FES 524 Winter 2022 Lab 2

Contents

A separate means model with R

In lab 2, you will learn how to fit a separate means model in R, check your assumptions, and answer research questions about specific group differences that are of practical interest. This is a review lab and the statistical material should already be familiar to you.

Reading in a dataset from Excel

Our dataset this week is in an Excel workbook. This was done so you can see an example of how to read Excel worksheets into R.

There are multiple add-on packages that were specifically designed to work with data stored in Excel. Today we will be working with package **readxl**, as I find it easy to use and doesn't need Java to work correctly.

Let's load package **readxl**.

```
library(readxl)
```

We are going to read in our example dataset, named `lab2.example.data.xlsx`, using the `read_excel()` function. Remember to set your working directory to wherever that data file is located.

Once the **readxl** package is loaded we can look at the help page for `read_excel()` to see what function arguments are available to us.

```
?read_excel
```

The function `read_excel()` works a lot like `read.table()` from last week. However, notice that the default for the `col_names` argument (which is the equivalent of last week's `header` argument) is `TRUE`. This means we won't have to write this out in our code if the first row of the dataset contains the column names.

The main difference when working the Excel workbooks compared to text or comma-delimited files is that we need to tell R which worksheet we want to read in when we have more than one. We can define the worksheet using either the name or the index number of the sheet we want to read as the `sheet` argument.

Today we will define the name of the worksheet, `TREEDBH`. Remember that R is case sensitive; notice the name of this worksheet is written in all capital letters.

```
treedbh = read_excel(path = "lab2.example.data.xlsx",
                     sheet = "TREEDBH")
```

Once the dataset is read in, look at the structure of the dataset in the Environment pane.

```
tibble [315 x 4] (S3: tbl_df/tbl/data.frame)
 $ tpa     : num [1:315] 100 100 100 100 100 100 100 100 100 100 ...
 $ stand   : num [1:315] 1 1 1 1 1 1 1 1 1 1 ...
 $ tree    : num [1:315] 1 2 3 4 5 6 7 8 9 10 ...
 $ deltadbh: num [1:315] 4.7 5 4.8 4.9 4.7 5.6 4.8 4.8 4.7 5 ...
```

Looking at the first six line of the dataset can also be helpful for getting a sense of what it looks like.

```
head(treedbh) # look at first 6 lines of the dataset
```

```
# A tibble: 6 x 4
    tpa stand  tree deltadbh
  <dbl> <dbl> <dbl>    <dbl>
1   100     1     1      4.7
2   100     1     2      5
3   100     1     3      4.8
4   100     1     4      4.9
5   100     1     5      4.7
6   100     1     6      5.6
```

Summarizing over a grouping variable

The dataset we're using contains information on the change in diameter at breast height (`dbh`) for individual trees within stands. The sampling unit (i.e., *replicate*) in the study, however, is *stands*. Currently we have multiple observations per stand, and it is likely safe to assume that the change in dbh measurements of trees within a stand are not independent of each other.

We could do an analysis that accounts for the *subsampling* of stands in the study design. However, our research questions today involves only a stand-level variable, amount of thinning. Different levels of thinning were assigned to stands (not individual trees). This means we can simplify our analysis by averaging the change in dbh for all trees within a stand. This will leave us with a single stand-level value we can use as the response variable in an analysis using statistical tools you learned in your introductory statistics classes.

There are many functions in R that allow us to summarize by groups. In this class, we'll be using the `group_by()` and `summarise()` functions from the **dplyr** package.

```
library(dplyr)
```

We'll start with the `group_by()` function. We will use the function on our dataset, `treedbh`. It allows us to define which variable or variables represent the groups we want to calculate summary statistics for. See the documentation for what the syntax looks like.

```
?group_by
```

Here is an example of using `group_by()`. In our specific case we have two grouping variables: `tpa` and `stand`. When we have multiple grouping variables we add them to the function by name with commas between them.

Notice that the resulting dataset contains the same data, but there is new information at the top indicating the groups of interest that we've defined.

```
group_by(treedbh, tpa, stand)
```

```
# A tibble: 315 x 4
# Groups:   tpa, stand [21]
     tpa stand  tree deltadbh
   <dbl> <dbl> <dbl>    <dbl>
 1   100     1     1      4.7
 2   100     1     2      5
 3   100     1     3      4.8
 4   100     1     4      4.9
 5   100     1     5      4.7
 6   100     1     6      5.6
 7   100     1     7      4.8
 8   100     1     8      4.8
 9   100     1     9      4.7
10   100     1    10      5
# ... with 305 more rows
```

Once we've set the grouping variables, we can use the *grouped* dataset within other functions in **dplyr** to add variables by group (using `mutate()` or `transmute()`) or summarize variables by group (using `summarise()`). We will be using the `summarise()` function today. `group_by()` is always used in conjunction with another function; it doesn't do anything for us on its own.

```
?summarise
```

We have to give `summarise()` the grouped dataset, so we'll either have to make a grouped object or *nest* the `group_by()` function call inside `summarise()`. In class we will be doing the latter. Users of **dplyr** most often also use the pipe function `%>%`, which is shown in the help documents. We won't take the time to learn to pipe in **dplyr** today but we might talk about it later in the quarter.

