Because we didn't get to it last week, we'll begin by generating dummy variables in R.

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 → 1, FALSE → 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.

Asymptopia is the best -topia

In this part we'll build up a simulation in `R` to show why living in asymptopia is great. With any remaining time we'll look into what happens when your errors have finite mean but infinite variance. First though, some brief notes on the problem set you all turned in Monday.

Abnormal errors, asymptotically

We'll verify in code what was showed in section 4.2 in lecture. We're going to demonstrate that, given linearity, population orthogonality, and our asymptotic full rank condition, no matter what distribution on the disturbances, the estimated $\boldsymbol{b}$ will be distributed normal as $N \to \infty$.

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.

```
set.seed(3172015)
popN <- 100000
n <- 1000
draws <- 5000
popX <- cbind(1, runif(popN, 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's made up. This is going to be awesome.

```
bimodalDistFunc <- function (n, weight) {
  # This part has mean 4
  d0 <- rgamma(n, shape = 1) + 3
  # This part has mean 1
  d1 <- rgamma(n, shape = 1)
  flag <- rbinom(n, size = 1, prob = weight)
  d <- d1 * (1 - flag) + d0 * flag
  # Subtract the mean so this has mean zero
  d <- d  - weight*4 - (1 - weight)*1
  d
}

popEps0 <- rnorm(popN)
popEps1 <- runif(popN, -5, 5)
popEps2 <- rpois(popN, 1) - 1
popEps3 <- bimodalDistFunc(n = popN, weight = 0.7)
```

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

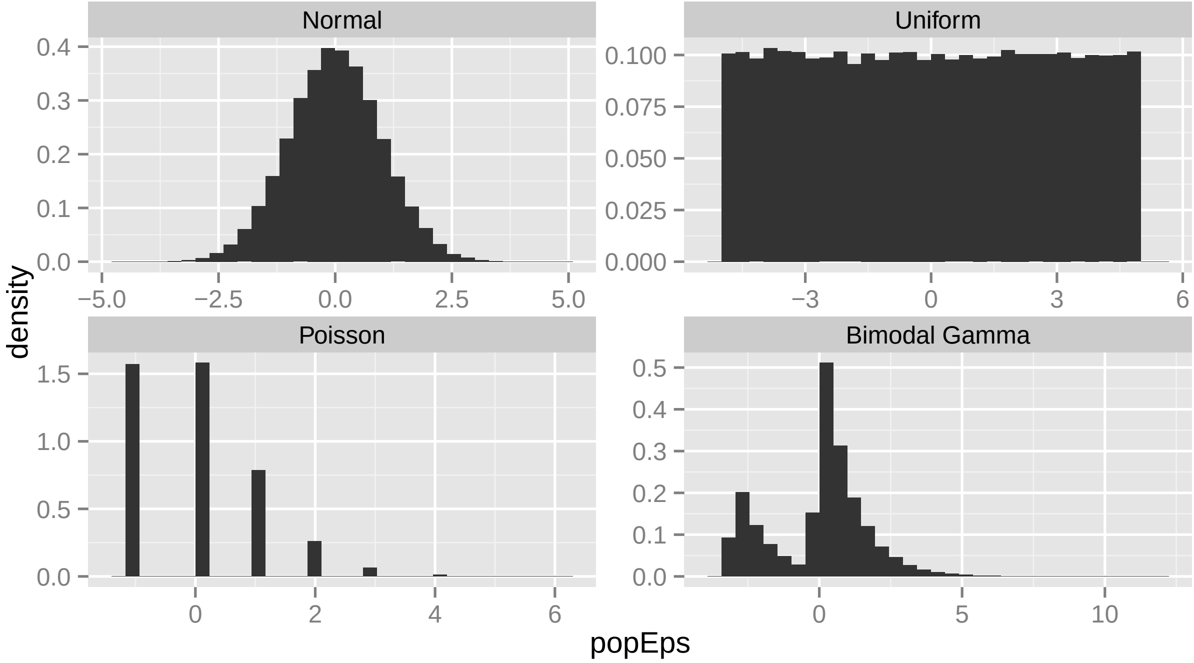
```
c(mean(popEps0), mean(popEps1), mean(popEps2), mean(popEps3))

## [1] -5.639668e-05 -1.663842e-03  1.040000e-03 -2.722085e-03
```

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. Recall that factor variables are just categorical variables in `R`.

