Lab Notes

Chapter 5

OpenIntro Biostatistics

Overview

1. Two-Sample Tests
   - *OI Biostat* Sections 5.2 - 5.3
2. Statistical Power
   - *OI Biostat* Section 5.4
3. Analysis of Variance (ANOVA)
   - *OI Biostat* Section 5.5
4. Multiple Testing
   - Extension to *OI Biostat* Section 5.5
5. Bayesian Hypothesis Testing
   - Extension to *OI Biostat* Section 5.6

Lab 1 introduces hypothesis testing in the two-sample context, discussing the two-sample t-test for paired data and independent group data.

Lab 2 discusses the control of Type I and Type II error and explores the factors influencing the power of a statistical test via simulation.

Lab 3 introduces the analysis of variance procedure for comparing the means of several groups.

Lab 4 examines the multiple testing problem and concept of experiment-wise error in the context of the Golub leukemia data.

Lab 5 integrates the ideas of conditional probability and hypothesis testing to present a broader understanding of p-values, Type I error, and statistical power in a research context.

Lab 1: Two-Sample Tests

Hypothesis Testing with `t.test()`, cont.

The **`t.test()`** function has the following generic structure:

```
t.test(x, y, alternative = "two.sided", mu = 0, conf.level = 0.95, paired = FALSE)
```

where `x` and `y` are numeric vectors of data values, `alternative` specifies the form of the alternative hypothesis, `mu` is $\mu_1 - \mu_2$ (in the paired context, δ_0), and `conf.level` refers to the confidence level. The argument for `alternative` can be either `"two.sided"` ($H_A : \mu_1 \neq \mu_2$), `"less"` ($H_A : \mu_1 < \mu_2$), or `"greater"` ($H_A : \mu_1 > \mu_2$). By default, confidence level is set to 95%, and a two-sided alternative is tested with the independent group test.

To conduct a test on data contained in variable `y` that is grouped by the variable `x`, use the tilde syntax:

```
t.test(y ~ x, ...)
```

The following example shows a hypothesis test for mean standing height in centimeters in the artificial NHANES population, using a random sample of 135 adults. The null hypothesis is that the population mean height for females is equal to the population mean height for males. A one-sided alternative is tested against the null; the output includes the t-statistic, degrees of freedom, p-value, 90% confidence interval, and the sample means of both groups.

```
#load the data
library(oibiostat)
data("nhanes.samp.adult")

#conduct test
t.test(nhanes.samp.adult$Height ~ nhanes.samp.adult$Gender, alternative = "less",
       conf.level = 0.90)
```

```
##
##  Welch Two Sample t-test
##
## data:  nhanes.samp.adult$Height by nhanes.samp.adult$Gender
## t = -10.777, df = 132.95, p-value < 2.2e-16
## alternative hypothesis: true difference in means is less than 0
## 90 percent confidence interval:
##       -Inf -11.91326
## sample estimates:
## mean in group female   mean in group male
##             162.9729             176.5031
```

The following example shows two ways to conduct a hypothesis test for the difference in mean maximal swim velocity between swimmers wearing wetsuits versus swimsuits. The data are paired, since each participant completed two trials: one wearing a wetsuit and one wearing a swimsuit. The null hypothesis of no difference of $H_0 : \delta = 0$ is tested against the two-sided alternative $H_A : \delta \neq 0$.

The first method uses the two-sample test syntax, while the second method uses the one-sample test syntax on the vector of velocity differences.

```
#load the data
library(oibiostat)
data("swim")

#two-sample test syntax
t.test(swim$wet.suit.velocity, swim$swim.suit.velocity, alternative = "two.sided",
       paired = TRUE)
```

```
##
##  Paired t-test
##
## data:  swim$wet.suit.velocity and swim$swim.suit.velocity
## t = 3.7019, df = 11, p-value = 0.00349
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.02534062 0.09965938
## sample estimates:
## mean of the differences
##                  0.0625
```

```
#one-sample test syntax
t.test(swim$velocity.diff, mu = 0, alternative = "two.sided")
```

```
##
##  One Sample t-test
##
## data:  swim$velocity.diff
## t = 3.7019, df = 11, p-value = 0.00349
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.02534062 0.09965938
## sample estimates:
## mean of x
##    0.0625
```

Lab 2: Statistical Power

Simulating Values from a Distribution

R has built-in functions for drawing random values from a distribution. The function `rnorm()` is used in Lab 2 to draw observations from normal distributions with specified parameter values. For reference, details for sampling values from other distributions are also discussed in this section.

