OEB 153 – Lab 1

Scripts

In Lab 0, you entered commands directly into the R terminal, exploring R. In this Lab it will be helpful to store your commands in an organized script. This is how one typically interacts with R when working on a larger project. Working interactively from a script, you write bits of code and then send them to the terminal, either line by line or chunk by chunk. It is crucial to include comment lines, which start with the `#` character. Below is an example script that could be used to answer some of the practice questions from Lab 0.

```
#########################
# Lab 0 practice problems
#########################

# set working directory
setwd("/home/peter/courses/oeb153/labs/lab0/")

# import the data
insects = read.csv("insect_stats.csv",header=TRUE)

# greatest mass
max(insects$mass)
# [1] 1662.94

# sample mean and variance of egg volume
mean(insects$eggvolume)
# [1] 9.272374
var(insects$eggvolume)
# [1] 619.3042

# number of insects with fecundity > 40
sum(insects$fecundity > 40)
# [1] 47
```

This is a simple script; usually there would be many variable assignments, custom function definitions, or maybe library importations. Notice the comments explaining each of the commands, and notice how the output is pasted into the script, commented out. This can be helpful for when you look at an old analysis file and want to make sure that you still get the same answer.

You can copy and paste commands into the terminal and then results back into your script, but this can get tedious. If you are using RStudio, you can use the keyboard shortcut `Ctrl+Enter` (`Command+Enter` in Mac OSX) to send the current line (or current selection) to the terminal. Other shortcuts can be found in the Code menu.

An important point is that if you want to go back and change some command that was given earlier, it is imperative that you change the command in your script and then *send it to the terminal again*. The script is only a place to store and organize commands; R only ever executes code that is sent to the R terminal.

Strings and Factors

So far we have worked only with numerical data. Categorical variables are a big part of statistics, and in R they are represented by **factors**. Factors are usually created from **strings** (a.k.a. **characters** in R), which are alphanumeric words or phrases surrounded by double quotations `"` in R. For example, one could enter the following:

```
x = "hello"
x
```

```
[1] "hello"
```

```
is.character(x)
```

```
[1] TRUE
```

Here `x` is a character object, or string. A **character vector** is a vector of multiple strings:

```
# hypothetical treatment
treatment = c("low", "medium", "medium", "high")
treatment
```

```
[1] "low"    "medium" "medium" "high"
```

To make use of the many functionalities R has for dealing with categorical variables, it is necessary to convert character vectors into factors. This is done using the function `as.factor()`:

```
treatment = as.factor(treatment)
treatment
```

```
[1] low    medium medium high
Levels: high low medium
```

Factors have **levels**, which are the possible values that a factor can take on. You can see the levels of a factor `fac` by calling `levels(fac)`. To extract members of a factor that match certain levels, we can use some of the same logical operators we used in Lab 0:

```
# simulated response data
response = rnorm(4,mean=10,sd=2)
# create a simple data frame
simdata = data.frame(treatment, response)
simdata
```

```
  treatment response
1       low   11.196
2    medium   13.261
3    medium    8.375
4      high    9.915
```

```
simdata$response[simdata$treatment == "high"]
```

```
[1] 9.915
```

```
simdata$response[simdata$treatment %in% c("medium", "high")]
```

```
[1] 13.261  8.375  9.915
```

Here, `simdata$treatment == "high"` produces a logical vector, checking whether each value of `simdata$treatment` is equal to `"high"`. Notice the two equals signs in `==`, and notice that the value being compared against is in double quotation marks. What would you expect to happen if instead we had entered `simdata$treatment = "high"`, with just one equals sign?

The expression `simdata$treatment %in% c("medium", "high")` produces a logical vector checking whether each element in `simdata$treatment` is either `"medium"` or `"high"`.

Missing data

It is common to be missing some data points in any data set. The built-in representation of missing data in R is `NA`. To create some vectors with missing data in R, we could define the following variables:

```
> x1 = c(1, NA, 4, NA, 7)
> x2 = c("a", "b", NA, "NA")
```

The first vector is a numerical vector with missing data, and the second is a character (or string) vector with missing data. Notice that `NA` is distinct from `"NA"` in `x2`. The first is interpreted as a missing value, the second is interpreted as a string.

