The Predictive Analytics R Study Manual

Sam Castillo

2019-10-29

Contents

1 Welcome **5**

2 The Exam **7**

3 Preface - What is Machine Learning? **9**

4 Getting started **11**

4.1 Download ISLR 11

4.2 Installing R 11

4.3 Installing RStudio 11

4.4 Set the R library 12

4.5 Download the data 12

5 R programming **15**

5.1 Notebook chunks 15

5.2 Basic operations 15

5.3 Lists 19

5.4 Functions 21

5.5 Data frames 22

5.6 Pipes 23

6 Data manipulation **27**

6.1 Look at the data 28

6.2 Transform the data 31

6.3 Exercises 35
6.4 Answers to exercises 36

7 Visualization **41**
7.1 Step 1. Put the data in a pivotable format 41
7.2 Step 2. Create a plot object (ggplot) 42
7.3 Step 3: Add a plot 43

8 Tree-based models **47**
8.1 Decision Trees 47
8.2 Advantages and disadvantages 51
8.3 Random Forests 52
8.4 Gradient Boosted Trees 53

9 A Mini-Exam Example **55**
9.1 Project Statement 55

10 Practice Exam **59**

11 Prior Exams **61**

Chapter 1

Welcome

This book prepares you for the SOA's Predictive Analytics (PA) Exam. Many candidates start with prior knowledge about parts of this exam. Very few are learning all of these topics for the first time. **This book allows you to skip the redundant sections and just focus on the new material.** If you are new to this whole business of machine learning and R programming, great! Every page will be useful.

Features

- All data sets used are packaged in a single R library
- Clean, easy-to-read, efficient R code
- Explanations of the statistical concepts
- Tips on taking the exam
- Two **original** practice exams

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)

Making decisions based on machine learning and statistics knowledge (40-50%)

- Understand several algorithms from a high level and be able to interpret and explain results in english
- Read R documentation about models and use this to make decisions

Chapter 3

Preface - What is Machine Learning?

All of use are already familiar with how to learn - by learning from our mistakes. By repeating what is successful and avoiding what results in failure, we learn by doing, by experience, or trial-and-error. Some study methods work well, but other methods do not. We all know that memorizing answers without understanding concepts is an ineffective method, and that doing many practice problems is better than doing only a few. These ideas apply to how computers learn as much as they do to how humans learn.

Take the example of preparing for an actuarial exam. 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, the percentage of incorrect problems. Later on, we will see how choosing the objective function changes how models are fit.

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. The SOA suggests 100 hours per hour of exam, which means that actuaries train on hundreds of problems before the real exam. We don't have access to the questions that will be on the exam ahead of time, and so this represents "validation" or "holdout" data. In the chapter on cross-validation, we will see how computers use hold-out sets to test model performance.

The more practice problems that we do, the larger the training data set, and the better our "mental model" becomes. When we see new problems, ones which have not appeared in the practice exams, we often have a difficult time. Problems which we have seen before are easier, and we have more confidence in our answers. Statistics tells us that as the sample size increases, model

performance tends to increase. More difficult concepts tend to require more practice, and more complex machine learning problems require more data.

We typically save time by only doing odd-numbered problems. This insures that we still get the same proportion of each type of question while doing fewer problems. If we are unsure of a question, we will often seek a second opinion, or ask an online forum for help. Later on, we will see how “down-sampling”, “bagging”, and “boosting” are all similar concepts.

Chapter 4

Getting started

This book is designed to get you set up within an hour. If this is your first time using R, then you will need to install two pieces of software, R and RStudio. Once these four steps have been done you should not need to download anything else needed for this book. You will be able to work offline.

4.1 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/

4.2 Installing R

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

4.3 Installing 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/

4.4 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")
```

4.5 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
```

Chapter 5

R programming

There are already many great R tutorials available. To save time, this book will cover the absolute essentials. The book "R for Data Science" provides one such introduction.

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

5.1 Notebook chunks

On the Exam, you will start with an .Rmd (R Markdown) template. The way that this book writes code is in the R Notebook. Learning markdown is useful for other web development and documentation tasks as well.

