Simulation

Chapter 11

Stats 20: Introduction to Statistical Programming with R

UCLA

Contents

Acknowledgements: Vivian Lew and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Understand that pseudorandom numbers are not truly random.
- Use the `sample()` function to simulate drawing random samples.
- Simulate sampling distributions.
- Simulate sampling from the uniform and normal distributions.
- Compute probabilities and quantiles from the normal distribution.

1 Introduction

A scenario (or experiment) is called **random** if its outcome cannot be predicted with certainty. A **random variable** is a variable whose possible values are outcomes from a random scenario.

Even though random variables cannot be predicted with certainty, there are patterns that emerge when observing a random scenario multiple times. Through modeling and summarizing data from random variables, we are able to learn how often different values occur and get an idea of what values are more common (or "typical") and what values are unusual. More formally, a random variable is characterized by its **probability distribution**, which describes the probabilities of the possible values that the random variable can take.

A **statistic** is a numerical characteristic of a sample of data. Collecting measurements on a different sample will produce different data, so statistics are also outcomes from a random scenario. Thus statistics are themselves random variables. A **sampling distribution** is the probability distribution of a statistic.

When a statistic is used to estimate an unobserved parameter (of a population or of a statistical model), it is called an **estimator**. Much of the theory behind statistical inference revolves around understanding the sampling distribution of estimators.

- The margin of error for a confidence interval of a parameter (such as the population mean or population proportion) is computed using the standard error of the sampling distribution of its estimator (such as the sample mean or sample proportion).
- In hypothesis testing, the p-value of a sample is defined as the probability of observing a test statistic at least as extreme as the observed value, assuming the null hypothesis is true.

When certain conditions are met (or can be reasonably assumed), these quantities can be computed analytically (i.e., with mathematical formulas) or with large sample approximations (e.g., the Central Limit Theorem). In these cases, there are built-in functions in R that will do these computations for you. However, when theory cannot be reasonably applied, we can use computer-based approximation methods.

2 Pseudorandom Number Generation

Simulations of random experiments rely on generating random numbers with a computer. However, in the strictest sense, computers cannot generate truly random numbers. Rather, computers follow a deterministic algorithm (hidden from the user) that generates **pseudorandom** numbers ("pseudo" means fake or false): Since we do not know the algorithm, the numbers appear random. If we knew the algorithm, the numbers would not be random at all.

In R (and other programs which generate random numbers), the hidden algorithm starts from an initial number (called the **seed**) and generates pseudorandom numbers from it. By default, R will generate the seed based on the current time. The user can specify (or **set**) the seed using the `set.seed()` function so that the "random" numbers that are generated from a given function are the same every time the function is run. Being able to set the seed allows researchers to reproduce simulation results.

Pseudorandom numbers are considered **statistically random**, in that they are "random enough" for statistical analysis and inference. We will use and refer to computer generated random numbers as if they are random, but it is implicitly understood that they are pseudorandom.

More on psuedorandomness: https://en.wikipedia.org/wiki/Pseudorandomness

3 The `sample()` Function

The `sample()` function is used to choose a subset of values (i.e., *sample*) at random from a fixed set of values. The `sample()` function inputs a vector `x` from which it will draw a sample of a specified `size`, either with or without replacement. The output is a vector of values drawn from the input vector `x`.

The syntax for `sample()` is `sample(x, size, replace = FALSE, prob = NULL)`, where the arguments are:

- `x`: The vector of values from which to sample.

  Note: If `x` is a numeric scalar (vector of length 1) greater than or equal to 1 (`x > 1`), then the sampling will be done on `1:x`.

- `size`: The number of values to draw from `x`. By default, `size` will be equal to the number of elements being sampled, so `sample()` will just rearrange the values in `x`.

- `replace`: A logical value that specifies whether to sample with or without replacement. The default setting is `replace = FALSE` (without replacement).

- `prob`: A vector that specifies the probability weights for each element being sampled.

```
# Set the seed for reproduceability
set.seed(20)
# Rearrange the numbers from 1 to 10
sample(1:10) # Same as sample(10)
```

```
 [1]  6  8  2  1  9  5 10  4  7  3
```

```
# Choose 5 numbers out of the numbers 1 to 10
sample(1:10, size = 5)
```

```
[1] 9 3 5 1 8
```

Note: The `sample()` function uses a pseudorandom number generator to decide which values to choose, so running the same code multiple times will result in different values. Setting the seed with `set.seed()` before running `sample()` results in the same random numbers being chosen every time.

