Lecture 13: Simulation

Now that we've gotten familiar with different kinds of probability distributions and different kinds of model structure, we're going to spend a few lectures thinking about different ways to test hypotheses, compare the fit/performance of models, select the 'best' model, etc. To start we'll learn about simulating fake data from a model, as well as simulation-ish techniques like cross-validation. Over time I've decided that understanding how to simulate statistical models is essential, because it touches on many aspects of data analysis: it helps you understand how statistical models actually work; it helps you understand concepts like confidence intervals, Type I error, and power; it helps you design research and figure out what kind of model makes sense for your analysis; and it is an important tool for doing inference on a model.

So far we've been looking at different kinds of models and thinking about whether they're appropriate for the data at hand. Now we'll reverse this process and say "If a certain model was true, what would the data look like?". To answer this we'll need to put together these components:

1) the *deterministic structure* of the model

2) the values of the *predictor variables*

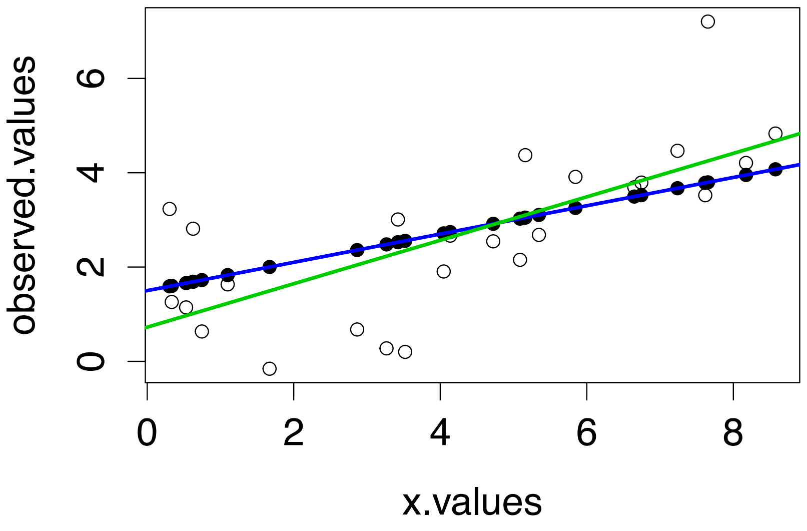
3) use the model and the predictors to calculate the *expected value of the response* variable

4) use the expected value of the response to make some *random draws of fake data* from the appropriate probability distribution

Let's start with a simple linear regression:

```
#make the predictor variable: pick 25 values from a uniform distribution
x.values = runif(25, min = 0, max = 10)

#this will be a linear regression. so pick values for the intercept and the sl
ope.
intercept = 1.5
slope = 0.3

#to get the expected/predicted values for the response variable, use the equat
ion for the line
predicted.values = intercept + slope*x.values

#to simulate observed values, we draw 25 random normal values, using the predi
cted values as the means for each draw. the standard deviation is set to 1.5 f
or all random draws.
observed.values = rnorm(length(predicted.values), predicted.values, sd = 1.5)
```

```r
par(cex = 1.5, mar = c(4,4,1,1))
#plot the simulated data vs the predictor
plot(observed.values ~ x.values)

#plot the expected values on top
points(predicted.values ~ x.values, pch = 19)

#plot the true relationship
abline(a = intercept, b = slope, col = 'blue', lwd = 3)

#fit a linear regression, and plot the fitted relationship
mod = lm(observed.values ~ x.values)
abline(mod, col = 'green3', lwd = 3)
```



Here I've defined a hypothetical predictor called x.values, for which I drew 25 values ranging from 0 to 10, using the uniform distribution. The uniform is good for this purpose, because you get values that are evenly spread out on the x-axis. Then I assume that the response variable will be related to the predictor according to a line with an intercept of 1.5 and a slope of 0.3. I create a vector of expected values for the response variable, by taking the predictor (x.values), and multiplying each value by 0.3 and then adding 1.5. Finally, I draw some random 'observed' response values, using the expeced values and rnorm(). I specify how many values to draw (the length of the predicted.values vector), the mean of the normal distribution for each draw (the vector predicted.values), and the standard deviation to use for all draws (sd = 1.5). The plot shows what the fake data look like (open circles), what the expected values were (closed circles), and what the true relationship is (blue line). Finally I fit a linear regression and plotted the estimated line in green. The estimate line is similar but not identical to the true line, because the randomness in the draws from the normal distribution leads to error in the estimated line.

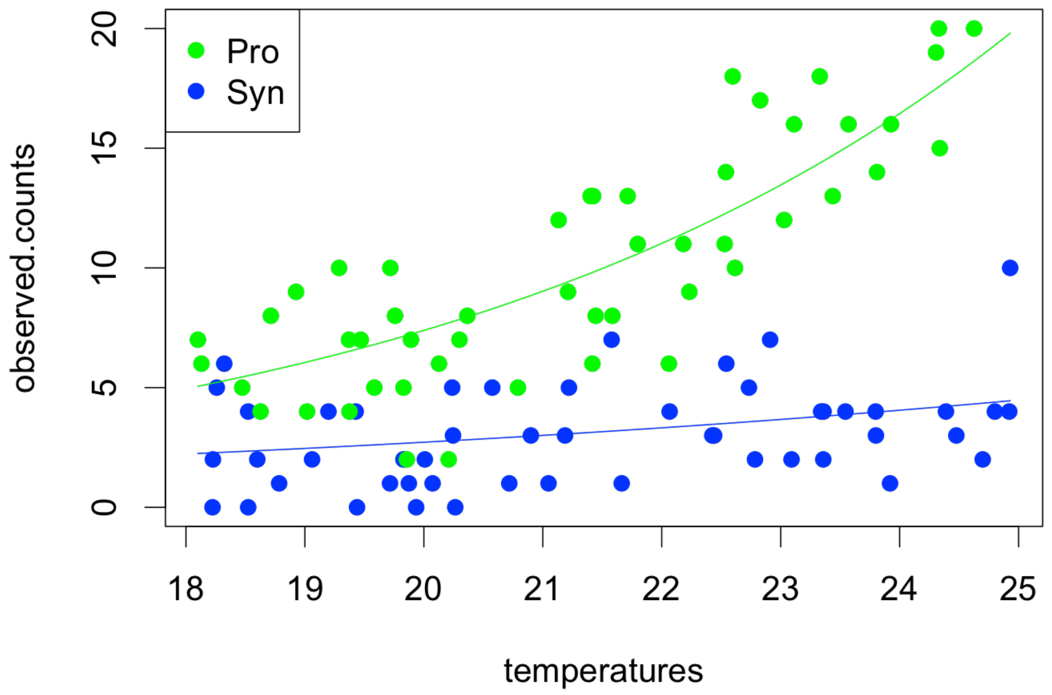
This example shows how you can easily create your own example stochastic models in R. We'll put this to use for a couple different purposes, but first let's do a slightly more complex example.

