STATS205-HW1

Isaac Kleisle-Murphy

4/2/2022

2.8.1

2.8.1. Verify, via simulation, the level of the `wilcox.test` when sampling from a standard normal distribution. Use $n = 30$ and levels of $\alpha = 0.1, 0.05, 0.01$. Based on the resulting estimate of α, the empirical level, obtain a 95% confidence interval for α.

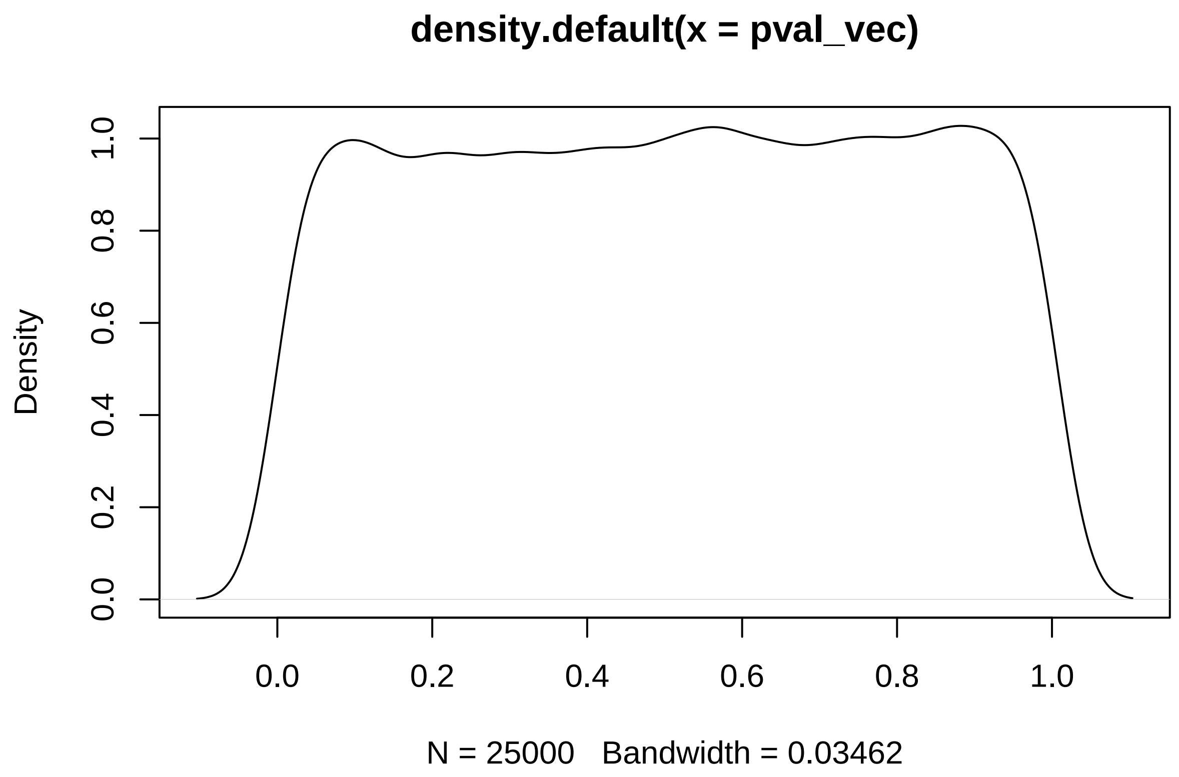
First, we write a simulator function to perform a single experiment under these conditions.

```r
sim_2_8_1_single <- function(){
  zvec = rnorm(30)
  wtest = wilcox.test(zvec)
  pval = wtest$p.value
  pval
}
```

We then write a function to do this $S = 25000$ times

```r
sim_2_8_1_mc <- function(S=25000){
  sapply(1:S, function(i) sim_2_8_1_single())
}

set.seed(2800)
pval_vec = sim_2_8_1_mc()
plot(density(pval_vec));
```



As expected, p-values appear to be distributed uniformly, indicating `wilcox.test` is behaving appropriately. Moreover, of the $S = 25000$ draws, we see that

```
cat("Percentage of P-Vals <= .10: ", mean(pval_vec <= .1), "\n")
```

```
## Percentage of P-Vals <= .10:  0.09772
```

```
cat("Percentage of P-Vals <= .05: ", mean(pval_vec <= .05), "\n")
```

```
## Percentage of P-Vals <= .05:  0.05132
```

```
cat("Percentage of P-Vals <= .01: ", mean(pval_vec <= .01), "\n")
```

```
## Percentage of P-Vals <= .01:  0.01004
```

which suggests appropriate calibration – i.e. we're incorrectly rejecting at the level we want to. This further "verifies the level of `wilcox.test`".

