Statistical Inference

Neil J. Hatfield

Last Updated: 2023-08-27

Welcome to the latest installment of using R and RStudio. This guide will be looking at statistical inference. As you look through this guide, attempt to calculate the for yourself using the example code. Do this with both the demo data and a new data set (say from HW #1.1).

CAUTION: While I will use my Oreo Demo data here, I make no claims that 1) the methods I demonstrate here are appropriate for my data, 2) that these are exactly what you need to so in HW #1.1, or 3) that you must use these methods in HW #1.1. Feel free to use the following as spring boards/launching platforms to do the analysis you see as appropriate.

I will be focusing on only two elements of statistical significance in this guide: doing null hypothesis tests and confidence intervals. While not a predominate focus, I will demonstrate some data manipulation techniques. Further, while I mention both checking assumptions (which you should do) and practical significance, I will not be demonstrating either of those here. (*Note*: the prior guides provide you will all of the tools you need to evaluate assumptions.)

Getting Started

To get started we need to do two major tasks: load the necessary packages and load our data.

Load Packages

When you start a new session of R, one of the things that you should do is to load the extra packages that will bring new functions and new capabilities. If you don't load the necessary packages, your code might not work.

For basic statistical inference, base R has plenty of functions which will assist you in conducting a hypothesis test and/or constructing confidence intervals. That being said, I will still use a couple of packages: `perm` (you'll need to install this package), and `boot` (should be installed if you used the `checkSetup` function). (Don't forget `tidyverse`.)

```
# Load useful packages ----
packages <- c("tidyverse", "perm", "boot")
lapply(
  X = packages,
  FUN = library,
  character.only = TRUE,
  quietly = TRUE
)
```

Load Data

For this guide I'm going to use two different data sets: the class demo Oreo data set.

```r
# Load Oreo data ----
oreoData <- read.table(
  file = "https://raw.github.com/neilhatfield/STAT461/master/dataFiles/classDemoOreo.dat",
  header = TRUE,
  sep = ","
)

# Recode Type as a factor ----
oreoData$Type <- as.factor(oreoData$Type)
```

I've loaded the data and set Type as a factor. We're almost ready to begin.

Inference Refresher

Before we delve too far into doing statistical inference with R, I believe that doing a quick refresher would be good.

Whether you're doing a Null Hypothesis Significance Test (NHST) or building a confidence interval, you have to rely on some logic.

1) You pick your esti**mator**
2) You come up with a way to describe the long-run behavior of that esti**mator**
3) You calculate the esti**mate** using your data
4) Depends on your approach...
    - NHSTs: You look to see how unusual your esti**mate** is given the long-run behavior of the esti**mator** under the null hypothesis
    - Confidence Intervals: you look to see if your esti**mate** and null hypothesis are consistent with each other.

Remember, the esti**mator** is a function, while the esti**mate** is the output of an esti**mator** for a particular data collection.

You have many choices to make throughout this process. While you can't just pick any set of methods, there are multiple which are valid for any particular statistical research question (SRQ).

Describing Long-Run Behavior

In my opinion, the most important step in the logic process is the second one: developing a way to describe the long-run behavior of your chosen estimator. Ultimately, you are asked to come up with the **sampling distribution for your estimator and samples of size n**. Here are ways that you can do this, in order from fewest to most assumptions:

1) Replication: this is the gold standard but is almost never done. *Why?* Because this is literally where you continuously redo the experiment. And we're not talking just once or twice, but essentially infinitely many times.
2) Simulations: this family of methods while old did not take off until the advent of computers. Doing these methods with anything larger than small data sets by hand is difficult. Computers have helped make these methods much more feasible.
    - Permutation Methods: we can permute data values between the various groups to test whether the grouping factor impacts the response. (We often use combinations rather than permutations to reduce the work load.)
    - Bootstrapping: we sample with replacement from our sample. This lets us treat our original sample as a proxy of the population and attempt to carry out the goals of Replication.

