

meat.r: Explanation of code

Goals of code:

- Fitting a regression line
- Calculating confidence intervals on model parameters
- Estimating mean Y at a specified X
- Creating data frames with specified X values
- Data frames, matrices, and lists
- Calculating confidence and prediction intervals
- Analyzing a subset of data
- Only printing a few observations
- Overlaying predicted values
- ANOVA lack of fit test

The initial commands read the meat.txt file and create a new variable called logtime.

### **Fitting a regression line: `lm()`**

The `lm()` function is the workhorse R model fitting function. When the X variable (right hand side of the `~` **is not** a factor, `lm()` fits a regression line.

The command `meat.lm <- lm(ph~logtime, data=meat)` will fit a regression model to predict  $y = \text{ph}$  from  $x = \text{logtime}$ , where both variables are in the data frame called `meat`. The result is stored in the `meat.lm` object.

If you print the object (`meat.lm`), you see what was done to create `meat.lm` and the estimated coefficients. Intercept is the intercept ( $\beta_0$ ) and `logtime` is the regression slope ( $\beta_1$ ). Slope coefficients are always labeled by the name of the associated X variable. That will be very useful when we have more than one X variable in a model.

If you use `summary()` on the output object, you get a lot more information. I don't find the first block of results (a five number summary of the residuals) very useful. The second block of results gives you the estimated coefficients, their se's, T values for the test of parameter = 0 and the associated p-value for each parameter. Again, the slope is labeled by the name of the associated X variable.

The residual se is the pooled sd ( $\sqrt{\text{MSE}}$ ). The rest of the numbers can be ignored or will be discussed later.

The `predict()` and `resid()` functions can be used to extract predicted values for X values in the data set used to fit the regression. This gives you the residual plot that we've used to assess assumptions.

### **Calculating confidence intervals on model parameters: `confint()`**

The `confint()` function calculates confidence intervals on model parameters. It is especially useful for regression models because the model parameters are easily interpretable. The required argument is the name of the model fit. By default, intervals are calculated for all parameters and are 95%. You can specify a `parms=` argument to only report intervals for selected parameters and `level=` to change the coverage (expressed as a number between 0 and 1, not a percentage).

### **Estimating mean Y for a specified X: `predict()` with a `newdata=` argument**

The `predict()` function can provide predicted values and associated information for X values not in the data set. This is done by providing a new data frame with the desired X values. That new data frame is specified by the `newdata=` argument. This must be a data frame (you can't just say: `newdata=log(5)`). Sorry. And, the name in the data frame must match the variable name in the model.

The `data.frame()` function creates a data frame from a set of values. In this case, we want to predict for `time = 5` hours, so `logtime` is `log(5)`.

This data frame **must** include a column with the same variable name as that in the data frame used to fit the model. The `logtime=` piece specifies the variable name. The `log(5)` specifies the value. That data frame is printed, then used as the value of the `newdata=` argument. The result of `predict()` is the predicted value. As always, this can be stored in a variable or in a data frame if you want to save the result.

Additional arguments to `predict()` request additional results. `se.fit=T` requests standard errors for the predicted mean. `interval='confidence'` requests a confidence interval for the predicted mean. `interval='prediction'` requests a prediction interval for an individual observation. Confidence and prediction intervals are discussed in detail further down.

### **Specifying many X values to predict at: `data.frame()`**

You can extend what you just did to many X values. In all cases, you need a data frame with a named vector specifying the desired values. The three examples of data illustrate three ways to produce the desired data frame. They create slightly different sets of X values. You only need one of these.

`c(1,1.5,2,2.5,3,3.5,4,4.5,5):`

The `c()` function concatenates (hence the c) values to make a vector. `c()` can be used anywhere you need to create or specify a vector of values. The values are separated by commas.

`seq(1, 8, 0.25):`

The `seq()` function generates a sequence of values. The first two arguments are the starting and ending values. The third is the step. If the step is omitted, e.g., `seq(1,8)`, a step of 1 is assumed.

1:8:

The 1:8 in the third use of `data.frame()` is a shortcut way to generate a sequence of integers from the starting to ending value. The code then computes the `logtime` variable, as we've done before.

