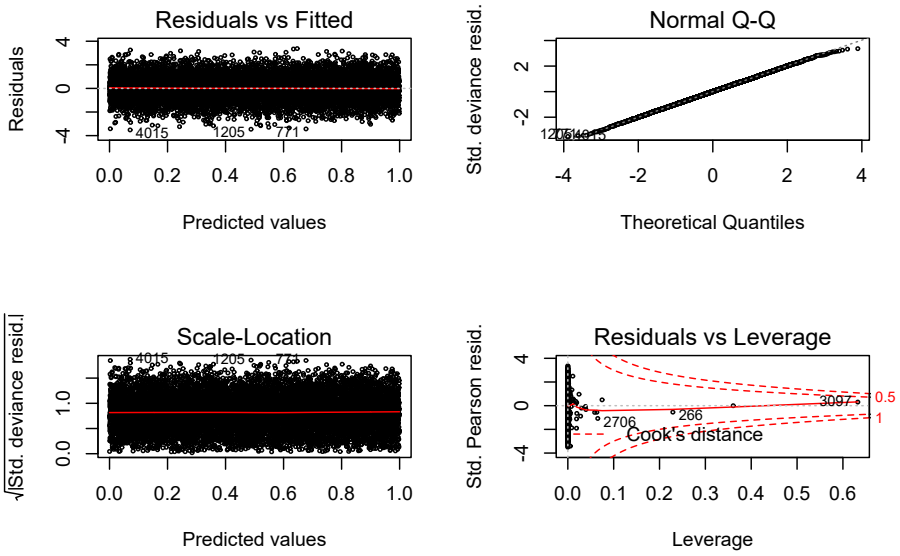
```
data <- tibble(x = runif(10000)) %>%
  mutate(y = x %>% map_dbl(sim_norm))
summary(data)
```

```
##        x                    y
##  Min.   :0.0001064   Min.   :  -1.957
##  1st Qu.:0.2532864   1st Qu.:   1.259
##  Median :0.5028875   Median :   2.334
##  Mean   :0.5018599   Mean   :  10.214
##  3rd Qu.:0.7507694   3rd Qu.:   4.232
##  Max.   :0.9998552   Max.   :9394.790
```

```
glm <- glm(y ~ x, family = gaussian(link = "inverse"), data = data)

summary(glm)
```

```
##
## Call:
## glm(formula = y ~ x, family = gaussian(link = "inverse"), data = data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.5186  -0.6747   0.0105   0.6883   3.3764
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 3.596e-08  2.587e-08    1.39    0.165
## x           9.998e-01  2.072e-04 4824.13   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 1.007483)
##
##     Null deviance: 203905704  on 9999  degrees of freedom
## Residual deviance:     10073  on 9998  degrees of freedom
## AIC: 28457
##
## Number of Fisher Scoring iterations: 4
```

```
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



10.5.3 Gaussian Response with Identity Link

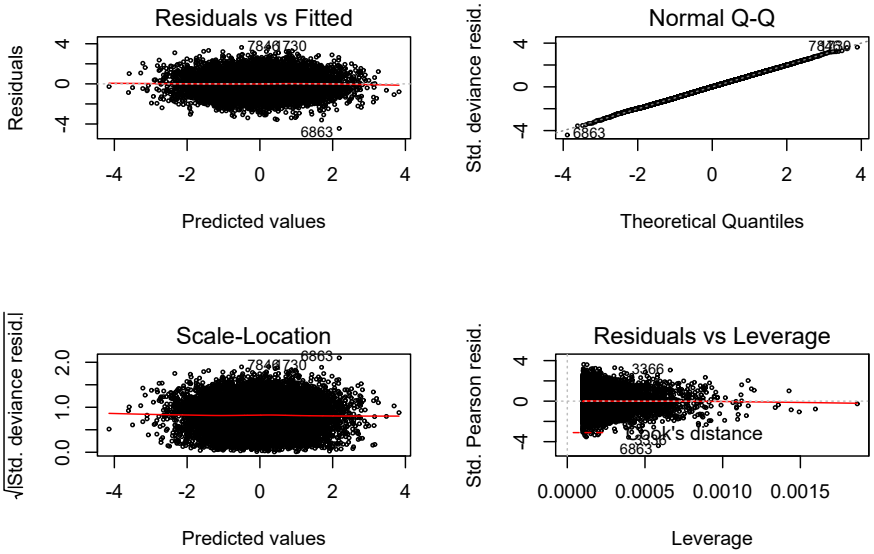
And now the link is the identity, `mean = x`.

```
sim_norm <- function(x) {
  rnorm(1, mean = x, 1)
}

data <- tibble(x = rnorm(10000)) %>%
  mutate(y = x %>% map_dbl(sim_norm))

glm <- glm(y ~ x, family = gaussian(link = "identity"), data = data)

summary(glm)
```

```
##
## Call:
## glm(formula = y ~ x, family = gaussian(link = "identity"), data = data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -4.4461  -0.6853   0.0129   0.6794   3.6661
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.01393    0.01010   1.379    0.168
## x            0.98236    0.01005  97.727   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 1.020901)
##
##     Null deviance: 19957  on 9999  degrees of freedom
## Residual deviance: 10207  on 9998  degrees of freedom
## AIC: 28590
##
## Number of Fisher Scoring iterations: 2
```

```
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



10.5.4 Gaussian Response with Log Link and Negative Values

By Gaussian response we say that the *mean* of the response is Gaussian. The range of a normal random variable is $(-\infty, +\infty)$, which means that negative

values are always possible. Now, if the mean is a large positive number, than negative values are much less likely but still possible: about 95% of the observations will be within 2 standard deviations of the mean.

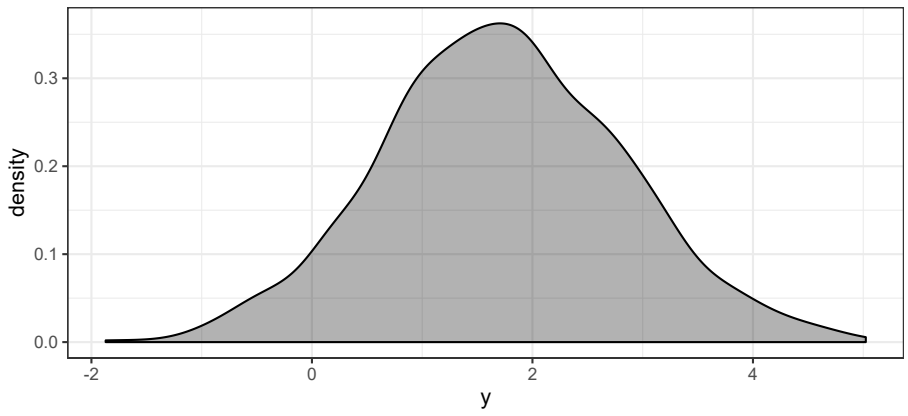
We see below that there are some Y values which are negative.

```
sim_norm <- function(x) {
  rnorm(1, mean = exp(x), sd = 1)
}

data <- tibble(x = runif(1000)) %>%
  mutate(y = x %>% map_dbl(sim_norm))
summary(data)
```

```
##        x                   y
##  Min.   :0.0000122   Min.   :-1.8709
##  1st Qu.:0.2354295   1st Qu.: 0.9815
##  Median :0.5055838   Median : 1.6969
##  Mean   :0.4968540   Mean   : 1.7275
##  3rd Qu.:0.7611768   3rd Qu.: 2.4854
##  Max.   :0.9993455   Max.   : 5.0233
```

We can also see this from the histogram.

```
data %>% ggplot(aes(y)) + geom_density( fill = 1, alpha = 0.3)
```



If we try to fit a GLM with a log link, there is an error.

```
glm <- glm(y ~ x, family = gaussian(link = "log"), data = data)
```

```
Error in eval(family$initialize) : cannot find valid starting
values: please specify some
```

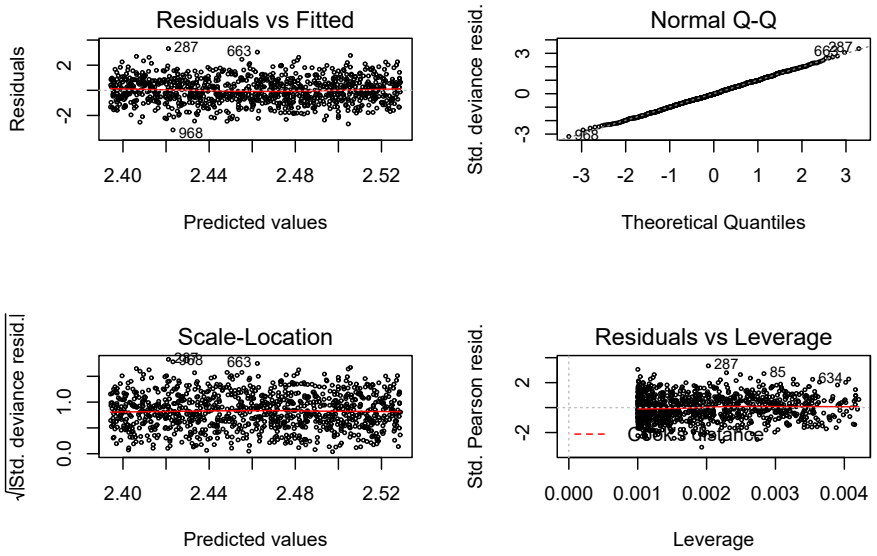
This is because the domain of the natural logarithm only includes positive numbers, and we just tried to take the log of negative numbers.

Our initial reaction might be to add some constant to each Y, say 10 for instance, so that they are all positive. This does produce a model which is a good fit.

```
glm <- glm(y + 10 ~ x, family = gaussian(link = "log"), data = data)
summary(glm)
```

```
##
## Call:
## glm(formula = y + 10 ~ x, family = gaussian(link = "log"), data = data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.1527  -0.6538  -0.0336   0.6753   3.3219
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.394232   0.005463  438.25   <2e-16 ***
## x           0.134685   0.009158   14.71   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.987688)
##
##     Null deviance: 1198.70  on 999  degrees of freedom
## Residual deviance:  985.71  on 998  degrees of freedom
## AIC: 2829.5
##
## Number of Fisher Scoring iterations: 4
```

```
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



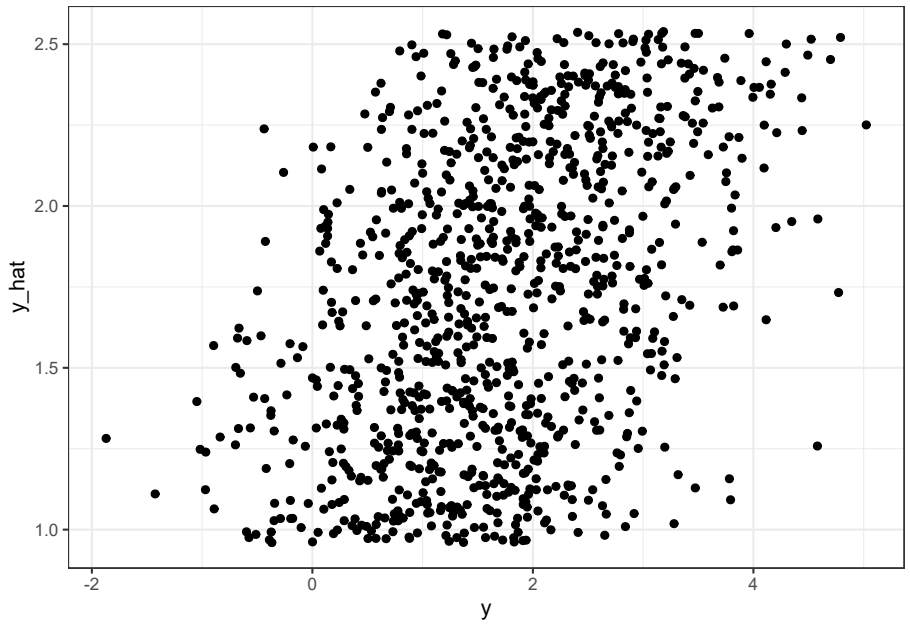
We see that on average, the predictions are 10 higher than the target. This is no surprise since $E[Y + 10] = E[Y] + 10$.

```
y <- data$y
y_hat <- predict(glm, type = "response")
mean(y_hat) - mean(y)
```

```
## [1] 9.99995
```

But we see that the actual predictions are bad. If we were to loot at the R-squared, MAE, RMSE, or any other metric it would tell us the same story. This is because our GLM assumption is **not** that Y is related to the link function of X, but that the **mean** of Y is.

```
tibble(y = y, y_hat = y_hat - 10) %>% ggplot(aes(y, y_hat)) + geom_point()
```

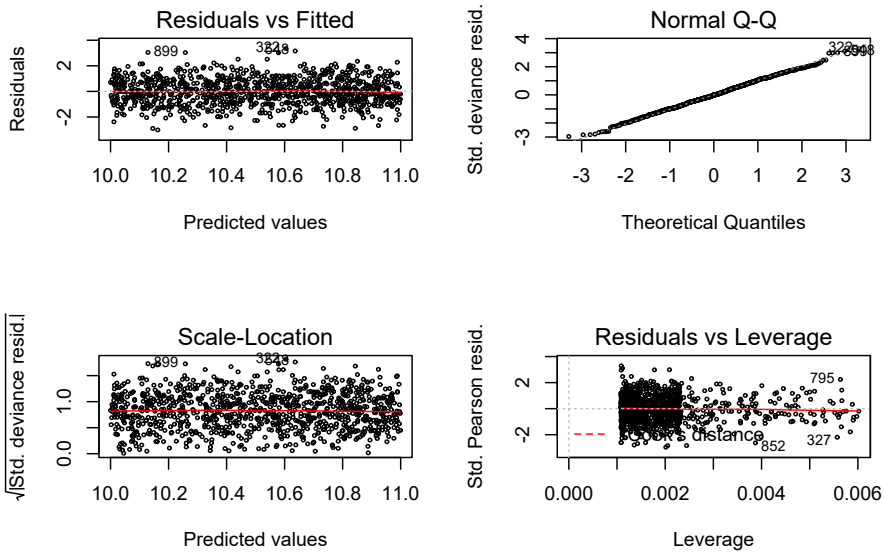


One solution is to adjust the X which the model is based on. Add a constant term to X so that the mean of Y is larger, and hence Y is non zero. While is a viable approach in the case of only one predictor variable, with more predictors this would not be easy to do.

```
data <- tibble(x = runif(1000) + 10) %>%
  mutate(y = x %>% map_dbl(sim_norm))
summary(data)
```

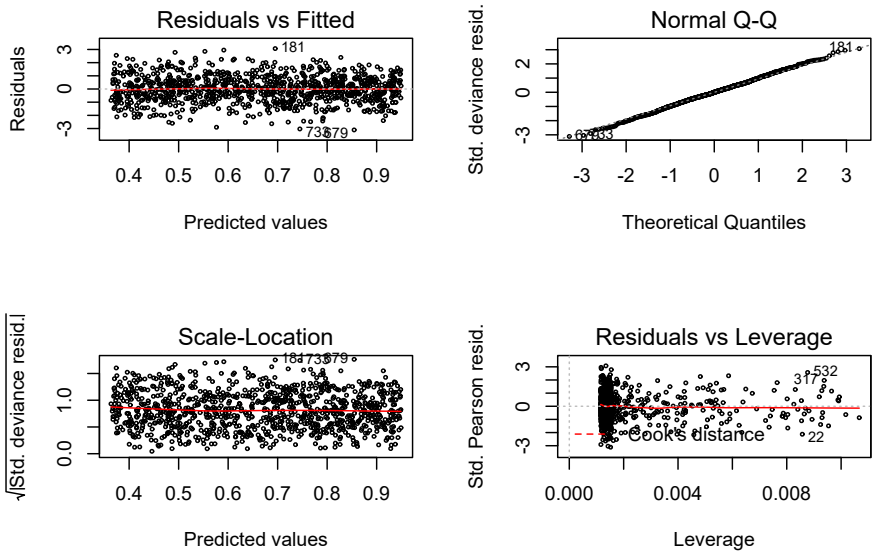
```
##        x               y
##  Min.   :10.00   Min.   :22028
##  1st Qu.:10.25   1st Qu.:28291
##  Median :10.52   Median :36893
##  Mean   :10.51   Mean   :38160
##  3rd Qu.:10.77   3rd Qu.:47441
##  Max.   :11.00   Max.   :59842
```

```
glm <- glm(y ~ x, family = gaussian(link = "log"), data = data)
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



A better approach may be to use an inverse link even though the data was generated from a log link. This is a good illustration of the saying "all models are wrong, but some are useful" in that the statistical assumption of the model is not correct but the model still works.

```
data <- tibble(x = runif(1000)) %>%
  mutate(y = x %>% map_dbl(sim_norm))
glm <- glm(y ~ x, family = gaussian(link = "inverse"), data = data)
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```




```
summary(glm)
```

```
##
## Call:
## glm(formula = y ~ x, family = gaussian(link = "inverse"), data = data)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -3.10622  -0.63739  -0.00542   0.63167   3.06741
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.94957    0.03515   27.02   <2e-16 ***
## x           -0.58667    0.04360  -13.46   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.9967997)
##
##     Null deviance: 1218.78  on 999  degrees of freedom
## Residual deviance:  994.82  on 998  degrees of freedom
## AIC: 2838.7
##
## Number of Fisher Scoring iterations: 6
```

10.5.5 Gamma Response with Log Link

The gamma distribution with rate parameter α and scale parameter θ is density.

$$f(y) = \frac{(y/\theta)^{\alpha}}{x\Gamma(\alpha)} e^{-x/\theta}$$

The mean is $\alpha\theta$.

Let's use a gamma with shape 2 and scale 0.5, which has mean 1.

```
gammas <- rgamma(1000, shape=2, scale = 0.5)
mean(gammas)
```

```
## [1] 0.9873887
```

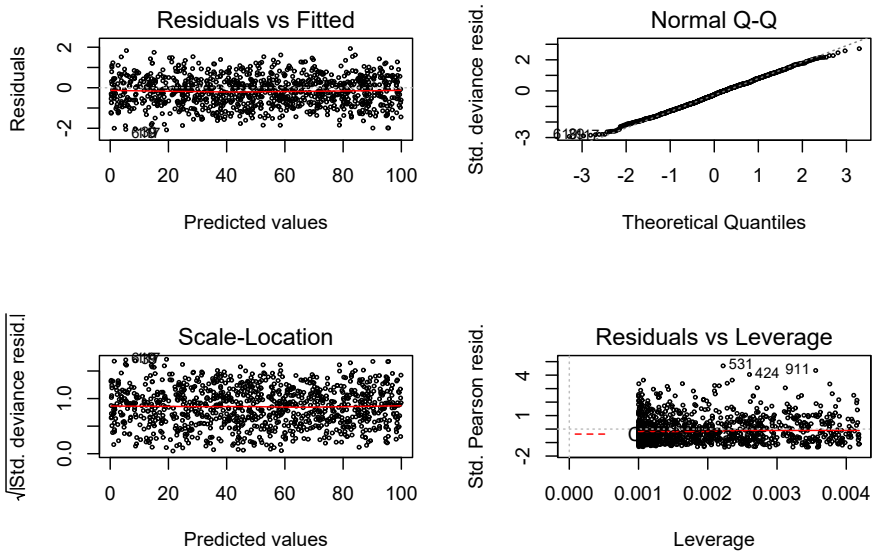
We then generate random gamma values. Because the mean now depends on two paramters instead of one, which was just μ in the Guassian case, we need to use a slightly different approach to simulate the random values. The link function here is seen in `exp(x)`.

