ExamPA.net Study Manual

Sam Castillo

Contents

Welcome

This is the study guide for ExamPA.net. While meeting all of the learning requirements of Exam PA, this book gives you data science and machine learning training. You will learn how to get your data into R, clean it, visualize it, and use models to derive business value. Just as a scientist sets up lab experiments to form and test hypothesis, you'll build models and then test them on holdout sets. The statistics is just the first phase, as you'll also learn how to explain the results in easy-to-understand, non-technical business language.

Chapter 1

How to use this book

- Run the examples on your own machine by downloading the ExamPAData library
- Download a PDF by clicking the “Download” button on the top of the page
- Use the arrow keys to navigate the chapters

Contact:

Support: info@exampa.net

Chapter 2

The exam

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

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

A non-technical translation is as follows:

Writing in Microsoft Word (30-40%)

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

Manipulating Data in R (15-25%)

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

Machine learning and statistics (40-50%)

- Interpret results within a business context
- Change model parameters

Follow the SOA's page for the latest updates

https://www.soa.org/education/exam-req/edu-exam-pa-detail/

The exam pass rates are about 50%.

http://www.actuarial-lookup.com/exams/pa

Chapter 3

Prometric Demo

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

SOAFinalCut from Prometric on Vimeo.

Chapter 4

Introduction

While “machine learning” is relatively new, the process of learning itself is not. All of use are already familiar with how to learn - by improving from our mistakes. By repeating what is successful and avoiding what results in failure, we learn by doing, by experience, or trial-and-error. Machines learn in a similar way.

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

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

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

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

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

Chapter 5

Getting started

5.1 Download the data

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

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

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

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

```
##     district        group               age               holders
##  Min.   :1.00   Length:64          Length:64          Min.   :   3.00
##  1st Qu.:1.75   Class :character   Class :character   1st Qu.:  46.75
##  Median :2.50   Mode  :character   Mode  :character   Median : 136.00
##  Mean   :2.50                                         Mean   : 364.98
##  3rd Qu.:3.25                                         3rd Qu.: 327.50
##  Max.   :4.00                                         Max.   :3582.00
##      claims
##  Min.   :  0.00
##  1st Qu.:  9.50
##  Median : 22.00
##  Mean   : 49.23
##  3rd Qu.: 55.50
##  Max.   :400.00
```

5.2 Download ISLR

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

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

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

5.3 New users

Install R:

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

Install RStudio

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

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

Set the R library

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

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

Chapter 6

R programming

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

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

6.1 Notebook chunks

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

```
# This is a chunk
```

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

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

6.2 Basic operations

The usual math operations apply.

```
# Addition
1 + 2
```

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

```
3 - 2
```

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

```
# Multiplication
2 * 2
```

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

```
# Division
4 / 2
```

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

```
# Exponentiation
2^3
```

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

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

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

```
# Variable assignment
y <- 2

# Equality
4 == 2
```

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

```
5 == 5
```

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

```
3.14 > 3
```

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

```
3.14 >= 3
```

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

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

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

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

```
x * y
```

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

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

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

```
z / 2
```

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

```
z + 3
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
bool_true + 1
```

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

Vectors work in the same way.

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

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

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

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

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

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

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

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

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

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

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

```
abc[-2]
```

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

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

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

6.3 Lists

Lists are vectors that can hold mixed object types.

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

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

Lists can be named.

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

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

The $ operator indexes lists.

```
my_list$numeric
```

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

```
my_list$numeric + 5
```

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

Lists can also be indexed using `[[`.

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

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

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

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

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

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

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

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

```
class(x)
```

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

```
class(everything)
```

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

```
str(everything)
```

```
## List of 3
##  $ vector   : num [1:3] 1 2 3
##  $ character: chr [1:3] "a" "b" "c"
##  $ list     :List of 3
##   ..$ bool     : logi TRUE
##   ..$ character: chr "character"
##   ..$ numeric  : num 3.14
```

```
summary(everything)
```

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

6.4 Functions

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

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

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

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

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

greet_me("Future Actuary")
```

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

A function that returns something

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

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

add_together(2, 5)
```

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

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

add_together(2, 5)
```

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

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

```r
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.

```r
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.

