Advanced Lab 3: Basic Linear Modeling and ANOVA

Introduction

In this lab we will study key linear modeling techniques. We will also practice some data cleanup and import steps.

To begin with, our data is in an SPSS file, which we can access using the haven library. You will probably want to create a new empty project first. Then you should download this data file[1], and upload it to your project directory. You can also find a link in the workshop resources page.

```
library(hanoverbase)
library(haven)
targeting <- read_sav("targeting.sav")
# View(targeting)
```

The data set contains a number of factor variables are currently coded into column names. There is there race of the target (White/Black), whether the target was armed or unarmed, and whether a correct or incorrect shot action was taken. We will need to create these factor variables as we process the columns.

Cleaning up the dataset

Let's take a look at the variables:

```
names(targeting)
```

We will only need the first 12 variables, the remaining are computed quantities. We will use a select for that. Then we will gather the 8 columns that contain observations (columns 3 through 10). Don't worry about the warning.

```
targetingLong <- targeting %>%
    select(1:12) %>%
    gather(key="key", value="time", 3:10)
```

Next we need to break the key variable into two parts, one showing the target's race and another showing the outcome. We'll first mutate the key field to remove everything up through the underscore. We will need the stringr package for that. This package allows us to perform various string-related tasks.

```
library(stringr)
targetingLong <- targeting %>%
    select(1:12) %>%
    gather(key="key", value="time", 3:10) %>%
    mutate(key=str_replace(key, "expressions.MeanRT_", ""))
```

Now we split the new key variable in two parts, splitting after the first 5 characters (to capture the White/Black part):

```
targetingLong <- targeting %>%
    select(1:12) %>%
    gather(key="condition", value="time", 3:10) %>%
```

[1] ../datasets/targeting.sav

```
    mutate(condition=str_replace(condition, "expressions.MeanRT_", "")) %>%
    separate(condition, into=c("race", "outcome"), sep=5)
```

Next we need to work on the outcome variable:

```
targetingLong %>% count(outcome)
```

This variable actually contains two different pieces of information, whether the target was armed and whether the subject took the correct action:

Hits	Target was armed, subject fired (Correct Action)
Misses	Target was armed, subject did not fire (Incorrect Action)
CRs	Target was unarmed, subject did not fire (Correct Rejection)
FAs	Target awas unarmed, subject did fire (False Alarm)

We will use mutate and recode_factor to create these new variables:

```
targetingFinal <- targetingLong %>%
    mutate(weapon=recode_factor(outcome, Hits="Armed", Misses="Armed",
                                         CRs="Unarmed", FAs="Unarmed"),
           action=recode_factor(outcome, Hits="Correct", Misses="Incorrect",
                                         CRs="Correct", FAs="Incorrect"))
```

To double-check that we did this correctly, we'll create counts:

```
targetingFinal %>%
    group_by(race, outcome, weapon, action) %>%
    summarize(count=n())
```

We should see 49 cases for each, corresponding to our initial 49 data rows.

Finally, a couple more cleanup steps are in order before we move on:

- We will fix the names of some of the variables, using rename.
- We will drop the outcome column as it is no longer needed, using select.
- We will code the the gender, race and age variables as factors, using mutate and factor for that (we would use recode_factor if we wanted to change the names of the labels, but we don't).
- We will remove the missing values from the time variable, using filter. The expression !is.na(time) picks up all those values that are *not* missing.

This can all be done in a series of pipelined steps.

```
targetingFinal <- targetingFinal %>%
    rename(subject="script.subjectid",
           iat="expressions.d",
           gender="gender_response",
           age="age_response") %>%
    select(-starts_with("outcome")) %>%
    mutate(gender=factor(gender), age=factor(age), race=factor(race)) %>%
    filter(!is.na(time))
```

Linear Modeling

Basic (constant) fit

We are looking for a linear regression model to understand the mean reaction time in terms of given inputs. Let us start with the simplest such model, often referred to as the "null model", where we would like to predict the time using no predictors at all. In that case all we can do is try to predict a single value, and then account for errors and variability around that value. Our model, as a formula, would look like this:

$$\text{time} = \beta_0 + \epsilon$$

where the β_0 is a parameter we need to choose, and ϵ is the error we are making (different error for each point). The key question to address here is how to determine the "best value" for the parameter β_0.

