# Generalized Linear Modeling Part II: Poisson 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 Poisson Generalized Linear Models

Now that you've learned the basics of GLMs using a log-normal family GLM, let's move on to another type: Poisson family.

We'll once again work with the Goldman dataset from https://web.utk.edu/~auerbach/GOLD.htm.

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

Let's prepare some count data from this dataset to model. Let's investigate whether three different locations have different numbers of missing elements per skeleton. This could be useful if we felt like taphonomic loss or degradation destroyed more elements in certain locations than others.

First, let's take the sum of columns 8 through 16 in the `goldman` dataset. These columns represent the presence or absence of measureable elements for each skeleton. A value of 0 means it is present, which is a rather counter-intuitive decision by the database designer, but it allows us to simply add up the number of "1"s in these columns by row to count the total number of missing elements - which will be the variable of interest in our model. To calculate this number, let's create a new column in the goldman dataframe called "Missing" and assign to it the summed missing element counts from columns 8-16. We can do this using the `apply()` function, which applies the specified funtion (`sum`) to the specified data `goldman[, 8:16]`), by either the row (`1`) or the column (`2`). In our case, we want the sums of each row across these columns, so we specify the arguments as follows:

```
#take the sum of all missing elements for each individual skeleton
goldman$Missing <- apply(goldman[, 8:16], 1, sum)
```

Now, let's subset the `goldman` data frame to just include three of the best-sampled locations: Alaska, Ohio, and Japan. We can use the `subset()` function to do this as follows.

```
#subset the data frame to just contain the three relevant locations
gold.sub <- subset(goldman, Location == "Alaska, United States" |
                            Location == "Ohio, United States" |
                            Location == "Japan")
```

With this new object, `gold.sub`, let's relabel the location names. These are pretty long, so let's replace them with abbreviations using the `recode()` function from the `dplyr` package.

```
#load the dplyr package
library(dplyr)

#rename the locations with abbreviated versions
gold.sub$Location <- recode(gold.sub$Location, "Alaska, United States" = "AK",
                                               "Ohio, United States" = "OH",
                                               "Japan" = "JP")
```

And finally, we ought to tell R that the `$Location` column in our `gold.sub` data frame is a vector of *factors*. Remember that factors are what R calls categorical data. Each of these three locations is a category that we'll use in our predictor variable to see whether the number of missing bones per skeleton is consistent across all three. Right now, R understands that the location names are part of a character vector (i.e. words), but it doesn't understand that all the `"AK"` words represent the same group of data. You can use the `str()` function to inspect the structure of this vector. Using the `as.factor()` function, we can change how R is saving the structure of that column.

```
#change the data type to factor so that R understands these are categories, not just words
gold.sub$Location <- as.factor(gold.sub$Location)
```

## 1.1 Poisson Violations of Linear Model Assumptions

Recall that a Poisson probability distribution is one that is discrete, and bounded by 0. Therefore, it works well for certain count data, which are integers (i.e. no decimals).

Integer/count data therefore violate several of the assumptions of linear models. Most obviously, the residuals from modeled count data will not be normally distributed. As counts cannot be decimals or negative, that really prevents residuals from a model based on count data from conforming to a normal distribution. Thus assumption 6 of linear modeling is violated right off the bat with this type of data.

Assumption 5, homoskedasticity of residuals, is also violated right off the bat. Very often, count data are quite heteroskedastic. This is why a Poisson distribution actually assumes that your data are heteroskedastic by default. We won't get into why, but a Poisson distribution actually only has one parameter - lambda - and assumes that the dispersion of your data increases with your predictor variable. The implication of this is that a Poisson GLM actually assumes that your data are heteroskedastic, not homoskedastic like a Gaussian linear model. So assumption 5 is also violated for count data, pretty much by default.

