Because you're in the middle of your midterm, we'll keep it light this section. We're first going to step away from the world of OLS to examine the maximum likelihood (ML) estimator. Then, very quickly we'll look at how to generate dummy variables in `R`.

Maximum likelihood

Maximum likelihood was covered in Charlie's class, so you should be familiar with the basics. We'll take a brief look at the theory and go on to some empirical examples that demonstrate how to perform ML using `R`.

Before that, though, let's begin with some intuition. The general principle behind ML estimation is that we have observe some data vector $\boldsymbol{z}$ which we assume to be drawn from a probability distribution $f(\boldsymbol{z};\theta)$, where θ is a vector of parameters that characterize the distribution. Once we pick the distribution, we then use either analytical or computational methods to determine the $\hat{\theta}$ that best explains the data $\boldsymbol{z}$ that we observe. In other words, in order to find $\hat{\theta}$, we simply search for the parameters that maximize the probability of observing the values of our data $\boldsymbol{z}$.

Now, let's dive into the estimation process with some specific examples.

ML on a Bernoulli distribution

The outcome of flipping a (potentially rigged) coin is described by the Bernoulli distribution, which is a special case of the binomial distribution. A Bernoulli random variable is 1 with probability p and 0 with probability $1-p$. p is the sole parameter to characterize this distribution, so in this case we have that $\theta \equiv p$.

Suppose that we observe a sequence of Bernoulli draws, each with the same unknown p. From the sequence, which looks something like $\boldsymbol{x} = [0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ ...]$, we want to estimate p using $\hat{p}$. We can do this using ML.

As we have done before, we'll use a simulation method so that we can set the "true" data to test our method[1]. First, we set up preliminaries and create our (very long) sequence of flips:

```
set.seed(3092015)
flips <- 1000
p.true <- 0.59
x <- rbinom(n = flips, size = 1, prob = p.true)
head(x)
```

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

Next, we'll use probability theory to define our likelihood function, which is just the joint probability of observing the particular sequence of $\boldsymbol{x}$ that we see, given that each flip is identically and independently distributed Bernoulli, and given some (not necessarily accurate or optimally determined)

[1] This section is based on a similar tutorial by John Myles White, available at: `http://www.johnmyleswhite.com/notebook/2010/04/21/doing-maximum-likelihood-estimation-by-hand-in-r/`.

$\bar{p}$. Remember, our goal is to choose $\bar{p}$ to maximize the likelihood function.

$$\mathrm{L}(\bar{p}) = \prod_{i=1}^{1000} \bar{p}^{x_i}(1-\bar{p})^{1-x_i}$$

We can take logs of the likelihood function to produce the often-more-tractable log-likelihood function:

$$\mathscr{L}(\bar{p}) = \sum_{i=1}^{1000} x_i \ln(\bar{p}) + (1-x_i)\ln(1-\bar{p})$$

By taking the derivative of the log likelihood function we can show[2] analytically that the best estimate of p is the sample mean of $\boldsymbol{x}$. This gives us a benchmark against which we can compare the estimates we get from optimization.

```
analytical.est <- mean(x)
```

We can actually optimize over either the likelihood or the log-likelihood function. Let's create both of them as functions that take in `p`, our guess at the value for p, and `x`, our sequences of 0s and 1s. The output of both functions will be the likelihood (or log-likelihood) of observing `x` given our guess at `p`.

```
likelihood <- function(p, x) {
  prod(p^x * (1-p)^(1-x))
}

log.likelihood <- function(p, x) {
  sum(x * log(p) + (1-x) * log(1-p))
}
```

