Exam PA Study Manual

Sam Castillo

2019-11-03

Contents

Chapter 1

Welcome

This book covers the source material for the SOA's Predictive Analytics Exam.

Features

- All data sets used are packaged in a single R library
- Reproducible R code
- Explanations of the statistical concepts
- Tips on taking the exam
- Two **original** practice exams (coming soon)

Chapter 2

The Exam

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

https://www.soa.org/globalassets/assets/files/edu/2019/2019-12-exam-pa-syllabus.pdf

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%)

- Build models
- Interpret results within a business context

Chapter 3

Learning

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 and then install ExamPAData from github
if("devtools" %in% installed.packages()){
  library(devtools)
  install_github("https://github.com/sdcastillo/ExamPAData")
} else{
  install.packages("devtools")
  library(devtools)
  install_github("https://github.com/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"
```

```r
class(everything)
```

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

```r
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
```

```r
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

```r
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**.

```
library(tibble) #the tibble library has functions for making tibbles
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`.

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

```
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

1. How many industries are represented?

```
actuary_salaries %>% count(industry)
```

```
## # A tibble: 4 x 2
##   industry     n
##   <chr>    <int>
## 1 Casualty    45
## 2 Health      31
## 3 Life        31
## 4 Pension     31
```

2. The `salary_high` column is a character type when it should be numeric. Change this column to numeric.

```
#method 1
actuary_salaries <- actuary_salaries %>% mutate(salary_high = as.numeric(salary_high))

#method 2
actuary_salaries <- actuary_salaries %>% modify_at("salary_high", as.numeric)
```

3. What are the highest and lowest salaries for an actuary in Health with 5 exams passed?

```
actuary_salaries %>%
  filter(industry == "Health", exams == 5) %>%
  summarise(highest = max(salary_high),
            lowest = min(salary_low))
```

```
## # A tibble: 1 x 2
##   highest lowest
##     <dbl>  <dbl>
## 1     126     68
```

4. Create a new column called `salary_mid` which has the middle of the `salary_low` and `salary_high` columns.

```
actuary_salaries <- actuary_salaries %>%
  mutate(salary_mid = (salary_low + salary_high)/2)
```

5. When grouping by industry, what is the highest `salary_mid`? What about `salary_high`? What is the lowest `salary_low`?

```
actuary_salaries %>%
  group_by(industry) %>%
  summarise(max_salary_mid = max(salary_mid),
            max_salary_high = max(salary_high),
            low_salary_low = min(salary_low))
```

```
## # A tibble: 4 x 4
##   industry max_salary_mid max_salary_high low_salary_low
##   <chr>             <dbl>           <dbl>          <dbl>
## 1 Casualty           302.             447             11
## 2 Health             272.             390             49
## 3 Life               244              364             51
## 4 Pension            224.             329             44
```

6. There is a mistake when `salary_low == 11`. Find and fix this mistake, and then rerun the code from the previous task.

```
actuary_salaries <- actuary_salaries %>%
  mutate(salary_low = ifelse(salary_low == 11, yes = 114, no = salary_low),
         salary_high = ifelse(salary_high == 66, yes = 166, no = salary_high))

#the minimum salary low is now 48
actuary_salaries %>%
```

```
  group_by(industry) %>%
  summarise(max_salary_mid = max(salary_mid),
            max_salary_high = max(salary_high),
            low_salary_low = min(salary_low))
```

```
## # A tibble: 4 x 4
##   industry max_salary_mid max_salary_high low_salary_low
##   <chr>             <dbl>           <dbl>          <dbl>
## 1 Casualty           302.             447             48
## 2 Health             272.             390             49
## 3 Life               244              364             51
## 4 Pension            224.             329             44
```

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

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

```
actuary_salaries <- actuary_salaries %>%
  mutate(n_exams = case_when(exams == "FSA" ~ 10,
                             exams == "ASA" ~ 7,
                             exams == "FCAS" ~ 10,
                             exams == "ACAS" ~ 7,
                             TRUE ~ as.numeric(substr(exams,1,2))))
```

```
## Warning in eval_tidy(pair$rhs, env = default_env): NAs introduced by
## coercion
```

```
actuary_salaries %>% count(n_exams)
```

```
## # A tibble: 8 x 2
##   n_exams     n
##     <dbl> <int>
## 1       1    12
## 2       2    17
```

```
## 3        3    19
## 4        4    21
## 5        5    17
## 6        6     5
## 7        7    29
## 8       10    18
```

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 %>%
  mutate(social_life = n_exams/experience) %>%
  group_by(industry) %>%
  summarise(avg_social_life = mean(social_life))
```

```
## # A tibble: 4 x 2
##   industry avg_social_life
##   <chr>              <dbl>
## 1 Casualty           0.985
## 2 Health             1.06
## 3 Life               1.06
## 4 Pension            1.00
```

```
#this is not REALLY an average as the number of people, or number of actuaries, are not taken int
```

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)

The first step is to create a blank canvas that holds the columns that are needed. Let's say that the goal is to graph `income` and `count`. We put these into a ggplot object called `p`.