Here's what this will look like with the `treedbh` dataset. We will average the `deltadbh` variable to the stand level, and will name the resulting stand-level dataset `standdbh`. Notice how we name the new column directly inside `summarise()`. We are naming the new stand-level value `sitegrowth`. This is the mean of the variable `deltadbh` for all trees measured in each stand.

There were only 21 stands in the study, so our new dataset should only have 21 rows. Paying attention to the expected and actual number of rows when summarizing a dataset is one way to check that we successfully did what we were trying to do.

Note: if you wanted a total instead of an average you could use `sum()` instead of `mean()`.

```
( standdbh = summarise( group_by(treedbh, tpa, stand),
                        sitegrowth = mean(deltadbh) ) )
```

```
`summarise()` has grouped output by 'tpa'. You can override using the `.groups` argument.
```

```
# A tibble: 21 x 3
# Groups:   tpa [3]
     tpa stand sitegrowth
   <dbl> <dbl>      <dbl>
 1   100     1       4.94
 2   100     2       3.77
 3   100     3       3.88
 4   100     4       3.88
 5   100     5       4.79
 6   100     6       4.59
 7   100     7       4.27
 8   225     8       4.59
 9   225     9       4.09
10   225    10       4.15
# ... with 11 more rows
```

We can see that this dataset is still "grouped" by the `tpa` variable (note the message). We'll *ungroup* it by adding in the `.groups` argument and dropping groups with "drop". Dropping groups isn't always necessary, but it is best practice to avoid any issues caused by grouping at some later step.

```
( standdbh = summarise( group_by(treedbh, tpa, stand),
                        sitegrowth = mean(deltadbh),
                        .groups = "drop") )
```

```
# A tibble: 21 x 3
     tpa stand sitegrowth
   <dbl> <dbl>      <dbl>
 1   100     1       4.94
 2   100     2       3.77
 3   100     3       3.88
 4   100     4       3.88
 5   100     5       4.79
 6   100     6       4.59
 7   100     7       4.27
 8   225     8       4.59
 9   225     9       4.09
10   225    10       4.15
# ... with 11 more rows
```

Let's take a look at the site-level dataset `standdbh`.

Check the structure in your Environment pane.

```
tibble [21 x 3] (S3: tbl_df/tbl/data.frame)
 $ tpa       : num [1:21] 100 100 100 100 100 100 100 225 225 225 ...
 $ stand     : num [1:21] 1 2 3 4 5 6 7 8 9 10 ...
 $ sitegrowth: num [1:21] 4.94 3.77 3.88 3.88 4.79 ...
```

And take a look at a `summary()` of each variable.

```
summary(standdbh)
```

```
      tpa            stand      sitegrowth
 Min.   :100.0   Min.   : 1   Min.   :3.093
 1st Qu.:100.0   1st Qu.: 6   1st Qu.:3.880
 Median :225.0   Median :11   Median :4.067
 Mean   :216.7   Mean   :11   Mean   :4.137
 3rd Qu.:325.0   3rd Qu.:16   3rd Qu.:4.467
 Max.   :325.0   Max.   :21   Max.   :4.940
```

Changing a variable to factor

We can see that the variable `tpa` is numeric (and stands for trees per acre). However, `tpa` is really our categorical variable of interest and should be a *factor* when we do our analysis, not numeric. We can encode this variable as a factor as well as change the names of each group level using the function `factor()`.

```
?factor
```

By default, `factor` converts a variable to a factor and uses all unique values in the dataset as the categories (or *levels*) of the categorical variable. The order of the levels are set alphanumerically by default, so `1` comes before `2` and `a` comes before `b`.

Here is what it looks like to use `factor()` on `tpa` using all the argument defaults.

```
factor(standdbh$tpa)
```

```
 [1] 100 100 100 100 100 100 100 225 225 225 225 225 225 225 325 325 325 325 325 325 325
Levels: 100 225 325
```

Controlling factor level order

The order of the levels can be important in an analysis, and changing the order of levels can be useful. For example, the control group in this study is the *light* thinning group, which has 325 trees per acre after thinning. It makes sense for us to have the light thinning group as our reference group because our research question is about each thinning group compared to

the control. However, because `factor()` sets the order of levels alphanumerically the heavy thinning group category, `100`, comes first by default.

We can control the order of the factor levels using the `levels` argument to put the categories into the order that we want. Notice in the output that the order of the dataset hasn't changed, just the order of the levels.

```
factor(standdbh$tpa, levels = c(325, 225, 100) )
```

```
 [1] 100 100 100 100 100 100 100 225 225 225 225 225 225 225 325 325 325 325 325 325 325
Levels: 325 225 100
```

Typos will matter when setting the order of the levels. Look what happens if we don't write the categories in the `levels` argument exactly as they appear in the dataset (notice the `326` instead of `325` in the code below). R sets the levels how we requested, and sets all values that weren't in the levels to `NA`. It's important to check what's happening as you go along to avoid these sorts of mistakes.

```
factor(standdbh$tpa, levels = c(326, 225, 100) )
```

```
 [1] 100  100  100  100  100  100  100  225  225  225  225  225  225  225  <NA> <NA> <NA> <NA> <NA>
[20] <NA> <NA>
Levels: 326 225 100
```

Controlling factor level names

We can also change the *names* of the categories/levels using the `labels` argument. In this case, changing the names will make the levels easier to understand for those of us who rarely work with tree-thinning data.