Missing data can be handled in a variety of ways in R. The function `is.na()` can be used to determine which values of a vector are missing. For example:

```
> is.na(x2)
```

```
[1] FALSE FALSE  TRUE FALSE
```

The `na.omit()` function can be used to produce an "omit" object, which includes a vector with missing values omitted:

```
> na.omit(x1)
```

```
[1] 1 4 7
attr(,"na.action")
[1] 2 4
attr(,"class")
[1] "omit"
```

The first part of `na.omit(x1)` is the vector `x1`, with missing values removed. `na.omit(x1)` also includes a vector of indices which were previously missing (`2` and `4`). The `attr(,"class")` part of the output is R's way of telling you that this is an omit object. Don't worry about the `attr` things, which are called attributes. If you want to get rid of those attributes, you can use `as.vector(na.omit(x1))` or `as.factor(na.omit(x1))` to convert to a plain vector or plain factor. This isn't recommended.

Some functions have options for how to handle missing data. For example,

```
> sum(x1)
```

```
[1] NA
```

```
> sum(x1, na.rm=TRUE)
```

```
[1] 12
```

By default, `sum(vec)` produces `NA` if any elements in the vector `vec` are missing. With the `na.rm=TRUE` option specified, `sum()` removes any missing values before calculating the sum. You could do the same with `sum(na.omit(x1))`. The `mean()` function also has a `na.rm` option.

Example data: Grueber et al. 2013

In this Lab we will be working with data from Grueber, C.E., Wallis, G.P., and Jamieson, I.G. (2013). Genetic drift outweighs natural selection at toll-like receptor (TLR) immunity loci in a re-introduced population of a threatened species. Molecular Ecology *22*, 4470–4482. The data file `grueber_et_al_2013.csv` is available on the course website, here. Download the data to your current working directory, which should be a folder you have created specifically for Lab 1. (Remember that you can change your current working directory using the `setwd()` function.)

Import the data using the `read.csv()` function, as in Lab 0.

```
setwd("/home/peter/courses/oeb153/labs/lab1/")
gendata = read.csv("grueber_et_al_2013.csv", header=TRUE, na.strings = "")
```

Note that in the import command above, we used `na.strings = ""` to specify that a blank field (i.e., an empty string, `""`) should be interpreted as missing data in the data file. (The authors inputted blank fields where genotypes were missing.)

Check out the data using the `head()` and `dim()` functions. The data consists of a year of birth (cohort) variable, a variable representing survival during the first year, and genotypes at six different toll-like receptor immunity genes. Each row represents a different banded Stewart Island Robin (*Petroica australis rakiura*), with each individual having its own unique banding identifier (the `Band` variable).

Estimating genotype frequencies

The major goal of the Grueber et al. study was to determine how genetic drift and natural selection changed the genetic composition of a bird population after re-introduction. In particular, they were interested in genetic variation at immunity genes known as the Toll-like Receptors (TLRs). TLR loci are involved in the immune response of many organisms (see Wikipedia), and they are thought to be under balancing selection. (This means that individuals with a greater variety of TLR alleles are thought to have higher fitness.)

This work involved estimating many genotype frequencies. In this Lab we will estimate the frequencies (i.e., proportions) of particular genotypes in the re-introduced *P. a. rakiura* population. To do this estimation, we will use **maximum likelihood estimation**.

To do our maximum likelihood estimation, we need a probability model for how the observed data were generated. We will assume that for a particular genotype, there is some true frequency p of the genotype in the population, and that each individual's genotype is independently and identically the genotype of interest with probability p. We want to estimate p.

Under our model, each individual's genotype can be thought of as an i.i.d. Bernoulli random variable representing whether or not the individual has the genotype of interest. If, as in class, $f(x;p)$ is the probability of observing a particular outcome x for one of the random variables, then $f(1;p) = p$ and $f(0;p) = 1 - p$, since each outcome is a Bernoulli random variable.

Grueber et al. made the same assumption throughout their paper, that each genotype is distributed as an i.i.d. Bernoulli random variable. This is an appropriate assumption when sampling with replacement or when sampling without replacement from a large population. However, Grueber et al. actually sampled nearly the entirety of the small, recently re-introduced population (683 out of 722 individuals). This brings into question the i.i.d. Bernoulli assumption, but we will will follow the authors in using this assumption.