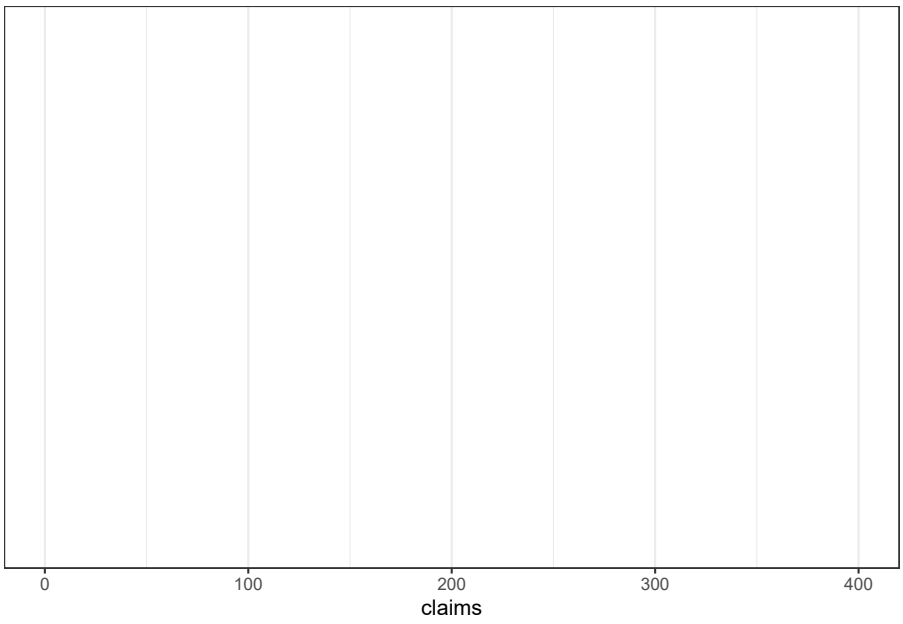
The `aesthetic` argument, `aes`, means that the x-axis will have `income` and the y-axis will have `count`.

```
library(Cairo)
```

```
library(tidyverse)
library(ExamPAData)
theme_set(theme_bw())
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
```