A simulation with two groups and a Poisson response

Here's another example that incorporates the various kinds of complexity we've discussed so far (nonlinear functions, nonnormal distributions, multiple groups). Let's imagine that we are sampling counts of two phytoplankton, *Prochlorococcus* and *Synechococcus*, at locations that vary in temperature. I'm going to make a model that assumes that *Prochlorococcus* increases with temperature more steeply than *Synechococcus*.

```
#simulate a poisson regression with two groups

#create a bunch of temperature from 18 to 25
temperatures = runif(100, min = 18, max = 25)

#create a species factor; 50 Synechoccus and 50 Prochlorococcus
species = factor(rep(c("Syn", "Pro"), each = 50))

#define a slope and intercept for each species
intercept.pro = -2
intercept.syn = -1
slope.pro = 0.2
slope.syn = 0.1

#how to index values using the grouping factor.
species

##   [1] Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn
##  [18] Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn
##  [35] Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Syn Pro
##  [52] Pro Pro Pro Pro Pro Pro Pro Pro Pro Pro Pro Pro Pro Pro

…

c(intercept.pro, intercept.syn)

## [1] -2 -1

c(intercept.pro, intercept.syn)[species]

##   [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
##  [24] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
##  [47] -1 -1 -1 -1 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
##  [70] -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2 -2
##  [93] -2 -2 -2 -2 -2 -2 -2 -2

#make a vector of intercepts, and a vector of slopes
intercepts = c(intercept.pro, intercept.syn)[species]
slopes = c(slope.pro, slope.syn)[species]

#calculate predicted counts using a linear model inside an exponential functio
n
```

```
predicted.counts = exp(intercepts + slopes*temperatures)

#draw fake data from a poisson distribution
observed.counts = rpois(length(predicted.counts), predicted.counts)

#define colors and plot
colors.use = c('green', 'blue')
par(cex = 1.7)
plot(observed.counts ~ temperatures, col = colors.use[species], pch = 19)
legend('topleft', pch = 19, col = colors.use, legend = levels(species))

#plot the 'true' lines
curve(exp(intercept.pro + slope.pro*x), col = 'green', add = TRUE)
curve(exp(intercept.syn + slope.syn*x), col = 'blue', add = TRUE)
```



OK let's unpack all the steps. First I create 100 fake temperatures, drawn randomly from 18 to 25. Then I create a factor to say that the first 50 values are samples from Synechoccous and the next 50 are from Prochlorococcus. The function rep() says to repeat the vector I give it, the character vector c("Syn", "Pro"), by repeating each entry 50 times in a row. The function factor() then turns this character vector into a factor.

Next I define some coefficients for a relationship between abundance and temperature for each species. I'm going to this function:

$$\exp(a + bx)$$

Which you will recognize as the default exponential function for a Poisson GLM, with 'intercept' *a* and 'slope' *b*. I am going to give Syn and Pro their own slopes and intercepts. The calculation of predicted.values is a little complex. For each of the 100 temperatures, I want to use the intercept and slope from the appropriate species. To select the right intercept, I first make a vector of the two intercepts, `c(intercept.pro, intercept.syn)`, then I index this vector with the factor "species". How can I index a vector with a factor? As I explained earlier, factors in R are stored as a vector of integers, with a set of *levels* that store the names associated with each integer. So when I say

```
c(intercept.pro, intercept.syn)[species]
```

The indexing vector 'species' will pull out the first entry, 'intercept.pro', when the species vector equal to its first level ('Pro'). And likewise the indexing vector will pull out the second entry, 'intercept.syn', when the species vector is equal to its second level ('Syn').

After I make the species-specific intercepts and slopes, I put them into the exponential curve to get the expected values. Finally, I use rpois() to pull random counts from the Poisson distribution, with each random draw having a mean (lambda) equal to the expected value from the model. Then I plot the simulated data and the corresponding true curves.

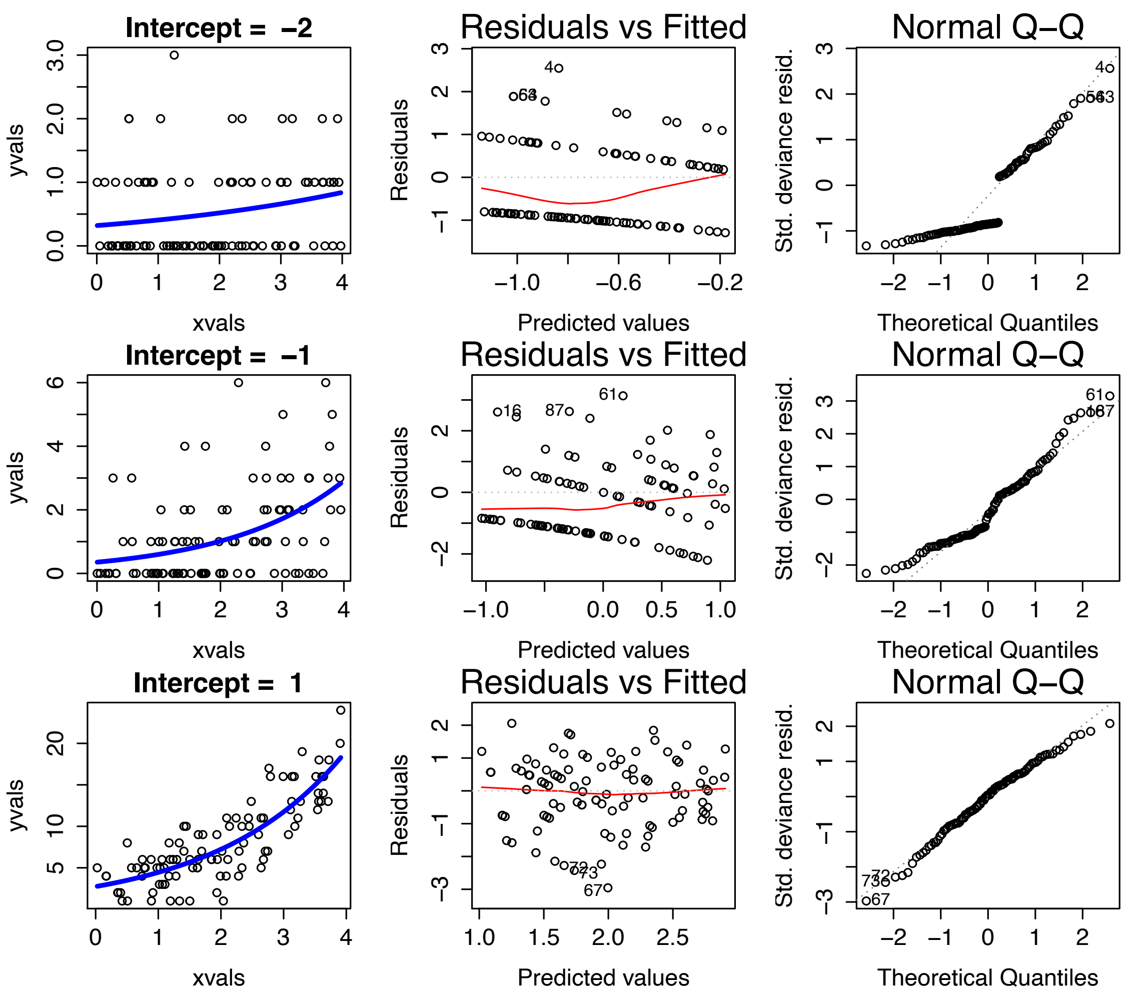
Simulation to figure out what the data is supposed to look like

