Exam PA Study Manual

Sam Castillo

2019-12-04

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 Get full access at ExamPA.net

- **Pass Guarantee**: Pass in December or get the next 7 months free!
- Three **original** practice exams and more coming soon
- **June Exam PA Video**
- **Hospital Readmissions Video**
- **Student Success Video**
- An online discussion forum
- Time-saving techniques
- An RStudio/Data Manipulation tutorial

1.2 Content Samples

1.2.1 A Response to a Subscriber's Question

An individual on our forum at ExamPA.net just asked this excellent question.

> Note: At 3:09, I say "The L1 norm is lambda", which is mispeaking.
> The L1 norm is the sum of the absolute value of the beta coefficients.

1.2.2 An Original Practice Exam Project Statement

Here you can see the project statement for a practice exam which is in the format of a real exam.

1.3 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

1.4 Contribute

This book was written in RMarkdown using bookdown. It is built from a github page, where updates can be sent to this page in minutes. A big shout out to those who have made suggestions already: Erlan Wheeler, David Hill, Caden Collier, Jon Lai, Peter Schelble, and Abhinav Gadde.

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 toaster-oven 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

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 4

Getting started

4.1 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.

```
#check if devtools is installed
#then install ExamPAData from github
if("devtools" %in% installed.packages()){
  library(devtools)
  install_github("sdcastillo/ExamPAData")
} else{
  install.packages("devtools")
  library(devtools)
  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
```

4.2 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".

4.3 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 5

R programming

This book 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/

5.1 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`.

5.2 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 assigning values in functions (see the functions section).

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

```
#variable assignment
x = 2
y <- 2

#equality
4 == 2 #False
```

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

```
5 == 5 #true
```

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

```
3.14 > 3 #true
```

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

```
3.14 >= 3 #true
```

```
## [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
```

Lists are like vectors but can take any type of object type. I already mentioned `numeric` types. There are also `character` (string) types, `factor` types, and `boolean` types.

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

Characters are combined with the `paste` function.

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

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

Factors are characters that expect only specific values. A character can take on any value. A factor is only allowed a finite number of values. This reduces the memory size.

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. When doing math, bools are converted to 0/1 values where 1 is equivalent to TRUE and 0 FALSE.

```
bool_true <- T
bool_false <- F
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(T,T, F)
sum(bool_vect)
```

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

Vectors are indexed using `[]`.

```
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"
```

5.3 Lists

Lists are vectors that can hold mixed object types. Vectors need to be all of the same type.

```
ls <- list(T, "Character", 3.14)
ls
```

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

Lists can be named.

```
ls <- list(bool = T, character = "character", numeric = 3.14)
ls
```

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

The $ operator indexes lists.

```
ls$numeric
```

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

```
ls$numeric + 5
```

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

Lists can also be indexed using `[]`.

```
ls[1]
```

```
## $bool
## [1] TRUE
```

```
ls[2]
```

```
## $character
## [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 = ls)
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
```

5.4 Functions

You only need to understand the very basics of functions for this exam. Still, 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.

A function that does not return anything

```
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 something, the `return` statement is optional.

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

add_together(2,5)
```

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

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

add_together(2,5)
```

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

Functions can work with vectors.

```
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
```

5.5 Data frames

R is an old programming language. The original `data.frame` object has been updated with the newer and better `tibble` (like the word "table"). **Tibbles are really lists of vectors, where each column is a vector**.

```
#the tibble library has functions for making tibbles
library(tibble)
data <- tibble(age = c(25, 35), has_fsa = c(F, T))
data
```

```
## # 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.

```
data$age
```

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

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

```
dim(data)
```

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

To fine a summary, use `summary`

```
summary(data)
```

```
##       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
```

5.6 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`.

```
#the pipe is from the dplyr library
library(dplyr)
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
```

Chapter 6

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
16	Dates and times
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)
```

6.1 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
```

6.2 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
```

6.3 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?

6.4 Answers to exercises

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

Chapter 7

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.

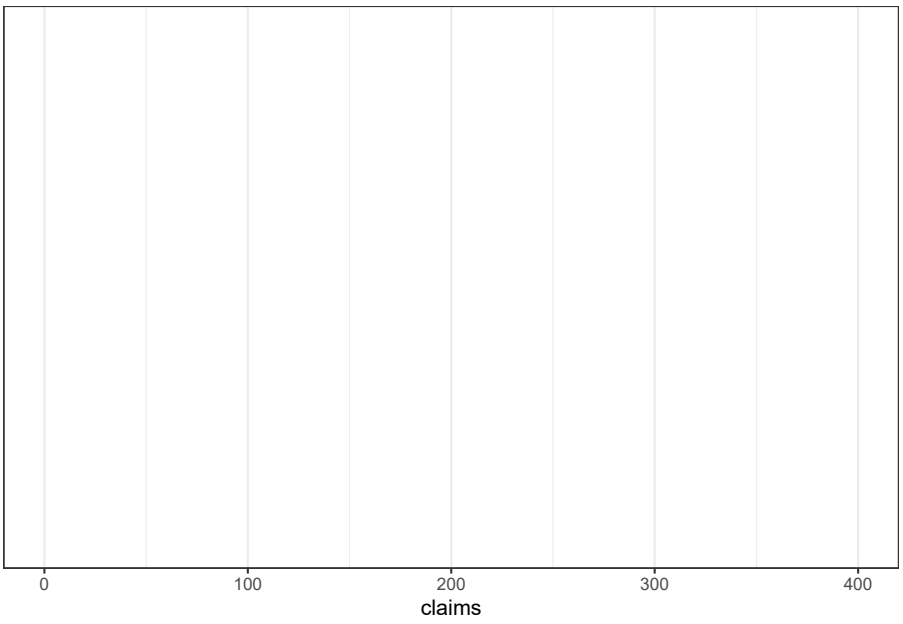
7.1 Create a plot object (ggplot)

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`.

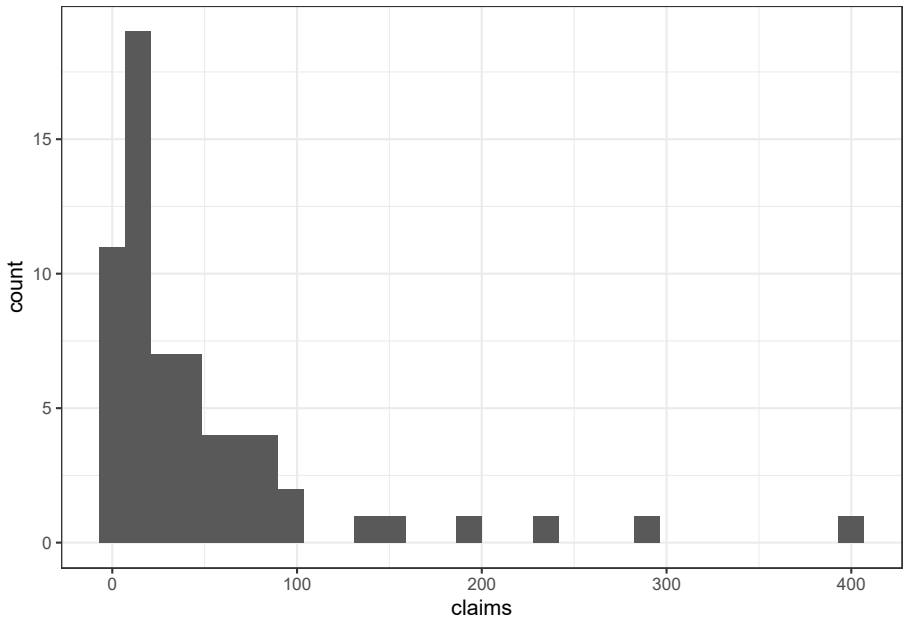
```
p
```



7.2 Add a plot

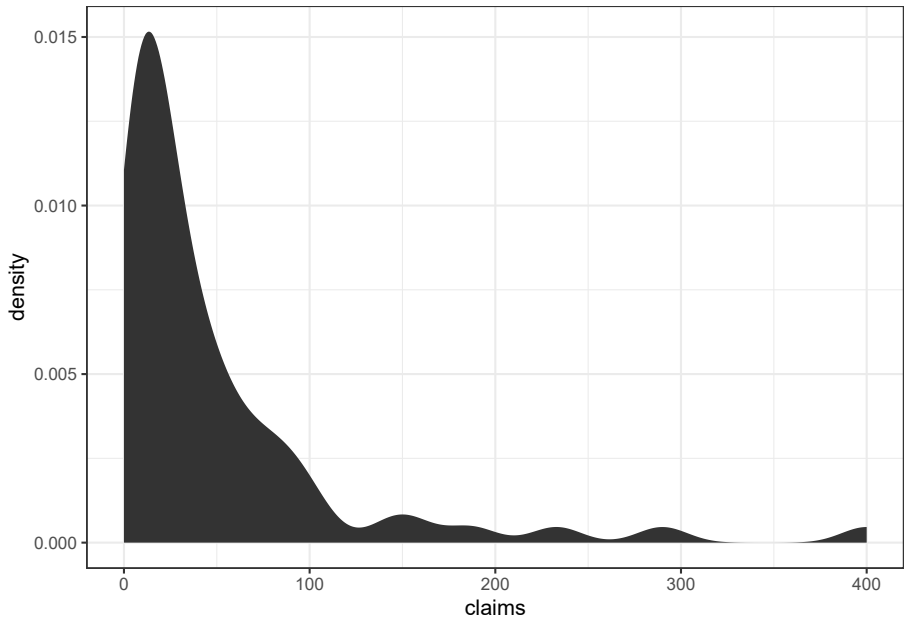
We add a histogram

