

In this section we'll briefly clarify some concepts from last time, and then we'll build up a simulation in R to show why living in asymptotic world is great.

The difference between prediction and confidence intervals

Last week I punted on demonstrating the difference between prediction and confidence intervals. It turns out that we had already demonstrated the difference between the two of them, I just didn't label them correctly. To remind you, we had the following population model:

$$\mathbf{y} = \mathbf{X}\mathbf{b} + \boldsymbol{\epsilon}$$

We computed that the variance of the *individual prediction*, $V(\hat{y}^0 - y^0)$ and the variance of the *mean prediction*, $V(\hat{y}^0)$, and showed they are different estimates¹:

$$V(\hat{y}^0 - y^0) = \sigma^2(1 + \mathbf{X}^0(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'^0) \quad (1)$$

$$V(\hat{y}^0) = \sigma^2(\mathbf{X}^0(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'^0) \quad (2)$$

It turns out that when we compute the *prediction interval* (i.e. the interval around the individual prediction), we use (1), while when we compute the *confidence interval* (i.e. the interval around the mean prediction), we use (2). Hopefully this terminology makes sense to you. The intuition is that the prediction interval tells us the error we should expect when we predict *new* values of y , whereas the confidence interval tells us the confidence we should have in the estimate $\hat{y}^0 = \mathbf{X}^0\mathbf{b}$.

The source of the confusion was that `predict()`, the canned R command, returns the standard error of the prediction (i.e. the square root of (1)), when we set `se.fit = TRUE`. But we can also use `interval` setting to return both types of intervals! Let's dive into R to demonstrate a key fact about the relationship between prediction and confidence intervals. First, we'll build up our simulated data:

```
OLS <- function(y,X) { b <- solve(t(X) %*% X) %*% t(X) %*% y }
set.seed(42)
n <- 200
X <- cbind(1, rnorm(n))
beta <- c(2, 3)
eps <- rnorm(n)
y <- X %*% beta + eps
```

Now we can construct both the standard errors for the prediction interval and the confidence interval. We'll name them intuitively.

```
b <- OLS(y,X)
pred.y <- X %*% b
e <- y - pred.y
n <- nrow(X); k <- ncol(X)
```

¹Simon Jackman has a nice, complete exploration of this topic here: <http://jackman.stanford.edu/classes/350B/07/predictionforWeb.pdf>.

```

s2 <- as.vector(t(e) %*% e / (n - k))
XpXinv <- solve(t(X) %*% X)
pred.se <- sqrt(diag(s2 * (1 + X %*% XpXinv %*% t(X))))
conf.se <- sqrt(diag(s2 * (0 + X %*% XpXinv %*% t(X))))
(cbind(head(pred.se), head(conf.se)))

```

```

      [,1]      [,2]
[1,] 0.9536325 0.11724104
[2,] 0.9494816 0.07645859
[3,] 0.9491422 0.07212048
[4,] 0.9498495 0.08089918
[5,] 0.9492266 0.07322351
[6,] 0.9487766 0.06713905