The order of the categories in `labels` must be the same as in `levels` for this to work correctly. Now we are changing the actual values in the dataset as well as controlling the information on the levels of the factor.

```
factor(standdbh$tpa,
       levels = c(325, 225, 100),
       labels = c("light", "moderate", "heavy") )
```

```
 [1] heavy    heavy    heavy    heavy    heavy    heavy    heavy    moderate moderate moderate
[11] moderate moderate moderate moderate light    light    light    light    light    light
[21] light
Levels: light moderate heavy
```

If the order of the categories in `labels` is not the same as the `levels` information, we can make an error that will drastically change the dataset and would lead to mistakes in all the rest of our work. Look at the result of mixing up the order of the `labels`. R does exactly what it is told to do, but now the `heavy` and `light` groups are mislabeled.

```
factor(standdbh$tpa,
       levels = c(325, 225, 100),
       labels = c("heavy", "moderate", "light") )
```

```
 [1] light    light    light    light    light    light    light    moderate moderate moderate
[11] moderate moderate moderate moderate heavy    heavy    heavy    heavy    heavy    heavy
[21] heavy
Levels: heavy moderate light
```

As we've been going along practicing with `factor()`, we haven't assigned a name to anything we've done; we've instead been printing the result to the Console. If you look at the structure of the dataset `standdbh` in your Environment pane, you will see that `tpa` is still numeric. A common mistake for R beginners is to use `factor()` without assigning a name to the new categorical variable and then wondering why changing the variable to a factor didn't work.

Let's create a new factor variable called `thin` in the dataset `standdbh` based on the `tpa` variable. We'll change the order of the levels so that the light thinning group is the lowest level, and name the groups `light`, `moderate`, and `heavy` to represent the amount of thinning rather than trees per acre.

Like we learned to do last week, we will do this within the **dplyr** function `mutate()` to avoid using dollar sign notation. Be sure to remove all dollar signs from within `mutate()`.

```
standdbh = mutate(standdbh, thin = factor(tpa,
                                          levels = c(325, 225, 100),
```

```
                                        labels = c("light", "moderate", "heavy") ) )
standdbh$thin
```

```
 [1] heavy    heavy    heavy    heavy    heavy    heavy    heavy    moderate moderate moderate
[11] moderate moderate moderate moderate light    light    light    light    light    light
[21] light
Levels: light moderate heavy
```

You can see `thin` in the structure of the dataset in your Environment pane.

```
tibble [21 x 4] (S3: tbl_df/tbl/data.frame)
 $ tpa       : num [1:21] 100 100 100 100 100 100 100 225 225 225 ...
 $ stand     : num [1:21] 1 2 3 4 5 6 7 8 9 10 ...
 $ sitegrowth: num [1:21] 4.94 3.77 3.88 3.88 4.79 ...
 $ thin      : Factor w/ 3 levels "light","moderate",..: 3 3 3 3 3 3 3 2 2 2 ...
```

Exploratory tables and graphics

Exploratory tables

We can use `group_by()` and `summarise()` to make a nice summary table by each treatment group for ease of data exploration. Notice that you can use a variety of summary functions within a single `summarise()` statement by putting commas between them. Each summary variable is given a unique name.

In this example I show you that the name of a grouping variable can be changed within `group_by()` if you want to make the name of the grouping variable nicer in the resulting table. This time we only have a single grouping variable, `thin`, which we will rename as `Thinning` in our new summary dataset.

In this example you also get to see the **dplyr** function `n()`, which is for calculating the number of rows that each group has in the dataset.

The purpose of making summary statistics is to better understand the dataset. Take a look at the group means and ranges and think about the research question. How big are the differences in group means? How much do the ranges overlap? Are you seeing any skew in the raw data? Is the variation similar in groups? We don't make summaries or graphs just to check a box that we did so; we make them so we understand the data prior to the analysis.

```
( sumdat = summarise( group_by(standdbh, "Thinning" = thin),
                  n = n(),
                  Mean = mean(sitegrowth),
                  SD = sd(sitegrowth),
                  Median = median(sitegrowth),
                  Minimum = min(sitegrowth),
                  Maximum = max(sitegrowth),
                  .groups = "drop") )
```

```
# A tibble: 3 x 7
  Thinning     n  Mean    SD Median Minimum Maximum
  <fct>    <int> <dbl> <dbl>  <dbl>   <dbl>   <dbl>
1 light        7  4.01 0.458   4.06    3.09    4.47
2 moderate     7  4.10 0.405   4.09    3.58    4.66
3 heavy        7  4.30 0.476   4.27    3.77    4.94
```

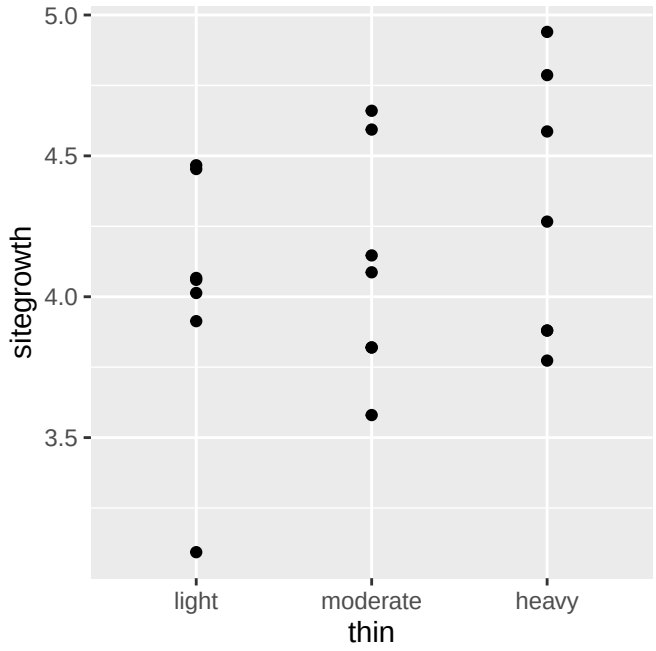
Exploratory graphics

Let's also make exploratory graphics. With a single explanatory variable and a group size of only 7, there are limited graphics to make since boxplots and histograms aren't particularly useful for such small sample sizes.