By default, the `sample()` function draws samples from the input vector *without replacement*. This is like drawing a card from a deck of playing cards and not putting it back in the deck: If you draw a second card from the remaining deck, you cannot draw the same card that you drew the first time. Sampling *with replacement* would mean that each time you draw a card, you return your previous draw(s) into the deck and reshuffle before drawing another card. To sample with replacement, set `replace = TRUE`.

```
# Choose 10 numbers out of the numbers 1 to 10, with replacement.
sample(1:10, replace = TRUE) # Values are repeated
```

```
 [1] 8 2 4 9 5 1 9 6 8 4
```

The optional argument `prob` inputs a vector that specifies the probability weights for each entry in the input vector `x`. The ith entry of `prob` (`prob[i]`) corresponds to the probability of drawing the ith entry of `x` (`x[i]`). By default, the probability weights are the same for every entry in the input, so each entry has an equal chance of being drawn.

As a simple example, we illustrate how to simulate flipping a coin using the `sample()` function. We consider flipping a weighted coin, where the probability of heads is 0.8 and the probability of tails is 0.2. We denote heads by "H" and tails by "T". We flip the coin 20 times.

```r
# Simulate flipping a weighted coin 20 times, where H is heads and T is tails.
coins <- sample(c("H", "T"), size = 20, replace = TRUE, prob = c(0.8, 0.2))
coins
```

```
 [1] "H" "H" "T" "H" "H" "H" "H" "T" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H" "H"
[20] "H"
```

```r
# Summarize the results of the 20 coin flips.
table(coins)
```

```
coins
 H  T 
18  2 
```

Note: In order to flip 20 coins from a vector of length 2, we must specify `replace = TRUE`. This makes sense, since we always have the choices "H" or "T" for every flip. If `replace = FALSE`, R will throw an error.

```r
sample(c("H", "T"), size = 20)
```

```
## Error in sample.int(length(x), size, replace, prob) :
##          cannot take a sample larger than the population when 'replace = FALSE'
```

Question: How would you simulate rolling a fair six-sided die 12 times?

4 Simulating Random Processes

A **random process** is a random scenario that can be repeated under identical conditions. Coin flips, die rolls, and lottery draws can all be considered random processes. The `sample()` function can be used to simulate one instance (outcome) of the random process. By repeatedly simulating the random process many times, we can estimate the probability (long-run proportion) of particular outcomes occurring.

4.1 The `replicate()` Function

The `replicate()` function is used to repeat (replicate) the execution of a command or set of commands a fixed number of times.

The syntax for `replicate()` is `replicate(n, expr, simplify = "array")`, where the arguments are:

- `n`: The number of replications to perform.
- `expr`: The expression (i.e., commands or set of commands) to evaluate repeatedly.
- `simplify`: Logical or character value that specifies whether to simplify the output to a matrix or array.

The `replicate()` function behaves similar to the `with()` function:

- Using curly braces `{}`, it is possible to input multiple commands inside the `expr` argument, but only the last command will return output.
- The `expr` commands are evaluated in a local environment. Objects created or reassigned inside `replicate()` will not appear in the global environment.

Note: The `replicate()` function is actually a wrapper for `sapply()`, so the output of `replicate()` is flexible depending on the last command in the `expr` argument. In particular:

- If the result of the commands in `replicate()` is a vector of length 1 (i.e., a scalar), then `replicate()` will return a vector.
- If the result of the commands in `replicate()` is a vector of the same length (greater than 1), then `replicate()` will return a matrix.

- If the result of the commands in `replicate()` is not a vector of the same length, then `replicate()` will return a list.

The most common use of `replicate()` is to replicate random number generation to simulate random processes.

```
set.seed(117)
replicate(n = 5, sample(10, size = 3))
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    3   10    3    8    1
[2,]    6    7    2    2    5
[3,]   10    4    1    6    8
```

```
replicate(n = 20, {
  sample_i <- sample(10, size = 5)
  mean(sample_i)
})
```

```
 [1] 4.8 5.0 5.8 6.8 5.6 6.4 6.2 5.8 6.4 6.6 8.0 5.2 6.6 4.0 5.0 6.0 6.0 6.4 5.0
[20] 7.2
```

```
replicate(n = 2, which(sample(10, size = 8) > 5))
```

```
[[1]]
[1] 1 2 3 8

[[2]]
[1] 1 6 7
```

4.1.1 Example: MEGA Millions

MEGA Millions is a nationwide lottery game in which players pick five numbers from 1 to 70 and one MEGA number from 1 to 25. The six winning numbers are drawn at random, and a player wins the grand prize if all six numbers are chosen correctly.

