Generalized Linear Modeling Part I: Log Normal GLMs

ANTH 3720-001 Archaeological and Forensic Science Lab Methods: Data Analysis with R

Elic Weitzel

Jan. 29-31, 2021

If you would like the original R Markdown file, find it on my GitHub page at https://github.com/weitzele/Basic-R-Tutorial.

1 Linear Modeling with Different Types of Data

At this point, you're probably getting sick of modeling spatial data and wondering why so many examples from the last few tutorials have to do with spatial coordinates. This is because (most) spatial coordinate systems are one of the few types of data that archaeologists and forensic anthropologists work with that could reasonably be treated as normally distributed! Latitude and longitude coordinates have 0 values and can range on either site of that 0 point, being negative (south or west) or positive (north or east). They're also able to take decimals, and are thus continuous. Calculated indices comparing two different values are also sometimes normally distributed, depending on how exactly they're calculated. The Fish-Mammal Index we modeled in the previous tutorial is an example of this, as the values were continuous and ranged above and below zero. Stable isotope delta values are also often normally distributed since they're calculations that measure some value relative to a standard.

But aside from these, very little data in archaeology and forensic anthropology is normally distributed.

Think of length or width measurements, proportions or percentages, and artifact or burial counts: none of these are normally distributed because values are not continuous and unbounded. Measurements are continuous, but often bounded by zero since you can't have a femur that is -40cm long. Proportions are continuous, but cannot range below 0 or greater than 1. And counts also cannot be negative (you can't have -3 individuals buried in a grave...), and are not continuous: count data are integers, and can't take decimals (how many sites is 5.63?).

Whenever we encounter data like these, which are not normally distributed, we are faced with a choice. In some cases, we can proceed with a regular linear model. Linear modeling is generally pretty robust to violations of the less important assumptions. As long as you're not egregiously violating the assumptions of normal residuals or homoskedasticity, you can sometimes get away with simply running a basic linear model. But sometimes it simply wouldn't make any sense to use an ordinary linear model on your data. What to do then?

2 Generalized Linear Models

You could use a generalized linear model, of course! A generalized linear model, or GLM, is a variation on a regular linear model. When you're dealing with data that violate certain assumptions of an ordinary linear model, you can often use some form of a GLM.

Recall that the six key assumptions of linear models are:

1. The model makes real-world sense
2. The model is additive
3. The model is linear
4. The model has independent errors
5. The model residuals are homoskedastic
6. The model residuals are normally distributed

When a dataset violates assumptions 1, 5, 6, and sometimes 3, we can turn to GLMs. If assumptions 2 and 4 (and sometimes 3) are violated, that requires a bit more of a sophisticated fix that is beyond the scope of this module (i.e. non-linear models and hierarchical/mixed effects models).

2.1 Load data

For this tutorial, let's turn to a human osteological dataset. Ben Auerbach at the University of Tennessee, Knoxville compiled a large dataset on osteometric measurements from 1,538 individuals spanning the Holocene (i.e. the last 11,700 years). The data can be downloaded here, and is also available alongside this tutorial.

https://web.utk.edu/~auerbach/GOLD.htm

The columns in the dataset are coded, so you would need the associated metadata pdf to understand what each means.

https://web.utk.edu/~auerbach/GoldMeasures.pdf

Let's load this dataset into R. Remember that you need to save the .csv file in your working directory, and make sure you set your working directory.

```
goldman <- read.csv("Goldman.csv")
```

If we view the data, or the metadata pdf linked above, we can see that this data frame contains 1,538 rows and 69 columns. The first few columns of this dataframe are the institution housing the remains, the individual's ID in this database, the sex, the age, and their location. Then we get into whether certain skeletal elements are present for that individual or absent, and then linear measurements on those elements followed by some calculated indices.

3 Log Normal Generalized Linear Models

Let's start by discussing log normal data.

Data which are log-normally distributed are very common and can often lead to a violation of linear model assumptions. Recall from our lecture on probability distributions that log normal data are continuous, but bounded by 0. Measurements are the most common type of log-normal data, and that's what we'll be modeling here.

3.1 Log Normal Violations of Linear Model Assumptions

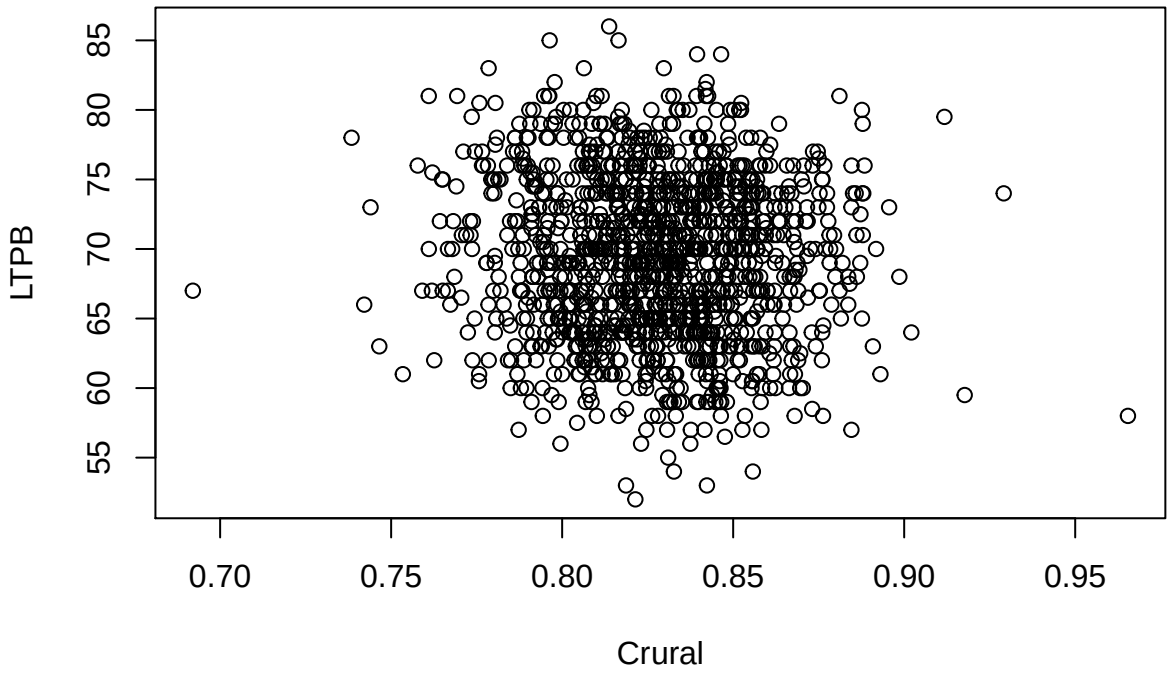
Let's start by modeling whether individuals' left tibia plateau breadth (how wide the top of the tibia is) is related to, or predicted by, their crural index (a metric of the ratio between lower and upper leg lengths).

First, there is some missing data (NAs) in the columns we're interested in, so let's get rid of any rows for these columns which are incomplete. We can do this using the `complete.cases()` function, which returns a logical vector (i.e. TRUE and FALSE values) about whether the input is complete or whether it's not (i.e. an NA value).

```
goldman <- subset(goldman, complete.cases(goldman$LTPB) & complete.cases(goldman$Crural))
```

So with this cleaned up data, let's first plot this relationship.

```
plot(LTPB ~ Crural, data = goldman)
```



