This section, we'll take another crack at ggplot2 before spending the remainder of our time on Maximum Likelihood estimation.

The grammar of ggplot2

Last week we demonstrated the use of ggplot2 as an alternative to the R base graphics without really talking about how ggplot2 works or why it might be a better graphing option. Today we'll attempt to fill that gap[1].

ggplot2 is an implementation of the "grammar of graphics," described in a book of the same title by Leland Wilkensen. The idea is that graphical representations of data, like language, have a logical grammatical structure. Most graphing packages ignore this structure and create one-off solutions for every different kind of graph that we might want to display. This is inefficient, and therefore displeasing to economists. ggplot2 allows users to control the composition of statistical graphs by directly controlling the grammar of the graphical components.

Plots in ggplot2 are built by putting together separate component parts. The three crucial components that we'll think about for now are:

- data
- aesthetics
- layers/geometric shapes

There are more, but these are the important ones. We'll tackle each separately.

Data

The *data* for ggplot2 should always be packaged into a data frame. After loading the ggplot2 library, we'll load the R iris dataset to demonstrate:

```
library(ggplot2)
data(iris)
ggplot(data = iris, ... ) # not a real command
```

The first argument we pass to ggplot() will be the data frame that we intend represent graphically.

Aesthetics

The second required argument for ggplot() is the *aesthetic mapping* of the plot. You might think of aesthetics here as "things that you can see", such as the position of the data on the axes, the color, the shape, et cetera. We use this argument to map the data we have into the aesthetics that we're going to display. Now we can create and display the ggplot2 object g using our data an aesthetics.

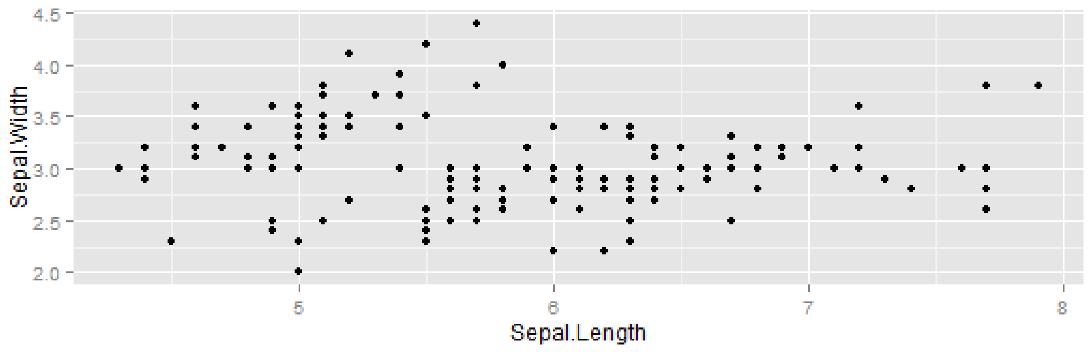
```
(g <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)))
```

[1]Sources for this section: http://vita.had.co.nz/papers/layered-grammar.pdf and http://www.slideshare.net/AndrewZieffler/unit-02ggplot2.

Layers/geometric shapes

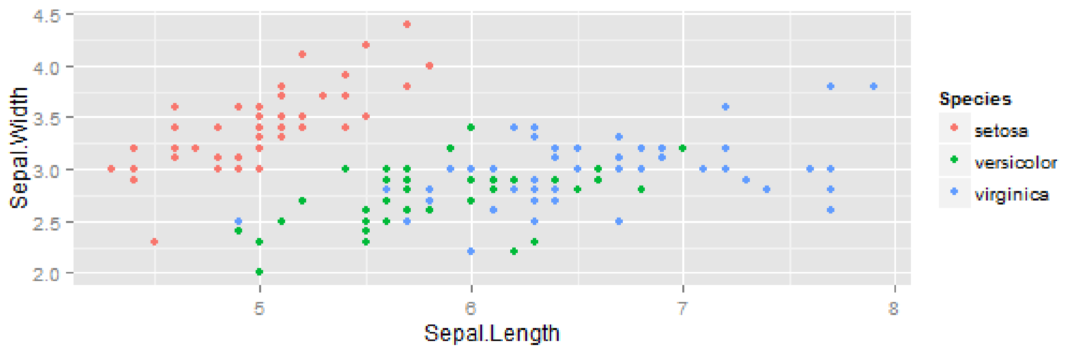
Or not. Why are we getting an error? We've specified our data, and our aesthetics, but not our graphical layers (i.e. geometric shapes). Here, we'll add a layer of points:

```
g + geom_point()
```



