dietall.r: Explanation of code

Goals of code:

- Computing summary statistics by group using summarize
- 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

This code requires functions in two packages: dplyr (for data summaries) and emmeans (for after the ANOVA). You will have to download them and install them once (that's why the code is commented out). You will have to enable the library, using library(), each time you start R.

Calculating group-specific summaries using pipes with dplyr functions:
We have previously used `tapply()` in base R and `group_by` and summarize in dplyr to compute group-specific summaries. A very convenient way to use dplyr is to pipe output from one function into another. The pipe operator is `%>%`. The output from one function is used as the first argument of the next function. Usually (as in the first bit of dplyr code), the first argument of summarize() is the name of a grouped object. In the piped version:
`group_by(diet, trt) %>% summarize(mean=mean(lifetime), sd=sd(lifetime), se= se(lifetime)`
takes the result from `group_by` and uses it in summarize. You just have to write the rest of the summarize() call.

Piped code is often much easier to read. It produces the same result. You're free to use whichever approach (tapply(), dplyr, or piped dplyr) works best for you.

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 # 1: This practice of explicitly creating a factor variable is why my older code has as.is=T in the read.csv() and read.table() functions. Prior to Version 4 of R, by default R would convert character values to a factor variable and leave numerical values as a continuous variable. In version 4 and higher (e.g. 4.1.3), R will not convert anything to a factor variable.

Aside # 2: The distinction between a numeric variable and a factor variable **really matters** for most R modeling functions. The distinction between a character variable and a factor variable doesn't matter for most R functions. So, you can be lazy (most of the time) and just use character variables to fit ANOVA models. If you do that, make absolutely sure you remember to convert any numeric variable that is supposed to define groups (i.e., be treated as a factor variable) to a factor variable. If you aren't sure what's which type, the best policy is to explicitly define as a factor variable anything intended to define groups.

Fitting an 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 $\sim$ 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(diet.lm)` calculates the ANOVA table, the SSE, and dfE for the fitted model. The results include the SS, MS, F and p-value for the test of the null hypothesis of equal means.

Explicitly indicating the models to compare: `anova(diet.lm0, diet.lm)`
Sometimes you want to explicitly indicate the pair of models to compare. In 587, this is relevant only if you want to understand what R usually does by default (e.g., by `anova(diet.lm)`). The first model is the reduced model; the second is the full model. You see the SSE and df error for the two models, and the details of the comparison, including the F statistic and p-value.

Fitting an equal means ANOVA model: `diet.lm0 <- lm(longevity ~ 1, data=diet)`
If you want to explicitly compare the equal means and separate means models, you need to fit the equal means model. Same syntax as before, except that the model (right-hand side) is only an intercept (single mean for all observations). THe `1` is necessary on the right-hand side. R does not let you type `~ ,`, i.e. leave the left-hand side blank. The 1 specifies an intercept. Again, `anova(diet.lm0)` would give you 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 use as a major diagnostic tool.

After the ANOVA: `library(emmeans)` and code below "calculating means"
I suggested 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 emmeans library provides functions that provide easily understood results that are statistically appropriate. emmeans is the replacement for the lsmeans library, so if you see code referring to lsmeans, it is conceptually doing the same thing as what emmeans will do. emmeans is being developed; lsmeans is now deprecated.

My practice is to put all my `library()` statements at the top of the code file so I remember what libraries will be used.

Before you can do anything useful, you have to fit a model and create an emmeans version of that model. We've already fit a model (stored in diet.lm). Creating the emmeans version is done by the `emmeans()` function. The subsequent lines of code are examples of what you can do. None of the statements after the `emmeans()` statement depend on any other statement, so each can be used in isolation or combination.

Create the emmeans object: `diet.emm <- emmeans(diet.lm, 'trt.f')`
The `emmeans()` function creates the emmeans object from a fitted lm() object. This is stored in a new variable. I call that variable diet.emm to remind me that it deals with the diet data and is a emmeans object.