Code is organized into chunks. 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). In R, the 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")
```

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"
```

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
```

Vectors work in the same way.

```
bool_vect <- c(T,T, F)
sum(bool_vect)
```

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

Vectors are indexed using `[]`.

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

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

```
abc[2]
```

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

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

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

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

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

```
abc[-2]
```

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

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

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

5.3 Lists

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

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

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

Lists can be named.

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

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

The $ operator indexes lists.

```
ls$numeric
```

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

```
ls$numeric + 5
```

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

Lists can also be indexed using `[]`.

```
ls[1]
```

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

```
ls[2]
```

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

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

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

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

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

```
class(x)
```

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

```
class(everything)
```

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

```
str(everything)
```

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

```
summary(everything)
```

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

5.4 Functions

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

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

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

A function that does not return anything

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

greet_me("Future Actuary")
```

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

A function that returns something

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

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

add_together(2,5)
```

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

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

add_together(2,5)
```

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

Functions can work with vectors.

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

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

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 more readable and easier to edit. The way that we are taught to do functional composition is by nesting, which is slow to read and write.

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

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

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

Did you get the answer of 1? If so, you are good at reading parenthesis. This 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
```

You may not be convinced by this simple example using numbers; however, once we get to data manipulations in the next section the advantage of piping will become obvious.

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
```

Those familiar with Python's Pandas will be see that `%>%` is quite similar to ".".

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.

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. The technical name for this is a "data dictionary", and one will be provided to you on the real exam.

- `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. Because this is more time-efficient, this will be the preferred method in this manual.

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

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 `cut` and `color` 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 all words for the task of changing the values in a data frame. For those Excel users out there, this can be interpreted as "what Pivot tables, VLOOKUP, INDEX/MATCH, and SUMIFF do".

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

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

The creator's of the exam do not expect candidates to be expert data scientists. Being able to create basic one-dimensional graphs and to interpret results is most important.

Creating graphs in R is easy. The most popular way to do this is with the `ggplot` library.

Three-Steps:

7.1 Step 1. Put the data in a pivotable format

Excel users will know this as "pivot-table format", or the way that a table is organized so it can be put into a pivot table. This is also known as "tidy format". There is one-row per record and one column per variable.

7.1.1 Example of "wide" or "matrix" format

The data below contains counts on a survey which asked people about their religion and annual income.

- `religion` is stored in the rows
- `income` is spread across the columns

This is difficult to work with because there are a lot of columns.

```
## # A tibble: 6 x 11
##   religion `<$10k` `$10-20k` `$20-30k` `$30-40k` `$40-50k` `$50-75k`
```

```
##   <chr>      <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 Agnostic      27        34        60        81        76       137
## 2 Atheist       12        27        37        52        35        70
## 3 Buddhist      27        21        30        34        33        58
## 4 Catholic     418       617       732       670       638      1116
## 5 Don't k~      15        14        15        11        10        35
## 6 Evangel~     575       869      1064       982       881      1486
## # ... with 4 more variables: `$75-100k` <dbl>, `$100-150k` <dbl>,
## #   `>150k` <dbl>, `Don't know/refused` <dbl>
```

7.1.2 Example of "pivotable", "long", or "tidy" format

Here is the same data only in a long format.

You don't need to know how to switch between the two for now, but only that the long format is what is needed to create graphs.