```
p + geom_histogram()
```



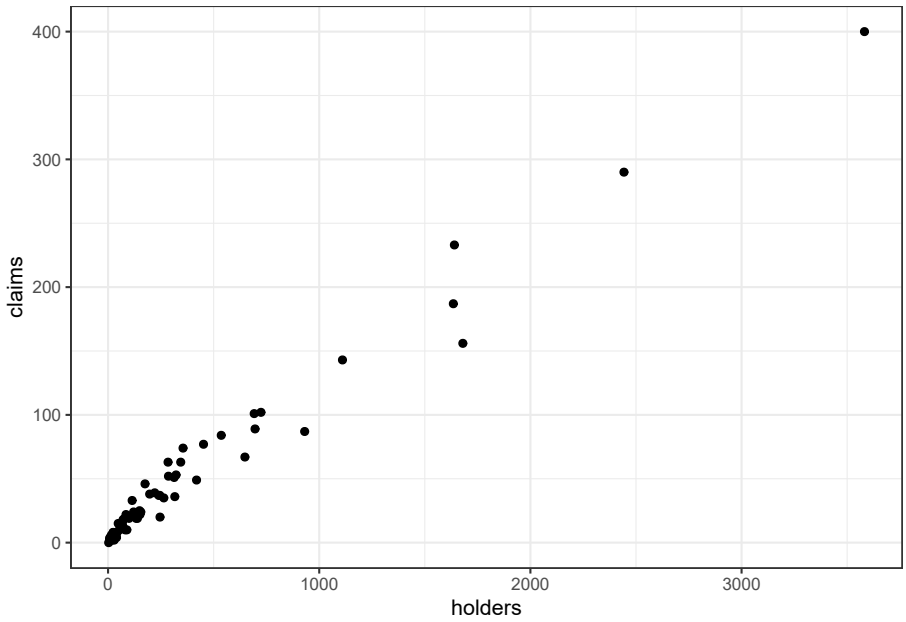
Different plots are called "geoms" for "geometric objects". Geometry = Geo (space) + metre (measure), and graphs measure data. For instance, instead of creating a histogram, we can draw a gamma distribution with `stat_density`.

```
p + stat_density()
```



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

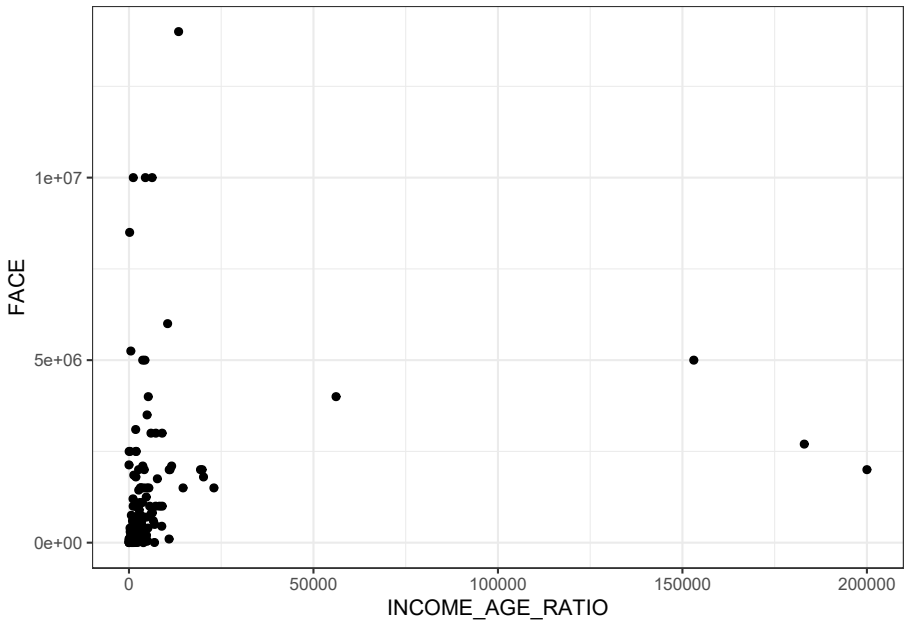
```
insurance %>%
  ggplot(aes(x = holders, y = claims)) +
  geom_point()
```



7.3 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 8

Introduction to Modeling

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

8.1 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 $\mathbf{y}$ 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).

8.2 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.

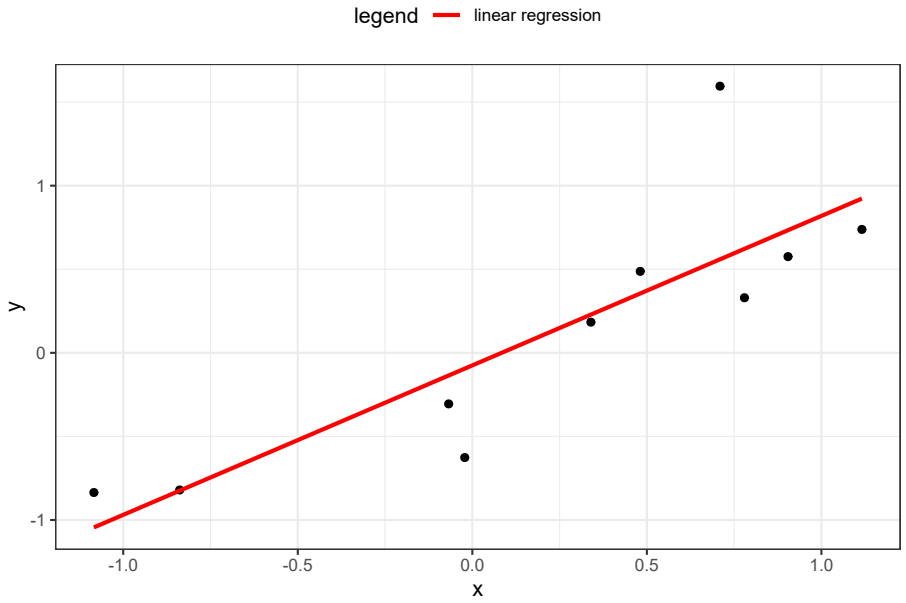
We observe the data X and the want to predict the target Y.

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

8.3 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.

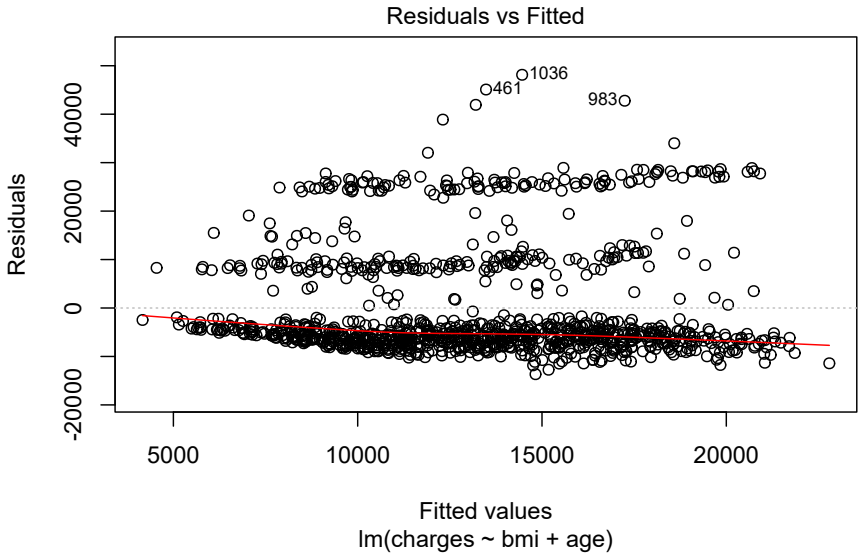


Figure 8.1: Residuals vs. Fitted