```r
sum_multiply <- function(x,y) {
  sum <- x + y
  product <- x * y
  list("Sum" = sum, "Product" = product)
}

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

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

```
result$Product
```

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

6.5 Data frames

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

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

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

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

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

```
## -- Attaching packages --------------------------------------------------------------- tidyvers

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

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

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

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

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

```
df$age
```

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

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

```
dim(df)
```

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

To find a summary, use `summary`

```
summary(df)
```

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

6.6 Pipes

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

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

```r
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`.

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

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

```r
# 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Yes, if you really want to.

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

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

Chapter 7

Data manipulation

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

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

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

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

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

```
library(ExamPAData)
```

7.1 Look at the data

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

The descriptions of the columns are below.

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

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

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

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

Using a pipe is an alternative way of doing this.

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

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

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

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

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

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

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

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

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

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

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

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

With incorrect data types

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

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

With correct data types

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

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

health_insurance %>%
  summary()
```

```
##       age            sex                bmi            children        smoker
##  Min.   :18.00   female:662   Min.   :15.96   Min.   :0.000   no :1064
##  1st Qu.:27.00   male  :676   1st Qu.:26.30   1st Qu.:0.000   yes: 274
##  Median :39.00                Median :30.40   Median :1.000
##  Mean   :39.21                Mean   :30.66   Mean   :1.095
##  3rd Qu.:51.00                3rd Qu.:34.69   3rd Qu.:2.000
##  Max.   :64.00                Max.   :53.13   Max.   :5.000
##         region       charges
##  northeast:324   Min.   : 1122
##  northwest:325   1st Qu.: 4740
##  southeast:364   Median : 9382
##  southwest:325   Mean   :13270
##                  3rd Qu.:16640
##                  Max.   :63770
```

7.2 Transform the data

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

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

R to SQL translation

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

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

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

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

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

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

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

The SQL translation is

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

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

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

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

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

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

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

In SQL, this would be

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

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

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

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

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

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

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

We can create as many new columns as we want.

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

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

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

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

```
## # A tibble: 1,338 x 2
##      age age_bucket
##    <dbl>      <dbl>
##  1    19          0
##  2    18          0
##  3    28          0
##  4    33          0
##  5    32          0
##  6    31          0
##  7    46          0
##  8    37          0
##  9    37          0
## 10    60          1
## # ... with 1,328 more rows
```

SQL translation:

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

7.3 Exercises

Run this code on your computer to answer these exercises.

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

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

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

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

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

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

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

7.4 Answers to exercises

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

Chapter 8

Visualization

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

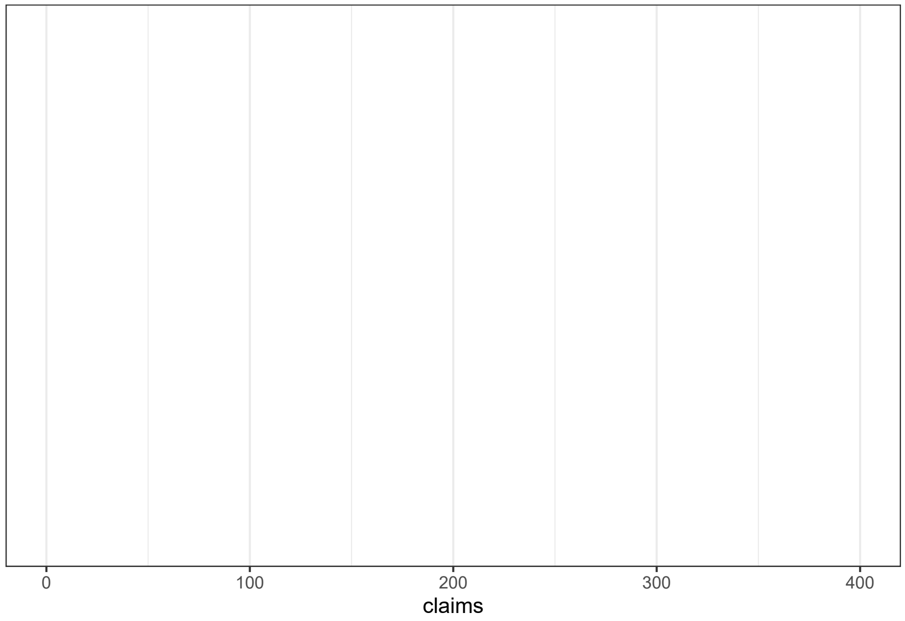
8.1 Create a plot object (ggplot)