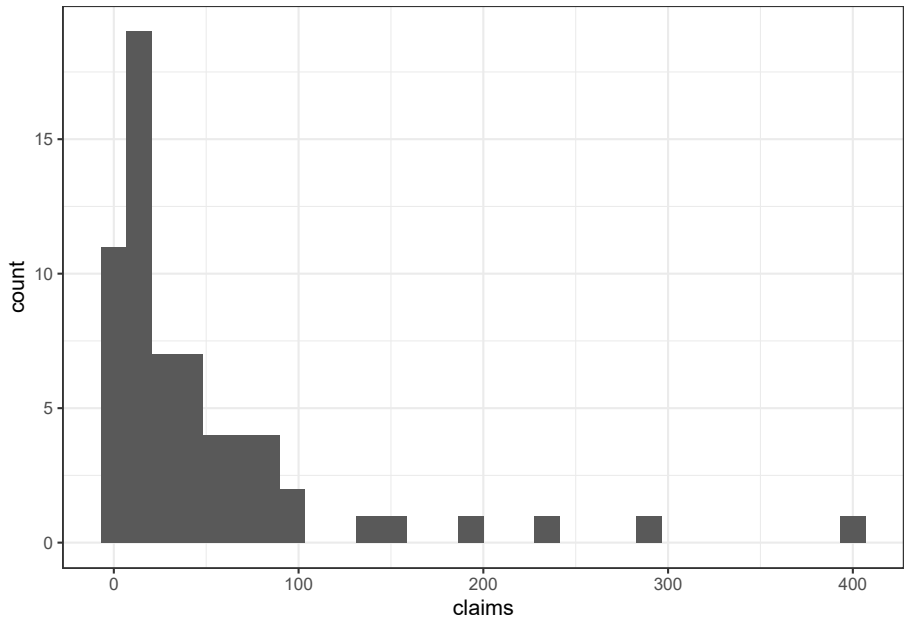


7.2 Add a plot

We add a histogram

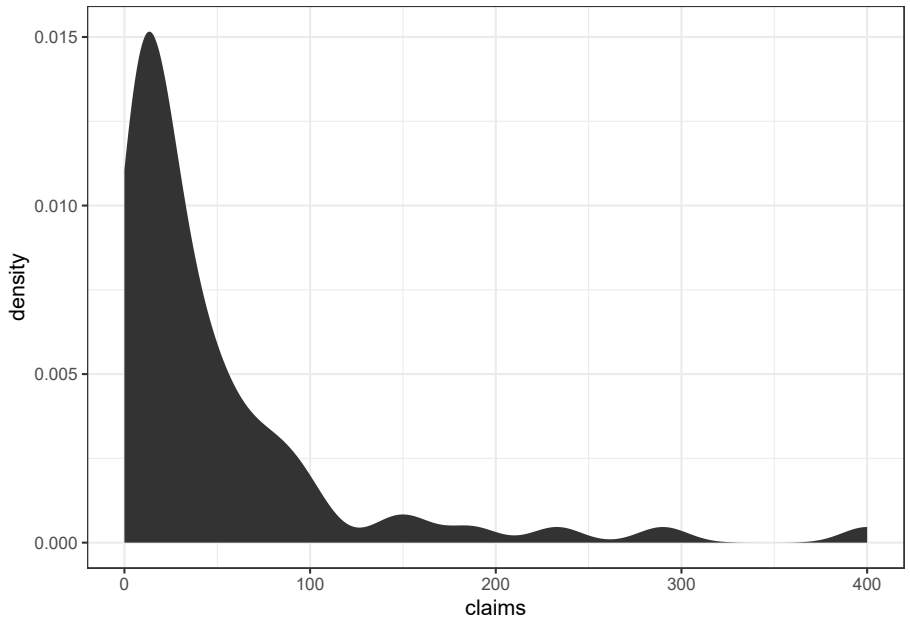
```
p + geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



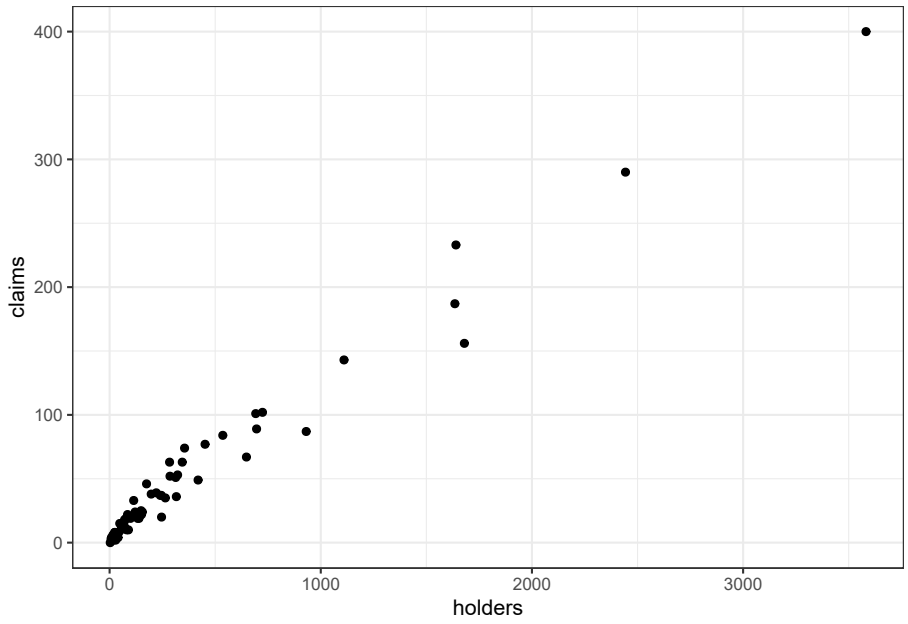
Different plots are called "geoms" for "geometric objects". Geometry = Geo (space) + metre (measure), and graphs measure data. For instance, instead of creating a histogram, we can draw a gamma distribution with `stat_density`.