In linear regression, we choose the parameters so as to *minimize* the "residual sum of squares", i.e. the sum of the squared residuals:

$$\text{RSS} = \sum \epsilon_i^2$$

In our case it can be seen easily that the choice of parameter value that minimizes this sum is the mean $\beta_0 = \text{mean(y)}$. We can then use that to compute the RSS:

```
m <- targetingFinal$time %>% mean()
m
rss <- sum((targetingFinal$time - m)^2)
rss
```

So we can see there is a total variability of $924,812$ to account for. Since we will often find ourselves computing the "sum of squared deviations" by subtracting the mean from a variable, then squaring, then summing all the values, let's simplify matters by writing a small function that computes the squared deviations:

```
sq.devs <- function (x) { (x-mean(x))^2 }
rss <- targetingFinal$time %>% sq.devs() %>% sum()
```

We could get the same number using R's modeling machinery:

```
fit0 <- lm(time~1, data=targetingFinal)
summary(fit0)
deviance(fit0)     # Essentially the sum of squared deviations/residuals.
```

The 1 on the right-hand-side of the model represents that we fit a constant model.

This null model is kind of a baseline against which we can compare our other models. This is essentially the simplest possible model; any other model should be doing better by comparison.

Optional background: Maximum Likelihood Estimation There is a slightly different approach to the least squares method described above, proves to be easier to generalize to other settings. It roughly works as follows:

- We assume that the residuals are independent of each other and are all distributed identically, following a normal distribution centered at 0 and with some standard deviation σ. In that case the y values follow a normal distribution centered at β_0.

- Therefore for each data point y_i we can discuss the *likelihood/probability* that the y_i would take this value, assuming the normal distribution and for a given value of β_0.
- We can then multiply all those likelihoods together, since the observations were independent, to get an *overall likelihood*. This is basically a number determining how how likely we are to observe this set of values given some fixed values for the parameters.
- We now can choose the parameters that maximize this likelihood. This is known as the **maximum likelihood estimate**.

It turns out that for linear regression, the solution to these two problems is exactly the same. So we can think of the coefficients provided by a linear regression fit in these two slightly different ways:

- They are those parameter values that minimize the overall error phrased as an RSS.
- They are also those parameter values that maximize the likelihood of the values that we observed.

Linear fit, one scalar predictor

Now we want to examine how the mean reaction time might be affected by other predictors. We will start by considering one such predictor, the iat score.

The iat score is the *implicit-association test*[2], which measures "the strength of a person's automatic associations between mental representations of objects in memory". In this particular case, the iat was measuring the strength of association between race (black/white) and pictures of weapons.

A graph is a good start, this would be a point plot and we will add a smooth line to it.

Do this now, use ggplot to draw a scatterplot of the iat score in the x-axis and the time in the y-axis, and use geom_smooth to add a smooth line through the fit. How would you describe the influence of iat on time?

In a linear model we seek a formula that would describe in a linear way the response variable from the independent variables, accounting for a possible error. So the equation we are after would look like this:

$$\text{time} = \beta_0 + \beta_1 \times \text{iat} + \epsilon$$

The linear part, $\beta_0 + \beta_1 \times \text{iat}$, provides our *predicted value*, while the ϵ term indicates the error we are making (called the *residual*). In typical linear modeling there are numerous questions we like to ask:

1. Since we have many choices for the parameters β_i, how do we define "the best choice"?
2. How can we assess whether the structure of the model is reasonable?
3. How do we determine how volatile our coefficients are to the variability in our data?
4. How can we use the model to make predictions, and what kind of error do we expect on those predictions?
5. How can we compare our model to other models?

We essentially answered question 1 earlier. We saw there were two different ways to compute the best choice, and in the standard setting of a linear model they both result in the same estimates. Let us now construct a linear fit in R using the iat predictor:

```
fit1 <- lm(time~iat, data=targetingFinal)
summary(fit1)
```

[2] https://en.wikipedia.org/wiki/Implicit-association_test

The output of this summary view tends to contain a lot of information. For now the one key piece of information is the fit coefficients, namely $\beta_0 = 492.223$ and $\beta_1 = 10.965$. Therefore we are claiming that we have a model relationship that looks like so:

$$\text{time} = 492.223 + 10.965 \times \text{iat} + \epsilon$$

For instance, let us try to predict what the time should be when iat equals 1. We can do this either by direct computation using the above linear equation, or by using the predict function:

```
predict(fit1, data.frame(iat=1))
482.223+19.965*1
```

One of the questions we'll want to answer is how reliable this prediction is; we will return to that later.