Suppose we are interested in the probability that all six of the winning numbers are greater than 20. The MEGA Millions game is played twice a week. Rather than observing the lottery in real-time, we can simulate the winning numbers many times using a `for()` loop to estimate how often the winning numbers are all greater than 20.

```
M <- 10000 # Number of repetitions
all_over_20 <- logical(M)

# Set the seed for reproduceability
set.seed(9999) # Always set the seed OUTSIDE the for loop. (Why?)

for (i in seq_len(M)) {
  winning <- c(sample(70, size = 5), sample(25, size = 1))
  all_over_20[i] <- all(winning > 20)
}

table(all_over_20) / M
```

```
all_over_20
 FALSE   TRUE
0.9668 0.0332
```

From the 10000 simulated draws of the lottery, we see that the winning numbers from MEGA Millions are all above 20 about 3.32% of the time.

Side Note: This probability can be computed theoretically to be equal to 10593800/302575350, or roughly about 0.035.

Question: How can this `for()` loop be modified to store the winning numbers from each simulated draw?

The above `for()` loop can be written using the `replicate()` function:

```
set.seed(9999)
M <- 10000
all_over_20 <- replicate(n = M, {
  winning <- c(sample(70, size = 5), sample(25, size = 1))
  all(winning > 20)
})
table(all_over_20) / M
```

```
all_over_20
 FALSE   TRUE
0.9668 0.0332
```

Question: How can this `replicate()` command be modified to store the winning numbers from each simulated draw?

5 Simulating Random Sampling

5.1 Simple Random Sampling

Simple random sampling is a method of sampling from a finite population in which every sample of size n is equally likely to be selected.

To take a simple random sample, we start with a **sampling frame**, which is a list of all the observations in the population.

As an example, consider the dataset found at: http://www.isi-stats.com/isi/data/chap3/CollegeMidwest.txt.

The data[1] contains two variables gathered from the registrar at a small midwestern college on all students at the college in spring 2011. The variables are:

- `OnCampus`: Whether or not a student lives on campus (`Y` or `N`)
- `CumGpa`: The student's cumulative GPA.

Since this is data on *all* students at the college, we will treat the students observed in this data to be the population.

```
college <- read.table("CollegeMidwest.txt", header = TRUE)
head(college, 3)
```

```
  OnCampus CumGpa
1        N   2.92
2        N   3.59
3        N   3.36
```

```
dim(college)
```

```
[1] 2919    2
```

[1]The dataset comes from the textbook *Introduction to Statistical Investigations* by Tintle et al., 2016.

To implement simple random sampling with the `sample()` function, we will use two steps:

(1) We will use `sample()` to randomly select n numbers between 1 and 2919 (the number of students in the population). This represents choosing the students based on ID numbers.

(2) We then use the selected numbers from Step (1) as an index for the rows of observations in the `college` data frame that we want to extract as our sample.

As an example, we take a simple random sample of size 5 from the `college` data frame.

```r
set.seed(20)
sample_index <- sample(2919, size = 5)
sample_index
```

```
[1]  166 1215 1899 1912 1666
```

```r
college[sample_index, ]
```

```
     OnCampus CumGpa
166         Y   3.65
1215        Y   2.68
1899        N   3.88
1912        N   2.84
1666        N   2.42
```

Question: How would you compute the proportion of college students in the sample (i.e., the sample proportion) who live off campus?

5.2 Simulating Sampling Distributions

We want to illustrate the sampling variability (also called sample-to-sample variability) of the sample proportion. That is, when we take different random samples, how does the sample proportion of students who live on campus vary from sample to sample?