```
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

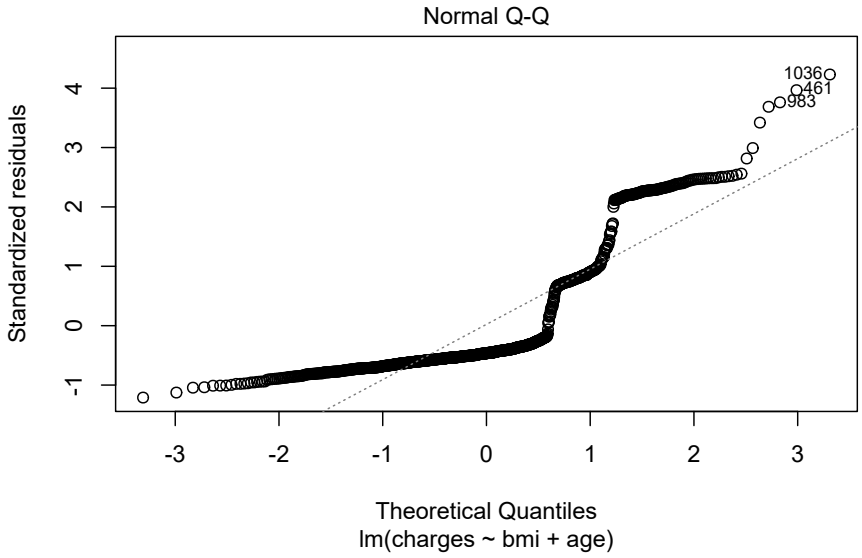


Figure 8.2: Normal Q-Q

Chapter 9

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.

9.1 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, 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 9.1: Distribution-Link Function Combinations

9.2 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.

9.3 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.

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

9.4 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()
```

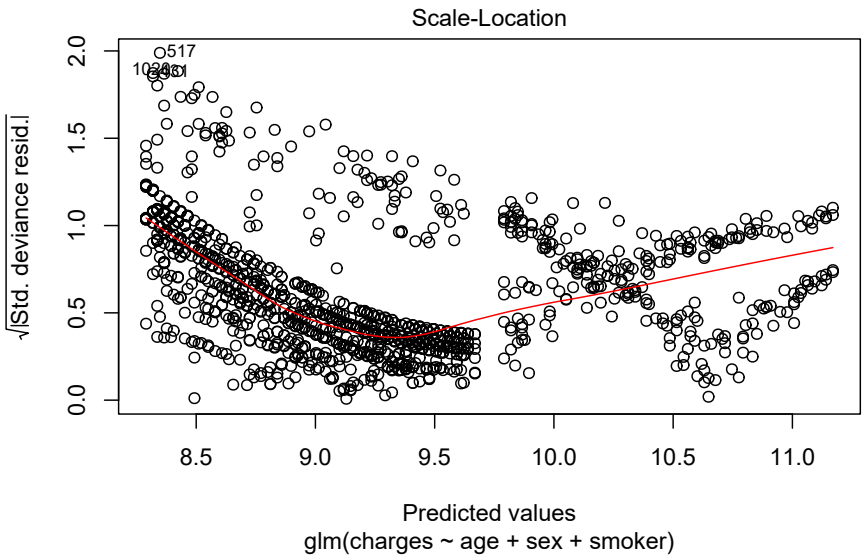
```
## # A tibble: 4 x 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)   7.82     0.0600     130.   0.
## 2 age           0.0290   0.00134     21.6  3.40e- 89
## 3 sexmale      -0.0468   0.0377      -1.24 2.15e-  1
##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



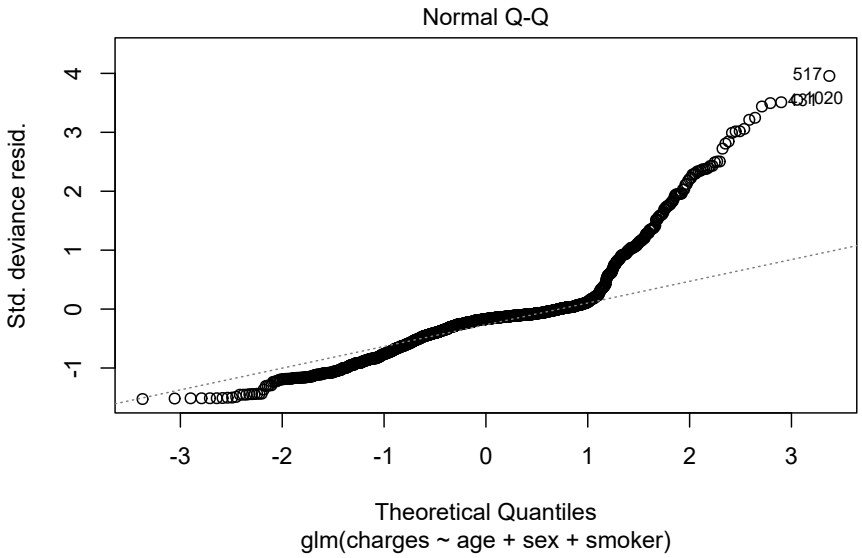
The quantile-quantile (QQ) plot shows the quantiles of the deviance residuals (i.e., after adjusting for the Gamma distribution) against theoretical Gaussian quantiles.

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)
```



9.5 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. In each of these examples, the model will be a perfect fit.

9.5.1 Gaussian Response with Log Link

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 $\mu = E[Y|X]$

We create a function that takes in X and returns a gaussian random variable with mean equal to the inverse link of X. If we say that the link is the log, then the inverse link is the exponent.

```
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. For illustration, here we use X's from a uniform distribution.

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

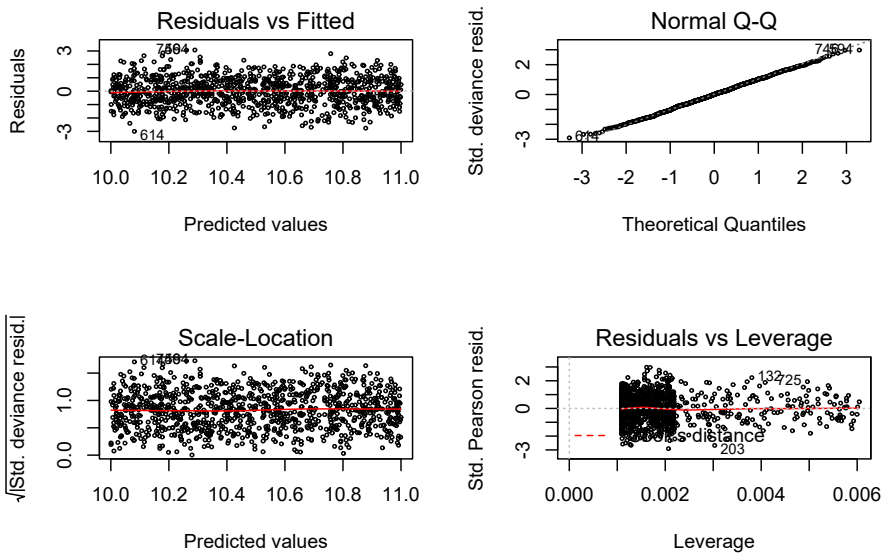
We already know what the answer is: a gaussian response with a log link. We fit a GLM and see a perfect fit.

```
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)
```



9.5.2 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)
```

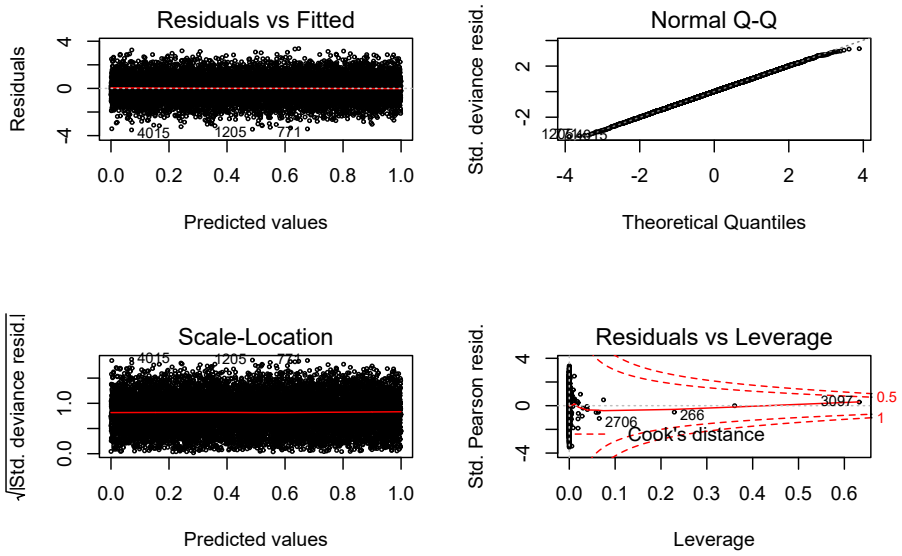
```
##        x                    y
##  Min.   :0.0001064   Min.   :  -1.957
```

