

Introduction to Computational BioStatistics with R: linear models

Erik Spence

24 October 2023

Today's slides

Today's slides can be found here. Go to the "Introduction to Computational BioStatistics with R" page, under Lectures, "Linear models".

<https://scinet.courses/1301>

Today's class

Today we will continue our adventures in data analysis.

- Initial data exploration.
- Linear models.
- Formulae.
- Quadratic models.

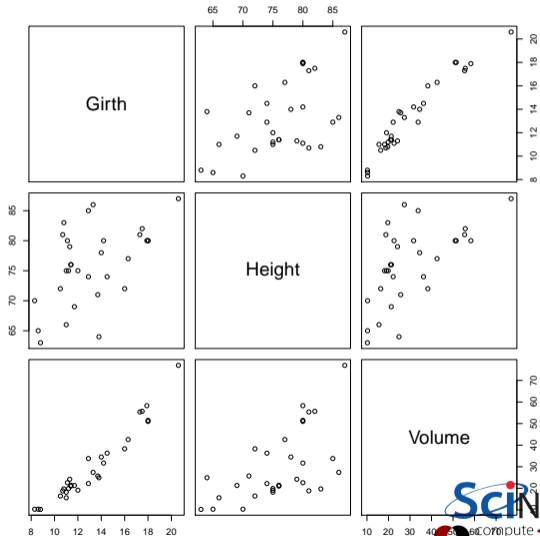
As always, ask questions.

Look at your data

The very first step to dealing with your data is to plot it. Always always always!

The 'pairs' function will do the same as demonstrated here.

```
>
> str(trees)
'data.frame':  31 obs.  of 3 variables:
 $Girth :  num  8.3  8.6  8.8 10.5 10.7 ...
 $Height:  num  70  65  63  72  81 ...
 $Volume:  num 10.3 10.3 10.2 16.4 18.8 ...
>
> plot(trees)
>
```

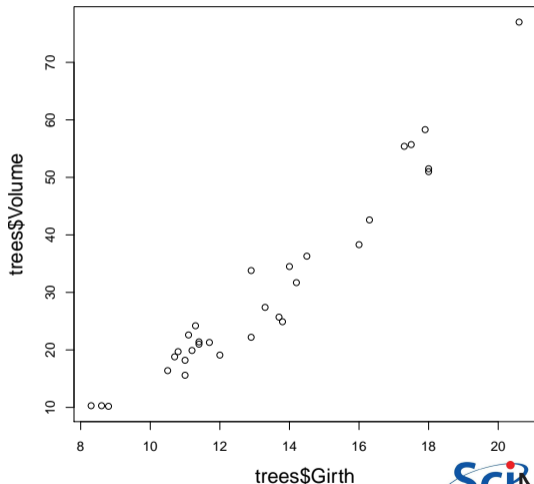


Look at your data, continued

Once you've had a first look, you might want to take a closer look at a particular pair of variables.

```
>  
-----  
> plot(trees$Girth, trees$Volume)  
-----  
>
```

Looks like they might be related.



Correlation and covariance

If we suspect that two variables might be related to one another, it's worth our time to look at the correlation and covariance of the variables.

Covariance:

$$\sigma_{XY} = E [(X - E (X)) (Y - E (Y))]$$

Correlation (Pearson's correlation):

$$\rho_{XY} = \frac{E [(X - E (X)) (Y - E (Y))]}{\sigma_X \sigma_Y}$$

Recall that the standard deviation:

$$\sigma_X = \sqrt{E [(x - E (x))^2]}$$

Where E is the expectation value of the quantity in question.

Correlation and covariance, continued

The "cor" function will produce the correlation from the previous slide.

The "var" function returns the variance or the covariance, depending on the number of arguments.