3) Shortcuts: statisticians created these methods to greatly cut down the amount of time needed to get to sampling distributions (hence they are shortcuts). These methods rely on multiple assumptions each and tend to be less robust than other methods

    - Nonparametric Shortcuts: these shortcuts are some times called "distribution free" but this is misnomer. You are still assuming that there is a distribution underpinning your data, you just don't specify exactly which one. These methods often use ranks of data rather than the actual data values.
    - Parametric Shortcuts: these are the methods that most students learn in Intro Stats. These require the most assumptions and the strongest. Here you will make assumptions about the distribution underpinning your data; the most common that your data follow a Gaussian ("normal") distribution with a particular Expected Value (μ) and Variance (σ^2).

I will show you how you can use the above methods with R.

Shortcut Methods

Most of the inference functions in base R relate to parametric and nonparametric shortcuts. Recall that when you are looking at the difference between two groups, you'll want to use the Two Sample t test (parametric) or the Wilcoxon/Mann-Whitney (nonparametric).

Parametric Shortcut

You will need to be sure that you check the assumptions for any shortcut method. I'm going *not* going to model doing so in this document. Rather I'm going to jump to conducting the inferential analysis.

For a t test, you'll make use of the `t.test` function in R:

```
# Generic t Test Function call ----
testOutput1 <- t.test(
  formula = response ~ factor,
  data = dataFrame,
  mu = 0,
  alternative = "two.sided",
  var.equal = FALSE,
  paired = FALSE,
  conf.level = 0.97,
  na.action = "na.omit"
)
```

Argument Details:

- The first two arguments (`formula` and `data`) are where you'll define the model (`response ~ factor`) and name of the data frame where your data live.
- `mu` is where you will set the value under the null hypothesis. For example, if your null hypothesis is that there is no difference between the groups, then `mu = 0` is what you would put. If instead, your null hypothesis was that one group is 4 units or larger than the other group then you would want `mu = 4`.
- The `alternative` is where you'll set whether you're looking at "not equal to" (`"two.sided"`), "Grp1 is less than Grp2" (`"less"`), or "Grp1 is greater than Grp2" (`"greater"`). Be sure that set this correctly as this will affect both the p-value and confidence interval that R produces.
- `var.equal` allows you switch between Student's Pooled t (`TRUE`) and Welch's t (`FALSE`) tests. The difference between the two is whether you make the assumption that the variances of the two groups are essentially the same. I almost always us Welch's as this test works in both cases while Student's does not. (EDA: use the method that applies in more situations)
- The `paired` argument is only applicable if your data are paired (e.g., pre-/post-, etc.). If that is the case then set this to `TRUE`, otherwise just leave this as is.

- If you want R to construct a confidence interval for you, then you'll need to supply the `conf.level` argument and you'll list what level of confidence. In my example, I've opted for a 97% confidence level.
- Remember how for descriptive statistics we needed to use `na.rm = TRUE` in our functions to ensure that we could get results? The `na.action` argument is the equivalent here. Thus, we will want to use `na.action = "na.omit"` to ensure any missing values are dropped before the function runs.

Let's now look at raw output using the my Oreo data.

```
# t Test Example with Raw Output ----
testOutput1 <- t.test(
  formula = Filling.Mass ~ Type,
  data = oreoData,
  mu = 0,
  alternative = "two.sided",
  var.equal = FALSE,
  paired = FALSE,
  conf.level = 0.97,
  na.action = "na.omit"
)

## Get Raw Output ----
testOutput1
```

```
##
##  Welch Two Sample t-test
##
## data:  Filling.Mass by Type
## t = 50.569, df = 47.456, p-value < 2.2e-16
## alternative hypothesis: true difference in means between group Double Stuf and group Regular is not
## 97 percent confidence interval:
##  3.109207 3.397060
## sample estimates:
## mean in group Double Stuf     mean in group Regular
##                  6.456967                  3.203833
```

You'll notice that I stored the results in the object `testOutput1`. This is so I can call up results in easy ways. You can find out the various components in the output by using `names(testOutput1)`: statistic, parameter, p.value, conf.int, estimate, null.value, stderr, alternative, method, data.name.

Here is a quick table of useful values; to call the value you'll use `testOutput1$name`.