```
##  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)
```



9.5.3 Gaussian Response with Identity Link

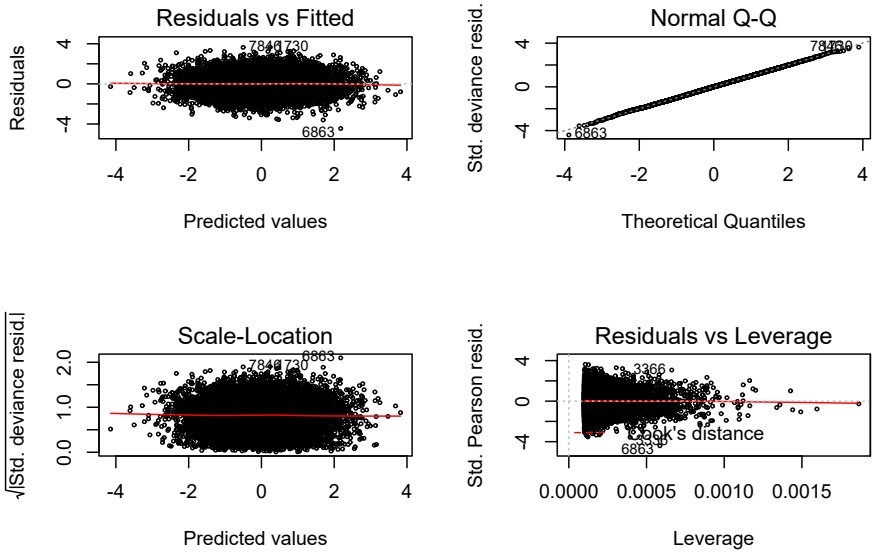
And now the link is the identity, `mean = x`.

```
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)
```



9.5.4 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.

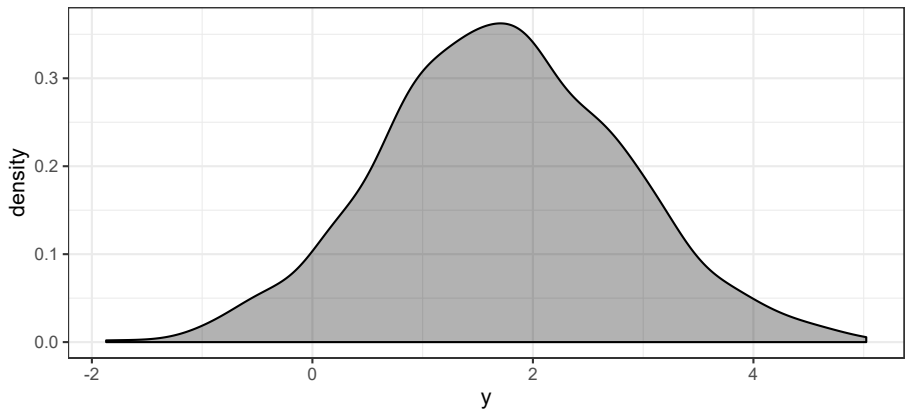
We see below that there are some Y values which are negative.

```
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
```

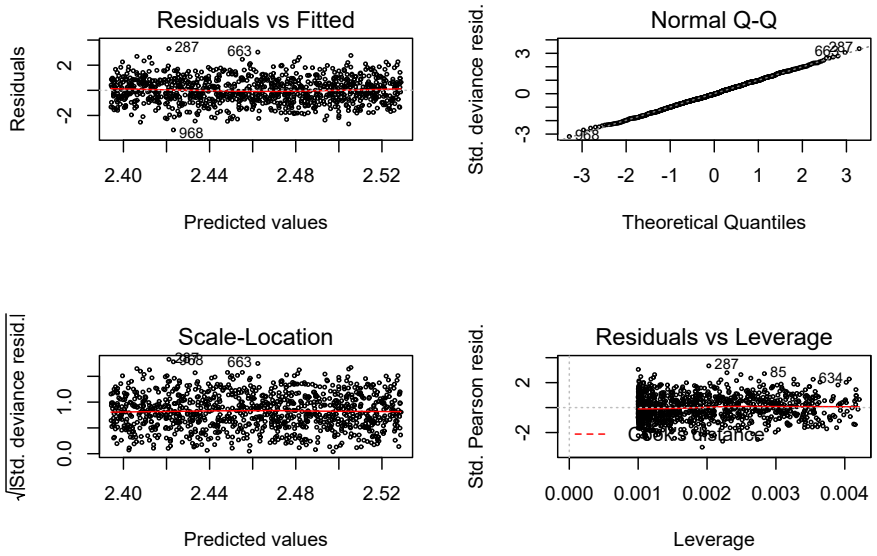
This is because the domain of the natural logarithm only includes positive numbers, and we just tried to take the log of negative numbers.

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)
```



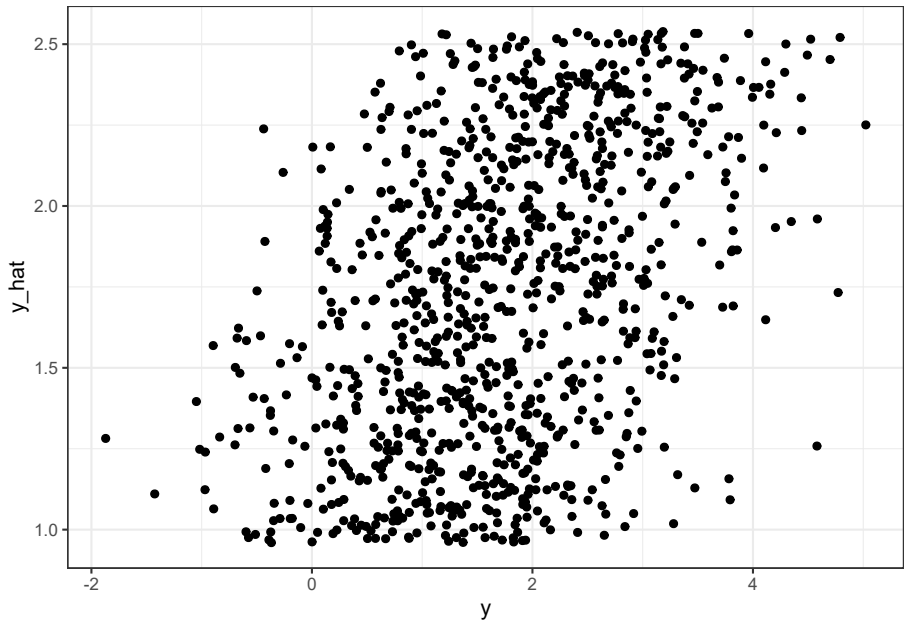
We see that on average, the predictions are 10 higher than the target. This is no surprise since $E[Y + 10] = E[Y] + 10$.

```
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()
```

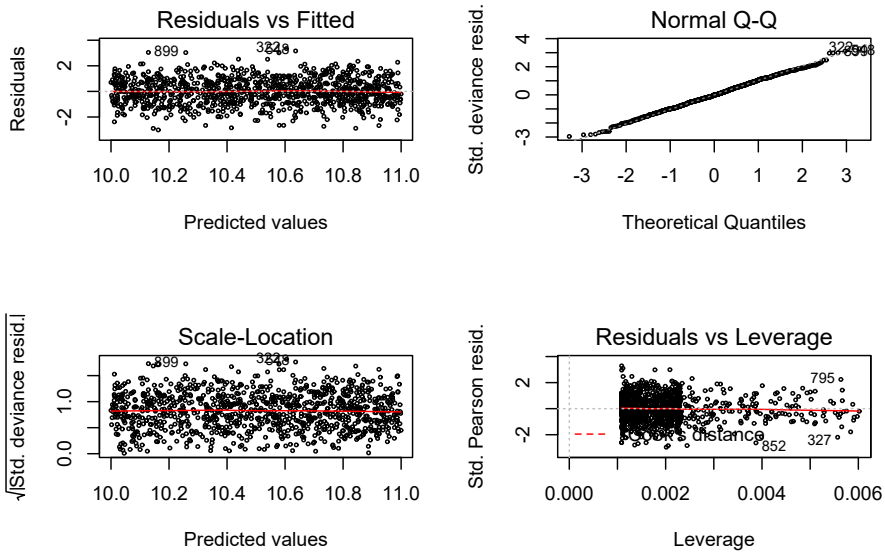


One solution is to adjust the X which the model is based on. Add a constant term to X so that the mean of Y is larger, and hence Y is non zero. While is a viable approach in the case of only one predictor variable, with more predictors this would not be easy to do.

