Lab Notes

Chapter 2

OpenIntro Biostatistics

Overview

1. Introduction to Probability
   - *OI Biostat* Sections 2.1
2. Conditional Probability
   - *OI Biostat* Sections 2.2.1 - 2.2.4
3. Positive Predictive Value (Bayes' Theorem)
   - *OI Biostat* Section 2.2.5
4. Extended Practice
   - *OI Biostat* Section 2.3

Lab 1 introduces the basic framework for estimating probability through simulated repetitions of an experiment. The `sample()` command and syntax for a `for` loop are introduced in the context of a simple coin flipping experiment, then extended to more complex scenarios.

Lab 2 continues the discussion of control structures by introducing the use of `if` statements for counting successes and for simulating populations based on conditional probabilities.

Lab 3 illustrates three common approaches for calculating the positive predictive value of a diagnostic test: contingency tables, tree diagrams, and simulation. The second part of the lab is a conceptual exploration of the relationships between prevalence, sensitivity, specificity, PPV, and NPV.

Lab 4 features probability problems that are more extensive than those in the text end-of-chapter-exercises, and covers both algebraic and simulation approaches.

Lab 1: Introduction to Probability

Simulation allows for an intuitive way to understand the definition of probability as a proportion of times that an outcome of interest occurs if the random phenomenon could be repeated infinitely. With R, it is possible to feasibly conduct simulations with enough replicates such that the proportion of occurrences with a particular outcome closely approximates the probability p.

This lab introduces the basic elements of R programming required to conduct such simulations: the `sample()` command and `for` loop control structure.

General Advice

Although the R scripts written in these labs are relatively short, it is still important to follow some best practices that help make the code easier to read and modify.

- Organize the code in a clear, logical manner. Be sure that the script can be run from beginning to end, line-by-line, without errors.
- Annotate the code with comments to make it easier to identify the purpose of specific sections.
- Start out by defining the parameters (i.e., variables) that will be used in the script. Use variables as much as possible so that the code is easy to re-use in similar settings.

Random Sampling with `sample()`

The use of `sample()` was introduced in Unit 1 (Lab 1: Introduction to Data), where it was used for selecting a random sample of gene transcripts such that each transcript had an equal chance of being chosen.

In a probability setting, it is not always the case that each outcome is equally likely. The `sample()` function has the generic structure

```
sample(x, size = , replace = FALSE, prob = NULL)
```

where the `prob` argument allows for the probability of sampling each element in `x` to be specified as a vector. When `prob` is omitted, the function will sample each element with equal probability.

The following code simulates the outcome of tossing a biased coin ten times, where the probability of a heads is 0.6; a heads is represented by `1` and a tails is represented by `0` The first argument is the vector `(0, 1)`, and the `prob` argument is the vector `(0.4, 0.6)`; the order indicates that the first element (`0`) is to be sampled with probability 0.4 and the second element (`1`) is to be sampled with probability 0.6.

```
#set the seed for a pseudo-random sample
set.seed(2018)

outcomes = sample(c(0, 1), size = 10, prob = c(0.4, 0.6), replace = TRUE)
outcomes
```

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

Using sum()

The sum() function is used in the lab to return the sum of the outcomes vector, where outcomes are recorded as either 0 or 1. For simplicity, the function was used in its most basic form:

```
sum(outcomes)       #number of 1's (heads)
```

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

```
10 - sum(outcomes)  #number of 0's (tails)
```

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

It is often convenient to combine the sum() function with logical operators.

```
sum(outcomes == 1)  #number of 1's (heads)
```

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

```
sum(outcomes == 0)  #number of 0's (tails)
```