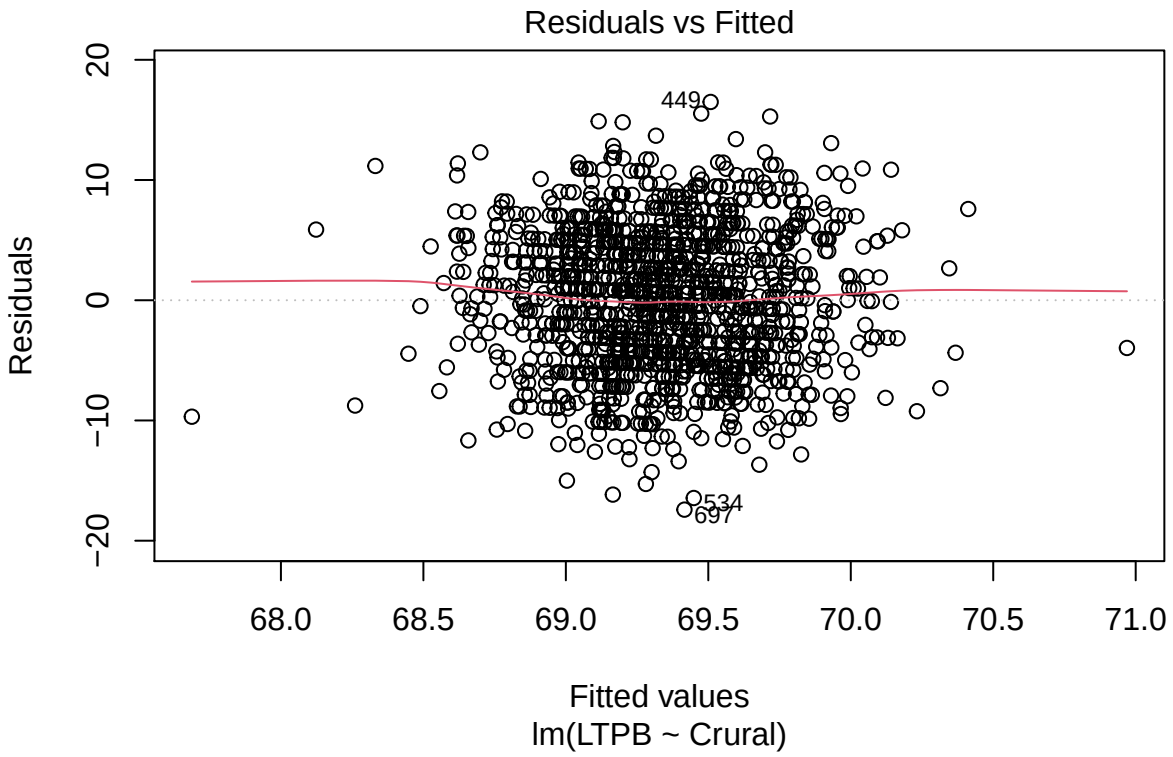
There doesn't appear to be a relationship here at all, though we can model it to see.

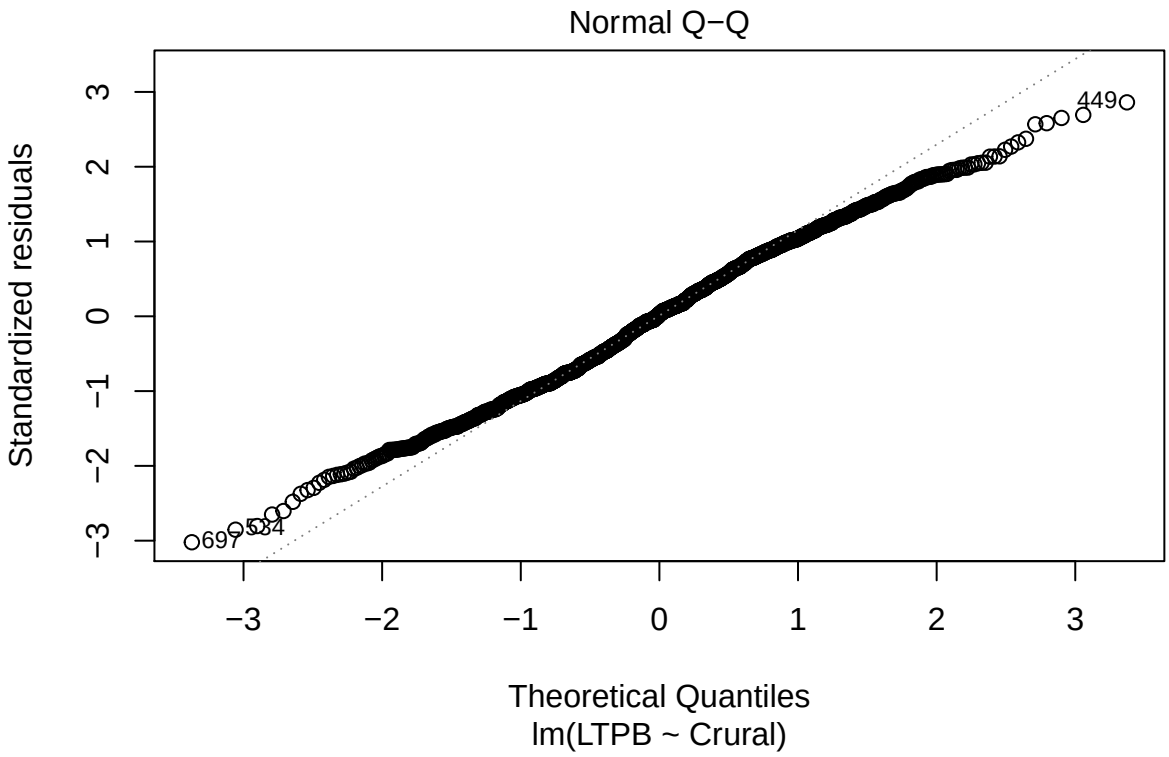
But... Can we use the `lm()` function for these data? Remember that the `lm()` function is meant for a linear model for which all six assumptions from above are met. Are we violating any assumptions with these data?

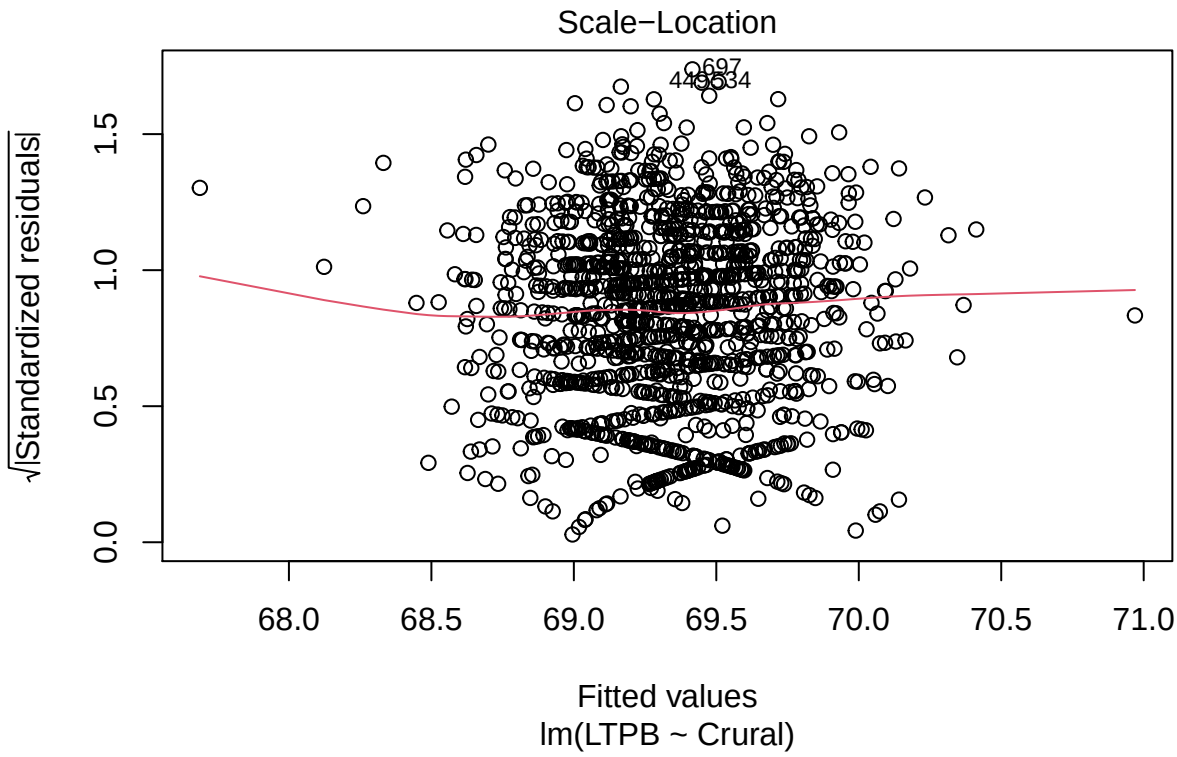
For the purposes of illustrating the problem, let's try to fit a linear model to these data.

