Advanced Lab 3: 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 which 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="condition", 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="condition", value="time", 3:10) %>%
    mutate(condition=str_replace(condition, "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 was 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 %>% count(race, outcome, weapon, action)
```

We should see 49 cases for each of 8 different combinations of values, 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(-outcome) %>%
    mutate(gender=factor(gender), age=ordered(age), race=factor(race)) %>%
    filter(!is.na(time))
```

Let us visualize this data for a second. There are of course numerous plots we could do, but here is one possibility:

```
ggplot(targetingFinal) + aes(x=action, y=time, color=race) + facet_grid(~weapon) + geom_boxpl
```

We would like to see if there is a relation between reaction time when facing white targets and reaction time when facing black targets. In order to do that, we need to do the opposite of gather, which is spread. The idea is that we could spread the time values across two columns, one for the case of white targets and one for the case of black targets.For use of analysis in the future steps, we will exclude the rows that did not have values for both white and black average response time.

```
targetingRaceDiff <- targetingFinal %>%
    spread(key=race, value=time) %>%
    filter(!is.na(White) & !is.na(Black))
targetingRaceDiff       # You can also use View
```

Let's do a simple scatterplot first:

```
ggplot(targetingRaceDiff) +
    aes(x=White, y=Black, color=action) +
    geom_point() +
    geom_smooth()
```

And a more advanced scatterplot:

```
ggplot(targetingRaceDiff) +
    aes(x=White, y=Black) +
    facet_grid(weapon~action, scales="free") +
    geom_point() +
    geom_smooth()
```

In the subsequent sections, we will consider models that describe the Black reaction time variable in terms of the other variables in targetingRaceDiff .

Linear Modeling

Basic (constant) fit

We are looking for a linear regression model to understand the mean reaction time for black targets 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 Black 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:

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

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 <- targetingRaceDiff$Black %>% mean()
m
rss <- sum((targetingRaceDiff$Black - m)^2)
rss
```

So we can see an average response time close to 500 milliseconds, and that there is a total variability of 410,973.6 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 <- targetingRaceDiff$Black %>% sq.devs() %>% sum()
```

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

```
fit0 <- lm(Black~1, data=targetingRaceDiff)
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, and it 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 Black reaction time might be affected by other predictors. We will start by considering one such predictor, the White reaction time.

A graph is a good start. We did this earlier.

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:

$$\text{Black} = \beta_0 + \beta_1 \times \text{White} + \epsilon$$

The linear part, $\beta_0 + \beta_1 \times \text{White}$, 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 White predictor:

```
fit1 <- lm(Black~White, data=targetingRaceDiff)
summary(fit1)
```

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 $\beta_0 = 235.6828$ and $\beta_1 = 0.5203$. Therefore we are claiming that we have a model relationship that looks like so:

$$\text{Black} = 235.6828 + 0.5203 \times \text{White} + \epsilon$$

For instance, let us try to predict what the Black reaction time should be when White equals 450. We can do this either by direct computation using the above linear equation, or by using the predict function:

```
predict(fit1, data.frame(White=450))
intercept <- coef(fit1)[1]      # 235.6828
slope <- coef(fit1)[2]          # 0.5203
intercept + slope * 450         # prediction at White = 450
```

In general, we can interpret the slope of 0.52 as telling us that the reaction time for Black targets changes at half the rate as the reaction time for White targets.

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

For now let us discuss how to assess how good of a fit this is. There are two questions to consider:

1. Is this model significantly better than the null model that uses no predictors?
2. Do the individual predictors that are used have a considerable contribution to the model? In this case as there is only one predictor, this is the same question as 1.

To answer question 1, a first place to look is the F-statistic and its p-value. In our case that is $F = 92.18$ on 1 and 189 degrees of freedom, with a p-value less than 2.2e−16, indicating that the overall model that includes the White variable is an improvement over the null model.

To answer question 2, we would typically look at the t-value for the coefficient, and its p-value, which in this case happens to be the same.

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

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

Perhaps the most useful diagnostic for assessing a model fit is to look at the fit's diagnostic plots. The following code automatically creates four different diagnostic plots:

```
plot(fit1)
```

The first of these plots is a typical *residual plot*, showing the residuals vs the fitted values. In this plot we typically look for two things: non-linear patterns might be an indication that we need a different form for our model (for example perhaps a quadratic term for White); also we hope to see consistent variance in the residuals across the range of the fitted values (homoskedasticity). Looking at the residual plot for our fit1 raises no concerns.

The second plot is a normal quantile plot of the standardized residuals. This helps us assess the assumption that the errors should be normally distributed (which translates to a normal distribution for the *standardized* residuals, which we will discuss more later). The normal quantile plot for our fit shows a linear pattern, which suggests that our standardized residuals are in fact normally distributed.

The third plot gives us an alternative method for assessing the assumption of homoskedasticity.

The fourth plot helps us identify influential observations, which may be unduly influencing the model fit. The x-axis is a point's *leverage*, which is a measure of how far that data point is from the rest of the data. Such points tend to pull the fit towards them, hence the term leverage. The y-axis is the point's standardized residual. We are typically interested in points that both influential and have large residuals, so the product of the two values, typically described by the so-called "Cook's distance". The red dashed lines indicate thresholds for the Cook's distance, and values beyond those thresholds warrant a closer look.

Standardized and Studentized residuals In a linear model we make an assumption that our errors are normally distributed. When trying to assess that assumption, we have at our disposal not the errors themselves but the estimates of those errors in the form of the residuals. These estimates turn out to not be exactly independent and normally distributed. But the corresponding *standardized residuals* should be normally distributed. The rstandard method in R computes these standardized 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*.

You may also run into another kind of residual, called *(externally) studentized residual*. These can be computed in R with the rstudent method.

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 additive *error* term 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 will provide us with confidence and prediction intervals:

```
predict(fit1, list(White=450), interval="confidence")
predict(fit1, list(White=450), interval="prediction")
```

As anticipated, for each x, the prediction interval at x is wider than the confidence interval for the y mean at that x.

Linear fit, one factor

Let us now consider a factor variable and look at its effect on Black. 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(targetingRaceDiff) +
    aes(x=weapon, y=Black) +
    geom_boxplot()
```

We could also do a t-test:

```
t.test(~Black|weapon, data=targetingRaceDiff)
```

This is treated as independent-samples test, which is clearly not the right thing in our case since we have matched pairs. In order to do that matched pairs, test, we would need to spread the data first:

```
targetingRaceDiff %>% select(-White) %>%
    spread(weapon, Black) %>%
    with(t.test(Armed, Unarmed, paired=TRUE))
```

We could also use dplyr to compute some numerical summaries for each group:

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

Let's take a look at a model fit:

```
fit2 <- lm(Black ~ weapon, data=targetingRaceDiff)
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 Black 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] = 485.5178 and the predicted value for the "Unarmed" case would be coef(fit2)[1] + coef(fit2)[2] = 505.4491. These are of course the same as the mean values we saw with dplyr.

Notice the p-value of 0.00284 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 (optional)

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

R's distribution functions

While most of the time you would rely on the outputs of the summary and anova functions for your p-value computations, you may on occasion find it useful to compute some of these directly. R offers us a whole bevy of distribution functions to help with that, and you can find more about them by typing ? Distributions in the console.

R knows about over 20 different distributions, and of each distribution it provides us with 4 functions:

- r ... can be used to produce random values following a specific distribution. For example 'rnorm(1000, mean=3, sd=1) would give us 1000 values drawn from a normal distribution with a mean of 3 and standard deviation of 1.
- d ... is the actual density function for the distribution, and is not particularly useful.
- p ... returns probability values for a given x value. For example pf(9.143, df1=1, df2=189, lower.tail=FALSE) is the p-value 0.00284 for an F-test with an F value of 9.143 and degrees of freedom 1 and 189.
- q ... returns the quantile for a specified probability. For example to find the 90th percentile on a t-distribution with 30 degrees of freedom, we would do qt(0.9, df=30, lower.tail=TRUE).

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.

Comparing level differences

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

```
TukeyHSD(irisFit)
TukeyHSD(irisFit) %>% plot()
```

This test is a single-step-multiple-comparison procedure that constructs confidence intervals for all pairwise differences between the factor levels, using a *studentized range distribution*. You can read more about this test in the documentation and the internet.