Chapter 6 examples

Brooke Anderson

3/12/2020

```
library(tidyverse)
library(purrr)
```

Coin flipping example

```
num_flips <- 100
p_heads <- 0.6

coin_flips <- tibble(outcome = sample(c("H", "T"),
                                      size = num_flips,
                                      replace = TRUE,
                                      prob = c(p_heads, 1 - p_heads)))
coin_flips %>%
  slice(1:6)
```

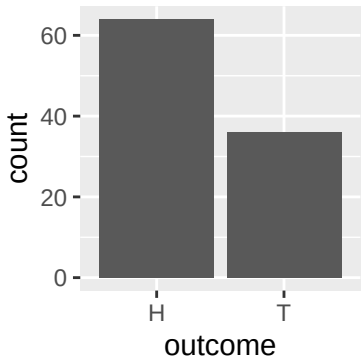
```
## # A tibble: 6 x 1
##   outcome
##   <chr>
## 1 H
## 2 H
## 3 H
## 4 H
## 5 H
## 6 H
```

```
coin_flips %>%
  group_by(outcome) %>%
  count()
```

```
## # A tibble: 2 x 2
## # Groups:   outcome [2]
##   outcome     n
##   <chr>   <int>
## 1 H          64
## 2 T          36
```

You can create a simple bar chart of the outcome if you want, too:

```
coin_flips %>%
  ggplot(aes(x = outcome)) +
  geom_bar()
```



Next, we can look at how the outcome of our "experiment" compares to what we might expect under the null hypothesis over a lot of similar experiments if that null hypothesis were true:

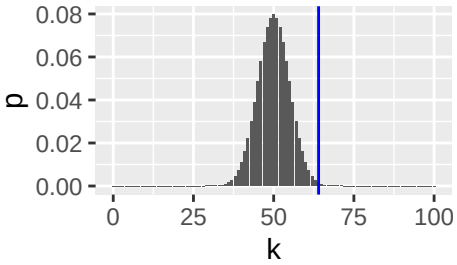
```
# Get the observed number of heads in our "experiment"
# Because TRUE is saved as a 1 and FALSE as a 0, we can
# count quickly using `sum` on the logical vector testing
# for heads in the outcome. The parentheses print out the
# value of `n_heads_observed` right after it's assigned
# (saves me typing it out again to print it).
(n_heads_observed <- sum(coin_flips$outcome == "H"))
```

```
## [1] 64
```

```
# Next, get the probability of every possible outcome (0 heads,
# 1 head, all the way up to all heads). I'm using `map` here from
# the `purrr` package. It makes it easy to apply a function
# across all the values of another column in the dataframe.
# By using the `*_dbl` version of the `map` function, I'm forcing
# the output to be in the format of a vector of numbers ("doubles":
# "dbl"). Otherwise, it would give me the output as a "list-column",
# which is more complex than we need here.
binom_density <- tibble(k = 0:num_flips) %>%  # All possible outcomes
  mutate(p = map_dbl(k, ~ dbinom(.x, size = num_flips, prob = 0.5)))

binom_density %>%
  slice(1:5)
```

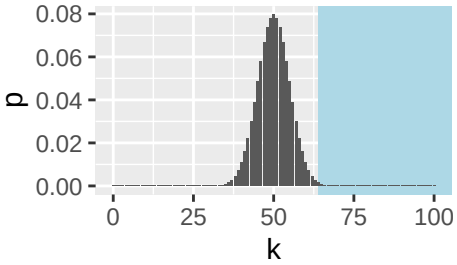
```
## # A tibble: 5 x 2
##       k        p
##   <int>    <dbl>
## 1     0 7.89e-31
## 2     1 7.89e-29
## 3     2 3.90e-27
## 4     3 1.28e-25
## 5     4 3.09e-24
```

```
binom_density %>%
  ggplot() +
  geom_col(aes(x = k, y = p)) +  # In the book, they use `geom_bar`
                                 # with `stat = "identity". This
                                 # gives the same result, but without
                                 # having to change the stat from the
                                 # default
  geom_vline(xintercept = n_heads_observed, col = "blue")
```



I think it's kind of interesting to show, instead, shading of the observed value *or higher*. You can do that like this:

```
# The "Inf" and "-Inf" stand for "infinity". Use these for any of the
# rectangle boundaries where you don't care where they end (they cover
# the full plot area in those directions). "geom_rect" is for
# rectangle.
binom_density %>%
  ggplot() +
  geom_rect(xmin = n_heads_observed, xmax = Inf,
            ymin = -Inf, ymax = Inf,
            fill = "lightblue") +
  geom_col(aes(x = k, y = p))
```



Calculate the same thing using Monte Carlo simulation:

```
# They use `replicate`. That's fine, but you could also get this
# done with a `map` from `purrr` approach.

sim_outcomes <- tibble(sim_number = 1:10000) %>%
  # Note that the formula I'm putting in for the second argument
  # of the map is exactly the same as the one we used to
  # do a single experiment (although in that case our probabilities
  # were different---now we want to experiment when the coin is
  # fair).
  mutate(simulation = map(sim_number, ~ sample(c("H", "T"),
                                               size = num_flips,
                                               replace = TRUE,
                                               prob = c(0.5, 0.5)))) %>%
  # That `map` created a list column with the 100 outcomes of
  # one experiment (i.e., an experiment of 100 coin flips, and
  # the vector gives the list of heads and tails). Now we
  # want to determine from that list the number of heads. We
  # can map again to do that. Since this is a single number,
  # I'm using the `_dbl` version of map to output as a regular
  # column of numbers
  mutate(n_heads = map_dbl(simulation, ~ sum(.x == "H")))
```

```
# Here's what the dataframe looks like now
sim_outcomes %>%
  slice(1:3)
```

