dietall.r: Explanation of code

Goals of code:

- Log transform data values, also other transformations
- Fitting ANOVA models
  - Ploting residuals against predicted values
  - Comparing models
- “After the ANOVA”
  - Estimating group means and standard errors
  - Estimating linear combinations of means
  - All pairwise differences
  - Common multiple comparisons adjustments

Log transform data values: lines like: `diet$loglife <- log(diet$longevity)`

After you create the data frame, you can add new variables to it. Remember `diet` is the name of the data frame; the stuff after the `$` is the name of the variable in that data frame. The following two lines illustrate the square root, `sqrt()`, and reciprocal transformations.

Note: You do not need to store information in a data frame. You can have variables that are stored “as is”. The line: `loglife <- log(diet$longevity)` stores the log transformed responses in the variable loglife. If you do this, you will see another entry in the Data window (top right window in Rstudio). I recommend keeping related information together. That simplifies using variables, especially when the desired function has a `data=` option.

Calculating group-specific summaries: `tapply(diet$longevity, diet$diet, mean)` and subsequent code

The code illustrates two ways to compute group-specific summaries. One uses functions in base R; the other uses functions in the dplyr library.

1. Using `tapply()`. The `tapply()` function applies a specified function to all subsets of the data. It has three arguments: the variable of interest, the grouping variable and the function to apply. The line: `tapply(diet$longevity, diet$diet, mean)` calculates the mean (3rd argument) of the `diet$longevity` values (1st argument) grouped by each unique value of `diet$diet` (2nd argument).

It can get tedious to repeatedly name the data frame for functions (like `tapply`) that do not have a `data=` argument. `with()` provides a shortcut. `with(data frame, something)` runs the `something` function using the `data frame`. To illustrate: the line
with(diet, tapply(longevity, diet, mean)) does exactly the same thing as tapply(diet$longevity, diet$diet, mean). Note that when using with, you don’t need to specify diet$ for each variable name; longevity and diet are “looked up” inside the diet data frame.

`tapply()` accepts any of the functions defined in R. You can also add your own function, either “on the fly” or by storing the definition of a new function. The command `with(diet, tapply(longevity, diet, length))` reports the number of observations and the command `with(diet, tapply(longevity, diet, function(x){sum(!is.na(x))}))` reports the number of non-missing observations in each group.

Explanation of these two lines of code: The `length()` function tells you the number of values (the length) of a vector. So, this is the total number of rows of data for each group. This can be misleading when some of the values are missing. R uses NA as the code for a missing value. The most reliable way to count the number of non-missing observations is `sum(!is.na(variable))`. `is.na()` returns TRUE if the value is missing and FALSE if it is non-missing. `!` is the NOT logical operator, which flips TRUE to FALSE and FALSE to TRUE. When used in an arithmetic computation, TRUE is 1 and FALSE is 0, so `sum(!is.na(variable))` is the number of non-missing values in the specified vector. We can use the function “on the fly” inside `tapply()` by providing the function definition: `function(x){sum(!is.na(x))}` as the function to apply to each subset.

A few lines further down, the line:

```r
se <- function(x) {sd(x, na.rm=T)/sqrt(sum(!is.na(x)))}
```

defines a new function and stores it as `se`. This calculates the sd of the non-missing values (that’s what `na.rm=T` does) and divides by the square root of the number of non-missing observations. That defines the se. The next line uses that `se()` function.

R, in general, makes no distinction between a user defined function and a system defined function. Just be careful with function names. If you define your own `mean()` function, R will use that instead of the system function. For example, you can specify `mean <- function(x){3}`, which returns the value 3 every time the `mean()` function is used. If you do this, `mean(diet$longevity)` will be 3, no matter what values are in `diet$longevity`. That’s because your mean function is used in preference to the system function. If you try this out, you can delete your mean function by typing `rm(mean)`. R will now return to using the system `mean()` function.