We can get all the predicted values and all the residuals by simply doing respectively:

```
predict(fit1)
resid(fit1)
```

Let's discuss how good this model fit is. There are numerous questions we could ask. A natural first question is how well the model manages to explain the variation in time as a result of the variation in iat.

We can break the variation in time into two parts:

- Variation due to the variation in iat (or all the predictors in general if we have multiple predictors).
- Variation due to randomness, measured by the residuals.

In the null/constant model we discussed earlier, there was essentially no variation due to the predictors; it was all due to randomness.

It turns out that these two variations are sort of "orthogonal" to each other, and we have a formula that essentially says that the sum of these two variabilities equals the total variability. We can see this in R:

```
variability.time <- targetingFinal$time %>% sq.devs() %>% sum()  # Total sum of squares
residual.sum.squares <- resid(fit1) %>% sq.devs() %>% sum()  # Residual sum of squares
variability.predicted <- predict(fit1) %>% sq.devs() %>% sum()
variability.predicted
residual.sum.squares
# The following two are equal
variability.predicted + residual.sum.squares
variability.time
```

So this breaks down the variance in two parts: The explained part and the unexplained part. In our case we can see that the explained part is a small part of the whole:

$$\frac{6273.656}{924811.7} = 0.006783712 = 0.68\%$$

This is the same as the r^2 and is extremely small in this case, indicating that our model explains very little of the variability in time.

This doesn't yet mean the model is "bad": Maybe that's as much of the variability as we *can* explain from iat, and all the rest is just "noise" or depends on other predictors. So we pose the question: If a linear equation in iat doesn't help much, what is *the best we could possibly hope to do using just iat as a predictor*?

Let us see how we could try to measure this: For each value of x there is only one corresponding prediction we can make. But we may have multiple y values for the same x. Whatever the variability of those values is, there is no way that we could do better than that, by just using x, because we can only make one prediction for each x.

Now as our x is scalar, it is fairly unlikely that we would see multiple ys for the same x. We could however "fake" it by breaking x into many small pieces, and assume that our prediction is more or less constant within each of those pieces. In effect we will imagine that our regression line is stepwise with really small steps. Then we'll measure the error we make in each step. We can't really have a function that jumps a lot within one of these steps, as they are so small. Here's how we can compute something like that:

- Cut iat into small pieces and create a new variable from that.
- Group the data set based on the levels of this new variable. Each of these groups essentially amounts to constant iat. Any function using iat can't really do much better than predicting a single value on each of thes groups.
- Measure the variability in time on each of those levels.
- Add all those together.

The dplyr machinery can help us do all that:

```
targetingFinal %>%
    mutate(iat.binned=cut_width(iat, 0.01)) %>%
    group_by(iat.binned) %>%
    summarize(
        sq.devs = sq.devs(time) %>% sum(),
        count = n()
    ) %>%
    summarize(total=sum(sq.devs))
```

We get as an answer $623,859.5$. This suggests that there is a great amount of variability of time, namely $\frac{623859.5}{924811.7} = 67.5\%$ that we cannot really hope to explain with any regression model that uses iat only. Still, we could in theory have explained up to 33.5%, yet our model does considerably worse.

Another approach is to do what is known as a loess fit, which essentially fits the best possible smooth curve. We can use the loess method for that and see the size of the residuals:

```
loess(time~iat, data=targetingFinal) %>%
    resid() %>% sq.devs() %>% sum()
```

This adds up to $855,951$ which is $\frac{855951}{924611.7} = 0.9257 = 92.6\%$. This suggests that we couldn't really predict more than 8% of the time variability using iat.

Whether we can do better with iat or not, it is clear that using iat as the only predictor is a bad idea. We may return to exploring the effect of iat further later.

Other diagnostics

Let us consider for the moment other diagnostics we could perform on this data. One of the most important diagnostics is of course looking at residual plots. There are two things we look for in a residual plot: The first is homoscedasticity, namely constant variance. The other is consistent patterns, which might suggest that other predictors or higher order contributions from our predictor might be desired.

```
targetingFinal %>% ggplot() +
    aes(x=fitted(fit1), y=residuals(fit1), color=weapon) +
    geom_point() +
    geom_smooth() +
    facet_grid(action~race)
```

We see that the residuals exhibit overall a slightly quadratic behavior, but more importantly there are differences in the behavior of the residuals due to the other factors. A better picture might arise from considering those factors as part of the model, which we will do at a later time.