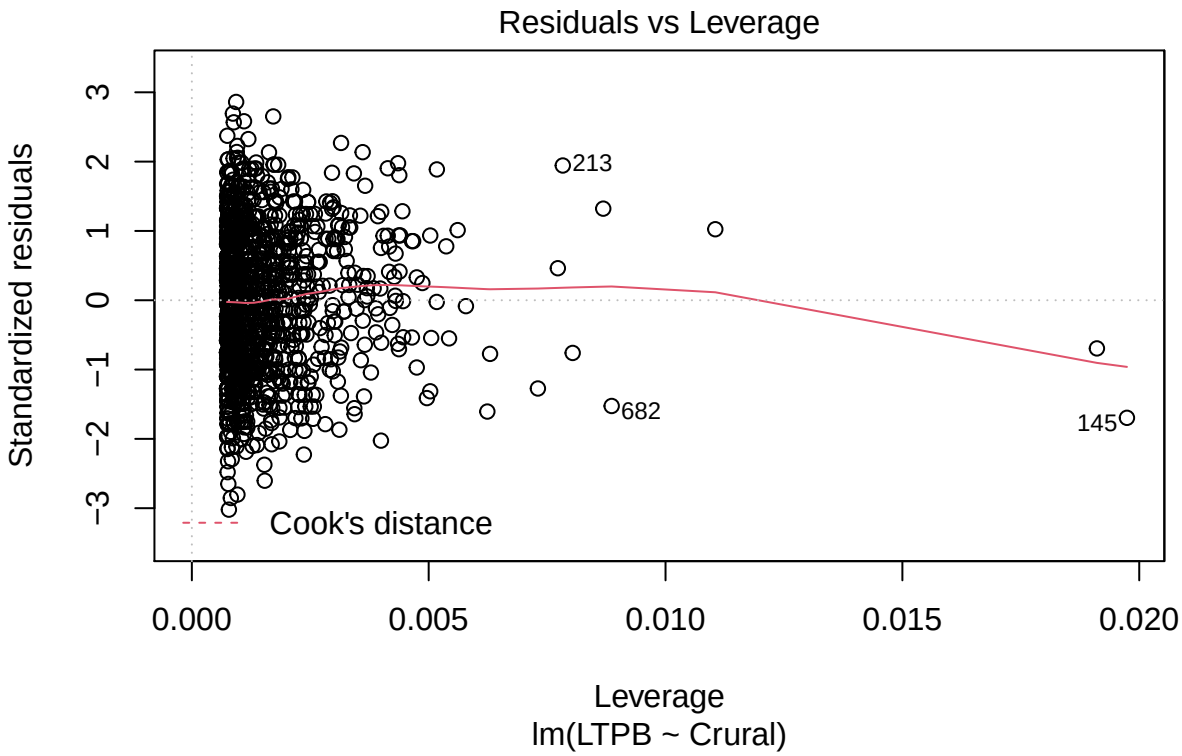
```
bad.mod <- lm(LTPB ~ Crural, data = goldman)

plot(bad.mod)
```


Residuals vs Fitted
Residuals
20
10
0
−10
−20
449
534
697
68.0
68.5
69.0
69.5
70.0
70.5
71.0
Fitted values
lm(LTPB ~ Crural)


Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(LTPB ~ Crural)
449
697
534
−3
−2
−1
0
1
2
3


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
68.0
68.5
69.0
69.5
70.0
70.5
71.0
Fitted values
lm(LTPB ~ Crural)



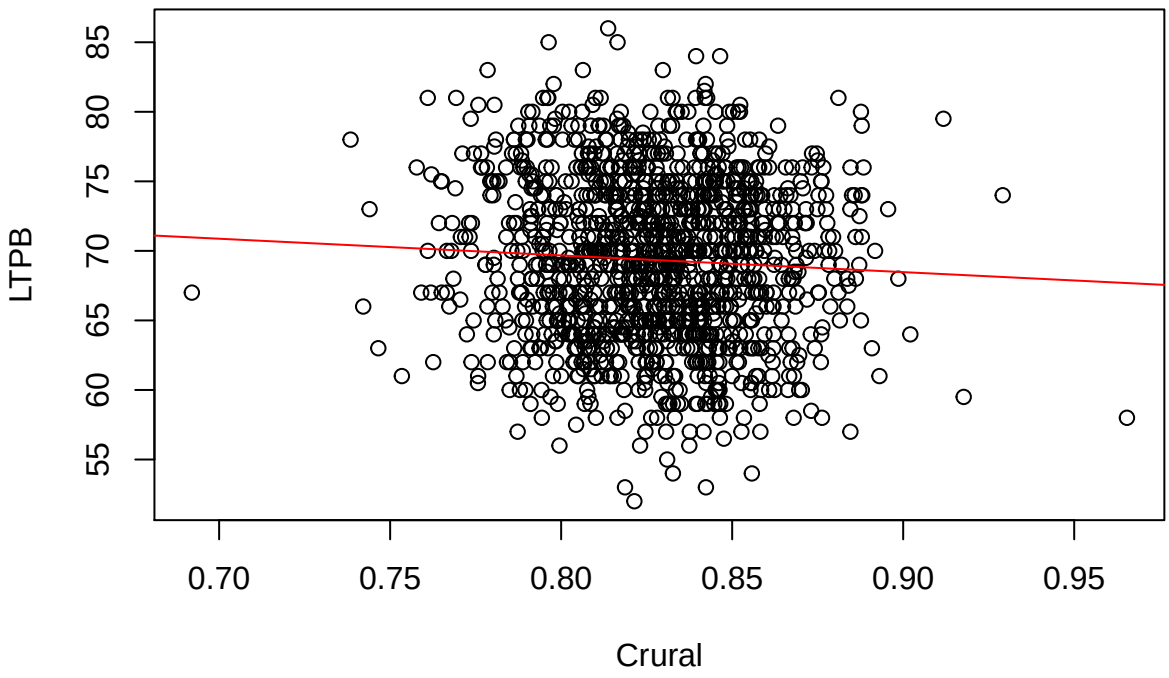
All of the model diagnostic plots look okay, right? They do, more or less. So the problem is not heteroskedasticity (assumption 5), normality of the residuals (assumption 6), a nonlinear fit (assumption 3), or even with non-independence (assumption 4). As a note however, part of the reason the residuals look okay is that our response values are far from zero. If we were modeling values closer to 0, the lower boundary for log-normal data, our residuals might not look so good since our response variable can't go below zero... Just something to keep in mind.

But the problem with this model isn't actually apparent from just the model diagnostic plots here. If we didn't think critically about our data, we might proceed with a regular old Gaussian linear model, and possibly end up with some bad results.

But without even plotting anything, I can tell you we are violating the most important assumption - assumption number 1. A Gaussian linear model applied to these data would not make biological sense based on what we know about linear measurements. **You cannot have a left tibia plateau that is -50mm wide** A negative length measurement can't exist in this sense, but a linear model doesn't know that. Let's visualize this problem.

