Lab Notes

Chapter 3

OpenIntro Biostatistics

Overview

1. Introduction to Random Variables
   - *OI Biostat* Sections 3.1 - 3.2
2. Distributions: Normal and Poisson
   - *OI Biostat* Sections 3.3 - 3.4
3. Distributions Related to Bernoulli Trials
   - *OI Biostat* Section 3.5

Lab 1 introduces the general notion of a random variable and its distribution using a simulation, then discusses the binomial distribution.

Lab 2 discusses the normal distribution and working with normal probabilities, as well as the Poisson distribution.

Lab 3 covers the geometric, negative binomial, and hypergeometric distributions. The simulations in this lab introduce `repeat` and `while` loops.

All three labs include practice problems that illustrate the use of R functions for probability distributions.

Lab 1: Introduction to Random Variables

This first part of the lab introduces the concept of a random variable by exploring the clinical trial example described at the beginning of Chapter 3. All R programming required to understand the simulation code has been previously covered.

The second part of the lab formally introduces the binomial distribution; examples of binomial sampling have appeared in previous labs, but without being associated with a named distribution. The use of `dbinom()` and `pbinom()` are shown.

Binomial Distribution Functions

The function **`dbinom()`** used to calculate $P(X = k)$ has the generic structure

```
dbinom(x, size, prob)
```

where `x` is k, `size` is the number of trials n, and `prob` is the probability of success p.

The following code shows how to calculate $P(X = 5)$ for $X \sim \text{Bin}(10, 0.35)$. It is not necessary to explicitly specify the names of the arguments.

```
#probability X equals 5
dbinom(x = 5, size = 10, prob = 0.35)    #argument names specified
```

```
## [1] 0.1535704
```

```
dbinom(5, 10, 0.35)                      #argument names omitted
```

```
## [1] 0.1535704
```

The function **`pbinom()`** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```
pbinom(q, size, prob, lower.tail = TRUE)
```

where `q` is k, `size` is the number of trials n, and `prob` is the probability of success p. By default, R calculates $P(X \leq k)$. In order to compute $P(X > k)$, specify `lower.tail = FALSE`.

The following code shows how to calculate $P(X \leq 5)$ and $P(X > 5)$ for $X \sim \text{Bin}(10, 0.35)$.

```
#probability X is less than or equal to 5
pbinom(5, 10, 0.35)
```

```
## [1] 0.9050659
```

```
#probability X is greater than 5
pbinom(5, 10, 0.35, lower.tail = FALSE)
```

```
## [1] 0.09493408
```

Lab 2: Distributions: Normal and Poisson

The first part of the lab demonstrates the use of `pnorm()` and `qnorm()` for working with normal probabilities. The practice problems involve both calculating probabilities associated with a particular observation and identifying observations corresponding to a particular probability.

The second part of the lab shows the use of `dpois()` and `ppois()` for calculating Poisson probabilities. The practice problems include examples of calculating λ for different population sizes and units of time.

Normal Distribution Functions

The function **`pnorm()`** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE)
```

where `q` is k, `mean` is the parameter μ, and `sd` is the parameter σ. By default, R calculates $P(X \leq k)$ (`lower.tail = TRUE`) and assumes that mean and standard deviation are 0 and 1, respectively. In order to compute $P(X > k)$, specify `lower.tail = FALSE`.

The following code shows how to calculate $P(X \leq 105)$ and $P(X > 105)$ for $X \sim \mathrm{N}(100, 5)$ and $P(Z \leq 1)$ and $P(Z > 1)$ for $Z \sim \mathrm{N}(0, 1)$.

```
#probability X is less than (or equal to) 105
pnorm(105, 100, 5)
```

```
## [1] 0.8413447
```

```
#probability X is greater than 105
pnorm(105, 100, 5, lower.tail = FALSE)
```

```
## [1] 0.1586553
```

```
#probability Z is less than (or equal to) 1
pnorm(1)
```

```
## [1] 0.8413447
```

```
#probability Z is greater than 1
pnorm(1, lower.tail = FALSE)
```

```
## [1] 0.1586553
```

The function **`qnorm()`** used to identify the observation that corresponds to a particular probability p has the generic structure

```
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE)
```

where `p` is p, `mean` is the parameter μ, and `sd` is the parameter σ. By default, R identifies the observation that corresponds to area p in the lower tail (i.e., to the left) and assumes that mean and standard deviation are 0 and 1. To identify the observation with area p in the upper tail, specify `lower.tail = FALSE`.