```
#random component
x <- runif(1000, min=0, max=100)

#relate Y to X with a log link function
y <- gammas*exp(x)

data <- tibble(x = x, y  = y)
summary(data)
```

```
##        x                  y
##  Min.   : 0.2452   Min.   :0.000e+00
##  1st Qu.:27.0464   1st Qu.:4.946e+11
##  Median :51.0196   Median :1.057e+22
##  Mean   :51.0666   Mean   :2.531e+41
##  3rd Qu.:75.3442   3rd Qu.:4.239e+32
##  Max.   :99.9213   Max.   :2.693e+43
```

As expected, the residual plots are all perfect because the model is perfect.

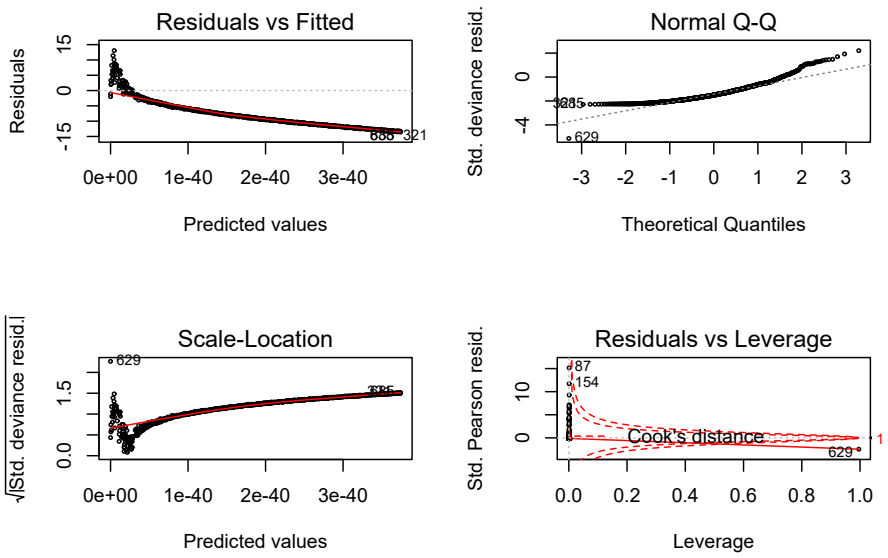
```
glm <- glm(y ~ x, family = Gamma(link = "log"), data = data)
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



If we had tried using an inverse instead of the log, the residual plots would look much worse.

```
glm <- glm(y ~ x, family = Gamma(link = "inverse"), data = data)
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```

```
## Warning in sqrt(crit * p * (1 - hh)/hh): NaNs produced

## Warning in sqrt(crit * p * (1 - hh)/hh): NaNs produced
```



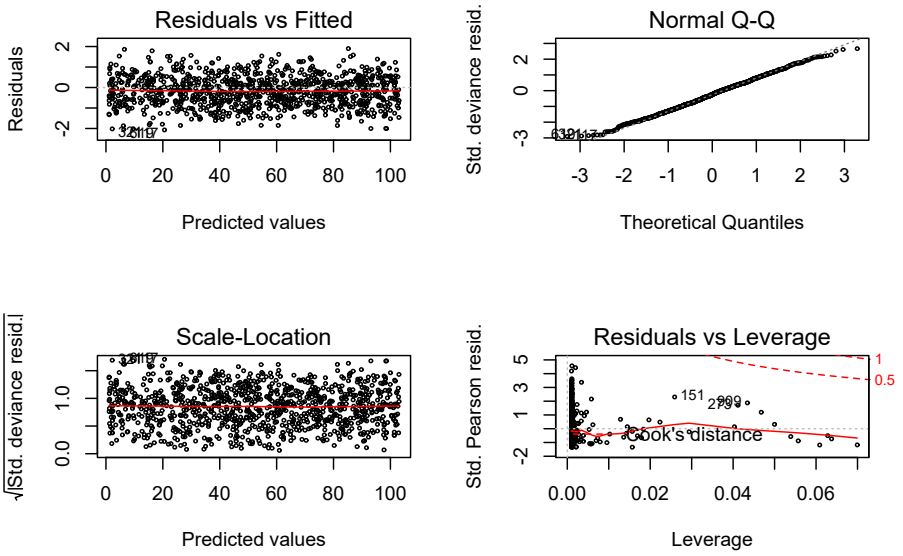
10.5.6 Gamma with Inverse Link

With the inverse link, the mean has a factor `1/(x + 1)`. Note that we need to add 1 to x to avoid dividing by zero.

```
#relate Y to X with a log link function
y <- gammas*1/(x + 1)

data <- tibble(x = x, y  = y)
summary(data)
```

```
##        x                y
##  Min.   : 0.2452   Min.   :0.0005277
##  1st Qu.:27.0464   1st Qu.:0.0084840
##  Median :51.0196   Median :0.0167640
##  Mean   :51.0666   Mean   :0.0485119
##  3rd Qu.:75.3442   3rd Qu.:0.0365838
##  Max.   :99.9213   Max.   :1.7125489
```

```
glm <- glm(y ~ x, family = Gamma(link = "inverse"), data = data)
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



10.6 Log transforms of continuous predictors

When a log link is used, taking the natural logs of continuous variables allows for the scale of each predictor to match the scale of the thing that they are predicting, the log of the mean of the response. In addition, when the distribution of the continuous variable is skewed, taking the log helps to make it more symmetric.

After taking the log of a predictor, the interpretation becomes a *power transform* of the original variable.

For μ the mean response,

$$log(\mu) = \beta_0 + \beta_1 log(X)$$

To solve for μ, take the exonent of both sides

$$\mu = e^{\beta_1} e^{\beta_1 log(X)} = e^{\beta_0} X^{\beta_1}$$

10.7 Reference levels

When a categorical variable is used in a GLM, the model actually uses indicator variables for each level. The default reference level is the order of the R factors.

For the `sex` variable, the order is `female` and then `male`. This means that the base level is `female` by default.

```
health_insurance$sex %>% as.factor() %>% levels()
```

```
## [1] "female" "male"
```

Why does this matter? Statistically, the coefficients are most stable when there are more observations.

```
health_insurance$sex %>% as.factor() %>% summary()
```

```
## female   male
##    662    676
```

There is already a function to do this in the `tidyverse` called `fct_infreq`. Let's quickly fix the `sex` column so that these factor levels are in order of frequency.

```
health_insurance <- health_insurance %>%
  mutate(sex = fct_infreq(sex))
```

Now `male` is the base level.

```
health_insurance$sex %>% as.factor() %>% levels()
```

```
## [1] "male"   "female"
```

10.8 Interactions

An interaction occurs when the effect of a variable on the response is different depending on the level of other variables in the model.

Consider this model:

Let x_2 be an indicator variable, which is 1 for some records and 0 otherwise.

$$\hat{y}_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$$

There are now two different linear models dependong on whether `x_1` is 0 or 1.

When $x_1 = 0$,

$$\hat{y}_i = \beta_0 + \beta_2 x_2$$

and when $x_1 = 1$

$$\hat{y}_i = \beta_0 + \beta_1 + \beta_2 x_2 + \beta_3 x_2$$

By rewriting this we can see that the intercept changes from β_0 to β_0^* and the slope changes from β_1 to β_1^*

$$(\beta_0 + \beta_1) + (\beta_2 + \beta_3)x_2 = \beta_0^* + \beta_1^* x_2$$

The SOA's modules give an example with the using age and gender as below. This is not a very strong interaction, as the slopes are almost identical across `gender`.

```
interactions %>%
  ggplot(aes(age, actual, color = gender)) +
  geom_line() +
  labs(title = "Age vs. Actual by Gender",
       subtitle = "Interactions imply different slopes",
       caption= "data: interactions")
```

Here is a clearer example from the `auto_claim` data. The lines show the slope of a linear model, assuming that only `BLUEBOOK` and `CAR_TYPE` were predictors in the model. You can see that the slope for Sedans and Sports Cars is higher than for Vans and Panel Trucks.

```
auto_claim %>%
  ggplot(aes(log(CLM_AMT), log(BLUEBOOK), color = CAR_TYPE)) +
  geom_point(alpha = 0.3) +
  geom_smooth(method = "lm", se = F) +
  labs(title = "Kelly Bluebook Value vs Claim Amount")
```

Any time that the effect that one variable has on the response is different depending on the value of other variables we say that there is an interaction. We can also use an hypothesis test with a GLM to check this. Simply include an interaction term and see if the coefficient is zero at the desired significance level.

10.9 Poisson Regression

When counting something, numbers can only be positive and increase by increments of 1. Statistically, the name for this is a Poisson Process, which is a model

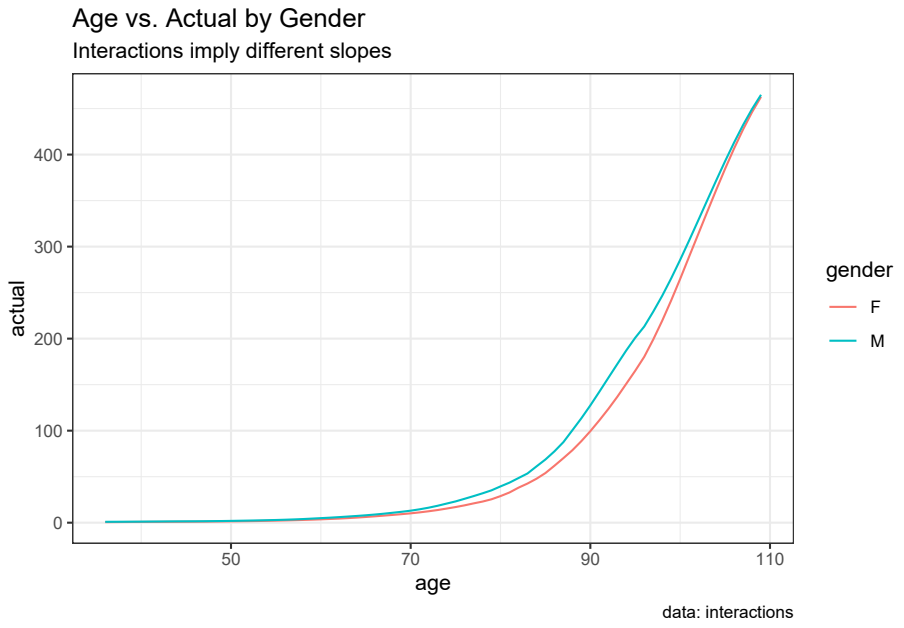


Figure 10.2: Example of weak interaction

for a serious of discrete events where the average time between events is known, called the "rate" λ, but the exact timing of events is unknown. We could just fit a single rate for all observations, but this would often be a simplification. For a time interval of length m, the expected number of events is λm.

By using a GLM, we can fit a different rate for each observation. Because the response is a count, the appropriate response distribution is the Poisson.

$$Y_i|X_i \sim \text{Poisson}(\lambda_i m_i)$$

When all observations have the same exposure, $m = 1$. When the mean of the data is far from the variance, an additional parameter known as the *dispersion parameter* is used. A classic example is when modeling insurance claim counts which have a lot of zero claims. Then the model is said to be an "over-dispersed Poisson" or "zero-inflated" model.

10.10 Offsets

In certain situations, it is convenient to include a constant term in the linear predictor. This is the same as including a variable that has a coefficient equal

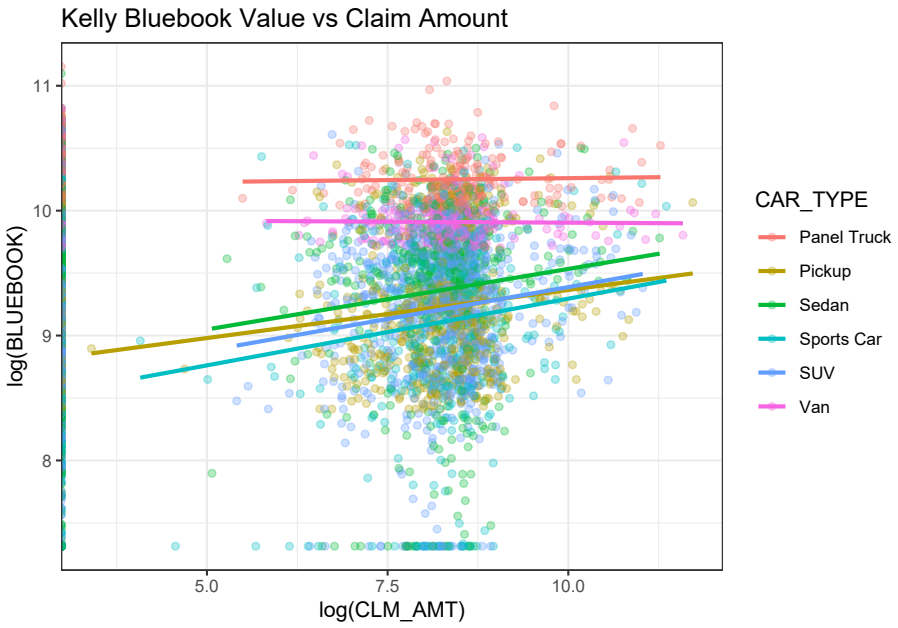


Figure 10.3: Example of strong interaction

to 1. We call this an *offset*.

$$g(\mu) = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p + \text{offset}$$

10.11 Tweedie regression

While this topic is briefly mentioned on the modules, the only R libraries which support Tweedie Regression (`statmod` and `tweedie`) are not on the syllabus, and so there is no way that the SOA could ask you to build a tweedie model. This means that you can be safely skip this section.

10.12 Stepwise subset selection

In theory, we could test all possible combinations of variables and interaction terms. This includes all p models with one predictor, all p-choose-2 models with two predictors, all p-choose-3 models with three predictors, and so forth. Then we take whichever model has the best performance as the final model.

This "brute force" approach is statistically ineffective: the more variables which are searched, the higher the chance of finding models that overfit.

A subtler method, known as *stepwise selection*, reduces the chances of overfitting by only looking at the most promising models.

Forward Stepwise Selection:

1. Start with no predictors in the model;
2. Evaluate all p models which use only one predictor and choose the one with the best performance (highest R^2 or lowest RSS);
3. Repeat the process when adding one additional predictor, and continue until there is a model with one predictor, a model with two predictors, a model with three predictors, and so forth until there are p models;
4. Select the single best model which has the best AIC,BIC, or adjusted R^2.

Backward Stepwise Selection:

1. Start with a model that contains all predictors;
2. Create a model which removes all predictors;
3. Choose the best model which removes all-but-one predictor;
4. Choose the best model which removes all-but-two predictors;
5. Continue until there are p models;
6. Select the single best model which has the best AIC,BIC, or adjusted R^2.

Both Forward & Backward Selection:

A hybrid approach is to consider use both forward and backward selection. This is done by creating two lists of variables at each step, one from forward and one from backward selection. Then variables from *both* lists are tested to see if adding or subtracting from the current model would improve the fit or not. ISLR does not mention this directly, however, by default the `stepAIC` function uses a default of `both`.

> **Tip**: Always load the `MASS` library before `dplyr` or `tidyverse`. Otherwise there will be conflicts as there are functions named `select()` and `filter()` in both. Alternatively, specify the library in the function call with `dplyr::select()`.

Readings
CAS Monograph 5 Chapter 2

10.13 Advantages and disadvantages

There is usually at least one question on the PA exam which asks you to "list some of the advantages and disadvantages of using this particular model", and so here is one such list. It is unlikely that the grader will take off points for including too many comments and so a good strategy is to include everything that comes to mind.

GLM Advantages

- Easy to interpret
- Can easily be deployed in spreadsheet format
- Handles skewed data through different response distributions
- Models the average response which leads to stable predictions on new data
- Handles continuous and categorical data

GLM Disadvantages

- Does not select features (without stepwise selection)
- Strict assumptions around distribution shape, randomness of error terms, and variable correlations
- Unable to detect non-linearity directly (although this can manually be addressed through feature engineering)
- Sensitive to outliers
- Low predictive power

Chapter 11

Logistic Regression

11.1 Model form

Logistic regression is a special type of GLM. The name is confusing because the objective is *classification* and not regression. While most examples focus on binary classification, logistic regression also works for multiclass classification.

The model form is as before

$$g(\hat{\mathbf{y}}) = \mathbf{X}\beta$$

However, now the target y_i is a category. Our objective is to predict a probability of being in each category. For regression, $\hat{y_i}$ can be any number, but now we need $0 \leq \hat{y_i} \leq 1$.

We can use a special link function, known as the *standard logistic function*, *sigmoid*, or *logit*, to force the output to be in this range of $\{0, 1\}$.

$$\hat{\mathbf{y}} = g^{-1}(\mathbf{X}\beta) = \frac{1}{1 + e^{-\mathbf{X}\beta}}$$

Other link functions for classification problems are possible as well, although the logistic function is the most common. If a problem asks for an alternative link, such as the *probit*, fit both models and compare the performance.

11.2 Example

Using the `auto_claim` data, we predict whether or not a policy has a claim. This is also known as the *claim frequency*.

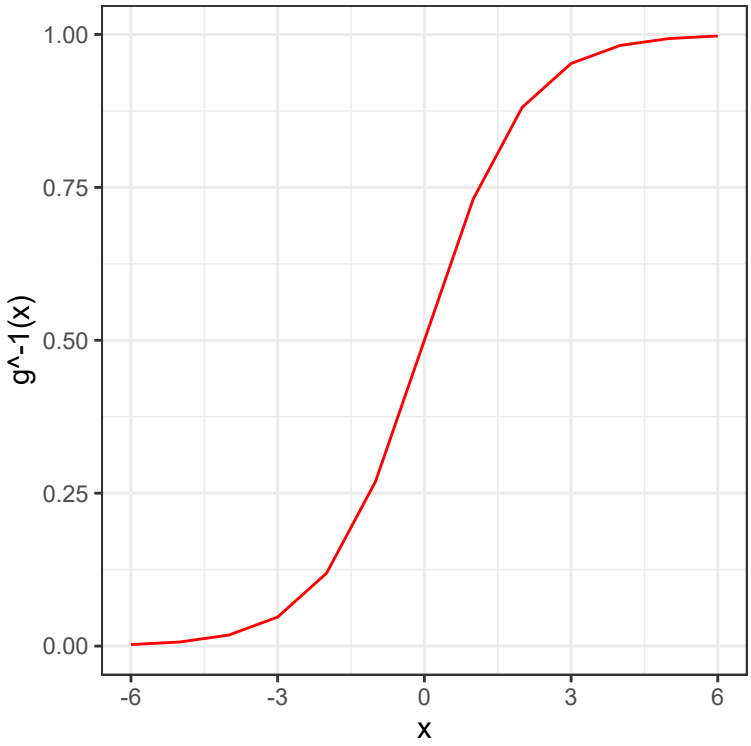


Figure 11.1: Standard Logistic Function

```
auto_claim %>% count(CLM_FLAG)
```

```
## # A tibble: 2 x 2
##   CLM_FLAG     n
##   <chr>    <int>
## 1 No        7556
## 2 Yes       2740
```

About 40% do not have a claim while 60% have at least one claim.

```
set.seed(42)
index <- createDataPartition(y = auto_claim$CLM_FLAG,
                             p = 0.8, list = F) %>% as.numeric()
auto_claim <- auto_claim %>%
  mutate(target = as.factor(ifelse(CLM_FLAG == "Yes", 1,0)))
train <-  auto_claim %>% slice(index)
test <- auto_claim %>% slice(-index)

frequency <- glm(target ~ AGE + GENDER + MARRIED + CAR_USE +
                   BLUEBOOK + CAR_TYPE + AREA,
                 data=train,
                 family = binomial(link="logit"))
```

All of the variables except for the `CAR_TYPE` and `GENDERM` are highly significant. The car types `SPORTS CAR` and `SUV` appear to be significant, and so if we wanted to make the model simpler we could create indicator variables for `CAR_TYPE == SPORTS CAR` and `CAR_TYPE == SUV`.

```
frequency %>% summary()
```

```
##
## Call:
## glm(formula = target ~ AGE + GENDER + MARRIED + CAR_USE + BLUEBOOK +
##     CAR_TYPE + AREA, family = binomial(link = "logit"), data = train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.8431  -0.8077  -0.5331   0.9575   3.0441
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -3.523e-01  2.517e-01  -1.400  0.16160
## AGE                -2.289e-02  3.223e-03  -7.102 1.23e-12 ***
```

```
## GENDERM            -1.124e-02  9.304e-02  -0.121  0.90383
## MARRIEDYes         -6.028e-01  5.445e-02 -11.071  < 2e-16 ***
## CAR_USEPrivate     -1.008e+00  6.569e-02 -15.350  < 2e-16 ***
## BLUEBOOK           -4.025e-05  4.699e-06  -8.564  < 2e-16 ***
## CAR_TYPEPickup     -6.687e-02  1.390e-01  -0.481  0.63048
## CAR_TYPESedan      -3.689e-01  1.383e-01  -2.667  0.00765 **
## CAR_TYPESports Car  6.159e-01  1.891e-01   3.256  0.00113 **
## CAR_TYPESUV         2.982e-01  1.772e-01   1.683  0.09240 .
## CAR_TYPEVan        -8.983e-03  1.319e-01  -0.068  0.94569
## AREAUrban           2.128e+00  1.064e-01  19.993  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 9544.3  on 8236  degrees of freedom
## Residual deviance: 8309.6  on 8225  degrees of freedom
## AIC: 8333.6
##
## Number of Fisher Scoring iterations: 5
```

The signs of the coefficients tell if the probability of having a claim is either increasing or decreasing by each variable. For example, the likelihood of an accident

- Decreases as the age of the car increases
- Is lower for men
- Is higher for sports cars and SUVs

The p-values tell us if the variable is significant.