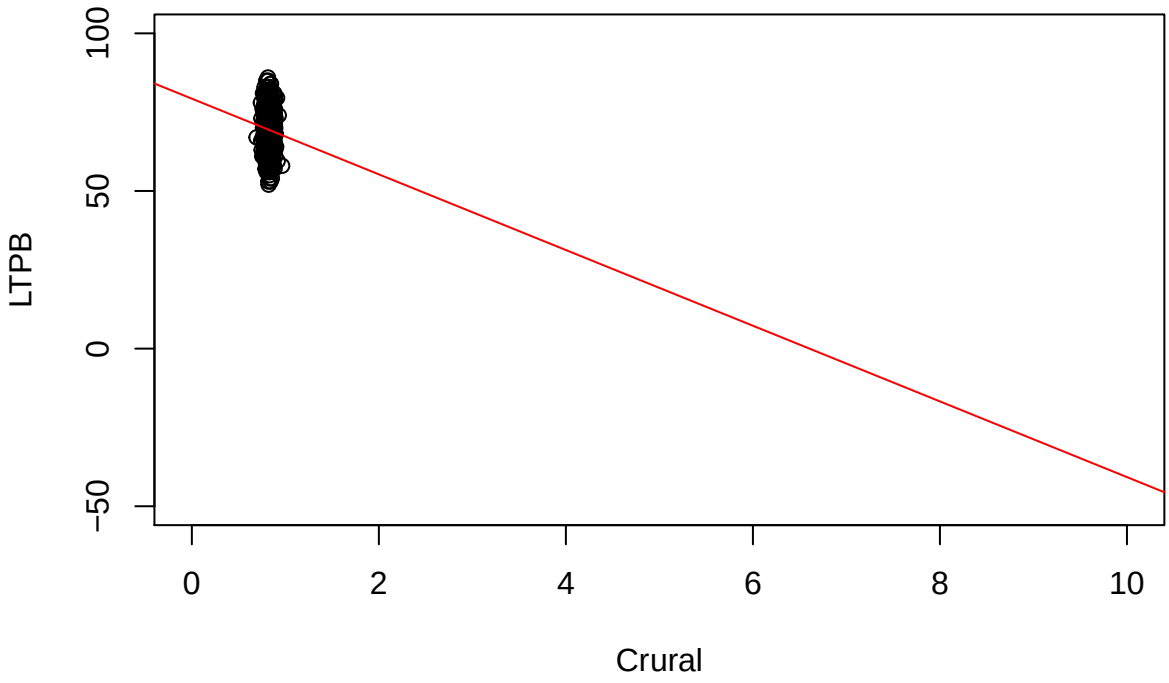
Here, I've plotted our model fit on top of our data on humerus measurements. The model fit is clearly bad, but it seems to be predicting realistic breadth values at least.

```
plot(LTPB ~ Crural, data = goldman)
abline(bad.mod, col = "red")
```



But what if we extend the plotting dimensions?

```
plot(LTPB ~ Crural, data = goldman, ylim = c(-50, 100), xlim = c(0, 10))
abline(bad.mod, col = "red")
```



You can see that our data are now clustered in the top left of our plot while our model fit just keeps on going towards negative infinity. That's what basic, Gaussian linear models fit with the `lm()` function do. This is because they think they're being fit to perfectly normally distributed data: data which can take any value between negative and positive infinity. So our model fit, shown in red, just keeps going in a negative direction as the Crural Index goes up.

Now in this particular case, a crural index of 9 is also biologically impossible, but ignore that for now since the response variable is what matters here. A person's tibial plateau cannot be -20 mm in breadth, but that's what our model predicts for a crural index value of 8. This isn't realistic at all! Thus we're violating the most important model assumption, even if our residuals look good.

3.2 Fitting a Log-Normal GLM

Since this Gaussian linear model isn't predicting realistic values, let's fit a model that can. This will be a log-normal generalized linear model. We'll fit this model using the `glm()` function.

```
tpb.mod <- glm(LTPB ~ Crural, data = goldman, family = gaussian(link = "log"))
```

You can see that using the `glm()` function is very similar to using the `lm()` function, just with two extra arguments: `family` and `link`. The `family` argument applies to the distribution of our model residuals. The `lm()` function is meant for Gaussian/normal data, so in essence the family is set to "gaussian" by default and it is assumed that the residuals are normally distributed. But in a GLM, we can change this family argument to specify different distributions. What this does is tell R that our model residuals are not necessarily normally distributed, but are distributed according to different types of probability distributions.

But here, since we're working with a log-normal model, we're actually still specifying a normal distribution for our model residuals. That's the "normal" in "log-normal". This is therefore referring to the *stochastic*

part of our model. What the "log" is then referring to is a transformation applied to the *deterministic* part of our model: the actual model fit, not the distribution of residuals around it.

After we specify `family = gaussian` in our `glm` function, we put `(link = "log")`. This bit of code tells R that the *link function* connecting our predictor and response data is logarithmic, not linear. To see the effect of this, let's run a logarithm.

Recall that log-normal distributions are bounded by zero, but can range up from there to positive infinity. In contrast, gaussian distributions are unbounded and run from negative infinity to positive infinity.

What's the logarithm of zero?

```
log(0)
```

```
## [1] -Inf
```

Negative infinity! The lower boundary for a Gaussian distribution!

So what's the log of positive infinity?

```
log(Inf)
```

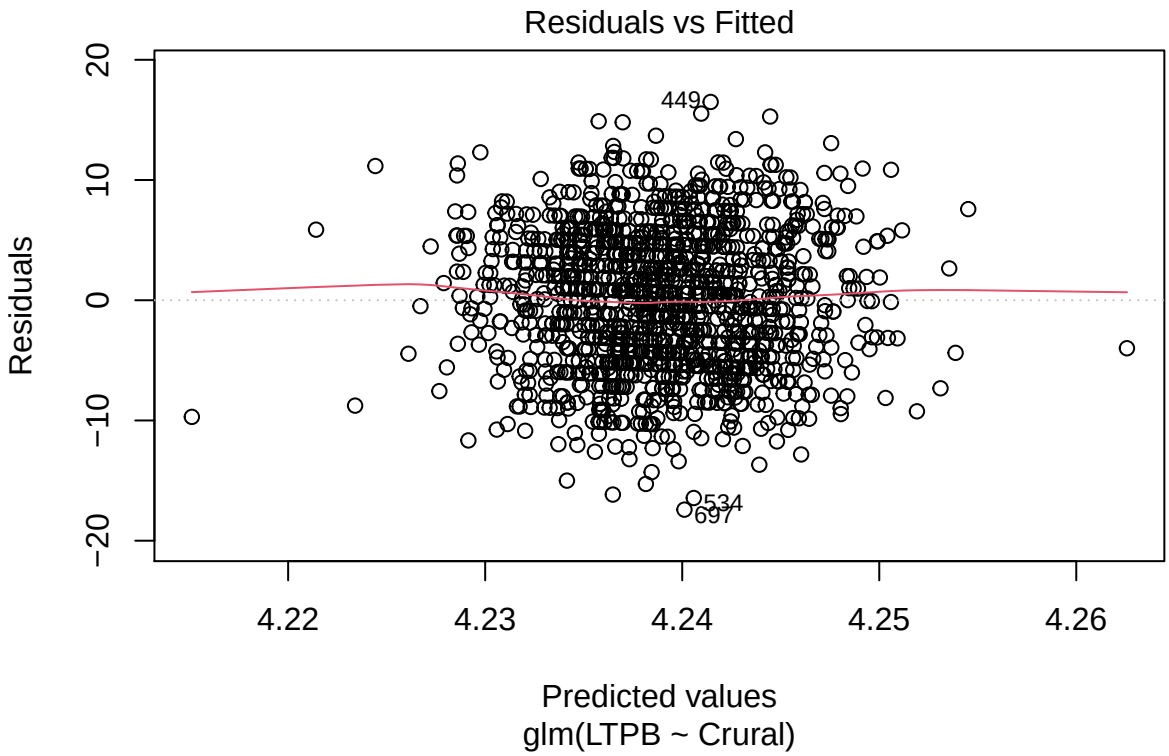
```
## [1] Inf
```

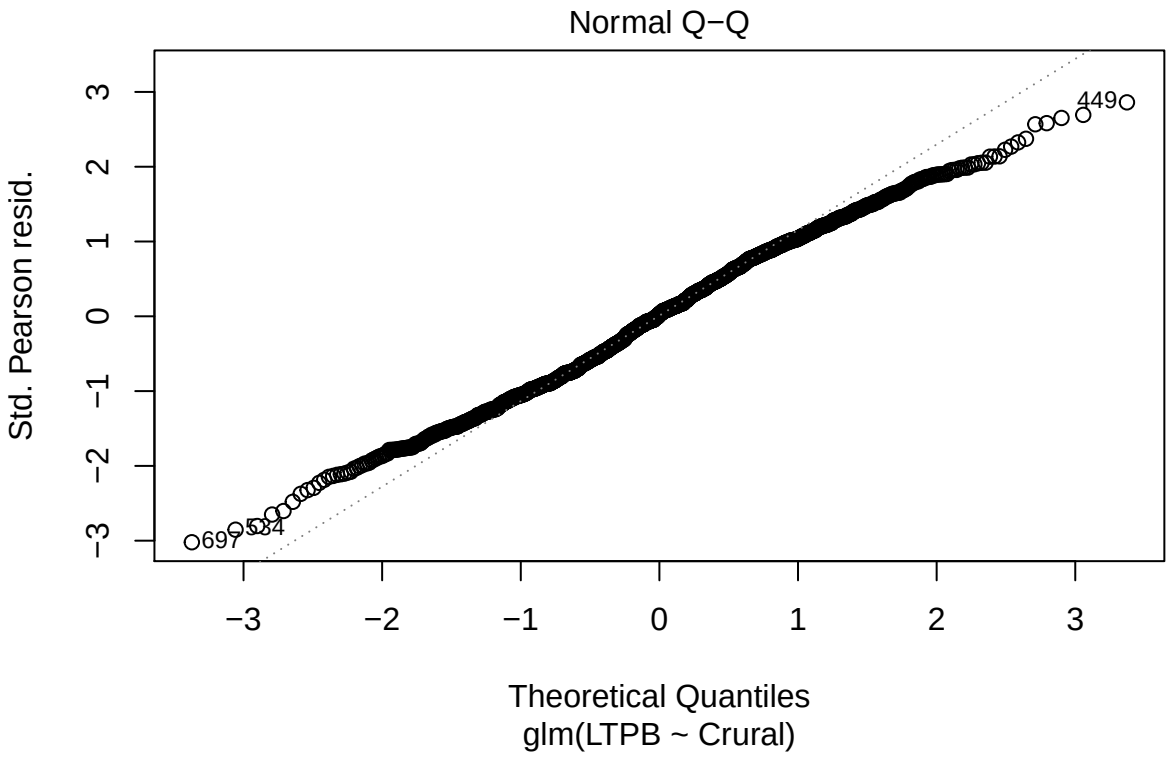
Positive infinity - the upper bound of a Gaussian distribution!

