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

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

Factor variables:

- target race (White/Black): Coded in the variable name
- object (armed/unarmed): Coded in the variable name but implicitly
- shot action taken (correct/incorrect): Coded in the variable name but implicitly

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 start by gathering the 8 columns that contain observations (don't worry about the warning):

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

Next we need break the key variable into two parts, one showing the race and another showing the outcome. We'll first split at the underscore, and basically discard the left part:

```
targetingLong <- targeting %>%
    select(1:12) %>%
    gather(key="key", value="meanRT", 3:10) %>%
    separate(key, c("_ignore", "key"), "_")
```

Now we split the new key variable in two parts, splitting after the first 5 characters:

```
targetingLong <- targeting %>%
    select(1:12) %>%
    gather(key="key", value="meanRT", 3:10) %>%
    separate(key, c("_ignore", "key"), "_") %>%
    separate(key, c("race", "outcome"), 5) %>%
    select(-starts_with("_ignore"))
```

Next we need to work on the outcome variable, which actually contains two different pieces of information:

- whether the object was armed (Hits/Misses) vs unarmed (CRs/FAs)

- whether the subject took the correct action (Hits/CRs) or incorrect action (Misses/FAs)

We will use mutate and recode_factor to create these:

```
targetingFinal <- targetingLong %>%
    mutate(object=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, object, 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 should fix the names of some of the variables. We will use rename for that.
- We should drop the outcome column as it is no longer needed. We will use select for that.
- The gender, race and age variables need to be coded as factors. We will use mutate and factor for that (we would use recode_factor if we wanted to change the names of the labels, but we don't).
- There are some missing values in the meanRT variable. As this variable will be our focus, we will omit those values with a filter.

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(meanRT))
```

There are of course numerous graphs we can construct and analyses we can perform, and we can choose to log-transform the mean reaction time or not.

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 meanRT 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 in formulas would look like this:

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

where the β_0 is a parameter we need to choose, and ϵ is the error we are making (different 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", 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 to set the parameter to equal the mean $\beta_0 = \text{mean(y)}$. We can then use that to compute the RSS:

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

So we can see there is a total variability of 924812 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 us simplify matters by writing a small function that computes the squared deviations:

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

We could get the same number using R's modeling machinery: The `1` on the right-hand-side represents that we fit a constant model. Here we are telling

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

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

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 for each data point x_i the predicted values would also follow a normal distribution centered at $\beta_0 + \beta_1 \times x_i$.
- Therefore for each data pair (x_i, y_i) we can discuss the *likelihood/probability* that the y_i would take this value, assuming the normal distribution and for given values for β_0, β_1.
- 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 variable

Now we want to look at to what extent the iat score influences the mean reaction time meanRT. The iat score is the *implicit-association test*[1], which measures "the strength of a person's automatic associations between mental representations of objects in memory". We have other variables in our dataset and later on we will want to consider their effect. We start by looking at the relation between meanRT and iat. A graph is a good start, this would be an 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 meanRT 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 meanRT?

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 so:

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

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 *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 structural form 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 for the iat variable:

```
fit <- lm(meanRT~iat, data=targetingFinal)
summary(fit)
```

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:

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

For instance, let us try to predict what the meanRT 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(fit, data.frame(iat=1))
482.223+19.965*1
```

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

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

For now let us 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 meanRT as a result of the variation in iat.

We can break the variation in meanRT in 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:

```r
fit1 <- lm(meanRT~iat, data=targetinFinal)
variability.meanRT <- targetinFinal$meanRT %>% sq.devs() %>% sum()
residual.sum.squares <- resid(fit1) %>% sq.devs() %>% sum()
variability.predicted <- predict(fit1) %>% sq.devs() %>% sum()
variability.predicted
residual.sum.squares
# The following two are equal
variability.predicted + residual.sum.squares
variability.meanRT
```

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 such 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 meanRT.

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 variables. 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.

- Measure the variability on each of those levels.
- Add all those together.

The dplyr machinery can help us do all that:

```
targeting2 %>%
    mutate(iat.binned=cut_width(iat, 0.01)) %>%
    group_by(iat.binned) %>%
    summarize(
        sq.devs = sq.devs(meanRT) %>% sum(),
        count = n()
    ) %>%
    summarize(total=sum(sq.devs))
```

We get as an answer 623859.5. This suggests that there is a great amount of variability of meanRT, 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.

Whether we can do better with iat or not, it is clear that the linear model we have above will not do. It turns out that adding a quadratic term might be desirable, but we will return to that topic later.

Linear fit, one factor

Let us now consider a factor variable and look at its effect on the meanRT. 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 object variable, which refers to whether the "target object" was armed or unarmed. We might expect faster reaction times if the target is armed. We start with a plot:

```
ggplot(targetingFinal) +
    aes(object, meanRT) +
    geom_boxplot()
```

We're curious if there's 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(object) %>%
    summarize(mean = mean(meanRT),
              sd   = sd(meanRT),
              n    = n(),
              se   = sd(meanRT)/sqrt(n))
```

We can see a slight difference in the standard deviations, though rather small given their scales, 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.