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

```r
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`.

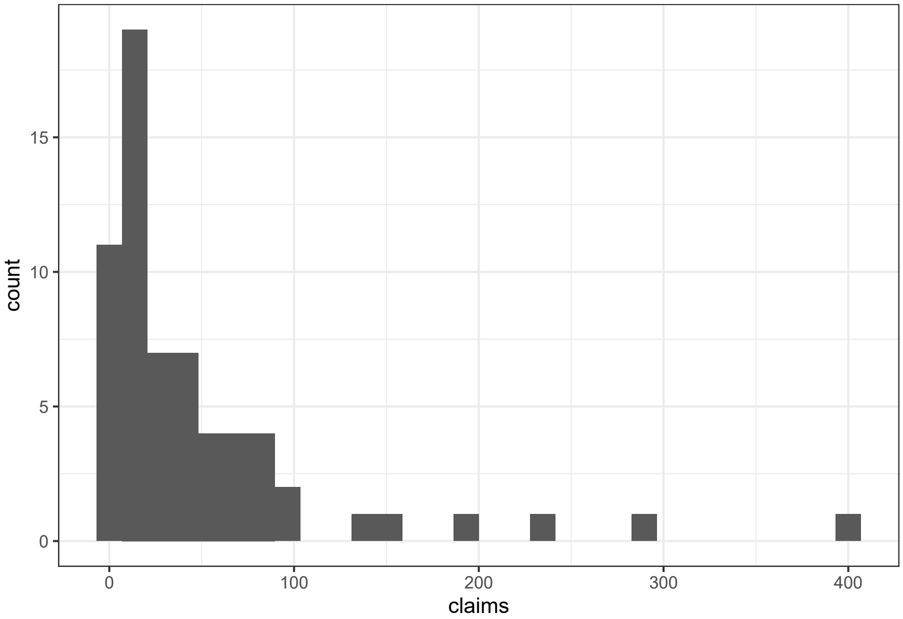
```r
p
```



8.2 Add a plot

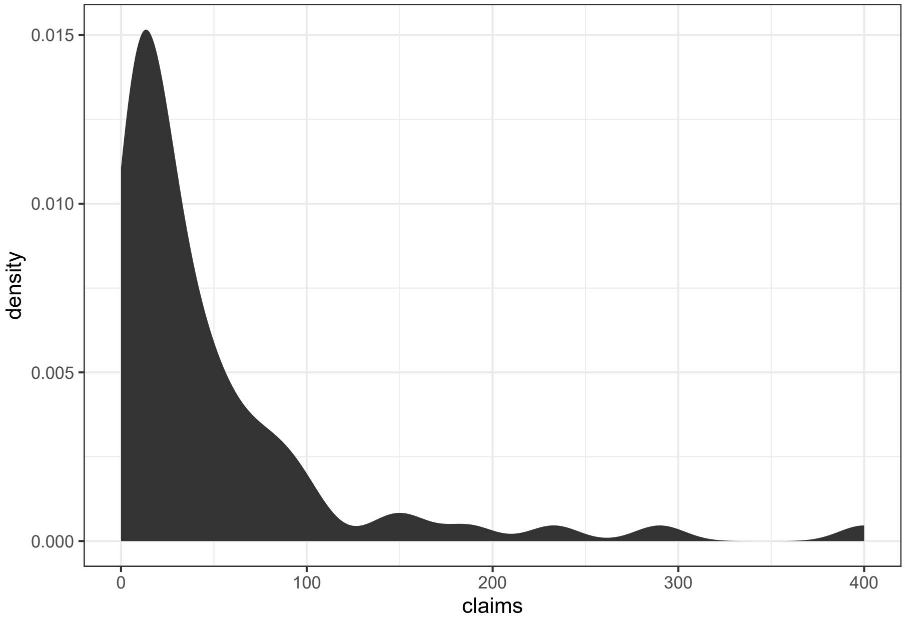
We add a histogram