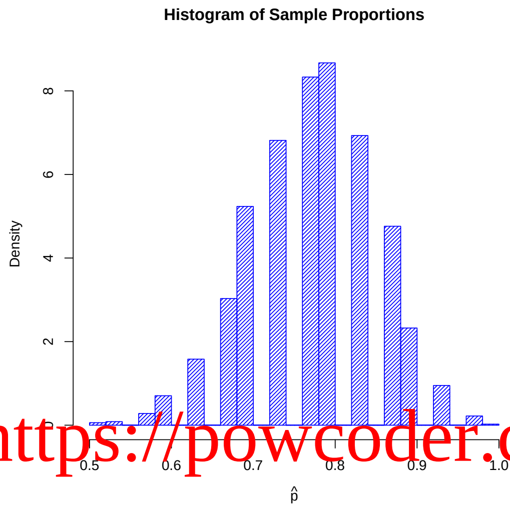
We can use a `for()` loop to simulate sampling many (10000) random samples of size 30 from the population of students in the `college` data. For each random sample, we will compute the proportion of the sample who live on campus.

```r
# We first create objects for common quantities.
n <- 30 # The sample size
N <- nrow(college) # The population size
M <- 10000 # Number of samples/repetitions

set.seed(9999)
phats <- replicate(M, {
  index <- sample(N, size = n)
  sample_i <- college[index, ]
  mean(sample_i$OnCampus == "Y")
})
```

The distribution of the `phats` vector is an approximation to the sampling distribution of the sample proportion for samples of size $n = 30$.

```r
hist(phats,
  prob = TRUE, breaks = 25, xlab = expression(hat(p)),
  main = "Histogram of Sample Proportions", col = "blue", density = 30
)
```



Question: How can the above code be modified to simulate the sampling distribution of the mean cumulative GPA for the students at the College of the Midwest?

6 Simulating From Probability Models

The `sample()` function allows us to choose at random from a list of objects. To sample from more complicated scenarios (such as sampling over a range or interval of values), we need other functions.

A **probability model** is a description of how we think data is generated. It represents a summary of the unobservable **data generating mechanism**. A probability model is not *actually* how the data is generated (this is why it is called a model), but it is used as a reasonable description for how the data appears to vary. For example, there are many factors that contribute to a person's height, including genetics, environmental factors, and diet. However, by combining all of those factors and only observing the end result (the height) of each person, we see that the variation in heights of many people follows a bell-shaped pattern.

6.1 The Uniform Distribution

A random variable has a **uniform distribution** if all possible outcomes are equally likely.

- The random variable has a **discrete uniform distribution** if it has finitely many possible values. If there are N possible values, each value has probability $1/N$.
- The random variable has a **continuous uniform distribution** on the interval $[a, b]$ if the possible values occur over $[a, b]$. The probability density function is given by

$$f(x) = \frac{1}{b-a}, \quad \text{for every } x \text{ in } [a, b].$$

As we have seen in the previous section, the `sample()` function with `replace = TRUE` can be used to generate samples from discrete uniform distributions.

The `runif()` function generates samples from continuous uniform distributions. The command `runif(n, min, max)` generates `n` random uniform numbers between `min` and `max`. The default values are `min = 0` and `max = 1`.

```r
# Generate 10 uniform numbers between 0 and 1.
runif(10)
```

```
 [1] 0.38891619 0.76590136 0.10613288 0.09096281 0.07999616 0.76425485
 [7] 0.04804261 0.84567624 0.50902595 0.60722144
```

```r
# Generate 20 uniform numbers between -5 and 5.
runif(20, min = -5, max = 5)
```

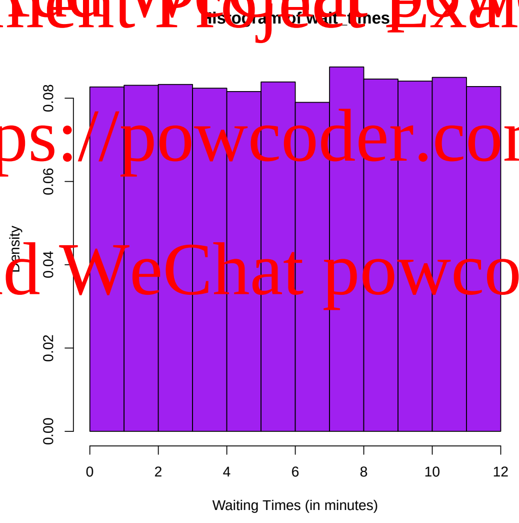
```
 [1]  3.9749887 -0.3483158  3.4615730 -0.2233110 -4.0830404 -2.2477559
 [7]  3.4702129 -3.4701198 -3.2834585 -0.1466863 -3.2061613 -2.4135078
[13]  4.4510890 -2.4032373 -1.2881156 -1.7837484  2.2997566 -1.3253091
[19] -0.1097930  0.1166595
```

6.1.1 Example: Waiting for the Bus

Suppose a local bus arrives every 12 minutes. If you arrive at the bus stop at a randomly chosen time (independent of the bus schedule), then the number of minutes you wait for the bus can be modeled by a continuous uniform distribution on the interval $[0, 12]$.