Under our model, the likelihood function for p, $L(p)$, is

$$L(p) = \prod_{i=1}^{n} f(x_i; p) = p^g (1-p)^{n-g},$$

where g is the number of individuals with the genotype of interest. An interesting thing to note is that $L(p)$ is proportional to the binomial probability function, which has an additional $\binom{n}{g}$ coefficient out front. This is not coincidence: the number of observed individuals with the genotype of interest has a binomial distribution under our model. Using either the definition of $L(p)$ above or the binomial probability function would produce the same maximum-likelihood estimate of p.

The log-likelihood function is

$$\log L(p) = \log(L(p)) = \log\left(p^g (1-p)^{n-g}\right) = \log\left(p^g\right) + \log\left((1-p)^{n-g}\right) = g\log(p) + (n-g)\log(1-p)$$

Given some number g of individuals with the genotype of interest out of a total of n individuals, we will maximize $\log L(p)$ over different values of p to find the maximum-likelihood estimate of p, $\hat{p}_{\text{MLE}}$.

Maximum likelihood in R

Custom functions In order to carry out maximum-likelihood estimation of a genotype frequency in R, we will need to define a likelihood function. To create a custom function in R, we use the `function()` function, which returns a function that you can call. To create the mathematical function $h(x) = 4x$, the syntax would be as follows:

```
# defining h()
h = function(x){
    return(4*x)
}
# an example call to h(): h(2.4) = 4*2.5 = 10
h(2.5)
```

```
[1] 10
```

We assign the function to a variable h, which stores as its value a function that can be called like `h(2.5)`. We specify the different parameters of our function by listing them in the call to `function()`. The `return()` call inside the function definition says that the value that we want the function to return is `4*x`. The `return()` function doesn't do anything outside of function definitions.

Another example, defining a function with two parameters, such as $g(x, y) = x/3 + 8/y$:

```
# defining g()
g = function(x,y){
    firstTerm = x/3
    secondTerm = 8/y
```

```
    return(firstTerm + secondTerm)
}
# example call to g(): g(2,24) = 2/3 + 8/24 = 1
g(2,24)
```

```
[1] 1
```

In these two example functions, the code defining the function is enclosed in curly brackets (`{}`). The code within these curly brackets is executed every time the function is called. The variables `firstTerm` and `secondTerm` are assigned within the function definition. In R and many other programming languages, this means that these variables aren't accessible outside of the function definition. They have "local scope." A variable assigned outside of a function definition has "global scope," meaning it is accessible from anywhere in the code, including within function definitions.

Genotype frequency (binomial) likelihood function We can now define in R the log-likelihood function we defined mathematically above. For the likelihood function of the genotype frequency p, there are three parameters needed: p, representing the parameter whose likelihood is being calculated, and x and n, representing the binomial data. [That is, $\log L(p)$ should really be written $\log L(p; g, n)$.]

```
loglike = function(p, g, n){
    return(g*log(p) + (n-g)*log(1-p))
}
```

Check that the R function definition corresponds to the expression for $\log L(p)$ above.

If at any point you want to see the code currently sdefining a function, you can call the name of the function without the parentheses. For example, after defining the function `loglike()` above, the command `loglike` shows the code that was used to define it. The ability to check code like this is especially useful and important when you are using someone else's code, imported from a library. By checking the code, you can see exactly what happens when you call the function.

Likelihood maximization Now that we have our likelihood function, we are ready to get genotype data and find the maximum-likelihood genotype frequency.

Grueber et al. observed that the genotype BE at the *TLR4* locus showed unusual behavior, and they focused much of their study on how the frequency of this genotype changed over time. The variable representing genotypes at the *TLR4* locus in the dataset is `gendat$TLR4`.