2. using functions in the dplyr library

We previously used `group_by()` and `summarize()` to calculate and store averages for each experimental unit. We can also use them to calculate and display summary statistics. The approach is the same: use `group_by()` to define the grouping variables, then use `summarize()` to compute the desired summaries. As before, the first argument to `group_by()` is the data frame name; the second is the name of the variable with the groups. The result is saved in a new variable. Although you could overwrite the old data frame, I find it less confusing to store the grouped information in a new data frame (actually data-frame like object). The first argument to `summarize()` is the name of the grouped data. The rest of the arguments are sets of `name = function()` that specify what you want calculated and what to call the result. Before we only computed the mean; you can compute any quantity, including something calculated by a user defined function. The first `summarize` call computes the mean, sample
size, and sd for each group. The second computes the se in addition and illustrates that you can choose the name for each result.

Note: the mean(), sd(), and se() functions need to be told which variable to summarize. That’s crucial when you have multiple numeric variables in a data frame. The n() function does not have a variable name, because n() gives you the total number of observations in the group, which does not depend on the variable. I believe n() will include missing values, but it has been a while since I last checked.

Factor variables: diet$diet.f <- factor(diet$diet)

The R model fitting functions care a lot about the distinction between a continuous variable and a factor variable. A continuous variable is used to define a regression (coming in a few weeks); a factor variable defines groups. The t.test() function didn’t care, because t.test() only compared means of two groups. Most modeling functions really care about the distinction because a model using a continuous variable is different from a model using a factor variable.

My practice is to be very clear whether I am treating a variable as continuous or as a factor. I do that by specifically creating the factor version of a variable whenever I need a factor. The factor() function creates the factor version of any variable. So, diet$diet.f <- factor(diet$diet) creates a new variable diet.f that is the factor version of the diet variable. My practice is to create a new variable and give a name that reminds me of the original variable (diet) and that it is a factor (.f). You can use any variable name you like.

Aside: This practice of explicitly creating a factor variable is why I use as.is=T in the read.csv() and read.table() functions. If you don’t specify as.is=T, by default R will convert character values to a factor variable and leave numerical values as a continuous variable. That’s probably what you want for a variable with values like “N/N85”, “NP” and “lopro”. If you omit as.is=T, you don’t need the command to explicitly create the factor variable. However, some of our data sets have numbers for the grouping variable. A recent example is the bee type (queen or worker) in the bee visit duration data set, which was coded as 1 or 2. When you use read.csv() or read.table(), that variable will be left as a number. To use bee type as a grouping variable in most modeling functions, you need to first create the factor version. If you don’t, you’re fitting a very different model. I find it less prone to mistakes to explicitly create the factor version any time I want a grouping variable. That way both numbers or character strings are converted to factors. I don’t have to remember whether the factor step is necessary or not.

Fitting a separate means ANOVA model: diet.lm <- lm(longevity ~ diet.f, data=diet)

The lm() function fits a regression or an ANOVA model. When the X variable is a factor, it fits an ANOVA model. The name to the left of the ~ is the response variable. The piece to the right specifies the model to be fit. This example fits a model with a different mean for each diet. The data= argument specifies the data set in which to “look up” the variable names.

This command fits the model and stores the results in the variable diet.lm. Most interesting results are obtained by using other functions to extract or calculate interesting things from the fit.
anova() calculates the ANOVA table, the SSE, and dfE for the fitted model.

**Fitting a separate means ANOVA model:**

diet.lm0 <- lm(longevity ~ +1, data=diet)

Same syntax as before, except that the model (right-hand side) is only an intercept (single mean for all observations). The +1 is necessary. Again, anova() gives the SSE and dfE for that model.

**Plotting residuals and predicted values:**

plot(predict(diet.lm), resid(diet.lm))

The predict() and resid() functions extract predicted values and residuals from the specified fitted model. Push those into a plot and you have the plot of y = residuals against x=predicted values that we will use a a major diagnostic tool.

**After the ANOVA:**

library lsmeans and succeeding code

I will argue in lecture that fitting an ANOVA model and calculating the F statistic is really just the start of a data analysis. All of what I call “After the ANOVA” is specified by using functions that do additional calculations from the fitted lm() model. anova(), resid(), and predict() are three of those functions. There are many others, but most require some understanding of linear model theory to understand the output.