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

This provides more flexibility and can make the code easier to parse quickly, such as for cases where there are more than two outcomes, or when more than two outcomes are of interest. For example, the following application of sum() identifies the number of rolls (out of twenty) of a fair six-sided that are either 1 or greater than 4.

```
#set the seed for a pseudo-random sample
set.seed(2018)

dice.rolls = sample(1:6, size = 20, replace = TRUE)
dice.rolls
```

```
##  [1] 3 4 5 2 5 1 3 4 2 4 3 3 6 1 1 6 5 3 1 3
```

```
sum(dice.rolls == 1 | dice.rolls > 4)
```

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

Additionally, since logical operators work with text strings, it is possible to use them with sum() to return the number of heads if heads is represented by, for example, H.

```
#set the seed for a pseudo-random sample
set.seed(2018)

outcomes = sample(c("T", "H"), size = 10, prob = c(0.4, 0.6), replace = TRUE)
outcomes
```

```
##  [1] "H" "H" "H" "H" "H" "H" "T" "H" "T" "H"
```

```
sum(outcomes == "H")  #number of heads
```

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

`for` Loops

A loop allows for a set of code to be repeated under a specific set of conditions; the `for` loop is one of the several types of loops available in R.

A `for` loop has the basic structure `for( counter ) { instructions }`. The loop below will calculate the squares of the integers from 1 through 5.

- Prior to running the loop, an empty vector `squares` is created to store the results of the loop. This step is referred to as initialization.
- The counter consists of the index variable that keeps track of each iteration of the loop; the index is typically a letter like `i`, `j`, or `k`, but can be any sequence such as `ii`. The index variable is conceptually similar to the index of summation k in sigma notation ($\sum_{k=1}^{n}$). In the example below, the counter can be read as "for every k in 1 through 5, repeat the following instructions..."
- The instructions are enclosed within the pair of curly braces. For each iteration, the value k^2 is to be stored in the k^{th} element of the vector `squares`. The empty vector `squares` was created prior to running the loop using `vector()`.
- So, for the first iteration, `R` sets $k = 1$, calculates 1^2, and fills in the first element of `squares` with the result. In the next iteration, $k = 2$... and this process repeats until $k = 5$ and 5^2 is stored as the fifth element of `squares`.
- After the loop is done, the vector `squares` is no longer empty and instead consists of the squares of the integers from 1 through 5: 1, 4, 9, 16, and 25.

```
#create empty vector to store results (initialize)
squares = vector("numeric", 5)

#run the loop
for(k in 1:5){

  squares[k] = k^2

}

#print the results
squares
```

```
## [1]  1  4  9 16 25
```

Of course, the same result could be easily achieved without a `for` loop:

```
(1:5)^2
```

```
## [1]  1  4  9 16 25
```

The `for` loop is a useful tool when the set of instructions to be repeated is more complicated, such as in the coin tossing scenario from the lab. The loop from Question 2 is reproduced here for reference.

- The loop is set to run for every k from 1 to the value of `number.replicates`, which has been previously defined as 50.
- There are two vectors defined within the curly braces.
  - The first vector, `outcomes.replicate`, consists of the outcomes for the tosses in a single replicate (i.e., iteration of the loop). The number of tosses in a single experiment has been defined as 5. These values are produced from `sample()`. This vector's values are re-populated each time the loop is run.
  - The second vector, `outcomes`, is created by calculating the sum of `outcomes.replicate` for each iteration of the loop. Once the loop has run, `outcomes` is a record of the number of 1's that occurred for each iteration.
- In the language of the coin tossing scenario: the loop starts by tossing 5 coins, counting how many heads occur, then recording that number as the first element of `outcomes`. Next, the loop tosses 5 coins again and records the number of heads in the second element of `outcomes`. The loop stops once it has repeated the experiment (of tossing 5 coins) 50 times.
  - The distinction between the `outcomes.replicate` vector and the `outcomes` vector, for each run of the loop, is that the `outcomes.replicate` vector stores the exact sequence of results while `outcomes` only records the number of heads.
  - For this particular problem, the only information needed is the number of heads; thus, one can think of `outcomes` as summarizing the relevant information from the experiment.

```
for(k in 1:number.replicates){

  outcomes.replicate = sample(c(0, 1), size = number.tosses,
                              prob = c(1 - prob.heads, prob.heads), replace = TRUE)

  outcomes[k] = sum(outcomes.replicate)

}
```

Lab 2: Conditional Probability

The direct relationship between the language of conditional probability and the logic of conditional statements in R can help reinforce the concept behind conditioning on an event.