One of the uses of simulation, which I've already been doing extensively in this course, is to figure out what the data should look like if the model is true. This is important because for complex/non-normal/nonlinear models it can be difficult to know whether some pattern in the data is strange or plausible. Let's use this to explore the issue of residuals diagnostics for GLMs, which we've discussed already but which has been causing some confusion on the homework assignments.

When we fit a Poisson or binomial GLM, we know the raw ('response') residuals are likely to have some pattern because of the mean-variance scaling. The discreteness of the data tends to case patterns as well. We can use the deviance or pearson residuals, which both account for the mean-variance scaling, but what patterns do we expect to see? Are these 'corrected' residuals going to be normally distributed? Will they tend to show other patterns? This is the kind of thing that simulation can help with.

Here is a simulated Poisson example, where there is an exponential relationship between predictor and response, and the response is Poisson distributed. I compare 3 different values of the intercept, because this makes it so that the counts are mostly small (0, 1, 2) or mostly larger (> 0). As we will see, this has a big effect on the residual patterns.

```r
#compare three different poisson relationships, that differ in the intercept,
which determines how close to zero the response variables tend to be
intercepts = c(-2, -1, 1)

par(mfrow = c(3,3), mar = c(4,4,2,2))

for (i in 1:3) {
#define an intercept and slope for an exponential relationship with Poisson re
sponse
intercept = intercepts[i]
slope = 0.5
#simulate 100 values that follow the relationship
xvals = runif(100, 0, 4)
yvals = rpois(100, lambda = exp(intercept+slope*xvals))

#plot the data
plot(yvals ~ xvals, main = paste("Intercept = ", intercept))

#fit a poisson glm
mod = glm(yvals ~ xvals, family = poisson)
#plot the fitted curve
curve(exp(coef(mod)[1]+coef(mod)[2]*x), col = 'blue', add = T, lwd = 3)
#plot residuals vs fitted values, and a normal quantile-quantile plot
plot(mod, which = c(1:2))
}
```