The following code shows how to calculate the value of the observation (unstandardized and standardized) where there is 0.841 area to the left (and 0.159 area to the right).

```
#identify X value
qnorm(0.841, 100, 5)
```

```
## [1] 104.9929
```

```
qnorm(0.159, 100, 5, lower.tail = FALSE)
```

```
## [1] 104.9929
```

```
#identify Z value
qnorm(0.841)
```

```
## [1] 0.9985763
```

```
qnorm(0.159, lower.tail = FALSE)
```

```
## [1] 0.9985763
```

Poisson Distribution Functions

The function **dpois()** used to calculate $P(X = k)$ has the generic structure

```
dpois(x, lambda)
```

where x is k and lambda is the rate parameter λ.

The following code shows how to calculate $P(X = 5)$ for $X \sim \text{Pois}(3)$.

```
#probability X equals 5
dpois(5, 3)
```

```
## [1] 0.1008188
```

The function **ppois()** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```
ppois(q, lambda, lower.tail = TRUE)
```

where q is k and lambda is the rate parameter λ. By default, R calculates $P(X \leq k)$. In order to compute $P(X > k)$, specify lower.tail = FALSE.

The following code shows how to calculate $P(X \leq 5)$ and $P(X > 5)$ for $X \sim \text{Pois}(3)$.

```
#probability X is less than or equal to 5
ppois(5, 3)
```

```
## [1] 0.9160821
```

```
#probability X is greater than 5
ppois(5, 3, lower.tail = FALSE)
```

```
## [1] 0.08391794
```

Lab 3: Distributions Related to Bernoulli Trials

This lab discusses the geometric, negative binomial, and hypergeometric distributions. Particular care should be taken when calculating geometric and negative binomial probabilities, since the convention for defining the distributions in the book differs from that used by R.

Simulating geometric and negative binomial random variables uses new programming control structures: the `repeat` and `while` loops.

Geometric Distribution Functions

R defines the geometric distribution as the distribution of the number of failed trials before the first success, while the text defines it as the distribution of the number of trials needed to observe the first success. To adjust for the difference in convention, enter $k-1$ into the R command for `x` (or `q`) when calculating a geometric probability.

The function **`dgeom()`** used to calculate $P(X = k)$ has the generic structure

```
dgeom(x, prob)
```

where `x` is k and `prob` is the probability of success p.

The following code shows how to calculate $P(X = 5)$ for $X \sim \text{Geom}(0.35)$.

```
#probability X equals 5
dgeom(5 - 1, 0.35)
```

```
## [1] 0.06247719
```

The function **`pgeom()`** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```
pgeom(q, prob, lower.tail = TRUE)
```

where `q` is k and `prob` is the probability of success p. By default, R calculates $P(X \leq k)$. In order to compute $P(X > k)$, specify `lower.tail = FALSE`.

The following code shows how to calculate $P(X \leq 5)$ and $P(X > 5)$ for $X \sim \text{Geom}(0.35)$.

```
#probability X is less than or equal to 5
pgeom(5 - 1, 0.35)
```

```
## [1] 0.8839709
```

```
#probability X is greater than 5
pgeom(5 - 1, 0.35, lower.tail = FALSE)
```

```
## [1] 0.1160291
```

Negative Binomial Distribution Functions

R defines the negative binomial distribution as the number of failed trials before r successes, while the text defines it as the distribution of the number of trials needed to observe r successes. To adjust for the difference in convention, enter $k-r$ into the R command for x (or q) when calculating a negative binomial probability.

The function **dnbinom()** used to calculate $P(X = k)$ has the generic structure

```
dnbinom(x, size, prob)
```

where x is k, size is the number of successes r, and prob is the probability of success p.