Assuming the same conditions every day, we can simulate 10000 days of waiting for the bus using `runif()` and plot the results in a histogram.

```r
# Set the seed for reproduceability
set.seed(24601)
# Generate 10000 uniform numbers between 0 and 12.
wait_times <- runif(10000, 0, 12)
# Plot the wait times on a histogram.
hist(wait_times, prob = TRUE, xlab = "Waiting Times (in minutes)", col = "purple")
```



```r
mean(wait_times)
```

```
[1] 6.027837
```

Question: How can we estimate the probability of waiting between 4 and 10 minutes for the bus?

6.2 The Normal Distribution

The **normal distribution** (or model or curve) is a probability model that is used to describe many real-life variables which exhibit a (symmetric and unimodal) bell-shaped curve in their distributions. The shape of the normal distribution is completely determined by two parameters: its mean and standard deviation.

6.2.1 The `rnorm()` Function

The **`rnorm()`** function generates a random sample of a given size from a normal distribution with specified mean and standard deviation. This is analogous to drawing samples from a population, where the population can be well represented by a normal model.

The command `rnorm(n, mean, sd)` generates a random sample of size `n` from the normal distribution with mean `mean` and standard deviation `sd`. The default mean is `mean = 0` and the default standard deviation is `sd = 1`.

For example, we can simulate drawing a sample of 20 heights from a population that follows a normal distribution with a mean height of 68 inches and a standard deviation of 2 inches.

```r
# Set the seed for reproduceability
set.seed(123)
# Sample 20 random numbers from N(68,2).
rnorm(n = 20, mean = 68, sd = 2)
```

```
 [1] 66.87905 67.53965 71.11742 68.14102 68.25858 71.43013 68.92183 65.46988
 [9] 66.62629 67.10868 70.44816 68.71963 68.80154 68.22137 66.88832 71.57383
[17] 68.99570 64.06677 69.40271 67.05442
```

6.2.2 The `dnorm()` Function

The probability density function for a normal random variable X is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2},$$

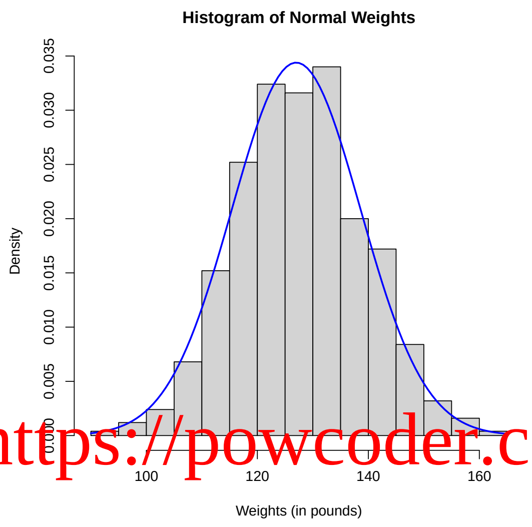
where μ is the mean of X and σ is the standard deviation of X. A **standard normal** random variable has mean $\mu = 0$ and $\sigma = 1$.

The **`dnorm()`** function evaluates the normal density function. The syntax for the main arguments of `dnorm()` is `dnorm(x, mean, sd)`. The function inputs a numeric value (or vector) in the `x` argument and computes the value of the normal density function at `x` when the mean is `mean` and the standard deviation is `sd`. The default mean is `mean = 0` and the default standard deviation is `sd = 1`.

The `dnorm()` function can be used to visualize the normal density curve. It is particularly useful to superimpose the normal density curve over a histogram of observed values from a normal distribution.