This lab introduces the `if` statement structure and demonstrates how to use `if` statements within `for` loops. Both algebraic and simulation approaches to calculating conditional probabilities are discussed.

`rep()`

The `rep(x, times)` function is a generic way to replicate elements of a vector `x` a certain number of times. In the lab, it is used to build a vector containing the elements that can be sampled for the scenario of drawing differently colored balls from a bag. The following code creates a vector `balls` that contains 5 `R`'s and 2 `W`'s.

```
balls = rep(c("R", "W"), c(5, 2))
balls
```

```
## [1] "R" "R" "R" "R" "R" "W" "W"
```

Explicitly listing the elements to sample from is typically only necessary when the scenario calls for sampling without replacement. If, for example, the balls were to be drawn with replacement, then the sampling could simply be done from a vector `(R, W)` where there is probability 5/7 of drawing an `R` and 2/7 of drawing a `W`.

`if` Statements

An `if` statement has the basic structure `if ( condition ) { statement }` ; if the condition is satisfied, then the statement will be carried out.

The following loop prints the message "x is greater than 5" if the condition `x > 5` is satisfied; if it is not satisfied, then no message will be printed.

```
x = 100/10

if(x > 5){

  print("x is greater than 5")

}
```

```
## [1] "x is greater than 5"
```

Counting successes

The first two problems in the lab introduce the structure of `if` statements in the context of calculating joint probabilities, such as the joint probability of drawing a white ball on the first pick and a red on the second. The `if` statements are used to record instances of successes, instances when both events of interest occurred.

These `if` statements are nested within `for` loops. The loop from Question 1 is reproduced here for reference.

- In the `if` statement, the condition requires that the first element of the vector `draw` must be a `"W"` and the second element an `"R"`.
- If the condition is satisfied for the k^{th} replicate, then a `1` is recorded as the k^{th} element of the vector `successes`.
- The joint probability can then be calculated by dividing the number of successes by the total number of replicates.

```
for(k in 1:replicates){

  draw = sample(balls, size = number.draws, replace = FALSE)

  if(draw[1] == "W" & draw[2] == "R"){

    successes[k] = 1

  }
}
```

In some cases, the event of interest can be more directly expressed with programming language than with an algebraic approach.

For example, the second part of Question 1 asks about the probability of drawing exactly one red ball. The algebraic approach involves using the logic that there are two scenarios in which exactly one red ball can be observed (red on the second draw or the first draw) and adding together the associated two joint probabilities.

This outcome can be expressed the same way in the condition of the `if` statement:

```
if( (draw[1] == "W" & draw[2] == "R") | (draw[1] == "R" & draw[2] == "W") ){

  successes[k] = 1

}
```

However, it can be expressed more simply with the help of the `sum()` function:

```
if(sum(draw == "R") == 1){

  successes[k] = 1

}
```

Simulating populations

The last two problems in the lab use `if` statements to simulate populations based on known conditional probabilities. This approach is useful when probabilities other than those provided are of interest, such as joint probabilities or the reverse conditional probabilities.

The loop from Question 3 is reproduced here for reference.