The first argument is the fitted lm object; the second is the variable we want to work with. The variable name is in quotes and must be one of the variables in the fitted model. For the diet analysis, the variable is trt.f.

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 emmeans() call. You have to use the name of a variable used on the right-hand side of the original lm() call. Here that is trt.f, not trt, and not diet.

Means, se's and confidence intervals for each group: `diet.emm`
Printing the emmeans object or using summary() on that object produces a table with estimated means, standard errors and confidence intervals for each mean.

The output from emmeans() 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 based on 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, use summary(, level=0.90) to specify the coverage you want.

Determining the order of the groups: You can see the order of the groups in at least three, perhaps five, different ways:
1) Look at the order of the groups along the X axis of the emmip() plot.
2) Print the emmeans object. `diet.emm`
3) Print out the levels of the treatment factor: `levels(diet$trt.f)`
The above 3 ways always work correctly.

Unless you explicitly used something other than the default ordering of factor levels (587 code never does this), you can also sort the unique values of the factor or the original variable:
4) Look at the sorted order of unique treatment values: `sort(unique(diet$trt))`
5) Look at the sorted order of the unique diet factor values.`sort(unique(diet$trt.f))`
In all cases, lopro is the first group.

Specific linear contrasts of means: `contrast()`
Any specified comparison of means can be computed using the contrast() function. You need to provide the coefficients for your comparison. Lecture has discussed constructing these.

One we didn't 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".

The coefficients are specified as a vector, created using `c()` with commas between the elements. You can write the coefficients as fractions, e.g., -1/5, or decimals, e.g. -0.2. I recommend fractions so you don't have to worry about roundoff of 1/3, which is not 0.33 and is not 0.33333. 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 emmeans. The second is a list of named vectors, where each vector gives your desired coefficients. `lopro` 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( lopro = 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 `lopro = c(1, -1/5,-1/5,-1/5,-1/5,-1/5),`. That comma separates the first contrast (lopro) from the second ('N/R - R/R'). If you want multiplicity adjustments, the family size is derived from the number of contrasts in the list you provide.

The default adjustment for all pairs, i.e., using pairs(), is tukey. The default adjustment for contrast(), is none.

Linear trend coefficients: `linear = c( 0, 26.66, -18.33, -8.33, 0, 0)`
Most of the contrasts in diet.c2 should be obvious. The linear one requires some explanation. We want to see if there is a difference between the N/N85, N/R50, and N/R40 diets focusing on a linear trend in the kcal amount. The kcal amounts are 85, 50, and 40. Their average is 58.333, so the linear trend coefficients are 85 - 58.33 = 26.66, 50 - 58.33 = -8.33 and 40 - 58.33 = -18.33. The order of groups (from printing the emmeans object) is lopro, N/N85, N/R40, N/R50, NP, R/R50. So the coefficients are 0 (for lopro) 26.66 (for N/N85), -18.33 (for N/R40) and -8.33 (for N/R50) and 0 for the other two groups.

Comparison of all pairs of groups: `pairs(diet.emm)`
The pairs() function computes all pairwise differences. pairs() is actually a front-end to the contrast() function that simplifies getting a common set of contrasts. `pairs()` simply creates contrast coefficients for you then runs contrast(). It's just an easy way to specific all pairs of groups. That means everything you did with the output from pairs (adjustment, confidence intervals, ...) can be done to the result from contrast. You can also use pipes to pass information from contrast() to summary().

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 adding `, infer=c(T,F)` to the pairs() call or by passing the pairs result into confint().

By default, pairs uses a Tukey adjustment for multiple comparisons. The `adjust=` argument changes that. Many different adjustments are available. Names of the ones we have discussed are: tukey, bonferroni, none, scheffe, sidak, dunnettx.

We have discussed multiple comparisons adjustments for tests. There is an analogous adjustment for confidence intervals, called simultaneous confidence intervals. pairs() is smart. If you ask for confidence intervals for a set of pairwise comparisons, you get the simultaneous intervals. The text at the bottom of the table tells you want has been adjusted and by what method.