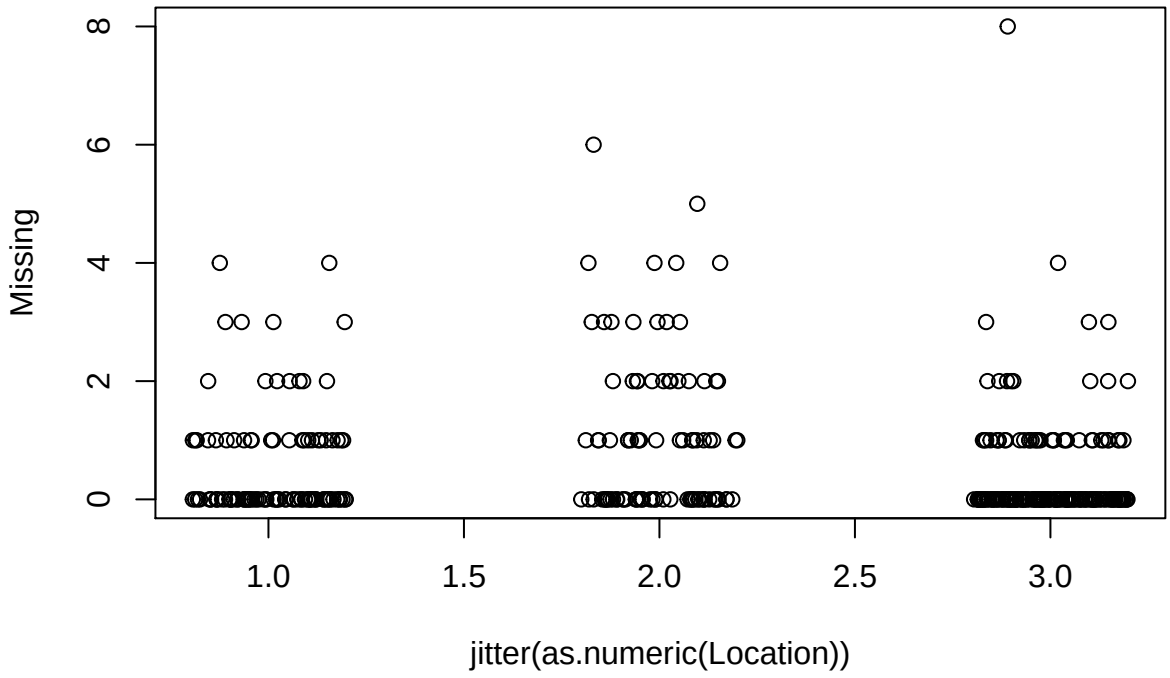
And finally, assumption 1 is also violated by definition. If you're modeling count data, you can't have negative values. Thus a Gaussian linear model, predicting values from negative to positive infinity, will predict unrealistic values for count data. Recall how we extended the plotting window for our log-normal model to see that it predicted impossible negative values.

Since assumptions 1, 5, and 6 are all being violated pretty much by definition, a Poisson family GLM can be your best course of action when modeling count data.

## 1.2 Fitting a Poisson family GLM

Before we build our model, as always, we should visualize our data to see what we're dealing with. Since boxplots obscure a lot of the data, I prefer scatterplots for this sort of thing. And since we're plotting values between three categories (i.e. factors, in R-speak), let's jitter the points so we can better see them. Remember that jittering involves adding random noise to each point to spread them out when a bunch would otherwise be plotted right on top of each other.

```
#plot missing element numbers by location
plot(Missing ~ jitter(as.numeric(Location)), data = gold.sub)
```