7.2 Step 2. Create a plot object (ggplot)

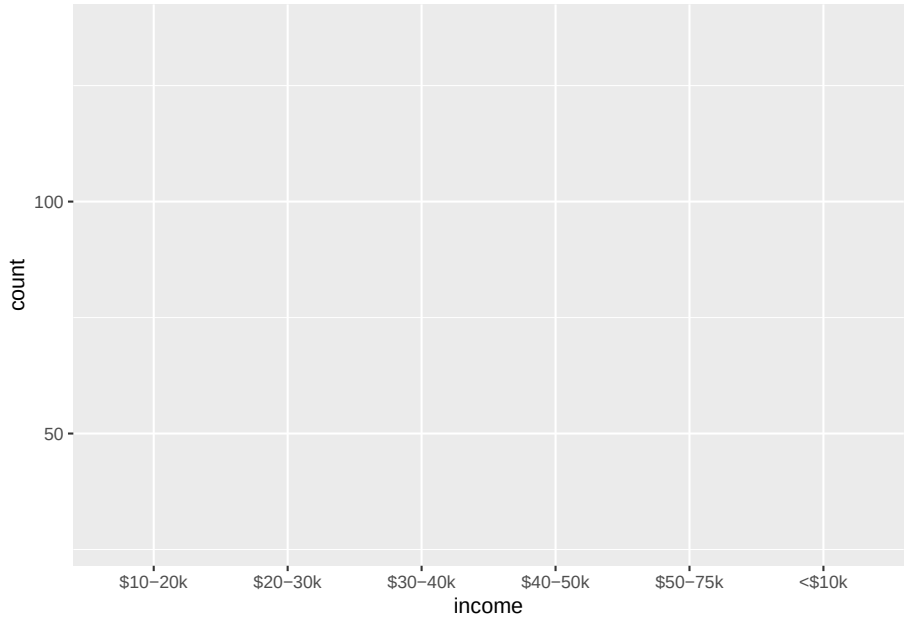
The first step is to create a blank canvas that holds the columns that are needed. Let's say that the goal is to graph `income` and `count`. We put these into a ggplot object called `p`.

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

```
p <- data %>% ggplot(aes(x = income, y = count))
```