```
popEps<- c(popEps0, popEps1, popEps2, popEps3)
popEpsType <- c(rep("Normal",popN), rep("Uniform", popN),
  rep("Poisson", popN), rep("Bimodal Gamma", popN))
class(popEpsType)

## [1] "character"

popEpsType <- factor(popEpsType, levels = c("Normal", "Uniform",
  "Poisson", "Bimodal Gamma"))
levels(popEpsType)

## [1] "Normal"        "Uniform"       "Poisson"       "Bimodal Gamma"

class(popEpsType)

## [1] "factor"

popEpsDf <- data.frame(popEps, popEpsType)
```

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

```
library(ggplot2)
g <- ggplot(data = popEpsDf, aes(x = popEps)) +
  geom_histogram(aes(y=..density..)) +
  facet_wrap( ~ popEpsType, ncol=2, scales = "free")
g
```



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)
# All four stacked again
popYDf <- data.frame(type= popEpsType, y =rep(popX %*% beta, 4) + popEpsDf$popEps)
```

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 `popX` and `popY`.

```
getb <- function(popY) {
  indices <- sample(1:popN, n)
  y <- popY[indices]
  X <- popX[indices, ]
  b <- solve(t(X) %*% X) %*% t(X) %*% y
  return(b)
}
```

Note that the above function depends on both the variable that is passed to it and variables that are defined in the global environment. Functions can access both, and if there are conflicts, the function will choose the one that is passed to the function.