- `Age`, `MarriedYes`, `CAR_USEPrivate`, `BLUEBOOK`, and `AreaUrban` are significant.
- Certain values of `CAR_TYPE` are significant but others are not.

The output is a predicted probability. We can see that this is centered around a probability of about 0.5.

```
preds <- predict(frequency, newdat=test,type="response")
qplot(preds)
```

In order to convert these values to predicted 0's and 1's, we assign a *cutoff* value so that if $\hat{y}$ is above this threshold we use a 1 and 0 othersise. The default cutoff is 0.5. We change this to 0.3 and see that there are 763 policies predicted to have claims.

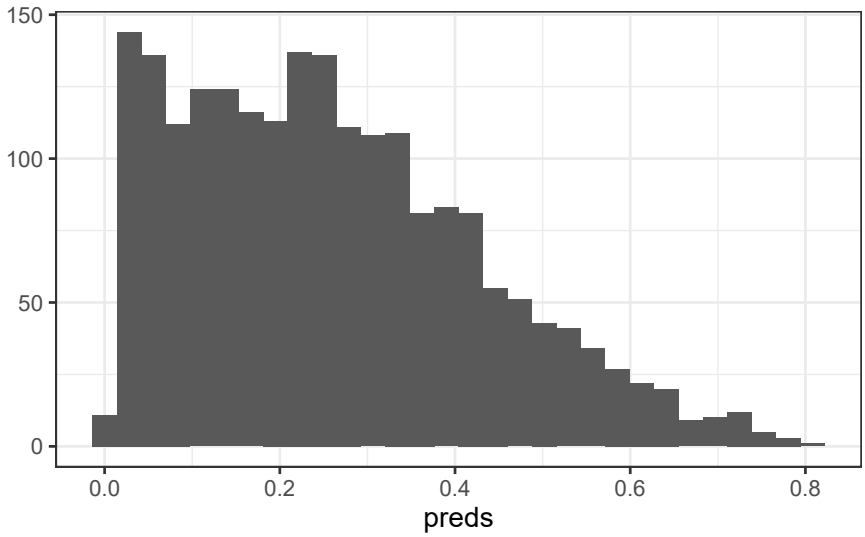


Figure 11.2: Distribution of Predicted Probability

```
test <- test %>% mutate(pred_zero_one = as.factor(1*(preds>.3)))
summary(test$pred_zero_one)
```

```
##    0    1
## 1296  763
```

How do we decide on this cutoff value? We need to compare cutoff values based on some evaluation metric. For example, we can use *accuracy*.

$$\text{Accuracy} = \frac{\text{Correct Guesses}}{\text{Total Guesses}}$$

This results in an accuracy of 70%. But is this good?

```
test %>% summarise(accuracy = mean(pred_zero_one == target))
```

```
## # A tibble: 1 x 1
##   accuracy
##      <dbl>
## 1    0.699
```

Consider what would happen if we just predicted all 0's. The accuracy is 74%.

```
test %>% summarise(accuracy = mean(0 == target))
```

```
## # A tibble: 1 x 1
##   accuracy
##      <dbl>
## 1    0.734
```

For policies which experience claims the accuracy is 63%.

```
test %>%
  filter(target == 1) %>%
  summarise(accuracy = mean(pred_zero_one == target))
```

```
## # A tibble: 1 x 1
##   accuracy
##      <dbl>
## 1    0.631
```

But for policies that don't actually experience claims this is 72%.

```
test %>%
  filter(target == 0) %>%
  summarise(accuracy = mean(pred_zero_one == target))
```

```
## # A tibble: 1 x 1
##   accuracy
##      <dbl>
## 1    0.724
```

How do we know if this is a good model? We can repeat this process with a different cutoff value and get different accuracy metrics for these groups. Let's use a cutoff of 0.6.

75%

```
test <- test %>% mutate(pred_zero_one = as.factor(1*(preds>.6)))
test %>% summarise(accuracy = mean(pred_zero_one == target))
```

```
## # A tibble: 1 x 1
##   accuracy
##      <dbl>
## 1    0.752
```

10% for policies with claims and 98% for policies without claims.

```
test %>%
  filter(target == 1) %>%
  summarise(accuracy = mean(pred_zero_one == target))
```

```
## # A tibble: 1 x 1
##   accuracy
##      <dbl>
## 1    0.108
```

```
test %>%
  filter(target == 0) %>%
  summarise(accuracy = mean(pred_zero_one == target))
```

```
## # A tibble: 1 x 1
##   accuracy
##      <dbl>
## 1    0.985
```

The punchline is that the accuracy depends on the cutoff value, and changing the cutoff value changes whether the model is accuracy for the "true = 1" classes (policies with actual claims) vs. the "false = 0" classes (policies without claims).

11.3 Classification metrics

For regression problems, when the output is a whole number, we can use the sum of squares RSS, the r-squared R^2, the mean absolute error MAE, and the likelihood. For classification problems where the output is in $\{0, 1\}$, we need to a new set of metrics.

A *confusion matrix* shows is a table that summarises how the model classifies each group.

- No claims and predicted to not have claims - **True Negatives (TN) = 1,489**
- Had claims and predicted to have claims - **True Positives (TP) = 59**
- No claims but predited to have claims - **False Positives (FP) = 22**
- Had claims but predicted not to - **False Negatives (FN) = 489**

```
confusionMatrix(test$pred_zero_one,factor(test$target))$table
```

```
##            Reference
## Prediction    0    1
##          0 1489  489
##          1   22   59
```

These definitions allow us to measure performance on the different groups.

Precision answers the question "out of all of the positive predictions, what percentage were correct?"

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall answers the question "out of all of positive examples in the data set, what percentage were correct?"

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The choice of using precision vs. recall depends on the relative cost of making a FP or a FN error. If FP errors are expensive, then use precision; if FN errors are expensive, then use recall.

Example A: the model trying to detect a deadly disease, which only 1 out of every 1000 patient's survive without early detection. Then the goal should be to optimize *recall*, because we would want every patient that has the disease to get detected.

Example B: the model is detecting which emails are spam or not. If an important email is flagged as spam incorrectly, the cost is 5 hours of lost productivity. In this case, *precision* is the main concern.

In some cases we can compare this "cost" in actual values. For example, if a federal court is predicting if a criminal will recommit or not, they can agree that "1 out of every 20 guilty individuals going free" in exchange for "90% of those who are guilty being convicted". When money is involed, this a dollar amount can be used: flagging non-spam as spam may cost $100 whereas missing a spam email may cost $2. Then the cost-weighted accuracy is

$$\text{Cost} = (100)(\text{FN}) + (2)(\text{FP})$$

Then the cutoff value can be tuned in order to find the minimum cost.

Fortunately, all of this is handled in a single function called `confusionMatrix`.

```
confusionMatrix(test$pred_zero_one,factor(test$target))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1489  489
##          1   22   59
##
##                Accuracy : 0.7518
##                  95% CI : (0.7326, 0.7704)
##     No Information Rate : 0.7339
##     P-Value [Acc > NIR] : 0.03366
##
##                   Kappa : 0.1278
##
##  Mcnemar's Test P-Value : < 2e-16
##
##             Sensitivity : 0.9854
##             Specificity : 0.1077
##          Pos Pred Value : 0.7528
##          Neg Pred Value : 0.7284
##              Prevalence : 0.7339
##          Detection Rate : 0.7232
##    Detection Prevalence : 0.9607
##       Balanced Accuracy : 0.5466
##
##        'Positive' Class : 0
##
```

11.3.1 Area Under the ROC Curv (AUC)

What if we look at both the true-positive rate (TPR) and false positive rate (FPR) simultaneously? That is, for each value of the cutoff, we can calculate the TPR and TNR.

For example, say that we have 10 cutoff values, $\{k_1, k_2, ..., k_{10}\}$. Then for each value of k we calculate both the true positive rates

$$\text{TPR} = \{\text{TPR}(k_1), \text{TPR}(k_2), .., \text{TPR}(k_{10})\}$$

and the true negative rates

$$\{\text{FNR} = \{\text{FNR}(k_1), \text{FNR}(k_2), .., \text{FNR}(k_{10})\}$$

Then we set `x = TPR` and `y = FNR` and graph x against y. The plot below shows

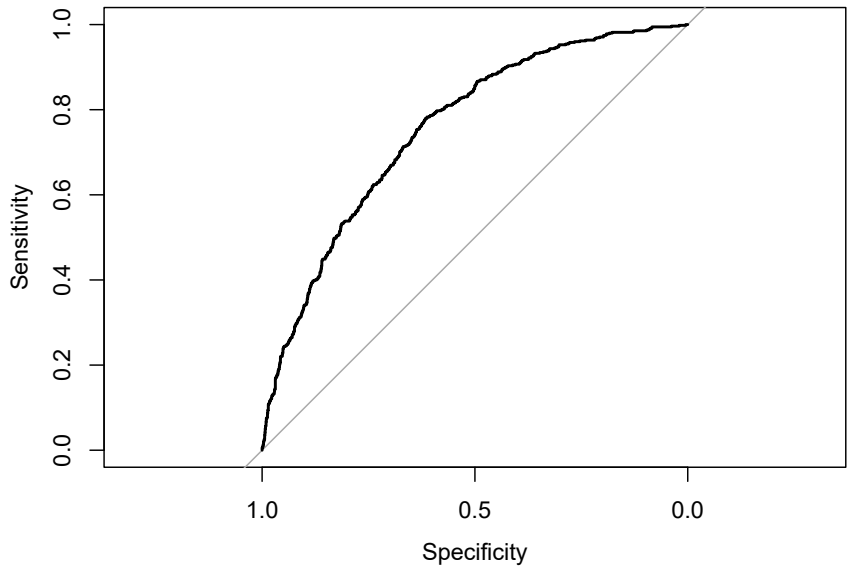


Figure 11.3: AUC for auto_claim

the ROC for the `auto_claims` data. The Area Under the Curv of 0.6795 is what we would get if we integrated under the curve.

```
library(pROC)
roc(test$target, preds, plot = T)
```

```
##
## Call:
## roc.default(response = test$target, predictor = preds, plot = T)
##
## Data: preds in 1511 controls (test$target 0) < 548 cases (test$target 1).
## Area under the curve: 0.7558
```

If we just randomly guess, the AUC would be 0.5, which is represented by the 45-degree line. A perfect model would maximize the curve to the upper-left corner.

AUC is preferred over Accuracy when there are a lot more "true" classes than "false" classes, which is known as having **class imbalance*. An example is bank fraud detection: 99.99% of bank transactions are "false" or "0" classes,

and so optimizing for accuracy alone will result in a low sensitivity for detecting actual fraud.

11.3.2 Additional reading

Title	Source
An Overview of Classification	ISL 4.1
Understanding AUC - ROC Curv	Sarang Narkhede, Towards Data Science
Precision vs. Recall	Shruti Saxena, Towards Data Science

Chapter 12

Penalized Linear Models

One of the main weaknesses of the GLM, including all linear models in this chapter, is that the features need to be selected by hand. Stepwise selection helps to improve this process, but fails when the inputs are correlated and often has a strong dependence on seemingly arbitrary choices of evaluation metrics such as using AIC or BIC and forward or backwise directions.

The Bias Variance Tradoff is about finding the lowest error by changing the flexibility of the model. Penalization methods use a parameter to control for this flexibility directly.

Earlier on we said that the linear model minimizes the sum of square terms, known as the residual sum of squares (RSS)

$$\text{RSS} = \sum_i (y_i - \hat{y})^2 = \sum_i (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2$$

This loss function can be modified so that models which include more (and larger) coefficients are considered as worse. In other words, when there are more β's, or β's which are larger, the RSS is higher.

12.1 Ridge Regression

Ridge regression adds a penalty term which is proportional to the square of the sum of the coefficients. This is known as the "L2" norm.

$$\sum_i (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

This λ controls how much of a penalty is imposed on the size of the coefficients. When λ is high, simpler models are treated more favorably because the $\sum_{j=1}^{p} \beta_j^2$ carries more weight. Conversely, then λ is low, complex models are more favored. When $\lambda = 0$, we have an ordinary GLM.

12.2 Lasso

The official name is the Least Absolute Shrinkage and Selection Operator, but the common name is just "the lasso". Just as with Ridge regression, we want to favor simpler models; however, we also want to *select* variables. This is the same as forcing some coefficients to be equal to 0.

Instead of taking the square of the coefficients (L2 norm), we take the absolute value (L1 norm).

$$\sum_i (y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

In ISLR, Hastie et al show that this results in coefficients being forced to be exactly 0. This is extremely useful because it means that by changing λ, we can select how many variables to use in the model.

Note: While any response family is possible with penalized regression, in R, only the Gaussian family is possible in the library `glmnet`, and so this is the only type of question that the SOA can ask.

12.3 Elastic Net

The Elastic Net uses a penalty term which is between the L1 and L2 norms. The penalty term is a weighted average using the mixing parameter $0 \leq \alpha \leq 1$. The loss fucntion is then

$$\text{RSS} + (1 - \alpha)/2 \sum_{j=1}^{p} \beta_j^2 + \alpha \sum_{j=1}^{p} |\beta_j|$$

When $\alpha = 1$ is turns into a Lasso; when $\alpha = 1$ this is the Ridge model.

Luckily, none of this needs to be memorized. On the exam, read the documentation in R to refresh your memory. For the Elastic Net, the function is `glmnet`, and so running `?glmnet` will give you this info.

> **Shortcut**: When using complicated functions on the exam, use `?function_name` to get the documentation.

12.4 Advantages and disadvantages

Elastic Net/Lasso/Ridge Advantages

- All benefits from GLMS
- Automatic variable selection for Lasso; smaller coefficients for Ridge
- Better predictive power than GLM

Elastic Net/Lasso/Ridge Disadvantages

- All cons of GLMs

Readings
ISLR 6.1 Subset Selection
ISLR 6.2 Shrinkage Methods

12.5 Example: Ridge Regression

```
library(ISLR)
library(glmnet)
library(dplyr)
library(tidyr)
```

We will use the `glmnet` package in order to perform ridge regression and the lasso. The main function in this package is `glmnet()`, which can be used to fit ridge regression models, lasso models, and more. This function has slightly different syntax from other model-fitting functions that we have encountered thus far in this book. In particular, we must pass in an x matrix as well as a y vector, and we do not use the $y \sim x$ syntax.

Before proceeding, let's first ensure that the missing values have been removed from the data, as described in the previous lab.

```
Hitters = na.omit(Hitters)
```

We will now perform ridge regression and the lasso in order to predict `Salary` on the `Hitters` data. Let's set up our data:

```
x = model.matrix(Salary~., Hitters)[,-1] # trim off the first column
                                         # leaving only the predictors
y = Hitters %>%
  select(Salary) %>%
  unlist() %>%
  as.numeric()
```

The `model.matrix()` function is particularly useful for creating x; not only does it produce a matrix corresponding to the 19 predictors but it also automatically transforms any qualitative variables into dummy variables. The latter property is important because `glmnet()` can only take numerical, quantitative inputs.

The `glmnet()` function has an alpha argument that determines what type of model is fit. If `alpha = 0` then a ridge regression model is fit, and if `alpha = 1` then a lasso model is fit. We first fit a ridge regression model:

```
grid = 10^seq(10, -2, length = 100)
ridge_mod = glmnet(x, y, alpha = 0, lambda = grid)
```

By default the `glmnet()` function performs ridge regression for an automatically selected range of λ values. However, here we have chosen to implement the function over a grid of values ranging from $\lambda = 10^10$ to $\lambda = 10^{-2}$, essentially covering the full range of scenarios from the null model containing only the intercept, to the least squares fit.

As we will see, we can also compute model fits for a particular value of λ that is not one of the original grid values. Note that by default, the `glmnet()` function standardizes the variables so that they are on the same scale. To turn off this default setting, use the argument `standardize = FALSE`.

Associated with each value of λ is a vector of ridge regression coefficients, stored in a matrix that can be accessed by `coef()`. In this case, it is a 20×100 matrix, with 20 rows (one for each predictor, plus an intercept) and 100 columns (one for each value of λ).

```
dim(coef(ridge_mod))
```

```
## [1]  20 100
```

We expect the coefficient estimates to be much smaller, in terms of l_2 norm, when a large value of λ is used, as compared to when a small value of λ is used. These are the coefficients when $\lambda = 11498$, along with their l_2 norm:

```
ridge_mod$lambda[50] #Display 50th lambda value
```

```
## [1] 11497.57
```

```
coef(ridge_mod)[,50] # Display coefficients associated with 50th lambda value
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
## 407.356050200   0.036957182   0.138180344   0.524629976   0.230701523
##           RBI         Walks         Years        CAtBat         CHits
##   0.239841459   0.289618741   1.107702929   0.003131815   0.011653637
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##   0.087545670   0.023379882   0.024138320   0.025015421   0.085028114
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
##  -6.215440973   0.016482577   0.002612988  -0.020502690   0.301433531
```

```
sqrt(sum(coef(ridge_mod)[-1,50]^2)) # Calculate l2 norm
```

```
## [1] 6.360612
```

In contrast, here are the coefficients when $\lambda = 705$, along with their l_2 norm. Note the much larger l_2 norm of the coefficients associated with this smaller value of λ.

```
ridge_mod$lambda[60] #Display 60th lambda value
```

```
## [1] 705.4802
```

```
coef(ridge_mod)[,60] # Display coefficients associated with 60th lambda value
```

```
##  (Intercept)         AtBat          Hits         HmRun          Runs
##  54.32519950   0.11211115   0.65622409   1.17980910   0.93769713
##          RBI         Walks         Years        CAtBat         CHits
##   0.84718546   1.31987948   2.59640425   0.01083413   0.04674557
##       CHmRun         CRuns          CRBI        CWalks       LeagueN
##   0.33777318   0.09355528   0.09780402   0.07189612  13.68370191
##    DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -54.65877750   0.11852289   0.01606037  -0.70358655   8.61181213
```

```
sqrt(sum(coef(ridge_mod)[-1,60]^2)) # Calculate l2 norm
```

```
## [1] 57.11001
```

We can use the `predict()` function for a number of purposes. For instance, we can obtain the ridge regression coefficients for a new value of λ, say 50:

```
predict(ridge_mod, s=50, type="coefficients")[1:20,]
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##  4.876610e+01 -3.580999e-01  1.969359e+00 -1.278248e+00  1.145892e+00
##           RBI         Walks         Years        CAtBat         CHits
##  8.038292e-01  2.716186e+00 -6.218319e+00  5.447837e-03  1.064895e-01
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##  6.244860e-01  2.214985e-01  2.186914e-01 -1.500245e-01  4.592589e+01
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -1.182011e+02  2.502322e-01  1.215665e-01 -3.278600e+00 -9.496680e+00
```

We now split the samples into a training set and a test set in order to estimate the test error of ridge regression and the lasso.

```
set.seed(1)

train = Hitters %>%
  sample_frac(0.5)

test = Hitters %>%
  setdiff(train)

x_train = model.matrix(Salary~., train)[,-1]
x_test = model.matrix(Salary~., test)[,-1]

y_train = train %>%
  select(Salary) %>%
  unlist() %>%
  as.numeric()

y_test = test %>%
  select(Salary) %>%
  unlist() %>%
  as.numeric()
```

Next we fit a ridge regression model on the training set, and evaluate its MSE on the test set, using $\lambda = 4$. Note the use of the `predict()` function again: this time we get predictions for a test set, by replacing `type="coefficients"` with the `newx` argument.

```
ridge_mod = glmnet(x_train, y_train, alpha=0, lambda = grid, thresh = 1e-12)
ridge_pred = predict(ridge_mod, s = 4, newx = x_test)
mean((ridge_pred - y_test)^2)
```

```
## [1] 139858.6
```

The test MSE is 101242.7. Note that if we had instead simply fit a model with just an intercept, we would have predicted each test observation using the mean of the training observations. In that case, we could compute the test set MSE like this:

```
mean((mean(y_train) - y_test)^2)
```

```
## [1] 224692.1
```

We could also get the same result by fitting a ridge regression model with a very large value of λ. Note that `1e10` means 10^{10}.

```
ridge_pred = predict(ridge_mod, s = 1e10, newx = x_test)
mean((ridge_pred - y_test)^2)
```

```
## [1] 224692.1
```

So fitting a ridge regression model with $\lambda = 4$ leads to a much lower test MSE than fitting a model with just an intercept. We now check whether there is any benefit to performing ridge regression with $\lambda = 4$ instead of just performing least squares regression. Recall that least squares is simply ridge regression with $\lambda = 0$.