The following code shows how to calculate $P(X = 5)$ for $X \sim \text{NB}(4, 0.8)$.

```
#probability X equals 5
dnbinom(5 - 4, 4, 0.8)
```

```
## [1] 0.32768
```

The function **pnbinom()** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```
pnbinom(q, size, prob, lower.tail = TRUE)
```

where q is k, size is the number of successes r, and prob is the probability of success p. By default, R calculates $P(X \leq k)$. In order to compute $P(X > k)$, specify lower.tail = FALSE.

The following code shows how to calculate $P(X \leq 5)$ and $P(X > 5)$ for $X \sim \text{NB}(4, 0.8)$.

```
#probability X is less than or equal to 5
pnbinom(5 - 4, 4, 0.8)
```

```
## [1] 0.73728
```

```
#probability X is greater than 5
pnbinom(5 - 4, 4, 0.8, lower.tail = FALSE)
```

```
## [1] 0.26272
```

Hypergeometric Distribution Functions

The function **dhyper()** used to calculate $P(X = k)$ has the generic structure

```
dhyper(x, m, n, k)
```

where x is k, m is the total number of successes m, n is the total number of failures $N - m$, and k is the sample size n.

The following code shows how to calculate $P(X = 5)$ for $X \sim \text{HGeom}(10, 15, 8)$, where $m = 10$, $N - m = 15$, and $n = 8$.

```
#probability X equals 5
dhyper(5, 10, 15, 8)
```

```
## [1] 0.1060121
```

The function **phyper()** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```
phyper(q, m, n, k, lower.tail = TRUE)
```

where q is k, m is the total number of successes m, n is the total number of failures $N - m$, and k is the sample size n. By default, R calculates $P(X \leq k)$. In order to compute $P(X > k)$, specify lower.tail = FALSE.

The following code shows how to calculate $P(X \leq 5)$ and $P(X > 5)$ for $X \sim \text{HGeom}(10, 15, 8)$, where $m = 10$, $N - m = 15$, and $n = 8$.

```
#probability X is less than or equal to 5
phyper(5, 10, 15, 8)
```

```
## [1] 0.9779072
```

```
#probability X is greater than 8
phyper(5, 10, 15, 8, lower.tail = FALSE)
```

```
## [1] 0.02209278
```

if-else Statements

An if-else statement is closely related to the if statement and has the basic structure if(condition) { statement 1 } else { statement 2 } . If the condition is satisfied, the first statement is carried out; else, the second statement is carried out.

The following code prints the message "x is greater than 5" if the condition x > 5 is satisfied; if it is not satisfied, then it prints the message "x is not greater than 5".

```
x = 100/50

if(x > 5){

  print("x is greater than 5")

} else {

  print("x is not greater than 5")

}
```

```
## [1] "x is not greater than 5"
```

The following code shows an example of an if-else statement nested within a for loop. The loop has five iterations, for x in 1 through 5. The %% operator (i.e., modulo operator) finds the remainder after the division of one number by the other; for example, 4 %% 2 returns the value 0, since there is no remainder after 4 is divided by 2.

```
for(x in 1:5){

  if(x %% 2 == 0){
```

```
  print("x is even")

} else {

  print("x is odd")

}

}
```

```
## [1] "x is odd"
## [1] "x is even"
## [1] "x is odd"
## [1] "x is even"
## [1] "x is odd"
```

repeat Loops

A `repeat` loop runs indefinitely until it encounters the `break` statement. It is necessary to place a condition inside the loop to stop it from running indefinitely. The condition can be in the form of an `if` statement or an `if-else` statement.

The loop below calculates the squares of the integers starting at 1 and stops once it encounters a squared value larger than 25.

- A `repeat` loop does not explicitly have a counter like a `for` loop. In this example, `x` performs the role of the counter. The value of `x` is initially set as `1`.
- Each time the loop runs, it executes the statements within the curly braces in `repeat { }`. Here, it calculates the square of `x` and prints the value, then adds `1` to the previous value of `x`.
- If the value of `squares` is greater than 25, then the loop encounters the `break` statement and stops running.
- Note how the loop stops after printing the value 36, *not* after printing 25. Since 25 is not greater than 25, the condition in the `if` statement is not fulfilled and the `break` statement is not executed.

```
#initialization
x = 1

#run the loop
repeat{

  squares = x^2
  print(squares)
  x = x + 1

  if(squares > 25){
```

```
    break

  }

}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
```

The loop structure used to simulate a geometric random variable is reproduced here for reference.