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

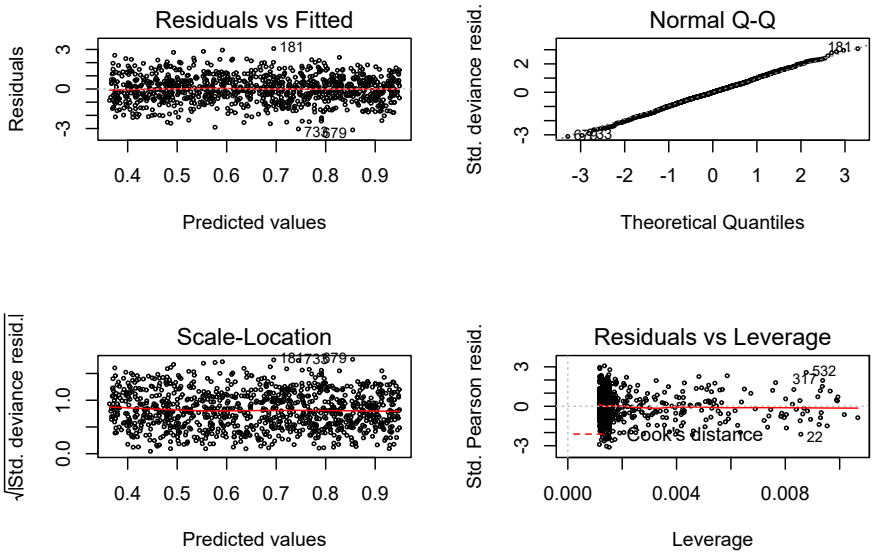
```
##        x               y
##  Min.   :10.00   Min.   :22028
##  1st Qu.:10.25   1st Qu.:28291
##  Median :10.52   Median :36893
##  Mean   :10.51   Mean   :38160
##  3rd Qu.:10.77   3rd Qu.:47441
##  Max.   :11.00   Max.   :59842
```

```
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
```

9.5.5 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
```

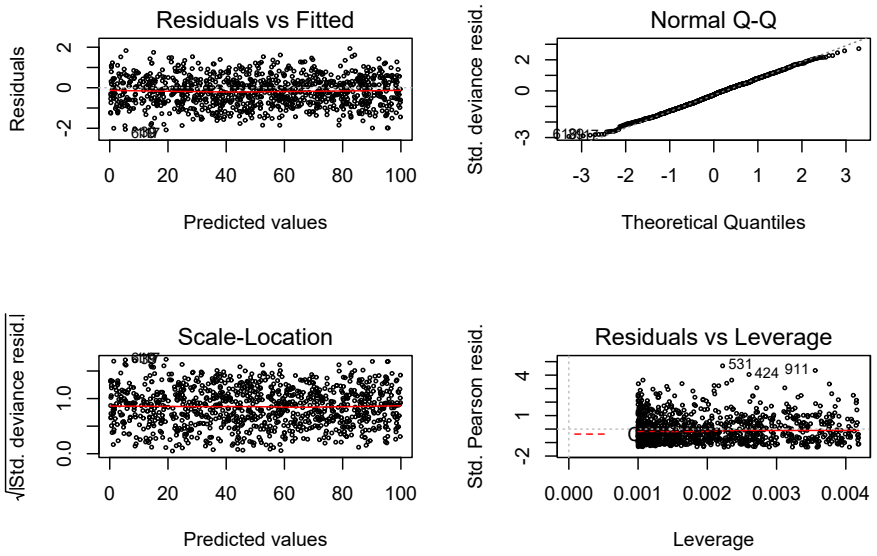
We then generate random gamma values. Because the mean now depends on two paramters instead of one, which was just μ in the Guassian case, we need to use a slightly different approach to simulate the random values. The link function here is seen in `exp(x)`.

```
#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.

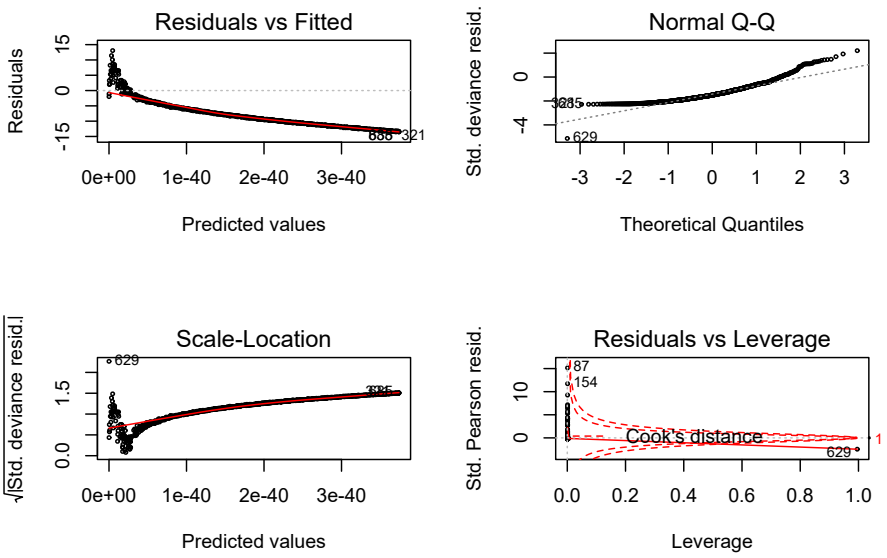
```
glm <- glm(y ~ x, family = Gamma(link = "log"), data = data)
par(mfrow = c(2,2))
plot(glm, cex = 0.4)
```



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
```



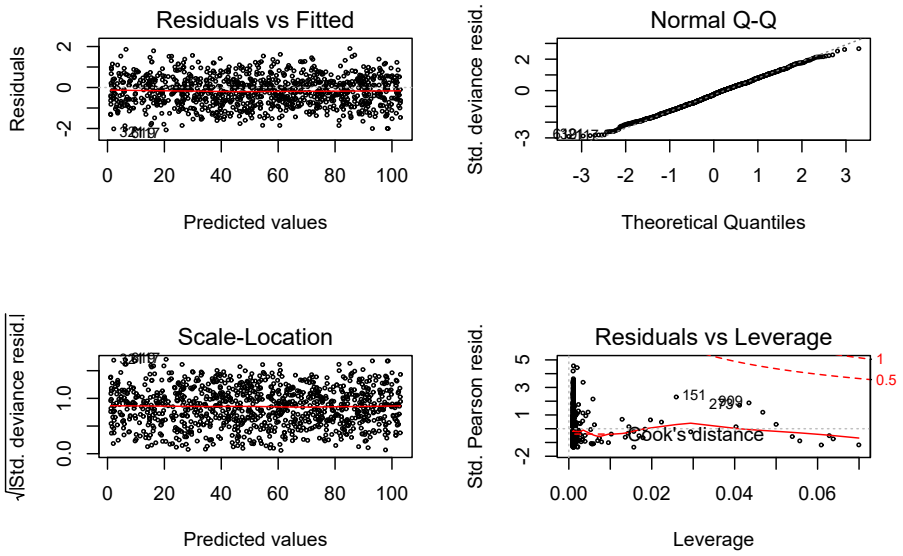
9.6 Gamma with Inverse Link

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)
```