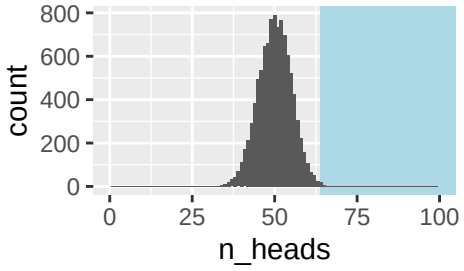
```
## # A tibble: 3 x 3
##   sim_number simulation  n_heads
##        <int> <list>        <dbl>
## 1          1 <chr [100]>      46
## 2          2 <chr [100]>      49
## 3          3 <chr [100]>      53
```

```
# And here's what's inside one of the `simulation` column values:
sim_outcomes %>%
  pull(simulation) %>% # `pull` pulls out a column of a dataframe
  `[[`(1) # Extract the first element of that column
```

```
##   [1] "T" "H" "T" "T" "T" "T" "H" "T" "H" "T" "H" "H" "H" "H" "T" "H" "H" "H"
##  [19] "T" "H" "T" "T" "H" "H" "T" "T" "T" "T" "H" "H" "T" "H" "H" "T" "H" "T"
##  [37] "H" "H" "T" "H" "T" "T" "T" "T" "T" "H" "H" "T" "T" "T" "T" "H" "T" "T"
##  [55] "H" "H" "T" "T" "H" "T" "H" "T" "T" "H" "T" "T" "T" "H" "H" "T" "T" "T"
##  [73] "H" "T" "H" "H" "T" "T" "H" "H" "T" "T" "T" "H" "T" "H" "T" "H" "H" "H"
##  [91] "H" "T" "H" "H" "T" "H" "T" "T" "T" "H"
```

Now you can plot the results of this simulation (which should look almost exactly the same as what we got from using the theoretical equation with `dbinom` earlier):

```
sim_outcomes %>%
  ggplot() +
  geom_rect(xmin = n_heads_observed, xmax = Inf,
            ymin = -Inf, ymax = Inf,
            fill = "lightblue") +
  # Since you need to count up the number of "experiments" (i.e.,
  # simulations) with each outcome, you need to use a histogram
  # geom instead of a column histogram. Note that the shape of
  # this plot will look the same as before, but the y-axis scale
  # will be different. With a histogram, we've got the number of
  # cases (out of 10000 simulations) within each bin of number of
  # heads, while when we used `dbinom`, we got the probability. If
  # you divide all the numbers on the y-axis by 10,000, you should
  # get things to the same scale
  geom_histogram(aes(x = n_heads), binwidth = 1) +
  xlim(c(0, num_flips)) # Make sure the x-axis covers all possible values
```



They show how you can use the `binom.test` function for this single test (from a single experiment). In this test, you're taking the outcome from your experiment (64, which was from a biased coin, but we're assuming now that we don't know that and are trying to determine if there's evidence that it's not unbiased).

```
binom_test <- binom.test(x = n_heads_observed,
                         n = num_flips,
                         p = 0.05)  # This is the probability of
                                    # heads under our null hypothesis
                                    # of an unbiased coin

# You can print the output directly to read the results
binom_test
```

```
##
##  Exact binomial test
##
## data:  n_heads_observed and num_flips
## number of successes = 64, number of trials = 100, p-value < 2.2e-16
## alternative hypothesis: true probability of success is not equal to 0.05
## 95 percent confidence interval:
##  0.5378781 0.7335916
## sample estimates:
## probability of success
##                   0.64
```

```
# However---and this will be *super* helpful as we move to
# making lots of test---you can use the `broom` package's functions
# to produce tidy output from the test result object
library(broom)
binom_test %>%
        # So, this is kind of cool... If you're piping into a
        # function and you don't need to specify any arguments,
        # you can leave off the parentheses. This would have
  tidy  # worked equally well with the more traditional `tidy()`.
```

```
## # A tibble: 1 x 8
##   estimate statistic  p.value parameter conf.low conf.high method    alternative
##      <dbl>     <dbl>    <dbl>     <dbl>    <dbl>     <dbl> <chr>     <chr>
## 1     0.64        64 1.74e-57       100    0.538     0.734 Exact bi~ two.sided
```

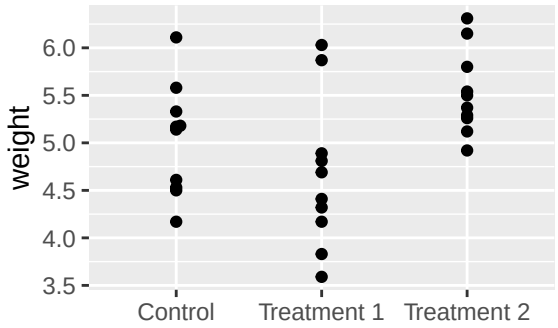
t-test

They're using an example dataset from the `datasets` package (I think that comes default with your base R installation) and the `ggbeeswarm` ggplot extension.

```
library("ggbeeswarm") # This is an extension of ggplot. There are
                      # loads of these now. Google "ggplot extensions"
                      # and look for the gallery RStudio has collected
                      # of some of them to check out more.
data("PlantGrowth")  # This data has a helpfile at `?PlantGrowth`
```