- The problem statement provides the conditional probabilities of being tall given sex. Height status must be assigned based on sex; `if` statements are used to specify the conditioning.
- Prior to running the loop, sex was randomly assigned using `sample()` and stored in the vector `sex`, where `0` represents a male and `1` represents a female.
- The vector `tall` will contain the height status of the simulated individuals.
- The first `if` statement has the condition `sex[k] == 0`. If the k^{th} element of `sex` is 0, then the following instructions are used to determine the k^{th} element of `tall`: sampling from `(0, 1)` with the probabilities `1 - p.tall.if.male` and `p.tall.if.male`, respectively.
- The second `if` statement has the condition `sex[k] == 1`. If the k^{th} element of `sex` is 1, then the sampling is done with the probabilities `1 - p.tall.if.female` and `p.tall.if.female`.
- Height status is assigned one at a time as the loop moves from `k in 1:population.size`, so `size = 1` for the `sample()` commands within the `if` statements. When `sample()` is used to assign sex, all the sampling is done in a single step, so `size = population.size`.

```
for(k in 1:population.size){

  if(sex[k] == 0){

    tall[k] = sample(c(0,1), prob = c(1 - p.tall.if.male, p.tall.if.male),
                     size = 1, replace = TRUE)
  }

  if(sex[k] == 1){

    tall[k] = sample(c(0,1), prob = c(1 - p.tall.if.female, p.tall.if.female),
                     size = 1, replace = TRUE)
  }
}
```

The simulated population can be used to estimate joint probabilities such as the probability of being tall and female, or a conditional probability not given in the problem statement like $P(F|T)$.

```
#probability of female and tall
sum(sex == 1 & tall == 1)/population.size
```

```
## [1] 0.017
```

```
#probability of female given tall
sum(sex == 1 & tall == 1)/sum(tall == 1)
```

```
## [1] 0.143339
```

Lab 3: Positive Predictive Value (Bayes' Theorem)

This lab illustrates three common approaches for calculating the positive predictive value of a diagnostic test: contingency tables, tree diagrams, and simulation. These approaches are generalizable to any conditioning problem that involves Bayes' Theorem.

All R programming required for the simulation approach is discussed in the two previous labs. The challenging aspect of this lab is the second half, which consists of concept-oriented questions focused on the relationships between prevalence, sensitivity, specificity, positive predictive value, and negative predictive value.

Using R as a calculator

One advantage R has over a standard hand calculator is the ability to easily store values as named variables. This can make numerical computations like the calculation of positive predictive value much less error-prone. Using a short script like the following to do the computation, rather than directly entering numbers into a calculator, is more efficient. The script can also be re-run with different starting values.

```
#define variables
prevalence = 0.10
sensitivity = 0.98
specificity = 0.95

#calculate ppv
ppv.num = prevalence*sensitivity
ppv.den = ppv.num + (1 - specificity)*(1 - prevalence)
ppv = ppv.num/ppv.den
ppv
```

```
## [1] 0.6853147
```

R can also handle vector operations, which allows for multiple calculations at once when the starting values are defined as vectors.

```
#define variables
prevalence = c(0.01, 0.05, 0.1, 0.2)
sensitivity = rep(0.98, 4)
specificity = rep(0.95, 4)

#calculate ppv
ppv.num = prevalence*sensitivity
ppv.den = ppv.num + (1 - specificity)*(1 - prevalence)
ppv = ppv.num/ppv.den
ppv
```

```
## [1] 0.1652614 0.5077720 0.6853147 0.8305085
```

Lab 4: Extended Practice

This lab consists of conditioning questions that are more challenging than those in the text end-of-chapter exercises, both in terms of the problem context and the extent of manipulation required to calculate probabilities.

The first problem is a classic allele inheritance scenario; problems like these are typical in introductory and mid-level genetics courses. It is an example of a probability problem where attempting to find a solution via intuition or mental logic, rather than the rules of probability, can prove misleading. For example, students often miss a layer of conditioning (part a) or think that eye color of a first child is uninformative with regards to eye color of a second child (part c).

The second problem is a layered conditioning problem that builds to involve three events and requires careful attention to detail with regards to algebraic notation. The lab solutions demonstrate the simulation approach, while the algebraic approach is explained in the text.