`name`	gives	example result
`statistic`	value of t	50.5691675
`parameter`	*Degrees of Freedom*	47.4561972
`p.value`	The p-value	$6.1823909 \times 10^{-43}$
`conf.int`	Your confidence interval	3.1092069, 3.3970598
`estimate`	Group Means	6.4569667, 3.2038333

Notice that these values aren't "pretty". You'll want to use `round` and/or `prettyNum` with them.

Raw Output Caution

Earlier I printed the raw output of the call by having a line that was just `testOutput1` and R then gave a display where each line began with `##`. That is an example of raw output. Raw output is fine for doing something for yourself but is ***inappropriate*** in a analysis report. You should make results look nice.

For a Two Sample t test, I would just report values in my narrative and not worry about a table. For example, I might write something like this:

> Using Welch's t test, we find that $t(47.46) \approx 50.57$); this yielded a p-value < 0.0001. The 97% confidence interval is (3.11, 3.4).

What I typed in my R Markdown file:

> Using Welch's *t* test, we find that *t*(`round(testOutput1$parameter, digits = 2)`) \(\approx\) `round(testOutput1$statistic, digits = 2)`; this yielded a *p*-value `ifelse(test = testOutput1$p.value < 0.0001, yes = "< 0.0001", no = round(testOutput1$p.value, digits = 4))`. The 97% confidence interval is (`round(testOutput1$conf.int, digits = 2)`).

Nonparametric Shortcut

Suppose that your data do not meet the assumptions and you did not transform your data in a way that would let you then meet them. You could choose to make use of a Nonparametric shortcut instead. For a difference between two groups, this would be the Wilcoxon/Mann-Whitney test.

```
# Generic Wilcoxon Test Call ----
testOutput1 <- wilcox.test(
  formula = response ~ factor,
  data = dataFrame,
  mu = 0,
  alternative = "two.sided",
  exact = FALSE,
  correct = TRUE,
  paired = FALSE,
  conf.int = TRUE,
  conf.level = 0.97,
  na.action = "na.omit"
)
```

Argument Details:

- The first two arguments (`formula` and `data`) are where you'll define the model (`response ~ factor`) and name of the data frame where your data live.
- `mu` is where you will set the value under the null hypothesis. For example, if your null hypothesis is that there is no difference between the groups, then `mu = 0` is what you would put. If instead, your null hypothesis was that one group is 4 units or larger than the other group then you would want `mu = 4`.
- The `alternative` is where you'll set whether you're looking at "not equal to" (`"two.sided"`), "Grp1 is less than Grp2" (`"less"`), or "Grp1 is greater than Grp2" (`"greater"`). Be sure that set this correctly as this will affect both the p-value and confidence interval that R produces.
- `exact` allows you to state that you want an an exact p-value (`TRUE`; computationally intensive) or if you're okay with an approximated one (`FALSE`; generally just as good)
- `correct` is if you want to apply the continuity correction for a Gaussian approximation for the p-value.
- The `paired` argument is only applicable if your data are paired (e.g., pre-/post-, etc.). If that is the case then set this to `TRUE`, otherwise just leave this as is.
- If you want confidence intervals, you'll need to set `conf.int = TRUE` and then supply `conf.level`. In my example, I've opted for a 97% confidence level.
- Remember how for descriptive statistics we needed to use `na.rm = TRUE` in our functions to ensure that we could get results? The `na.action` argument is the equivalent here. Thus, we will want to use `na.action = "na.omit"` to ensure any missing values are dropped before the function runs.

Let's now look at raw output using the my Oreo data.

```r
# Wilcoxon Test Example with Raw Output ----
testOutput2 <- wilcox.test(
  formula = Filling.Mass ~ Type,
  data = oreoData,
  mu = 0,
  alternative = "two.sided",
  exact = FALSE,
  correct = TRUE,
  paired = FALSE,
  conf.int = TRUE,
  conf.level = 0.97,
  na.action = "na.omit"
)

## Get Raw Output ----
testOutput2
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  Filling.Mass by Type
## W = 900, p-value = 3.018e-11
## alternative hypothesis: true location shift is not equal to 0
## 97 percent confidence interval:
##  3.082993 3.329990
## sample estimates:
## difference in location
##                3.19898
```

Just as with the Parametric Shortcut, you'll want to store the output as an object and then call what parts you need for your narrative. Here is a quick table of useful values; to call the value you'll use `testOutput2$name`.