Plot the outcomes for the three groups (control, treatment 1, and treatment 2). They're using a beeswarm geom. In this case, there's not loads of data, so you don't really see the "swarm"—you probably would have been fine with `geom_point`. However, as you move to more data, `geom_beeswarm` is really nice in showing points that otherwise would overlap and hide each other. I don't like that they used both color and position to show the different groups. It's generally better to use just one aesthetic per thing, so that's what I'm doing here. Also, it's much better to use clearer labels than "ctrl", "trt1", and "trt2", so I've used a function from the `forcats` package (loads as part of `tidyverse`) to change those labels.

```
PlantGrowth %>%
  # You need the backticks for some of these to "protect" the space in
  # them.
  mutate(group = fct_recode(group,
                            Control = "ctrl",
                            `Treatment 1` = "trt1",
                            `Treatment 2` = "trt2")) %>%
  ggplot(aes(x = group, y = weight)) +
  geom_beeswarm() +
  labs(x = "") # Really don't need the x-axis label since each group
```




```
               # is now well-labeled
```

Next, they want to apply a t-test comparing the weights in the control group and the treatment 1 group. They use `with`, which (more or less) attaches the dataframe for a minute (within the `with` call) so you can just refer to column names unstead of having to start everything with `PlantGrowth$`. This used to be pretty popular before the tidyverse got big, but now most of the tidyverse functions automatically allow this kind of non-standard evaluation, so you really don't see `with` all that much anymore.

So, what we want to do here essentially is to input a vector with just the weights from the control group as the first argument of `t.test` and just the second treatment group as the second argument. Without `with`, the classic way to do this would be (notice all the `PlantGrowth$`'s!):

```
t.test(PlantGrowth$weight[PlantGrowth$group == "ctrl"],
       PlantGrowth$weight[PlantGrowth$group == "trt2"],
       var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  PlantGrowth$weight[PlantGrowth$group == "ctrl"] and PlantGrowth$weight[PlantGrowth$group == "
## t = -2.134, df = 18, p-value = 0.04685
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.980338117 -0.007661883
## sample estimates:
## mean of x mean of y
##     5.032     5.526
```

Here's a tidyverse alternative:

```
PlantGrowth %>%
  # Restrict to only the two groups you want to test
  filter(group %in% c("ctrl", "trt2")) %>%
  # Pipe straight into `t.test`. Why not? To do this, you need to use
```

```
  # the "pronoun" `.`. This refers to the dataframe you're currently
  # piping
  t.test(weight ~ group, data  = .)
```

```
##
##  Welch Two Sample t-test
##
## data:  weight by group
## t = -2.134, df = 16.786, p-value = 0.0479
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.98287213 -0.00512787
## sample estimates:
## mean in group ctrl mean in group trt2
##              5.032              5.526
```

To get all the way to a tidy output, add a line using `tidy` from the `broom` package:

```
PlantGrowth %>%
  filter(group %in% c("ctrl", "trt2")) %>%
  t.test(weight ~ group, data  = .) %>%
  # Move back to a tidy dataframe
  tidy()
```

```
## # A tibble: 1 x 10
##   estimate estimate1 estimate2 statistic p.value parameter conf.low conf.high
##      <dbl>     <dbl>     <dbl>     <dbl>   <dbl>     <dbl>    <dbl>     <dbl>
## 1   -0.494      5.03      5.53     -2.13  0.0479      16.8   -0.983  -0.00513
## # ... with 2 more variables: method <chr>, alternative <chr>
```

Not as pretty to read, but *much* easier to work with to do more stuff (plotting, etc.) and for when you're doing multiple tests.

Here's the tidyverse version of duplicating the data by adding a second copy of the dataframe before running the t-test:

```
PlantGrowth %>%
  bind_rows(PlantGrowth) %>% # Add the duplicate of the dataset
  filter(group %in% c("ctrl", "trt2")) %>%
  t.test(weight ~ group, data  = .) %>%
  tidy()
```

```
## # A tibble: 1 x 10
##   estimate estimate1 estimate2 statistic p.value parameter conf.low conf.high
##      <dbl>     <dbl>     <dbl>     <dbl>   <dbl>     <dbl>    <dbl>     <dbl>
## 1   -0.494      5.03      5.53     -3.10 0.00377      35.4   -0.817    -0.171
## # ... with 2 more variables: method <chr>, alternative <chr>
```

Note how the p-value is much lower in this case.

```
PlantGrowth %>%
  sample_frac(size = 0.5) %>%
  bind_rows(., .) %>%
  filter(group %in% c("ctrl", "trt2")) %>%
  t.test(weight ~ group, data  = .) %>%
  tidy()
```

```
## # A tibble: 1 x 10
```

```
##   estimate estimate1 estimate2 statistic p.value parameter conf.low conf.high
##      <dbl>     <dbl>     <dbl>     <dbl>   <dbl>     <dbl>    <dbl>     <dbl>
## 1   -0.397      5.26      5.66     -1.30   0.221      11.2    -1.07     0.275
## # ... with 2 more variables: method <chr>, alternative <chr>
```

```
pg1 <- PlantGrowth %>%
  sample_frac(size = 0.5)
pg2 <- pg1 %>%
  mutate(noise = rnorm(15, mean = 0, sd = 0.2),
         weight = weight + noise) %>%
  select(-noise)

pg1 %>%
  bind_rows(pg2) %>%
  filter(group %in% c("ctrl", "trt2")) %>%
  t.test(weight ~ group, data  = .) %>%
  tidy()
```

```
## # A tibble: 1 x 10
##   estimate estimate1 estimate2 statistic p.value parameter conf.low conf.high
##      <dbl>     <dbl>     <dbl>     <dbl>   <dbl>     <dbl>    <dbl>     <dbl>
## 1   -0.572      4.88      5.45     -3.97 0.00184      12.1   -0.886    -0.258
## # ... with 2 more variables: method <chr>, alternative <chr>
```