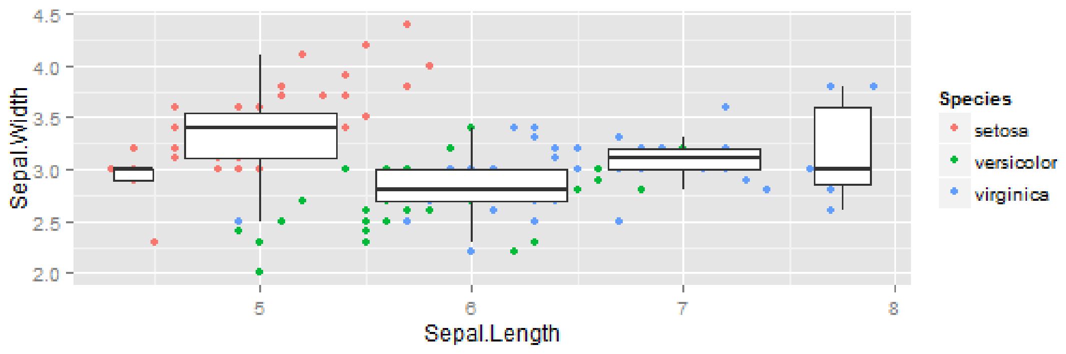
We can also specify additional aesthetic options for each layer. Below, we'll tell `ggplot2` to graph the points again, this time specifying that each species should have a different color. Aesthetic options specified in the `ggplot()` function are the default for all layers, but aesthetics specified within layers can override the defaults for that layer only.

```
g + geom_point(aes(color=Species))
```



We're not limited to scatterplots. Although it's a little nonsensical, we can add boxplots of `Sepal.Width` to the current graph, after sorting the groups into their (rounded) `Sepal.Length` categories.

```
g + geom_point(aes(color=Species)) + geom_boxplot(aes(group = round(Sepal.Length)), outlier.size
```



Initially, putting together the grammar of `ggplot2` may seem cumbersome. In fact, the code to construct simple scatterplots or histograms in `ggplot2` is almost certainly going to be more complex than a simple `plot()` or `hist()` from the base graphics package[2]. But as your graphics needs become more complex, you will almost certainly find that `ggplot2` scales much better and is far more powerful than the base functions provided by `R`.

Maximum likelihood

In this section, we're going to step away from the world of OLS to examine the maximum likelihood (ML) estimator. Max has already covered the theory behind ML in class, so we'll be exploring the theory with some empirical examples that demonstrate how to perform ML using `R`.

Before that, though, let's begin with some intuion. The general principle behind ML estimation is that we have observe some data vector $\mathbf{z}$ which we assume to be drawn from a probability distribution $f(\mathbf{z};\theta)$, where θ is a vector of parameters that characterize the distribution. Once we pick the distribution, we then use either analytical or computational to determine the $\hat{\theta}$ that best explains the data $\mathbf{z}$ 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 $\mathbf{z}$.

If you're like me, you found the above explanation mostly baffling the first few times you heard it. Rather than repeating it to you once more, let's dive into the estimation process with some specific examples.

ML on a Bernouilli distribution

The outcome of flipping a (potentially rigged) coin is described by the **Bernouilli distribution**, which is a special case of the binomial distribution. A Bernouilli 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 Bernouilli draws, each with the same unknown p. From the sequence, which looks something like $\mathbf{x} = [0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ \ldots]$, 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[3]. First, we set up preliminaries and create our (very long) sequence of flips:

```
set.seed(42)
flips <- 1000
p.true <- 0.59
x <- rbinom(flips, 1, p.true)
head(x)
```

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

[2]In fact, `ggplot2` provides a function called `qplot()` that replicates the simpler syntax from the base graphing package, if you prefer.

[3]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/`.