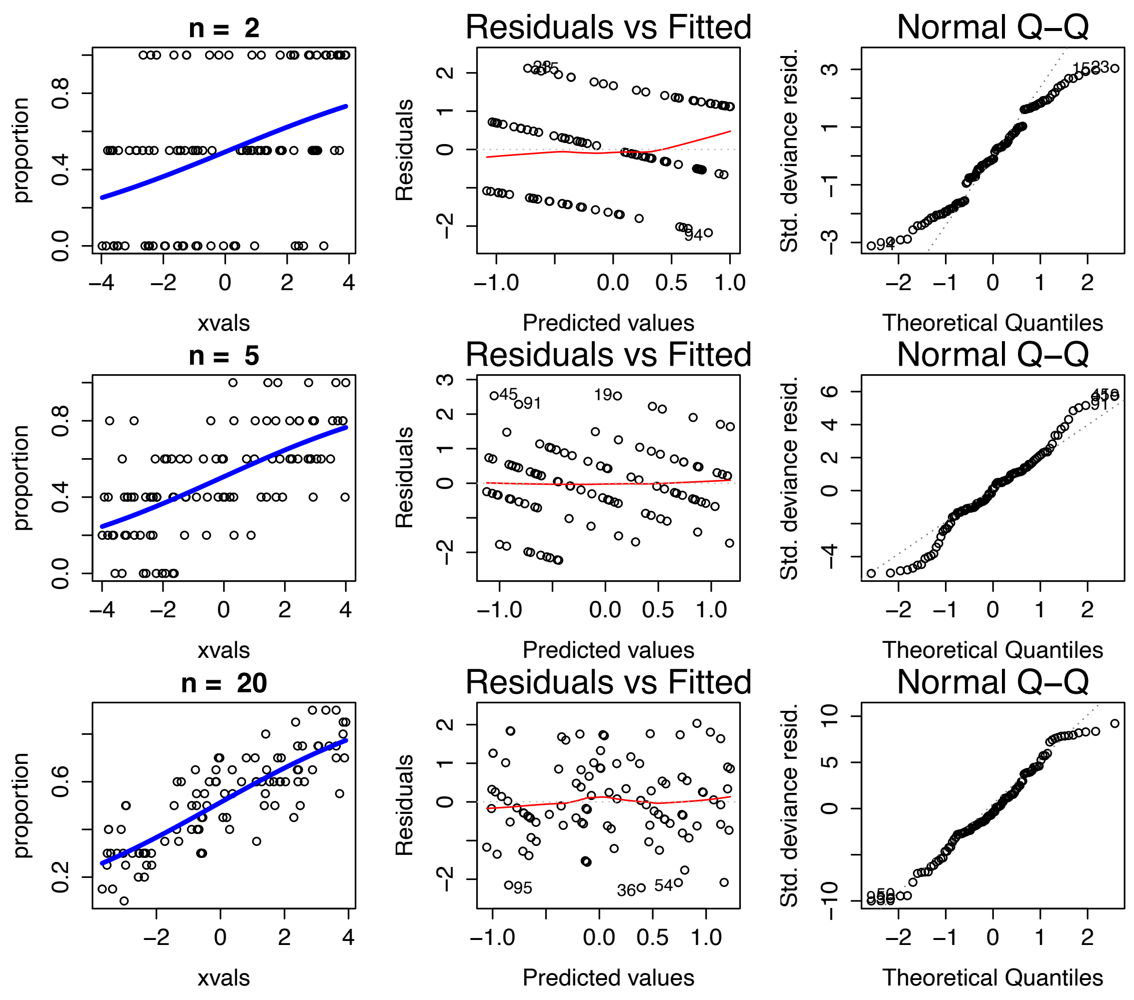


Each row shows the simulated data + fitted curve, the deviance residuals vs fitted values, and a normal quantile-quantile plot for the standardized deviance residuals. The rows differ in the value of the intercept in the exponential relationship (-2, -1, and 1, respectively). For the top row, the response variable has a lot of zeros, and some ones and twos. Because the response only takes on these three values the residuals have a distinct pattern. Nonetheless, if we look at residuals vs fitted there isn't a strong pattern in the mean value of the residuals (the red smoother line). The normal quantile-quantile plot definitely shows an aberrant pattern at the lower values. What can we conclude from this? We know this data isn't breaking any assumptions; we simulated the data from the exact model structure that we used for the GLM. So this give us a sense for the limited utility of residual diagnostics for Poisson GLMs when the counts are mostly small. The residuals vs fitted should still be roughly trendless, but the residuals will not be normally distributed.

When we compare this case to the other intercept values, the story changes. When the intercept is 1(bottom row), there are few zeros in the response, and the plots of residuals vs fitted and quantile-quantile look essentially like a gaussian linear model. The case with intercept = -1 is intermediate; the residual patterns are not as strong as the top row, but the counts are still small enough that the residuals look a bit weird. So what I take from this simulation is that for count data where the counts are mostly large numbers, the deviance residuals should look similar to a normal linear model, and if they don't that may indicate a problem (and pearson residuals would show similar patterns to what I've shown here). But when the counts are mostly small, some strong patterning is inevitable due to the discreteness of the data. Of course one should still check for overdispersion, which is the most common issue with count data; I haven't done that here because we know *a priori* that the data are not overdispersed.

We can do a similar exercise for the binomial distribution. In this case the patterning in the residuals is strongly effected by *n*, the number of 'trials' for the binomial variable. We've already seen that binary data (*n* = 1) produces strongly patterned residuals. How large does *n* have to be to look more normal?

```r
#same kind of loop as above, but now using binomial data with logistic curve.
varying N, the number of trials in the binomial response.
binomial.N = c(2,5,20)

par(mfrow = c(3,3), mar = c(4,4,2,2))

for (i in 1:3) {
N = binomial.N[i]
slope = 0.3
xvals = runif(100, -4, 4)
yvals = rbinom(100, size = N, prob = exp(slope*xvals)/(1+exp(slope*xvals)))
proportion = yvals/N

plot(proportion ~ xvals, main = paste("n = ", N))

mod = glm(proportion ~ xvals, weights = rep(N, 100), family = binomial)
curve(exp(coef(mod)[1]+coef(mod)[2]*x)/(1+exp(coef(mod)[1]+coef(mod)[2]*x)),
col = 'blue', add = T, lwd = 3)
```



I've compared n = 2, 5, and 20. The discreteness of the data is clearly evident for n = 2 and n = 5, though like the Poisson case the mean of the residuals does not show a strong trend vs fitted values. The normal quantile-quantile plot only starts to look good for n = 20. So the punchline here is similar to that for the Poisson; the mean of the residuals should not show a strong pattern, but the residuals will only be approximately normal when #successes can take on a wide range of values. And overdispersion needs to be looked at as well.

Frequentist Inference

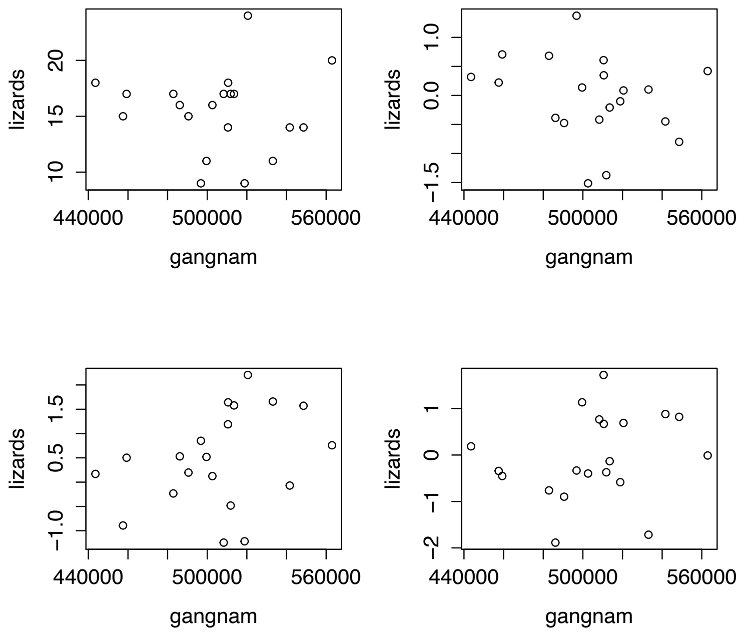
Simulation is very useful for understanding and quantifying the classic inferential methods of confidence intervals, p-values, and null hypothesis tests. These methods are typically grouped together under a common philosophy called *frequentist* inference (though once you start digging into statistical philosophy, internal disagreements are evident). The frequentist approach is all about asking "how many times would X happen, over many repetitions of the sampling process?". For example, P-values are quantifying how many times you would get a result as extreme as the observed result, if the null model were actually true, and if you repeated the sampling process an infinite number of times. 95% confidence intervals are quantifying an interval that will contain the true value of the parameter 95% of the time, if the sampling process were repeated an infinite number of times.

Simulating Type I error

One of the fundamental concepts in frequentist inference is Type I error. The Type I error rate is the probability that you will find a 'significant' relationship even when none exists, i.e. it is the probability that you reject the null hypothesis even though it's true. Lets use a simulation to think about how Type I error works. I'm going to simulate data for a linear regression where there is *no relationship* between the 'predictor' and the 'response'. Lets imagine that the response is 'how many lizards I saw while walking to campus this morning'. And let's imagine that the predictor is 'how many people watched the video for Gangnam Style today'. I'm pretty sure these will have no relationship, but I don't have actual data, rather I'm going to simulate values that have no underlying relationship. I'm going to iterate this process many times, and each time I will run a linear regression to see what the estimated slope is for lizards vs Gangnam, and what the p-value for the slope is. The point of this exercise is that even though the two variables have no relationship, because the response variable is sampled randomly, by chance you will occasionally see a low p-value.