```
p + stat_density()
```



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

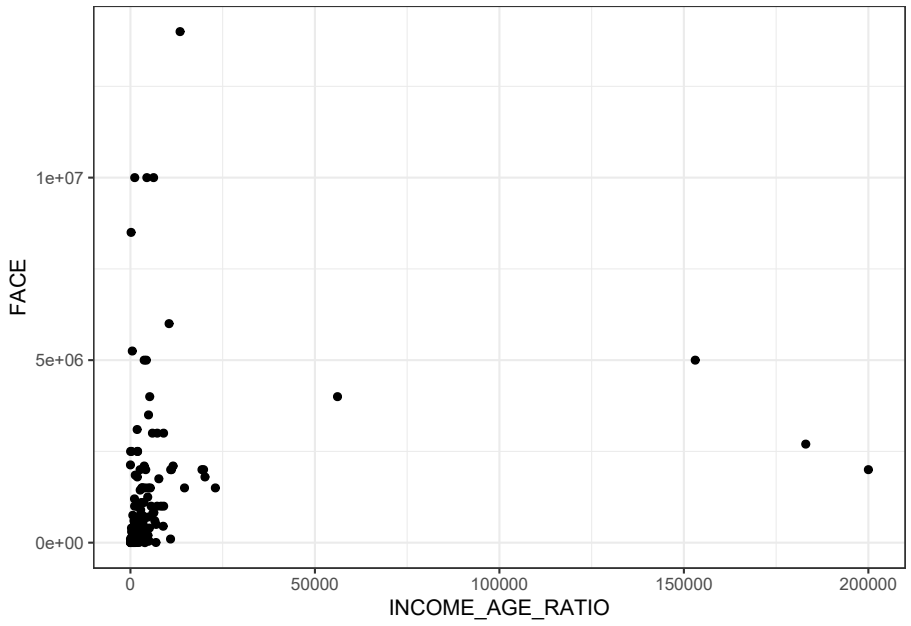
```
insurance %>%
  ggplot(aes(x = holders, y = claims)) +
  geom_point()
```



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()
```




```
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 `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 $\mathbf{X}$ to predict $\mathbf{y}$. We call this "y-hat", or simply the prediction. This is based on a function of the data X.

$$\hat{\mathbf{y}} = f(\mathbf{X}) = \begin{pmatrix} \hat{y_1} \\ \hat{y_2} \\ \hat{y_3} \end{pmatrix}$$

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.

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

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. If f is non-parametric (does not have input parameters), then it is non-parametric modeling. Linear models are linear in the parameters, β.

We have the data $\mathbf{X}$ and the target $\mathbf{y}$, where all of the y's are real numbers, or $y_i \in \mathbb{R}$.

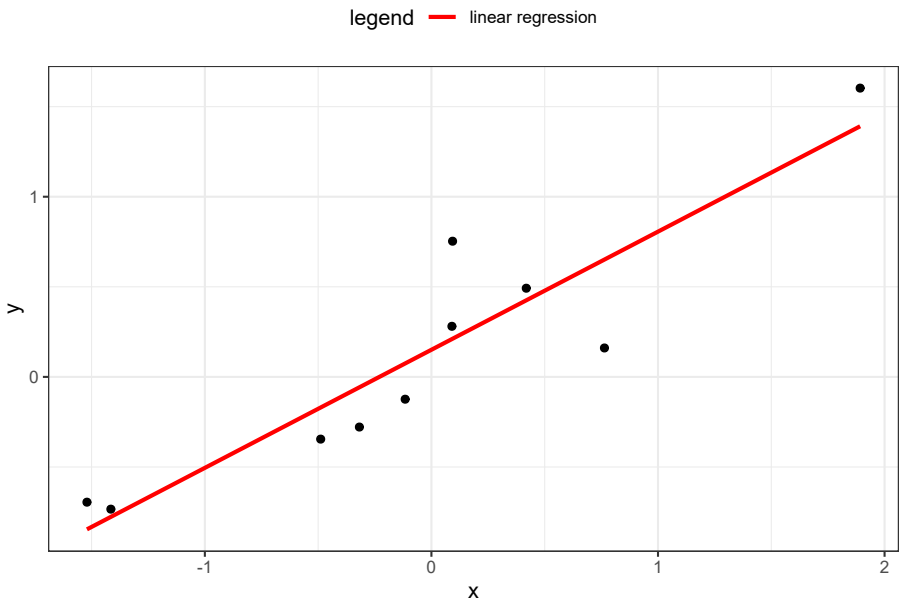
We want to find a β so that

$$\hat{\mathbf{y}} = \mathbf{X}\beta$$

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.

$$y_i = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$$

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 $\mathbf{Y}|\mathbf{X}$ 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 rejected 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 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 a) using a hold out set, or b) 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.

```
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] 11344.01
```

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] 12138.19
```

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.