We'll use `qplot()` from package **ggplot2** (which we need to load) to create a scatterplot of `sitegrowth` vs `thin`. What kind of differences do you see among the groups? The low point in the "light" group stood out to me. I also was interested to see the amount of overlap in values across groups.

```
library(ggplot2)

qplot(x = thin, y = sitegrowth, data = standdbh)
```



Fitting a separate means model with lm()

Let's fit an initial model that allows mean site growth to differ among thinning density using `lm()`. We'll explore the resulting linear model object that we name `mod1`. If you don't remember how to use `lm()` from your introductory statistics classes, see the help page (`?lm`) or ask me.

```
# Fit the model
mod1 = lm(sitegrowth ~ thin, data = standdbh)
```

We can see the names of everything in the object and in the summary of the object. These are all things we could pull out of the model using dollar sign notation.

```
# The names of the model components
names(mod1)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "contrasts"     "xlevels"       "call"          "terms"
[13] "model"
```

```
# The names of the summary model components
names( summary(mod1) )
```

```
 [1] "call"          "terms"         "residuals"     "coefficients"  "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squared" "fstatistic"    "cov.unscaled"
```

Functions that work with lm objects

There are many functions that work on **lm** objects (i.e., models fit with the `lm()` function). As the quarter progresses, we'll see that many functions that have the same names work on different types of objects based on different kinds of models. Even though the function name doesn't change, the function often works differently and has different arguments depending on the object type we are using it on. This means it is important to understand how to find help for functions that work on a specific type of model object.

To find functions that work on specific types of model objects, we can use the `methods()` function. Here we'll ask to see the functions that were specifically created to work with **lm** objects.

```
methods(class = "lm")
```

```
 [1] add1           alias          anova          case.names     coerce         confint
 [7] cooks.distance deviance       dfbeta         dfbetas        drop1          dummy.coef
[13] effects        extractAIC     family         formula        fortify        hatvalues
[19] influence      initialize     kappa          labels         logLik         model.frame
[25] model.matrix   nobs           plot           predict        print          proj
[31] qr             residuals      rstandard      rstudent       show           simulate
[37] slotsFromS3    summary        variable.names vcov
see '?methods' for accessing help and source code
```

We can use any of these functions on **lm** objects directly, but if we need the object-type-specific help page then we need to add ".lm" to the end of the function name.

We'll use the `anova()` function as an example. When we look at the help page for the generic `anova()` function, we get a generic help page that is not very useful in figuring out how how to use the function

```
?anova
```

To find specific help for how the `anova()` function works on **lm** objects, we need to use the name of the function with the ".lm" addition: `anova.lm`. Functions I often need to check the specific help page for are functions like `anova()` and `residuals()`, which return different outputs and have different defaults depending on the object type.

```
?anova.lm
```

Checking model assumptions

We need to check that the assumptions of the model are reasonably met using the residual and fitted values from the model before we use the results from the model. While it's tempting to do so, you never want to start looking at any model results via `summary()` until you're satisfied that all assumptions are reasonably met.

To keep ourselves organized, we will add the residuals and fitted values to the dataset `standdbh`. This becomes more important as we start using more complicated models, so we will start practicing this today.

For linear model objects, you can use the functions `fitted()` and `residuals()` to extract this information from the model or you can pull them from the linear model object by name using the dollar sign notation, (i.e., `mod1$fit` and `mod1$resid`).

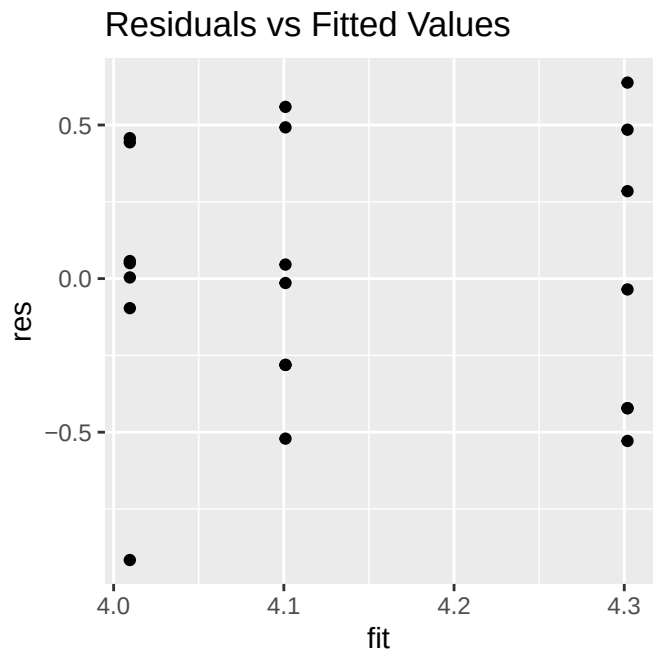
```
# Get the fitted value for each observation
standdbh$fit = fitted(mod1)

# Get the residuals of the model
standdbh$res = residuals(mod1)
```