```
p + geom_histogram()
```



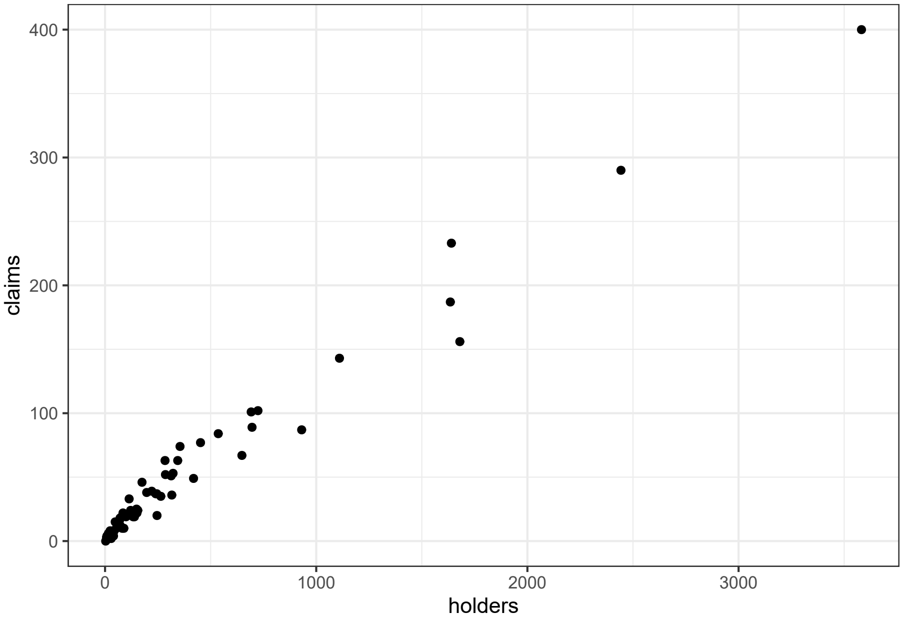
Different plots are called “geoms” for “geometric objects”. Geometry = Geo (space) + metre (measure), and graphs measure data. For instance, instead of creating a histogram, we can draw a gamma distribution with `stat_density`.