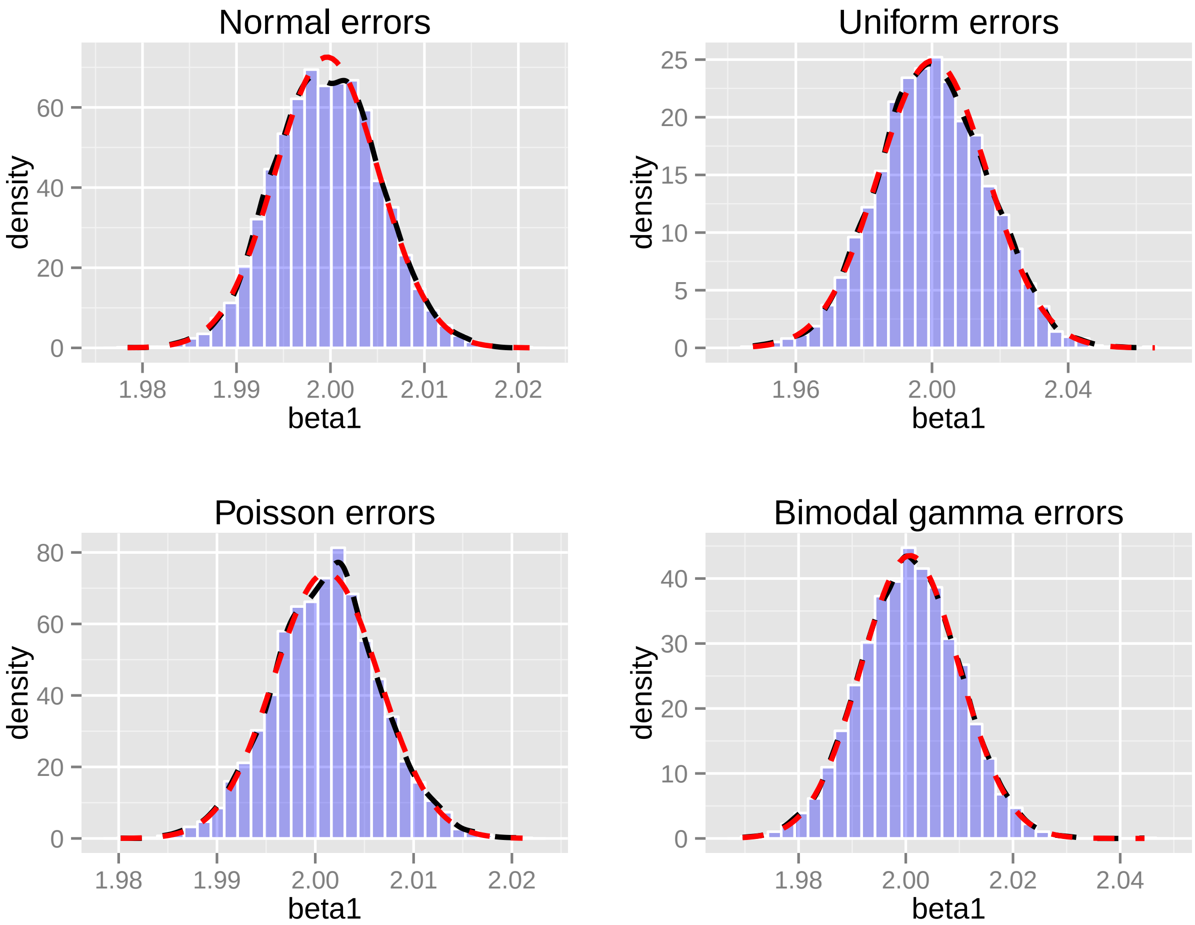
```
library(foreach)
bListDf <- data.frame(foreach(type = unique(popEpsType), .combine = rbind) %do% {
  popY <- popYDf[popYDf$type == type,]
  foreach(draw = rep(NA, draws), .combine = rbind) %do% {
    b <- getb(popY$y)
    data.frame(alpha = b[1], beta1 = b[2], type = type)
  }
})
```

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.

```
makePlots <- function(blist) {
  estmean <- mean(blist$beta1)
  estsd <- sd(blist$beta1)
  g <- ggplot(data = blist, aes(x = beta1)) +
   geom_histogram(aes(y=..density..), fill="blue", colour="white", alpha=0.3) +
    geom_line(colour = "black", size = 1, linetype = "dashed", stat="density") +
   stat_function(geom="line", fun=dnorm, arg=list(mean = estmean, sd = estsd),
     colour = "red", size = 1, linetype = "dashed")
  return(g)
}

g0 <- makePlots(bListDf[bListDf$type == "Normal",]) + ggtitle("Normal errors")
g1 <- makePlots(bListDf[bListDf$type == "Uniform",]) + ggtitle("Uniform errors")
g2 <- makePlots(bListDf[bListDf$type == "Poisson",]) + ggtitle("Poisson errors")
g3 <- makePlots(bListDf[bListDf$type == "Bimodal Gamma",]) + ggtitle("Bimodal gamma errors")
```

Our function `makePlots` 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 could also use `facets` here to grid up the plots, but using `gridExtra` can sometimes be more user friendly.

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

## Warning:  position_stack requires constant width:  output may be incorrect
```



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. The command `shapiro.test()` returns a number of objects, but we'll focus on the p-value.

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

##         [,1]      [,2]
## p.value 0.6331787 1.353845e-22
```

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(bListDf[bListDf$type == "Normal","beta1"])[2],
  shapiro.test(bListDf[bListDf$type == "Uniform","beta1"])[2],
  shapiro.test(bListDf[bListDf$type == "Poisson","beta1"])[2],
  shapiro.test(bListDf[bListDf$type == "Bimodal Gamma","beta1"])[2])

##              [,1]      [,2]      [,3]      [,4]
## p.value 0.3929794 0.2436704 0.6138136 0.7504847
```

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