Recall that the standard deviation is the square root of the variance.

```
>
> cor(trees$Girth, trees$Volume)
[1] 0.9671194
>
> var(trees$Girth, trees$Volume)
[1] 49.88812
>
> var(trees$Girth)
[1] 9.847914
>
> sd(trees$Girth)
[1] 3.138139
>
```

Model fitting

One important application of statistics is the fitting of models to empirical data. There are many ways to do this, but they are all based on the same principles:

- collect some data.
- propose a relationship between the 'features', and the 'target' in your data (if there is a 'target').
 - ▶ 'features' are the independent variables in your data (\mathbf{x}),
 - ▶ the 'target' is the dependent variable (\mathbf{y}). Not all data sets have dependent variables.
- Fit your model to the data,
- Test and evaluate the quality of the model.

Depending on the field, this is called modelling, fitting, regression.

Linear models

Let's consider the simplest possible case, that the relationship between the independent and dependent variables is linear.

$$y \simeq \beta_0 + \beta_1 x_1 + \dots + \beta_n x_n + \delta$$

As usual:

- y is the dependent variable,
- x_1, \dots, x_n are the independent variables,
- β_0 is the intercept,
- β_1, \dots, β_n are the coefficients, and
- δ is noise.

For example, we might assume a relationship for plant growth:

$$\text{growth} \simeq \text{water} + \text{temp} + \text{fertilizer} \dots$$

The plant growth is linearly related to the temperature, amount of fertilizer and water, etc.

Fitting a linear model

We use the "lm" function to fit a linear model to our data.

The weird thing with the ~ ("tilde") is called a "formula".

Formulae are used, in R, to describe the functional relationship between variables when building models.

```
>


---


> model <- lm(trees$Volume ~ trees$Girth)


---


>


---


> model

Call:
lm(formula = trees$Volume ~ trees$Girth)

Coefficients:
(Intercept)  trees$Girth
      -36.943         5.066


---


>
```

$\text{Volume} \simeq -36.943 + (5.066 \times \text{Girth}).$

Fitting a linear model, continued

The `lm` function returns an object of class 'lm'.

This is essentially just a very-deep named list.

If we so desire, we can now use the model to calculate the model's prediction for a new tree, with a girth of, say, 15.12.

- Use the 'predict' function.
- Use the coefficients directly.

```
>
> model <- lm(Volume ~ Girth, data = trees)
>
> predict(model, newdata = data.frame(Girth = 15.12))
      1
39.65229
>
> coef(model)
(Intercept)  trees$Girth
-36.943459    5.065856
>
> coef(model)[1] * coef(model)[2] * 15.12
(Intercept)
39.65229
>
```

Using formulae, an aside

Formulae show up all over the place in R. There are two ways of building a formula:

- Use vectors of data as the arguments to the formula.
- Specify the names of the columns of a data frame, and then pass the data frame as an argument to the function.
- The entry to the left of the tilde is the dependent variable.
- All the entries to the right of the tilde are the independent variables (there can be more than one).

```
>
-----
> model <- lm(trees$Volume ~ trees$Girth)
-----
>
-----
> model <- lm(Volume ~ Girth, data = trees)
-----
>
-----
> model2 <- lm(Volume ~ Girth + Height,
+             data = trees)
-----
>
```

The second option, specifying column names, is unfortunate due to its syntax being inconsistent with the rest of R, but it is the one more commonly used. It's also somewhat more flexible for making predictions using the trained model.

Using formulae, an aside, continued

There are a few other ways to specify a formula.

- A formula can be assigned to a variable, and used later.
- To specify "all columns which have not yet been mentioned" use the ".".
- To remove an already-specified feature, use the minus sign.
- You can also mix data frame columns and non-column vectors.

```
> f <- Volume ~ .  
-----  
> class(f)  
[1] "formula"  
-----  
>  
-----  
> trees.model <- lm(f, data = trees)  
-----  
>  
-----  
> library(boot)  
-----  
> model <- lm(ulcer ~ . - sex - year, data = melanoma)  
-----  
>  
-----  
> model  
Call:  
lm(formula = ulcer ~ . - sex - year, data = melanoma)  
Coefficients:  
(Intercept)      time      status      age      thickness  
6.146e-01    -5.595e-05  -1.417e-01  4.210e-04  6.045e-02  
-----  
>
```

Calculating confidence intervals, an aside

Suppose that you've calculated the mean, \bar{x} , of some quantity. What is the uncertainty on that quantity?