```
p + stat_density()
```



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

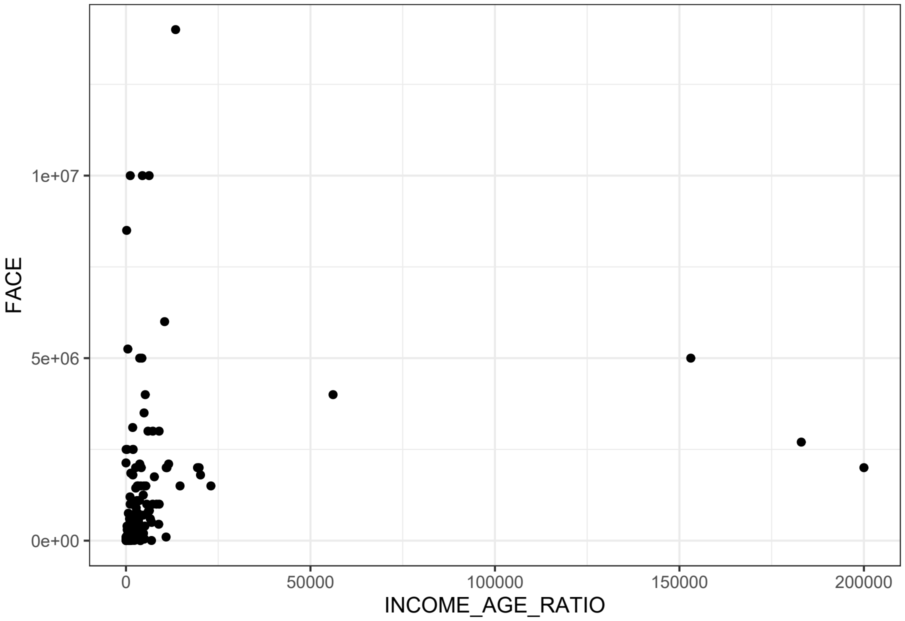
```
insurance %>%
  ggplot(aes(x = holders, y = claims)) +
  geom_point()
```



8.3 Data manipulation chaining

Pipes allow for data manipulations to be chained with visualizations.

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



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

Chapter 9

Introduction to modeling

About 40-50% of the exam grade is based on modeling. The goal is to be able to predict an unknown quantity. In actuarial applications, this tends to be claims that occur in the future, death, injury, accidents, policy lapse, hurricanes, or some other insurable event.

9.1 Modeling vocabulary

Modeling notation is sloppy because there are many words that mean the same thing.

The number of observations will be denoted by n. When we refer to the size of a data set, we are referring to n. Each row of the data is called an *observation* or *record*. Observations tend to be people, cars, buildings, or other insurable things. These are always independent in that they do not influence one another. Because the Prometric computers have limited power, n tends to be less than 100,000.

Each observation has known attributes called *variables*, *features*, or *predictors*. We use p to refer the number of input variables that are used in the model.

The *target*, *response*, *label*, *dependant variable*, or *outcome* variable is the unknown quantity that is being predicted. We use Y for this. This can be either a whole number, in which case we are performing *regression*, or a category, in which case we are performing *classification*.

For example, say that you are a health insurance company that wants to set the premiums for a group of people. The premiums for people who are likely to incur high health costs need to be higher than those who are likely to be low-cost. Older people tend to use more of their health benefits than younger people, but there are always exceptions for those who are very physically active and healthy. Those who have an unhealthy Body Mass Index (BMI) tend to have higher costs than those who have a healthy BMI, but this has less of an impact on younger people. **In short, we want to be able to predict a person's future health costs by taking into account many of their attributes at once**.

This can be done in the `health_insurance` data by fitting a model to predict the annual health costs of a person. The target variable is `y = charges`, and the predictor variables are `age`, `sex`, `bmi`, `children`, `smoker` and `region`. These six variables mean that $p = 6$. The data is collected from 1,338 patients, which means that $n = 1,338$.

9.2 Modeling notation

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 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 "3-by-2") matrix.

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

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.

Modeling 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).

Another way of saying this is in terms of expected value: the model $f(X)$ estimates the expected value of the target $E[Y|X]$. That is, once we condition on the data X, we can make a guess as to what we expect Y to be "close to". There are many ways of measuring "closeness", as we will see.

9.3 Ordinary Least Squares (OLS)

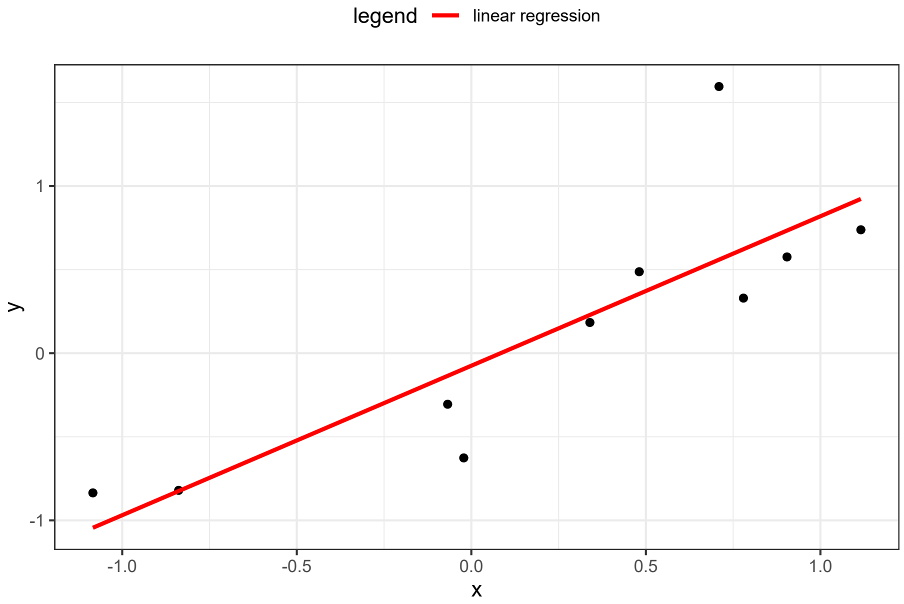
Also known as *simple linear regression*, OLS predicts the target as a weighted sum of the variables.

We find a β so that

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

Each y_i is a *linear combination* of $x_{i1}, ..., x_{ip}$, 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 red line shows the *expected value* of the target, as the target $\hat{Y}$ is actually a random variable. For each of the data points, the model assumes a Gaussian distribution. If there is just a single predictor, x, then the mean is $\beta_0 + \beta_1 x$.