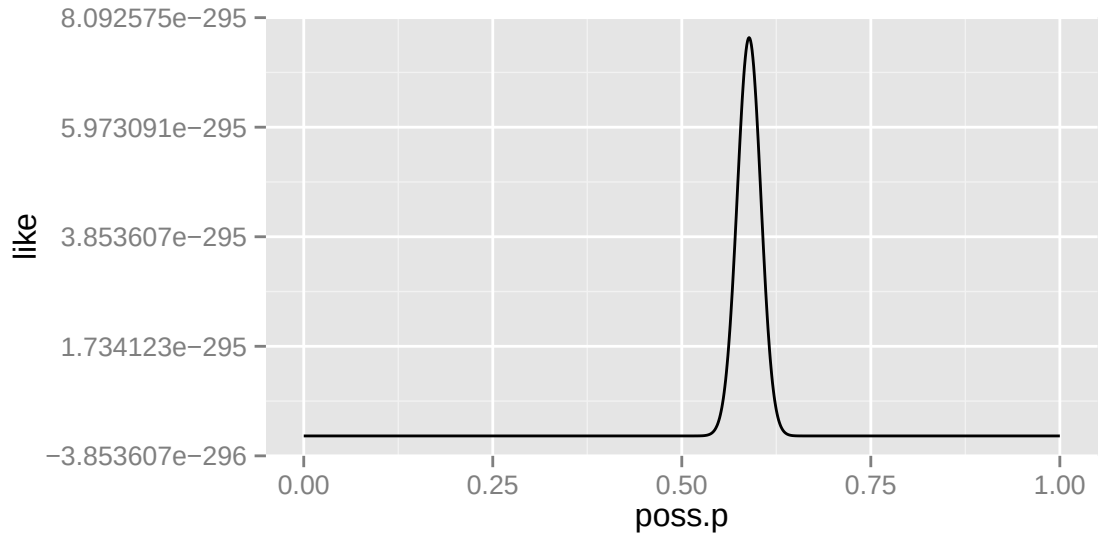
Now, let's use `sapply()` to calculate the values of the these functions over the range $[0, 1]$, since we know that the true value of p has to be within that range.

```
poss.p <- seq(0,1,0.001)
like <- sapply(poss.p, likelihood, x = x)
loglike <- sapply(poss.p, log.likelihood, x = x)
```

So what is this thing we're trying to maximize, exactly? Let's find out. Using `ggplot2`, we plot the likelihood function:

```
mydata <- data.frame(poss.p, like, loglike)
library(ggplot2)
ggplot(data=mydata, aes(x=poss.p, y=like))  + geom_line()
```

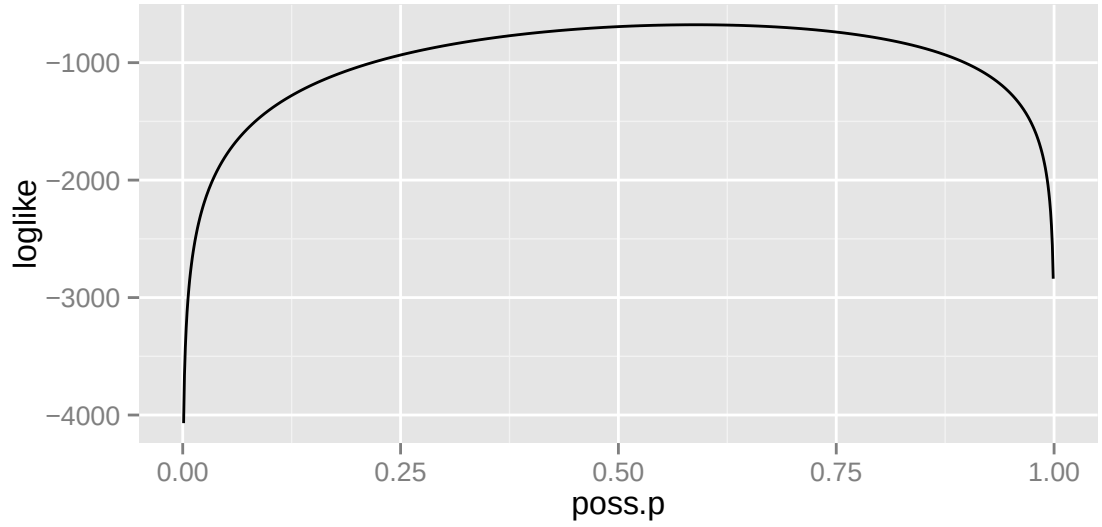
[2] But won't. See http://mathworld.wolfram.com/MaximumLikelihood.html.