Residuals vs fitted values

The first plot we'll make is a residual vs fitted values plot. Each of us will need to decide if we see anything in this plot to indicate lack of fit. This looks fine to me, showing similar spread in the residuals along the fitted values and no unusual patterns. In our own dataset it would be worth investigating the one point in the `light` group that stands to see if we could figure out why it was unusual.

```
qplot(x = fit, y = res, data = standdbh,
      main = "Residuals vs Fitted Values")
```

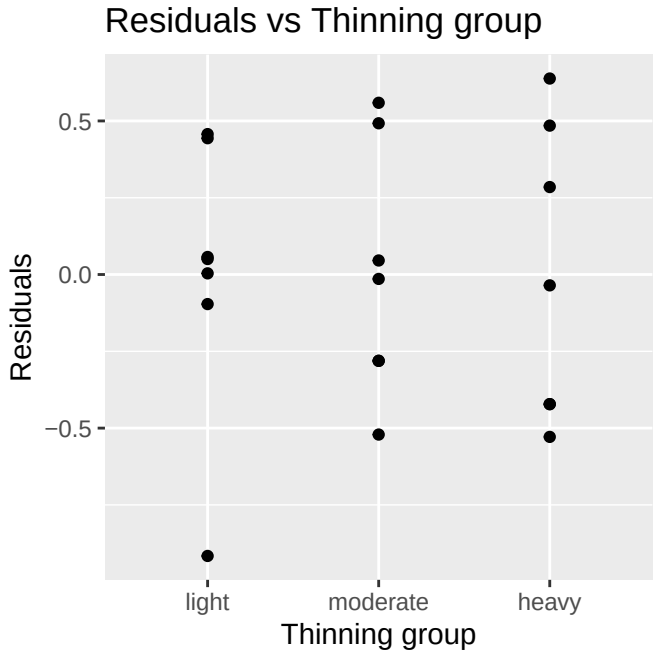


Residuals vs explanatory variables

It's also good practice to make residual vs explanatory variable plots. In this case our explanatory variable is the thinning variable `thin`. With only a single explanatory variable, we don't gain much more from this plot, although we can see which group the residuals are associated with more easily. This can help us see which group any problems we note are associated with.

Does it look like assuming constant variance of errors among groups is reasonable? Look at the spread of points for each group to help you decide.

```
qplot(x = thin, y = res, data = standdbh,
      xlab = "Thinning group",
      ylab = "Residuals",
      main = "Residuals vs Thinning group")
```

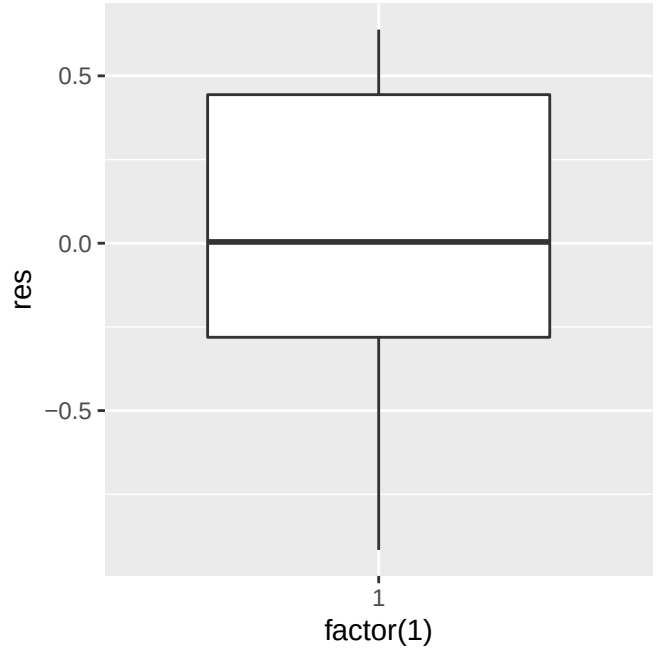


Residual normality/symmetry

We also need to check for normality/symmetry. We can do this with histograms, boxplots or quantile-quantile plots (aka q-q plots). For a single boxplot of all the residuals using `qplot()` we need to set the x-axis to an arbitrary single value.

Does it look like the residuals are reasonably symmetric?

```
qplot(x = factor(1), y = res,
      data = standdbh, geom = "boxplot")
```



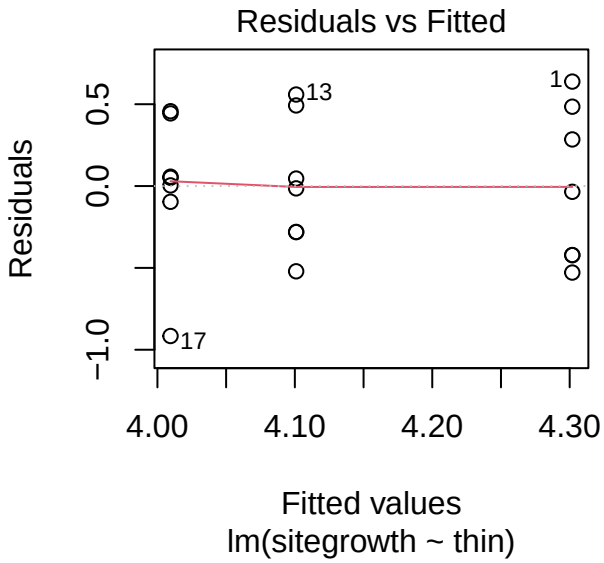
Using `plot.lm` for checking assumptions graphically

There is a `plot()` function that works directly with **lm** objects. It gives four plots by default. You can put all four plots into a single window if you run the code `par(mfrow = c(2, 2))` prior to making the plots. Make sure you reset your plotting device to a single plot, though, using `par(mfrow = c(1, 1))`, after you are done with these plots.