If we want confidence intervals for the α (I prefer to think of it as CIs for a RV that is an indicator of if $p = g(X)$ for the g that is effectively `wilcox.test()$p.value` is less than α, i.e. $\tilde{X} = \delta(g(X) < \alpha)$), we can repeat the above via Monte Carlo (10,000 samples of the above, for example). However, this takes a long time to run, so for simplicity we revert to a Wald CI, using the sampled mean of $S^{-1} \sum_{i=1}^{S} \delta(g(X_i) < \alpha)$:

```
compute_wald <- function(x, alpha){
  phat = mean(x <= alpha)
  sigma = sqrt(phat * (1 - phat) / length(x))
  c(
    phat - qnorm(.975) * sigma,
    phat + qnorm(.975) * sigma
  )
}
```

For $\alpha = .1$:

```
compute_wald(pval_vec, .1)
```

```
## [1] 0.09403921 0.10140079
```

For $\alpha = .05$:

```
compute_wald(pval_vec, .05)
```

```
## [1] 0.04858485 0.05405515
```

For $\alpha = .01$:

```
compute_wald(pval_vec, .01)
```

```
## [1] 0.008804184 0.011275816
```

These too appear appropriate. I'd call things verified.

2.8.7

Write an R function which computes the sign analysis. For example, the following commands compute the statistic S^+, assuming that the sample is in the vector x.

Following instructions on pp. 16-17, we have a function that tallies up positive signs, sets up a corresponding binomial problem, and computes the appropriate p-value. A demo run on 1,000 randomly drawn unit normals is provided at the bottom for demonstrative purposes.

```
sign_test_single <- function(x, alternative="not.equal"){
  # drop zeros
  x_ = x[x!=0]
  # count positives
  s_plus = sum(x_ > 0)
  # num trials
  n = length(x_)
  # compute binomial CDFs
  p_left_tail = pbinom(s_plus, n, .5, lower.tail = T)
  p_right_tail = pbinom(s_plus - 1, n, .5, lower.tail = F)
  # definitions here: https://en.wikipedia.org/wiki/Sign_test
  if (alternative == "not.equal"){
```

```r
    return(
      min(
        2 * min(p_left_tail, p_right_tail),
        1 # note that if at 1/2 exactly, you spill over, so round to 1
      )
    )
  }else if(alternative == "greater"){
    return(p_right_tail)
  }else{
    return(p_left_tail)
  }
}

### note the implementation of conf.int, here by bootstrap
sign_test <- function(x, alternative="not.equal", conf.int=NULL){
  # doing a single sign test
  if(is.null(conf.int)){
    return(
      sign_test_single(x=x, alternative=alterantive)
    )
  }else{
    # otherwise, bootstrap 25000 times
    pval_boot = rep(NA, 25000)
    for (i in 1:25000){
      pval_boot[i] = sign_test_single(
        sample(x, length(x), replace=T), # bootstrap sample
        alternative=alternative # pass args
      )
    }
    # return CI
    return(
      quantile(
        pval_boot,
        c(
          (1 - conf.int) / 2,
          1 - (1 - conf.int) / 2
        )
      )
    )
  }
}

### demo run
set.seed(123)
sign_test(
  rnorm(1000),
  conf.int=.95
)
```

```
##       2.5%      97.5%
## 0.02092711 0.97477498
```

Given p-values should be uniform, this aligns with general expectations.

2.8.9

The nursery data is updated:

```
school = c(82, 69, 73, 43, 58, 56, 76, 65)
home = c(63, 42, 74, 37, 51, 43, 80, 82) # <- last element changed: 62 to 82
response = school - home
```

And tests are re-run:

```
wilcox.test(response, alternative="greater", conf.int=T)
```

```
##
##  Wilcoxon signed rank exact test
##
## data:  response
## V = 27, p-value = 0.125
## alternative hypothesis: true location is greater than 0
## 95 percent confidence interval:
##   -4 Inf
## sample estimates:
## (pseudo)median
##           6.25
```

```
t.test(response, alternative="greater", conf.int=T)
```

```
##
##  One Sample t-test
##
## data:  response
## t = 1.2789, df = 7, p-value = 0.1209
## alternative hypothesis: true mean is greater than 0
## 95 percent confidence interval:
##  -3.00903      Inf
## sample estimates:
## mean of x
##      6.25
```

The high p-values for both give us little motivation to reject, and proclaim any meaningful difference greater than zero.

2.8.10

i.)