The function **`rnorm()`** has the generic structure

```
rnorm(n, mean = 0, sd = 1)
```

where `n` is the number of observations sampled. By default, R assumes that mean and standard deviation are 0 and 1, respectively.

The following code shows how to draw 10 values from a normal distribution with mean 100 and standard deviation 5. As with any random sampling, it is necessary to specify a seed with `set.seed()` for the results to be reproducible.

```
#set seed for pseudorandom sampling
set.seed(2018)

#draw values
rnorm(10, mean = 100, sd = 5)
```

```
##  [1]  97.88508  92.25061  99.67785 101.35441 108.67642  98.67644 110.49735
##  [8] 104.31676  96.94706 103.18528
```

The function **`rbinom()`** has the generic structure

```
rbinom(n, size, prob)
```

where `n` is the number of observations sampled, `size` is the number of trials `n`, and `prob` is the probability of success p.

The following code shows how to draw 10 values from a binomial distribution with 10 trials and success probability 0.35.

```
rbinom(10, 10, 0.35)
```

```
##  [1] 2 4 2 2 5 4 3 6 4 1
```

The function **`rpois()`** has the generic structure

```
rpois(n, lambda)
```

where `n` is the number of observations sampled and `lambda` is the rate parameter λ.

The following code shows how to draw 10 values from a Poisson distribution with rate parameter $\lambda = 3$.

```
rpois(10, 3)
```

```
##  [1] 1 5 0 3 3 4 0 1 3 1
```

The function **rgeom()** has the generic structure

```
rgeom(n, prob)
```

where n is the number of observations sampled and prob is the probability of success p.

The following code shows how to draw 10 values from a geometric distribution with probability of success $p = 0.35$.

```
rgeom(10, 0.35)
```

```
##  [1] 1 0 0 1 4 2 3 1 2 0
```

The function **rnbinom()** has the generic structure

```
rnbinom(n, size, prob)
```

where n is the number of observations sampled, size is the number of successes r, and prob is the probability of success p.

The following code shows how to draw 10 values from a negative binomial distribution with number of successes $r = 4$ and probability of success $p = 0.8$.

```
rnbinom(10, 4, 0.8)
```

```
##  [1] 0 0 0 0 0 2 1 0 1 1
```

The function **rhyper()** has the generic structure

```
rhyper(nn, m, n, k)
```

where nn is the number of observations sampled, m is the total number of successes m, n is the total number of failures $N - m$, and k is the sample size n.

The following code shows how to draw 10 values from a hypergeometric distribution with total number of successes $m = 10$, total number of failures $N - m = 15$, and sample size $n = 8$.

```
rhyper(10, 10, 15, 8)
```

```
##  [1] 3 3 3 4 3 2 2 3 3 4
```

Power and Sample Size Calculations with power.t.test()

The **power.t.test()** function can both compute the power of a one- or two-sample t-test and determine necessary parameters (e.g., sample size) to obtain a target power. The function has the generic structure

```
power.t.test(n = NULL, delta = NULL, sd = 1, sig.level = 0.05,
             power = NULL, type, alternative)
```

where `n` is the sample size (per group), `delta` is the effect size, `sd` is the standard deviation, `sig.level` is the significance level, and `power` is the statistical power. The argument for `type` can be either `"one.sample"`, `"two.sample"`, or `"paired"`, where two-sample implies independent groups. The argument for `alternative` can be either `"two.sided"` or `"one.sided"`.

Exactly one out of `n`, `delta`, `sd`, or `sig.level` must be entered as `NULL`; this is the parameter of interest that will be calculated based on the provided information.