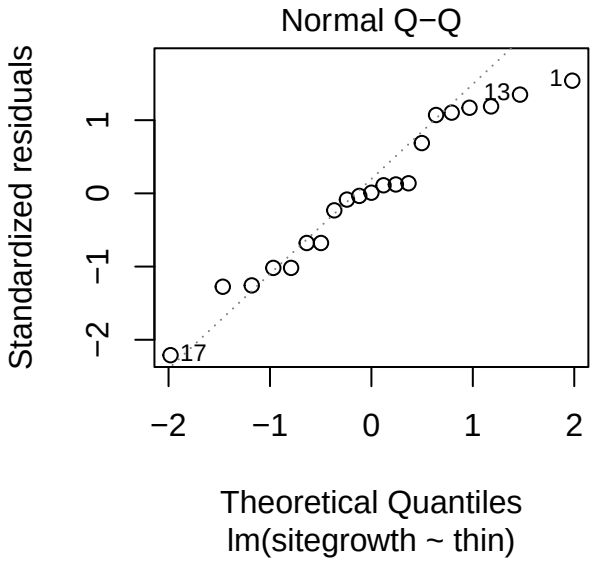
Because I only ever work with two of the plots, I don't usually want to make all four of them. I like to take advantage of the `which` argument to ask for the plots I want to see. See `?plot.lm` for more information.

Below I plot the residual vs fitted value plot and the q-q plot of the residuals. Only use a quantile-quantile plot when checking for residual symmetry if you know how to interpret it. It is possible that histograms or boxplots will be more useful for you when assessing the symmetry of the residuals. We will talk about the interpretation of q-q plots later in the quarter.

```
plot(mod1, which = 1) # residual vs fitted values
```




```
plot(mod1, which = 2) # qqnorm plot of residuals
```



Model results

Once we are comfortable that the model assumptions are reasonably met, we can report any model results of interest from `anova()` and/or `summary()`. If they are not met we need to find an alternative model. In this case I would say the assumptions were reasonably met.

The `anova()` function for **lm** objects gives an ANOVA table, showing the overall F-tests for explanatory variables. We know this because that's what it says in the help page for `anova.lm`, which we looked at a few minutes ago. This is not true for other object types, so don't get into the habit of confusing the name of the function (`anova()`) with the statistical analysis term **ANOVA**.

I'll say it one more time since this is such a common problem: using the `anova()` function does not mean you "did an ANOVA", so practice avoiding that language.

```
anova(mod1)
```

```
Analysis of Variance Table

Response: sitegrowth
          Df Sum Sq Mean Sq F value Pr(>F)
thin       2 0.3132 0.15660  0.7815 0.4726
Residuals 18 3.6068 0.20038
```

```
summary(mod1)
```

```
Call:
lm(formula = sitegrowth ~ thin, data = standdbh)

Residuals:
     Min       1Q   Median       3Q      Max
-0.91619 -0.28095  0.00381  0.44381  0.63810

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   4.00952    0.16919  23.698 5.05e-15 ***
thinmoderate  0.09143    0.23927   0.382    0.707
thinheavy     0.29238    0.23927   1.222    0.237
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4476 on 18 degrees of freedom
Multiple R-squared:  0.0799,    Adjusted R-squared:  -0.02234
F-statistic: 0.7815 on 2 and 18 DF,  p-value: 0.4726
```

Estimating group differences

This study was designed to provide evidence for the following two questions:

> On average, is the site growth dbh for heavy thinning at least 0.5 inches more than light thinning?
> On average, is the site growth dbh for moderate thinning at least 0.5 inches more than light thinning?

We can take advantage of the default *treatment contrasts* in `lm()` to answer these questions. This is especially easy for us because `light` is the reference level for the grouping variable `thin`.

In the model summary, the `Estimate` on the line that says `thinmoderate` is the estimated difference in mean site growth between the moderate and light thinning densities. The `Estimate` on the line that says `thinheavy` is the estimated difference in mean site growth between the heavy and light thinning groups. These are the two estimates we are interested in.