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

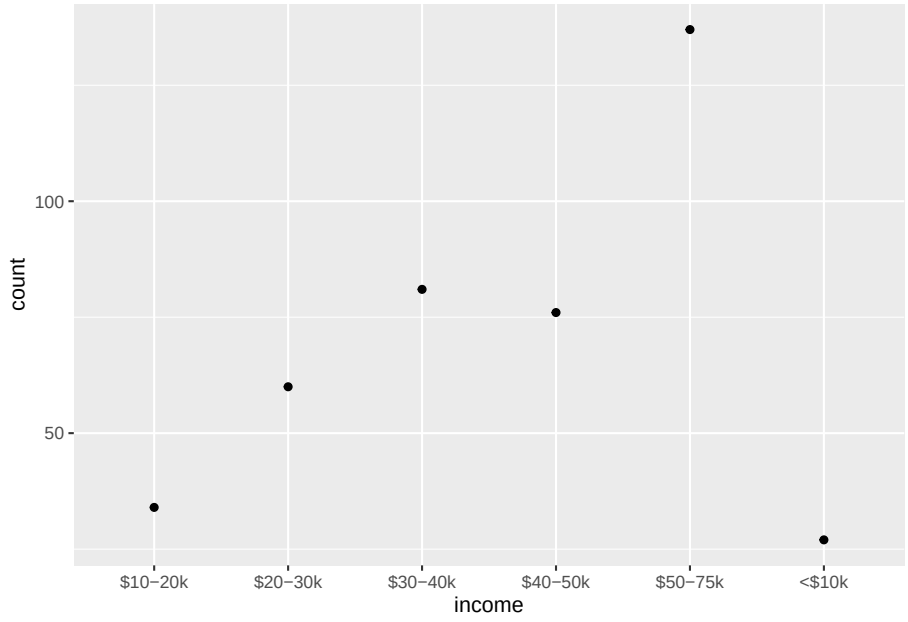
```
p
```



7.3 Step 3: Add a plot

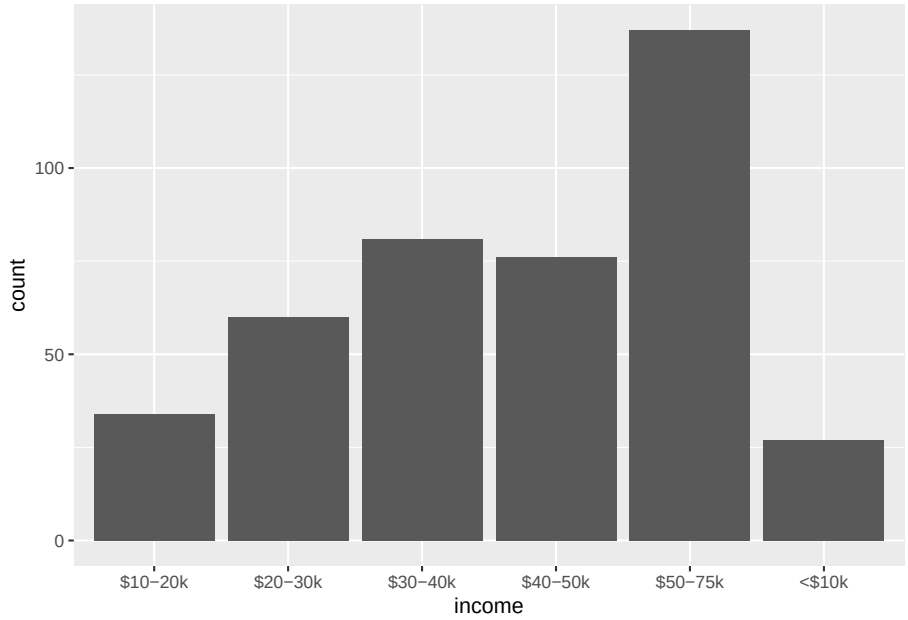
We add an xy plot.

```
p + geom_point()
```



We can also create a bar plot.

```
p + geom_bar(stat = "identity")
```



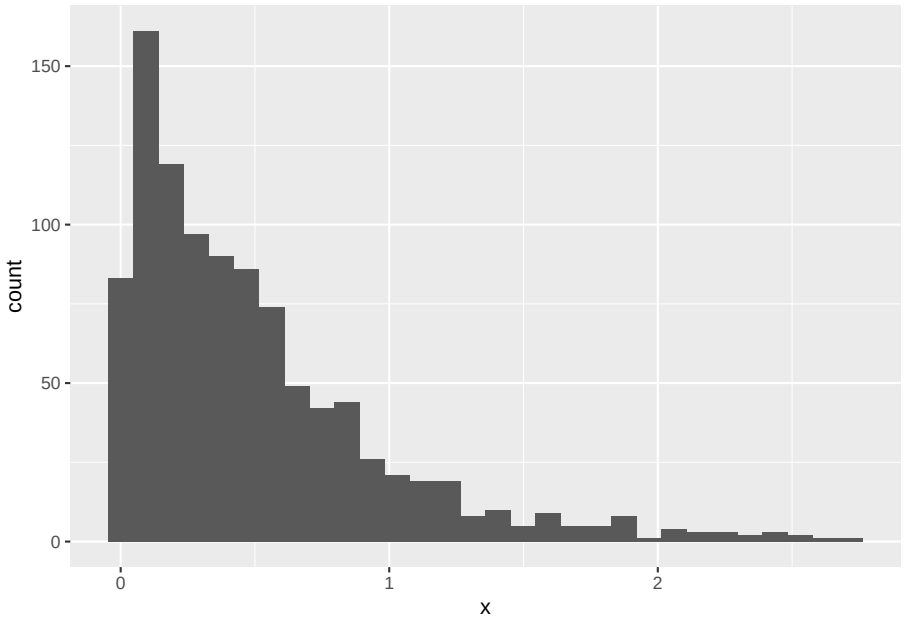
Creating histograms is even easier. Just specify the column that you want to graph as the x column. No y is needed because a histogram is one-dimensional.