The following code shows how to calculate the power for a one-sample test where $n = 100$, $\Delta = 3$, $\sigma = 12$, $\alpha = 0.05$, with a two-sided alternative.

```
power.t.test(n = 100, delta = 3, sd = 12, sig.level = 0.05,
             power = NULL, type = "one.sample", alternative = "two.sided")
```

```
##
##      One-sample t test power calculation
##
##               n = 100
##           delta = 3
##              sd = 12
##       sig.level = 0.05
##           power = 0.6969757
##     alternative = two.sided
```

The following code shows how to calculate the sample size for a one-sample test where $\Delta = 3$, $\sigma = 12$, $\alpha = 0.05$, and power of 0.70, with a two-sided alternative.

```
power.t.test(n = NULL, delta = 3, sd = 12, sig.level = 0.05,
             power = 0.70, type = "one.sample", alternative = "two.sided")
```

```
##
##      One-sample t test power calculation
##
##               n = 100.6887
##           delta = 3
##              sd = 12
##       sig.level = 0.05
##           power = 0.7
##     alternative = two.sided
```

Lab 3: Analysis of Variance (ANOVA)

The tapply() Function

The **tapply()** function is related to the apply() function introduced in Chapter 1. As with apply(), tapply() allows a specific function to be applied to a matrix; the function can be a pre-defined R function like mean() or a user-defined function. The power of tapply() is that it allows for a vector to be split into groups, with the function applied to each group.

The function has the generic structure

```
tapply(y, x, FUN)
```

where y is the vector of data, x is the grouping variable, and FUN is the function of interest.

The following code shows how to calculate the mean change in non-dominant arm strength for each genotype group in the FAMuSS data.

```
#load the data
library(oibiostat)
data("famuss")

tapply(famuss$ndrm.ch, famuss$actn3.r577x, mean)
```

```
##       CC       CT       TT
## 48.89422 53.24904 58.08385
```

Fitting an ANOVA Model

The **aov()** function fits an ANOVA model to data; wrapping with the summary() function outputs the ANOVA table, which contains the F-statistic and associated p-value. The input to aov() must be in the form of a formula using the tilde syntax:

```
aov(y ~ x)
```

where y is the data vector and y is the grouping variable.

The following code shows the summary of the ANOVA model fit for the association of change in non-dominant arm strength by genotype at the $r577x$ locus on the $ACTN3$ gene.

```
#output summary of anova model
summary(aov(famuss$ndrm.ch ~ famuss$actn3.r577x))
```

```
##                     Df Sum Sq Mean Sq F value Pr(>F)
## famuss$actn3.r577x   2   7043    3522   3.231 0.0402 *
## Residuals          592 645293    1090
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Conducting Pairwise Tests with `pairwise.t.test`

The **`pairwise.t.test`** function is used to conduct pairwise comparisons with corrections for multiple testing. Note that the input to this function uses different syntax from `aov()`: instead of the tilde, the comma is used to separate the data variable and grouping variable. The generic structure of the function is

```
pairwise.t.test(y, x, p.adj)
```

where `y` is the data vector, `x` is the grouping vector, and `p.adj` can be one of several adjustment choices, such as `"none"` for no correction and `"bonf"` for Bonferroni.

The following code shows how to conduct pairwise two-sample t-tests between mean change in non-dominant arm strength for each of the genotype groups in the FAMuSS data.

```
#no correction
pairwise.t.test(famuss$ndrm.ch, famuss$actn3.r577x, p.adj = "none")
```

```
##
##  Pairwise comparisons using t tests with pooled SD
##
## data:  famuss$ndrm.ch and famuss$actn3.r577x
##
##    CC    CT
## CT 0.179 -
## TT 0.011 0.144
##
## P value adjustment method: none
```

```
#Bonferroni correction
pairwise.t.test(famuss$ndrm.ch, famuss$actn3.r577x, p.adj = "bonf")
```