We can extract estimated values and confidence intervals around the estimates from the model `mod1` using the functions `coef()` and `confint()`.

```
# All the coefficients
coef(mod1)
```

```
 (Intercept) thinmoderate    thinheavy
  4.00952381   0.09142857   0.29238095
```

```
# All the confidence intervals for the coefficients
confint(mod1)
```

```
                2.5 %    97.5 %
(Intercept)   3.6540657 4.3649819
thinmoderate -0.4112650 0.5941222
thinheavy    -0.2103127 0.7950746
```

We are only interested in the results that address our questions of interest, which are estimated differences in mean site growth between each thinning group and the light thinning group. We can pull just those values out of the model summary output by

combining the extract brackets ([) with the indexes of the values of interest. In this case, we want the second and third values in the coefficients vector and the second and third rows from the confidence interval matrix.

```
# Just the coefficients we are interested in
coef(mod1)[2:3]
```

```
thinmoderate    thinheavy
  0.09142857   0.29238095
```

```
# Just the confidence intervals we are interested in
confint(mod1)[2:3, ]
```

```
                 2.5 %    97.5 %
thinmoderate -0.4112650 0.5941222
thinheavy    -0.2103127 0.7950746
```

Changing the reference level based on the research question

What if our research question was about each thinning group vs the `heavy` thinning group? A simple way to get the results we wanted from our model is by using the function `relevel()` to change the reference level to `heavy` and refit the model.

Changing the reference level does not change our model fit. If we've already checked and are satisfied that the assumptions are reasonably met we don't need to do any additional work.

Here is an example, making a new variable called `thin2` with `heavy` as the reference level and fitting a new model called `mod2` using this new variable.

```
standdbh$thin2 = relevel(standdbh$thin, "heavy")

standdbh$thin # original order
```

```
 [1] heavy    heavy    heavy    heavy    heavy    heavy    heavy    moderate moderate moderate
[11] moderate moderate moderate moderate light    light    light    light    light    light
[21] light
Levels: light moderate heavy
```

```
standdbh$thin2 # new order
```

```
 [1] heavy    heavy    heavy    heavy    heavy    heavy    heavy    moderate moderate moderate
[11] moderate moderate moderate moderate light    light    light    light    light    light
[21] light
Levels: heavy light moderate
```

```
# Fit new model
mod2 = lm(sitegrowth ~ thin2, data = standdbh)
summary(mod2) # Show the output with "heavy" as the reference level
```

```
Call:
lm(formula = sitegrowth ~ thin2, data = standdbh)

Residuals:
     Min       1Q   Median       3Q      Max
-0.91619 -0.28095  0.00381  0.44381  0.63810

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)     4.3019     0.1692  25.426 1.47e-15 ***
thin2light     -0.2924     0.2393  -1.222    0.237
thin2moderate  -0.2010     0.2393  -0.840    0.412
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4476 on 18 degrees of freedom
Multiple R-squared:  0.0799,    Adjusted R-squared:  -0.02234
```

```
F-statistic: 0.7815 on 2 and 18 DF,  p-value: 0.4726
```

Organizing estimates from the model into a data frame

Making a data.frame containing our results of interest will make our graphical display easier to create, as **ggplot2** was designed to work with data.frames. We can achieve this with the `data.frame()` function. We will want to make the names of the columns nicer using the `names()` function and assigning new names like we practiced last week.

```
( diffmod1 = data.frame(coef(mod1)[2:3],
                        confint(mod1)[2:3, ]) )
```

```
             coef.mod1..2.3.     X2.5..   X97.5..
thinmoderate      0.09142857 -0.4112650 0.5941222
thinheavy         0.29238095 -0.2103127 0.7950746
```

```
# Rename column names
names(diffmod1) = c("Estimate", "Lower", "Upper")

# Check that the names are changed
diffmod1
```

```
               Estimate      Lower     Upper
thinmoderate 0.09142857 -0.4112650 0.5941222
thinheavy    0.29238095 -0.2103127 0.7950746
```

For graphing purposes we'll need a variable with levels that represent the two groups that are part of each comparison in this "results" data.frame. One way to do this by taking the row names from `diffmod1`, adding them to the dataset with the name `group`. Then we can change the level order and labels with `factor()`. This is good practice using what we learned about `factor()` earlier today.

This is what the row names of `diffmod1` look like.

```
rownames(diffmod1)
```

```
[1] "thinmoderate" "thinheavy"
```

It made sense to me aesthetically to have the `moderate` minus `light` comparison come before the `heavy` minus `light` comparison in the graphic, which is why I changed the order of the levels when I create the new factor variable.

I make my group labels very explicit to make it clear which of the thinning groups had a larger mean site growth. Plan on doing something similar in your own graphs.

```
( diffmod1$group = factor(rownames(diffmod1),
                          levels = c("thinmoderate", "thinheavy"),
                          labels = c("Moderate minus Light", "Heavy minus Light") ) )
```

```
[1] Moderate minus Light Heavy minus Light
Levels: Moderate minus Light Heavy minus Light
```

```
diffmod1
```

```
               Estimate      Lower     Upper                 group
thinmoderate 0.09142857 -0.4112650 0.5941222 Moderate minus Light
thinheavy    0.29238095 -0.2103127 0.7950746    Heavy minus Light
```

Wrapping up an analysis

Tables

We already made a summary table, which we may or may not want for our write-up. However, I forgot to round the information to a reasonable number of significant digits.

```
sumdat
```

```
# A tibble: 3 x 7
  Thinning     n  Mean    SD Median Minimum Maximum
  <fct>    <int> <dbl> <dbl>  <dbl>   <dbl>   <dbl>
```

```
1 light        7  4.01 0.458   4.06    3.09    4.47
2 moderate     7  4.10 0.405   4.09    3.58    4.66
3 heavy        7  4.30 0.476   4.27    3.77    4.94
```

We can round the last five columns (columns "Mean" through "Maximum") to one digit via `across()` in function `mutate()` from package **dplyr**. `across()` allows us to change multiple columns at once, which is convenient when we want to apply the same function to many columns.