```

As with our coefficient estimates, we'll calculate the confidence intervals using the 95% critical value from a t_{n-k} distribution.

```

crit.t <- qt(0.975,n-k)
confint.t <- cbind(pred.y - crit.t * conf.se, pred.y + crit.t * conf.se)
predint.t <- cbind(pred.y - crit.t * pred.se, pred.y + crit.t * pred.se)

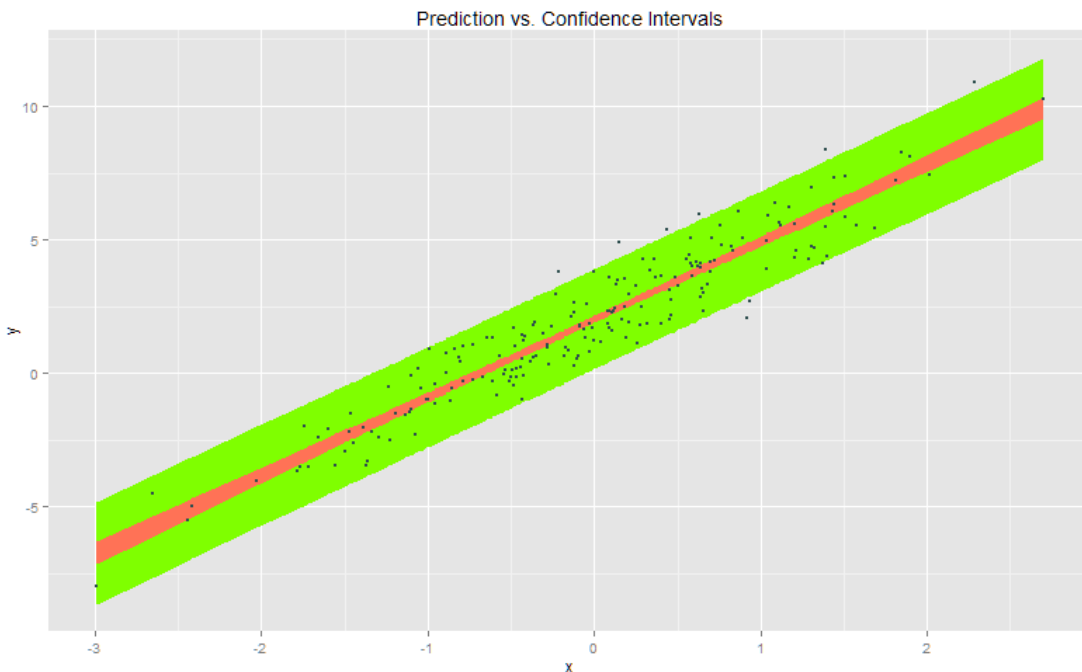
```

We could have also calculated and checked these using `predict()`. But they're right. Trust me. Now, let's graph these to see what we have!

```

draw.df <- data.frame(X[,2], y, pred.y, confint.t, predint.t)
names(draw.df) <- c("x", "y", "pred.y", "conf.low", "conf.high", "pred.low", "pred.high")
library(ggplot2)
(g <- ggplot(data = draw.df, aes(x = x, y = y)) + ggtitle("Prediction vs. Confidence Intervals") +
  geom_ribbon(aes(ymin = pred.low, ymax = pred.high), fill = "chartreuse1", linetype = 0) +
  geom_ribbon(aes(ymin = conf.low, ymax = conf.high), fill = "coral1", linetype = 0) +
  geom_point(colour = "darkslategray", size = 0.2))

```



Of course... we could also have just looked at the two formulas and noted that $\sigma^2 > 0$ to see that $V(\hat{y}^0 - y^0) > V(\hat{y}^0)$.

The glory of asymptopia

Finally, the main event. In this section, we'll verify in code what we showed in section 4.2 in lecture. We'll demonstrate that, given linearity, population orthogonality, and our asymptotic full rank condition, no matter what distribution on the disturbances, the estimated \mathbf{b} will be distributed normal as $N \rightarrow \infty$.

We'll be doing another simulation exercise. Let's get started. First, we'll define our true population of interest, the size of sample for each draw, and the number of draws.

```
pop.n <- 10000
n <- 500
draws <- 1000
pop.X <- cbind(1, runif(pop.n,0,20))
```

Next we want to define some errors. To make it interesting, we'll do this whole exercise with four different sets of errors. The sets will be distributed normal, uniform, poisson, and finally a crazy bimodal gamma distribution that I made up. This is going to be awesome.

```
bimodalDistFunc <- function (n, cpct, mu1, mu2, sig1, sig2) {
  d0 <- rgamma(n, 1) + 3
  d1 <- rgamma(n, 1)
  flag <- rbinom(n, size = 1, prob = cpct)
  d <- d1 * (1 - flag) + d0 * flag
}

pop.eps.0 <- rnorm(pop.n)
pop.eps.1 <- runif(pop.n, -5, 5)
pop.eps.2 <- rpois(pop.n, 1) - 1
pop.eps.3 <- bimodalDistFunc(n=pop.n, cpct = 0.7, mu1 = 2, mu2 = 10, sig1 = 1, sig2 = 2) - 3.1
```