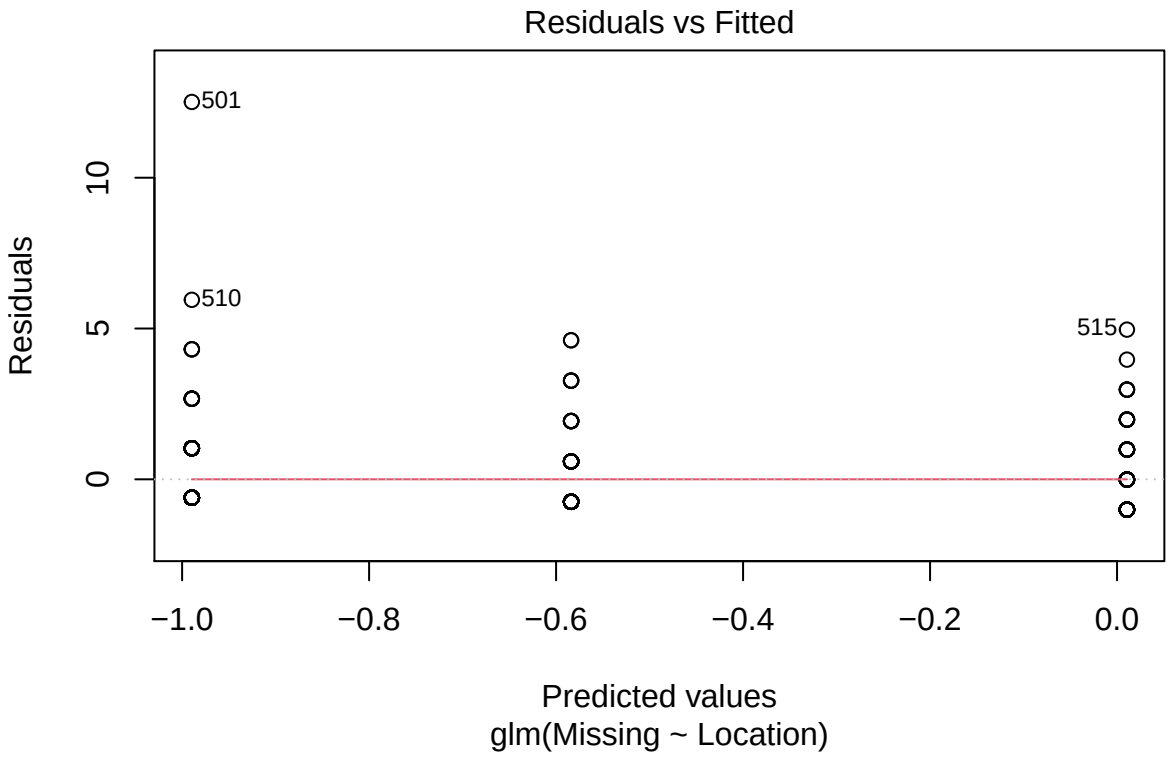

It looks like maybe groups 1 and 3 are sort of similar, and group 2 has more variance? But this could be misleading since we still can't get a great sense for how many points are plotted here... Regardless, our data are very clearly integers, given that they only take values of 1, 2, 3, 4, etc. So we definitely need a Poisson (or similar) distribution.