We give the dataset we want to change in `mutate()` and then use `across()` to give the names of the columns that we want to round to a single digit in `.cols` and the name of the function we want to use on the columns in `.fns` (we'll use function `round()` in this case). At the end we give the arguments to pass to the function; here I want to round to a single digit in `round()` so use `digits = 1`.

```
( sumdat = mutate(sumdat,
                  across(.cols = Mean:Maximum,
                         .fns = round,
                         digits = 1) ) )
```

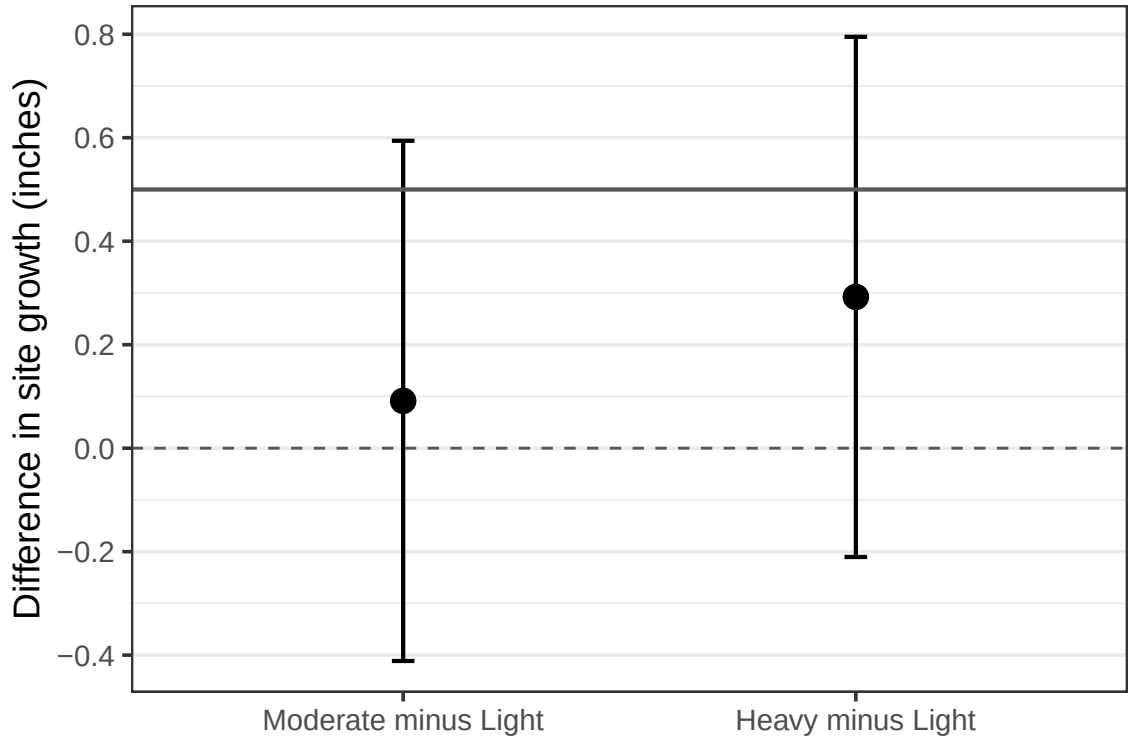
```
# A tibble: 3 x 7
  Thinning     n  Mean    SD Median Minimum Maximum
  <fct>    <int> <dbl> <dbl>  <dbl>   <dbl>   <dbl>
1 light        7   4     0.5    4.1     3.1     4.5
2 moderate     7   4.1   0.4    4.1     3.6     4.7
3 heavy        7   4.3   0.5    4.3     3.8     4.9
```

Graphics

The last thing we will do is plot the results. Today we will create a graphic with results to help address the research questions. The graphic will show the estimated differences in means and a 95% confidence interval around those differences. I added a line at 0.5 inches to help with the interpretation of important practical differences. The dashed line at 0 is to give the reader a reference point.

The code below is annotated to give you a sense of what each layer does.

```
( g1 = ggplot(diffmod1, aes(x = group, y = Estimate ) ) +  # define x and y axes
       # and dataset with the information
       geom_errorbar(width = .05, lwd = .75, aes(ymin = Lower, ymax = Upper) ) +
                                                          # Put in CI as error bars
       geom_point(size = 4) + # Add point at differences in means, change size
       geom_hline(yintercept = 0, color = "grey34", lty = 2) +  # horizontal line at 0 for reference
       geom_hline(yintercept = .5, color = "grey34", lwd = .75) +  # horizontal line at .5 (practical diff)
       labs(x = NULL, y = "Difference in site growth (inches)" ) +  # change axis titles
       theme_bw(base_size = 14) +  # Make black and white for ease of printing
       theme(panel.grid.major.x = element_blank() ) + # Take out vertical gridlines
       scale_y_continuous(breaks = seq(-.4, .8, by = .2) ) ) # Add breaks on y axis
```



If you were not going to caption your graphic, you might consider adding a title using `ggtitle` as below. The example uses `\n`, which represents a line break. Including the line break puts the title on two lines instead of one. Generally, though, your graphics will need captions to properly describe the graphic rather than titles.

```
g1 + ggtitle("Difference in mean site growth of light thinning\nand other thinning groups")
```