First, we will focus on calculating the maximum-likelihood estimate of the frequency of the BE genotype amongst Stewart Island Robins born in 2003. To extract the *TLR4* genotypes of the individuals born in 2003, we can use indexing:

```
tlr4.2003 = gendata$TLR4[gendata$Cohort == 2003]
tlr4.2003
```

```
 [1] <NA> <NA> AE   BE   BB   AB   AB   AC   AC   BE   BD   AD   AB   BD
[15] AA   AC   AB   AC   AC   AC   AC   BC   CC   BE   BE   AD   AD   AB
[29] BB   AB   AA   AC   AC   BE   AB
Levels: AA AB AC AD AE BB BC BD BE CC CD CE DD DE EE
```

Here, we've defined a variable with a period in the name. In R, this is fine. In other programming languages, this might mean something else. In R, a variable name can't start with a symbol or a number. Other names are fair game, but be careful not to write over frequently used built-in functions like `mean` and `sd`.

Notice that the first two individuals have missing genotypes. It will be most convenient to remove them from the vector we analyze at this point.

```
tlr4.2003.narm = na.omit(tlr4.2003)
tlr4.2003.narm
```

```
 [1] AE BE BB AB AB AC AC BE BD AD AB BD AA AC AB AC AC AC AC BC CC BE BE
[24] AD AD AB BB AB AA AC AC BE AB
attr(,"na.action")
[1] 1 2
attr(,"class")
[1] "omit"
Levels: AA AB AC AD AE BB BC BD BE CC CD CE DD DE EE
```

Now we can calculate the number of individuals with the BE genotype (g, above) and the total number of genotypes collected (n, the sample size).

```
countBE2003 = sum(tlr4.2003.narm == "BE")
countBE2003
```

```
[1] 5
```

```
sampleSize2003 = length(tlr4.2003.narm)
sampleSize2003
```

```
[1] 33
```

To calculate the log-likelihood of a particular value, say $p = 0.4$, we can call our custom `loglike()` function:

```
loglike(p=0.4, g=countBE2003, n=sampleSize2003)
```

```
[1] -18.88
```

But we want to calculate the log-likelihood of many values of p between 0 and 1, not just 0.4. We can create a vector of values for which to calculate the log-likelihood using the `seq()` function from Lab 0.

```
ps = seq(from=0, to=1, by = 0.0001)
head(ps)
```

```
[1] 0e+00 1e-04 2e-04 3e-04 4e-04 5e-04
```

And we can calculate the log-likelihoods of each value of p in the vector `ps` by calling `loglike()` with `ps` as an argument:

```
loglikes = loglike(p=ps, g=countBE2003, n=sampleSize2003)
head(loglikes)
```

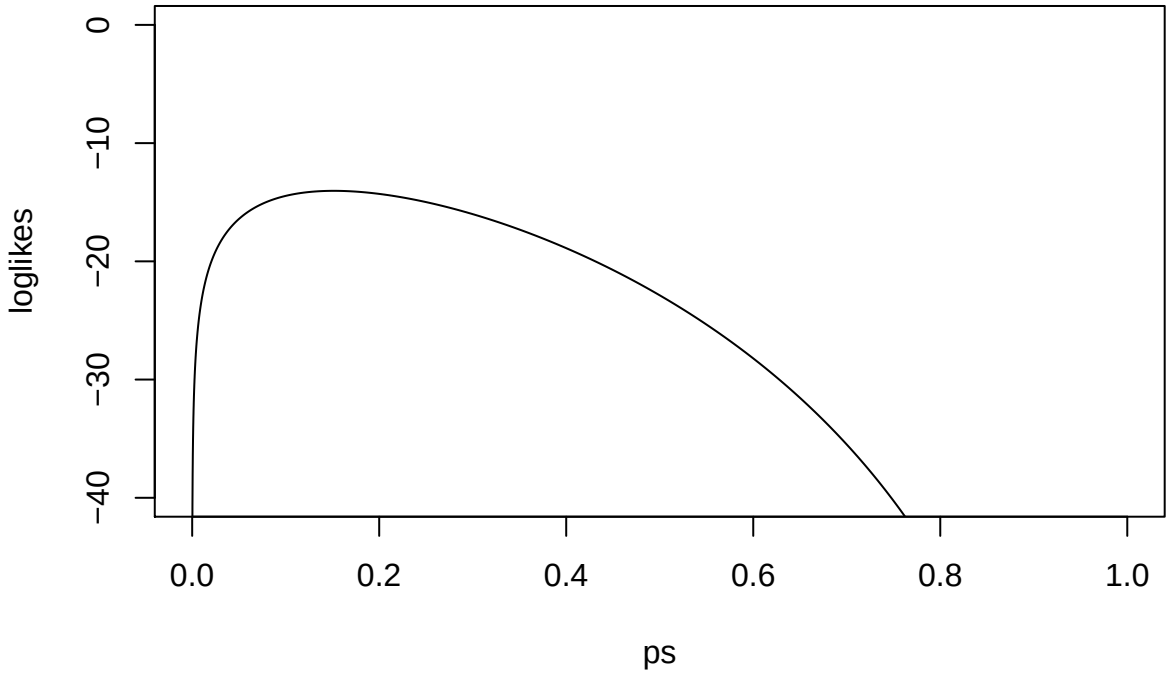
```
[1]   -Inf -46.05 -42.59 -40.57 -39.13 -38.02
```