First, we initialize a function to draw from a contaminated normal:

```
rcontamnorm <- function(n, p, mean=0, sd=1){
    # draw actual normal
    x = rnorm(n, mean, sd)
    # draw Z
    z = rnorm(n, mean, 1)
    # draw mixing
    epsilon = rbinom(n, 1, p)
    epsilon * z + (1 - epsilon) * x
}
```

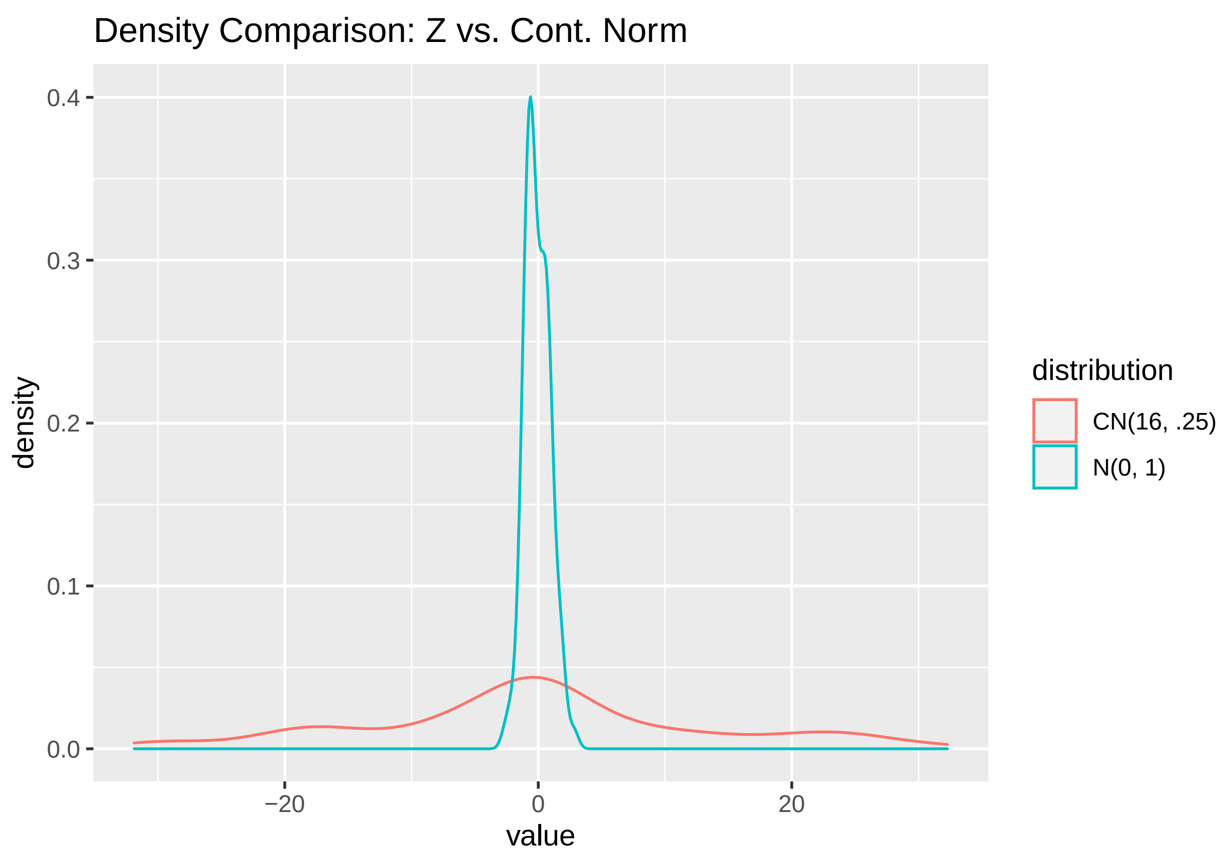
ii.)

Then, we obtain the requisite samples:

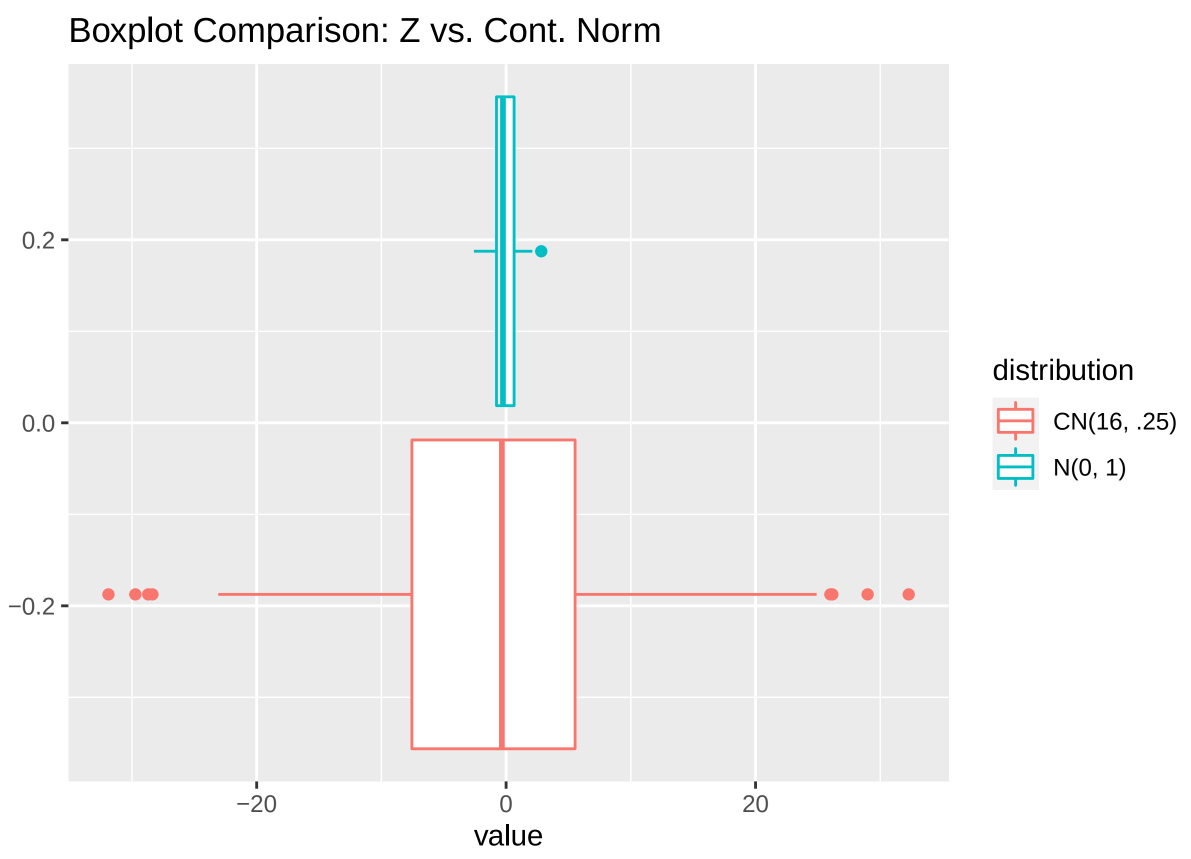
```
set.seed(345)
norm_samples = rnorm(100)
cnorm_samples = rcontamnorm(100, .25, 0, 16)
```

And then visualize:

```
ggplot(
  data.frame(
    value = c(norm_samples, cnorm_samples),
    distribution = c(rep("N(0, 1)", 100), rep("CN(16, .25)", 100))
  ),
  aes(
    x=value,
    color=distribution
  )
) +
  geom_density() +
  labs(x="value", y="density", title="Density Comparison: Z vs. Cont. Norm")
```



```
ggplot(
  data.frame(
    value = c(norm_samples, cnorm_samples),
    distribution = c(rep("N(0, 1)", 100), rep("CN(16, .25)", 100))
  ),
  aes(
    x=value,
    color=distribution
  )
) +
  geom_boxplot() +
  labs(x="value", y="", title="Boxplot Comparison: Z vs. Cont. Norm")
```



Unsurprisingly, we see that the contaminated model has a much fatter tail than does the unit normal. This is because 25 percent of the time, the contaminated normal is sampling from a normal of much higher variance (16^2) and 75 percent of the time it is sampling from the unit normal (with smaller, equal-to-one variance); by contrast, the unit normal is always sampling from smaller variance, hence the lack of tail. As both are centered on zero, the result is unimodal about zero; if we introduced a location parameter to one distribution (e.g. the unit normal standalone, or into one or both of the normals in the contaminated normal), this could easily change.

3.7.1

First, load the data

```
player_pro = Rfit::baseball

pitcher_heights = player_pro %>%
  filter(field == 0) %>%
  pull(height)

batter_heights = player_pro %>%
  filter(field == 1) %>%
  pull(height)
```