Before I code the whole loop, let's just look at four examples of simulated data

```
#simulate what the distribution of p-values looks like when a null hypothesis
is true
#this code generates a 'response' variable, how many lizards seen, that has no
 relation to the 'predictor', the number of Gangnam Style views today.

#before the loop, look at a few examples of what the simulated data will look
like
#define the sample size
sample.size = 20

#generate some predictor values
gangnam = rnorm(sample.size, mean = 500000, sd = 50000)

#generate some 'response' values
lizards = rpois(sample.size, lambda = 15)

#plot 4 examples of randomly drawn response variable
par(mfrow = c(2,2))
plot(lizards ~ gangnam)
lizards = rnorm(sample.size, mean = 0, sd = 1)
plot(lizards ~ gangnam)
lizards = rnorm(sample.size, mean = 0, sd = 1)
plot(lizards ~ gangnam)
lizards = rnorm(sample.size, mean = 0, sd = 1)
plot(lizards ~ gangnam)
```

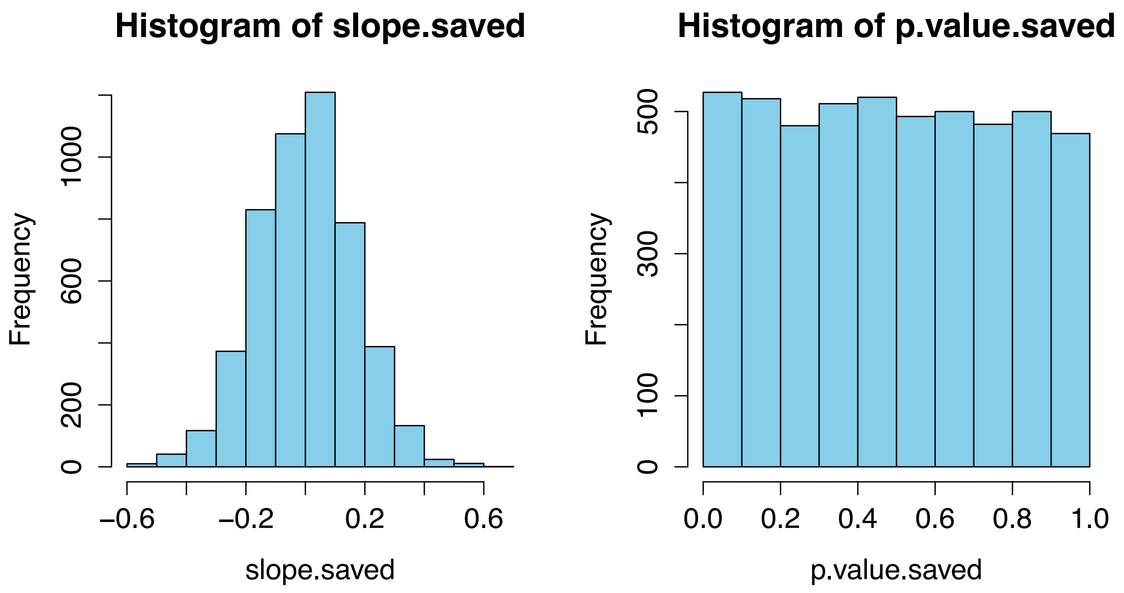


These four plots are four different instances of randomly drawing 20 values from a poisson distribution with mean=15, and plotting those values against a predictor that was created completely separately. None of these plots looks like a significant relationship, but you can see how slightly different patterns are generated, and occasionally a signficant pattern will be generated by chance.

Now for the whole loop, where this process is iterated 5000 times.

```
#now put the simulation into a loop, to see how often the null relationship ge
nerates a significant p-value

#define the number of iterations of the simulation
nsims = 5000
#make a vector to save the estimated slope, and its p-value
slope.saved = p.value.saved = vector()

for (i in 1:nsims) {
  #generate a predictor and a response with no relationship
  gangnam = rnorm(sample.size, mean = 500000, sd = 50000)
  lizards = rpois(sample.size, lambda = 15)

  #fit a linear regression model
  model = glm(lizards ~ gangnam, family = poisson)
  #save the slope estimate
  slope.saved[i] = coef(model)[2]
  #save the p value for the slope
  p.value.saved[i] = summary(model)$coefficients[2,4]
}

par(mfrow = c(1,2))
```

```r
#plot the 1000 simulated slope estimates
hist(slope.saved, col = 'skyblue')
#plot the 1000 simulated p-values
hist(p.value.saved, col = 'skyblue')
```



In this loop, in every iteration I generated new fake values, and then did fit a GLM with poisson error. I saved the slope estimate, and also the p-value (this was extracted from the info returned by summary()). The plots show what the distribution of slope estimates and p-values looks like, over the 5000 iterations. The slope estimates are centered around 0, which is good because in reality the slope is 0, because there is no relationship. With a larger sample size the spread around 0 would be smaller, and vice versa. The plot of the distribution of p-values is essentially flat. This is good! *Under the null hypothesis, p-values should be uniformly distributed.* If the p-values are normally distributed, this means that roughly 5% of the p-values are less than 0.05, and roughly 10% of the p-values are less than 0.10, and so on. And we can calculate this directly as well:

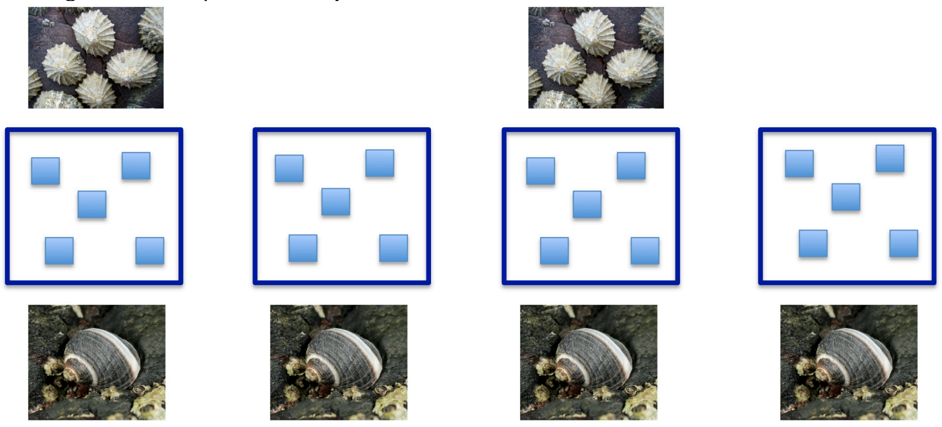
```r
#what proportion of times is the p-value less than 0.05?
sum(p.value.saved < 0.05)/nsims

## [1] 0.0514
```