`name`	gives	example result
`statistic`	value of W	900
`p.value`	The p-value	$3.0179668 \times 10^{-11}$
`conf.int`	Your confidence interval	3.0829927, 3.3299901
`estimate`	Difference in location	3.1989801

What might you type in R Markdown to produce the following?

> Using the Wilcoxon/Mann-Whitney test, we find that W=900; this yielded a p-value < 0.0001. The 97% confidence interval is (3.083, 3.33). There is approximately 3.2 grams of créme filling between the two groups' location parameters.

Permutation Simulations

You could also use a permutation simulation rather than a shortcut. This is where the `perm` package will come into play.

```
# Generic Permutation Test Function Call ----
permOutput <- perm::permTS(
  formula = response ~ factor,
  data = dataFrame,
  alternative = "two.sided",
  exact = FALSE,
  na.action = "na.omit"
)
```

There are fewer arguments to the Two Sample Permutation test (`permTS`) function that you'll need to worry about.

- The first two arguments (`formula` and `data`) are where you'll define the model (`response ~ factor`) and name of the data frame where your data live.
- `mu` is where you will set the value under the null hypothesis. For example, if your null hypothesis is that there is no difference between the groups, then `mu = 0` is what you would put. If instead, your null hypothesis was that one group is 4 units or larger than the other group then you would want `mu = 4`.
- The `alternative` is where you'll set whether you're looking at "not equal to" (`"two.sided"`), "Grp1 is less than Grp2" (`"less"`), or "Grp1 is greater than Grp2" (`"greater"`). Be sure that set this correctly as this will affect both the *p*-value.
- `exact` allows you to state that you want an an exact p-value (`TRUE`; computationally intensive) or if you're okay with an approximated one (`FALSE`; generally just as good)
- Remember how for descriptive statistics we needed to use `na.rm = TRUE` in our functions to ensure that we could get results? The `na.action` argument is the equivalent here. Thus, we will want to use `na.action = "na.omit"` to ensure any missing values are dropped before the function runs.

Let's now see this in action:

```
# Permutation Test Example and Raw Output ----
permOutput <- perm::permTS(
  formula = Filling.Mass ~ Type,
  data = oreoData,
  alternative = "two.sided",
  exact = FALSE,
  na.action = "na.omit"
)

## Get Raw Output ----
permOutput
```

```
##
##  Permutation Test using Asymptotic Approximation
##
## data:  Filling.Mass by Type
## Z = 7.5955, p-value = 3.064e-14
## alternative hypothesis: true mean Type=Double Stuf - mean Type=Regular is not equal to 0
## sample estimates:
## mean Type=Double Stuf - mean Type=Regular
##                                  3.253133
```

When we Permutation Simulation methods, we often are after just *p*-values, hence why there is no mention of confidence intervals. Just before, you'll want to avoid raw output in your reports.

name	gives	example result
statistic	value of Z	7.5954932
p.value	The p-value	$3.0642155 \times 10^{-14}$
estimate	Difference in location	3.2531333

Bootstrapping

The last method I'm going to show is for Bootstrapping. This method typically results in confidence intervals, rather than p-values and comes with an added bonus: you have much greater flexibility to define what you want your estimator to be.

This extra flexibility does mean that we need to first define our estimator and make sure our data is in wide format.

```
# Change the shape of our data ----
twoColOreos <- unstack(
  x = oreoData,
  form = Filling.Mass ~ Type
)
```

The `unstack` function will take our data frame and re-arrange the columns so that instead of having a column of filling masses and a column of types, we will have two columns of filling masses; the column names will then refer to the type of Oreo. (If you want to go the opposite direction, you would use the function `stack`.)