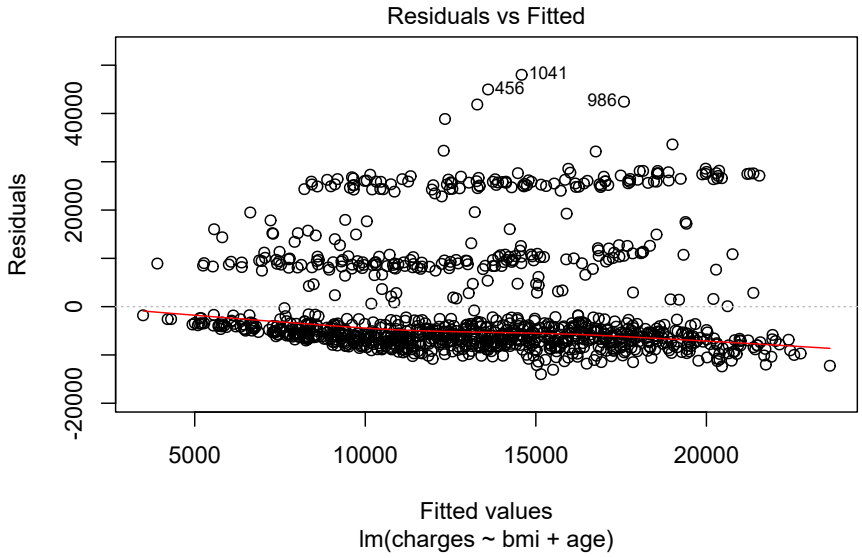


Figure 8.1: Residuals vs. Fitted

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

Caution: This test only applies to linear models which have a gaussian response distribution.

The below is from an excellent post of Stack Exchange.

> R does not have a distinct plot.glm() method. When you fit a model with glm() and run plot(), it calls ?plot.lm, which is appropriate for linear models (i.e., with a normally distributed error term).
>
> More specifically, the plots will often 'look funny' and lead people to believe that there is something wrong with the model when it is perfectly fine. We can see this by looking at those plots with a couple of simple simulations where we know the model is correct:

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.

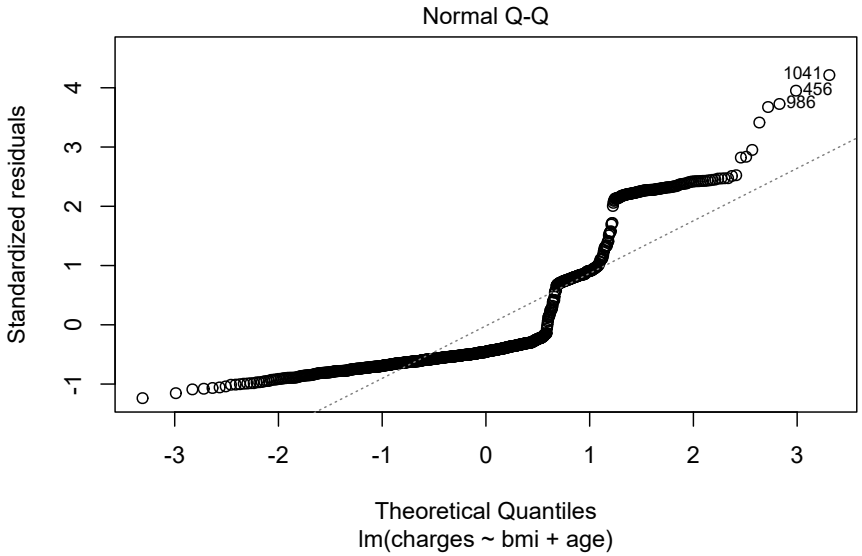


Figure 8.2: Normal Q-Q

```
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	3920.4
bmi	51.4	115.4
age	22.3	48.8

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
##  -5907.4188    313.7926    243.6417
```

Translating the above into an equation we have

$$\hat{y_i} = -6,424.80 + 332.97\text{bmi} + 241.93\text{age}$$

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

$$\hat{y_1} = -6,424.80 + (332.97)(27.9) + (241.93)(19) = 7,461.73$$

Chapter 9

Generalized linear models (GLMs)

9.1 Model form

Instead of the model being a direct linear combination of the variables, there is an intermediate step called a *link function* g.

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

This implies that the response $\mathbf{y}$ is related to the linear predictor $\mathbf{X}\beta$ through the *inverse* link function.

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

This means that $g(.)$ must be an invertable. For example, if g is the natural logarithm (aka, the "log-link"), then

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

This is useful when the distribution of Y is skewed, as taking the log corrects skewness.

You might be asking, what if the distribution of Y is not normal, no matter what choice we have for g? The short answer is that we can change our assumption of the distribution of Y, and use this to change the parameters. If you have taken exam STAM then you are familiar with *maximum likelihood estimation*.