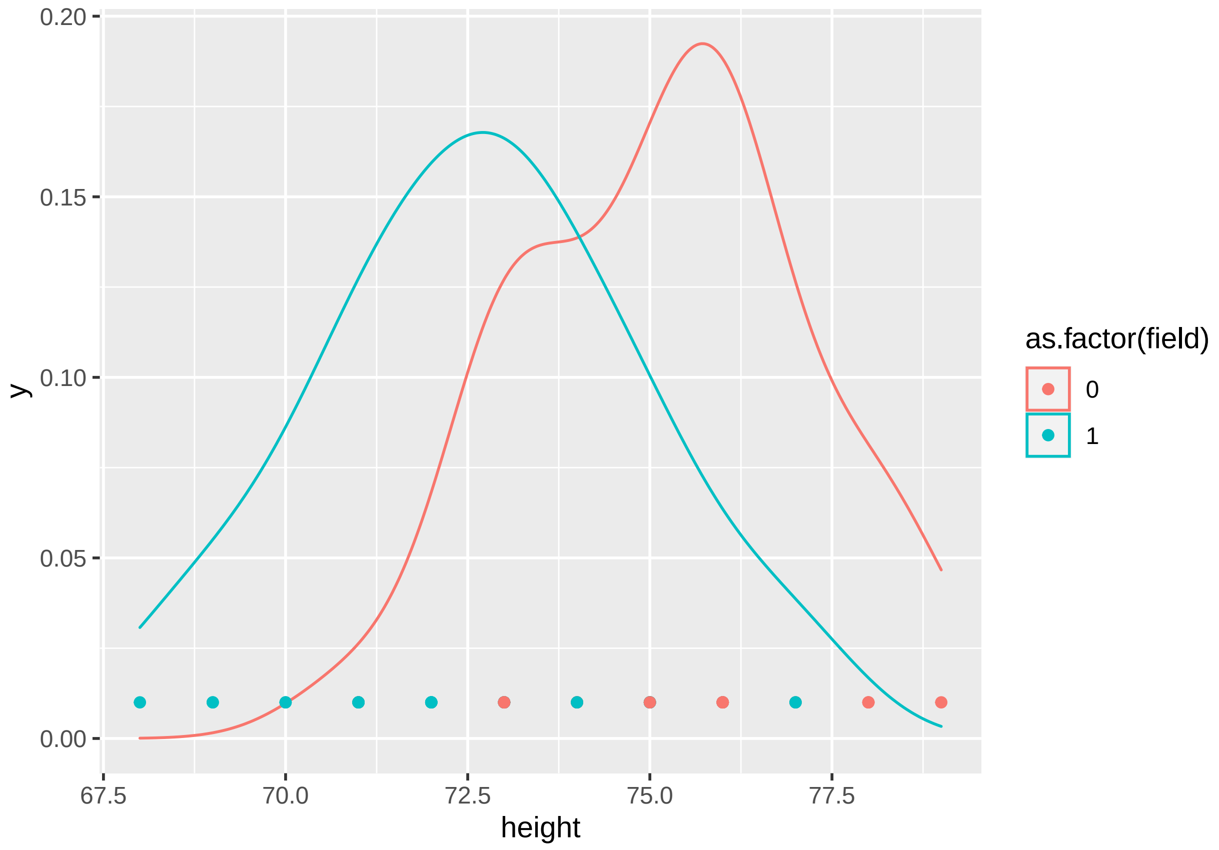
Given the difference in sample size, a Wilcoxon rank-based analysis for two-sample location problem seems ideal here. Among the assumptions we can plausibly (and necessarily, for the test) make:

- Independence of samples.

- Height is continuous.
- Height could reasonably be exponential family, and hence follow a location model.

We first visualize heights/weights. By the looks of it, pitchers appear taller (perhaps selection bias from improved release extension and/or arm angle, middle-IFs being short):

```
ggplot(player_pro %>% mutate(y = 0.01), aes(x=height, color=as.factor(field))) +
  geom_density() +
  geom_point(aes(x=height, y=y, color=as.factor(field)))
```



Proceeding to the test, we have

```
wilcox.test(pitcher_heights, batter_heights, alternative = "two.sided", exact=F)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  pitcher_heights and batter_heights
## W = 686.5, p-value = 7.412e-05
## alternative hypothesis: true location shift is not equal to 0
```

Indeed, the location param appears to be non-zero (unsurprisingly), presumably due to pitchers being taller.

Unfortunately, `npsm` appears to have been removed from the CRAN, so `rank.test` is unavailable. However, if it were available, code would look like:

```r
rank.test(pitcher_heights, batter_heights)
```

However, as set forth on pp. 55-56, we can compute by hand via normal approximation/asymptotics, i.e, and get a p-value (level $\alpha = .05$ assumed). Specifically, following the procedures outlined on pp.55-56, we have:

```r
# rank Y within the combined [X, Y]
xy_combined = data.frame(
  xy = c(batter_heights, pitcher_heights),
  is_pitcher = c(rep(0, length(batter_heights)), rep(1, length(pitcher_heights)))
) %>%
  arrange(xy) %>%
  mutate(rank = row_number())

# compute T by summing relevant ranks
T_stat = xy_combined %>%
  filter(is_pitcher == 1) %>%
  pull(rank) %>%
  sum()

# dims
n2 = length(pitcher_heights)
n1 = length(batter_heights)
n = nrow(xy_combined)

# null mean
u0 = n2 * (n + 1) / 2
# null sd
sigma0 = sqrt(
  n1 * n2 * (n + 1) / 12
)

Z_stat = (T_stat - u0) / sigma0

# two-sided p-value ("unequal", as stated in problem)
2 * pnorm(.975, Z_stat)
```

```
## [1] 0.0003505615
```

Again, this p-value shakes out well less than $\alpha = .05$, adding further support for our rejection above.