In this case 5.1% of the p-values are less than 0.05. Let's think about what that means for doing significance tests. We know *a priori* that there is no relationship between these two variables, because that's how we coded the simulation. But because the response variable is measured with a random sample, there is a small chance that an apparent pattern will be generated. This is how frequentist inference works, it assumes that your data is a random sample from a larger population. For example, I count a certain number of lizards while I'm walking to work, but this is not the whole lizard population, it's a random sample. And this is how the formulas for p-values and confidence intervals are derived. If your slope estimate θ for lizards vs. Gangnam has a p-value of 0.05, that means that even when there is no real relationship, 5% of the time you will get a slope estimate that is at least as large as θ. When we did the simulation and got a uniform distribution of p-values, that confirmed that the formula for calculating p-values is accurate. Because 5% of the time we got a slope estimate that was calculate as having $p < 0.05$.

Pseudoreplication and Type I error

Now we have a sense for how simulation is closely related to the whole frequentist approach to statistics. Let's use simulation to see what happens when we break model assumptions. We spend a lot of time thinking about model assumptions, but what exactly are the consequences of breaking them? One of the big ones is independence of the data. For normally distributed data, this is often referred to as 'independently distributed residuals'. For GLMs the idea of residuals is trickier, but but regardless of the probability distribution used in a model, it is assumed that each data point is independently drawn from that distribution, once the effect of the predictors are accounted for. One of the classic examples of breaking this assumption is called *pseudoreplication*. About 30 years ago ecologists were starting to use field experiments in a lot of different kinds of ecosystems, to test things like whether competition between species was important. However, there was a bit of sloppiness in how many experiments were designed/analyzed, which was clarified and critiqued in a paper by Hurlbert, "Pseudoreplication and the design of ecological experiments".

Let's simulate an example of pseudoreplication. Imagine that we perform an experiment in the rocky intertidal, where we remove a species of limpet from 2 plots, and we also have 2 control plots with no removal. We want to see if the abundance of a snail changes when limpets are removed, because they might compete for food or space. Imagine that the experimental plots are large (maybe 3 m by 3 m), and to measure the abundance of snails we take 5 subsamples per plot (using 30 cm by 30 cm quadrats).

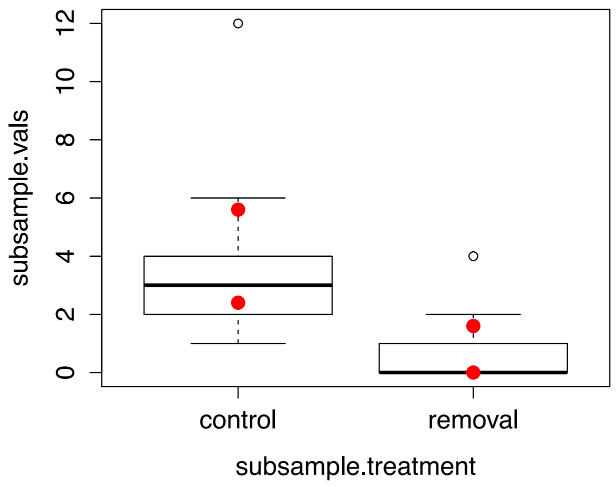


Here's the wrong way to analyze this data: treat each subsample (the little squares) as an independent sample, and see whether the count of snails differs between treatments, with n=10 subsamples per treatment. This is pseudoreplication because we really only have 4 experimental units (2 replicates per treatment), and the subsamples are unlikely to be independent samples. Rather, subsamples from the same plot are going to be similar to each other, because of whatever plot-level conditions that affect snails. This will likely be the case even if there is no treatment effect.

Here's some code that simulates this design, and the wrong/right way to analyze it.

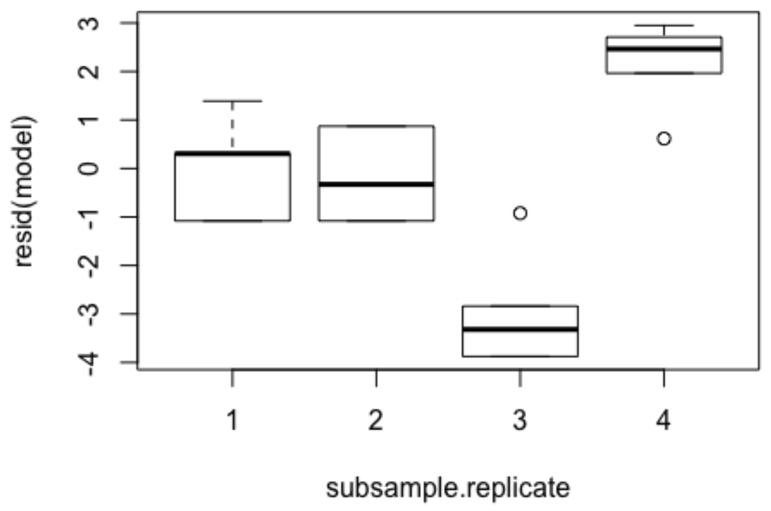
```r
#breaking assumptions - treating subsamples as independent samples, i.e. pseud
oreplication. can have like an experiment with two replicates but vary the num
ber of subsamples
#define number of subsamples per plot
num.subsamples = 5

#treatment of each plot
treatment = as.factor(c('control', 'control', 'removal', 'removal'))

#label each replicate plot from 1 to 4
replicate = as.factor(1:4)

#the 4 plots differ randomly in overall snail density - make that density a ra
ndom draw from a lognormal distribution
replicate.densities = rlnorm(4, meanlog = 1, sdlog = 1)

#define the treatment factor for each subsample
subsample.treatment = rep(treatment, each = num.subsamples)
#define the plot label for each subsample
subsample.replicate = rep(replicate, each = num.subsamples)

#draw random count values from a Poisson for each subsample. the mean of the P
oisson is the plot-level density defined above
subsample.vals = rpois(length(subsample.treatment), lambda = replicate.densiti
es[subsample.replicate])

#calculate the mean count for each plot
replicate.means = tapply(subsample.vals, subsample.replicate, mean)

#plot the subsample counts by treatment
plot(subsample.vals ~ subsample.treatment)
#plot the mean count for each replicate on top
points(treatment, replicate.means, col = 'red', pch = 19, cex = 1.5)
```

I assume that the four plots (i.e. the replicates) each have a different overall density of snails, due to whatever factors (tidal height, topography, predators, etc.). These plot densities are drawn randomly from a lognormal distribution (to make sure they're positive numbers). Then I code factors for the treatment (control or removal) for each plot, as well as treatment and replicate factors for each subsample. Then I simulate a snail count for each subsample, assuming it comes from a Poisson distribution with a mean equal to the plot-level density. Finally, I make boxplots of the subsamples by treatment, and also the plot-average counts on top of that.