### Data frames, matrices, and lists:

You need to know a bit about the various ways R will store information, because some of the functions we're about to use deliver results in novel ways.

R provides many ways to store information. So far, we've met scalars (e.g., 5) and data frames, and we've just met vectors (e.g., `c(1, 2, 4, 8)`). A matrix has rows and columns, just like a data frame, but the entire matrix is either numeric or character. A data frame can have some columns that contain numbers and other columns that contain character strings. If a data frame with both numbers and character strings was converted to a matrix, it would become a matrix of character strings and all the numbers would be converted to character strings. The `as.matrix()` function converts another type of object into a matrix. The code involving `test`, `test2`, and `testm` demonstrates the difference between a data frame and a matrix.

You can access individual rows or columns by subscripting the matrix, just like you would subscript a data frame. Subscripts go in single square brackets. A matrix has two subscripts, which are separated by a comma. The first subscript indicates rows, the second indicates columns. A negative number omits the specified row(s) or column(s). If a subscript is omitted, e.g., `[,1]`, all rows or columns are selected. So `[,1]` will extract the first column (and all rows). You can also subscript vectors (only one index, so no comma) and data frames (rows and columns). If you omit the comma, the matrix is subscripted as if it were one long vector composed of stacked columns. You probably want to specify a comma.

The `names()` function extracts the column names from a data frame. If you try `names()` on a matrix, the result is nothing (NULL). The `dimnames()` function extracts row and column names from a matrix. You can also use `dimnames()` on a data frame; the row names are generated automatically by R. When you run the code, you'll see that the default for a matrix is to have column names but no row names.

The output from `dimnames()` is a list, which is a collection of related pieces of information. Unlike a data frame, in which all columns must be the same length, the components of a list can be different sizes, shapes, or types of data. You can have lists of lists and even lists of lists of lists.

A list can be recognized by either double square brackets `[[ ]]` or by `$` name followed by information. The output from `dimnames()` is a list with two elements. The first is the vector of row names; the second is the vector of column names. When printed as `[[1]]`, the components of the list are unnamed. You can access each component by subscripting in double square brackets, e.g., `temp[[2]]`.

Components of a named list, such as produced by `predict()` in the next section, can be accessed by number or by name. `testlist` is a list with two components, named `a` and `b`. Both `testlist$a` and `testlist[[1]]` refer to the first component. If referring to list elements by name seems like

referring to columns in a data frame, that's because a data frame shares many of the characteristics of a list and also many characteristics of a matrix. A data frame just has a regular structure than a list usually doesn't have.

### **Calculating confidence or prediction intervals: `predict(, interval=)`**

The `predict()` function returns predicted values. The required argument is the name of a saved model fit (e.g., `meat.lm`). The result is a vector of predicted values at each of the X's in the original data set. These are often more useful (e.g., for drawing plots) if you save the predictions into the original data frame. This works because there is one prediction for each observation in the data frame.

Predictions are made for new X values not in the data frame when `newdata=` is specified. The argument to `newdata=` is the name of a data frame with new values for the X variable(s). The new data can have additional variables, which are ignored. But, it must have a column (or columns) with the same names as the variable(s) used to fit the model. We created `meat.new` at the beginning of today's lab. Again, it is often useful to store these predictions in the `newdata` data frame (not the original data frame!). You almost certainly can't save the new predictions in the original data frame because the number of predictions probably doesn't match the number of observations. It does match the number of rows of new X values.

`predict()` has optional arguments that request more information about each predicted value (either for X's in the original data frame or a `newdata=` data frame). The `se.fit=T` argument requests standard errors for each predicted value. The `interval='confidence'` argument requests a 95% confidence interval for the predicted values. Adding `level=` with a number between 0 and 1 changes the coverage to the specified value. Changing to `interval='prediction'` gives you prediction intervals instead of confidence intervals.

The output from `predict()` is a vector when all you request are predictions. When you add `se.fit=T` to request standard errors, the result is a list. The `$fit` component is the vector of predicted (fitted) values. The `$se.fit` component is the vector of standard errors. When you request an interval, the output is a matrix, with the 2nd and 3rd columns being the endpoints of the requested interval. (The first column is the predicted value). If you request both the standard error and an interval, you get a list with the `$fit` component as a matrix. You can extract the pieces you want by indexing or subscripting the list or matrix as described above.