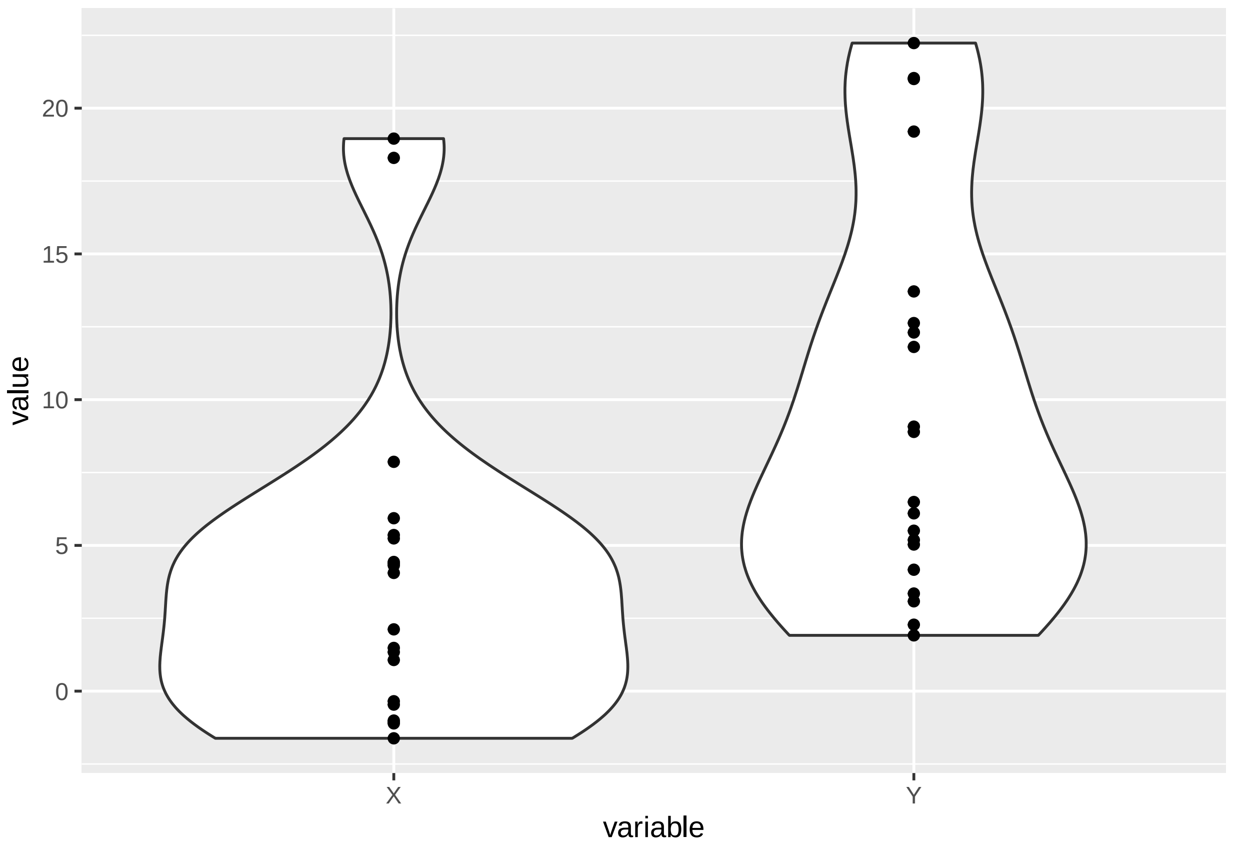
3.7.4

a. / b.)

We draw the samples, and visualize the dotplots

```r
set.seed(2022)
x = rf(20, 1, .25) %>% log()
y = rf(20, 1, .25) %>% log() + 7.

ggplot(
  data.frame(
    value=c(x, y),
    variable=c(
      rep("X", length(x)),
      rep("Y", length(y))
    )
  ),
  aes(y=value, x=variable)) +
  geom_violin() +
  geom_point()
```



c.)

OLS By 3.23, we have $Z = \langle X_1, \ldots, X_{n_X}, Y_1, \ldots, Y_{n_Y} \rangle$ and model $Z_i = \alpha + c_i \cdot \delta + \epsilon_i$, which straightforwardly returns $\hat{\Delta} = \bar{Y} - \bar{X}$. That is,

```
mean(y) - mean(x)
```

```
## [1] 5.788008
```

Further, as set forth on p.61, the Wilcoxon score test will give us an estimate that is the median of all pairwise differences (Hodges-Lehmann), namely

```
x_ = lapply(1:length(y), function(i) x) %>% do.call("rbind", .)
y_ = lapply(1:length(y), function(i) y) %>% do.call("cbind", .)
median(as.numeric(y_ - x_))
```

```
## [1] 4.986485
```

note that by using Hodges-Lehmann, I'm really using the pseudo-median of the Wilcox test. Per `DescTools`, `wilcox.test` uses sampling to compute it's estimate of the median (https://search.r-project.org/CRAN/refmans/DescTools/html/HodgesLehmann.html).