9.7 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

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

9.8 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"
```

9.9 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.

9.10 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

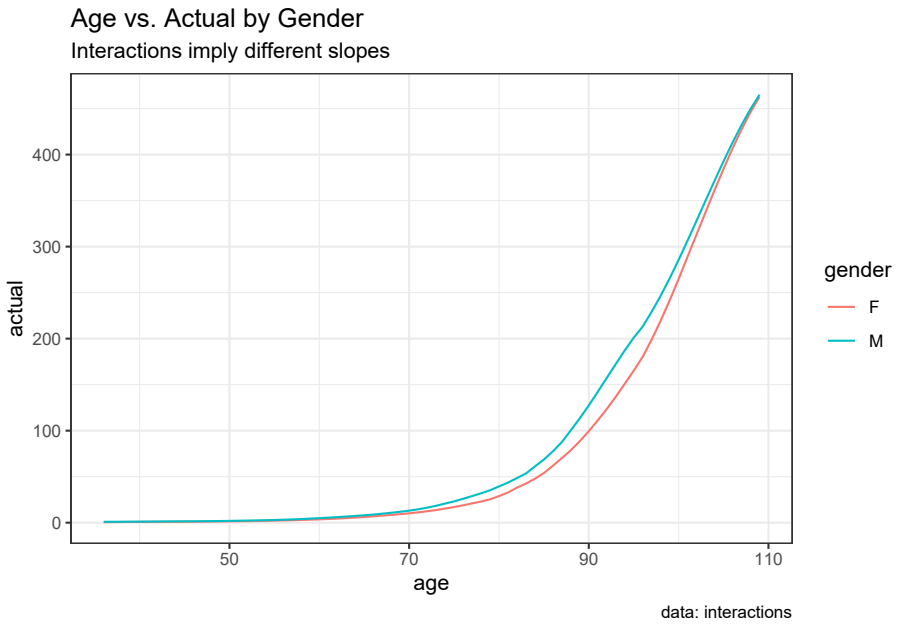


Figure 9.2: Example of weak interaction

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.

9.11 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

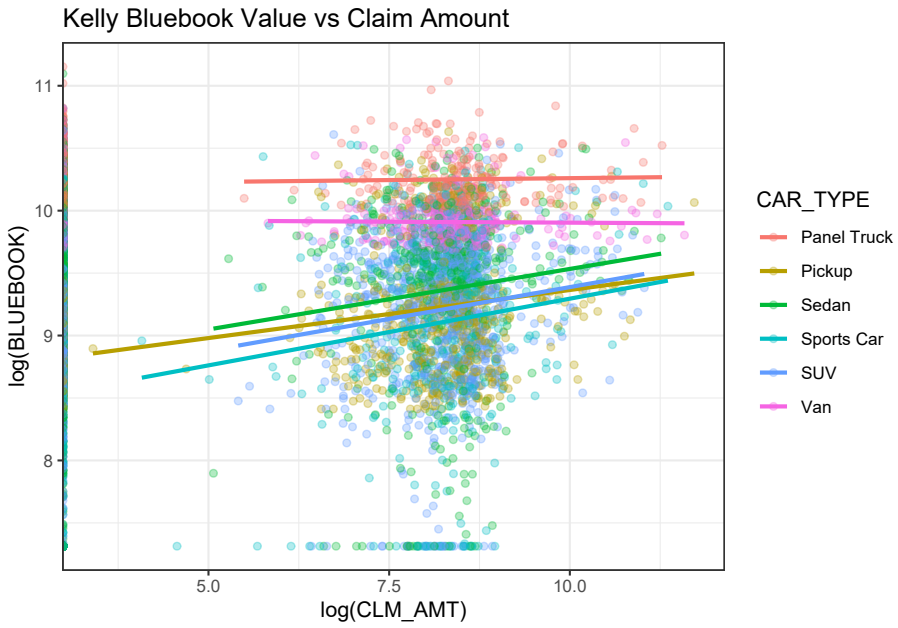


Figure 9.3: Example of strong interaction

to 1. We call this an *offset*.

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

9.12 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.

9.13 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

9.14 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 10

Logistic Regression

10.1 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.

10.2 Example

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

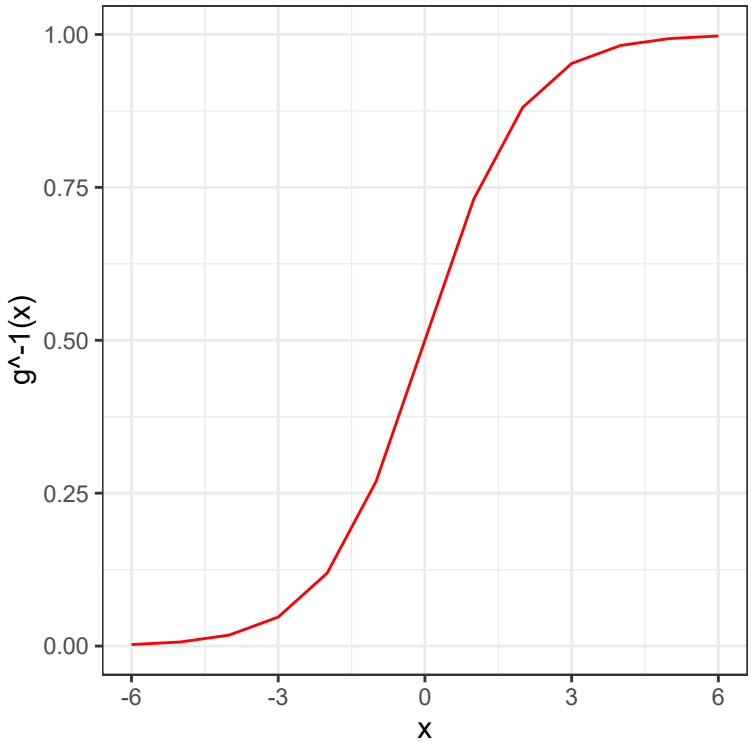


Figure 10.1: Standard Logistic Function

```
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.

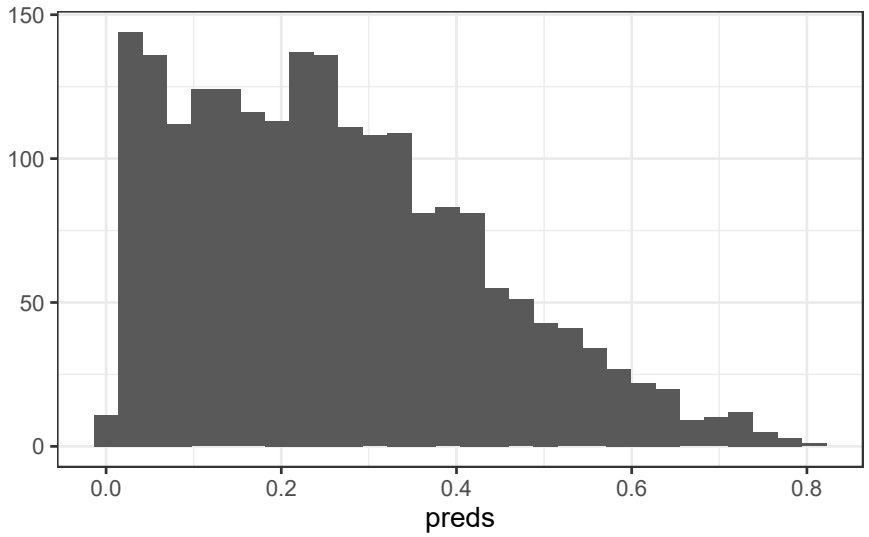


Figure 10.2: Distribution of Predicted Probability

```
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).

10.3 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
##
```

10.3.1 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

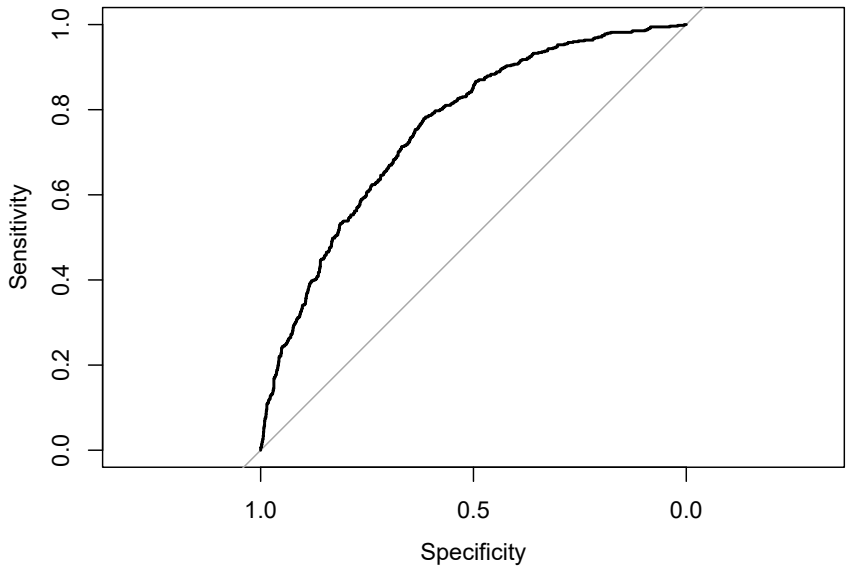


Figure 10.3: AUC for auto_claim

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.

10.3.2 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 11

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.

11.1 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.

11.2 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.

11.3 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.