Let's now come up with an estimator to use. How about we use the ratio of *Sample Medians* to see whether Double Stuf have twice the filling of Regular Oreos?

```
# Define bootstrap estimator ----
myEstimator <- function(df, w) {
  return(
    median(df$Double.Stuf * w) / median(df$Regular * w)
  )
}
```

Our custom estimator, `myEstimator` will allow us to look at our desired ratio. Notice the inclusion of the `w`. We need to place this into our estimator's definition for the bootstrapping call.

```
# Generic Bootstrap Function Call ----
bootOutput <- boot::boot(
  data = dataFrame,
  statistic = functionName,
  stype = "w",
  R = 1000
)
```

Notice that are there are only a few arguments we need to attend to:

- `data`: we will need to use the unstacked version of our data here
- `statistic`: you will need to put the name of your custom function here,
- `stype = "w"` tells the `boot` function to use weights in our custom statistic
- `R` sets how many replications we want to do.

You'll want to save this output into an object as I have done. This is because to get confidence intervals, we will have to pass this output into the function `boot::boot.ci`.

```
# Example Bootstrapping with Raw Output----
bootOutput <- boot::boot(
```

```
  data = twoColOreos,
  statistic = myEstimator,
  stype = "w",
  R = 1000
)

## Form Confidence Intervals ----
bootCI <- boot::boot.ci(
  boot.out = bootOutput,
  conf = 0.97,
  type = c("perc", "bca")
)

## Get Raw Output ----
bootCI
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 994 bootstrap replicates
##
## CALL :
## boot::boot.ci(boot.out = bootOutput, conf = 0.97, type = c("perc",
##     "bca"))
##
## Intervals :
## Level     Percentile            BCa
## 97%   ( 1.898,  2.184 )   ( 1.920,  2.314 )
## Calculations and Intervals on Original Scale
## Some BCa intervals may be unstable
```

The structure of the `bootCI` object is a bit different. In the Shortcuts, we could do `testOutput1$conf.int` and get the intervals. This time, we'll have to a bit more work.

For percentile method confidence intervals (`type = "perc"`), we will need to do the following (`round(bootCI$percent[1,4], digits = 2)`, `round (bootCI$percent[1, 5], digits = 2)`). This would give us: (1.9, 2.18).

For the Bias Corrected and Accelerated intervals (`type = "bca"`), we do pretty much the same thing, just using `bca` instead of `percent`: (`round(bootCI$bca[1,4], digits = 2)`, `round (bootCI$bca[1, 5], digits = 2)`). This would give us: (1.92, 2.31).

Data Transformations

Some times when you are doing inference, you find yourself needing to transform your data. Within `dplyr` (part of `tidyverse`) is the very useful `mutate` function which allows you to create new columns in your data frame. I'm going to show you how to use this function to create several new columns:

- `halfMass`: divide all filling masses by 2
- `logMass`: take the log of all filling masses (that's $\log_e$)
- `doubleR`: double only the Regular Oreo's filling masses

```
# Creating new columns for transformations of data ----
oreoData <- oreoData %>%
  dplyr::mutate(
    halfMass = Filling.Mass / 2,
    logMass = log(Filling.Mass),
    doubleR = ifelse(
      test = Type == "Regular",
      yes = 2 * Filling.Mass,
      no = Filling.Mass
    )
  )

# Display 3 rows of each group, selected randomly
oreoData %>%
  dplyr::group_by(Type) %>%
  dplyr::slice_sample(n = 3)
```

```
## # A tibble: 6 x 5
## # Groups:   Type [2]
##   Filling.Mass Type        halfMass logMass doubleR
##          <dbl> <fct>          <dbl>   <dbl>   <dbl>
## 1         6.33 Double Stuf     3.17   1.85     6.33
## 2         6.16 Double Stuf     3.08   1.82     6.16
## 3         6.29 Double Stuf     3.14   1.84     6.29
## 4         2.29 Regular         1.15   0.829    4.58
## 5         3.28 Regular         1.64   1.19     6.55
## 6         3.27 Regular         1.64   1.19     6.55
```

Final Remarks

This guide served two purposes: demonstrate how you can use R to calculate the values of various descriptive/incisive statistics and how to prepare tables of values in a document. Again, the best way to improve your skills is by practicing.

- Motor Trend Car Road Tests—access with `data(mtcars)`
- Weight versus Age of Chicks on Different Diets—access with `data(ChickWeight)`
- Effectiveness of Insect Sprays—access with `data(InsectSprays)`
- The Iris data set—access with `data(iris)`
- Palmer Penguin Data—install the `palmerpenguins` package first, then access with `palmerpenguins::penguins`

This concludes this guide to descriptive statistics in R/RStudio.