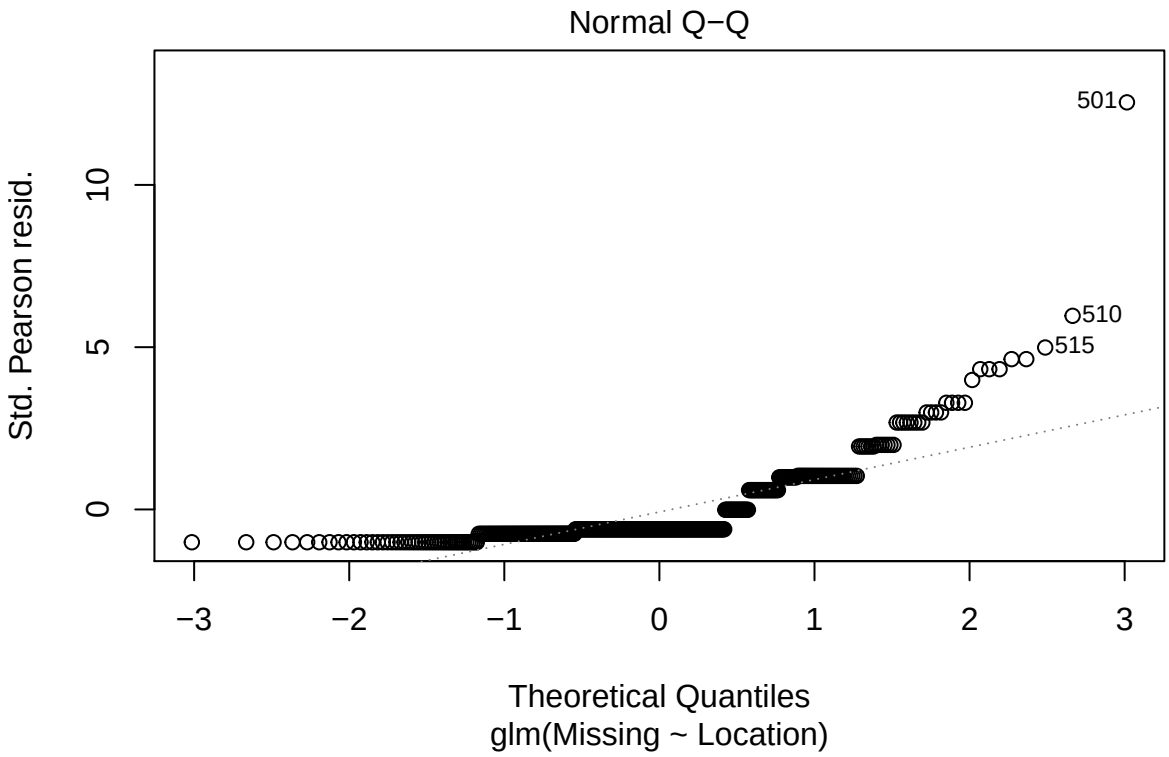
So let's build the model! We'll specify the usual arguments, but this time, `family = poisson`, because our residuals will be Poisson distributed, not normally distributed. And we also specify `link = "log"` in parentheses after `poisson` because our data are bounded by zero. Hence, we'll need to fit our model to log-transformed data.

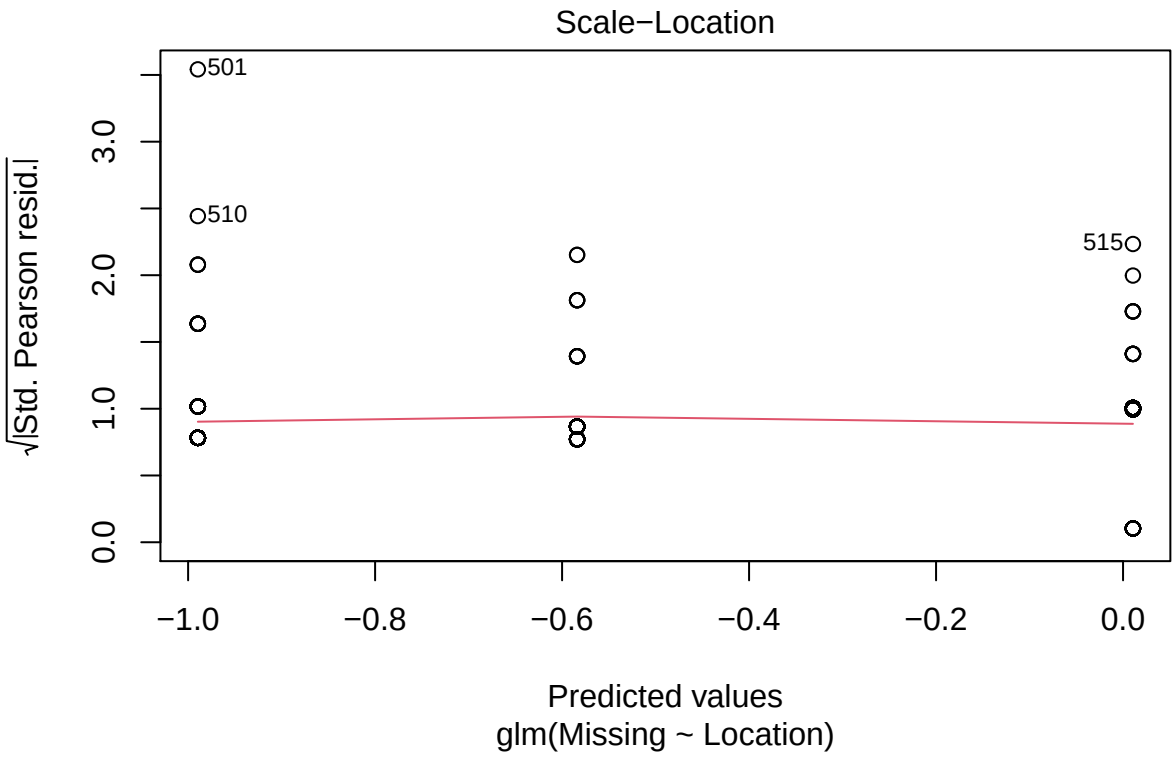
```
missing.glm <- glm(Missing ~ Location, data = gold.sub, family = poisson(link = "log"))
```