11.4 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

Chapter 12

Tree-based models

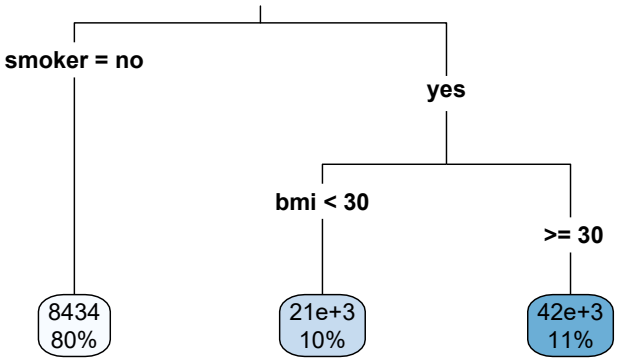
12.1 Decision Trees

12.1.1 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

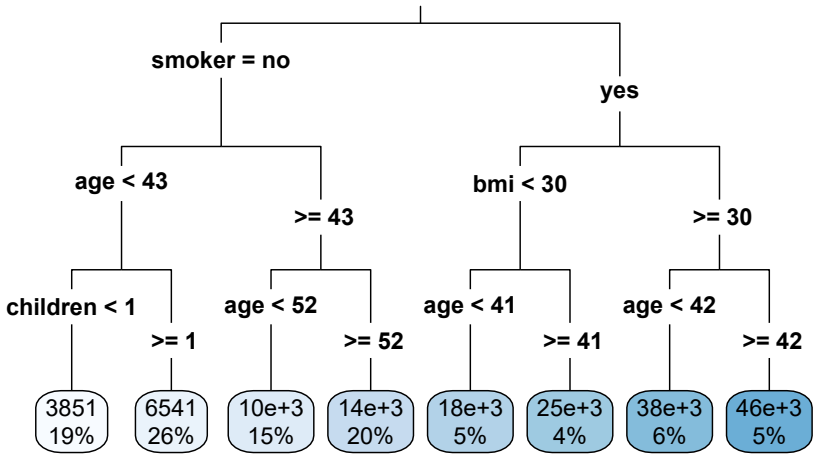
- *Entropy* (aka, information gain)
- *Gini*
- *Classification error*

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`.

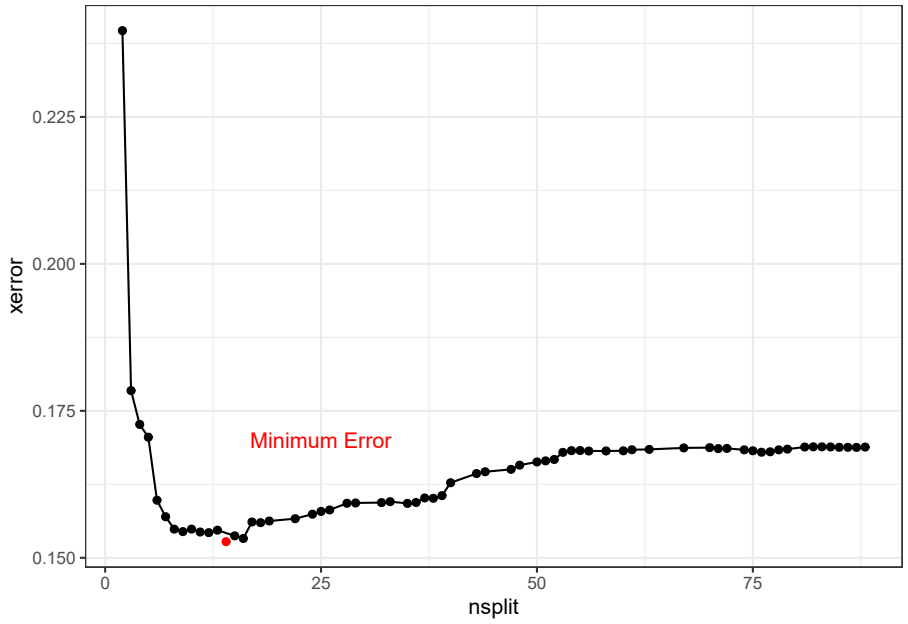
This is stored in the `cptable`

```
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    0.382
```

```
## 3       2 0.0637   0.240
## 4       3 0.00967  0.178
## 5       4 0.00784  0.173
## 6       5 0.00712  0.171
```

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     14 0.00119  0.153
## 2     16 0.00105  0.153
## 3     15 0.00116  0.154
## 4     12 0.00137  0.154
## 5     11 0.00145  0.154
## 6      9 0.00173  0.154
```

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.

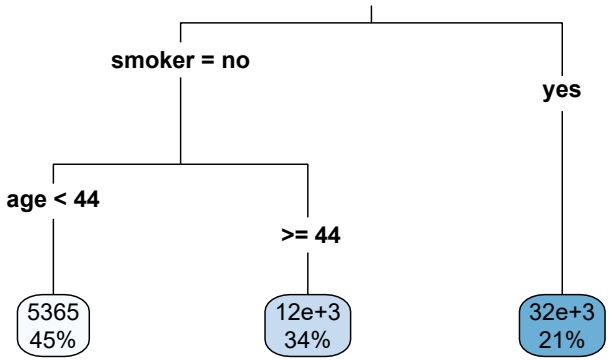
```
pruned_tree <- prune(tree,
                     cp = tree$cptable[which.min(tree$cptable[, "xerror"]), "CP"])
```

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
```

12.1.2 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)

12.2 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*.

12.2.1 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.

12.2.2 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.

12.3 Random Forests

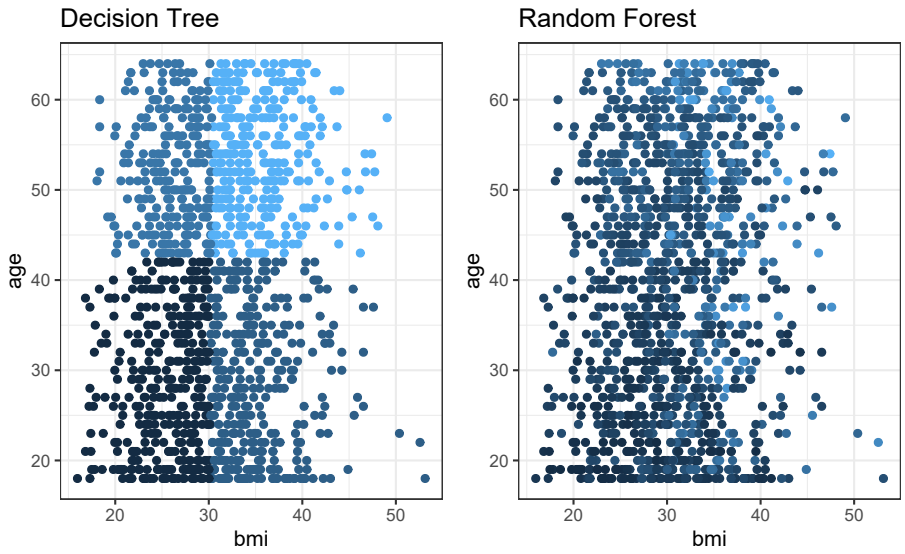
12.3.1 Model form

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.