Code Appendix

```r
# Setting Document Options ----
knitr::opts_chunk$set(
  echo = FALSE,
  warning = FALSE,
  message = FALSE,
  fig.align = "center"
)

# Load useful packages ----
packages <- c("tidyverse", "perm", "boot")
lapply(
  X = packages,
  FUN = library,
  character.only = TRUE,
  quietly = TRUE
)

# Load Oreo data ----
oreoData <- read.table(
  file = "https://raw.github.com/neilhatfield/STAT461/master/dataFiles/classDemoOreo.dat",
  header = TRUE,
  sep = ","
)

# Recode Type as a factor ----
oreoData$Type <- as.factor(oreoData$Type)

# Generic t Test Function call ----
testOutput1 <- t.test(
  formula = response ~ factor,
  data = dataFrame,
  mu = 0,
  alternative = "two.sided",
  var.equal = FALSE,
  paired = FALSE,
  conf.level = 0.97,
  na.action = "na.omit"
)

# t Test Example with Raw Output ----
testOutput1 <- t.test(
  formula = Filling.Mass ~ Type,
  data = oreoData,
  mu = 0,
  alternative = "two.sided",
  var.equal = FALSE,
  paired = FALSE,
  conf.level = 0.97,
  na.action = "na.omit"
)

## Get Raw Output ----
```

```r
testOutput1

# Generic Wilcoxon Test Call ----
testOutput1 <- wilcox.test(
  formula = response ~ factor,
  data = dataFrame,
  mu = 0,
  alternative = "two.sided",
  exact = FALSE,
  correct = TRUE,
  paired = FALSE,
  conf.int = TRUE,
  conf.level = 0.97,
  na.action = "na.omit"
)

# Wilcoxon Test Example with Raw Output ----
testOutput2 <- wilcox.test(
  formula = Filling.Mass ~ Type,
  data = oreoData,
  mu = 0,
  alternative = "two.sided",
  exact = FALSE,
  correct = TRUE,
  paired = FALSE,
  conf.int = TRUE,
  conf.level = 0.97,
  na.action = "na.omit"
)

## Get Raw Output ----
testOutput2

# Generic Permutation Test Function Call ----
permOutput <- perm::permTS(
  formula = response ~ factor,
  data = dataFrame,
  alternative = "two.sided",
  exact = FALSE,
  na.action = "na.omit"
)

# Permutation Test Example and Raw Output ----
permOutput <- perm::permTS(
  formula = Filling.Mass ~ Type,
  data = oreoData,
  alternative = "two.sided",
  exact = FALSE,
  na.action = "na.omit"
)

## Get Raw Output ----
permOutput
```

```r
# Change the shape of our data ----
twoColOreos <- unstack(
  x = oreoData,
  form = Filling.Mass ~ Type
)
# Define bootstrap estimator ----
myEstimator <- function(df, w) {
  return(
    median(df$Double.Stuf * w) / median(df$Regular * w)
  )
}

# Generic Bootstrap Function Call ----
bootOutput <- boot::boot(
  data = dataFrame,
  statistic = functionName,
  stype = "w",
  R = 1000
)
# Example Bootstrapping with Raw Output----
bootOutput <- boot::boot(
  data = twoColOreos,
  statistic = myEstimator,
  stype = "w",
  R = 1000
)

## Form Confidence Intervals ----
bootCI <- boot::boot.ci(
  boot.out = bootOutput,
  conf = 0.97,
  type = c("perc", "bca")
)

## Get Raw Output ----
bootCI

# Creating new columns for transformations of data ----
oreoData <- oreoData %>%
  dplyr::mutate(
    halfMass = Filling.Mass / 2,
    logMass = log(Filling.Mass),
    doubleR = ifelse(
      test = Type == "Regular",
      yes = 2 * Filling.Mass,
      no = Filling.Mass
    )
  )

# Display 3 rows of each group, selected randomly
oreoData %>%
  dplyr::group_by(Type) %>%
  dplyr::slice_sample(n = 3)
```