False discovery rate

```
# Uncomment and run the following line if you need to install the package
# BiocManager::install("DESeq2")
library("DESeq2")

# Uncomment and run the following line if you need to install the package
# BiocManager::install("airway")
library("airway") # For more, run `vignette("airway")`
data("airway") # For more, run `?airway`

summary(airway)
```

```
##                    Length                      Class
##                     64102 RangedSummarizedExperiment
##                      Mode
##                        S4
```

```
class(airway)
```

```
## [1] "RangedSummarizedExperiment"
## attr(,"package")
## [1] "SummarizedExperiment"
```

```
str(airway, max.level = 2) # Check the top levels of the object
```

```
## Formal class 'RangedSummarizedExperiment' [package "SummarizedExperiment"] with 6 slots
##   ..@ rowRanges      :Formal class 'GRangesList' [package "GenomicRanges"] with 5 slots
##   ..@ colData        :Formal class 'DataFrame' [package "IRanges"] with 6 slots
##   ..@ assays         :Reference class 'ShallowSimpleListAssays' [package "GenomicRanges"] with 1 fie
##   .. ..and 12 methods.
##   ..@ NAMES          : NULL
```

```
##   ..@ elementMetadata:Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
##   ..@ metadata       :List of 1
```

It looks like there's some data (maybe meta-data?) in the `colData` slot:

```
airway@colData
```

```
## DataFrame with 8 rows and 9 columns
##             SampleName     cell      dex    albut        Run avgLength
##               <factor> <factor> <factor> <factor>   <factor> <integer>
## SRR1039508 GSM1275862   N61311    untrt    untrt SRR1039508       126
## SRR1039509 GSM1275863   N61311      trt    untrt SRR1039509       126
## SRR1039512 GSM1275866  N052611    untrt    untrt SRR1039512       126
## SRR1039513 GSM1275867  N052611      trt    untrt SRR1039513        87
## SRR1039516 GSM1275870  N080611    untrt    untrt SRR1039516       120
## SRR1039517 GSM1275871  N080611      trt    untrt SRR1039517       126
## SRR1039520 GSM1275874  N061011    untrt    untrt SRR1039520       101
## SRR1039521 GSM1275875  N061011      trt    untrt SRR1039521        98
##             Experiment    Sample    BioSample
##               <factor>  <factor>     <factor>
## SRR1039508  SRX384345 SRS508568 SAMN02422669
## SRR1039509  SRX384346 SRS508567 SAMN02422675
## SRR1039512  SRX384349 SRS508571 SAMN02422678
## SRR1039513  SRX384350 SRS508572 SAMN02422670
## SRR1039516  SRX384353 SRS508575 SAMN02422682
## SRR1039517  SRX384354 SRS508576 SAMN02422673
## SRR1039520  SRX384357 SRS508579 SAMN02422683
## SRR1039521  SRX384358 SRS508580 SAMN02422677
```

The `rowRanges` slot seems to have some data in it...

```
airway@rowRanges
```

```
## Warning in showList(object, showFunction, print.classinfo = TRUE): Note that starting with BioC 3.7,
##   GRangesList **instances** needs to be set to "CompressedGRangesList".
##   Please update this object with 'updateObject(object, verbose=TRUE)' and
##   re-serialize it.
```

```
## GRangesList object of length 64102:
## $ENSG00000000003
## GRanges object with 17 ranges and 2 metadata columns:
##        seqnames            ranges strand |   exon_id       exon_name
##           <Rle>         <IRanges>  <Rle> | <integer>     <character>
##    [1]        X 99883667-99884983      - |    667145 ENSE00001459322
##    [2]        X 99885756-99885863      - |    667146 ENSE00000868868
##    [3]        X 99887482-99887565      - |    667147 ENSE00000401072
##    [4]        X 99887538-99887565      - |    667148 ENSE00001849132
##    [5]        X 99888402-99888536      - |    667149 ENSE00003554016
##    ...      ...               ...    ... .       ...             ...
##   [13]        X 99890555-99890743      - |    667156 ENSE00003512331
##   [14]        X 99891188-99891686      - |    667158 ENSE00001886883
##   [15]        X 99891605-99891803      - |    667159 ENSE00001855382
##   [16]        X 99891790-99892101      - |    667160 ENSE00001863395
##   [17]        X 99894942-99894988      - |    667161 ENSE00001828996
##
## ...
## <64101 more elements>
```

```
## -------
## seqinfo: 722 sequences (1 circular) from an unspecified genome
```

The `metadata` slot has some information about the experiment:

```
airway@metadata
```

```
## [[1]]
## Experiment data
##   Experimenter name: Himes BE
##   Laboratory: NA
##   Contact information:
##   Title: RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid responsive gene tha
##   URL: http://www.ncbi.nlm.nih.gov/pubmed/24926665
##   PMIDs: 24926665
##
##   Abstract: A 226 word abstract is available. Use 'abstract' method.
```

To figure out more about this data, I think we'd need to look into the help documentation for the class that the data's in (`RangedSummarizedExperiment`).