It's important to note that I rigged these epsilons so that they all have mean zero:

```
c(mean(pop.eps.0), mean(pop.eps.1), mean(pop.eps.2), mean(pop.eps.3))
```

```
[1] -0.005091301  0.006637526  0.027300000  0.003020903
```

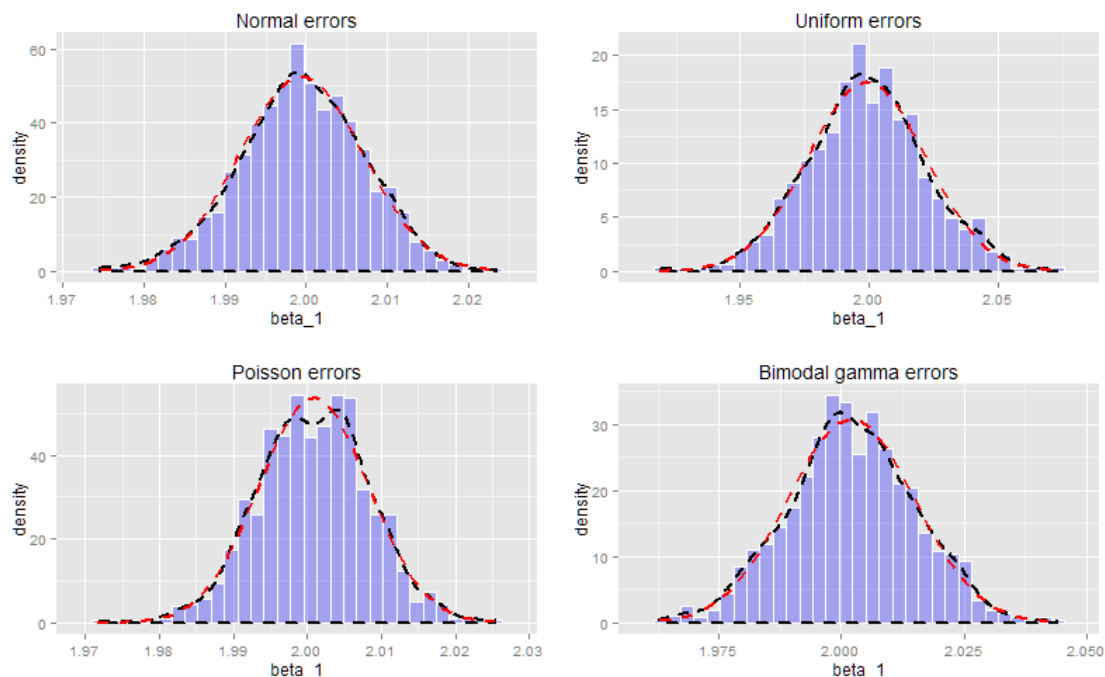
This isn't strictly necessary, but it makes our lives easier down the road. Now we'll create a data frame with our errors that we can easily graph using `ggplot2`. This takes a little doing — we need to stack the errors on top of each other and add a factor variable for the type of error they are. Factor variables are just categorical variables in R.

```
pop.eps <- c(pop.eps.0, pop.eps.1, pop.eps.2, pop.eps.3)
pop.eps.type <- c(rep("Normal",pop.n), rep("Uniform", pop.n), rep("Poisson", pop.n), rep("Bimodal Gamma", pop.n))
class(pop.eps.type)
pop.eps.type <- factor(pop.eps.type, levels = c("Normal", "Uniform", "Poisson", "Bimodal Gamma"))
levels(pop.eps.type)
class(pop.eps.type)
pop.eps.df <- data.frame(pop.eps, pop.eps.type)
```

```
[1] "character"
[1] "Normal"          "Uniform"          "Poisson"          "Bimodal Gamma"
[1] "factor"
```

Now that we've set all of this up, why not see what these crazy errors look like?

```
(g <- ggplot(data = pop.eps.df, aes(x = pop.eps)) + geom_histogram(aes(y=..density..)) + facet_wrap( ~ pop.eps, ncol = 4))
```



Cool. The last three all violate our normality assumption pretty egregiously. Let's see how they do... in **asymptopia**!