```
##
##  Pairwise comparisons using t tests with pooled SD
##
## data:  famuss$ndrm.ch and famuss$actn3.r577x
##
##    CC    CT
## CT 0.537 -
## TT 0.034 0.433
##
## P value adjustment method: bonferroni
```

Note that when the Bonferroni correction is applied, `R` multiples the p-value by K, the number of comparisons; thus, the values output from `pairwise.t.test` when `p.adj = "bonf"` should be compared to α, not $\alpha^\star$. Comparing an unadjusted p-value to α/K is equivalent to comparing the quantity ($K \times p$-value) to α.

Lab 4: Multiple Testing

The `for` loop was introduced in Chapter 2; nested loops were introduced in Chapter 3 in the context of simulating geometric, negative binomial, and hypergeometric random variables. This section specifically discusses nested `for` loops and the logic behind the simulation code for estimating experiment-wise error in the Golub leukemia dataset.

Nested `for` Loops

Understanding a nested `for` loop requires keeping track of more than two counters (i.e., index variables). In the following basic example, there are two counters: the outer counter, k, runs from 1 through 4, while the inner counter, j, runs from 1 through 2.

- For the first iteration, $k = 1$. Upon encountering the second loop, R cycles through $j = 1$ and $j = 2$. Thus, there are two values of the product $k \times j$ for this first iteration: $1 \times 1 = 1$ and $1 \times 2 = 2$.
- For the fourth iteration, $k = 4$. The two values of the product $k \times j$ are then $4 \times 1 = 4$ and $4 \times 2 = 8$.

```
for(k in 1:4){

  for(j in 1:2){
    print(k*j)
  }

}
```

```
## [1] 1
## [1] 2
## [1] 2
## [1] 4
## [1] 3
## [1] 6
## [1] 4
## [1] 8
```

Question 2 of the lab refers to a simulation for estimating experiment-wise error rate when two independent one-sample hypothesis tests are conducted. The approach shown in the lab is to create two separate vectors of observations. While this approach is straightforward, it is impractical for a large number of tests.

The following code demonstrates a more flexible approach that hinges on using nested `for` loops. When the number of tests is specified as a parameter, the simulation can simply be re-run to model experiment-wise error for any number of tests.

- The outer loop, with index variable k, runs from 1 to the specified number of iterations. The inner loop, with index variable j, runs from 1 to the specified number of tests.
- Each time the outer loop runs, a set of data (i.e., observations in samples to be tested) is

generated. The matrix `obs.matrix` has number of columns equivalent to `num.tests` and number of rows equivalent to `num.obs`. It is populated with $num.tests \times num.obs = 100 \times 100$ draws from a standard normal distribution. This is a more efficient way to generate the simulated data than running `rnorm()` 100 times and creating 100 vectors.

- The inner loop proceeds through each column of `obs.matrix`, conducting a t-test on the values in column j and storing the p-value as the j^{th} entry of the vector `p.vals`.
- The last instruction in the outer loop is to record the minimum value in `p.vals` as the k^{th} entry in the vector `min.p.vals`. Note how the values in `obs.matrix` and `p.vals` are rewritten with each iteration of the outer loop, but not the values in `min.p.vals`.
- The `reject` vector is also defined more efficiently than in the version of the simulation shown in the lab. If the k^{th} entry in `min.p.vals` is larger than α, then the k^{th} iteration represents one instance of experiment-wise error occurring. This logic was discussed in Question 7.
- The result of the simulation agrees closely with the algebraic solution from Question 3 of the lab. From simulation, the estimated experiment-wise error is 0.995; the probability of at least one incorrect rejection in 100 independent tests conducted at $\alpha = 0.05$ is 0.994.

```
#set parameters
num.tests = 100
num.obs = 100
num.iterations = 1000

alpha = 0.05

#set seed
set.seed(2018)

#create empty lists
p.vals = vector("numeric", num.tests)
min.p.vals = vector("numeric", num.iterations)

#run simulation
for(k in 1:num.iterations){

  obs.matrix = matrix(rnorm(num.tests*num.obs),
                      nrow = num.obs, ncol = num.tests)

  for(j in 1:num.tests){

    p.vals[j] = t.test(obs.matrix[, j], mu = 0)$p.val

  }

  min.p.vals[k] = min(p.vals)

}
```

```r
#view results
reject = (min.p.vals <= alpha)
table(reject)
```

```
## reject
## FALSE  TRUE
##     5   995
```

Lab 5: Bayesian Hypothesis Testing