Run the code from the book to test for differential expression for each gene. It looks like this is testing by cell line (the helpfile says there are four cell lines; `cell`) and by treatment (they are either control or treated with dexamethasone; `dex`).

```
aw <- DESeqDataSet(se = airway, design = ~ cell + dex)
aw
```

```
## class: DESeqDataSet
## dim: 64102 8
## metadata(2): '' version
## assays(1): counts
## rownames(64102): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
## rowData names(0):
## colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
## colData names(9): SampleName cell ... Sample BioSample
```

Here's more of a summary of the output we get:

```
str(aw, max.level = 2)
```

```
## Formal class 'DESeqDataSet' [package "DESeq2"] with 8 slots
##   ..@ design            :Class 'formula'  language ~cell + dex
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   ..@ dispersionFunction:function ()
##   ..@ rowRanges         :Formal class 'CompressedGRangesList' [package "GenomicRanges"] with 5 slots
##   ..@ colData           :Formal class 'DataFrame' [package "IRanges"] with 6 slots
##   ..@ assays            :Reference class 'ShallowSimpleListAssays' [package "GenomicRanges"] with 1 f
##   .. ..and 12 methods.
##   ..@ NAMES             : NULL
##   ..@ elementMetadata   :Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
##   ..@ metadata          :List of 2
```

I think that, so far, we've just set up the data in the format we need to run the differential expression analysis. I think the next step will run the actual analysis:

```
aw <- DESeq(aw)
```

```
## estimating size factors
```

```
## estimating dispersions
## gene-wise dispersion estimates
## mean-dispersion relationship
## final dispersion estimates
## fitting model and testing
```

```
aw
```

```
## class: DESeqDataSet
## dim: 64102 8
## metadata(2): '' version
## assays(4): counts mu H cooks
## rownames(64102): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
## rowData names(34): baseMean baseVar ... deviance maxCooks
## colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
## colData names(10): SampleName cell ... BioSample sizeFactor
```

Looking more at this object:

```
str(aw, max.level = 2)
```

```
## Formal class 'DESeqDataSet' [package "DESeq2"] with 8 slots
##   ..@ design            :Class 'formula'  language ~cell + dex
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   ..@ dispersionFunction:function (q)
##   .. ..- attr(*, "coefficients")= Named num [1:2] 0.00952 3.60009
##   .. .. ..- attr(*, "names")= chr [1:2] "asymptDisp" "extraPois"
##   .. ..- attr(*, "fitType")= chr "parametric"
##   .. ..- attr(*, "varLogDispEsts")= num 0.953
##   .. ..- attr(*, "dispPriorVar")= num 0.529
##   ..@ rowRanges         :Formal class 'CompressedGRangesList' [package "GenomicRanges"] with 5 slots
##   ..@ colData           :Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
##   ..@ assays            :Reference class 'ShallowSimpleListAssays' [package "SummarizedExperiment"] w
##   .. ..and 14 methods.
##   ..@ NAMES             : NULL
##   ..@ elementMetadata   :Formal class 'DataFrame' [package "S4Vectors"] with 6 slots
##   ..@ metadata          :List of 2
```

It looks like there's a `results` method for this object class that will pull out the results we want:

```
results(aw)
```

```
## log2 fold change (MLE): dex untrt vs trt
## Wald test p-value: dex untrt vs trt
## DataFrame with 64102 rows and 6 columns
##                        baseMean      log2FoldChange             lfcSE
##                       <numeric>           <numeric>         <numeric>
## ENSG00000000003 708.602169691234   0.381253982523043 0.100654411867396
## ENSG00000000005               0                  NA                NA
## ENSG00000000419 520.297900552084  -0.206812601085043 0.112218646508368
## ENSG00000000457 237.163036796015 -0.0379204312050886 0.143444654933749
## ENSG00000000460 57.9326331250967  0.0881681758708941 0.287141822933388
## ...                           ...                 ...               ...
## LRG_94                        0                  NA                NA
## LRG_96                        0                  NA                NA
## LRG_97                        0                  NA                NA
```

```
## LRG_98                          0                 NA                NA
## LRG_99                          0                 NA                NA
##                              stat             pvalue              padj
##                         <numeric>          <numeric>         <numeric>
## ENSG00000000003   3.78775232451125 0.000152016273081244 0.00128362968201811
## ENSG00000000005                 NA                 NA                NA
## ENSG00000000419  -1.84294328545142  0.0653372915124384  0.196546085664315
## ENSG00000000457 -0.264355832725887   0.791505741607837  0.911459479590443
## ENSG00000000460  0.307054454729667   0.758801923977348  0.895032784968832
## ...                            ...                ...               ...
## LRG_94                          NA                 NA                NA
## LRG_96                          NA                 NA                NA
## LRG_97                          NA                 NA                NA
## LRG_98                          NA                 NA                NA
## LRG_99                          NA                 NA                NA
```