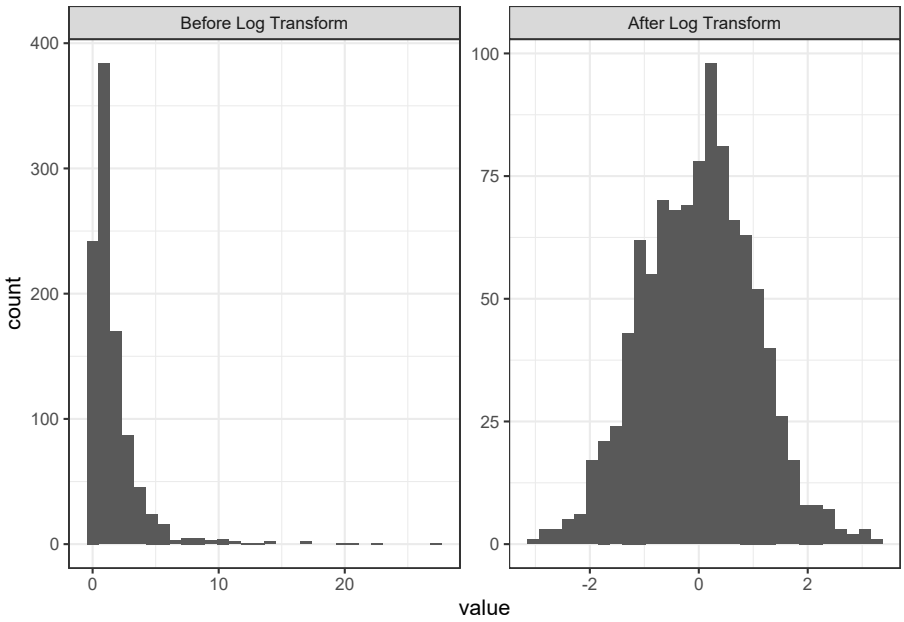


Figure 9.1: Taking the log corrects for skewness

We have a response $\mathbf{Y}$, and we fit a distribution to $\mathbf{Y}|\mathbf{X}$. This is the target variable conditioned on the data. For each y_i, each observation, we assign a probability $f_Y(y_i)$

$$f_y(y_i|X_1 = x_1, X_2 = x_2, ..., X_p = x_p) = Pr(Y = y_i|\mathbf{X})$$

Now, when we choose the response family, we are simply changing f_Y. If we say that the response family is Gaussian, then f has a Gaussian PDF. If we are modeling counts, then f is a Poisson PDF. This only works if f is in the *exponential family* of distributions, which consists of the common names such as Gaussian, Binomial, Gamma, Inverse Gamma, and so forth. Reading the CAS Monograph 5 will provide more detail into this.

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

Figure 9.2: Distribution-Link Function Combinations

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)

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.

9.2 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 + children,
            family = gaussian(link = "log"),
            data = health_insurance)
```

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

- Increase as age increases
- Are higher for men
- Are slightly higher for patients wich children

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

```
## # A tibble: 4 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)   8.55     0.0953      89.7  0.
## 2 age           0.0201   0.00179     11.3  3.12e-28
## 3 sexmale       0.112    0.0459       2.44 1.49e- 2
## 4 children      0.0489   0.0182       2.69 7.29e- 3
```

9.3 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"
```