As you can see, the, function seems to peak sharply around $p = 0.60$. That's pretty good! Since log is a monotonically increasing function[3], we should expect to see the maximum at the same point p:

```
ggplot(data=mydata, aes(x=poss.p, y=loglike)) + geom_line()
```

```
## Warning:  Removed 2 rows containing missing values (geom_path).
```



And we do! More or less. However, if we want to be more precise about our estimate, we ought to use an optimization algorithm. Let's try it, first with the likelihood function.

```
opt.like <- optimize(f = likelihood, c(0,1), maximum = TRUE, x = x)
cbind(opt.like$maximum, opt.like$objective)
```

```
##           [,1]          [,2]
```

[3] i.e. $x > y \leftrightarrow \ln(x) > \ln(y)$

```
## [1,] 0.5890176 7.707209e-295
```

That's pretty darn close to our analytical estimate, the sample mean, which was 0.589. `optimize()` is a very straightforward function: with this call, we pass it the function we want to optimize, our limits on the parameter of interest p, a flag that indicates we want it to maximize the likelihood function, and an additional parameter that we pass through to the likelihood function. We can use it again to find the estimate of p given by the log-likelihood function:

```
opt.loglike <- optimize(f = log.likelihood, c(0,1), maximum = TRUE, x = x)
cbind(opt.loglike$maximum, opt.loglike$objective)
```

```
##           [,1]      [,2]
## [1,] 0.5889968 -677.2204
```

Cool. You'll recall from lecture that we use the log-likelihood because it provides analytical convenience — it's typically much simpler to take the derivative over a sum than a product. But you might not know that the log-likelihood has computational advantages as well! To see this, try setting `flips = 100000`. Cover your ears.

Generating dummy variables in R

This is easy. Let's go back to the `iris` dataset and turn the `Species` variable into dummies. First we'll do this manually using conditional statements and then replicate the results using a canned function.

```
data(iris)
dummiesManual <- sapply(unique(iris$Species), function(spec){
  as.numeric(iris$Species == spec)
})
dummies <- model.matrix(~ Species - 1, data = iris)
all(dummies == dummiesManual)
```

```
## [1] TRUE
```

The manual version uses an `sapply` loop to go through each of the unique elements of `iris$Species` and generate the dummy by coercing the `logical` vector `iris$Species == spec` to a `numeric` (`TRUE` $\rightarrow$ 1, `FALSE` $\rightarrow$ 0). The `sapply` is smart enough to also `cbind` these vectors for us. Great!

The canned version is a tiny bit simpler. The tricky part is getting what the `model.matrix()` function is actually doing. This function takes a formula call (you can recognize these as they contain variable names and a '~') and spits out a matrix (intended for use in OLS) corresponding to the right hand side of the formula (what we've been calling the `X` matrix usually). Note that we have to tell the formula to remove the intercept by adding `- 1`. While this was not the original motivation for writing this function, this gives us the ability to nicely generate dummy variables of categorical `factor` or `character` variables. This method will not work for `numeric` variables.

What about `numerics`?

Say we want to split the `Sepal.Length` variable into quartile categories. We can use the `quantile()` function to get the splits and then use the `ifelse()` function to generate the dummies in an `sapply` loop.

```
quartiles <- quantile(iris$Sepal.Length)
dummiesNumericManual <- sapply(1:(length(quartiles) - 1), function(i) {
    if (i == 1) {
        ifelse(test = iris$Sepal.Length <= quartiles[i + 1], yes = 1, no = 0)
    } else if (i <= length(quartiles) - 2) {
        ifelse(test = iris$Sepal.Length > quartiles[i] & iris$Sepal.Length <=
            quartiles[i + 1], yes = 1, no = 0)
    } else {
        ifelse(test = iris$Sepal.Length > quartiles[i], yes = 1, no = 0)
    }
})

# And using canned functions
quartileCuts <- quantile(iris$Sepal.Length)
cuts <- cut(iris$Sepal.Length, unique(quartileCuts), include.lowest = TRUE)
dummiesNumeric <- model.matrix(~cuts - 1)
all(dummiesNumericManual == dummiesNumeric)
```

```
## [1] TRUE
```

Here, the manual version is a lot messier. The `quantile()` function generates our cut points and we go through each bin and generate the dummy. I have included separate cases for the first and last elements, though this is only necessary for one (Why?).

The `cuts()` function does most of the work for us in the canned version. This function generates factor variables for each category defined by given cut points. We then use the `model.matrix()` method from before to finish the job.

Note that we also have access to the `dummies` package on CRAN, but it seems to have a coding error for individual variables, so I'd avoid using it.