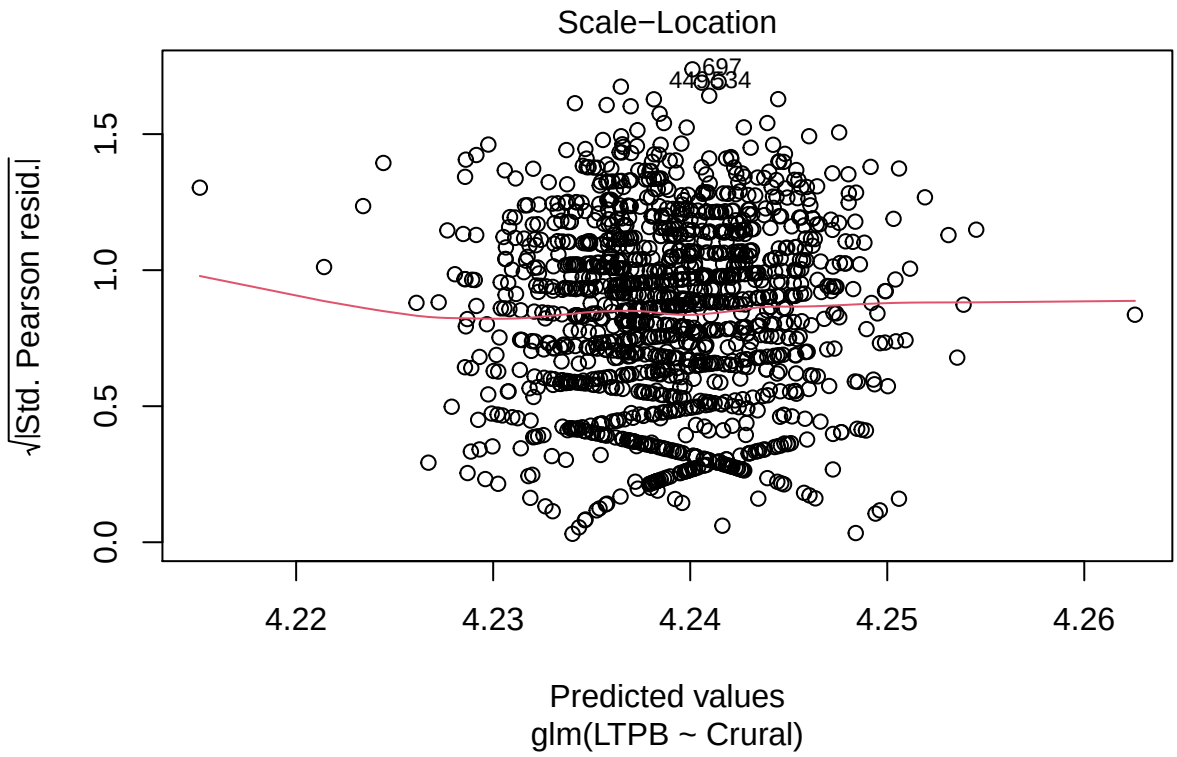
So when you take the log of data like ours, which are bounded by zero, it transforms it onto a Gaussian scale! This is why log transformation works on things like measurement data so well. Our GLM is applying this transformation internally, however, so we don't need to ever manipulate our data.

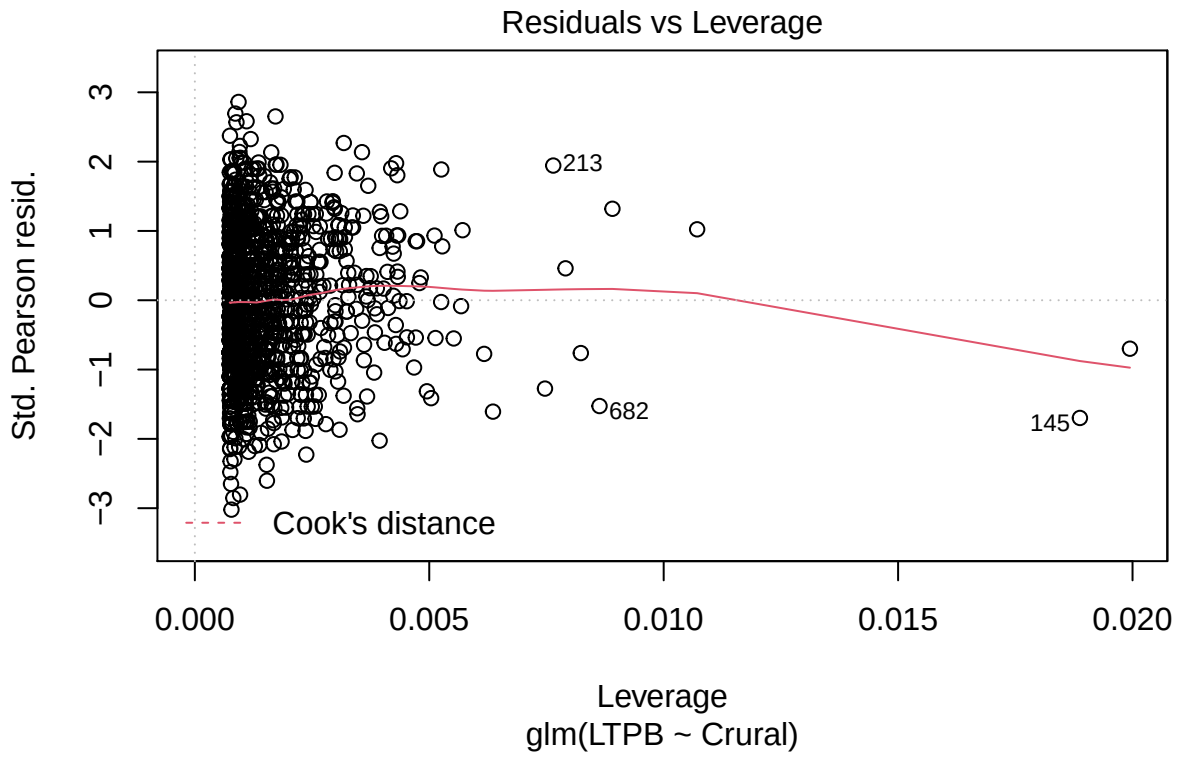
Now let's inspect the diagnostic plots of our GLM.

```
plot(tpb.mod)
```


Residuals vs Fitted
Residuals
20
10
0
−10
−20
449
534
697
4.22
4.23
4.24
4.25
4.26
Predicted values
glm(LTPB ~ Crural)


Normal Q−Q
Std. Pearson resid.
Theoretical Quantiles
glm(LTPB ~ Crural)
449
697
534
−3
−2
−1
0
1
2
3


Scale−Location
697
√|Std. Pearson resid.|
0.0
0.5
1.0
1.5
4.22
4.23
4.24
4.25
4.26
Predicted values
glm(LTPB ~ Crural)



The plots generally look good, but they already did before. To really see the difference, we should plot our model fit.