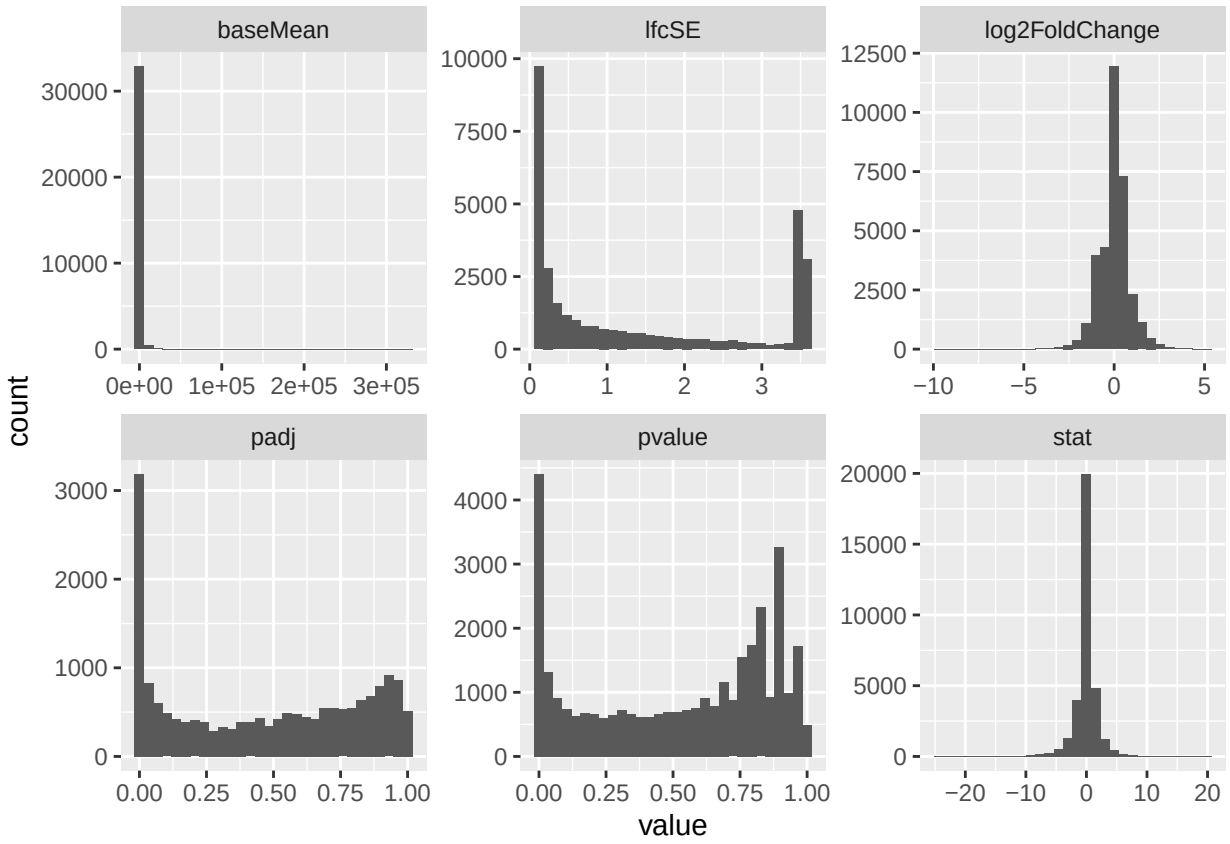
In the text, they pull this out and convert it to a dataframe, while also filtering out results where the test result (p-value) is missing. We could do the *whole* process in a pipeline to avoid all this making new objects deal like they do in the book code. I think we could do the whole process like this:

```
airway %>%
  DESeqDataSet(design = ~ cell + dex) %>% # Set up the data
  DESeq() %>% # Perform the differential expression analysis
  results() %>% # Extract the results
  as_tibble() %>%  # Convert to a regular dataframe
  filter(!is.na(pvalue)) -> airway_diff_expr # You can assign at the end!

airway_diff_expr %>%
  # There seems to be `slice` in another loaded package, too, so
  # we need to use the `dplyr::` notation to say we want the one from
  # dplyr
  dplyr::slice(1:3)
```

```
## # A tibble: 3 x 6
##   baseMean log2FoldChange lfcSE   stat   pvalue    padj
##      <dbl>          <dbl> <dbl>  <dbl>    <dbl>   <dbl>
## 1     709.          0.381  0.101  3.79  0.000152 0.00128
## 2     520.         -0.207  0.112 -1.84  0.0653   0.197
## 3     237.         -0.0379 0.143 -0.264 0.792    0.911
```

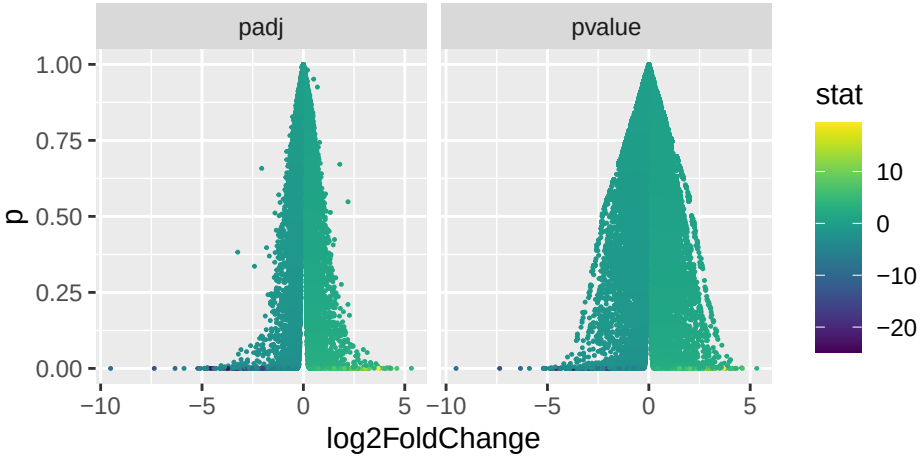
One way to explore the data is to plot histograms of all the columns to see how the values are distributed. You can do this pretty easily by pivoting all the data to be longer (previously, the function for this was `gather`, but the `tidyr` package has been updated to use `pivot_longer`). Then, you can make histograms and facet to separate the columns. You should set the scales to "free", since they'll likely all be on different scales.

```
airway_diff_expr %>%
  pivot_longer(cols = baseMean:padj,
               names_to = "column",
               values_to = "value") %>%
  ggplot(aes(x = value)) +
  geom_histogram() +
  facet_wrap(~ column, scales = "free")
```