* Note: In order for `glmnet()` to yield the **exact** least squares coefficients when $\lambda = 0$, we use the argument `exact=T` when calling the `predict()` function. Otherwise, the `predict()` function will interpolate over the grid of λ values used in fitting the `glmnet()` model, yielding approximate results. Even when we use `exact = T`, there remains a slight discrepancy in the third decimal place between the output of `glmnet()` when $\lambda = 0$ and the output of `lm()`; this is due to numerical approximation on the part of `glmnet()`.

```
ridge_pred = predict(ridge_mod, s = 0, x = x_train, y = y_train, newx = x_test, exact =
mean((ridge_pred - y_test)^2)
```

```
## [1] 175051.7
```

```
lm(Salary~., data = train)
```

```
##
## Call:
## lm(formula = Salary ~ ., data = train)
##
## Coefficients:
## (Intercept)        AtBat         Hits        HmRun         Runs
##   2.398e+02   -1.639e-03   -2.179e+00    6.337e+00    7.139e-01
##         RBI        Walks        Years       CAtBat        CHits
##   8.735e-01    3.594e+00   -1.309e+01   -7.136e-01    3.316e+00
##      CHmRun        CRuns         CRBI       CWalks      LeagueN
##   3.407e+00   -5.671e-01   -7.525e-01    2.347e-01    1.322e+02
##   DivisionW      PutOuts      Assists       Errors   NewLeagueN
##  -1.346e+02    2.099e-01    6.229e-01   -4.616e+00   -8.330e+01
```

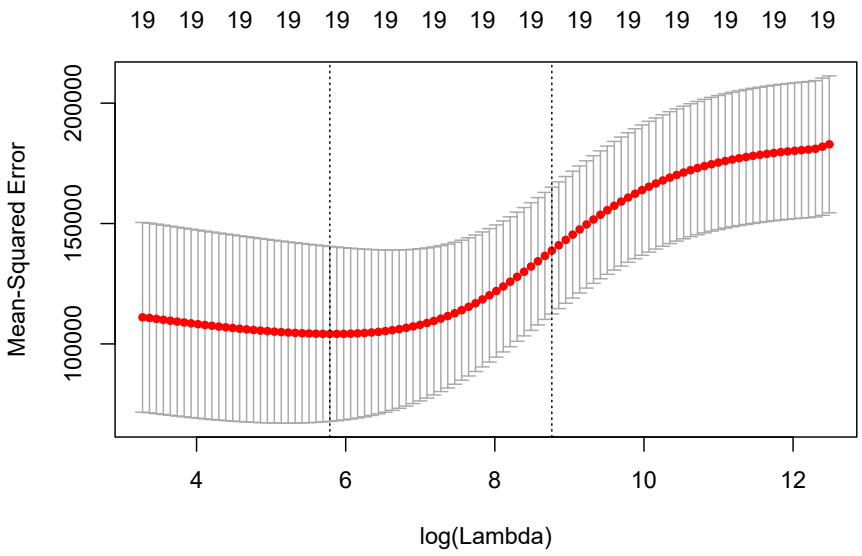
```
predict(ridge_mod, s = 0, x = x_train, y = y_train, exact = T, type="coefficients")[1:20,]
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##  239.83274953   -0.00175359   -2.17853087    6.33694957    0.71369687
##           RBI         Walks         Years        CAtBat         CHits
##    0.87329878    3.59421378  -13.09231408   -0.71351092    3.31523605
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##    3.40701392   -0.56709530   -0.75240961    0.23467433  132.15949536
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -134.58503816    0.20992473    0.62288126   -4.61583857  -83.29432536
```

It looks like we are indeed improving over regular least-squares! Side note: in general, if we want to fit a (unpenalized) least squares model, then we should use the `lm()` function, since that function provides more useful outputs, such as standard errors and p-values for the coefficients.

Instead of arbitrarily choosing $\lambda = 4$, it would be better to use cross-validation to choose the tuning parameter λ. We can do this using the built-in cross-validation function, `cv.glmnet()`. By default, the function performs 10-fold cross-validation, though this can be changed using the argument `folds`. Note that we set a random seed first so our results will be reproducible, since the choice of the cross-validation folds is random.

```
set.seed(1)
cv.out = cv.glmnet(x_train, y_train, alpha = 0) # Fit ridge regression model on training data
plot(cv.out) # Draw plot of training MSE as a function of lambda
```




```
bestlam = cv.out$lambda.min  # Select lamda that minimizes training MSE
bestlam
```

```
## [1] 326.1406
```

Therefore, we see that the value of λ that results in the smallest cross-validation error is 339.1845 What is the test MSE associated with this value of λ?

```
ridge_pred = predict(ridge_mod, s = bestlam, newx = x_test) # Use best lambda to predi
mean((ridge_pred - y_test)^2) # Calculate test MSE
```

```
## [1] 140056.2
```

This represents a further improvement over the test MSE that we got using $\lambda = 4$. Finally, we refit our ridge regression model on the full data set, using the value of λ chosen by cross-validation, and examine the coefficient estimates.

```
out = glmnet(x, y, alpha = 0) # Fit ridge regression model on full dataset
predict(out, type = "coefficients", s = bestlam)[1:20,] # Display coefficients using l
```

```
##  (Intercept)        AtBat         Hits        HmRun         Runs
##  15.44835008   0.07716945   0.85906253   0.60120339   1.06366687
```

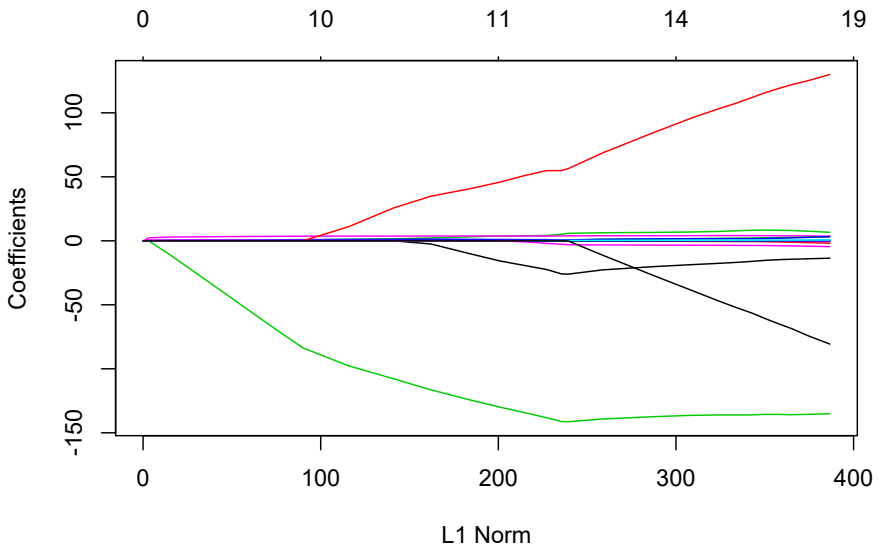
```
##          RBI        Walks        Years       CAtBat        CHits
##   0.87936073   1.62437580   1.35296287   0.01134998   0.05746377
##       CHmRun        CRuns         CRBI       CWalks      LeagueN
##   0.40678422   0.11455696   0.12115916   0.05299953  22.08942749
##    DivisionW      PutOuts      Assists       Errors   NewLeagueN
## -79.03490973   0.16618830   0.02941513  -1.36075644   9.12528398
```

As expected, none of the coefficients are exactly zero - ridge regression does not perform variable selection!

12.6 Example: The Lasso

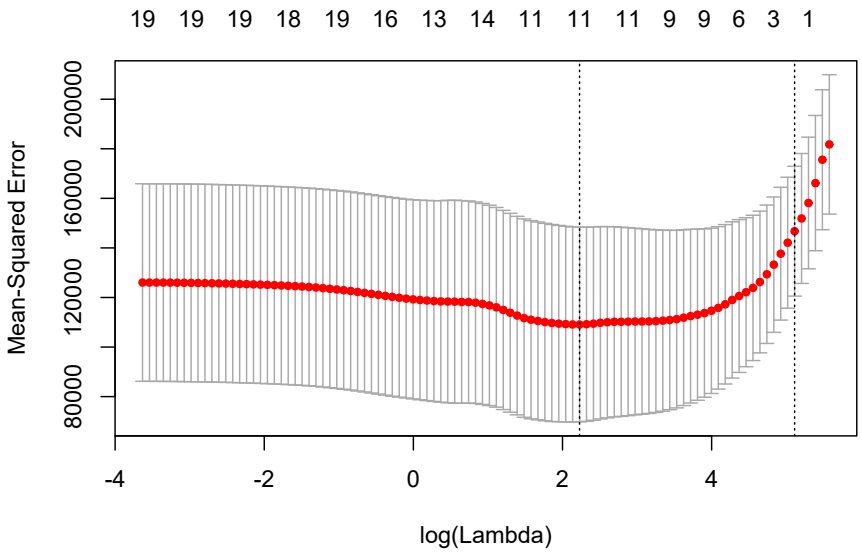
We saw that ridge regression with a wise choice of λ can outperform least squares as well as the null model on the Hitters data set. We now ask whether the lasso can yield either a more accurate or a more interpretable model than ridge regression. In order to fit a lasso model, we once again use the `glmnet()` function; however, this time we use the argument `alpha=1`. Other than that change, we proceed just as we did in fitting a ridge model:

```
lasso_mod = glmnet(x_train, y_train, alpha = 1, lambda = grid) # Fit lasso model on training data
plot(lasso_mod)                                                # Draw plot of coefficients
```



Notice that in the coefficient plot that depending on the choice of tuning parameter, some of the coefficients are exactly equal to zero. We now perform cross-validation and compute the associated test error:

```
set.seed(1)
cv.out = cv.glmnet(x_train, y_train, alpha = 1) # Fit lasso model on training data
plot(cv.out) # Draw plot of training MSE as a function of lambda
```




```
bestlam = cv.out$lambda.min # Select lamda that minimizes training MSE
lasso_pred = predict(lasso_mod, s = bestlam, newx = x_test) # Use best lambda to predi
mean((lasso_pred - y_test)^2) # Calculate test MSE
```

```
## [1] 143273
```

This is substantially lower than the test set MSE of the null model and of least squares, and very similar to the test MSE of ridge regression with λ chosen by cross-validation.

However, the lasso has a substantial advantage over ridge regression in that the resulting coefficient estimates are sparse. Here we see that 12 of the 19 coefficient estimates are exactly zero:

```
out = glmnet(x, y, alpha = 1, lambda = grid) # Fit lasso model on full dataset
lasso_coef = predict(out, type = "coefficients", s = bestlam)[1:20,] # Display coefficients using
lasso_coef
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##    1.27429897   -0.05490834    2.18012455    0.00000000    0.00000000
##           RBI         Walks         Years        CAtBat         CHits
##    0.00000000    2.29189433   -0.33767315    0.00000000    0.00000000
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##    0.02822467    0.21627609    0.41713051    0.00000000   20.28190194
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -116.16524424    0.23751978    0.00000000   -0.85604181    0.00000000
```

Selecting only the predictors with non-zero coefficients, we see that the lasso model with λ chosen by cross-validation contains only seven variables:

```
lasso_coef[lasso_coef!=0] # Display only non-zero coefficients
```

```
##   (Intercept)         AtBat          Hits         Walks         Years
##    1.27429897   -0.05490834    2.18012455    2.29189433   -0.33767315
##        CHmRun         CRuns          CRBI       LeagueN     DivisionW
##    0.02822467    0.21627609    0.41713051   20.28190194 -116.16524424
##       PutOuts        Errors
##    0.23751978   -0.85604181
```

Practice questions:

- How do ridge regression and the lasso improve on simple least squares?
- In what cases would you expect ridge regression outperform the lasso, and vice versa?

12.7 References

These examples of the Ridge and Lasso are an adaptation of p. 251-255 of "Introduction to Statistical Learning with Applications in R" by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. Adapted by R. Jordan Crouser at Smith College for SDS293: Machine Learning (Spring 2016), and re-implemented in Fall 2016 in `tidyverse` format by Amelia McNamara and R. Jordan Crouser at Smith College.

Used with permission from Jordan Crouser at Smith College. Additional Thanks to the following contributors on github:

- github.com/jcrouser
- github.com/AmeliaMN
- github.com/mhusseinmidd
- github.com/rudeboybert
- github.com/ijlyttle

Chapter 13

Tree-based models

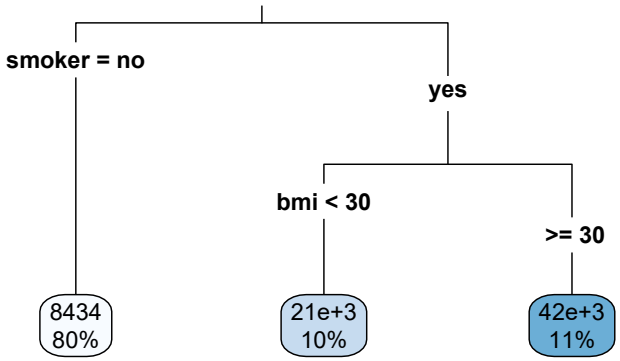
13.1 Decision Trees

13.1.1 Model form

Decision trees can be used for either classification or regression problems. The model structure is a series of yes/no questions. Depending on how each observation answers these questions, a prediction is made.

The below example shows how a single tree can predict health claims.

- For non-smokers, the predicted annual claims are 8,434. This represents 80% of the observations
- For smokers with a `bmi` of less than 30, the predicted annual claims are 21,000. 10% of patients fall into this bucket.
- For smokers with a `bmi` of more than 30, the prediction is 42,000. This bucket accounts for 11% of patients.



We can cut the data set up into these groups and look at the claim costs. From this grouping, we can see that `smoker` is the most important variable as the difference in average claims is about 20,000.

smoker	bmi_30	mean_claims	percent
no	bmi < 30	$7,977.03	0.38
no	bmi >= 30	$8,842.69	0.42
yes	bmi < 30	$21,363.22	0.10
yes	bmi >= 30	$41,557.99	0.11

This was a very simple example because there were only two variables. If we have more variables, the tree will get large very quickly. This will result in overfitting; there will be good performance on the training data but poor performance on the test data.

The step-by-step process of building a tree is

Step 1: Choose a variable at random.

This could be any variable in `age`, `children`, `charges`, `sex`, `smoker`, `age_bucket`, `bmi`, or `region`.

Step 2: Find the split point which best seperates observations out based on the value of y. A good split is one where the y's are very different. *

In this case, `smoker` was chosen. Then we can only split this in one way: `smoker = 1` or `smoker = 0`.

Then for each of these groups, smokers and non-smokers, choose another variable at random. In this case, for no-smokers, `age` was chosen. To find the best cut point of `age`, look at all possible age cut points from 18, 19, 20, 21, ..., 64 and choose the one which best separates the data.

There are three ways of deciding where to split

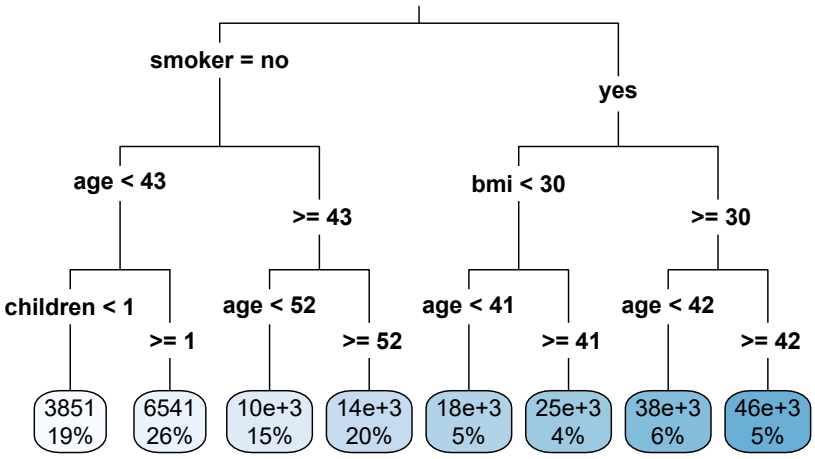
- *Entropy* (aka, information gain)
- *Gini*
- *Classification error*

Of these, only the first two are commonly used. The exam is not going to ask you to calculate either of these. Just know that neither method will work better on all data sets, and so the best practice is to test both and compare the performance.

Step 3: Continue doing this until a stopping criteria is reached. For example, the minimum number of observations is 5 or less.

As you can see, this results in a very deep tree.

```
tree <- rpart(formula = charges ~ ., data = health_insurance,
              control = rpart.control(cp = 0.003))
rpart.plot(tree, type = 3)
```



Step 4: Apply cost comlexity pruning to simplify the tree

Intuitively, we know that the above model would perform poorly due to overfitting. We want to make it simpler by removing nodes. This is very similar to how in linear models we reduce complexity by reducing the number of coefficients.

A measure of the depth of the tree is the *complexity*. A simple way of measuring this from the number of terminal nodes, called $|T|$. This is similar to the "degrees of freedom" in a linear model. In the above example, $|T| = 8$. The amount of penalization is controlled by α. This is very similar to λ in the Lasso.

Intuitively, merely only looking at the number of nodes by itself is too simple because not all data sets will have the same characteristics such as n, p, the number of categorical variables, correlations between variables, and so fourth. In addition, if we just looked at the error (squared error in this case) we would overfit very easily. To address this issue, we use a cost function which takes into account the error as well as $|T|$.

To calculate the cost of a tree, number the terminal nodes from 1 to $|T|$, and let the set of observations that fall into the mth bucket be R_m. Then add up the squared error over all terminal nodes to the penalty term.

$$\text{Cost}_\alpha(T) = \sum_{m=1}^{|T|} \sum_{R_m} (y_i - \hat{y}_{Rm})^2 + \alpha|T|$$

Step 5: Use cross-validation to select the best alpha

The cost is controlled by the `CP` parameter. In the above example, did you notice the line `rpart.control(cp = 0.003)`? This is telling `rpart` to continue growing the tree until the CP reaches 0.003. At each subtree, we can measure the cost `CP` as well as the cross-validation error `xerror`.