9.4 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")
```

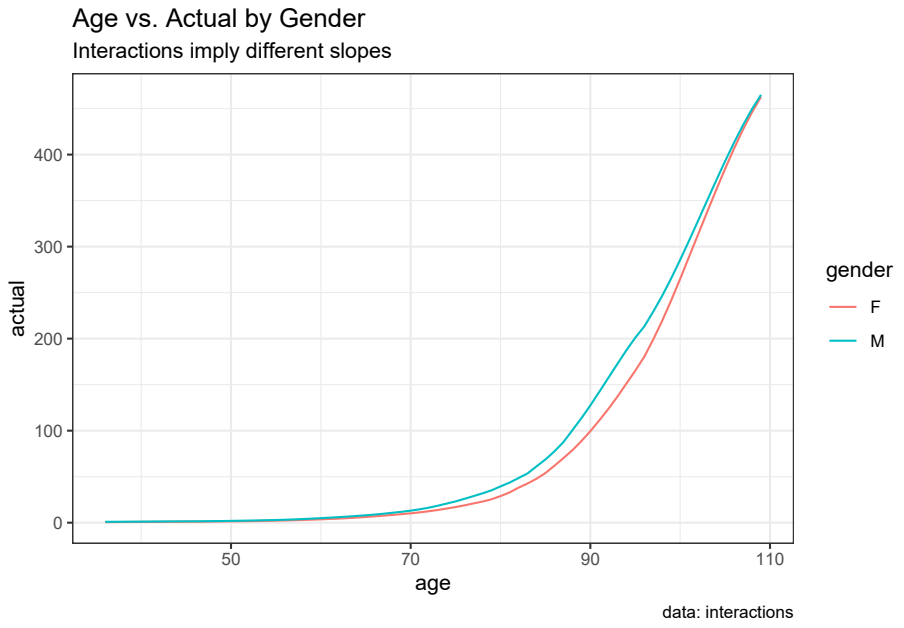


Figure 9.3: Example of weak interaction

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.

9.4.1 Poisson Regression

When the dependent variable is a count, such as the number of claims per month, Poisson regression is appropriate. This requires that each claim is independent

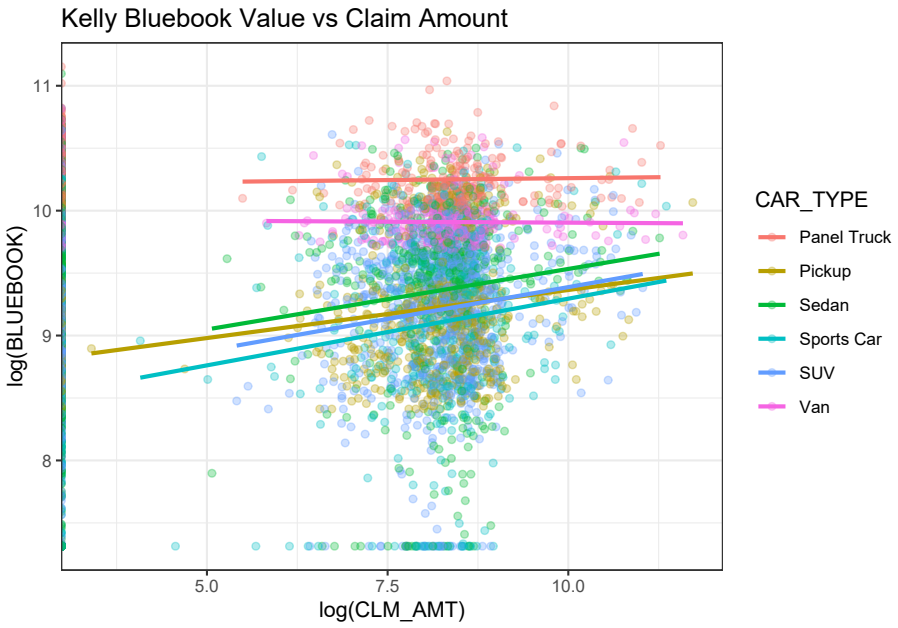


Figure 9.4: Example of strong interaction

in that one claim will not make another claim more or less likely. This means that the target variable is actually a rate, $\frac{\text{claims}}{\text{months}}$. More generally, we call the months the *exposure*.

Let m_i by the units of exposure and y_i the target. We use a log-link function to correct for skewness.

$$log(\frac{\hat{y_i}}{m_i}) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$$

By using the fact that $log(\frac{a}{b}) = log(a) - log(b)$ this turns into

$$log(\hat{y_i}) = log(m_i) + \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$$

We call the $log(m_i)$ the `offset` term. Notice that there is no coefficient (beta) on this value, because we already know what the impact will be.

In R, the code for this equation would be

```
glm(y ~ offset(log(m)) + x, family=poisson(link=log) )
```

9.4.2 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.

9.4.3 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

9.4.4 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
- Works well on small data sets

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*.

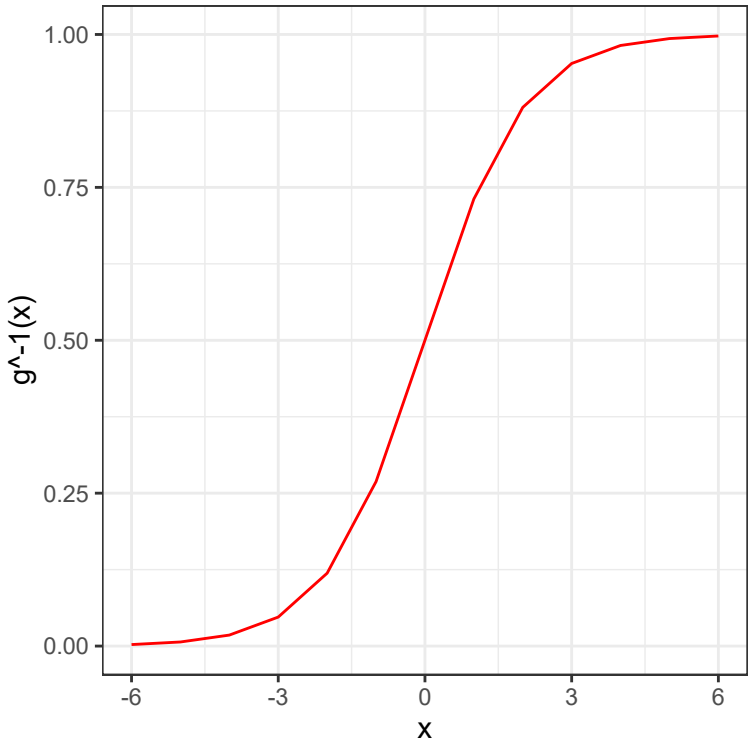


Figure 10.1: Standard Logistic Function