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 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 but the betas would not be the maximum likelihood parameters. In fact, using the absolute error results in the model predicting the *median* as opposed to the *mean*. Two reasons why RSS is popular are:

- 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.
- It is computationally easier, and computers used to have a difficult time optimizing for MAE

> What does it mean when a log transform is applied to Y? I remember from my statistics course on regression that this was done.

This is done so that the variance is closer to being constant. For example, if the units are in dollars, then it is very common for the values to fluctuate more for higher values than for lower values. Consider a stock price, for instance. If the stock is $50 per share, then it will go up or down less than if it is $1000 per share. The log of 50, however, is about 3.9 and the log of 1000 is only 6.9, and so this difference is smaller. In other words, the variance is smaller.

Transforming the target means that instead of the model predicting $E[Y]$, it predicts $E[log(Y)]$. A common mistake is to then the take the exponent in an attempt to “undo” this transform, but $e^{E[log(Y)]}$ is not the same as $E[Y]$.

9.4 Regression vs. classification

Regression modeling is when the target is a number. Binary classification is when there are two outcomes, such as “Yes/No”, “True/False”, or “0/1”. Multiclass regression is when there are more than two categories such as “Red, Yellow, Green” or “A, B, C, D, E”. There are many other types of regression that are not covered on this exam such as ordinal regression, where the outcome is an ordered category, or time-series regression, where the data is time-dependent.

9.5 Regression metrics

For any model, the goal is always to reduce an error metric. This is a way of measuring how well the model can explain the target. The phrases “reducing error”, “improving performance”, or “making a better fit” are synonymous with reducing the error. The word “better” means “lower error” and “worse” means “higher error”.

The choice of error metric has a big difference on the outcome. When explaining a model to a businessperson, using simpler metrics such as R-Squared and Accuracy is convenient. When training the model, however, using a more nuanced metric is almost always better.

These are the regression metrics that are most likely to appear on Exam PA. Memorizing these formulas for AIC and BIC is not necessary as they are in the R documentation by typing `?AIC` or `?BIC` into the R console.

9.6 Example

In our health insurance data, we can predict a person's health costs based on their age, body mass index, and gender. Intuitively, we expect that these costs would increase as a person's age increases, would be different for men than for women, and would be higher for those who have a less healthy BMI. We 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

> While you can use formulas to create new variables, the exam questions tend to have you do this in the data itself. For example, if taking the log transform of a bmi, you would add a column log_bmi to the data and remove the original bmi column.

Below we fit a simple linear model to predict charges.

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

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

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 p value tells you how "big" this error really is based on standard deviations. A small p-value (Pr (>|t|))) means that we can safely reject the null hypothesis that says the cofficient is equal to zero.

The little *, **, *** tell you the significance level. A variable with a *** means that the probability of getting a coefficient of that size given that the data was randomly generated is less than 0.001. The ** has a significance level of 0.01, and * of 0.05.