Further, if we instead pursue rank-based normal scores, we have

```
z <- c(x, y)
ci = c(rep(0, length(x)), rep(1, length(y)))
coef(summary(rfit(z ~ ci, scores=nscores)))
```

```
##             Estimate Std. Error  t.value     p.value
## (Intercept) 3.087391   1.385188 2.228860 0.031814133
## ci          4.627127   1.522454 3.039256 0.004276699
```

4.62. All are roughly in the same neighborhood for $\hat{\delta}$

Lastly, bent scores go as follows:

```
bscores = bentscores1
bscores@param = c(.75, -2, 1)
fit = rfit(z ~ ci, scores=bscores)
summary(fit)
```

```
## Call:
## rfit.default(formula = z ~ ci, scores = bscores)
##
## Coefficients:
##             Estimate Std. Error t.value  p.value
## (Intercept)   3.0874     1.3767  2.2427 0.030831 *
## ci            4.7026     1.4272  3.2950 0.002137 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Multiple R-squared (Robust): 0.2343608
## Reduction in Dispersion Test: 11.63173 p-value: 0.00155
```

I.e. an estimate of $\hat{\delta} = 4.7026$.

3.7.5

We perform the steps above, albeit 10,000 times over. Conveniently, for the purposes of MSE, we know $\delta = 7$. We have:

```r
set.seed(1234)
delta_estimates = lapply(1:2000, function(i){
  ### progress updates
  if ((i %% 10000) == 0){
    cat("Iter: ", i, "/10000\n")
  }
  ### take draws
  x = rf(20, 1, .25) %>% log()
  y = rf(20, 1, .25) %>% log() + 7.

  ### OLS
  delta_ols = mean(y) - mean(x)

  ### Hodges-Lehmann
  x_ = lapply(1:length(y), function(i) x) %>% do.call("rbind", .)
  y_ = lapply(1:length(y), function(i) y) %>% do.call("cbind", .)
  delta_hl = median(as.numeric(y_ - x_))

  ### bentscores: note that I"m using same parameterization
  ### as given in book example, with a bend at .75
  bscores = bentscores1
  bscores@param = c(.75, -2, 1)
  fit = rfit(z ~ ci, scores=bscores)
  delta_bent = as.numeric(coef(fit)[2])

  ### compile estimates
  c(delta_ols, delta_hl, delta_bent)
  }
) %>%
  do.call("rbind", .)

mses = colMeans(
  (delta_estimates - 7) ^ 2
)

cat("ARE(OLS, Wilcox): ", mses[1] / mses[2], "\n")
```

```
## ARE(OLS, Wilcox):  1.621853
```

```r
cat("ARE(Wilcox, bentscores1): ", mses[2] / mses[3], "\n")
```

```
## ARE(Wilcox, bentscores1):  0.7526738
```

```r
cat("ARE(OLS, bentscores1): ", mses[1] / mses[3], "\n\n")
```

```
## ARE(OLS, bentscores1):  1.220726
```

```r
cat("MSES (OLS/Wilcox/bentscores1): ", mses)
```

```
## MSES (OLS/Wilcox/bentscores1):  6.443301 3.972802 5.278252
```

Above, we see that the Wilcoxon performs the `bentscores1` and `OLS` methods; in part this is unsurprising: as set forth on pp. 71-72, OLS is not robust, and the Wilcoxon estimator "estimator has a substantial gain in efficiency over the LS estimator for error distributions with heavier tails than the normal distribution," which is the case here (p. 72). Further, while the Wilcoxon outperforms the `bentscores1`, it is not entirely clear if this is a data-specific behavior (as implied in the textbook example: "Hence, the Wilcoxon analysis is about 16% less precise than the bentscore analysis for this dataset") or more generally a tuning issue (no code or background for how the `u=.75` parameterization was determined, and while I used it as a default, it may not be best here.), or both. However, provisionally, Wilcoxon wins out.