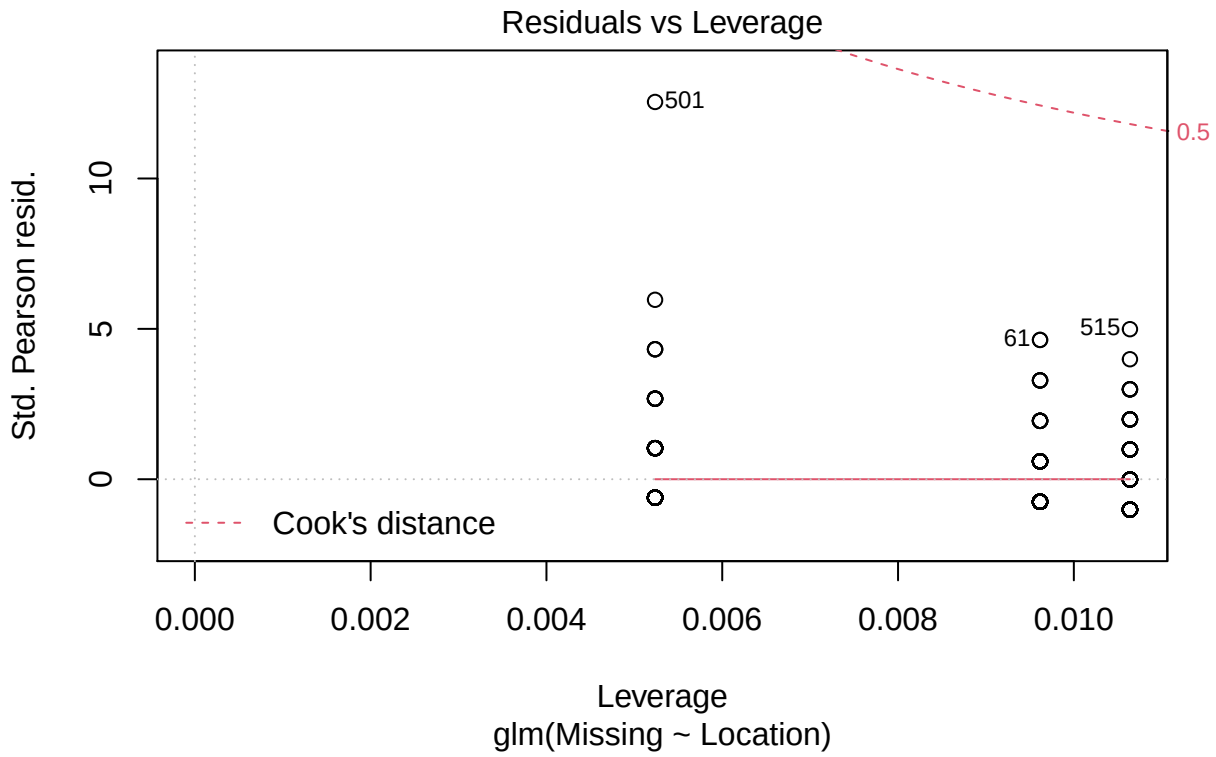
Now, we discussed above that with count data, pretty much all of our Gaussian linear model assumptions are violated by definition. So inspecting the model diagnostic plots becomes somewhat unnecessary in these cases... But let's do it anyways so 1) we can see that our data are indeed violating assumptions of normality as we said, and 2) to ensure nothing else is strange about the data/model (e.g. non-linearity or non-independence of residuals).

```
plot(missing.glm)
```