```
summary(model)
```

```
##
## Call:
## lm(formula = charges ~ bmi + age + sex, data = health_insurance)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -14974  -7073  -5072   6953  47348
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6986.82    1761.04  -3.967 7.65e-05 ***
## bmi           327.54      51.37   6.377 2.49e-10 ***
## age           243.19      22.28  10.917  < 2e-16 ***
## sexmale      1344.46     622.66   2.159    0.031 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11370 on 1334 degrees of freedom
## Multiple R-squared:  0.1203, Adjusted R-squared:  0.1183
## F-statistic: 60.78 on 3 and 1334 DF,  p-value: < 2.2e-16
```

> For this exam, variable selection tends to be based on the 0.05 significance level (single star *).

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
```

And the Mean Absolute Error as well.

```
get_mae <- function(y, y_hat){
  sqrt(mean(abs(y - y_hat)))
}

get_mae(pred, test$charges)
```

```
## [1] 94.32336
```

The above metrics do not tell us if this is a good model or not by themselves. 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`, which is about $13,000.

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

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

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

```
## [1] 96.63604
```

The RMSE and MAE are both **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.

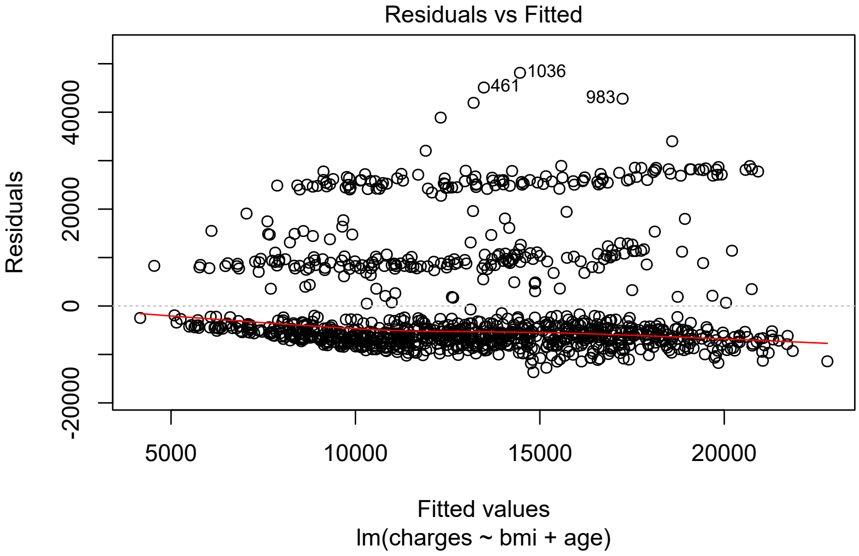


Figure 9.1: Residuals vs. Fitted

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, we are going to continue using it.

```
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.

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

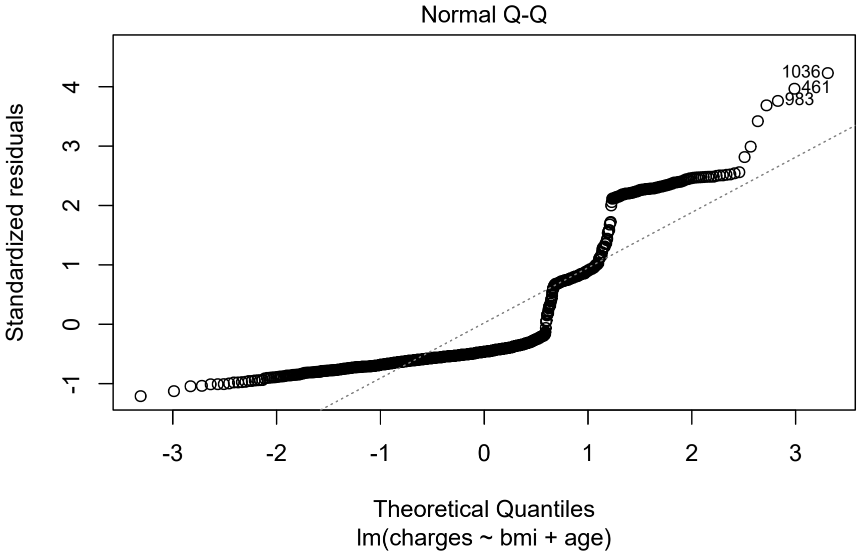


Figure 9.2: Normal Q-Q