Now we'll define our population outcome variables, one for each set of errors:

```
beta <- c(4,2)
pop.y.0 <- pop.X %*% beta + pop.eps.0
pop.y.1 <- pop.X %*% beta + pop.eps.1
pop.y.2 <- pop.X %*% beta + pop.eps.2
pop.y.3 <- pop.X %*% beta + pop.eps.3
```

We also need to build a function that returns OLS estimates for a given set of population variables. `getb` will be that function, and the code it in should be familiar from section 6. As we did then, we'll randomly sample indices (without replacement) and then pull the corresponding data from `pop.x` and `pop.y`.

```
getb <- function(draw, pop.n, pop.y, pop.X) {
  indices <- sample(1:pop.n, n)
  y <- pop.y[indices]
  X <- pop.X[indices, ]
  b <- OLS(y,X)
  return(b)
}
```

The code above won't win any points for efficiency, since we're passing the whole population every time, but it does generalize pretty well. In fact, we'll only modify the population of y that we pass.

```
blist.0 <- t(sapply(1:draws, getb, pop.n = pop.n, pop.y = pop.y.0, pop.X = pop.X))
blist.1 <- t(sapply(1:draws, getb, pop.n = pop.n, pop.y = pop.y.1, pop.X = pop.X))
blist.2 <- t(sapply(1:draws, getb, pop.n = pop.n, pop.y = pop.y.2, pop.X = pop.X))
blist.3 <- t(sapply(1:draws, getb, pop.n = pop.n, pop.y = pop.y.3, pop.X = pop.X))
cbind(head(blist.0), head(blist.1), head(blist.2), head(blist.3))
```

```
      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]      [,8]
[1,] 4.038741 1.993606 3.833315 2.008610 4.069614 1.995341 4.228349 1.968838
[2,] 4.211471 1.979258 3.997741 2.001081 3.838118 2.012667 4.039696 1.998951
[3,] 3.982757 1.990092 3.947543 2.003154 3.991602 2.005622 4.234144 1.992493
[4,] 4.081422 1.988224 3.983327 2.016137 3.975044 2.003647 4.087297 1.975133
[5,] 4.045553 1.994834 4.177116 1.983157 4.090498 1.996028 3.908850 2.006231
[6,] 4.002158 1.999111 3.936177 2.006627 4.067400 1.997383 3.752625 2.027198
```