Once we determine that no systematic patterns can be observed in the residuals, we can examine the assumption of constant variance in relation to the fitted values. To achieve this it turns out that we can improve our precision by considering the *square root of the absolute residuals*. The idea here is that in order to consider variability the difference between positive and negative shouldn't be a factor, and by taking absolute values we essentially double our precision. However as the resulting distribution is skewed, it turns out that we can get a better view if we square these absolute value residuals, which results in an approximately normal distribution if the original distribution was normal. Here is how that might look in R:

```
targetingFinal %>% ggplot() +
    aes(x=fitted(fit1), y=residuals(fit1) %>% abs() %>% sqrt(), color=weapon) +
    geom_point() +
    facet_grid(action~race)
```

There does not appear to be any systematic change in the variance, which is good.

An important consideration is whether the residuals are normally distributed. Technically, one has to be careful how to assess this, and a slight detour to some statistical theory might be needed.

The hat matrix and its effect When we consider a linear regression model, we usually write it in a linear algebra form:

$$y = X\beta + \epsilon$$

where β is the vector of coefficients, including the constant, and X is a matrix containing all our predictors. We can inspect the matrix for our model:

```
model.matrix(fit1) %>% head(10)
```

We determine the parameter estimates $\hat{\beta}$ via a matrix multiplication whose details are not that important right now:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

And from there we can find the fitted values $\hat{y}$ by multiplying these estimates by the matrix X:

$$\hat{y} = X\hat{\beta} = X(X^T X)^{-1} X^T y$$

The end result of all this is that if we consider the matrix $H = X(X^T X)^{-1} X^T$, then we can find the fitted values from the original values by simple matrix multiplication:

$$\hat{y} = Hy$$

For that reason, this matrix is called the *hat matrix*. For our current discussion, the importance of this matrix is that we can use it to express the observed residuals in terms of the true errors:

$$\hat{\epsilon} = (I - H)\epsilon$$

where here I is the identity matrix.

The important thing to take out of this is that even though we assume that the errors in our model are uncorrelated and with constant variance, the observed residuals that result form the fit in fact do not have to be, due to the above relation. In particular, the variance of the i-th residual can be found by multiplying the variance σ^2 of the errors by $1 - h_{ii}$, where h_{ii} is the i-th entry in the diagonal of the hat matrix.

The method hatvalues returns the diagonal elements of the hat matrix, which are also called *leverages*:

```
hatvals <- hatvalues(fit1)
plot(1:length(hatvals), hatvals)
plot(targetingFinal$iat, hatvals)
```

We can see how some data points have larger leverages than others. In particular, predictor values that are more towards the extremes have larger leverages. This makes sense in a way: A large leverage value means small variance for the error. Points that are far in the x direction will tend to influence the regression line more and will tend to have smaller variance to their errors.

Leverages always add up to the number p of predictors, in our case 2. We can easily check that:

```
sum(hatvals)
```

This means that on average we expect leverages to be about equal to $\frac{p}{n} = \frac{2}{387} = 0.005168$. Values that are at least twice as much are worth looking at closer:

```
abline(h=2*2/387, col="red")
sum(hatvals >= 2*2/387)
```

In our case that's 47 observations that might be worth a closer look, as far as this model is concerned.

Standardized and Studentized residuals Due to the effect of the hat values, we should consider rescaling the residuals before considering normality. There are two standard adjustments to consider. The first is what is known as *standardized residuals*. These are simply the residuals divided by the estimate of the standard deviation and the hat matrix effect, so that they have variance 1:

$$r_i = \frac{\hat{\epsilon}_i}{\hat{sigma}\sqrt{1 - h_i}}$$

The rstandard method in R returns those residuals:

```
rstandard(fit1)
```

In order to test for normality, it is best to use these standardized residuals, though typically they won't be all that different from the raw residuals. These are also often called *internally studentized residuals*.

The other kind of residual is called *(externally) studentized residual*. A studentized residual is the residual that a point produces, but where we use a model fit that excluded that point. So in theory for each point we would: exclude it, compute the model fit, predict the value at the point and compute the residual.