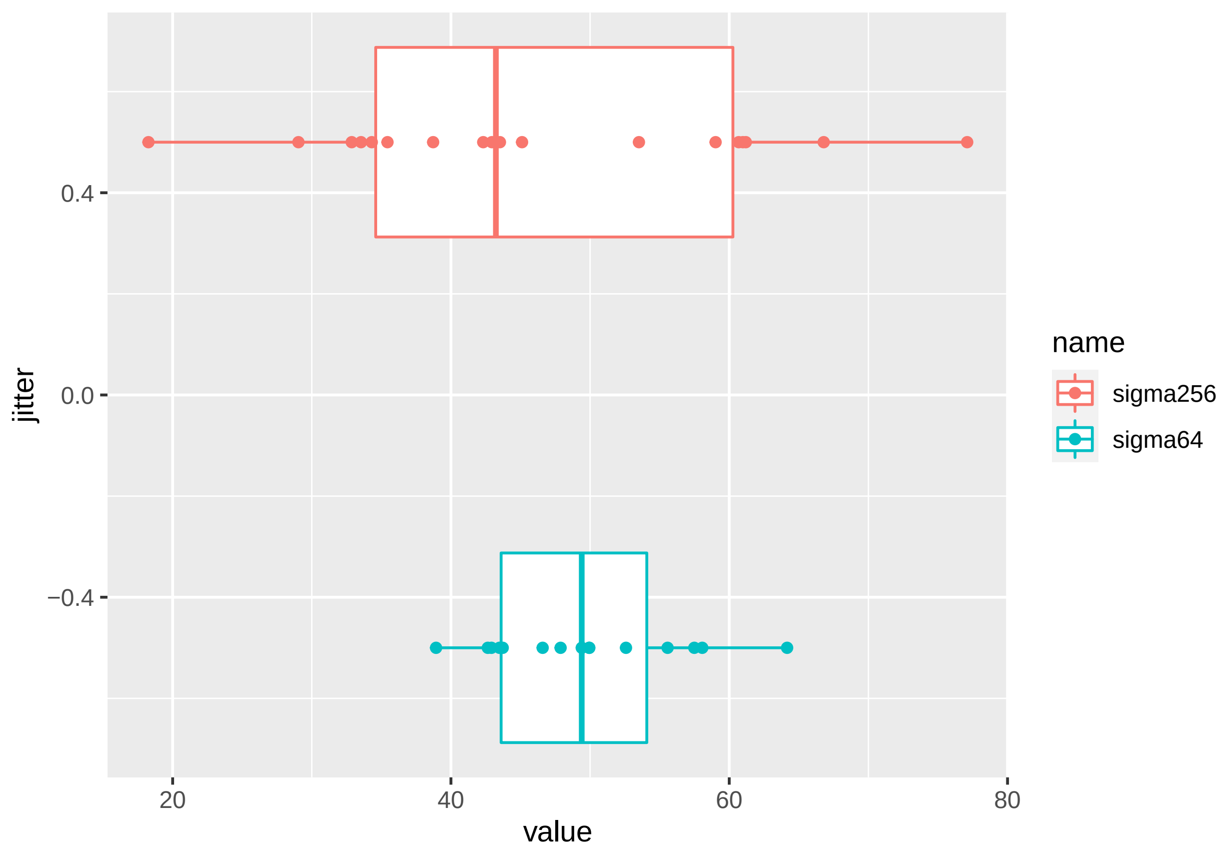
OLS performs worst by this analysis; as set forth in class and in the textbook, it is not robust and hence may be compromised by outliers.

3.7.11

Load the data and make the boxplot

```
# ~ N(50, 64)
x = c(43.72, 58.06, 55.57, 57.49, 64.16, 43.49,
      49.94, 49.94, 52.58, 49.40, 38.93, 42.65,
      42.91, 46.59, 47.88)
# ~ N(50, 256)
y = c(77.10, 42.94, 42.32, 53.51, 66.79, 18.26,
      38.72, 29.04, 33.54, 61.19, 32.87, 60.98,
      59.02, 60.68, 45.11, 43.52, 35.44, 34.31)

ggplot(
  data.frame(
    value=c(x, y),
    name=c(
      rep("sigma64", length(x)),
      rep("sigma256", length(y))
    ),
    jitter=c(
      rep(-.5, length(x)),
      rep(.5, length(y))
    )
  ),
  aes(x = value, y=jitter, color=name)
) +
  geom_boxplot() +
  geom_point(aes(x = value, y=jitter, color=name))
```



Unsurprisingly, the $N(50, 256)$ takes a bigger spread; that's how the Gaussian works. We then do a Fligner-Killen test.

```
### struggle to get npsm
# install.packages(pkgs="~/Downloads/npsm_0.5.1.tar.gz", type="source", repos=NULL)
library(npsm)

fk.test(x, y)
```

```
## statistic =  2.496442 , p-value =  0.01254462
## 95  percent confidence interval:
## 1.310775 4.017257
## Estimate: 2.294715
```

The test returns a p-value of .01, so at $\alpha = .05$ we're likely to reject, and conclude that y carries larger variability. The estimate of this scale is roughly 2.5, as wrapped in a 95% CI of (1.31, 4.02). Importantly, 1 (i.e. no difference in scale) falls outside of this CI (as occurs under the duality of the hypothesis test) – again indicating difference in scale. That is, the "no difference in scale" outcome was not trapped by the CI (consistent w/ duality of hypothesis test); however, the true value of 2.0 was successfully trapped.