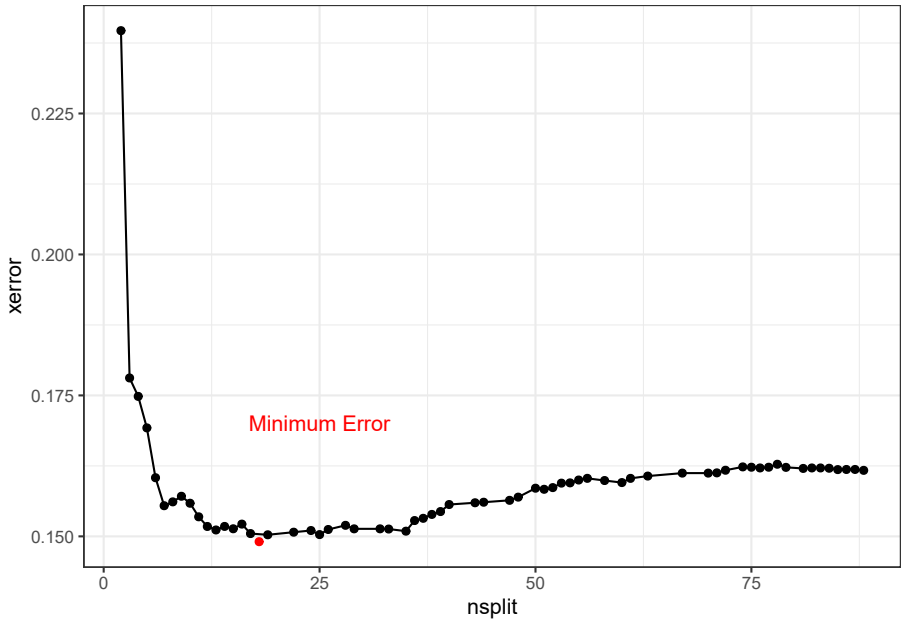
This is stored in the `cptable`

```
tree <- rpart(formula = charges ~ ., data = health_insurance,
              control = rpart.control(cp = 0.0001))
cost <- tree$cptable %>%
  as_tibble() %>%
  select(nsplit, CP, xerror)

cost %>% head()
```

```
## # A tibble: 6 x 3
##   nsplit      CP xerror
##    <dbl>   <dbl>  <dbl>
## 1      0 0.620    1.00
## 2      1 0.144    0.383
```

```
## 3       2 0.0637   0.240
## 4       3 0.00967  0.178
## 5       4 0.00784  0.175
## 6       5 0.00712  0.169
```

As more splits are added, the cost continues to decrease, reaches a minimum, and then begins to increase.



To optimize performance, choose the number of splits which has the lowest error. Often, though, the goal of using a decision tree is to create a simple model. In this case, we can err or the side of a lower `nsplit` so that the tree is shorter and more interpretable. All of the questions on so far have only used decision trees for interpretability, and a different model method has been used when predictive power is needed.

Once we have selected α, the tree is pruned. Sometimes the CP with the lowest error has a large number of splits, such as the case is here.

```
tree$cptable %>%
  as_tibble() %>%
  select(nsplit, CP, xerror) %>%
  arrange(xerror) %>%
  head()
```

```
## # A tibble: 6 x 3
```

```
##   nsplit       CP xerror
##    <dbl>    <dbl>  <dbl>
## 1     18 0.000910  0.149
## 2     19 0.000837  0.150
## 3     25 0.000681  0.150
## 4     17 0.000913  0.150
## 5     22 0.000759  0.151
## 6     35 0.000497  0.151
```

The SOA will give you code to find the lowest CP value such as below. This may or may not be useful depending on if they are asking for predictive performance or interpretability.

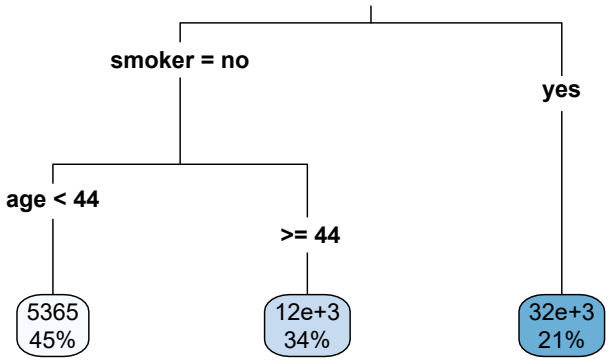
```
pruned_tree <- prune(tree,
                     cp = tree$cptable[which.min(tree$cptable[, "xerror"]), "CP"])
```

To make a simple tree, there are a few options

- Set the maximum depth of a tree with `maxdepth`
- Manually set `cp` to be higher
- Use fewer input variables and avoid categories with many levels
- Force a high number of minimum observations per terminal node with `minbucket`

For instance, using these suggestions allows for a simpler tree to be fit.

```
library(caret)
set.seed(42)
index <- createDataPartition(y = health_insurance$charges,
                             p = 0.8, list = F)
train <- health_insurance %>% slice(index)
test <- health_insurance %>% slice(-index)

simple_tree <- rpart(formula = charges ~  .,
              data = train,
              control = rpart.control(cp = 0.0001,
                                      minbucket = 200,
                                      maxdepth = 10))
rpart.plot(simple_tree, type = 3)
```



We evaluate the performance on the test set. Because the target variable `charges` is highly skewed, we use the Root Mean Squared Log Error (RMSLE). We see that the complex tree has the best (lowest) error, but also has 8 terminal nodes. The simple tree with only three terminal nodes has worse (higher) error, but this is still an improvement over the mean prediction.

```
tree_pred <- predict(tree, test)
simple_tree_pred <- predict(simple_tree, test)

get_rmsle <- function(y, y_hat){
  sqrt(mean((log(y) - log(y_hat))^2))
}

get_rmsle(test$charges, tree_pred)
```

```
## [1] 0.3920546
```

```
get_rmsle(test$charges, simple_tree_pred)
```

```
## [1] 0.5678457
```

```
get_rmsle(test$charges, mean(train$charges))
```

```
## [1] 0.9996513
```

13.1.2 Advantages and disadvantages

Advantages

- Easy to interpret
- Captures interaction effects
- Captures non-linearities
- Handles continuous and categorical data
- Handles missing values

Disadvantages

- Is a "weak learner" because of low predictive power
- Does not work on small data sets
- Is often a simplification of the underlying process because all observations at terminal nodes have equal predicted values
- Is biased towards selecting high-cardinality features because more possible split points for these features tend to lead to overfitting
- High variance (which can be alleviated with stricter parameters) leads the "easy to interpret results" to change upon retraining Unable to predict beyond the range of the training data for regression (because each predicted value is an average of training samples)

Readings
ISLR 8.1.1 Basics of Decision Trees
ISLR 8.1.2 Classification Trees
rpart Documentation (Optional)

13.2 Ensemble learning

The "wisdom of crowds" says that often many are smater than the few. In the context of modeling, the models which we have looked at so far have been single guesses; however, often the underlying process is more complex than any single model can explain. If we build separate models and then combine them, known as *ensembling*, performance can be improved. Instead of trying to create a single perfect model, many simple models, known as *weak learners* are combined into a *meta-model*.

The two main ways that models are combined are through *bagging* and *boosting*.

13.2.1 Bagging

To start, we create many "copies" of the training data by sampling with replacement. Then we fit a simple model, typically a decision tree or linear model, to each of the data sets. Because each model is looking at different areas of the data, the predictions are different. The final model is a weighted average of each of the individual models.

13.2.2 Boosting

Boosting always uses the original training data and iteratively fits models to the error of the prior models. These weak learners are ineffective by themselves but powerful when added together. Unlike with bagging, the computer must train these weak learners *sequentially* instead of in parallel.

13.3 Random Forests

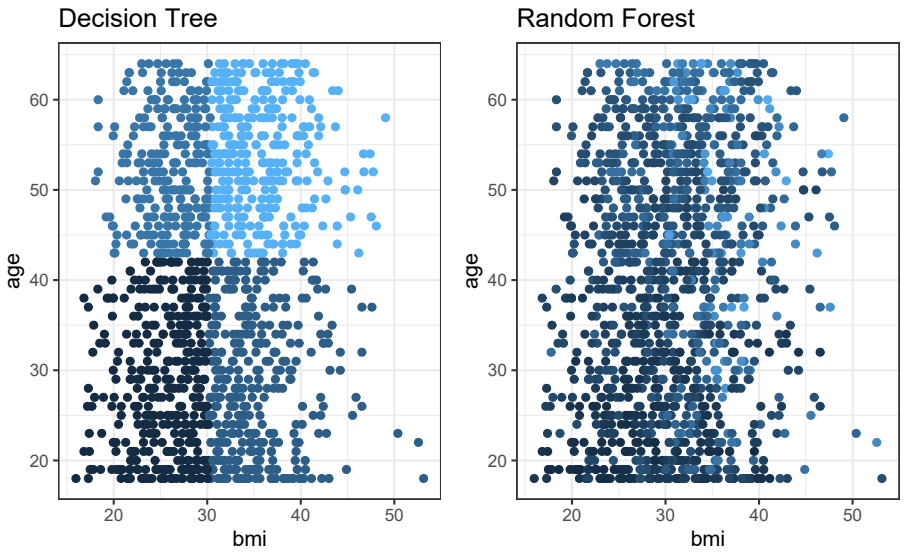
13.3.1 Model form

A random forest is the most common example of bagging. As the name implies, a forest is made up of *trees*. Seperate trees are fit to sampled datasets. For random forests, there is one minor modification: in order to make each model even more different, each tree selects a *random subset of variables*.

1. Assume that the underlying process, Y, has some signal within the data $\mathbf{X}$.
2. Introduce randomness (variance) to capture the signal.
3. Remove the variance by taking an average.

When using only a single tree, there can only be as many predictions as there are terminal nodes. In a random forest, predictions can be more granular due to the contribution of each of the trees.

The below graph illustrates this. A single tree (left) has stair-like, step-wise predictions whereas a random forest is free to predict any value. The color represents the predicted value (yellow = highest, black = lowest).



Unlike decision trees, random forest trees do not need to be pruned. This is because overfitting is less of a problem: if one tree overfits, there are other trees which overfit in other areas to compensate.

In most applications, only the `mtry` parameter, which controls how many variables to consider at each split, needs to be tuned. Tuning the `ntrees` parameter is not required; however, the soa may still ask you to.

13.3.2 Example

Using the basic `randomForest` package we fit a model with 500 trees.

This expects only numeric values. We create dummy (indicator) columns.

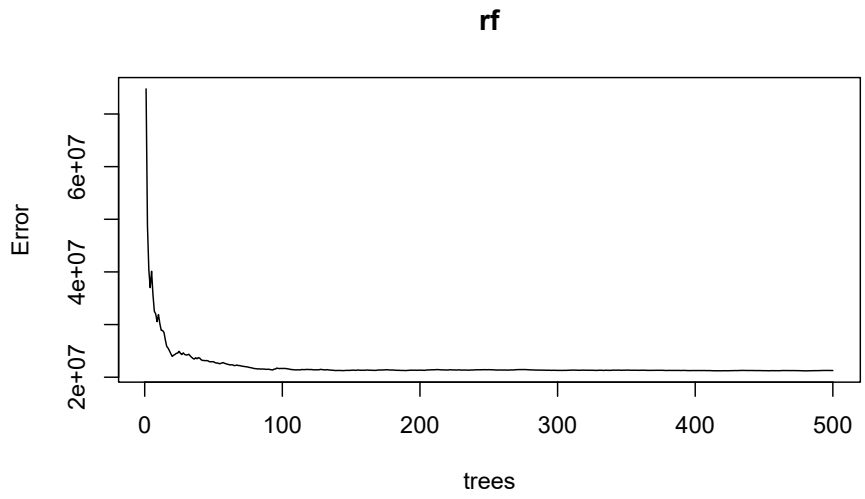
```
rf_data <- health_insurance %>%
  mutate(sex = ifelse(sex == "male", 1, 0),
         smoker = ifelse(smoker == "yes", 1, 0),
         region_ne = ifelse(region == "northeast", 1,0),
         region_nw = ifelse(region == "northwest", 1,0),
         region_se = ifelse(region == "southeast", 1,0),
         region_sw = ifelse(region == "southwest", 1,0)) %>%
  select(-region)
rf_data %>% glimpse(50)
```

```
## Observations: 1,338
## Variables: 10
## $ age       <dbl> 19, 18, 28, 33, 32, 31, 46,...
```

```
## $ sex       <dbl> 0, 1, 1, 1, 1, 0, 0, 0, 1, ...
## $ bmi       <dbl> 27.900, 33.770, 33.000, 22....
## $ children  <dbl> 0, 1, 3, 0, 0, 0, 1, 3, 2, ...
## $ smoker    <dbl> 1, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ charges   <dbl> 16884.924, 1725.552, 4449.4...
## $ region_ne <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, ...
## $ region_nw <dbl> 0, 0, 0, 1, 1, 0, 0, 1, 0, ...
## $ region_se <dbl> 0, 1, 1, 0, 0, 1, 1, 0, 0, ...
## $ region_sw <dbl> 1, 0, 0, 0, 0, 0, 0, 0, 0, ...
```

```
library(caret)
set.seed(42)
index <- createDataPartition(y = rf_data$charges,
                             p = 0.8, list = F)
train <- rf_data %>% slice(index)
test <- rf_data %>% slice(-index)
```

```
rf <- randomForest(charges ~ ., data = train, ntree = 500)
plot(rf)
```



We again use RMSLE. This is lower (better) than a model that uses the average as a baseline.

```
pred <- predict(rf, test)
get_rmsle <- function(y, y_hat){
  sqrt(mean((log(y) - log(y_hat))^2))
```

```
}

get_rmsle(test$charges, pred)
```

```
## [1] 0.4772576
```

```
get_rmsle(test$charges, mean(train$charges))
```

```
## [1] 0.9996513
```

13.3.3 Variable Importance

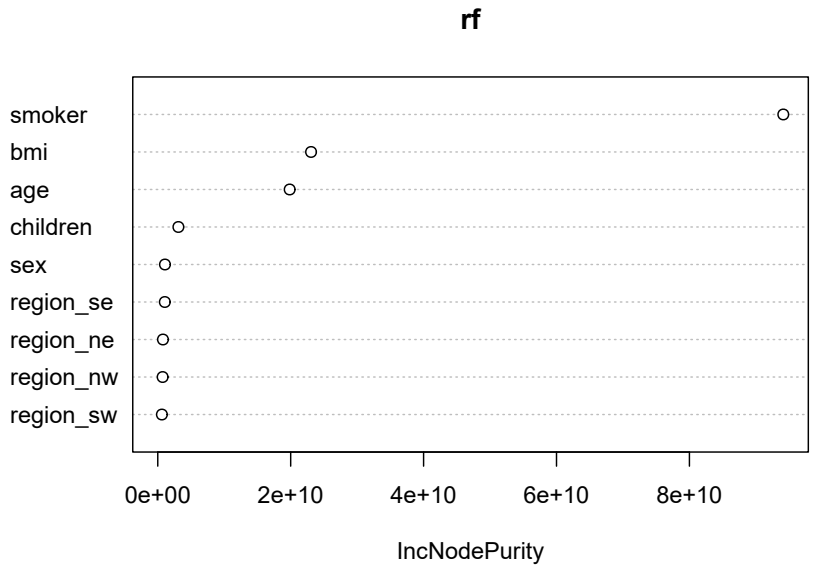
Variable importance is a way of measuring how each variable contributes to overall model. For single decision trees, if a variable was "higher" up in the tree, then this variable would have greater influence. Statistically, there are two ways of measuring this:

1) Look at the reduction in error when a the variable is randomly permuted verses using the actual values. This is done with `type = 1`.

2) Use the total decrease in node impurities from splitting on the variable, averaged over all trees. For classification, the node impurity is measured by the Gini index; for regression, it is measured by the residual sum of squares RSS. This is `type = 2`.

`smoker`, `bmi`, and `age` are the most importance predictors of charges. As you can imagine, variable importance is a highly useful tool for building models. We could use this to test out newly engineered features, or perform feature selection by taking the top-n features and use them in a different model. Random forests can handle very high dimensional data which allows for many tests to be run at once.

```
varImpPlot(x = rf)
```



13.3.4 Partial dependence

We know which variables are important, but what about the direction of the change? In a linear model we would be able to just look at the sign of the coefficient. In tree-based models, we have a tool called *partial dependence*. This attempts to measure the change in the predicted value by taking the average $\hat{\mathbf{y}}$ after removing the effects of all other predictors.

Although this is commonly used for trees, this approach is model-agnostic in that any model could be used.

Take a model of two predictors, $\hat{\mathbf{y}} = f(\mathbf{X}_1, \mathbf{X_2})$. For simplicity, say that $f(x_1, x_2) = 2x_1 + 3x_2$.

The data looks like this

```
df <- tibble(x1 = c(1,1,2,2), x2 = c(3,4,5,6)) %>%
  mutate(f = 2*x1 + 3*x2)
df
```

```
## # A tibble: 4 x 3
##      x1    x2     f
##   <dbl> <dbl> <dbl>
## 1     1     3    11
## 2     1     4    14
```

```
## 3     2     5    19
## 4     2     6    22
```

Here is the partial dependence of `x1` on to `f`.

```
df %>% group_by(x1) %>% summarise(f = mean(f))
```

```
## # A tibble: 2 x 2
##      x1     f
##   <dbl> <dbl>
## 1     1  12.5
## 2     2  20.5
```

This method of using the mean is know as the *Monte Carlo* method. There are other methods for partial dependence that are not on the syllabus.

For the RandomForest, this is done with `pdp::partial()`.

```
library(pdp)
bmi <- pdp::partial(rf, pred.var = "bmi",
                    grid.resolution = 20) %>%
  autoplot() + theme_bw()
age <- pdp::partial(rf, pred.var = "age",
                    grid.resolution = 20) %>%
  autoplot() + theme_bw()

ggarrange(bmi, age)
```

13.3.5 Advantages and disadvantages

Advantages

- Resilient to overfitting due to bagging
- Only one parameter to tune (mtry, the number of features considered at each split)
- Very good a multi-class prediction
- Nonlinearities
- Interaction effects
- Handles missing data
- Deals with unbalanced after over/undersampling

Disadvantages

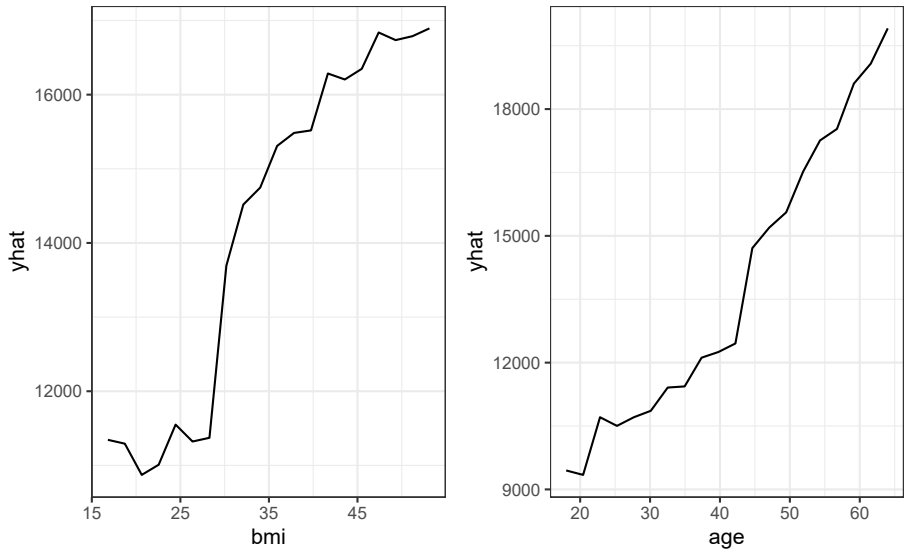


Figure 13.1: Partial Dependence

- Does not work on small data sets
- Weaker performance than other methods (GBM, NN)
- Unable to predict beyond training data for regression

Readings
ISLR 8.2.1 Bagging
ISLR 8.1.2 Random Forests

13.4 Gradient Boosted Trees

Another ensemble learning method is *gradient boosting*, also known as the Gradient Boosted Machine (GBM). Although this is unlikely to get significant attention on the PA exam due to the complexity, this is the most widely-used and powerful machine learning algorithms that are in use today.