The lsmeans library provides functions that provide easily understood results that are statistically appropriate. Before you can do anything useful, you have to create what lsmeans calls a reference grid. That is done by the ref.grid() function. The subsequent lines of code, using lsmeans(), pairs(), or contrast(), are examples of what you can do. None of the statements after the ref.grid() statement depend on any other statement, so each can be used in isolation or combination.

**Create the reference grid:**

diet.rg <- ref.grid(diet.lm)

The ref.grid() function creates the reference grid from a fitted lm() object. This is stored in a new variable. I call that variable diet.rg to remind me that it deals with the diet data and is a reference grid.

**Means, se’s and confidence intervals for each group:**

lsmeans(diet.rg, 'diet.f')

The arguments are the name of the reference grid and the name of the factor to compute lsmeans for. The factor name needs to be in quotes, because it is a character string. In this model, there is only one factor variable, so it would seem that 'diet.f' is unnecessary. Strictly, yes, but lsmeans() needs it anyway.

If you get the error: argument "specs" is missing, with no default, you forgot to name the factor variable.

If you get the error: No variable named diet in the reference grid, you used the wrong name in the lsmeans() call. You have to use the name of a variable used on the right-hand side of the original lm() call. Here that is diet.f, not diet.

The output from lsmeans() is a table with the group name, the estimated average, the standard error, the error df, and the 95% confidence interval.
Note: the standard errors and hence the confidence intervals are calculated from the pooled sd. Hence, the t quantile used to compute the confidence interval is the error df (pooled over all groups). Both are desired things when the equal variance assumption is reasonable.

To get something other than 95% confidence intervals, you need two steps: store the lsmeans output, then use the confint() function with the levels= argument to specify the desired level. It would be nice if lsmeans accepted level= (and that may happen someday).

**Comparison of all pairs of groups:** `pairs(diet.rg)`

The `pairs()` function computes all pairwise differences. The output is a table with the estimated difference, the se of that difference (computed from the pooled sd), the error df, the t-statistic testing H0: difference = 0, and the p-value for that test. Confidence intervals can be obtained by storing the result from `pairs()` and passing that into `confint()`.

By default, `pairs()` uses a Tukey adjustment for multiple comparisons. The `adjust=` argument changes that. Many different adjustments are available. The code illustrates the four discussed in the text: no adjustment, Bonferroni, Tukey, and Scheffe.

**Specific linear contrasts of means:** `contrast()`

Any specified comparison of means can be computed using the `contrast()` function. The `pairs()` function is actually a front-end to the `contrast()` function that simplifies getting a common set of contrasts.

If you want to specify a comparison other than all pairs of differences, you need to provide the coefficients for your comparison. Lecture will discuss constructing these. One we will discuss is the comparison between the low protein diet (lopro) and the average of the other 5 diets. The coefficients of that comparison are 1 for the lopro group and -1/5 for the other five groups. To use `contrast`, you **must** know the order of the groups, so that you can put the 1 “in the right place”.

You can see the order of the groups in at least four different ways:
1) Look at the order of the groups in the lsmeans() output.
2) Print the reference grid.
3) Look at the sorted order of unique diet values: `sort(unique(diet))`
4) Look at the sorted order of the unique diet factor values `sort(unique(diet.f))`

In all cases, lopro is the first group.

The coefficients are specified as a vector, created using `c()` with commas between the elements. So you can see the pieces in action separately, the code piece `c(1, -1/5, -1/5, -1/5, -1/5, -1/5)` prints the vector of values created by `c()`.

The contrasts are obtained using `contrast()`. The first argument is the reference grid. The second is a list of named vectors, where each vector gives your desired coefficients. `loproRest` is my name for the contrast that compares lopro to the average of the other five groups. You can use whatever name you like; that name is printed in the output. You can use spaces in the name by putting the name in quotes. If you only have one comparison, the second argument is something like
list( loproRest = c(1, -1/5,-1/5,-1/5,-1/5,-1/5) ). The second argument must be a list, so even when you only have one contrast, it must be inside list( ).

When you have more than one contrast, you can specify each in a separate call to contrast(), or provide multiple elements to that list, with commas between each piece, as illustrated in the code. Notice the comma at the end of loproRest = c(1, -1/5,-1/5,-1/5,-1/5,-1/5),. That comma separates the first contrast (loproRest) from the second (N/R - R/R)