Luckily there is a formula that allows us to compute this number easily. The precise formula is complicated, but the rstudent method can compute it for us:

```
rstudent(fit1)
```

A typical diagnostic plot would be a "normal quantile plot" of the standardized residuals. These plots draw the values of each percentile mapped against the corresponding theoretical values from a normal distribution. If our values are normally distributed then the resulting plot is a straight line:

```
rstandard(fit1) %>% qqnorm()
```

Prediction and Estimation When trying to use a model to make a prediction for the y value at a particular value x, there are two slightly different values that we might be trying to predict:

1. The average of all the possible y values for that particular x (i.e. a *predicted mean response*).
2. An actual possible y value for that particular x (i.e. a *prediction of a future observation*).

Even though in both cases the estimate is the same, namely the result of plugging in the x value to the formula of the estimates, the two cases differ considerably in the estimation of the standard error and consequently the construction of confidence intervals.

For the former, we simply need to account for the variability in the estimation of the parameters β. This is essentially the standard deviation σ of the residuals suitably scaled to account for the x value. The resulting intervals are called *confidence intervals*.

For the latter, we have to account for the extra variability σ due to the possible error at that particular x. The combination of the two independent variabilities is the desired variability. The resulting intervals are called *prediction intervals*.

In R, the predict method can provide us with confidence and prediction intervals:

```
predict(fit1, list(iat=-2:4), interval="confidence")
predict(fit1, list(iat=-2:4), interval="prediction")
```

Linear fit, one factor

Let us now consider a factor variable and look at its effect on the time. A factor variable can make a single prediction for each factor level, and so it may be able to do better than the initial null model, which was only making a single prediction. Let's consider the weapon variable, which refers to whether the target was armed or unarmed. We might expect faster reaction times if the target is armed. We start with a plot:

```
ggplot(targetingFinal) +
    aes(weapon, time) +
    geom_boxplot()
```

We're curious if there is a difference in the mean reaction times between the two groups, and they seem to be fairly close to each other. Let us use dplyr to compute some numerical summaries for each group:

```
targetingFinal %>%
    group_by(weapon) %>%
    summarize(mean = mean(time),
              sd   = sd(time),
              n    = n(),
              se   = sd(time)/sqrt(n))
```

We can see a slight difference in the standard deviations, and we can see the two means fairly close to each other. We can see that the standard errors within each category are somewhere in the 3-4 range, indicating that the mean difference of close to 17 is considerable.

Let's take a look at a model fit:

```
fit2 <- lm(time ~ weapon, data=targetingFinal)
summary(fit2)
coef(fit2)
```

We can see that the model output has treated the "Armed" case as a baseline, and the intercept represents the mean/predicted value for time for those subjects in the "Armed" case. The effect for the "Unarmed" case is then considered as an additive factor to that. So the predicted value for the "Armed" case would be coef(fit2)[1] = 489 and the predicted value for the "Unarmed" case would be coef(fit2)[1] + coef(fit2)[2] = 506.3. These are of course the same as the mean values we saw with dplyr.

Notice the p-value of 0.000495 which appears in two places. It is the P-value for an F test that measures if our model is better than the null model, i.e. than the case where the values for armed and unarmed were the same. Or it can be thought of as the P-value for a t test on whether the coefficient for the term weaponUnarmed is non-zero. We discuss these tests in more detail in the next section.

The F statistic

In general, if we have two models M_1 and M_2 with M_2 being the an extension of M_1, and with degrees of freedom df_1 and df_2 respectively, then we can consider the difference between the residual sums of squares of the two models scaled by the difference in the degrees of freedom, and divide that by the scaled residual for the larger model:

$$F = \frac{(\text{RSS}(M_1) - \text{RSS}(M_2))/(df_1 - df_2)}{\text{RSS}(M_2)/df_2}$$

Assuming that the larger model M_2 does not provide any improvement over the smaller model, then this number F follows an $F_{df_1-df_2,df_2}$ distribution.