Residuals vs Fitted
501
510
515
Residuals
10
5
0
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
Predicted values
glm(Missing ~ Location)


Normal Q−Q
Std. Pearson resid.
0
5
10
501
510
515
−3
−2
−1
0
1
2
3
Theoretical Quantiles
glm(Missing ~ Location)


Scale−Location
501
510
515
√|Std. Pearson resid.|
0.0
1.0
2.0
3.0
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
Predicted values
glm(Missing ~ Location)




We can see from the first plot that our residuals are certainly not normally distributed around the model fit (as we already knew), but it's not really clear whether any other assumptions are being violated since there are many superimposed residuals. The qq plot then confirms that our residuals are not normally distributed (look at the horizontal rows of points and the sharp deviation from the dotted line). The third plot reveals only slight heteroskedasticity, which is really hardly noticeable, and the fourth plot indicates that there are no especially influential points.

### 1.3 Model Summary

Next, let's inspect the model summary.

```
summary(missing.glm)
```

```
##
## Call:
## glm(formula = Missing ~ Location, family = poisson(link = "log"),
##     data = gold.sub)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.4217  -1.0561  -0.8622   0.5322   5.8179
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -0.5839     0.1313  -4.447  8.7e-06 ***
```

```
## LocationJP    0.5945     0.1666   3.568  0.00036 ***
## LocationOH   -0.4056     0.1770  -2.292  0.02191 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 565.68  on 388  degrees of freedom
## Residual deviance: 524.66  on 386  degrees of freedom
## AIC: 837.67
##
## Number of Fisher Scoring iterations: 6
```

We can see that our model has three coefficients: an intercept, and coefficients for the JP and OH locations. This structure of summary output should look familiar because it's the same as the ANOVA output we looked at previously. R assigns one category, in this case "AK", to the intercept and compares the effects of each other group relative to that one.

But we must remember that this is a GLM, so we're dealing with coefficients on the link scale! As a Poisson family GLM is on the log link scale, that means that the intercept is not actually -0.911, but the *exponent* of -0.911. Recall that exponentiation is the inverse of logging. This fact makes sense because the intercept shouldn't be able to predict negative values, but the exponent of -0.911 is a positive number: 0.402.

So if we wanted to calculate the estimated mean counts of missing bones for each category, we could run the following calculations.

```
#mean estimate for Alaskan missing bones
exp(missing.glm$coefficients[1])
```

```
## (Intercept)
##   0.5576923
```

```
#mean estimate for Japanese missing bones
exp(missing.glm$coefficients[1] + (missing.glm$coefficients[2] * 1))
```

```
## (Intercept)
##    1.010638
```

```
#mean estimate for Ohioan missing bones
exp(missing.glm$coefficients[1] + (missing.glm$coefficients[3] * 1))
```

```
## (Intercept)
##   0.3717277
```