We can plot the log-likelihood curve using the `plot()` function. To have `plot()` create a line plot rather than a plot of points, we specify that `type = "l"` (for "line").

```
plot(ps, loglikes, type = "l")
```

The likelihood surface looks somewhat flat, and a maximum value is hard to pick out from it. We can change this by changing the limits of the y-axis:

```
plot(ps, loglikes, type = "l", ylim = c(-40,0))
```



Now it's easier to see the part of the curve near the maximum, and we can say that the maximizing value of p is probably somewhere between 0.1 and 0.3. To figure out which of our values of p actually produces the greatest likelihood, we can use R's indexing again:

```
phatMLE = ps[loglikes == max(loglikes)]
phatMLE
```

```
[1] 0.1515
```

Thus the maximum-likelihood estimate of the frequency of the BE genotype amongst individuals born in 2003 is approximately $\hat{p}_{\text{MLE}} \approx 0.1515$. Not surprisingly, this is equal to

$$\hat{p} = \frac{\#\text{ BE's}}{n} = \frac{5}{33} = 0.1515,$$

our method-of-moments estimator of p, the sample mean of Bernoulli random variables. In fact, you can work through the calculus to show that $\hat{p} = g/n$ is the value of p that maximizes the log-likelihood function $\log L(p)$ above, for any g and n.

We can add the our plot a vertical line at $\hat{p}_{\text{MLE}}$ (so long as you haven't closed it) using the `abline()` function. In general, `abline()` adds a line of the form $y = ax + b$ to a plot, but it can also add vertical and horizontal lines to a plot using its `h` and `v` options, respectively.

```
abline(v = phatMLE, col = "blue")
```

The line is blue because we specified `col = "blue"`. `col` is frequently the parameter to change the color of something in R.

Confidence interval estimation

Likelihood contour intervals As mentioned in Lecture on Wednesday (Oct 8), one can construct an approximately 95% confidence interval using the log-likelihood curve. To construct such a confidence interval, we find the values of p that produce a log-likelihood that is within 1.92 log-likelihood units from the maximum log-likelihood. The maximum log-likelihood (i.e., the log-likelihood of $\hat{p}_{\text{MLE}}$) is -14.0358 and can be found using `max(loglikes)`.

To find the values of p that produce a log-likelihood within 1.92 of `max(loglikes)`, we use indexing again:

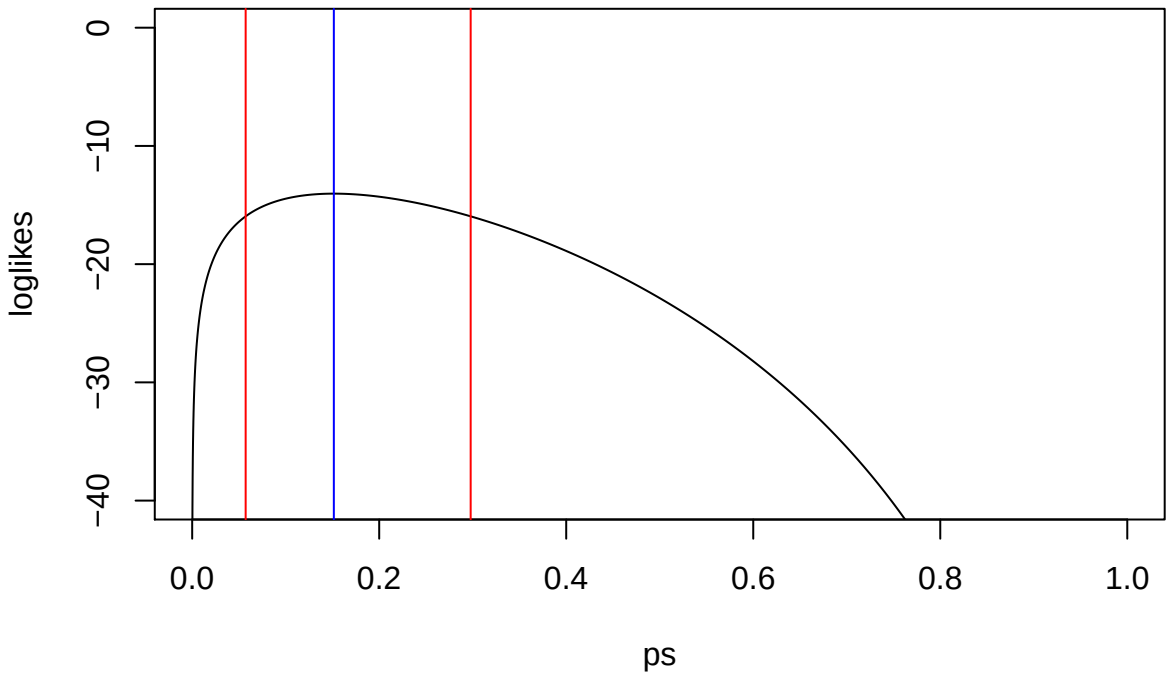
```
interval.ps = ps[loglikes > max(loglikes)-1.92]
likelihoodCI = c(min(interval.ps), max(interval.ps))
likelihoodCI
```

```
[1] 0.0572 0.2978
```

In the above, we constructed the interval by creating a vector of the minimum and maximum values of p that produce log-likelihoods within 1.92 of the maximum log-likelihood. The interval is (0.0572, 0.2978). Notice that this confidence interval is asymmetric around $\hat{p}_{\text{MLE}} = 0.1515$.

Add red vertical lines showing the confidence interval:

```
abline(v = likelihoodCI, col = "red")
```



`abline()` added two lines because there are two elements in `loglikelihoodCI`.

Wald's method of constructing confidence intervals (CLT approximation) As we learned in class, there are multiple ways of constructing confidence intervals for proportions. We can also use the Central Limit Theorem approximation, specifically that $\hat{p}$ is approximately normally distributed. This is also called Wald's method. The approximately 95% confidence interval produced from the Central Limit Theorem is

$$\left(\hat{p} - 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}, \hat{p} + 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right).$$

To calculate this in R, we need to calculate $1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$.

```
phat = countBE2003/sampleSize2003
waldsDistance = 1.96*sqrt(phat*(1-phat)/(sampleSize2003))
waldsCI = c(phat-waldsDistance, phat+waldsDistance)
waldsCI
# [1] 0.02918 0.27385
abline(v = waldsCI, col="orange")
```

External libraries As was mentioned in Lab 0, one of R's greatest strengths is the huge wealth of libraries (packages) written and shared by R users. People write functions for a huge variety of things related to statistics, including confidence interval estimation for proportions.

We will use a function called `binom.confint()` from the library `binom` to produce a confidence interval using the Agresti-Coull method. This library doesn't come pre-installed with R, so you will need to install it using the following command:

```
install.packages("binom")
```

A dialog may come up, asking you which mirror you would like to download the package from. Pick one.

To load the library, use the `library()` function:

```
library(binom)
```

Notice that the name of the package is surround in quotes in `install.packages()` but not in `library()`. If R doesn't say anything after calling the `library()` function, the library loaded. You should now be able to use the `binom.confint()` function. After checking `?binom.confint`, you would know to calculate the Agresti-Coull confidence interval using the following command:

```
ACresult = binom.confint(countBE2003, sampleSize2003, 0.95, methods = "ac")
ACresult
```

```
         method x  n   mean   lower upper
1 agresti-coull 5 33 0.1515 0.06173 0.314
```

```
agrestiCoullCI = c(ACresult$lower, ACresult$upper)
agrestiCoullCI
```

```
[1] 0.06173 0.31398
```

Notice also that `binom.confint()` returned a single-row data frame object with variables `ACresult$lower` and `ACresult$upper` representing the upper and lower bounds of the confidence interval. To get the confidence interval into the same vector form that we have used to represent the previous confidence intervals, we used `c()`.