Next, we'll use probability theory to define our likelihood function, which is just the joint probability of observing the particular sequence of $\mathbf{x}$ that we see, given that each flip is identically and independently distributed Bernoulli with parameter given some $\tilde{p}$. Remember, our goal is to choose $\tilde{p}$ to maximize the likelihood function.

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

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

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

By taking the derivative of the log likelihood function we can show[4] analytically that the best estimate of p is the sample mean of $\mathbf{x}$. This gives us a benchmark against which we can compare the estimates we get from optimization.

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

```
[1] 0.619
```

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))
}
```

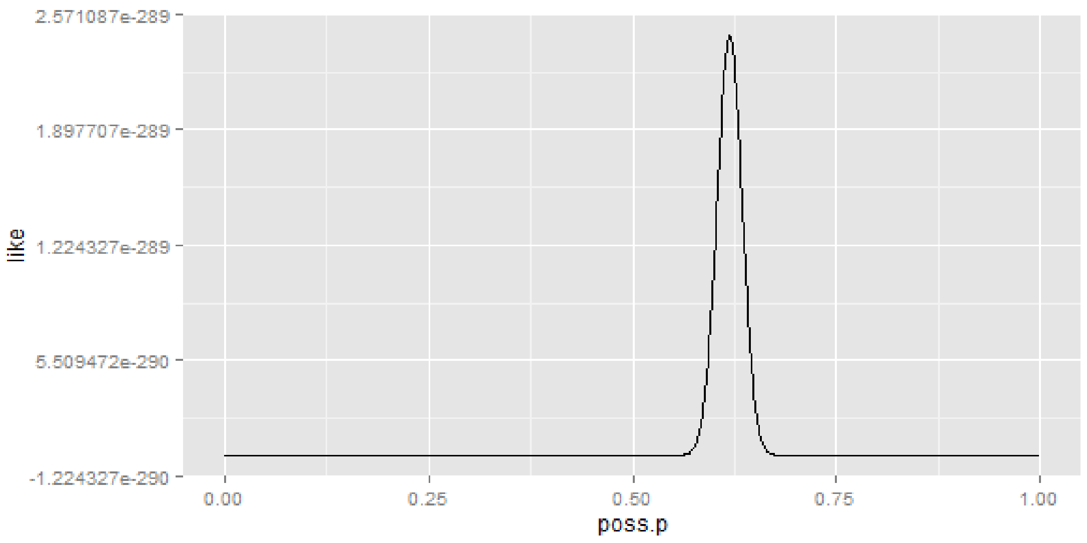
Now, let's use `sapply()` to calculate the values of the these functions over the range $[0, 1]$, since we know that the true value of p has to be within that range.

```
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:

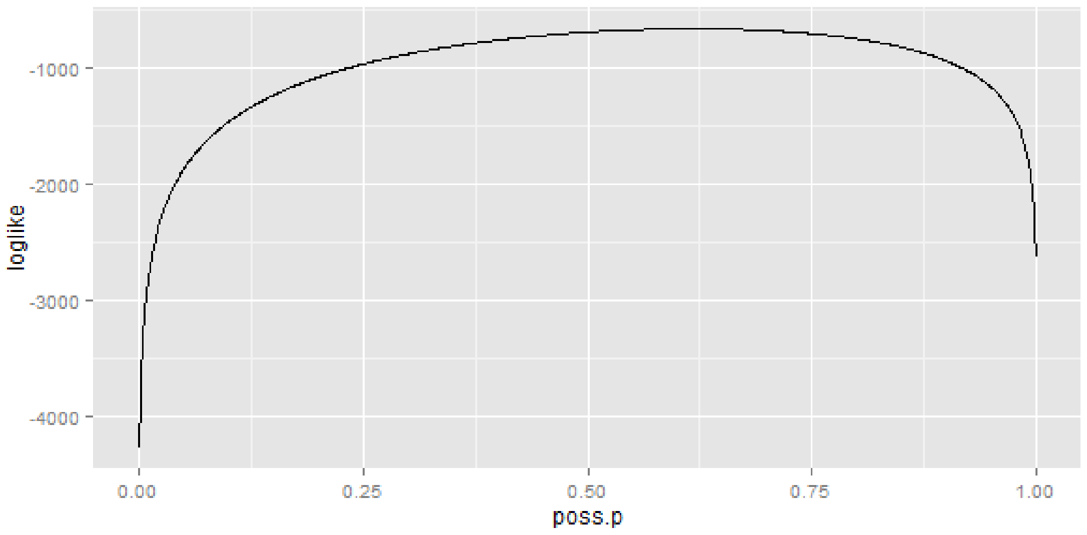
```
data <- data.frame(poss.p,like,loglike)
library(ggplot2)
ggplot(data=data, aes(x=poss.p, y=like))  + geom_line()
```