Or we could use the `predict()` function, feeding it our GLM object and the value of Location that we want to predict (in a named data frame, because the function require it). We also need to specify `type = "response"` because we want our result to be in real-world units, not on the link scale of our model. This option can be good if you just can't remember what the inverse link function is for your model since the function handles that un-transformation internally.

```
#estimated mean for Alaskan missing bones
predict(missing.glm, data.frame("Location" = "AK"), type = "response")
```

```
##         1
## 0.5576923
```

```
#estimated mean for Japanese missing bones
predict(missing.glm, data.frame("Location" = "JP"), type = "response")
```

```
##        1
## 1.010638
```

```
#estimated mean for Ohioan missing bones
predict(missing.glm, data.frame("Location" = "OH"), type = "response")
```

```
##         1
## 0.3717277
```

### 1.4 P-values and D2

To calculate our actual p-value for the model, remember that we need to run a likelihood ratio test comparing our model to an intercept-only version serving as a null model. We can do this using the `lrtest()` function from the `lmtest` package.

```
library(lmtest)

lrtest(missing.glm)
```

```
## Likelihood ratio test
##
## Model 1: Missing ~ Location
## Model 2: Missing ~ 1
##   #Df  LogLik Df  Chisq Pr(>Chisq)
## 1   3 -415.84
## 2   1 -436.34 -2 41.017   1.24e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Here we see that our model is indeed significant. Location significantly improves our model's ability to explain what's going on with the number of missing bones.

We can also check our $D^2$ value using the `Dsquared()` function from the `modEvA` package.

```
library(modEvA)

Dsquared(missing.glm)
```

```
## [1] 0.07250813
```

The $D^2$ value for this model is only 0.089, which is a very weak effect. We can say that including Location in our model only accounts for 9% of the total deviance, indicating that there are other variables out there needed to predict the count of missing bones per skeleton.

This means that, once again, we have a case where our model p-value is significant and our model $D^2$ value is very low. Our model is therefore identifying an effect that is unlikely, but that effect is only capable of

explaining a small portion of what's going on with missing bone numbers. This doesn't mean the model is worthless, but you should probably be a bit skeptical... If you're simply interested in significant differences between the groups, however, the low $D^2$ value might not be that important to you. The missing bone counts of the three groups being significantly different from each other is a bit of a separate question from the proportion of deviance explained.

Since we got a significant p-value, we can proceed with a post hoc test as we did with regular Gaussian ANOVA models. But in this case, we can't use pairwise t-tests because t-tests, like ANOVAs, assume that your data meet the assumptions of Gaussian linear models. So we need something different.

We can use the `glht()` function (which stands for general linear hypothesis test, but that's not particularly important) fromt he `multcomp` package for this. What we need to do is run several pairwise models, and this function/package can help us with that. Let's load the package, and then use the `glht()` function.

In this `glht()` function, we have to feed it our model object and then the `mcp()` function. This is another function from the `multcomp` package which tells the `glht`function that our `Location` variable is what we're interested in, and we want to use Tukey's method. Tukey's is an alternative to Bonferroni's method that we used previously.

```
library(multcomp)

summary(glht(missing.glm, mcp(Location = "Tukey")))
```

```
##
##   Simultaneous Tests for General Linear Hypotheses
##
## Multiple Comparisons of Means: Tukey Contrasts
##
##
## Fit: glm(formula = Missing ~ Location, family = poisson(link = "log"),
##     data = gold.sub)
##
## Linear Hypotheses:
##               Estimate Std. Error z value Pr(>|z|)
## JP - AK == 0   0.5945     0.1666   3.568  0.00108 **
## OH - AK == 0  -0.4056     0.1770  -2.292  0.05674 .
## OH - JP == 0  -1.0002     0.1569  -6.375  < 0.001 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## (Adjusted p values reported -- single-step method)
```

The output from this proces is three sets of model comparisions: JP - AK, OH - AK, and OH - JP. Each one has an associated p-value, among other things. In the case of our `missing.glm` model, all three groups are significantly different from each other, as you can see in the `Pr(>|z|)` column. We could report these p-values, and even the associated coefficient estimates (on the appropriate scale!) in our results.