If we had specified `methods = "profile"`, `binom.confint()` would have produced the same confidence interval we calculated based on the 1.92 drop in the log-likelihood. You can check this.

Estimating many genotype frequencies

Figure 3A in Grueber et al. shows how the frequency of the *BE* genotype at the *TLR4* locus changes in cohorts born between 2001 and 2009. As part of the assignment for this Lab, you will replicate part of this figure.

We've just calculated $\hat{p}_{\text{MLE}} = \hat{p}$ for the cohort in 2003, and we could go through this process several more times to calculate $\hat{p}$ for the other years. This would be tedious. To make this process more efficient, we will use a **for loop**.

For loops are one of the most useful parts of almost all programming languages. A for loop allows you to iteratively repeat some calculation, with some aspect of the calculation varying with each iteration. As an example in R:

```
eightToTwelve = 8:12
for(i in eightToTwelve){
    print(i)
    y = (i+1)/2
    print(y)
}
```

```
[1] 8
[1] 4.5
[1] 9
[1] 5
[1] 10
[1] 5.5
[1] 11
[1] 6
[1] 12
[1] 6.5
```

Here, we created a vector `eightToTwelve` containing $(8, 9, 10, 11, 12)$. The for loop iterates through the elements in `eightToTwelve`, temporarily storing each element as `i`. During each iteration, R executes the code between the curly brackets `{}`. It prints (i.e., outputs) `i`, stores the value of `(i+1)/2` in the variable `y`, and then prints the value of `y`. After outputting `y`, the loop returns to perform the same calculations on the next value of `i` until it's performed the calculations for every element in `eightToTwelve`.

If we wanted to store the different values of `y` that are calculated in each iteration, we could create an empty vector called `result` and then store each calculated value of `y` at the correct position in `result`. Instead of looping through `eightToTwelve` itself, we would loop through its indices, `1:length(eightToTwelve)`.

```
numIterations = length(eightToTwelve) # should be 5
result = numeric(numIterations)       # create empty vector for results
for(i in 1:numIterations){     # iterating through indices now, rather than 8:12 itself
    x = eightToTwelve[i]       # store this iteration's element of eightToTwelve in x
```

```
    print(x)
    result[i] = (x+1)/2
}
```

```
[1] 8
[1] 9
[1] 10
[1] 11
[1] 12
```

```
result
```

```
[1] 4.5 5.0 5.5 6.0 6.5
```