[4]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 monotone operator[5], we should expect to see the maximum at the same point p:

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



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 = T, x = x)
cbind(opt.like$maximum, opt.like$objective)
```

```
          [,1]          [,2]
[1,] 0.6189997 2.448654e-289
```

That's pretty darn close to our analytical estimate, the sample mean, which was `0.619`. `optimize()` is a very straightforward function: with this call, we pass it the function we want to optimize, our

[5]I.E. $x > y \leftrightarrow \ln(x) > \ln(y)$

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 = T, x = x)
cbind(opt.loglike$maximum, opt.loglike$objective)
```

```
          [,1]      [,2]
[1,] 0.6190052 -664.5516
```

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.

Linear regression and maximum likelihood
Now that we've got a little maximum likelihood experience under our belts, let's see if we can replicate the theoretical results from lecture by fitting a linear regression model using ML rather than OLS. To use ML, we need to make some sort of distributional assumption that we can use to estimate our parameters. Following the lecture notes (Section 2.7.3), we use the convenient assumption A6, which gives us that $\varepsilon|\mathbf{X} \sim \mathbf{N}(\mathbf{0}, \sigma^{\mathbf{2}}\mathbf{I_n})$. From the linearity of our model, we get that $\mathbf{y}|\mathbf{X} \sim (\mathbf{X}\beta, \sigma^2\mathbf{I}_n)$, which we plug into the pdf for a multivariate normal distribution. Taking logs over the distribution gives us:

$$\log L(\tilde{\mathbf{b}}, \tilde{\sigma}^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(\tilde{\sigma}^2) - \frac{1}{2\tilde{\sigma}^2}(\mathbf{y} - \mathbf{X}\mathbf{b})'(\mathbf{y} - \mathbf{X}\mathbf{b})$$

Following Max's notes and solve for $\mathbf{b}$ and σ^2 analytically, we should find that $\mathbf{b}_{ML} = \mathbf{b}_{OLS} = (X'X)^{-1}X'y$ and that $\sigma^2 = \frac{\mathbf{e}'\mathbf{e}}{n}$. **Or**, we could think jointly estimate $\mathbf{b}$ and σ^2 using the `optim` command (which is similar to `optimize`, except that it is capable of optimizing over more than one dimension at once).

```
n <- 2000
set.seed(42)
X <- cbind(1,runif(n))
eps <- rnorm(n)
b.true <- rbind(5,3)
y <- X %*% b.true + eps

log.likelihood.optim <- function(theta) {
  sigma2 <- tail(theta, n = 1)
  b <- theta[1:nrow(b)]
  e <- y - X %*% b
  output <- -n/2*log(2*pi) - n/2*log(sigma2) - 1/(2 * sigma2) * t(e) %*% e
  return(-output)
}
optim(par = c(3,2,1), fn = log.likelihood.optim)$par
```

```
[1] 4.988389 3.014805 1.015007
```

The first argument to optim is a set of starting parameters for θ. In general, in order to maximize our chances of finding the correct optimum (and to increase the rate of convergence), we want to choose reasonable parameters. The second argument is just the log-likelihood function that we created. Let's confirm that OLS gives the same parameter estimates for β as ML.

```
OLS <- function(y,X) { b <- solve(t(X) %*% X) %*% t(X) %*% y }
(b <- OLS(y,X))
```

```
         [,1]
[1,] 4.988142
[2,] 3.015419
```

Whew! They're nearly exactly the same! We can also calculate the s^2_{OLS} and the $\tilde{\sigma}^2_{ML}$, which we know to be different.

```
e <- y - X %*% b
k <- ncol(X)
cbind(s2.OLS <- (t(e) %*% e) / (n - k), s2.ML <- (t(e) %*% e) / n)
```

```
         [,1]     [,2]
[1,] 1.016147 1.015131
```