```
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 probably not make a difference.

```
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

$$\hat{y}_1 = 4,526 + (287)(27.9) + (228)(19) = 16,865$$

Chapter 10

Generalized linear Models (GLMs)

GLMs are a broad category of models. Ordinary Least Squares and Logistic Regression are both examples of GLMs.

10.0.1 Assumptions of OLS

We assume that the target is Gaussian with mean equal to the linear predictor. This can be broken down into two parts:

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

This says that each observation follows a normal distribution which has a mean that is equal to the linear predictor. 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

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

10.0.2 Assumptions of GLMs

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 $\mu = E[Y|X]$.

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

10.1 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

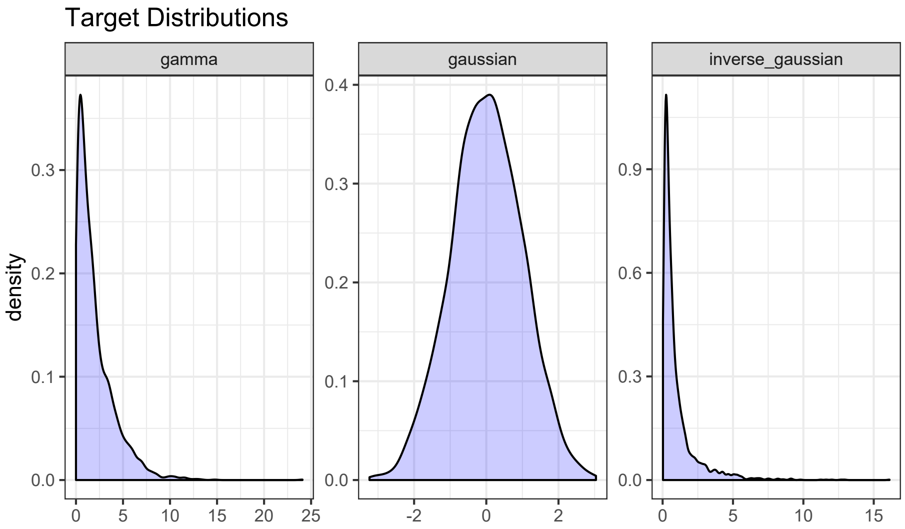
- Easy to interpret
- Can easily be deployed in spreadsheet format
- Handles different response/target distributions
- Is commonly used in insurance ratemaking

GLM Disadvantages

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

10.2 GLMs for regression

For regression problems, we try to match the actual distribution to the model's distribution being used in the GLM. These are the most likely distributions.



The choice of target distribution should be similar to the actual distribution of Y. For instance, if Y is never less than zero, then using the Gaussian distribution is not ideal because this can allow for negative values. If the distribution is right-skewed, then the Gamma or Inverse Gaussian may be appropriate because they are also right-skewd.

There are five link functions for a continuous Y, although the choice of distribution family will typically rule-out several of these immediately. The linear predictor (a.k.a., the *systemic component*) is z and the link function is how this connects to the expected value of the resonse.

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

If the target distribution *must* have a positive mean, such as in the case of the Inverse Gaussian or Gamma, then the Identity or Inverse links are poor choices because they allow for negative values; the range of the mean is $(-\infty, \infty)$. The other link functions force the mean to be positive.