### **Analyzing a subset of the data: `subset()` or `subset=`**

R provides a couple of ways to restrict an analysis to a subset of the data. If you want to use a subset many times (i.e., many analyses or plots using the same subset), I find it easier to create a second data frame with the desired subset of data. If you only want to do one analysis, it can be easier to specify that subset in the analysis.

To create a new data frame with a subset of values: There are two ways: selecting rows, or using `subset()`. I illustrate `subset()`. The first argument is a data frame; the second argument specifies the rows to keep, using a logical operator. The example, `subset(meat, time <= 6)` starts with all rows of the `meat` data frame and keeps only those where `time` is less than or equal to 6. The logical

expression is evaluated in the data frame, so you don't need to write `meat$time` to specify where to find the time variable. To specify the subset as part of an analysis, add `subset=` logical operation specifying rows to keep. Again, the expression is evaluated in the data frame, so you don't need to write `subset=(meat$time <= 6)`.

My practice is to put the logical expression inside `()` just so R doesn't get confused.

### Logical operators:

R provides an extensive list of logical operators. These include

Symbol	meaning	notes
<code>==</code>	equals	requires two equals signs
<code>&lt;</code>	less than	
<code>&gt;</code>	greater than	
<code>&lt;=</code>	less than or equal	
<code>&gt;=</code>	greater than or equal	
<code>!=</code>	not equal	
<code>%in%</code>	in	followed by a vector of values, e.g. <code>c(1, 3, 6)</code> will be true if first argument matches any of the second

### Only printing a few observations: `head()`

Sometimes, I want to print a few observations in a data set to get variable names (e.g., after reading a data set) or check for gross errors. The `head()` function prints the first six rows of a data frame, matrix, or vector. `tail()` prints the last six rows.

### Overlaying observations and predicted values: `plot()` followed by `lines()`

We have previously used `plot()` to plot data. You can follow `plot()` with commands that add stuff to the plot. The `lines()` function draws lines (as connect the dots, so it usually helps to to have observations in sorted order). The arguments to `lines()` are the X and Y information. Additional arguments modify the appearance of the line. The first `lines()` command adds the fitted line, drawn as a solid black line. fitted line. The second and third adds the lower and upper prediction limits as a dashed line (`lty=2`, where `lty` is "line type"). The default line type is `lty=1`. The `legend()` function adds a legend to the plot. The first argument is the location. All the rest can be in any order and specify what to put on the plot. I prefer legends without boxes; `bty='n'` (box type) suppresses the box. I want legends for the two line types, so I specify two line types in `lty=` and my legend as a vector of character strings in `legend=`. If you used `pch=` and `legend=`, you would label points in the legend. The help file for `legend` gives you a long list of options.

### ANOVA lack of fit test: `anova()`

To construct the ANOVA lack of fit test, we need to fit two analyses using `X = time`. The regression treats `X` as continuous; the ANOVA treats `X` as a factor. My practice is to create a copy of the `X` variable that is a factor variable. `time.f` is that factor version of the time variable. We will use this variable to indicate groups, each with its own mean, so it doesn't matter whether the groups are defined by time or `logtime`.

We can compare the fit of the regression and the fit of the ANOVA model in two different ways.

1. Fit both models, then use `anova()` to compare the two fits. That is done by providing two models to `anova()`. If you wanted to sequentially compare more than two models (e.g., a simple linear regression, a quadratic regression, then an ANOVA fit), you provide three arguments to `anova()`.
2. Fit one model with both terms, e.g., `lm(ph ~ logtime + time.f)`, with a `+` separating the two terms. Passing the output from that fit to `anova()` gives you the sequential change in fit as each term is added to the model.

Note: If the first variable in the model is the factor variable, R refuses to fit the second term. If you specify two models to `anova()` and the factor variable is first, the changes in df and SS are both negative. Fix by reversing the order of the variables in the `lm()` formula or the order of the models in `anova()`