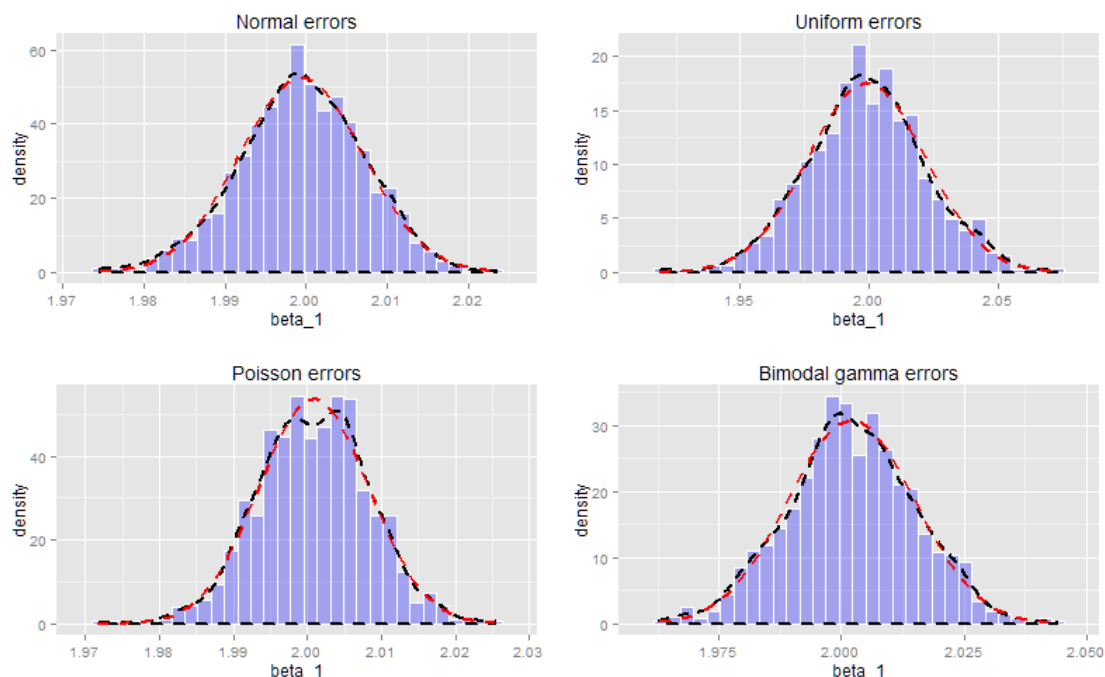
This looks pretty good so far. The odd columns are our point estimates for α , the intercept, and the even columns are our estimates for β_1 , the coefficient estimate. From now on we'll focus on β_1 . We've generated a huge amount of these point estimates, but now we'd like to see if they're truly distributed normally. To do this, we'll create a function that graphs each of them.

```
make.plots <- function(blist) { # makehist returns a graph
  blist.df <- data.frame(blist)
  names(blist.df) <- c("alpha", "beta_1")
  estmean <- mean(blist.df$beta_1)
  estsd <- sd(blist.df$beta_1)
  g <- ggplot(data = blist.df, aes(x = beta_1)) +
    geom_histogram(aes(y=..density..), fill="blue", colour="white", alpha=0.3) +
    geom_density(colour = "black", size = 1, linetype = "dashed") +
    stat_function(geom="line", fun=dnorm, arg=list(mean = estmean, sd = estsd), colour = "red", size = 1)
  return(g)
}

g0 <- make.plots(blist.0) + ggtitle("Normal errors")
g1 <- make.plots(blist.1) + ggtitle("Uniform errors")
g2 <- make.plots(blist.2) + ggtitle("Poisson errors")
g3 <- make.plots(blist.3) + ggtitle("Bimodal gamma errors")
```

Our function `makehist` returns a graph a histogram, smoothed density, and a normal curve with the same mean and standard deviation as the data, so we just run it 4 times to get our 4 graphs (one for each list of our estimated b values). Next, we'll use the `gridExtra` package (which you may need to install) to display the graphs together, just as we did with our errors. Note that we use `gridExtra` and not the `facets` command as we did before just because `facets` wasn't displaying the normal distribution correctly. What we'd like to see is that our smoothed density function and histogram look roughly similar to the normal distribution. Cross your fingers!

```
library(gridExtra)
grid.arrange(g0, g1, g2, g3, ncol = 2)
```



Hey, these actually look pretty good! But this isn't a rigorous test. There are actually statistical methods available to test whether a distribution is normal or not. We'll use one called the Shapiro-Wilk's test of normality. Unfortunately, the computation of Shapiro-Wilk's is a bit complicated, so we'll be leaning on a canned command here to perform it. There are other, simpler, tests (like Kolmogorov-Smirnov), but I like Shapiro-Wilk's because it's meant specifically for normal distributions.

First, we'll verify that the test works by testing it against a normal distribution and a non-normal distribution.

```
(cbind(shapiro.test(rnorm(draws))[1:2], shapiro.test(rpois(draws,10))[1:2]))
```

	[,1]	[,2]
statistic	0.9992255	0.9844206
p.value	0.9619986	7.629168e-09

Great! The null is that the distribution is distributed normal, so we fail to reject (at any reasonable level) the null for the normal distribution and reject the null with a great deal of confidence for the poisson distribution. Now let's try it on our estimated coefficients:

```
(cbind(
  shapiro.test(blist.0[,2])[1:2],
  shapiro.test(blist.1[,2])[1:2],
  shapiro.test(blist.2[,2])[1:2],
  shapiro.test(blist.3[,2])[1:2]))
```

	[,1]	[,2]	[,3]	[,4]
statistic	0.9974901	0.9983627	0.9981639	0.9988518
p.value	0.127683	0.4682959	0.3581798	0.7879152

Great! We fail to reject the null for all three distributions, so we can feel some confidence that our coefficients are truly distributed normal! All is well in asymptopia!