12.3.2 Example

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

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

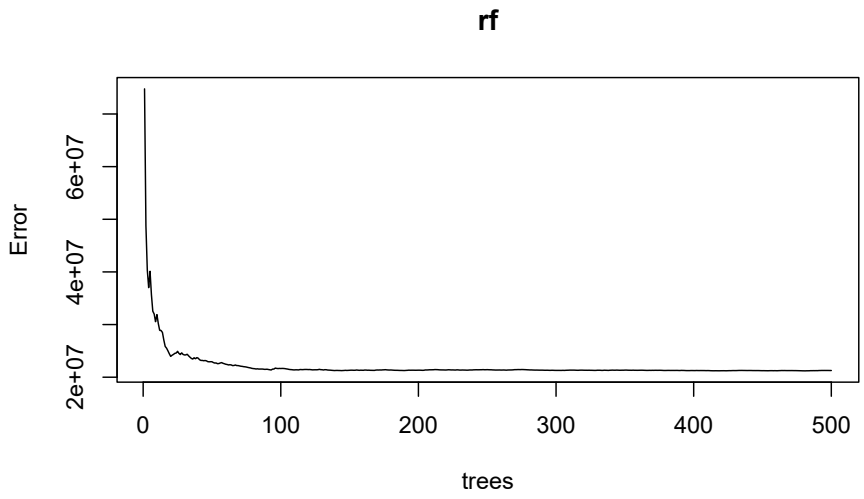
```
rf_data <- health_insurance %>%
  mutate(sex = ifelse(sex == "male", 1, 0),
         smoker = ifelse(smoker == "yes", 1, 0),
         region_ne = ifelse(region == "northeast", 1,0),
         region_nw = ifelse(region == "northwest", 1,0),
         region_se = ifelse(region == "southeast", 1,0),
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

12.3.3 Variable Importance

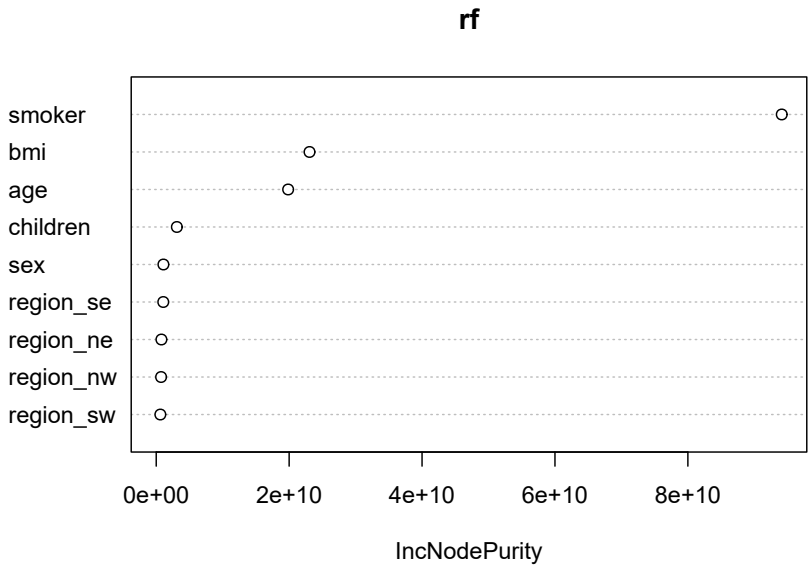
Variable importance is a way of measuring how each variable contributes to overall model. For single decision trees, if a variable was "higher" up in the tree, then this variable would have greater influence. Statistically, there are two ways of measuring this:

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)
```



12.3.4 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)
```

12.3.5 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

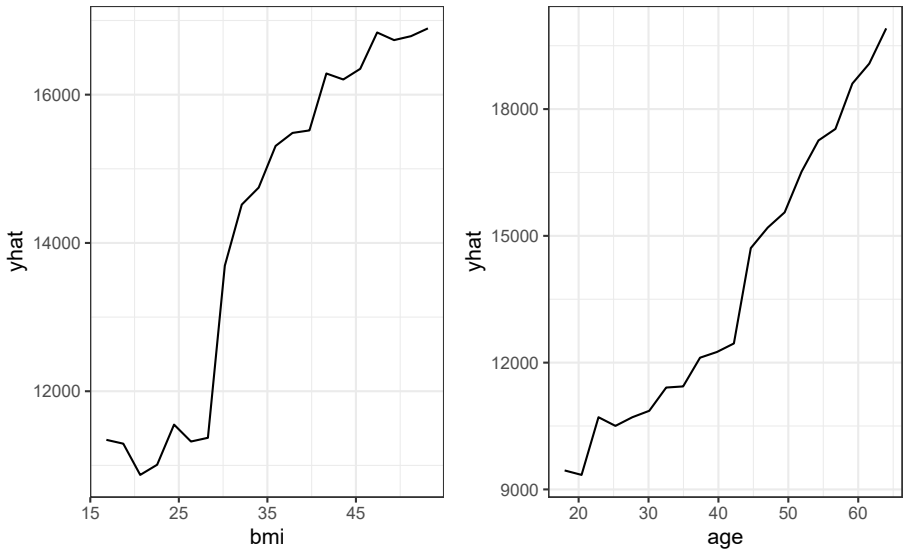


Figure 12.1: Partial Dependence

- 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

12.4 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.

12.4.1 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`.

12.4.2 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
```

12.4.3 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

12.5 Exercises

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

Run this code on your computer to answer these exercises.

12.5.1 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)
```

12.5.2 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)
```

12.5.3 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)

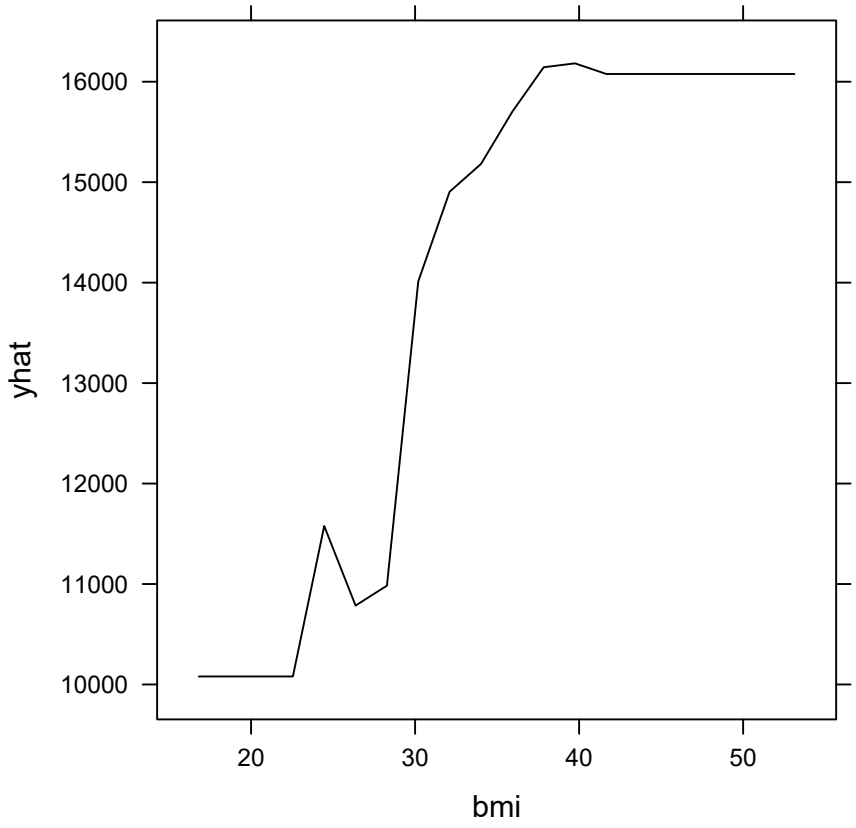
2. Suppose that the optimization measure was RMSE. The below table shows the results from three models. Explain why some sets of parameters have better RMSE than the others.

```
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
```

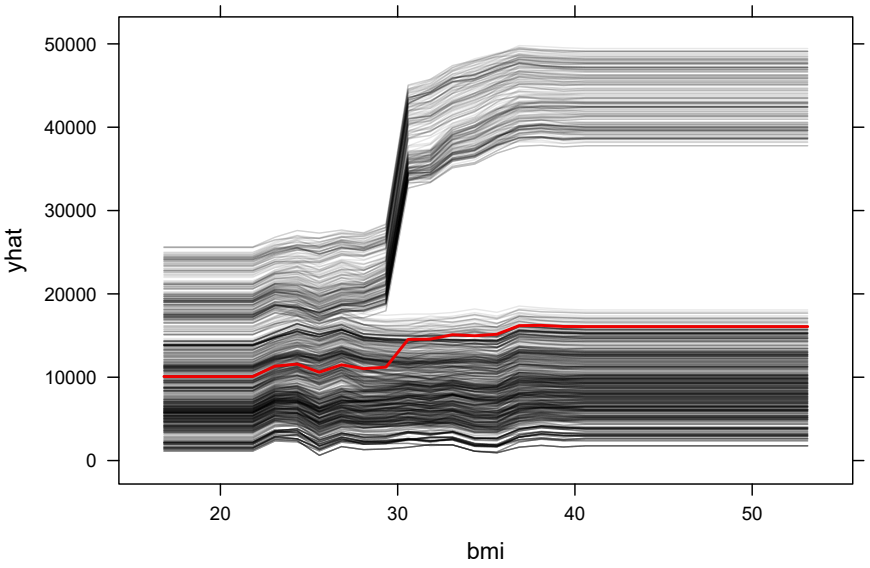
3. The partial dependence of `bmi` onto `charges` makes it appear as if `charges` increases monotonically as `bmi` increases.

```
pdp::partial(gbm, pred.var = "bmi", grid.resolution = 20, 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 = 30, plot = T, ice = T, alpha = 0
```



12.6 Answers to Exercises

Answers to these exercises are available at ExamPA.net.

Chapter 13

Unsupervised Learning

The chapter on unsuporvised learning, topics from module 8, clustering and PCA, are available at ExamPA.net.

Chapter 14

Practice Exams

Practice exams are available at ExamPA.net.

Chapter 15

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

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/