We start with an initial model, which is just a constant prediction of the mean.

$$f = f_0(\mathbf{x_i}) = \frac{1}{n}\sum_{i=1}^{n} y_i$$

Then we update the target (what the model is predicting) by subtracting off the previously predicted value.

$$\hat{y}_i \leftarrow y_i - f_0(\mathbf{x_i})$$

This $\hat{y}_i$ is called the *residual*. In our example, instead of predicting `charges`, this would be predicting the residual of $\text{charges}_i - \text{Mean(charges)}$. We now use this model for the residuals to update the prediction.

If we updated each prediction with the prior residual directly, the algorithm would be unstable. To make this process more gradual, we use a *learning rate*

We then iterate through this process hundreds or thousands of times, slowly improving the prediction.

Because each new tree is fit to *residuals* instead of the response itself, the process continuously improves the prediction. As the prediction improves, the residuals get smaller and smaller. In random forests, or other bagging algorithms, the model performance is more limited by the individual trees because each only contributes to the overall average. The name is *gradient boosting* because the residuals are an approximation of the gradient, and gradient descent is how the loss functions are optimized.

Similarly to how GLMs can be used for classification problems through a logit transform (aka logistic regression), GBMs can also be used for classification.

13.4.1 Parameters

For random forests, the individual tree parameters do not get tuned. For GBMs, however, these parameters can make a significant difference in model performance.

Boosting parameters:

- `n.trees`: Integer specifying the total number of trees to fit. This is equivalent to the number of iterations and the number of basis functions in the additive expansion. Default is 100.
- `shrinkage`: a shrinkage parameter applied to each tree in the expansion. Also known as the learning rate or step-size reduction; 0.001 to 0.1 usually work, but a smaller learning rate typically requires more trees. Default is 0.1.

Tree parameters:

- `interaction.depth`: Integer specifying the maximum depth of each tree (i.e., the highest level of variable interactions allowed). A value of 1 implies an additive model, a value of 2 implies a model with up to 2-way interactions, etc. Default is 1.
- `n.minobsinnode`: Integer specifying the minimum number of observations in the terminal nodes of the trees. Note that this is the actual number of observations, not the total weight.

GBMs are easy to overfit, and the parameters need to be carefully tuned using cross-validation. In the Examples section we go through how to do this.

> **Tip:** Whenever fitting a model, use `?model_name` to get the documentation. The parameters below are from `?gbm`.

13.4.2 Example

We fit a gbm below without tuning the parameters for the sake of example.

```
library(gbm)
gbm <- gbm(charges ~ ., data = train,
           n.trees = 100,
           interaction.depth = 2,
           n.minobsinnode = 50,
           shrinkage = 0.1)
```

```
## Distribution not specified, assuming gaussian ...
```

```
pred <- predict(gbm, test, n.trees = 100)

get_rmsle(test$charges, pred)
```

```
## [1] 0.4411494
```

```
get_rmsle(test$charges, mean(train$charges))
```

```
## [1] 0.9996513
```

13.4.3 Advantages and disadvantages

This exam covers the basics of GBMs. There are many variations of GBMs not covered in detail such as `xgboost`.

Advantages

- High prediction accuracy
- Shown to work empirically well on many types of problems
- Nonlinearities, interaction effects, resilient to outliers, corrects for missing values
- Deals with class imbalance directly by weighting observations

Disadvantages

- Requires large sample size
- Longer training time
- Does not detect linear combinations of features. These must be engineered Can overfit if not tuned correctly

Readings
ISLR 8.2.3 Boosting

13.5 Exercises

```
library(ExamPAData)
library(tidyverse)
```

Run this code on your computer to answer these exercises.

13.5.1 1. RF with `randomForest`

(Part 1 of 2)

The below code is set up to fit a random forest to the `soa_mortality` data set to predict `actual_cnt`.

There is a problem: all of the predictions are coming out to be 1. Find out why this is happening and fix it.

```
set.seed(42)
#For the sake of this example, only take 20% of the records
df <- soa_mortality %>%
  sample_frac(0.2) %>%
  mutate(target = as.factor(ifelse(actual_cnt == 0, 1, 0))) %>%
  select(target, prodcat, distchan, smoker, sex, issage, uwkey) %>%
  mutate_if(is.character, ~as.factor(.x))

#check that the target has 0's and 1's
df %>% count(target)
```

```
library(caret)
library(randomForest)
index <- createDataPartition(y = df$target, p = 0.8, list = F)

train <- df %>% slice(index)
test <- df %>% slice(-index)

k = 0.5
cutoff=c(k,1-k)
```

```
model <- randomForest(
  formula = target ~ .,
  data = train,
  ntree = 100,
  cutoff = cutoff
  )

pred <- predict(model, test)
confusionMatrix(pred, test$target)
```

(Part 2 of 2)

Downsample the majority class and refit the model, and then choose between the original data and the downsampled data based on the model performance. Use your own judgement when choosing how to evaluate the model based on accuracy, sensitivity, specificity, and Kappa.

```
down_train <- downSample(x = train %>% select(-target),
                         y = train$target)

down_test <- downSample(x = test %>% select(-target),
                         y = test$target)

down_train %>% count(Class)
```

```
model <- randomForest(
  formula = Class ~ .,
  data = down_train,
  ntree = 100,
  cutoff = cutoff
  )

down_pred <- predict(model, down_test)
confusionMatrix(down_pred, down_test$Class)
```

Now up-sample the minority class and repeat the same procedure.

```
up_train <- upSample(x = train %>% select(-target),
                         y = train$target)

up_test <- upSample(x = test %>% select(-target),
                         y = test$target)

up_train %>% count(Class)
```

```
model <- randomForest(
  formula = Class ~ .,
  data = up_train,
  ntree = 100,
  cutoff = cutoff
  )

up_pred <- predict(model, up_test)
confusionMatrix(up_pred, up_test$Class)
```

13.5.2 2. RF tuning with caret

The best practice of tuning a model is with cross-validation. This can only be done in the `caret` library. If the SOA asks you to use `caret`, they will likely ask you a question related to cross validation as below.

An actuary has trained a predictive model and chosen the best hyperparameters, cleaned the data, and performed feature engineering. They have one problem, however: the error on the training data is far lower than on new, unseen test data. Read the code below and determine their problem. Find a way to lower the error on the test data *without changing the model or the data.* Explain the rational behind your method.

```
set.seed(42)
#Take only 1000 records
#Uncomment this when completing this exercise
data <- health_insurance %>% sample_n(1000)

index <- createDataPartition(
  y = data$charges, p = 0.8, list = F) %>%
  as.numeric()
train <-  health_insurance %>% slice(index)
test <- health_insurance %>% slice(-index)

control <- trainControl(
  method='boot',
  number=2,
  p = 0.2)

tunegrid <- expand.grid(.mtry=c(1,3,5))
rf <- train(charges ~ .,
            data = train,
            method='rf',
            tuneGrid=tunegrid,
```

```
            trControl=control)

pred_train <- predict(rf, train)
pred_test <- predict(rf, test)

get_rmse <- function(y, y_hat){
  sqrt(mean((y - y_hat)^2))
}

get_rmse(pred_train, train$charges)
get_rmse(pred_test, test$charges)
```

13.5.3 3. Tuning a GBM with `caret`

If the SOA asks you to tune a GBM, they will need to give you starting hyper-parameters which are close to the "best" values due to how slow the Prometric computers are. Another possibility is that they pre-train a GBM model object and ask that you use it.

This example looks at 135 combinations of hyper parameters.

```
set.seed(42)
index <- createDataPartition(y = health_insurance$charges,
                             p = 0.8, list = F)
train <- health_insurance %>% slice(index)
test <- health_insurance %>% slice(-index)

tunegrid <- expand.grid(
    interaction.depth = c(1,5, 10),
    n.trees = c(100, 200, 300, 400, 500),
    shrinkage = c(0.5, 0.1, 0.0001),
    n.minobsinnode = c(5, 30, 100)
    )
nrow(tunegrid)
```

```
## [1] 135
```

```
control <- trainControl(
  method='repeatedcv',
  number=5,
  p = 0.8)
```

```
gbm <- train(charges ~ .,
             data = train,
             method='gbm',
             tuneGrid=tunegrid,
             trControl=control,
             #Show detailed output
             verbose = FALSE
             )
```

The output shows the RMSE for each of the 135 models tested.

(Part 1 of 3)

Identify the hyperpameter combination that has the lowest training error.

(Part 2 of 3)

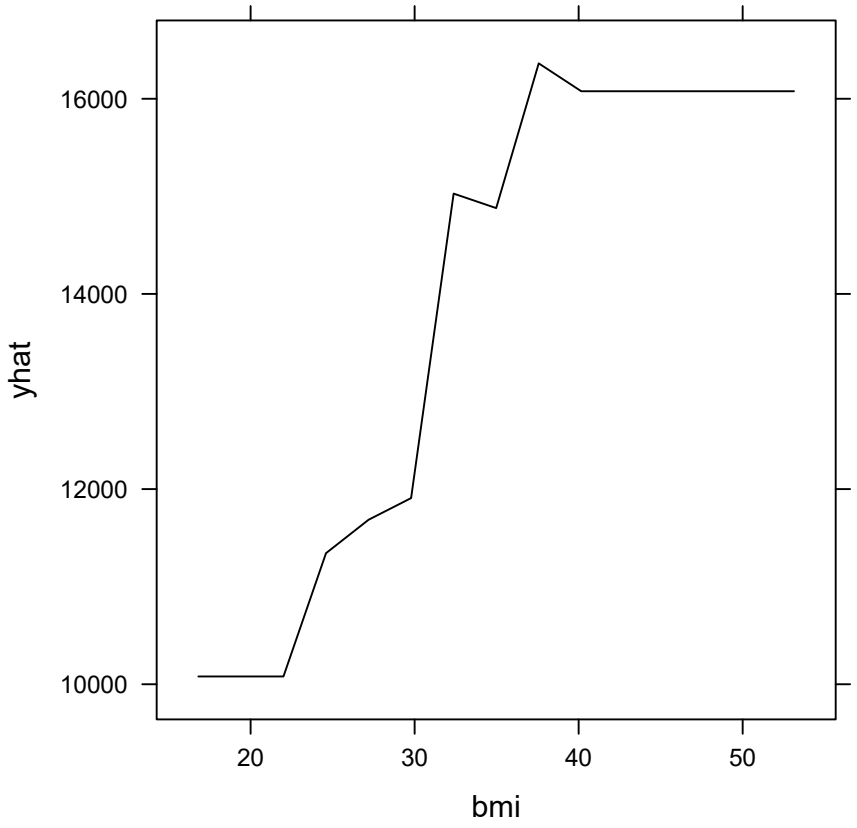
2. Suppose that the optimization measure was RMSE. The below table shows the results from three models. Explain why some sets of parameters have better RMSE than the others.

```
results <- gbm$results %>% arrange(RMSE)
top_result <- results %>% slice(1)%>% mutate(param_rank = 1)
tenth_result <- results %>% slice(10)%>% mutate(param_rank = 10)
twenty_seventh_result <- results %>% slice(135)%>% mutate(param_rank = 135)

rbind(top_result, tenth_result, twenty_seventh_result) %>%
  select(param_rank, 1:5)
```

```
##   param_rank shrinkage interaction.depth n.minobsinnode n.trees      RMSE
## 1          1     1e-01                 5             30     100  4396.814
## 2         10     1e-01                10             30     300  4630.433
## 3        135     1e-04                 1            100     100 12108.185
```

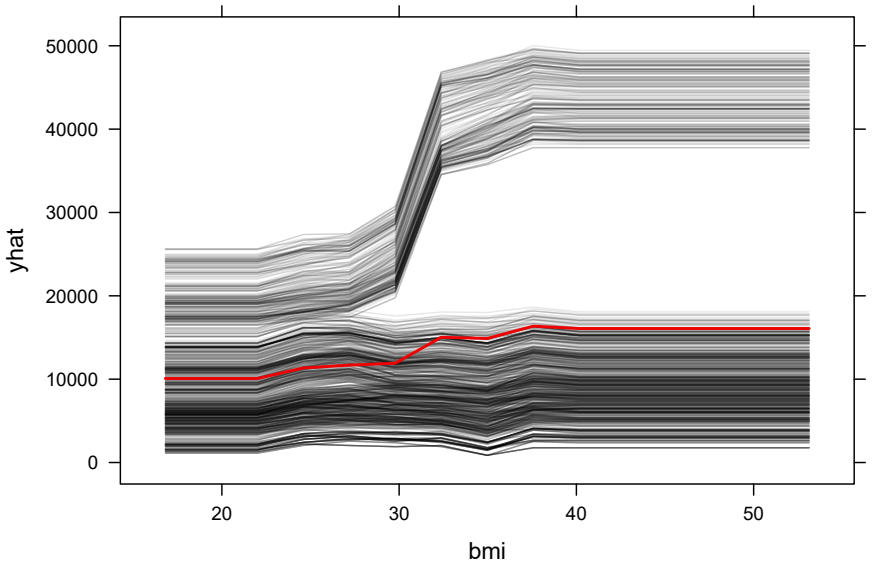
3. The partial dependence of `bmi` onto `charges` makes it appear as if `charges` increases monotonically as `bmi` increases.

```
pdp::partial(gbm, pred.var = "bmi", grid.resolution = 15, plot = T)
```



However, when we add in the `ice` curves, we see that there is something else going on. Explain this graph. Why are there two groups of lines?

```
pdp::partial(gbm, pred.var = "bmi", grid.resolution = 15, plot = T, ice = T, alpha = 0.1, palette
```



13.6 Answers to Exercises

Answers to these exercises are available at ExamPA.net.

Chapter 14

Unsupervised Learning

Up to this point we have been trying to predict something. We use data X to predict an unknown Y, which is a number in the case of regression and a category in the case of classification. These problems are known as *supvervised* because we are "supervising" how the model learns by giving it a target.

Here is an easy way to remember this: If you are in kindergarden class, and the teacher provides cards with pictures and words, and then asks each kid to look at the picture and then say the word, this is supervised learning. The label is the name of the picture, and the data is the picture itself. Instead, if the teacher gives out finger paints and says "express yourself!", then this is *unsupervised learning*. There is no label but only data.

14.1 Principal Componant Analysis (PCA)

Often there are a lot of columns in the data that contain redundant information. For example, if your data consists of city traffic, such as (1) the number of cars on the road, (2) number of taxis, (3) number of pedestrians, and (4) number of Ubers, then knowing any one of these values will tell you about how busy the given road is. If there are a lot of Ubers, then there are probably also a lot of Taxis. Intuitively, we probably don't need four variables to measure this info and we could have a single variable called "traffic level". This would be reducing the dimension from 4 to 1.

PCA is a dimensionality reduction method which reduces the number of variables needed to retain most of the information in a matrix. If there are predictor variables $X_1, X_2, X_3, X_4, X5$, then running PCA and choosing the first three Principal Components (PCs) will reduce the dimension from 5 to 3.

Each PC is a linear combination of the original Xs. For example, PC1 might be

$$PC_1 = 0.2X_1 + 0.3X_2 - 0.2X_3 + 0X_5 + 0.3X_5$$

The weights here are also called "loadings" or "rotations", and are (0.2, 0.3, -0.2, 0, 0.3). Each of the PCs can be interpreted as explaining part of the data. In the traffic example, PC1 might explain the traffic level, PC2 the weather, and PC3 the time of day.

Readings
ISLR 10.2 Principal Component Analysis
ISLR 10.3 Clustering Methods

14.1.1 Example: PCA on US Arrests

In this example, we perform PCA on the `USArrests` data set, which is part of the base `R` package. The rows of the data set contain the 50 states, in alphabetical order:

```
library(tidyverse)
states=row.names(USArrests)
states
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
```

```
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

The columns of the data set contain four variables relating to various crimes:

```
glimpse(USArrests)
```

```
## Observations: 50
## Variables: 4
## $ Murder   <dbl> 13.2, 10.0, 8.1, 8.8, 9.0, 7.9, 3.3, 5.9, 15.4, 17.4,...
## $ Assault  <int> 236, 263, 294, 190, 276, 204, 110, 238, 335, 211, 46,...
## $ UrbanPop <int> 58, 48, 80, 50, 91, 78, 77, 72, 80, 60, 83, 54, 83, 6...
## $ Rape     <dbl> 21.2, 44.5, 31.0, 19.5, 40.6, 38.7, 11.1, 15.8, 31.9,...
```

Let's start by taking a quick look at the column means of the data.

```
USArrests %>% summarise_all(mean)
```

```
##   Murder Assault UrbanPop   Rape
## 1  7.788  170.76    65.54 21.232
```

We see right away the the data have **vastly** different means. We can also examine the variances of the four variables.

```
USArrests %>% summarise_all(var)
```

```
##     Murder  Assault UrbanPop     Rape
## 1 18.97047 6945.166 209.5188 87.72916
```

Not surprisingly, the variables also have vastly different variances: the `UrbanPop` variable measures the percentage of the population in each state living in an urban area, which is not a comparable number to the number of crimes committeed in each state per 100,000 individuals. If we failed to scale the variables before performing PCA, then most of the principal components that we observed would be driven by the `Assault` variable, since it has by far the largest mean and variance.

Thus, it is important to standardize the variables to have mean zero and standard deviation 1 before performing PCA. We'll perform principal components analysis using the `prcomp()` function, which is one of several functions that perform PCA. By default, this centers the variables to have mean zero. By using the option `scale=TRUE`, we scale the variables to have standard deviation 1:

```
pca = prcomp(USArrests, scale=TRUE)
```

The output from `prcomp()` contains a number of useful quantities:

```
names(pca)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

The `center` and `scale` components correspond to the means and standard deviations of the variables that were used for scaling prior to implementing PCA:

```
pca$center
```

```
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

```
pca$scale
```

```
##    Murder   Assault  UrbanPop      Rape
##  4.355510 83.337661 14.474763  9.366385
```

The rotation matrix provides the principal component loadings; each column of `pr.out\$rotation` contains the corresponding principal component loading vector:

```
pca$rotation
```

```
##                 PC1        PC2        PC3         PC4
## Murder   -0.5358995  0.4181809 -0.3412327  0.64922780
## Assault  -0.5831836  0.1879856 -0.2681484 -0.74340748
## UrbanPop -0.2781909 -0.8728062 -0.3780158  0.13387773
## Rape     -0.5434321 -0.1673186  0.8177779  0.08902432
```

We see that there are four distinct principal components. This is to be expected because there are in general `min(n - 1, p)` informative principal components in a data set with n observations and p variables.

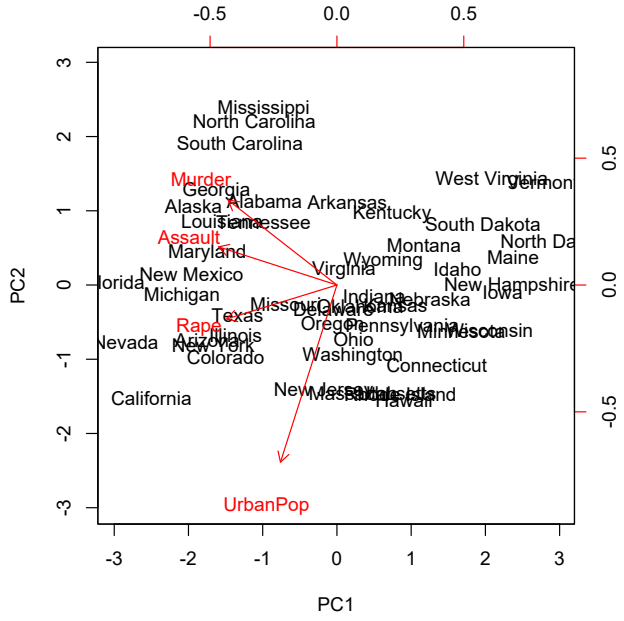
Using the `prcomp()` function, we do not need to explicitly multiply the data by the principal component loading vectors in order to obtain the principal component score vectors. Rather the 50×4 matrix x has as its columns the principal component score vectors. That is, the \$k^{thcolumn is the \$k^{th principal component score vector. We'll take a look at the first few states:

```
head(pca$x)
```

```
##                  PC1        PC2         PC3          PC4
## Alabama    -0.9756604  1.1220012 -0.43980366  0.154696581
## Alaska     -1.9305379  1.0624269  2.01950027 -0.434175454
## Arizona    -1.7454429 -0.7384595  0.05423025 -0.826264240
## Arkansas    0.1399989  1.1085423  0.11342217 -0.180973554
## California -2.4986128 -1.5274267  0.59254100 -0.338559240
## Colorado   -1.4993407 -0.9776297  1.08400162  0.001450164
```

We can plot the first two principal components using the `biplot()` function:

```
biplot(pca, scale=0)
```



The `scale=0` argument to `biplot()` ensures that the arrows are scaled to represent the loadings; other values for `scale` give slightly different biplots with different interpretations.

The `prcomp()` function also outputs the standard deviation of each principal component. We can access these standard deviations as follows:

```
pca$sdev
```

```
## [1] 1.5748783 0.9948694 0.5971291 0.4164494
```

The variance explained by each principal component is obtained by squaring these:

```
pca_var=pca$sdev^2
pca_var
```

```
## [1] 2.4802416 0.9897652 0.3565632 0.1734301
```

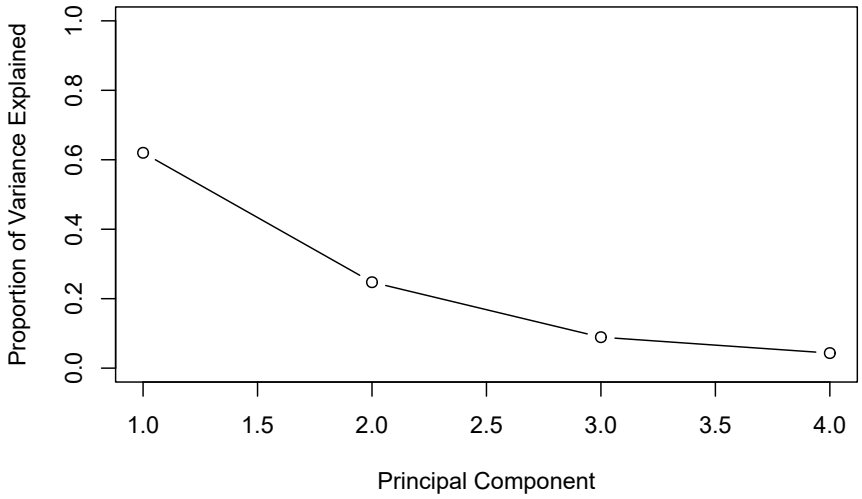
To compute the proportion of variance explained by each principal component, we simply divide the variance explained by each principal component by the total variance explained by all four principal components:

```
pve=pca_var/sum(pca_var)
pve
```

```
## [1] 0.62006039 0.24744129 0.08914080 0.04335752
```

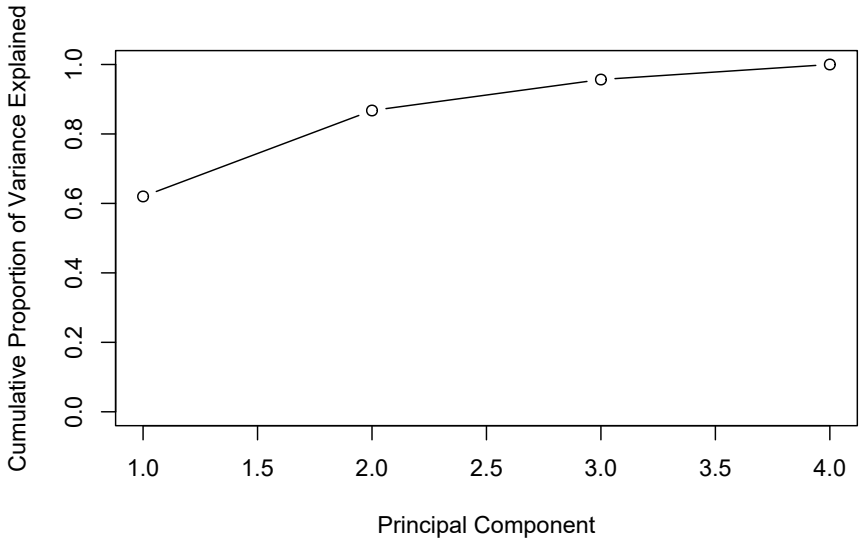
We see that the first principal component explains 62.0% of the variance in the data, the next principal component explains 24.7% of the variance, and so forth. We can plot the PVE explained by each component as follows:

```
plot(pve, xlab="Principal Component", ylab="Proportion of Variance Explained", ylim=c(0
```



We can also use the function `cumsum()`, which computes the cumulative sum of the elements of a numeric vector, to plot the cumulative PVE:

```
plot(cumsum(pve), xlab="Principal Component", ylab="Cumulative Proportion of Variance Explained",
```



```
a=c(1,2,8,-3)
cumsum(a)
```

```
## [1]  1  3 11  8
```

14.1.2 Example: PCA on Cancel Cells

The data `NCI60` contins expression levels of 6,830 genes from 64 cancel cell lines. Cancer type is also recorded.

```
library(ISLR)
nci_labs=NCI60$labs
nci_data=NCI60$data
```

We first perform PCA on the data after scaling the variables (genes) to have standard deviation one, although one could reasonably argue that it is better not to scale the genes:

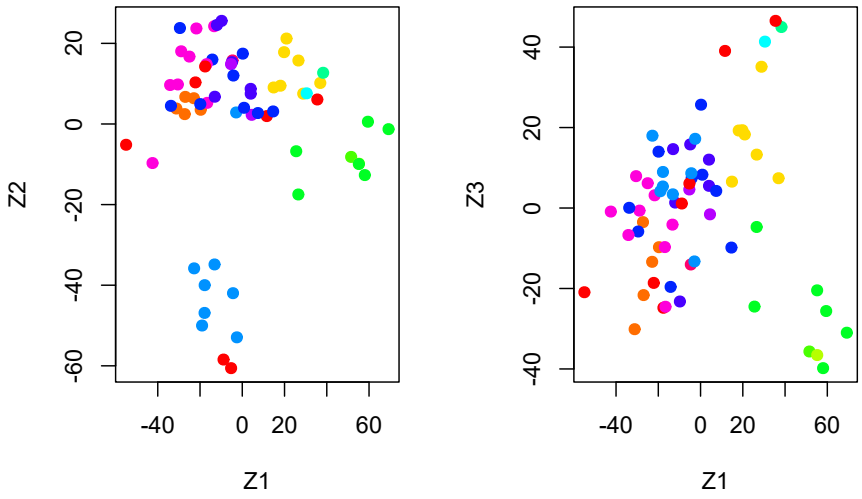
```
pca = prcomp(nci_data, scale=TRUE)
```

We now plot the first few principal component score vectors, in order to visualize the data. The observations (cell lines) corresponding to a given cancer type will be plotted in the same color, so that we can see to what extent the observations within a cancer type are similar to each other. We first create a simple function that assigns a distinct color to each element of a numeric vector. The function will be used to assign a color to each of the 64 cell lines, based on the cancer type to which it corresponds. We'll make use of the `rainbow()` function, which takes as its argument a positive integer, and returns a vector containing that number of distinct colors.

```
Cols=function(vec){
    cols=rainbow(length(unique(vec)))
    return(cols[as.numeric(as.factor(vec))])
  }
```

We now can plot the principal component score vectors:

```
par(mfrow=c(1,2))
plot(pca$x[,1:2], col=Cols(nci_labs), pch=19,xlab="Z1",ylab="Z2")
plot(pca$x[,c(1,3)], col=Cols(nci_labs), pch=19,xlab="Z1",ylab="Z3")
```



On the whole, cell lines corresponding to a single cancer type do tend to have similar values on the first few principal component score vectors. This indicates that cell lines from the same cancer type tend to have pretty similar gene expression levels.

We can obtain a summary of the proportion of variance explained (PVE) of the first few principal components using the `summary()` method for a `prcomp` object:

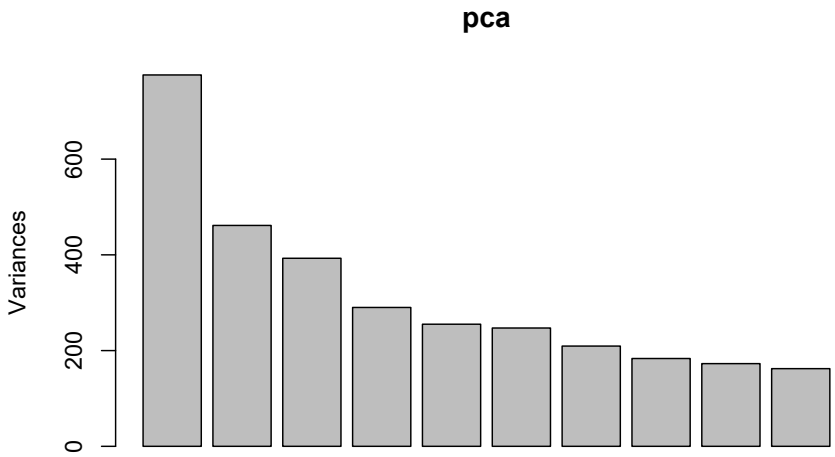
```
summary(pca)
```

```
## Importance of components:
##                             PC1      PC2      PC3      PC4      PC5
## Standard deviation     27.8535 21.48136 19.82046 17.03256 15.97181
## Proportion of Variance  0.1136  0.06756  0.05752  0.04248  0.03735
## Cumulative Proportion   0.1136  0.18115  0.23867  0.28115  0.31850
##                             PC6      PC7      PC8      PC9     PC10
## Standard deviation     15.72108 14.47145 13.54427 13.14400 12.73860
## Proportion of Variance  0.03619  0.03066  0.02686  0.02529  0.02376
## Cumulative Proportion   0.35468  0.38534  0.41220  0.43750  0.46126
##                            PC11     PC12     PC13     PC14     PC15
## Standard deviation     12.68672 12.15769 11.83019 11.62554 11.43779
## Proportion of Variance  0.02357  0.02164  0.02049  0.01979  0.01915
## Cumulative Proportion   0.48482  0.50646  0.52695  0.54674  0.56590
##                            PC16     PC17     PC18     PC19     PC20
```

```
## Standard deviation     11.00051 10.65666 10.48880 10.43518 10.3219
## Proportion of Variance  0.01772  0.01663  0.01611  0.01594  0.0156
## Cumulative Proportion   0.58361  0.60024  0.61635  0.63229  0.6479
##                            PC21    PC22    PC23    PC24    PC25    PC26
## Standard deviation     10.14608 10.0544 9.90265 9.64766 9.50764 9.33253
## Proportion of Variance  0.01507  0.0148 0.01436 0.01363 0.01324 0.01275
## Cumulative Proportion   0.66296  0.6778 0.69212 0.70575 0.71899 0.73174
##                            PC27   PC28    PC29    PC30    PC31    PC32
## Standard deviation     9.27320 9.0900 8.98117 8.75003 8.59962 8.44738
## Proportion of Variance 0.01259 0.0121 0.01181 0.01121 0.01083 0.01045
## Cumulative Proportion  0.74433 0.7564 0.76824 0.77945 0.79027 0.80072
##                            PC33    PC34    PC35    PC36    PC37    PC38
## Standard deviation     8.37305 8.21579 8.15731 7.97465 7.90446 7.82127
## Proportion of Variance 0.01026 0.00988 0.00974 0.00931 0.00915 0.00896
## Cumulative Proportion  0.81099 0.82087 0.83061 0.83992 0.84907 0.85803
##                            PC39    PC40    PC41   PC42    PC43   PC44
## Standard deviation     7.72156 7.58603 7.45619 7.3444 7.10449 7.0131
## Proportion of Variance 0.00873 0.00843 0.00814 0.0079 0.00739 0.0072
## Cumulative Proportion  0.86676 0.87518 0.88332 0.8912 0.89861 0.9058
##                            PC45   PC46    PC47    PC48    PC49    PC50
## Standard deviation     6.95839 6.8663 6.80744 6.64763 6.61607 6.40793
## Proportion of Variance 0.00709 0.0069 0.00678 0.00647 0.00641 0.00601
## Cumulative Proportion  0.91290 0.9198 0.92659 0.93306 0.93947 0.94548
##                            PC51    PC52    PC53    PC54    PC55    PC56
## Standard deviation     6.21984 6.20326 6.06706 5.91805 5.91233 5.73539
## Proportion of Variance 0.00566 0.00563 0.00539 0.00513 0.00512 0.00482
## Cumulative Proportion  0.95114 0.95678 0.96216 0.96729 0.97241 0.97723
##                            PC57   PC58    PC59    PC60    PC61    PC62
## Standard deviation     5.47261 5.2921 5.02117 4.68398 4.17567 4.08212
## Proportion of Variance 0.00438 0.0041 0.00369 0.00321 0.00255 0.00244
## Cumulative Proportion  0.98161 0.9857 0.98940 0.99262 0.99517 0.99761
##                            PC63      PC64
## Standard deviation     4.04124 2.148e-14
## Proportion of Variance 0.00239 0.000e+00
## Cumulative Proportion  1.00000 1.000e+00
```

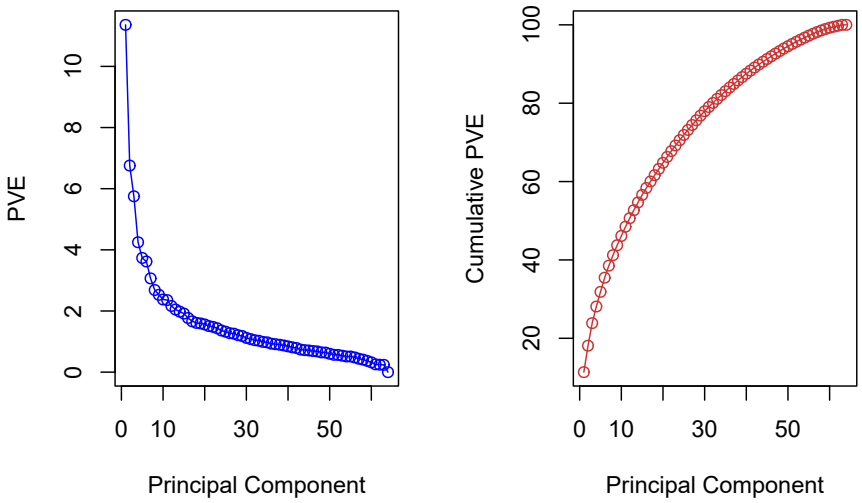
Using the `plot()` function, we can also plot the variance explained by the first few principal components:

```
plot(pca)
```



Note that the height of each bar in the bar plot is given by squaring the corresponding element of `pr.out\$sdev`. However, it is generally more informative to plot the PVE of each principal component (i.e. a **scree plot**) and the cumulative PVE of each principal component. This can be done with just a little tweaking:

```
pve = 100*pca$sdev^2/sum(pca$sdev^2)
par(mfrow=c(1,2))
plot(pve,  type="o", ylab="PVE", xlab="Principal Component", col="blue")
plot(cumsum(pve), type="o", ylab="Cumulative PVE", xlab="Principal Component", col="brown3")
```



We see that together, the first seven principal components explain around 40% of the variance in the data. This is not a huge amount of the variance. However, looking at the scree plot, we see that while each of the first seven principal components explain a substantial amount of variance, there is a marked decrease in the variance explained by further principal components. That is, there is an **elbow** in the plot after approximately the seventh principal component. This suggests that there may be little benefit to examining more than seven or so principal components (phew! even examining seven principal components may be difficult).

14.2 Clustering

Imagine that you are a large retailer interested in understanding the customer base. There may be several "types" of customers, such as those shopping for business with corporate accounts, those shopping for leisure, or debt-strapped grad students. Each of these customers would exhibit different behavior, and should be treated differently statistically. But how can a customer's "type" be defined? Especially for large customer data sets in the millions, one can imagine how this problem can be challenging.

Clustering algorithms look for groups of observations which are similar to one another. Because there is no target variable, measuring the quality of the "fit" is much more complicated. There are many clustering algorithms, but this exam only focuses on the two that are most common.

14.2.1 K-Means Clustering

Kmeans takes continuous data and assigns observations into k clusters, or groups. In the two-dimensional example, this is the same as drawing lines around points. This consists of the following steps:

a) Start with two variables (X_1 on the X-axis, and X_2 on the Y-axis.) **b)** Randomly assign cluster centers. **c)** Put each point into the cluster that is closest. **d) - f)** Move the cluster center to the mean of the points assigned to it and continue until the centers stop moving. **g)** Repeated steps a) - f) a given number of times (controlled by `n.starts`). This reduces the uncertainty from choosing the initial centers randomly.

In R, the function kmeans() performs K-means clustering in R. We begin with a simple simulated example in which there truly are two clusters in the data: the first 25 observations have a mean shift relative to the next 25 observations.

```
set.seed(2)
x = matrix(rnorm(50*2), ncol = 2)
x[1:25,1] = x[1:25,1]+3
x[1:25,2] = x[1:25,2]-4
```

We now perform K-means clustering with K = 2:

```
km_out = kmeans(x,2,nstart = 20)
```

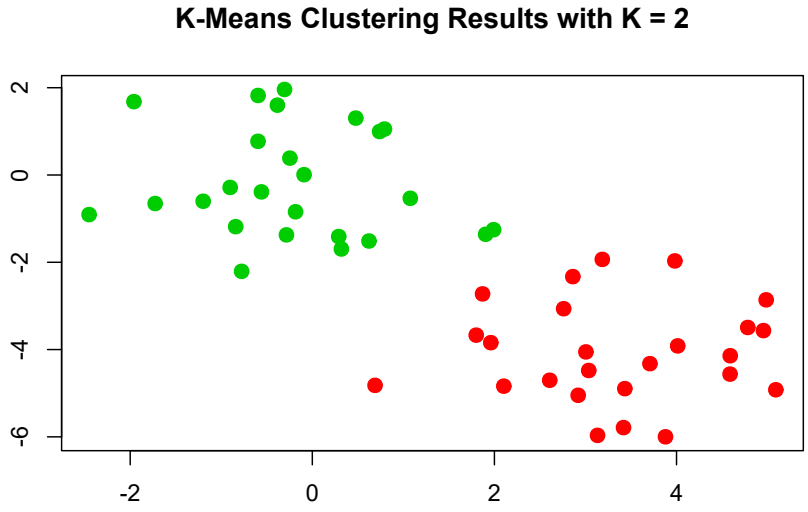
The cluster assignments of the 50 observations are contained in km_out$cluster:

```
km_out$cluster
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
## [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

The K-means clustering perfectly separated the observations into two clusters even though we did not supply any group information to kmeans(). We can plot the data, with each observation colored according to its cluster assignment:

```
plot(x, col = (km_out$cluster+1), main = "K-Means Clustering Results with K = 2", xlab = "", ylab
```



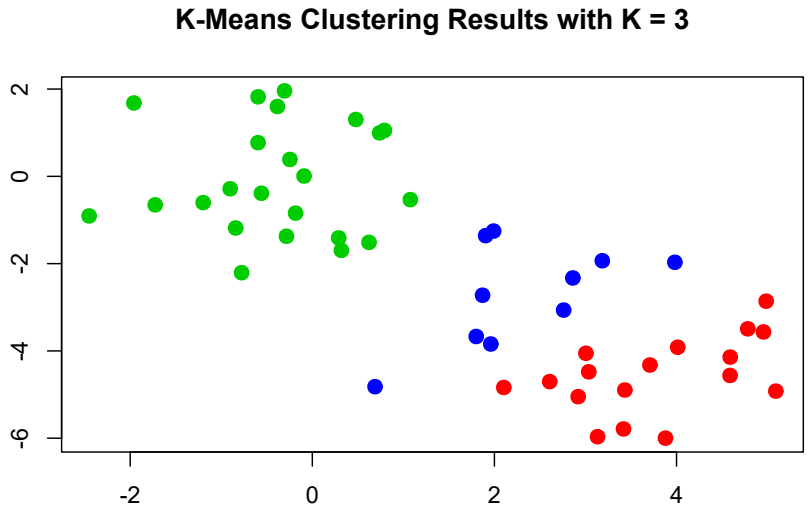
Here the observations can be easily plotted because they are two-dimensional. If there were more than two variables then we could instead perform PCA and plot the first two principal components score vectors.