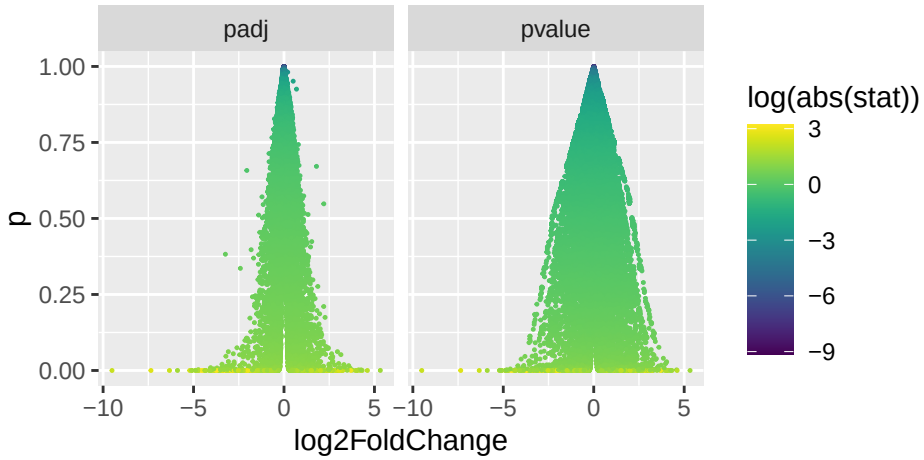


Let's see how `pvalue` and `padj` compare with the log2FoldChange and the test statistic columns:

```
library(viridis)
airway_diff_expr %>%
  select(pvalue, padj, stat, log2FoldChange) %>%
  pivot_longer(cols = pvalue:padj, names_to = "type",
               values_to = "p") %>%
  ggplot(aes(x = log2FoldChange, y = p, color = stat)) +
  geom_point(size = 0.2) +
  facet_wrap(~ type) +
  scale_color_viridis()
```

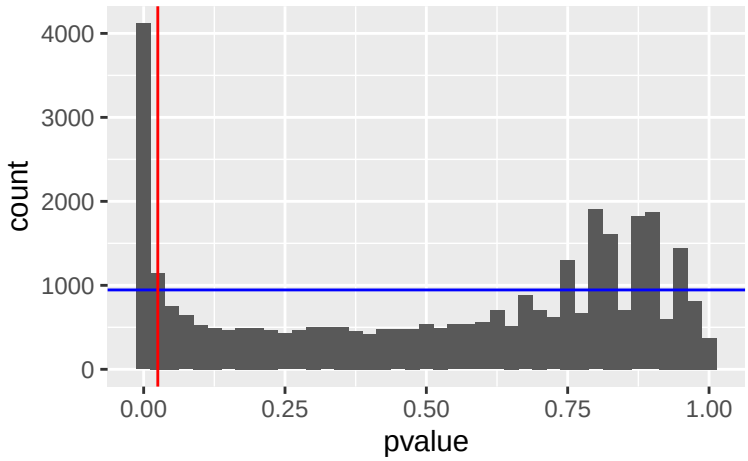


```
# Try with taking the log of the absolute value of the test
# statistic, so everything isn't so crowded around the middle
# in terms of the color
airway_diff_expr %>%
  select(pvalue, padj, stat, log2FoldChange) %>%
  pivot_longer(cols = pvalue:padj, names_to = "type",
               values_to = "p") %>%
  ggplot(aes(x = log2FoldChange, y = p, color = log(abs(stat)))) +
  geom_point(size = 0.2) +
  facet_wrap(~ type) +
  scale_color_viridis()
```



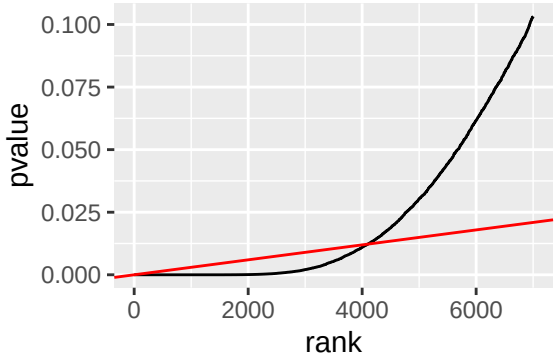
Make a histogram of the p-values:

```
# Binwidth we're using in the histogram
# (also the alpha value we're using, so all counts in the first
# bin of the histogram are below this value for their p-value)
alpha <- 0.025
# Two times the proportion of time that the p-value is over 0.5
pi0 <- 2 * mean(airway_diff_expr$pvalue > 0.5)

airway_diff_expr %>%
  ggplot(aes(x = pvalue)) +
  geom_histogram(binwidth = alpha) +
  geom_hline(yintercept = alpha * pi0 * nrow(airway_diff_expr),
             color = "blue") +
  geom_vline(xintercept = alpha, color = "red")
```