We could use the means to make one prediction for each of the two values. Lets take a look at a model fit:

```
fit2 <- lm(meanRT ~ object, 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 its value. The “Unarmed” case is then considered as an additive factor on 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 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 term objectUnarmed is non-zero.

The F statistic

In general, if we have two models M_1 and M_2 with M_2 being the larger model, 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, divided by the scaled residual for the larger model:

$$\frac{(\mathrm{RSS}(M_1) - \mathrm{RSS}(M_2))/(df_1 - df_2)}{\mathrm{RSS}(M_2)/df_2}$$

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

As an example in our case, we have our larger model M_2 that uses object to determine meanRT, and we want to compare it to the null model, which uses just the constant. We have $df_2 = n - 2$ and $df_1 = 1$. 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.

You may be familiar with these computations under a different terminology. The demoninator 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: For each point we consider the difference between the mean of the group the point belongs to from the overall mean, then we look at the sum of squares of these differences. In R this would be:

```
targetingFinal %>%
    mutate(totalMean=mean(meanRT)) %>%
    group_by(object) %>%
    mutate(groupMean=mean(meanRT)) %>%
    ungroup() %>%
    summarize(total=sum((groupMean-totalMean)^2))
```

The idea of the test is that if the model with object is not a considerable improvement over the model without object, 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 is:

```
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 would present 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 need to install it, via the Packages pane. Make sure you spell it correctly, with two capital Gs.

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

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. We will come back to those in a minute, but first we should consider the coding of the levels of a factor variable.

Factor Codings What we see in use here is what is known as **treatment coding**, with one baseline entry, typically representing the control group, and a 0/1 coding for each of the subsequent levels. You can also use **Helmert coding**:

```
irisHelmert <- lm(Petal.Width~Species, data= iris, contrasts=list(Species="contr.helmert"))
summary(irisHelmert)
```

Here the first coefficient represents the overall mean, 1.2, while the second coefficient represents how much higher the versicolor

Care should be taken when considering these, as the p-values for the t-tests are harder to interpret.

Another alternative is **sum coding**

CLEANUP

We will leave it as is for now as the data did not show any signs of extreme skewness. Here is a starting plot that shows the density distribution for meanRT for armed and unarmed objects, and with different graphs for each race and gender combination:

```
ggplot(targetingFinal) +
    aes(x=meanRT, color=object) +
    geom_density() +
    facet_grid(race~gender)
```

We can see that mean reaction times were slower for the unarmed objects.

Let us compute some numerical summaries:

```
targetingCorrect %>%
    group_by(race, object, gender) %>%
    summarize(mean=mean(meanRT),
              se=sd(meanRT)/sqrt(n()))
```

We can also plot these:

```
ggplot(targetingCorrect) +
    aes(x=object, y=meanRT, color=race) +
    stat_summary(fun.data=mean_se, position=position_dodge(0.2)) +
    facet_wrap(~gender)
```

We probably expected the marked difference in reaction times between armed and unarmed subjects. For female subjects, the race of the subject seems to play a small factor.

```
fit1 <- lm(meanRT~race*object, data=targetingCorrect)
summary(fit1)
anova(fit1)
```

We can see a significant overall effect, but we can also see that the interaction terms are not significant. We remove them from the model:

```
fit2 <- lm(meanRT~race+object, data=targetingCorrect)
summary(fit2)
anova(fit2)
```

We can compare the two models to see if there are differences, and there is no significant difference:

```
anova(fit1, fit2)
```

We can get some default diagnostics from plotting the fit:

```
par(mfrow=c(2,2))
plot(fit2)
```

The residuals appear to be normal and with constant variance. We can visualize their effect against the other predictors:

```
ggplot(targetingCorrect) +
    aes(x=object, y=resid(fit2), color=race) +
ggplot(targetingCorrect) +
    aes(x=race, y=resid(fit2), color=object) +
    geom_point(position=position_jitter(0.1))
ggplot(targetingCorrect) +
    aes(x=race, y=resid(fit2), color=object) +
    geom_point(position=position_dodge(0.1))
ggplot(targetingCorrect) +
    aes(x=gender, y=resid(fit2), color=interaction(race, object)) +
    geom_point(position=position_dodge(0.2))
```

We can look at how iat might be related to those residuals, there's clearly a relation there:

```
ggplot(targetingCorrect) +
    aes(x=iat, y=resid(fit2), color=object) +
    geom_point() +
    geom_smooth()
```

Now let's add the subject's gender into the model:

```
fit3 <- lm(meanRT~race+object+gender+race:gender+object:gender, data=targetingCorrect)
summary(fit3)
anova(fit3)
anova(fit2, fit3)
```

We see that the subject's gender does not appear to be significant.

Finally, we look at whether we should remove race from the model as well:

```
fit4 <- lm(meanRT~object, data=targetingCorrect)
summary(fit4)
anova(fit4)
anova(fit4, fit3)
```

```
fit5 <- lm(meanRT~poly(iat, 2)+object, data=targetingCorrect)
summary(fit5)
anova(fit5)
```

ggplot(targetingCorrect) + aes(x=race, y=meanRT, color=gender) + geom_point() + geom_line(aes(group=subject))