The important thing to note here is that I've defined no treatment effect, i.e. I'm assuming t
he limpet removal has no effect on the snails. The four plots (red points) just differ randoml
y from each other. But because each plot has 5 subsamples, if we treat the subsamples as i
ndependent samples it looks like there might be a spurious treatment effect. The non-indep
endence of the samples is clear if we fit a GLM and look at the residuals by plot:

```
#fit a glm to the subsamples (pseudoreplication)
model = glm(subsample.vals ~ subsample.treatment, family = poisson)

#the residuals are clearly not independently distributed. they differ between
the plots.
plot(resid(model) ~ subsample.replicate)
```

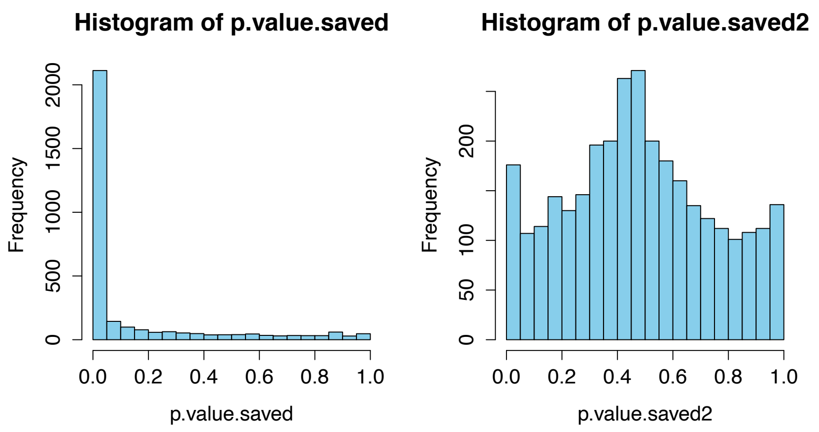


The residuals for each plot should be centered at zero, but they clearly aren't, which means that the residuals within a plot are not independent from each other.

Let's quantify the effect of pseudoreplication by iterating this simulation many times, and each time fitting the pseudoreplicated model. We'll also compare to a model where the response variable is the plot-level count averaged over the subsamples.

```
#now iterate the above simulation 5000 times
nsims = 5000

num.subsamples = 5
p.value.saved = p.value.saved2 = vector()
library(car)

for (i in 1:nsims) {
treatment = as.factor(c('control', 'control', 'removal', 'removal'))
replicate = as.factor(1:4)
replicate.densities = rlnorm(4, meanlog = 1, sdlog = 1)
subsample.treatment = rep(treatment, each = num.subsamples)
subsample.replicate = rep(replicate, each = num.subsamples)
subsample.vals = rpois(length(subsample.treatment), lambda = replicate.densiti
es[subsample.replicate])
replicate.means = tapply(subsample.vals, subsample.replicate, mean)

  #fit the pseudoreplicated model, and store the p-value for the likelihood ra
tio test
  model = glm(subsample.vals ~ subsample.treatment, family = poisson)
  p.value.saved[i] = Anova(model)$P

  #fit a non-pseudoreplicated model to the plot means, and store the p-value f
or the F test
```

```
  model2 = lm(replicate.means ~ treatment)
  p.value.saved2[i] = Anova(model2)$P[1]
}

par(mfrow = c(1,2))
hist(p.value.saved, col = 'skyblue', breaks = 20)
hist(p.value.saved2, col = 'skyblue', breaks = 20)
```



The plot on the left shows the distribution of p-values for the pseudo-replicated model. This is really problematic! It's nowhere close to uniform, and there are many very small p-values.

```
sum(p.value.saved < 0.05)/nsims

## [1] 0.6538
```

In 65% of the simulated datasets, the effect of treatment was significant at the p=0.05 level. So under the simulated conditions, you would get a spurious treatment effect 65% of the time, even if there is not treatment effect. This kind of psuedoreplication is one of the most extreme examples of the consequences of non-independence in a statistical analysis, but it is a mistake that is still made in modern times.

The plot on the right shows the distribution of p-values for the model that is not pseudo-replicated. Instead, I took the plot-level average count for the 4 plots, and tested with lm() whether the plot average is predicted by treatment. In this case the distribution of p-values is closer to uniform, and the Type I error rate is about right:

```
sum(p.value.saved2 < 0.05)/nsims

## [1] 0.0458
```

This distribution still looks slightly funky, which may be due to the small number of replicates (2 per treatment) or may be due to the fact that I used a normal distribution on a variable that is always positive (maybe a lognormal would be slightly better).

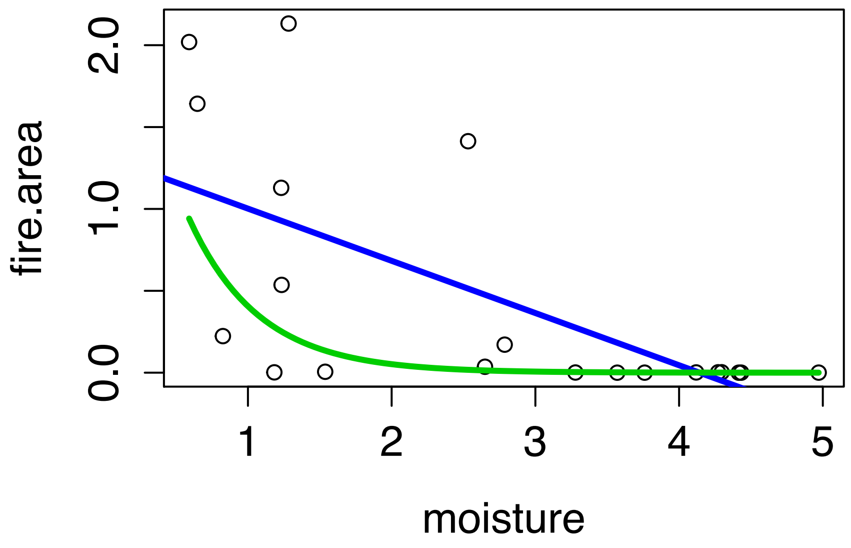
Power

We've covered getting the wrong answer (Type I error), now let's think about getting the right answer. In the frequentist framework, p-values are used to reject null hypotheses. In the example above, the null hypothesis was 'no treatment effect', and the p-value told us how likely the (simulated) data were under the null hypothesis. We don't actually get a p-value for the so-called 'alternative' hypothesis, i.e. that there is real a treatment effect. Rather, we say 'the null is unlikely, so I guess the alternative is supported'. The ability of a dataset/model to reject the null, when the alternative is actually true, is called the statistical *power*. Power will in general depend on a number of conditions: the sample size, which affects how confident we are in the parameter estimates; the effect size, because a larger effect is less likely under the null hypothesis; and the ability of the model to accurately represent the data. Let's look at the third one, because it it probably the trickiest. Our psuedoreplication example showed how a mis-specified model can result in high type I error. It is also the case that a mis-specified model can result in low power.