3.3 Plotting a GLM

To visualize a GLM, we can't just use the `abline()` function as we do for `lm` objects. GLMs are a bit more complicated... But what we can do is make use of the `predict()` function, as we did previously to plot confidence intervals.

First, we need a sequence of many values along our x-axis. I like to make sure these values extend beyond the limits of our predictor variable so that the model fit illustrates that values could be predicted beyond the range of the Crural Index values we've observed.

```
new.x <- data.frame("Crural" = seq(0, 10, length.out = 100))
```

Now to predict the model fit, we can use the following code.

```
pred.y <- predict(tpb.mod, newdata = new.x, type = "response")
```

The `predict` function takes our GLM model object and predicts response values over the range specified in our `new.x` variable: it feeds `new.x` into the model equation in place of x_i and calculates μ_i, which is our model fit.

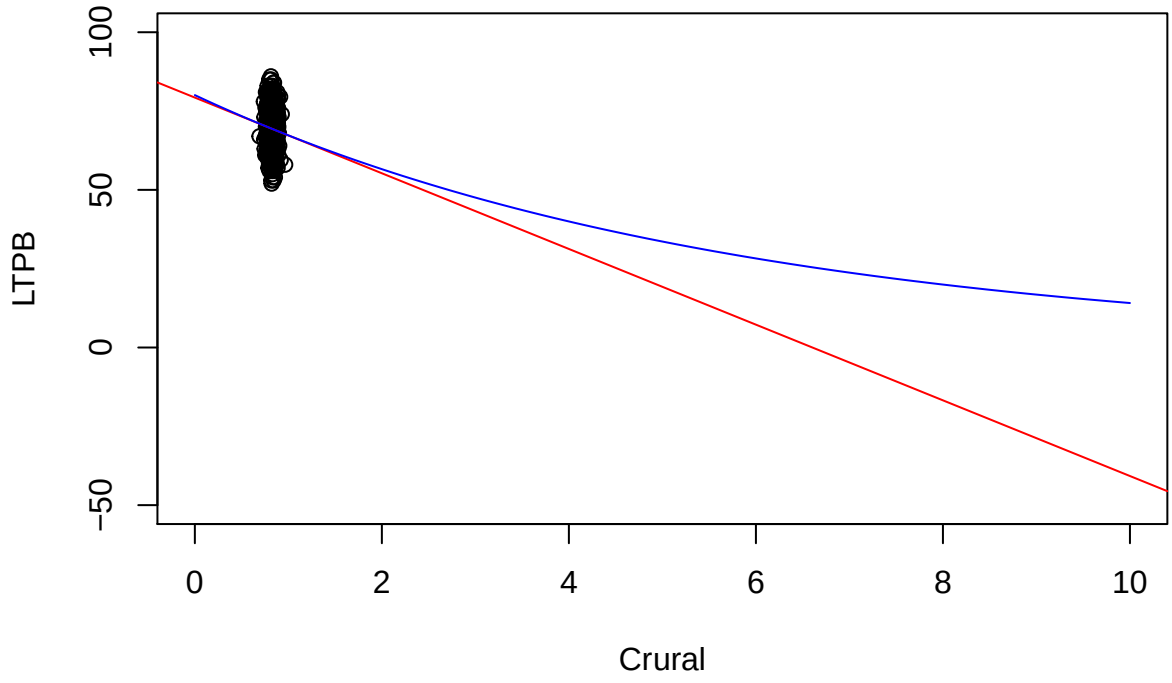
We must also specify the `type` argument. This argument controls whether we want values to be predicted on the *link* scale or the *response* scale. The "link" scale is the scale of the transformed response variable. In our case, this is log scale because our link function was set to "log". So values on the link scale are in log

space. In contrast, the "response" scale is the real-world scale: it's our un-transformed response values. So here, by saying `type = "response"`, we're telling the predict function to return real-world values that are un-transformed, not values on the transformed scale (which would be harder to interpret).

So let's plot our data, then plot the original `lm` model fit that extended from positive to negative infinity. Then, let's add our `glm` model fit from a log-normal model.

```
plot(LTPB ~ Crural, data = goldman, ylim = c(-50, 100), xlim = c(0, 10))
abline(bad.mod, col = "red")

#add the GLM model fit
lines(pred.y ~ new.x$Crural, col = "blue")
```



We can see that our GLM, in blue, is a much better representation of reality! Like our measurement data, which can't be negative, our model is now constrained so that it can't be negative either. We're now no longer violating assumption 1 of linear modeling - the most important assumption!

While we're plotting model fits, let's add the confidence intervals. This is also more complex that confidence intervals for regular, Gaussian linear models.

Just like all 95% confidence intervals, what we basically want to do is take our model fit and add or subtract 1.96 standard deviations (the z-score capturing the middle 95% of a probability distribution) to get the upper or lower confidence interval.

This would work if our data were Gaussian... But in a GLM, our model has transformed our data onto the link scale. If we just took the model fit we plotted above in blue, which is on the response scale, and subtracted 1.96 standard deviations, we might accidentally end up including negative values in our confidence interval. This would be impossible, because our data are bounded by zero. So we actually need to subtract

1.96 standard deviations from the model fit on the *link* scale, not the response scale like we used to plot our model object.

To do this, we'll still use the `predict` function and that `new.x` object we made. But now, we'll specify `type = "link"`. We'll also add another argument, `se.fit = TRUE`, because we want the predict function to not only return the model fit but also the standard errors (i.e. standard deviations) associated with the model fit.

```
pred.y <- predict(tpb.mod, newdata = new.x, type = "link", se.fit = T)
```

Now we have overwritten our `pred.y` object from before. This `pred.y` object contains a few different vectors now, most importantly `$fit` and `$se.fit`. These are our model fit and the associated errors, but on the link scale (i.e. in log space) not on the scale of our actual data.

So now that we have these pieces, we need to calculate our confidence intervals. We do this by simply taking the model fit (`$fit`) and adding or subtracting our errors (`$se.fit`) multiplied by 1.96. We can do this in a data frame to keep everything in one place.

```
tpb.mod.fit <- data.frame("fit" = pred.y$fit,
                          "upper" = pred.y$fit + (pred.y$se.fit * 1.96),
                          "lower" = pred.y$fit - (pred.y$se.fit * 1.96))
```

So now we have calculated our confidence intervals. But everything in this data frame is still on the link scale, so we need to transform it to the response scale. We can do this by using the inverse of the link function. Mathematical functions like logatithms often have an inverse function that will undo the math. In the case of logarithms like ours, the inverse function is exponentiation. If we take the exponent of a value on our link scale, which is a log scale, we will return our value to our response/real-world scale.

Recall that when we ran `log(0)`, R told us the answer was negative infinity. Let's now take the exponent of the log of 0.

```
exp(log(0))
```