Plot of Benjamini-Hochberg procedure:

```
airway_diff_expr %>%
  mutate(rank = rank(pvalue))  %>% # Returns the ranks of the pvalues
  filter(rank <= 7000) %>% # Limit to lowest 7000 p-values
  ggplot(aes(x = rank, y = pvalue)) +
  geom_line() +
  geom_abline(slope = 0.10 / nrow(airway_diff_expr), color = "red")
```



A "tidier" way to get the rank of the last (i.e., highest p-value) test to reject under the Benjamini-Hochberg procedure (k_{max}):

```
airway_diff_expr %>%
  mutate(rank = rank(pvalue))  %>%
  arrange(rank) %>%
  filter(pvalue <= 0.10 * rank / n()) %>%
  dplyr::count()
```

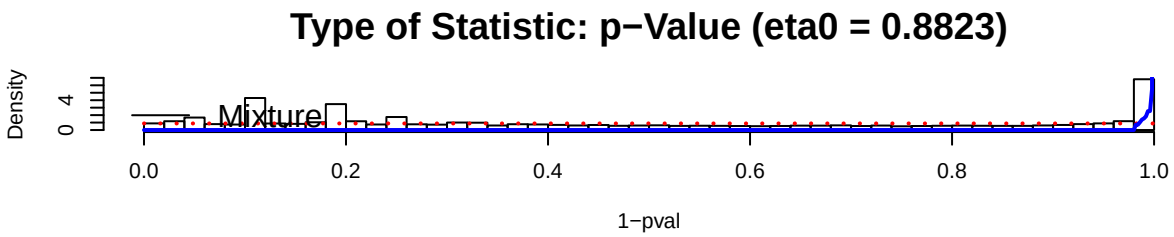
```
## # A tibble: 1 x 1
##       n
##   <int>
## 1  4099
```

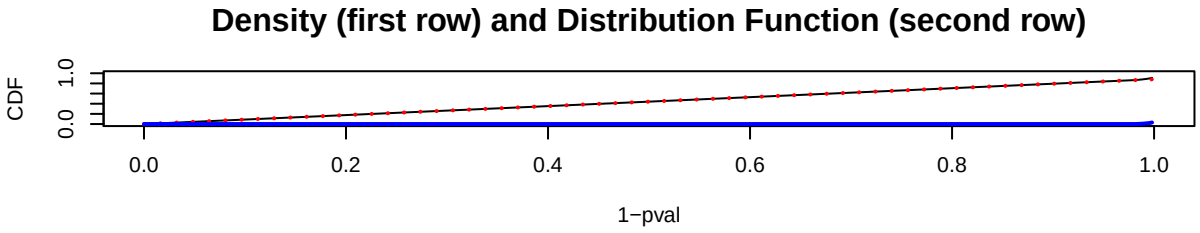
Local FDR

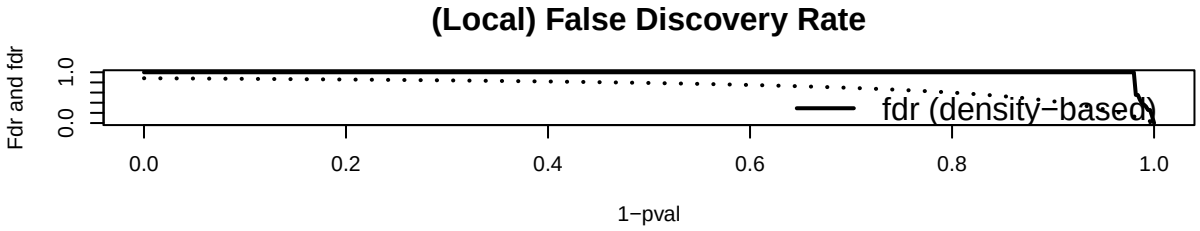
```
library("fdrtool")

ft <- airway_diff_expr %>%
```

```
  pull(pvalue) %>%  # The next function needs a vector, so pull `pvalue`
  fdrtool(statistic = "pvalue")
```

```
## Step 1... determine cutoff point
## Step 2... estimate parameters of null distribution and eta0
## Step 3... compute p-values and estimate empirical PDF/CDF
## Step 4... compute q-values and local fdr
## Step 5... prepare for plotting
```








```
# Check out the resulting object
str(ft)
```

```
## List of 5
##  $ pval     : num [1:33469] 0.000152 0.065337 0.791506 0.758802 0.693734 ...
##  $ qval     : num [1:33469] 0.0021 0.3287 0.8558 0.8505 0.8387 ...
##  $ lfdr     : num [1:33469] 0.0193 1 1 1 1 ...
##  $ statistic: chr "pvalue"
##  $ param    : num [1, 1:4] 1.05e-01 2.64e+04 8.82e-01 2.49e-03
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:4] "cutoff" "N.cens" "eta0" "eta0.SE"
```

```
ft %>%
  `[[`("param")
```

```
##        cutoff N.cens      eta0     eta0.SE
## [1,] 0.104961  26430 0.8822922 0.002488846
```