- The `repeat` loop is nested within the `for` loop. The `for` loop runs for every k from 1 to the value of `replicates`. For each iteration of the `for` loop, the `repeat` loop runs until a successful trial occurs.
- The value of `trial.number` is initially set as 1. This variable acts as the counter in the `repeat` loop. Recall that for a geometric random variable, it is of interest to track how many trials have occurred.
- Each iteration of the `repeat` loop represents one independent Bernoulli trial. The outcome of the trial is recorded in `outcome.individual`.
  - If the trial is a failure, then a 1 is added to `trial.number` and the loop continues.
  - If the trial is a success, then the current value of `trial.number` is recorded as the k^{th} element of `trials.for.first.success` and the `break` statement is encountered. The next iteration of the `for` loop starts.

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

  trial.number = 1

  repeat{

    outcome.individual = sample(c(0, 1), size = 1,
                                prob = c(1 - p, p), replace = TRUE)

    if(outcome.individual == 0){

      trial.number = trial.number + 1

    } else {

      trials.for.first.success[k] = trial.number

      break
```

```
    }

  }

}
```

The version of the negative binomial simulation in the lab uses a `while` loop, but it is also possible to write the simulation with a `repeat` loop.

- In addition to an `if` statement that contains the `break` statement, this `repeat` loop contains an `if` statement that keeps track of the number of successful trials in `successes`.
- The values of both `trial.number` and `successes` are initially set as `0`.
- Each iteration of the `repeat` loop represents one independent Bernoulli trial. The outcome of each trial is recorded in `outcome.individual`.
  - With each iteration, a `1` is added to `trial.number`.
  - If the outcome is a success, then a `1` is added to `successes`.
  - If the number of successes equals r, then the current value of `trial.number` is recorded as the k^{th} element of `trials.for.rth.success` and the `break` statement is encountered. The next iteration of the `for` loop starts.

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

  trial.number = 0
  successes = 0

  repeat {

    outcome.individual = sample(c(0,1), size = 1,
                                prob = c(1 - p, p), replace = TRUE)

    trial.number = trial.number + 1

    if(outcome.individual == 1){

      successes = successes + 1

    }

    if(sum(successes) == r){

      trials.for.rth.success[k] = trial.number

      break

    }
```

```
  }

}
```

`while` Loops

A `while` loop runs until the specified condition is no longer satisfied and has the basic structure `while( condition ) { instructions }`.

The loop below calculates the squares of the integers starting at 1 as long as the value of `x` is less than 5.

- The value of `x` is initially set as `1`.
- Each time the loop runs, it executes the statements within the curly braces. Here, it calculates the squares of `x` and prints the value, then adds `1` to the previous value of `x`.
- The loop will run as long as the current value of `x` is less than 5.

```
#initialization
x = 1

#run the loop
while(x < 5){

  squares = x^2
  print(squares)
  x = x + 1

}
```

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

The loop structure used to simulate a negative binomial random variable is reproduced here for reference.

- The version of the simulation code with a `while` loop is very similar to the version with a `repeat` loop.
- Since the `while` loop explicitly has a condition stated at the outset, it does not contain an `if` statement that counts the number of successes and leads to a `break` statement. Instead, the loop runs as long as the value of `successes` is less than `r`.
- Note how the line that specifies the recording of the current value of `trial.number` in the vector `trials.for.rth.success` is outside the `while` loop. It will record the value of `trial.number` right after the `while` loop stops; i.e., after the `while` loop has encountered an iteration in which `successes` has value `r`.

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

  trial.number = 0
  successes = 0

  while(successes < r){

    outcome.individual = sample(c(0, 1), size = 1,
                                prob = c(1 - p, p), replace = TRUE)

    if(outcome.individual == 1){

      successes = successes + 1

    }

    trial.number = trial.number + 1

  }

  trials.for.rth.success[k] = trial.number

}
```