Diagnostics for two sample data using tidyverse and ggplot2

Jingtao

2024-09-24

## Libraries

library(dplyr)

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
## filter, lag

## The following objects are masked from 'package:base':
##
## intersect, setdiff, setequal, union

library(ggplot2)

## read data from the .xslx sheet

The non-tidyverse document describes how to read .xlsx files into an R data frame. That code copied here so this file is a complete package.

library(readxl)
hb <- read\_excel("../data/hamburger.xlsx")

## group specific standard deviation

We saw code like this in the week 1 tidyverse files (those with V2 in their name). In week one, we used code like this to compute means and medians for each group of observations. Now we want to compute sd’s.

Reminder (from week 1 ): %>% is the pipe operator. The data frame on the left is the data set used by the commands on the right. So hb %>% group\_by() feeds the hb data frame into the group\_by() function,

group\_by() tells R that any operations that follow are to be done separately for each group. You can specify multiple variables inside group\_by(), e.g., group\_by(a, b) where the unique combinations of a and b define groups.

summarize() takes groups of observations and computes summary statistics. Previously, that summary statistic was either the mean or median. Now we want the sd. sd() is the R function that computes sd’s. If you want variances, use the var() function. sd() or var() needs to know which variable you need the summaries for. That variable name goes inside the (). So sd(cfu) computes the sd of the cfu variable. This is done separately for each group (treatment) because of the preceding group\_by(). The result is stored in the variable name given on the left of the =. (Yes, = is needed, not <-). You can compute multiple summary statistics, just separate each bit by commas inside the summarize(). E.g., summarize(sd = sd(cfu), nobs = n(), mean = mean(cfu) ) will compute 3 variables for each group: the sd of cfu, the sample size (nobs) and the mean (mean). n() doesn’t take any variable names because it doesn’t need it.

Note: spreading a sequence of commands over multiple lines aids readability. It is not crucial. Except that you need to make sure R “knows” there is more to come. If your code looked like this:

hb\_sd <- hb %>%

group\_by(treatment)

I.e., lacks the %>% at the end of the second line, R will think that the sequence (pipe, group\_by) is finished. Need to pipe that result into the 3rd line. The summarize( … ) can be spread over multiple lines because summarize isn’t finished until the ) that ends summarize( … ).

The result is stored in the data frame called hb\_sd. The last line prints that data frame.

hb\_sd <- hb %>%
 group\_by(treatment) %>%
 summarize(
 sd = sd(cfu))

hb\_sd

## # A tibble: 2 × 2
## treatment sd
## <chr> <dbl>
## 1 active 0.0717
## 2 control 0.885

## using mutate to do log-trans

Similar code to the previous except: don’t need to group\_by() because we want to do the same operation to every observation.

mutate( … ) creates new variables or modifies current variables. The right-hand side specifies the calculation you want to do. This can be any R operation or function. This has to be something that accepts one row of data and returns a single value. The left-hand side names the variable to store that result. You can overwrite the original variable if you really want to do that.

The ggplot() call plots logcfu on the Y axis against cfu on the X axis. This demonstrates what a log transformation does to values.

The call to ggplot() specifies the data set to use: data= and the variables to use: aes(). The most common aesthetic has an x and a y variable. The ggplot() function sets up a plot but doesn’t draw anything. + geom\_point() draws points using the variable names specified in aes().

Plots can be saved in variables (actually objects). The plot gets drawn only when the plot object is printed. The next block of code demonstrates saving a plot then adding addition features to that plot before showing everything.

hb <- hb %>%
 mutate(
 logcfu = log(cfu))

head(hb)

## # A tibble: 6 × 3
## treatment cfu logcfu
## <chr> <dbl> <dbl>
## 1 control 0.17 -1.77
## 2 control 0.35 -1.05
## 3 control 0.22 -1.51
## 4 control 2.5 0.916
## 5 control 0.53 -0.635
## 6 control 0.57 -0.562

ggplot(data = hb, aes(x = cfu, y = logcfu)) +
 geom\_point()



## the residuals vs predicted values plot

The easiest way to get residuals and predicted values is by using the lm() function to do the t-test. lm() will be our workhorse for most of the semester. It fits a huge number of models. But, its output isn’t as easily interpretable as t.test(). We’ll talk a lot about interpreting lm() output when we need it. For now, we’re just using it to get residuals and predicted values.

lm() requires that you specify a model. For a t-test, this is exactly the same as the model used in t.test(): response variable on the left-hand side of the tilde (~ character) and treatment variable on the right-hand side. The code saves that fitted model so we can subsequently extract residuals and predicted values.

resid(hb.lm) extracts residuals from the specified model

predict(hb.lm) extracts predicted values.

ggplot() sets up the plot. The code below explicitly species the data frame to use (data = ) and the aesthetic to use (aes = ). The first two arguments to ggplot() are data= and mapping=, so you will often see those omitted. ggplot(hb, aes(x = predict(hb.lm), y = resid(hb.lm))) will do exactly the same thing.

The second line adds two things to the blank plot:

geom\_point() adds points to the plot

geom\_hline() adds a horizontal line at y = 0 to the plot. The additional arguments configure that line by changing the line type and color. Those arguments are general and can be used for any “add a feature” command. So geom\_point(color = ‘blue’) will color the points blue.

hb.lm <- lm(cfu~treatment, data=hb)

p1 <- ggplot(data = hb,
 mapping = aes(
 x = predict(hb.lm),
 y = resid(hb.lm)))
p1 + geom\_point() + geom\_hline(yintercept = 0, linetype = "dashed", color = "red")



## Normal qqplot of residuals

The stat\_qq() draws a normal quantile-quantile plot. It requires the sample of values to evaluate, specified in the aes() by sample =. To draw a QQ plot of residuals, you need sample = resid(hb.lm). You don’t specify x= and y=. The stat\_qq() feature will automatically calculate the X axis plotting positions (the theoretical quantiles for a standard normal distribution).

The stat\_qq\_line() feature adds a line to the QQ plot.

p2 <- ggplot(hb, aes(sample = resid(hb.lm))) + stat\_qq()
p2 + stat\_qq\_line(linetype = "dashed", color = "blue")