Suppose we simulate drawing a sample of 500 weights from a population that follows a normal distribution with a mean weight of 127 pounds and a standard deviation of 11.6 pounds. We can plot the histogram of the weights in the sample and superimpose the normal density curve over it.

```r
# Set the seed for reproduceability
set.seed(321)
# Sample 500 random numbers from N(127,11.6).
weights <- rnorm(500, 127, 11.6)
# Plot the weights on a histogram.
hist(weights, prob = TRUE,
     xlab = "Weights (in pounds)", main = "Histogram of Normal Weights"
)
# Add the N(127,11.6) density curve.
curve(dnorm(x, 127, 11.6), lwd = 2, col = "blue", add = TRUE)
```



Note: The `curve()` function is a high level plot function that draws a curve corresponding to a function over a specified interval. The syntax for `curve()` is `curve(expr, from, to, n, ...)`, where the `expr` argument is a vectorized numeric function in terms of `x`. The `expr` function is evaluated at `n` equally spaced points over the range [`from`, `to`]. Like `hist()`, the `curve()` function has an optional `add` argument that superimposes a curve to an existing plot.

6.2.3 The `pnorm()` Function

The function that computes the cumulative (left-tail) area under the probability density curve of a random variable is called the **cumulative distribution function** (CDF) or simply **distribution function** of the random variable. The `pnorm()` function is the distribution function for normal random variables. More simply put, `pnorm()` computes probabilities from a normal distribution with specified mean and standard deviation.

The syntax for the main arguments of `pnorm()` is `pnorm(q, mean, sd, lower.tail)`. The function inputs a numeric value (or vector) in the `q` argument and computes the probability that a value drawn from a normal distribution with mean `mean` and standard deviation `sd` will be *less than or equal to* `q`. The default mean is `mean = 0` and the default standard deviation is `sd = 1`.

The optional argument `lower.tail` inputs a logical value and changes the direction of the probability. The default is `lower.tail = TRUE`, so the `pnorm()` function will compute the probability that a value drawn from the normal distribution will be *less than or equal to* `q`. If we set `lower.tail = FALSE`, then the `pnorm()` function will compute the probability that a value drawn from the normal distribution will be *greater than or equal to* `q`.

```r
# The probability of drawing a value less than 1.96 from N(0,1).
pnorm(1.96)
```

```
[1] 0.9750021
```

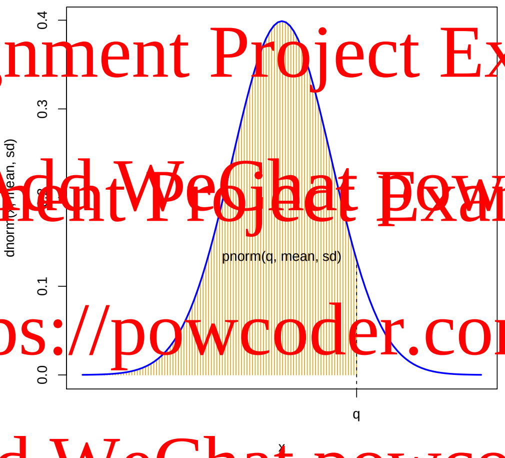
```r
# The probability of drawing a value between 66.5 and 69 from N(68,2).
pnorm(69, 68, 2) - pnorm(66.5, 68, 2)
```

```
[1] 0.4648351
```

Question: For any normal distribution, what is the probability of drawing a value within 2 standard deviations of the mean?

Note: The `pnorm()` function computes the area under the `dnorm()` density curve. The plot below visualizes the output of `pnorm()`. The goldenrod shaded region is `pnorm(q, mean, sd)`, which is the area under `dnorm(x, mean, sd)` to the left of the value `q`.

```r
# Shade the dnorm(x,0,1) curve from -4 to 1.5 in goldenrod.
# The type="h" argument draws vertical lines ("h" stands for "height").
# The xaxt="n" argument suppresses the labels on the x-axis.
curve(dnorm,
  from = -4, to = 1.5, type = "h", col = "goldenrod",
  xlim = c(-4, 4), ylab = "dnorm(x, mean, sd)", xaxt = "n"
)
# Add the dnorm(x, 0, 1) curve in blue.
curve(dnorm, from = -4, to = 4, lwd = 2, col = "blue", add = TRUE)
# Add a horizontal line segment at q (q = 1.5 for illustration purposes).
segments(1.5, -1, 1.5, dnorm(1.5), lty = 2)
# Add the label "q" on the x-axis at x = 1.5.
axis(1, at = 1.5, labels = "q")
# Add text to explain that the shaded region is pnorm(q, mean, sd).
text(0, dnorm(0) / 3, "pnorm(q, mean, sd)")
```



6.2.4 The `qnorm()` Function

The q **quantile** of a probability distribution is the value where q probability lies to the left and $1 - q$ probability lies to the right. The q quantile is also called the $100q$th percentile. For example, the 0.99 quantile of a distribution is the same as the 99th percentile. The first, second, and third quartiles are, respectively, the 0.25, 0.5, and 0.75 quantiles.