Take a x to be a random variable from a gamma distribution.

```
gamma = tibble(x = rgamma(1000, shape = 1, rate = 2))
```

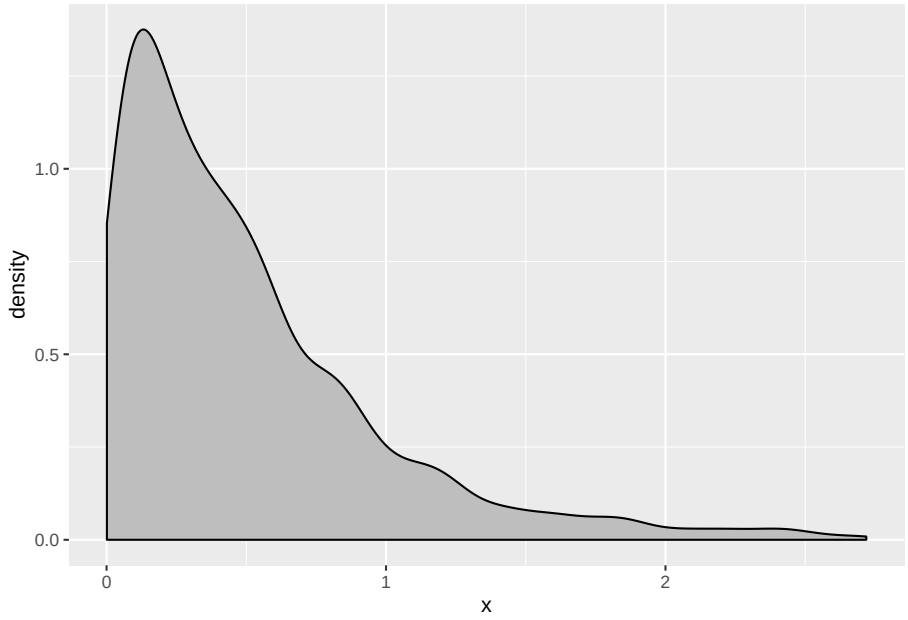
```
p <- gamma %>% ggplot(aes(x = x))
p + geom_histogram()
```

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



We can graph a density instead of a histogram by using `geom_density` instead of `geom_hist`.

```
p + geom_density(fill = "grey")
```



Chapter 8

Tree-based models

8.1 Decision Trees

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

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

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

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

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

This was a very simple example because there were only two variables. If we have more variables, the tree will get large very quickly. This will result in overfitting;

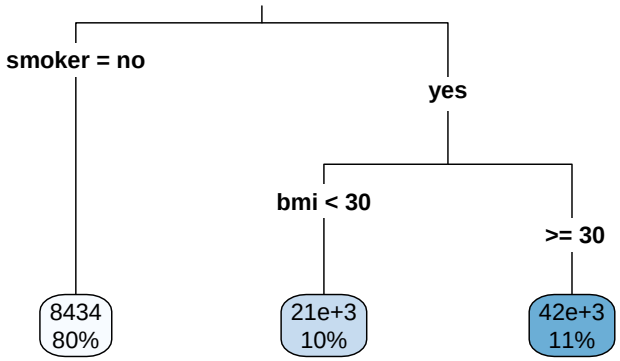


Figure 8.1: Decision tree of health costs

there will be good performance on the training data but poor performance on the test data.

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

Step 1: Choose a variable at random.

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

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

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

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

There are three ways of deciding where to split

- *Entropy* (aka, information gain)

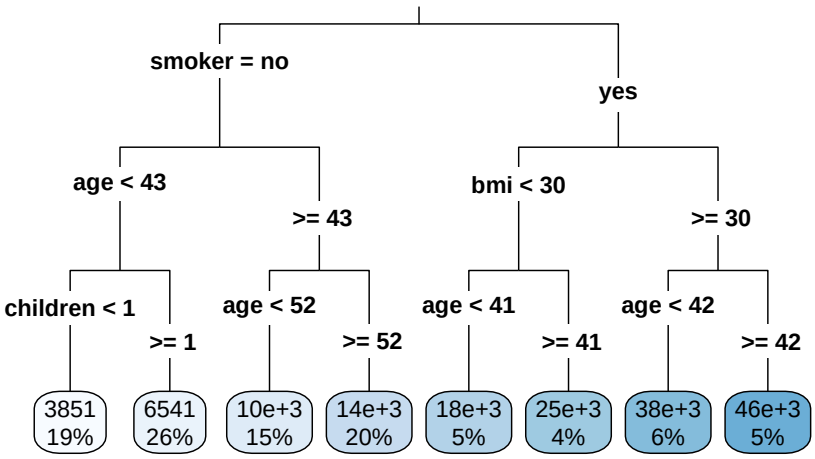
- *Gini*
- *Classification error*

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

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

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

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



Step 4: Apply cost comlexity pruning to simplify the tree

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

A measure of the depth of the tree is the *complexity*. You can think of $|T|$ as the "degrees of freedom" in a linear model. In the above example, $|T| = 8$. The amount of penalization is controlled by α. This is very similar to λ in the Lasso.