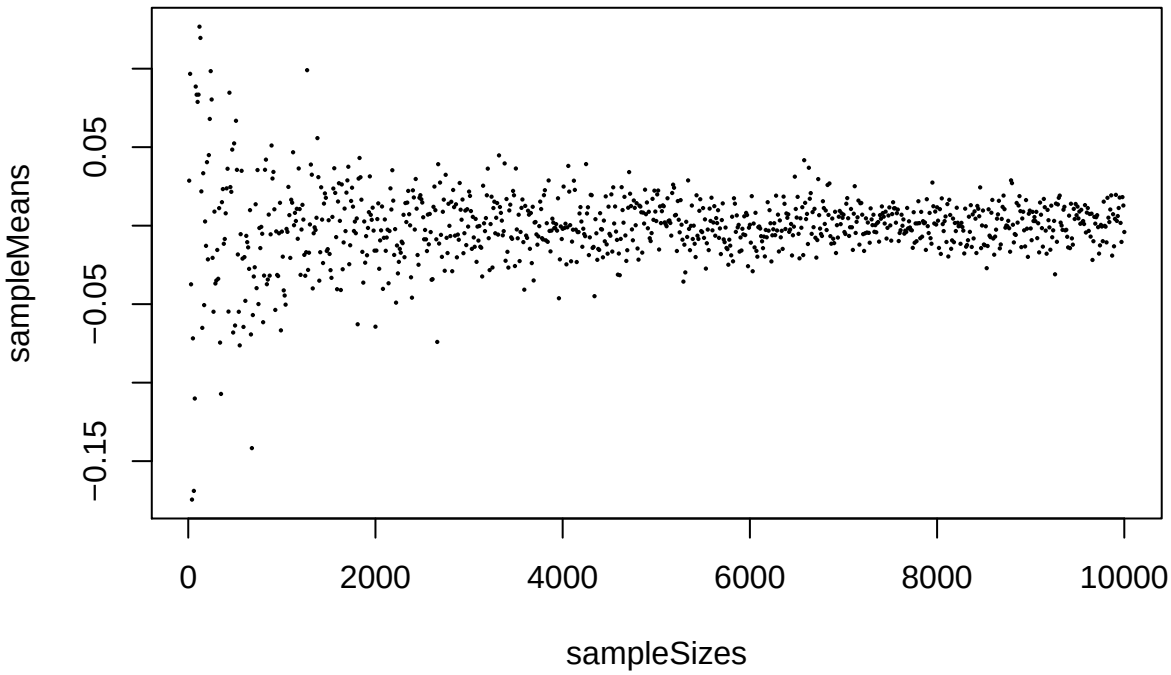
The `result = numeric(numIterations)` statement creates the empty numerical vector `result`, with length `numIterations`. The loop then places each iteratively calculated value of `(x+1)/2` into the currently-indexed position in the vector `result`.

Of course we could also calculate `result` using a simple assignment in R: `result = (eightToTwelve+1)/2`. Not all results will be so easy to calculate, though, and it will be necessary in many cases to use a for loop.

As a somewhat more typical example, imagine that we wanted to calculate the sample mean of a sample of size j taken from a standard Normal distribution, for each value of j between 10 and 10000, by 10. The way to do that would be with a for loop.

```
sampleSizes = seq(from = 10, to=10000, by=10) # the different sample sizes j, (10, 20, ..., 9990, 10000
sampleMeans = numeric(length(sampleSizes)) # empty vector for storing the sample means.
numIterations = length(sampleSizes)
for(i in 1:numIterations){
    curSampleSize = sampleSizes[i]  # get the i'th sample size
    curSample = rnorm(curSampleSize, mean = 0, sd = 1) # get a sample of size curSampleSize
    sampleMeans[i] = mean(curSample) # calculate and store the i'th sample mean.
}

plot(sampleSizes, sampleMeans, cex = 0.3, pch = 16)
```



(Note that this is not quite demonstration of the Law of Large Numbers from lecture, but it's similar!)

Conclusion and Lab 1 assignment

Conclusion

At this point, we have covered the new material for Lab 1.

If you've never seen for loops before, they may take some practice getting used to. For loops are some of the most important and useful tools in computer programming. They allow you to make calculations that would be practically impossible otherwise.

In this Lab we also started to use R's graphics functionality more fully than we did in Lab 0. It is possible to make publication-quality graphics using just R. There are many graphical parameters that allow you to tweak how a graph looks. Many of these are options to the `plot()` function. A few of the most commonly used options are given here.

option	description
`xlab`	x-axis label
`ylab`	y-axis label
`xlim`	x-axis limits
`ylim`	y-axis limits
`main`	graph title
`type`	type of graph
`lty`	line type, for line graphs
`lwd`	line width
`pch`	point character, for scatterplots
`cex`	point size, for scatterplots

option	description
`col`	point or line color

For other options, see `?plot` and `?par` (for graphical parameters). Additional functions can be called after `plot()` has been called to add features to graphs. `abline()`, which we used in this lab, is an example. Others include `legend()`, `points()`, `axis()`, and `text()`. Plots types are not limited to line plots, scatter plots, and histograms, either. Most types of plots are available in R; see Quick-R or Google.

Lab 1 assignment

For this Lab's assignment, download `lab1_assignment.R` from the course website. This R script contains commented-out questions, to which you will respond by writing *well-commented* code within the relevant sections of the script. Email me (`pwilton@fas.harvard.edu`) your script by 10am on October 24. Optionally, you can save and send the plots that are part of the assignment as well, but this is not strictly necessary because I should be able to run your script to produce the plots.