

dietall2.r: Explanation of code

Goals of code:

- Fitting ANOVA models
- “After the ANOVA”
  - estimating group means and standard errors
  - estimating linear combinations of means

This code requires functions in an add-on package, `emmeans` (for after the ANOVA). You will have to download it and install it once (that’s why the `install.packages()` code is commented out). You will have to enable the library, using `library()`, each time you start R.

**Installing a new package:** `install.packages('emmeans')`

This will link to one of the many mirror sites for CRAN, the Comprehensive R Archive Network. This is the repository for the over 10,000 add-on packages developed and contributed for public use.

Notes:

The package name has to be spelled and capitalized correctly

The package name is in quotes

Installing the package only has to be done once. If you update your R version, you may need newer versions of packages, so you will have to repeat the `install.packages()`.

**Linking to a package:** `library(emmeans)`

This loads the package into your working environment. You need to do this every time you start R (but you don’t have to do it if you’re not going to use any functions from the package). If you ever get a error message about an unknown function, it probably is because you forgot to load a package, so R doesn’t know about any of the functions in that package.

**Factor variables:** `diet$trt.f <- as.factor(diet$trt)`

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 `as.factor()` function creates the factor version of any variable. The `factor()` is related and usually does the same thing. So, `diet$trt.f <- factor(diet$trt)` creates a new variable `trt.f` that is the factor version of the `trt` variable. My practice is to create a new variable and give a name that reminds me of the original variable (`trt`) and that it is a factor (`.f`). You can use any variable name you like.

Aside: This practice of explicitly creating a factor variable avoids an unfortunate and almost untraceable error. When you use a character variable (not converted to a formula) in an `lm()` model,

R will assume it was meant to be a factor. But, when you use a numeric variable, R will treat it as numeric and fit a regression model, even when you really wanted to fit an ANOVA model. This problem is avoided by explicitly converting a variable to a factor when it intended to be used as a factor.

**Fitting an ANOVA model:** `diet.lm <- lm(longevity ~ trt.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. This computes what are called type I, aka sequential, tests. `joint_tests()` in the `emmeans` library computes type III, aka partial tests. General practice in the US is to prefer type III tests. For a one-way ANOVA, there is no difference. We will discuss these issues in 571 later in the semester.

**After the ANOVA:** The `emmeans` library

I argued in 587 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()` is one 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 actively maintained; `lsmeans` is now deprecated.

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 trt 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 `trt.f`, not `trt`.

### 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 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.

### 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 has discussed 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 three ways:

- 1) Look at the order of the groups in the `lsmeans()` output.
- 2) Print the `emmeans` object.
- 3) Print the levels of the factor object: `levels(trt.f)`

Looking at the order that R “sees” is especially important when you have a mix of upper and lower case names.

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 `emmeans` object. 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( )`.

You can save the contrast output either to print it or to use it for subsequent computations. `summary()` applied to contrast output provides additional information about the contrast. Adding `infer=` to the summary adds a confidence interval and/or hypothesis test. `infer =` has a vector of 2 logical values. The first is whether to provide confidence intervals, the second a hypothesis test. So `infer = c(T,T)` gives both.

## Other things you can do - not needed for 571, but might be useful

### Comparison of all pairs of groups: `pairs(diet.emm)`

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  $H_0$ : difference = 0, and the p-value for that test. Confidence intervals can be obtained by storing the result from `pairs()` and passing that into `summary( , infer=c(T,F))` or 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`, `dunnett`. All of those names go in quotes.

You can save the pairwise summary information. This allows you to perform different adjustments, e.g., `summary( , adjust='none')`; or extract additional information, e.g., `confint( )`; for each contrast.

`pairs()` is actually creating contrasts for you. It's just an easy way to specify 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()`.

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.