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

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

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

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

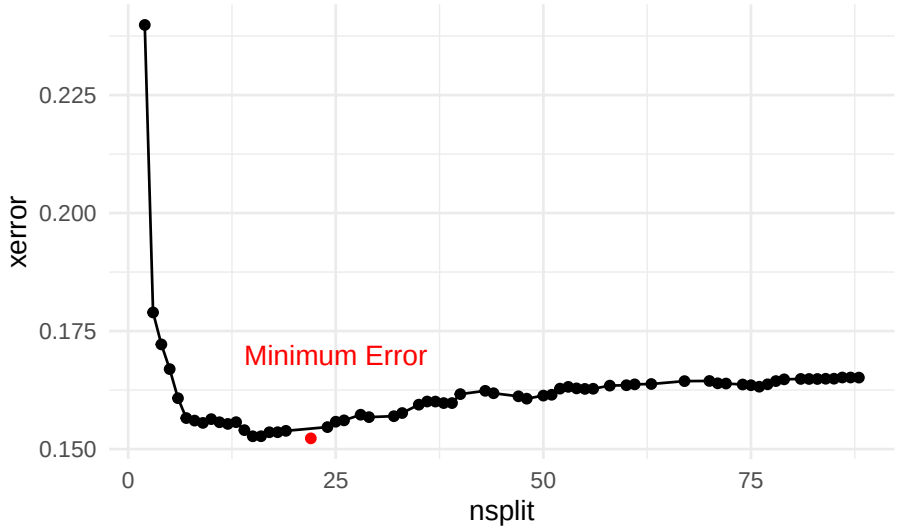
This is stored in the `cptable`

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

cost %>% head()
```

```
## # A tibble: 6 x 3
##   nsplit      CP xerror
##    <dbl>   <dbl>  <dbl>
## 1      0 0.620    1.00
## 2      1 0.144    0.382
## 3      2 0.0637   0.240
## 4      3 0.00967  0.179
## 5      4 0.00784  0.172
## 6      5 0.00712  0.167
```

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



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

You will typically be given the below code, which does this automatically. To get full credit on decision tree questions, mention that you used cross-validation to select the number of splits.

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

8.2 Advantages and disadvantages

Advantages

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

Disadvantages

- Is a “weak learner” because of low predictive power

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

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

8.3 Random Forests

Advantages

- Resilient to overfitting due to bagging
- Only one parameter to tune (mtry, the number of features considered at each split)
- Very good a multi-class prediction
- Nonlinearities
- Interaction effects
- Deal with unbalanced and missing data*Usually requires over/undersamplin

Disadvantages

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

Readings
ISLR 8.1.1 Basics of Decision Trees
ISLR 8.1.2 Classification Trees

8.4 Gradient Boosted Trees

- High prediction accuracy
- Closest model to a "silver bullet" that exists
- Nonlinearities, interaction effects, resilient to outliers, corrects for missing values
- Deals with class imbalance directly through by weighting observations

Disadvantages

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

Readings
ISLR 8.1.1 Basics of Decision Trees
ISLR 8.1.2 Classification Trees

Chapter 9

A Mini-Exam Example

A common regret of students who failed exam PA is that they did not start doing practice exams soon enough. Here is a simple practice exam to help you to understand what concepts that you should focus on learning, to become familiar with the format, and to test your technical skills with R, RStudio, and MS Word.

9.1 Project Statement

ABC Health Insurance company is building a model to predict medical claims. Using only `age` and `sex` information from the prior year, build a model to predict claims for the next year.

9.1.1 Describe the data (1 point)