```
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` 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
```

There is no easy way of interpreting the coefficients when using a logit link function. The most inference that we can make is to note which variables are significant.

- `CAR_USE`, `MARRIED`, `BLUEBOOK` are highly 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.

```
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*.

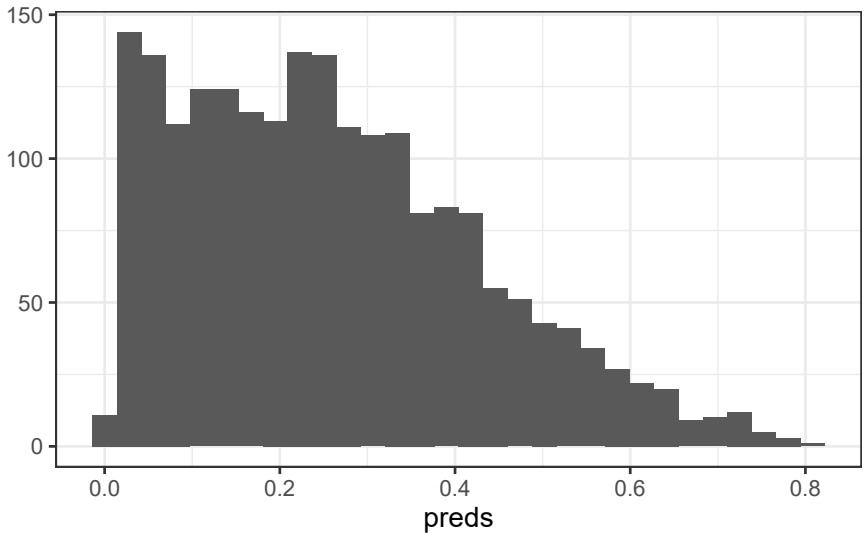


Figure 10.2: Distribution of Predicted Probability

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

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 positive classes (policies with actual claims) vs. the negative 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 Negatives (FN) = 22**
- Had claims but predicted not to - **False Positives (FP) = 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 or 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 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
```

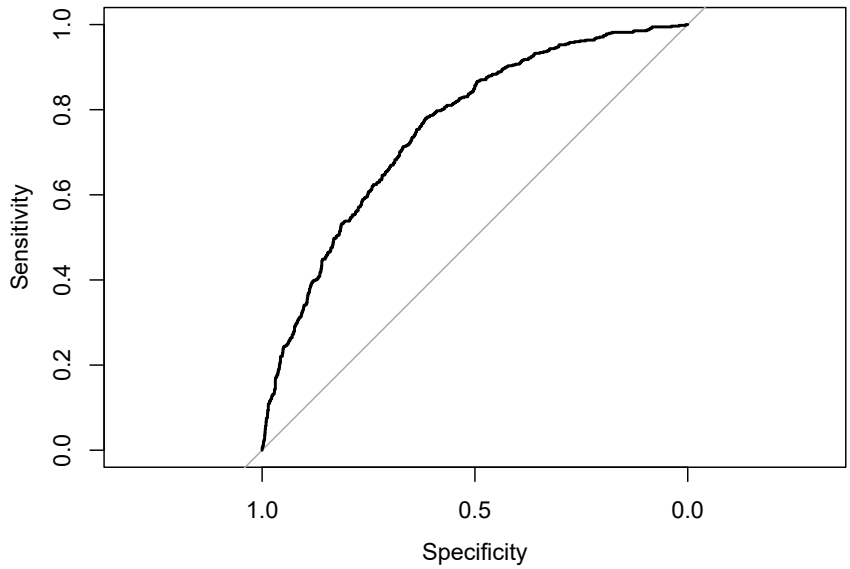


Figure 10.3: AUC for auto_claim

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.

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.

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