As an example in our case, we have our larger model M_2 that uses weapon in addition to a constant to determine time, and we want to compare it to the null model, which uses just the constant. We have $df_2 = n - 2$ and $df_1 = n - 1$. We can directly compute the sums of squared residuals of the two models:

```
rss1 <- residuals(fit0) %>% sq.devs() %>% sum()
rss2 <- residuals(fit2) %>% sq.devs() %>% sum()
n <- nrow(targetingFinal)
df1 <- n-1
df2 <- n-2
fstat <- ((rss1-rss2)/(df1-df2)) / (rss2/df2)
fstat; df1-df2; df2
pf(fstat, df1-df2, df2, lower.tail=FALSE)
```

The last line tells R to compute the upper-tail probability for the value fstat in an F distribution with df1−df2 and df2 degrees of freedom. There are a number of functions like pf for all kinds of distributions. You can read more about them via ? Distributions .

You may be familiar with these computations under a different terminology. The denominator can be interpreted as the **within-groups variability**, while the numerator can be interpreted as the **between-groups variability**. Let's check this in our instance. We can define the between-groups variability as follows: Data points form two groups based on their weapon value. For each point we consider the difference between the mean of the point's group vs the overall mean, then we look at the sum of squares of these differences. In R this would be:

```
between.groups <- targetingFinal %>%
    mutate(totalMean=mean(time)) %>%
    group_by(weapon) %>%
    mutate(groupMean=mean(time)) %>%
    ungroup() %>%
    summarize(between.groups=sum((groupMean-totalMean)^2))
within.groups <- targetingFinal %>%
    group_by(weapon) %>%
    mutate(sqdevs = sq.devs(time)) %>%
    ungroup() %>%
    summarize(within.groups=sum(sqdevs))

between.groups; within.groups
(between.groups / (df1 - df2)) / (within.groups / df2)
```

The idea of the test is that if the model with weapon is not a considerable improvement over the model without weapon, then the between-groups variability will be small compared to the within-groups variability.

Of course, instead of doing all this by hand, the summary method for the fit does the work for us:

```
summary(fit2)
```

We can also see the same computation in the anova function, which compares two models via the method described above:

```
anova(fit0, fit2)
```

Analogous tests can be performed on the coefficients of the fit directly, using the t distribution. Testing for a coefficient equaling 0 is equivalent to an F-test where we compare the full model with the smaller model without that coefficient. In the cases we have seen so far, this coincides with the test of the full model against the null model.

Factors with multiple levels

Let us briefly discuss a case with a factor that has more than two levels. Such a factor will add one more parameter, hence one less degree of freedom. This presents an opportunity to discuss how factor levels may be coded and their various effects, and it will also be an opportunity to demonstrate the package GGally for producing some interesting plots.

We will use the iris data set, which contains measurements on the petal and sepal lengths and widths of 150 different iris plants, from three different species.

The GGally package offers us a nice visualization of the whole dataset. You will first need to install the package, via "Install" button in the Packages pane. Make sure you spell it correctly, with two capital Gs. After you have installed it, the following code should work:

```
library(GGally)
iris %>% group_by(Species) %>%
    summarize(mean=mean(Petal.Width), sd=sd(Petal.Width), n=n())
ggpairs(iris, aes(color = Species), progress=FALSE)
```

Looking at this plot, we can see that each species distinguishes itself in some way. For example the setosa irises have unusually small petal lengths and widths, while the virginica irises tend to have relatively large petal lengths and widths.

For practice, let us set up a model to fit the petal width against the species:

```
irisFit <- lm(Petal.Width~Species, data=iris)
summary(irisFit)
```

We see in this example that R has set up the setosa species as a baseline, and has introduced two additive coefficients, one for versicolor and one for virginica. So we can see that the average petal width for setosas is 0.246, while for versicolors it would be $0.246+1.08 = 1.326$, and for virginicas it would be $0.246+1.78 = 2.026$.

We can also see a very small p-value for the F statistic, meaning that the species variable definitely has a significant effect. We also notice the t-tests for the two terms against the base point of setosa.

What we see in use here is what is known as **treatment contrast**, with one baseline entry, typically representing the control group, and a 0/1 coding for each of the subsequent levels. There are many available contrast coding systems, and you can search the documentation to learn more.

We will use a package called broom to do some work with the model outputs and produce suitable plots.

```
library(broom)
tidy(irisFit, conf.int=TRUE)
glance(irisFit)
augment(irisFit)
augment(irisFit) %>% ggplot() +
    aes(x=Species, color=Species, y=.resid) +
    geom_boxplot()
```

Note that this last graph suggests a violation of the homoscedasticity assumption.

We can get pairwise comparisons between the levels by using Tukey's HSD (honest significant difference) test:

```
TukeyHSD(irisFit)
```