The **`qnorm()`** function is the **quantile function** for the normal distribution. The syntax for the main arguments of `qnorm()` is `qnorm(p, mean, sd)`. The function inputs a probability value (or vector) in the `p` argument and computes the quantile for that probability from a normal distribution with mean `mean` and standard deviation `sd`. The default mean is `mean = 0` and the default standard deviation is `sd = 1`.

```r
# The 0.975 quantile of N(0,1)
qnorm(0.975)
```

```
[1] 1.959964
```

```r
# The 0.05 and 0.95 quantiles of N(68,2)
qnorm(c(0.05, 0.95), mean = 68, sd = 2) # Can input a vector of probabilities
```

```
[1] 64.71029 71.28971
```

For example, we can compute the 0.99 quantile (i.e., the 99th percentile) for the weights of a population that follows a normal distribution with a mean of 127 pounds and a standard deviation of 11.6 pounds.

```r
# The 0.99 quantile of N(127,11.6)
qnorm(0.99, 127, 11.6)
```

```
[1] 153.9856
```

Side Note: The `pnorm(q, mean, sd)` and `qnorm(p, mean, sd)` functions are *inverses* of each other. The `pnorm()` function inputs quantiles `q` and outputs left-tail probabilities `p`. The `qnorm()` function inputs left-tail probabilities `p` and outputs quantiles `q`.

```r
pnorm(qnorm(0.95))
```

```
[1] 0.95
```

```r
qnorm(pnorm(1.96))
```

```
[1] 1.96
```

Note: The `quantile()` function computes (sample) quantiles for an input vector of numeric data. The syntax for the main arguments is `quantile(x, probs, na.rm)`, where:

- `x`: The numeric input vector for which we want to compute (sample) quantiles.
- `probs`: A numeric vector of probabilities. By default, `probs = seq(0, 1, by = 0.25)`.
- `na.rm`: A logical value for whether to remove `NA` values. By default, `na.rm = FALSE`.

By default, `quantile(x)` computes the five-number summary.

```r
quantile(trees$Height)
```

```
  0%  25%  50%  75% 100% 
  63   72   76   80   87 
```

```r
quantile(trees$Height, 0.85)
```

```
 85% 
81.5 
```

```r
quantile(trees$Height, c(0.05, 0.95))
```

```
  5%  95% 
64.5 85.5 
```

6.3 Simulating from Other Known Distributions

As one might expect, parametric families of probability distributions that are commonly used in statistics have built-in functions in base R for their densities, distribution functions, quantile functions, and for generating random numbers from them.

We have previously introduced the `rnorm()`, `dnorm()`, `pnorm()`, and `qnorm()` functions for the normal distribution:

- `rnorm()` generates random numbers from the normal distribution.
- `dnorm()` computes the density function.
- `pnorm()` computes the distribution function.
- `qnorm()` computes the quantile function.

In each of these functions, we need to specify the `mean` and `sd` parameters.

It turns out that the `r`, `d`, `p`, and `q` prefixes specify the analogous functions for other common parametric families of distributions. The syntax used for each function naturally follows the same syntax as the corresponding `norm` version.

For the ***t*-distribution**:

- `rt()` generates random numbers from the *t*-distribution.
- `dt()` computes the density function.
- `pt()` computes the distribution function.
- `qt()` computes the quantile function.

In each of these functions, we need to specify the `df` parameter (`df` stands for degrees of freedom).

For the **binomial distribution**:

- `rbinom()` generates random numbers from the binomial distribution.
- `dbinom()` computes the density (probability mass) function.
- `pbinom()` computes the distribution function.
- `qbinom()` computes the quantile function.

In each of these functions, we need to specify the `size` and `prob` parameters, where `size` is the number of trials and `prob` is the probability of success on each trial.

There are also `r`, `d`, `p`, and `q` functions for `unif` (uniform), `chisq` (Chi-squared), `f` (F), `exp` (Exponential), `pois` (Poisson), `gamma` (Gamma), and other parametric families.

Side Note: `r` stands for random, `d` stands for density, `p` stands for probability (distribution function), and `q` stands for quantile.