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

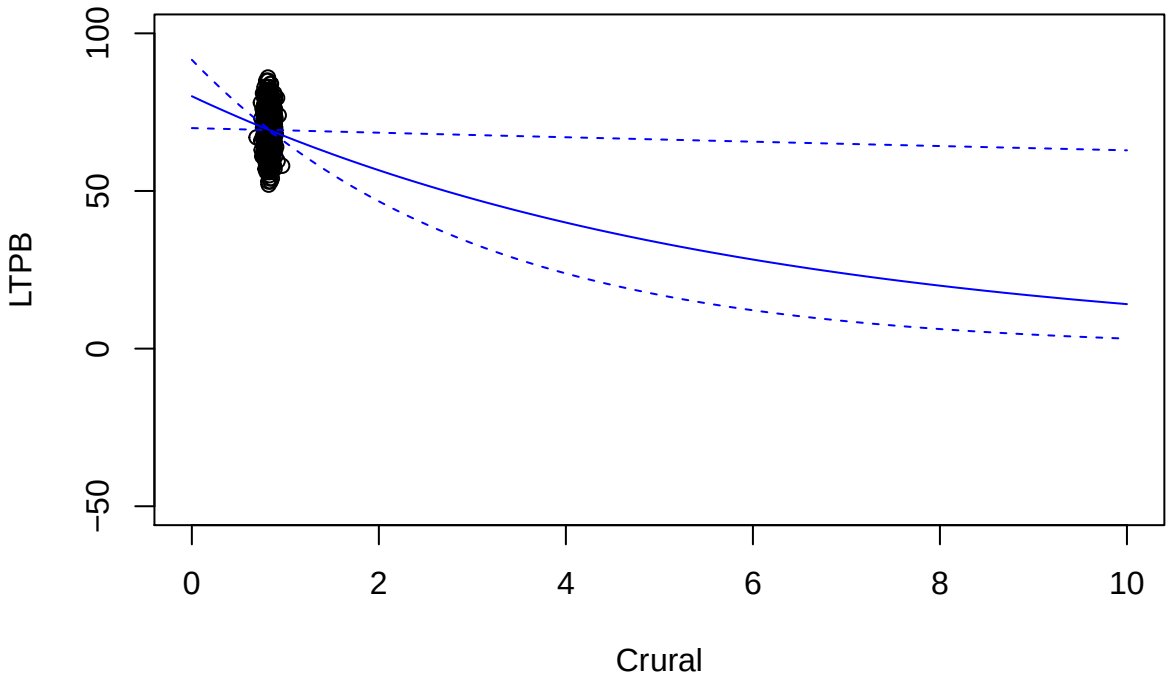
Our answer is zero - we've just used exponentiation to transform a log-scale value back to the real-world! We can do this for our model fit and confidence intervals now.

```
tpb.mod.fit <- exp(tpb.mod.fit)
```

Now our tpb.mod.fit object contains our model fit, upper, and lower confidence intervals, but all on our response scale again! Let's plot these to make sure they are predicting reasonable values.

```
plot(LTPB ~ Crural, data = goldman, ylim = c(-50, 100), xlim = c(0, 10))

lines(tpb.mod.fit$fit ~ new.x$Crural, col = "blue")
lines(tpb.mod.fit$upper ~ new.x$Crural, col = "blue", lty = 2)
lines(tpb.mod.fit$lower ~ new.x$Crural, col = "blue", lty = 2)
```

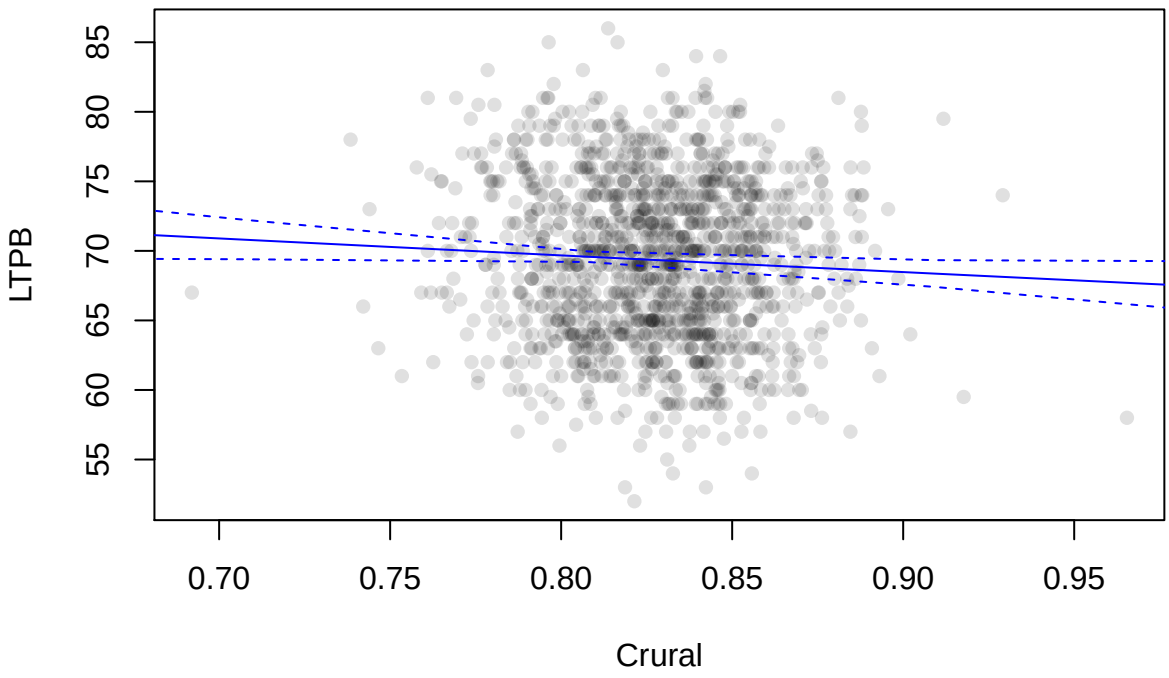


Sure enough, our confidence intervals do not extend below zero, nor does our model fit. We're predicting realistic values with our model.

And of course, we can clean this plot up and make it include a more realistic range of x-values too.

```
plot(LTPB ~ Crural, data = goldman, col = "#00000020", pch = 16)

lines(tpb.mod.fit$fit ~ new.x$Crural, col = "blue")
lines(tpb.mod.fit$upper ~ new.x$Crural, col = "blue", lty = 2)
lines(tpb.mod.fit$lower ~ new.x$Crural, col = "blue", lty = 2)
```



3.4 GLM Model Summary

So now we know that our model is accounting for the violated assumptions of linear modeling through a log transformation, let's look at the model summary.

```
summary(tpb.mod)
```

```
##
## Call:
## glm(formula = LTPB ~ Crural, family = gaussian(link = "log"),
##     data = goldman)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -17.4157   -4.3808    0.1515    4.5043   16.4919
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.38270    0.06882  63.684   <2e-16 ***
## Crural      -0.17359    0.08314  -2.088    0.037 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 33.26489)
```

```
##
##     Null deviance: 44786  on 1343  degrees of freedom
## Residual deviance: 44641  on 1342  degrees of freedom
## AIC: 8528.2
##
## Number of Fisher Scoring iterations: 4
```

You'll see that the summary output for a `glm` object looks similar to that of an `lm` object, but with some interesting differences...

At first glance, the coefficients section looks like that of an `lm` object. But actually, the coefficient estimates provided here are on the link scale. So for example, the intercept of 4.38 is not actually an intercept of 4.38, but an intercept of `exp(4.38)`. This is because, again, we have to use the inverse link function to get our link-scale coefficients/values back into the real world. The inverse of a logarithm is an exponent, so we need to exponentiate our values.

But! The most important thing to remember here is that you must add your coefficient estimates together **before** you exponentiate them! So if you want to calculate the estimated mean LTBP value for a crucal index of 3, you would take $4.38 + (-0.17 * 3)$ and **then** exponentiate it. So the predicted LTPB value for a crucal index of three would be $exp(4.38 + (-0.17 * 3))$. The intercept plus the coefficient, then exponentiated. The exponent of 4.38 plus the exponent of -0.17 is not the same thing! Check it out:

```
exp(tpb.mod$coefficients[1] + (tpb.mod$coefficients[2] * 3)) #right
```

```
## (Intercept)
##      47.557
```

```
exp(tpb.mod$coefficients[1]) + exp(tpb.mod$coefficients[2] * 3) #wrong!
```

```
## (Intercept)
##    80.64797
```

Those values are not even close. So you really have to make sure you're doing all the stuff you need to do, and then you take the exponent of the answer - not the exponent of each piece.

But of course, the predict function largely means that you don't need to do this math by hand. We can just plug in a crural index value of 3 to a predict function and run that to get the answer. Just remember to include `type = "response` if you want the answer on the response scale of real-world units.

```
predict(tpb.mod, newdata = data.frame("Crural" = 3), type = "response")
```

```
##      1
## 47.557
```

3.5 GLM p-values and D2 values

Now, returning to the summary output, remember that the p-values reported in the coefficients section are not necessarily the ones you want to care about. But if you look at the bottom of the summary output, there is no p-value for the overall model! There's also no R^2 value! Where did they go?

Remember from lecture that GLMs are not fitted using the same method as Gaussian linear models. Because the assumptions of a Gaussian distribution make the math easier, linear models are fitted by minimizing the sum of the squared residuals. But GLMs, since they don't assume a Gaussian distribution, can't be fitted

that way. GLMs use maximum likelihood estimation, which is a computational approach that iteratively explores parameter options to find the one that makes our data most likely.