Imagine we're studying wildfires, and we want to test whether the extent of wildfires (area burned) is related to the soil moisture content before the fire started. Let's simulate data where the area burned is lognormally distributed, and declines exponentially with moisture content. Then let's compare how a standard linear regression compares to a lognormal model:

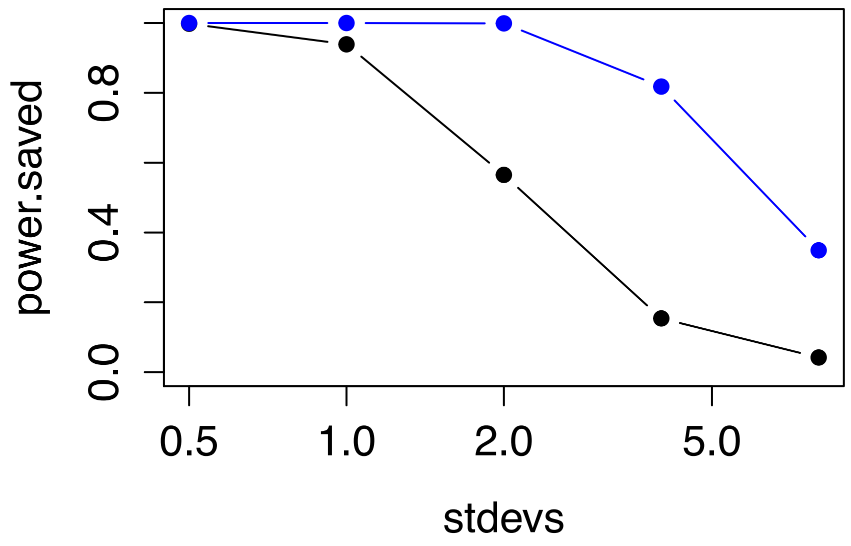
```
#simulation showing how power is reduced by poor model assumptions: lognormal
data vs normal linear model. pretty strong at high variance.
sample.size = 20
nsims = 1000
intercept = 1
slope = -2
sdlog = 2

#make the moisture predictor
moisture = runif(sample.size, 0, 5)
#make the expected values for the response (fire area), exponential relationsh
ip
predicted.values = exp(intercept + slope*moisture)

#simulate fire area from a lognormal distribution. note that for the lognorma
l, the mean and sd of the distribution are defined on the log scale
fire.area = rlnorm(sample.size, meanlog = log(predicted.values), sdlog = sdlo
g)

#fit a linear model
model.linear = lm(fire.area ~ moisture)

#fit a lognormal model
model.lognormal = lm(log(fire.area) ~ moisture)
```

```
#plot the data and fits
plot(fire.area ~ moisture)
abline(model.linear, col = 'blue', lwd = 3)
curve(exp(coef(model.lognormal)[1]+coef(model.lognormal)[2]*x), col = 'green3
', lwd = 3, add = T)
```



We know *a priori* that the lognormal model is better, and in this example it does look better, though the linear model is not the worst thing ever. Let's simulate a bunch of models to see how often the lognormal vs linear model detects a relationship between moisture and fire area. This will be our estimate of power: how often we get a 'significant' result, given that we know *a priori* that a relationship does exist. This will be slightly more complex than the prior examples, because I will also look at 5 different values for the residual noise for the lognormal distribution. So the loop will be nested.

```
power.saved = power.saved.log = vector()
#gonna calculate power for 5 different residual st devs
stdevs = c(0.5, 1, 2, 4, 8)

#This is a nested loop. The outer loops cycles through the 5 different
values of the standard deviation for the residual noise. The inner loop
 simulates 1000 datasets under those conditions, and fits the models, s
ave the p-values.
for (k in 1:length(stdevs)) {
  sdlog = stdevs[k]
  p.value.saved = p.value.saved.log = vector()
  for (i in 1:nsims) {
    moisture = runif(sample.size, 0, 5)
    predicted.values = exp(intercept + slope*moisture)
    fire.area = rlnorm(sample.size, meanlog = log(predicted.values), sd
log = sdlog)
    model.linear = lm(fire.area ~ moisture)
    model.lognormal = lm(log(fire.area) ~ moisture)
    p.value.saved[i] = summary(model.linear)$coefficients[2,4]
    p.value.saved.log[i] = summary(model.lognormal)$coefficients[2,4]
  }
  #calculate the power: proportion of p-values that are less than 0.05
  power.saved[k] = sum(p.value.saved < 0.05, na.rm = T)/sum(!is.na(p.va
lue.saved))
  power.saved.log[k] = sum(p.value.saved.log < 0.05, na.rm = T)/sum(!i
s.na(p.value.saved.log))
  }

plot(power.saved ~ stdevs, type = 'b', pch = 19, ylim = c(0,1), log = '
x')
points(power.saved.log ~ stdevs, type = 'b', pch = 19, ylim = c(0,1), c
ol = 'blue')
```



I wanted to look at different values for the residual noise, because low residual noise will make it easier to detect a signficant relationship, and vice versa. So to compare the two models (linear vs lognormal), I wanted to see their relative performance across different amounts of noise. For each value of residual noise, I made 1000 fake datasets, fit a linear and lognormal model to each, and then saved the p-value for the effect of moisture. Then I calculated how many times the p-value was less than 0.05, for each value of the residual noise. The plot above shows the results for the lognormal model (blue) and the linear model (black). When the residual noise is very low, both models always find a significant relationship. As the noise increases the power declines, but it declines faster for the linear model. When the stdev is 4, the lognormal model has over 4x better power than the linear model. This is sensible because this model is a better fit to both the deterministic relationship (exponential) as well as the stochastic part of the model (lognormal variation), and so it should be more sensitive at detecting relationships.