```
library(tidyverse)
data <- read_csv("C:/Users/sam.castillo/Desktop/R Manual Data/health_insurance.csv") %>%
  select(age, sex, charges) %>%  #put this into an r library
  rename(claims = charges)

data %>% summary()
```

```
##       age            sex                claims
##  Min.   :18.00   Length:1338        Min.   : 1122
##  1st Qu.:27.00   Class :character   1st Qu.: 4740
##  Median :39.00   Mode  :character   Median : 9382
```

```
##  Mean   :39.21                    Mean   :13270
##  3rd Qu.:51.00                    3rd Qu.:16640
##  Max.   :64.00                    Max.   :63770
```

```
data %>% dim()
```

```
## [1] 1338    3
```

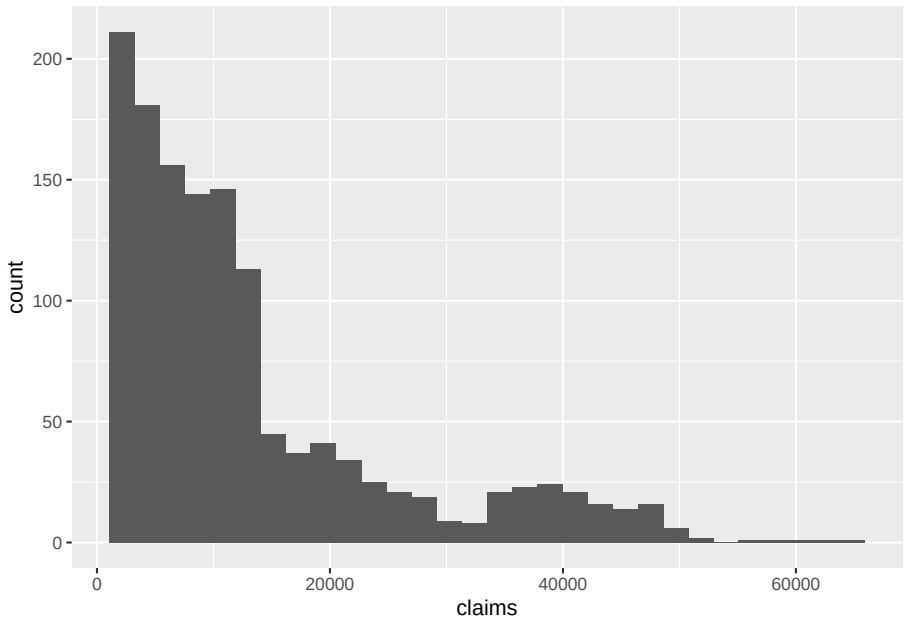
The data consists of 1,338 policies with age and sex information. The objective is to predict future claims.

9.1.2 Create a histogram of the claims and comment on the shape (1 point)

The distribution of claims is strictly positive and right skewed.

```
data %>% ggplot(aes(claims)) + geom_histogram()
```

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



9.1.3 Fit a linear model (1 point)

```
model = lm(claims ~ age + sex, data = data)
```

9.1.4 Describe the relationship between age, sex, and claim costs (1 point)

We fit a linear model to the claim costs using age and sex as predictor variables. The coefficient of `age` is 258, indicating that the claim costs increase by $258 for every one-unit increase in the policyholder's age. The cofficient of 1538 on `sexmale` indicates that on average, men have $1538 higher claims than women do.

```
coefficients(model)
```

```
## (Intercept)         age     sexmale
##   2343.6249    258.8651   1538.8314
```

9.1.5 Write a summary of steps 1-4 in non-technical language (1 point)

ABC Health is interested in predicting the future claims for a group of policyholders. We began by collecting data on 1,538 policy holders which recorded their age, sex, and annual claims. We then created a histogram of the claim costs. A linear model which shows that claim costs increase as age increases, and are higher for men on average.

Chapter 10

Practice Exam

This is a practice exam that is based on the December 2018 PA exam. This is easier than the actual exam.

Chapter 11

Prior Exams

There are the exams and solutions published by the SOA.