What this means for us is that we can't calculate p-values and R^2 values the usual way. This is why the `summary(glm)` output doesn't include these important statistics. So we need to take additional steps.

What we need to do is run what's called a *likelihood ratio test*. A likelihood ratio test, or LRT, is a type of statistical hypothesis test that compares one model to another form of the same model.

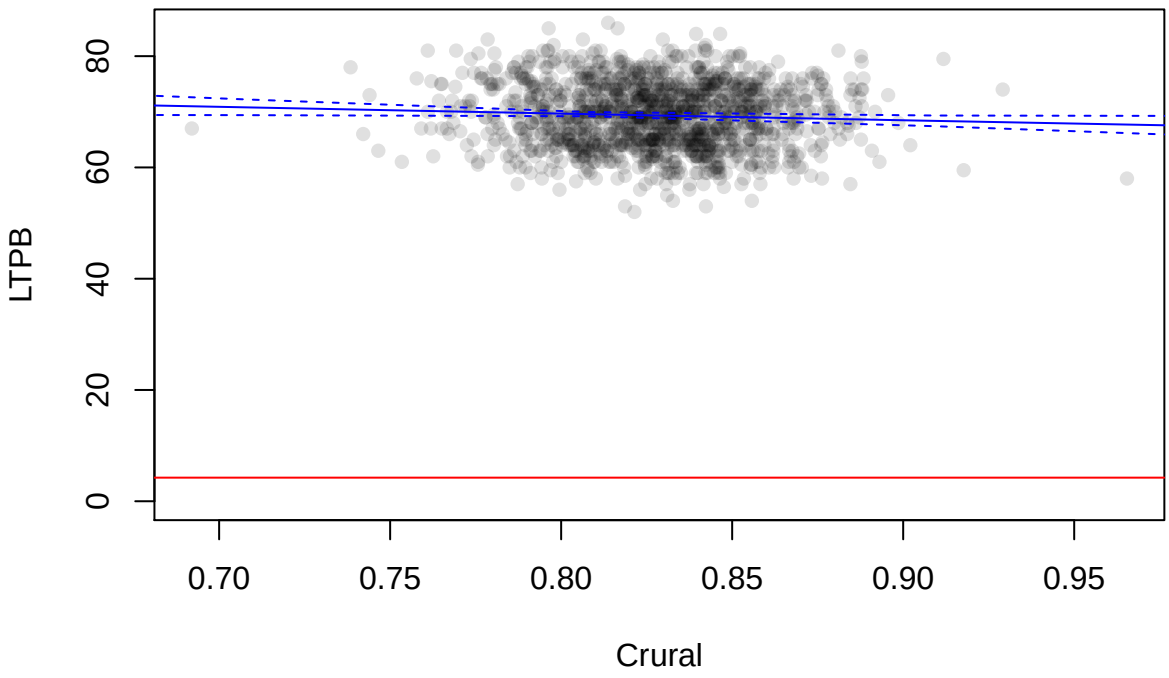
When we want to calculate a p-value for a GLM, we use an LRT to compare our model to a stripped down version. This stripped down version is (often) an *intercept-only* model. This means that instead of having a β_0 and a β_1, the model only has a β_0. It tries to predict our data using only the intercept part of the model equation. This is obviously not often a good model because it doesn't actually account for the predictor variable at all, and the model fit is a horizontal line.

Thus, an intercept-only version of our model would look like the red line here, while our actual model fit is in blue. To make an intercept-only model, simply write `y ~ 1` instead of `y ~ x`. In this context, `1` denotes an intercept-only model where there is no predictor variable.

```
plot(LTPB ~ Crural, data = goldman, ylim = c(0, 85), col = "#00000020", pch = 16)

#define an intercept only model for LTPB and plot it
int.only <- glm(LTPB ~ 1, data = goldman, family = gaussian(link = "log"))
abline(int.only, col = "red")

#plot our actual model fit and CIs
lines(tpb.mod.fit$fit ~ new.x$Crural, col = "blue")
lines(tpb.mod.fit$upper ~ new.x$Crural, col = "blue", lty = 2)
lines(tpb.mod.fit$lower ~ new.x$Crural, col = "blue", lty = 2)
```



We can see that these two models are very clearly quite different. The intercept-only model, in red, is just a horizontal line and is actually very far removed from our actual data. It is clearly not useful at predicting the relationship between the crural index and the LTPB values. But it's not meant to - it's intended to be a null model to which we can compare our actual model.

To do that, we can use the `lrtest()` function from the `lmtest` package. There are other options for likelihood ratio tests, but I like this one. To use this function, load the `lmtest` package and check out the help page for the `lrtest()` function.

```
library(lmtest)
```

The `lrtest()` function requires only your model object, but can take other arguments (like additional model objects). So let's feed it our model and let it compare our model to an intercept-only version.

```
lrtest(tpb.mod)
```

```
## Likelihood ratio test
##
## Model 1: LTPB ~ Crural
## Model 2: LTPB ~ 1
##   #Df  LogLik Df  Chisq Pr(>Chisq)
## 1   3 -4261.1
## 2   2 -4263.3 -1 4.3457     0.0371 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This `lrtest` output tells us that our p-value for the model is 0.037: significant at the 0.05 level. The column labeled Df (not #Df) contains our degrees of freedom for our LRT. In this case, the degrees of freedom that is relevant is the difference in degrees of freedom between our intercept-only model and our full model. In this case, our model has one more parameter than an intercept only model, so our degrees of freedom is equal to 1. We also want to know the X^2 test statistic (this likelihood ratio test uses a X^2 distribution to calculate a p-value). So, to report the result of this LRT, I would say that "our model performs significantly better than an intercept-only model" or that "the crural index has a significant effect on LTPB" and report the relevant statistics as: (X^2=4.35, df = 1, p < 0.05).

But just because our model is significant doesn't mean that it's necessarily useful or powerful. To figure out the explanatory power of our model, we need something like an R^2 value. But because of the way they're fit, GLMs don't have R^2 values...

There are several versions of R^2 values out there that one could calculate for a GLM. These are called pseudo-R^2 values and each has its strengths and weaknesses. I prefer to avoid calculating a pseudo-R^2 value at all, however. What I like to calculate is called D^2, or the deviance-squared statistic. D^2 is an expression of the proportion of deviance explained by your model. This is not quite the same as an R^2 value, which reflects the proportion of variance in your response variable that is explained by your predictor variable. Deviance is not really the same thing as variance (but it is related in a sense).

Deviance is a measure of how poorly your model is fitting the data. Specifically, this is the null deviance: the deviance of the null model relative to a perfect model. There's *a lot* more to understand about deviance than what we'll discuss here, but for now let's just say that if your model explains a lot of the deviance, it's better than a model that explains very little deviance. Therefore, as a proportion, the D^2 statistic ranges from 0 to 1, just like an R^2 value. The closer to 1.0, the better your model is at explaining what's going on with the response variable.

To calculate D^2, let's load the `modEvA` package.

```
library(modEvA)
```

Now let's use the `Dsquared()` function from within the `modEvA` package to calculate the D^2 statistic for our model.

```
Dsquared(tpb.mod)
```

```
## [1] 0.003228216
```

Our model's D^2 is equal to 0.003. This is terrible. That means that our model is only explaining 0.3% of the null deviance... That's nothing at all. We'd report this by saying that our model explains very little deviance (D^2=0.003), meaning that our model has very little explanatory capability over LTPB.

Our p-value of 0.03 and our D^2 value of 0.003 illustrate a really great point about modeling. Significance is not everything! We have a statistically significant model that explains next to nothing about our response variable. What do we make of this?

In this case, it's important to remember what a p-value really means. A p-value is simply the probability of our data (or more extreme data) if our null hypotheses were true. that means that, in this case, our model is significant because we're unlikely to get such a result if the null hypothesis (intercept only model) were true. But that doesn't actually mean that our model is worth a damn. In this case, it's clearly not. We're only explaining less than half a percent of the null deviance, so our our model of the crural index really isn't capable of explaining anything at all about LTPB!

It's therefore very important that you do not convince yourself that a model is good by only a significant p-value. You *must* consider the fit of the model too, and in this case, our model sucks. I would report this model as significant (LRT: X^2=4.35, df = 1, p < 0.05), but that there is no real effect of crural index on LTPB (D^2 = 0.003).