In this example, we knew that there really were two clusters because we generated the data. However, for real data, in general we do not know the true number of clusters. We could instead have performed K-means clustering on this example with `K = 3`. If we do this, K-means clustering will split up the two "real" clusters, since it has no information about them:

```
set.seed(4)
km_out = kmeans(x, 3, nstart = 20)
km_out
```

```
## K-means clustering with 3 clusters of sizes 17, 23, 10
##
## Cluster means:
##         [,1]        [,2]
## 1  3.7789567 -4.56200798
## 2 -0.3820397 -0.08740753
## 3  2.3001545 -2.69622023
##
## Clustering vector:
##  [1] 1 3 1 3 1 1 1 3 1 3 1 3 1 3 1 3 1 1 1 1 1 3 1 1 1 2 2 2 2 2 2 2 2 2
## [36] 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2
##
## Within cluster sum of squares by cluster:
## [1] 25.74089 52.67700 19.56137
##  (between_SS / total_SS =  79.3 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

```
plot(x, col = (km_out$cluster+1), main = "K-Means Clustering Results with K = 3", xlab
```



To run the `kmeans()` function in R with multiple initial cluster assignments, we use the `nstart` argument. If a value of `nstart` greater than one is used, then K-means clustering will be performed using multiple random assignments, and the `kmeans()` function will report only the best results. Here we compare using `nstart = 1`:

```
set.seed(3)
km_out = kmeans(x, 3, nstart = 1)
km_out$tot.withinss
```

```
## [1] 97.97927
```

to `nstart = 20`:

```
km_out = kmeans(x,3,nstart = 20)
km_out$tot.withinss
```

```
## [1] 97.97927
```

Note that `km_out\$tot.withinss` is the total within-cluster sum of squares, which we seek to minimize by performing K-means clustering. The individual within-cluster sum-of-squares are contained in the vector `km_out\$withinss`.

It is generally recommended to always run K-means clustering with a large value of `nstart`, such as 20 or 50 to avoid getting stuck in an undesirable local optimum.

When performing K-means clustering, in addition to using multiple initial cluster assignments, it is also important to set a random seed using the `set.seed()` function. This way, the initial cluster assignments can be replicated, and the K-means output will be fully reproducible.

14.3 Hierarchical Clustering

Kmeans required that we choose the number of clusters, k. Hierarchical clustering is an alternative that does not require that we choose only one value of k.

The most common type of hiearchical clustering uses a *bottom-up* approach. This starts with a single **observation** and then looks for others which are close and puts them into a cluster. Then it looks for other **clusters** that are similar and groups these together into a **megacluster**. It continues to do this until all observations are in the same group.

This is analyzed with a graph called a dendrogram (dendro = tree, gram = graph). The height represents "distance", or how similar the clusters are to one another. The clusters on the bottom, which are vertically close to one another, have similar data values; the clusters that are further apart vertically are less similar.

Choosing the value of the cutoff height changes the number of clusters that result.

Certain data have a natural hiearchical structure. For example, say that the variables are City, Town, State, Country, and Continent. If we used hiearchical clustering, this pattern could be established even if we did not have labels for Cities, Towns, and so forth.

The `hclust()` function implements hierarchical clustering in R. In the following example we use the data from the previous section to plot the hierarchical clustering dendrogram using complete, single, and average linkage clustering, with Euclidean distance as the dissimilarity measure. We begin by clustering observations using complete linkage. The `dist()` function is used to compute the 50×50 inter-observation Euclidean distance matrix:

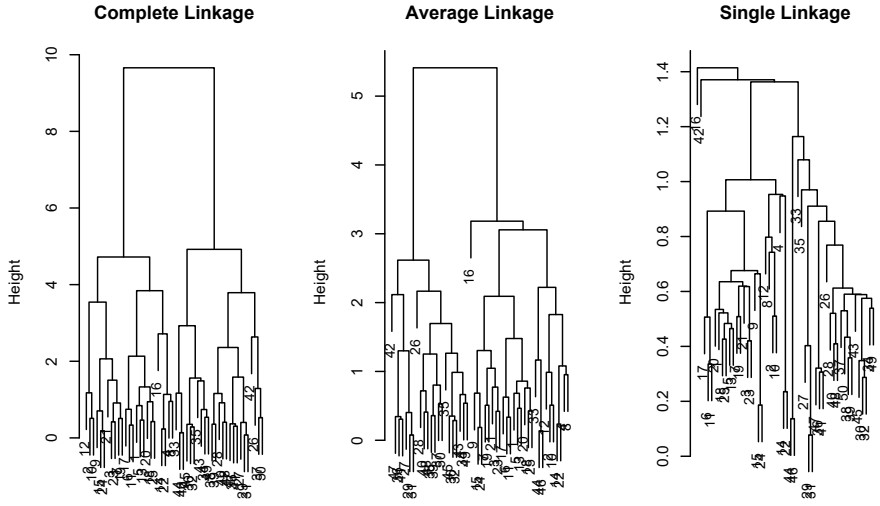
```
hc_complete = hclust(dist(x), method = "complete")
```

We could just as easily perform hierarchical clustering with average or single linkage instead:

```
hc_average = hclust(dist(x), method = "average")
hc_single = hclust(dist(x), method = "single")
```

We can now plot the dendrograms obtained using the usual `plot()` function. The numbers at the bottom of the plot identify each observation:

```
par(mfrow = c(1,3))
plot(hc_complete,main = "Complete Linkage", xlab = "", sub = "", cex = .9)
plot(hc_average, main = "Average Linkage", xlab = "", sub = "", cex = .9)
plot(hc_single, main = "Single Linkage", xlab = "", sub = "", cex = .9)
```



To determine the cluster labels for each observation associated with a given cut of the dendrogram, we can use the `cutree()` function:

```
cutree(hc_complete, 2)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
## [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

```
cutree(hc_average, 2)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 2 2
## [36] 2 2 2 2 2 2 2 2 2 1 2 1 2 2 2 2 2
```

```
cutree(hc_single, 2)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

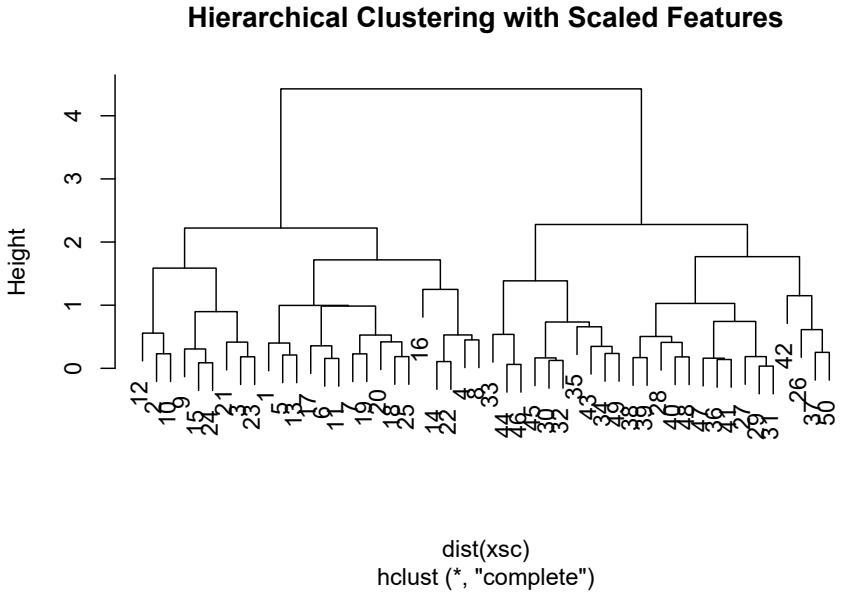
For this data, complete and average linkage generally separate the observations into their correct groups. However, single linkage identifies one point as belonging to its own cluster. A more sensible answer is obtained when four clusters are selected, although there are still two singletons:

```
cutree(hc_single, 4)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3
## [36] 3 3 3 3 3 3 4 3 3 3 3 3 3 3 3
```

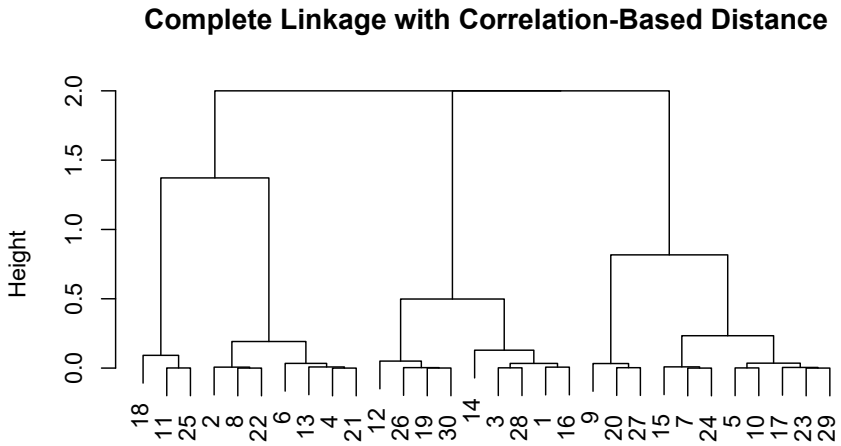
To scale the variables before performing hierarchical clustering of the observations, we can use the `scale()` function:

```
xsc = scale(x)
plot(hclust(dist(xsc), method = "complete"), main = "Hierarchical Clustering with Scale
```



Correlation-based distance can be computed using the `as.dist()` function, which converts an arbitrary square symmetric matrix into a form that the `hclust()` function recognizes as a distance matrix. However, this only makes sense for data with **at least three features** since the absolute correlation between any two observations with measurements on two features is always 1. Let's generate and cluster a three-dimensional data set:

```
x = matrix(rnorm(30*3), ncol = 3)
dd = as.dist(1-cor(t(x)))
plot(hclust(dd, method = "complete"), main = "Complete Linkage with Correlation-Based D
```



14.3.1 Example: Clustering Cancel Cells

Unsupervised techniques are often used in the analysis of genomic data. In this example, we'll see how hierarchical and K-means clustering compare on the `NCI60` cancer cell line microarray data, which consists of 6,830 gene expression measurements on 64 cancer cell lines:

```
# The NCI60 data
library(ISLR)
nci_labels = NCI60$labs
nci_data = NCI60$data
```

Each cell line is labeled with a cancer type. We'll ignore the cancer types in performing clustering, as these are unsupervised techniques. After performing clustering, we'll use this column to see the extent to which these cancer types agree with the results of these unsupervised techniques.

The data has 64 rows and 6,830 columns.

```
dim(nci_data)
```

```
## [1]   64 6830
```

Let's take a look at the cancer types for the cell lines:

```
table(nci_labels)
```

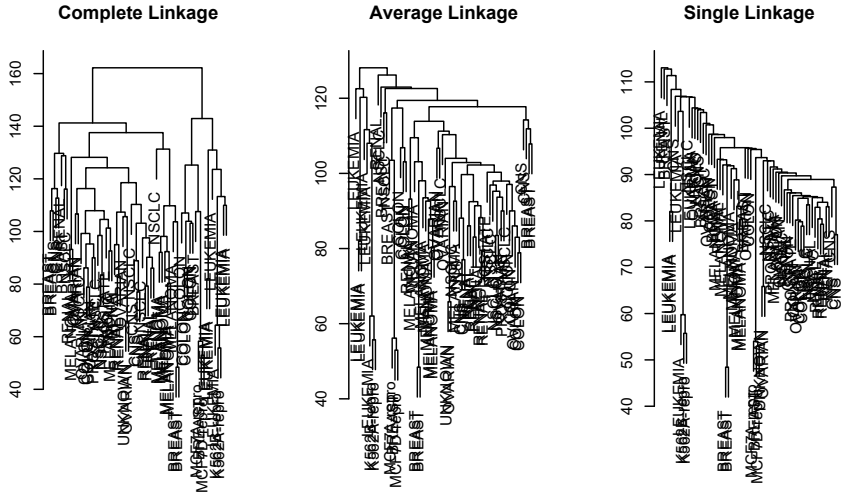
```
## nci_labels
##      BREAST         CNS       COLON K562A-repro K562B-repro    LEUKEMIA
##           7           5           7           1           1           6
## MCF7A-repro MCF7D-repro    MELANOMA       NSCLC     OVARIAN    PROSTATE
##           1           1           8           9           6           2
##       RENAL     UNKNOWN
##           9           1
```

We now proceed to hierarchically cluster the cell lines in the `NCI60` data, with the goal of finding out whether or not the observations cluster into distinct types of cancer. To begin, we standardize the variables to have mean zero and standard deviation one. This step is optional, and need only be performed if we want each gene to be on the same scale:

```
sd_data = scale(nci_data)
```

We now perform hierarchical clustering of the observations using complete, single, and average linkage. We'll use standard Euclidean distance as the dissimilarity measure:

```
par(mfrow = c(1,3))
data_dist = dist(sd_data)
plot(hclust(data_dist), labels = nci_labels, main = "Complete Linkage", xlab = "", sub
plot(hclust(data_dist, method = "average"), labels = nci_labels, main = "Average Linka
plot(hclust(data_dist, method = "single"), labels = nci_labels,  main = "Single Linkag
```



We see that the choice of linkage certainly does affect the results obtained. Typically, single linkage will tend to yield trailing clusters: very large clusters onto which individual observations attach one-by-one. On the other hand, complete and average linkage tend to yield more balanced, attractive clusters. For this reason, complete and average linkage are generally preferred to single linkage. Clearly cell lines within a single cancer type do tend to cluster together, although the clustering is not perfect.

Let's use our complete linkage hierarchical clustering for the analysis. We can cut the dendrogram at the height that will yield a particular number of clusters, say 4:

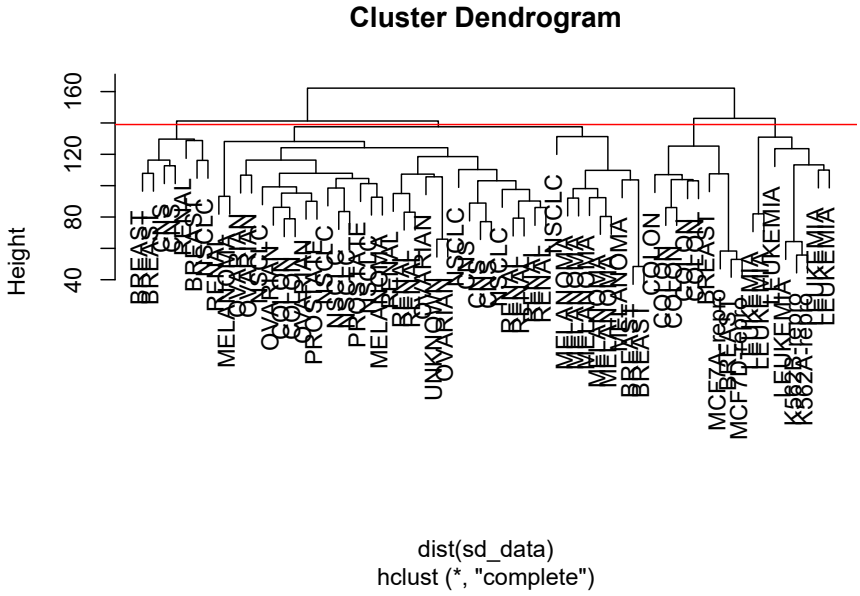
```
hc_out = hclust(dist(sd_data))
hc_clusters = cutree(hc_out,4)
table(hc_clusters,nci_labels)
```

```
##             nci_labels
## hc_clusters BREAST CNS COLON K562A-repro K562B-repro LEUKEMIA MCF7A-repro
##           1      2   3     2           0           0        0           0
##           2      3   2     0           0           0        0           0
##           3      0   0     0           1           1        6           0
##           4      2   0     5           0           0        0           1
##             nci_labels
## hc_clusters MCF7D-repro MELANOMA NSCLC OVARIAN PROSTATE RENAL UNKNOWN
##           1           0        8     8       6        2     8       1
```

```
##           2          0          0          1          0          0          1          0
##           3          0          0          0          0          0          0          0
##           4          1          0          0          0          0          0          0
```

There are some clear patterns. All the leukemia cell lines fall in cluster 3, while the breast cancer cell lines are spread out over three different clusters. We can plot the cut on the dendrogram that produces these four clusters using the `abline()` function, which draws a straight line on top of any existing plot in R:

```
par(mfrow = c(1,1))
plot(hc_out, labels = nci_labels)
abline(h = 139, col = "red")
```



Printing the output of `hclust` gives a useful brief summary of the object:

```
hc_out
```

```
##
## Call:
## hclust(d = dist(sd_data))
##
## Cluster method   : complete
## Distance         : euclidean
## Number of objects: 64
```

We claimed earlier that K-means clustering and hierarchical clustering with the dendrogram cut to obtain the same number of clusters can yield **very** different results. How do these `NCI60` hierarchical clustering results compare to what we get if we perform K-means clustering with `K = 4`?

```
set.seed(2)
km_out = kmeans(sd_data, 4, nstart = 20)
km_clusters = km_out$cluster
```

We can use a confusion matrix to compare the differences in how the two methods assigned observations to clusters:

```
table(km_clusters,hc_clusters)
```

```
##            hc_clusters
## km_clusters  1  2  3  4
##           1 11  0  0  9
##           2 20  7  0  0
##           3  9  0  0  0
##           4  0  0  8  0
```

We see that the four clusters obtained using hierarchical clustering and Kmeans clustering are somewhat different. Cluster 2 in K-means clustering is identical to cluster 3 in hierarchical clustering. However, the other clusters differ: for instance, cluster 4 in K-means clustering contains a portion of the observations assigned to cluster 1 by hierarchical clustering, as well as all of the observations assigned to cluster 2 by hierarchical clustering.

14.3.2 References

These examples are an adaptation of p. 404-407, 410-413 of "Introduction to Statistical Learning with Applications in R" by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. Adapted by R. Jordan Crouser at Smith College for SDS293: Machine Learning (Spring 2016), and re-implemented in Fall 2016 in `tidyverse` format by Amelia McNamara and R. Jordan Crouser at Smith College.

Used with permission from Jordan Crouser at Smith College, and to the following contributors on github:

- github.com/jcrouser
- github.com/AmeliaMN
- github.com/mhusseinmidd
- github.com/rudeboybert
- github.com/ijlyttle

Chapter 15

Practice Exams

Practice exams are available at ExamPA.net.

Chapter 16

References

Burkov, Andriy. 2019. *The Hundred-Page Machine Learning Book.* http://themlbook.com/

Goldburd, Mark et al. 2016.*Generalized Linear Models for Insurance Rating: CAS Monograph Series Number 5.* https://www.casact.org/pubs/monographs/papers/05-Goldburd-Khare-Tevet.pdf

Hastie, Trevor, et al. 2002. *The Elements of Statistical Learning.* Print.

James, Gareth, et al. 2017. *An Introduction to Statistical Learning.* http://faculty.marshall.usc.edu/gareth-james/ISL/ISLR%20Seventh%20Printing.pdf

Piech, Chris and Ng, Andrew. 2019. Stanford CS221. Course Notes. https://stanford.edu/~cpiech/cs221/handouts/kmeans.html

Rigollet, Philippe (2017). *Lecture 21: Generalized Linear Models.* Video. https://www.youtube.com/watch?v=X-ix97pw0xY&t=899s

Wickham, Hadley. 2019. *R for Data Science.* https://r4ds.had.co.nz/