To answer this question, we estimate the Standard Error

$$\text{SE}(x)^2 = \frac{s^2}{n}$$

where $s^2 = \sum (x_i - \bar{x})^2 / (n - 1)$. The 95% confidence interval, in which there is a 95% chance the true mean of the population lies, is given by

$$\mu \pm 1.96 \text{ SE}(x)$$

This is because 1.96 standard deviations is approximately what contains 95% of the normal distribution.

```
>
> x <- trees$Girth
>
> my.mean <- mean(x)
>
> se2 <- var(x) / length(x)
>
> my.mean - 1.96 * sqrt(se2)
[1] 12.14368
>
> my.mean + 1.96 * sqrt(se2)
[1] 14.35309
>
```

Calculating confidence intervals, an aside, continued

We can also use the Standard Errors to perform hypothesis tests on the coefficients of our linear model.

- The Null Hypothesis is that the coefficients in our linear model have a value of zero, meaning that the dependent variable does not depend on the independent variable.
- To test this we need to determine whether our estimate of the coefficient is sufficiently far from zero to reject the Null Hypothesis. How far is far enough?
- We can check in the usual way, by calculating a t-statistic

$$t_i = \frac{\beta_i - 0}{\text{SE}(\beta_i)}$$

- This estimates the number of standard deviations that β_i is away from zero.
- The question then becomes: what is the probability of getting a t statistic of value $|t|$, or larger? This is your p value.

Calculating confidence intervals, an aside, more

Given our t statistic (for a Null Hypothesis of zero):

$$t_i = \frac{\beta_i - 0}{\text{SE}(\beta_i)}$$

we need only determine the probability of getting $|t|$ or greater. To determine this we use a 't' distribution, which is very close to the Gaussian CDF for $n > 30$. The second argument is the number of degrees of freedom. The factor of 2 is because it could be left or right tailed.

The p value is small. The probability of committing a Type I error is quite low.

```
>
> coefs <- coef(summary(trees.model))
>
> est <- coefs["Height", "Estimate"]
> std.err <- coefs["Height", "Std. Error"]
>
> my.t <- (est - 0) / std.err
>
> my.t
[1] 2.606594
>
> 2 * (1 - pt(my.t, length(x) - 3))
[1] 0.01449097
>
```


Fitting a linear model, continued more

Important details about the model can be found in the summary:

- The "t value" is the estimate divided by the standard error.
- The p-value is the probability of achieving a value of t, or larger, under the Null Hypothesis (estimate = 0).

```
> model <- lm(Volume ~ Girth + Height, data = trees)
> summary(model)
Call:
lm(formula = Volume ~ Girth + Height, data = trees)

Residuals:
    Min       1Q   Median       3Q      Max
-6.4065 -2.6493 -0.2876  2.2003  8.4847

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  -57.9877    8.6382  -6.713  2.75e-07 ***
Girth         4.7082     0.2643  17.816 < 2e-16 ***
Height        0.3393     0.1302   2.607  0.0145  *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.882 on 28 degrees of freedom
Multiple R-squared:  0.948, Adjusted R-squared:  0.9442
F-statistic: 255 on 2 and 28 DF, p-value: < 2.2e-16
```

Fitting a linear model, continued even more

What about that
F-statistic at the bottom?

- The fit also gives an analysis of the null hypothesis that $\beta_1 = \beta_2 = \dots = \beta_n = 0$.
- This means that there's no dependence on the independent variables at all.
- This null hypothesis can be rejected in this case.

```
> model <- lm(Volume ~ Girth + Height, data = trees)
> summary(model)
Call:
lm(formula = Volume ~ Girth + Height, data = trees)

Residuals:
    Min       1Q   Median       3Q      Max
-6.4065 -2.6493 -0.2876  2.2003  8.4847

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -57.9877    8.6382  -6.713  2.75e-07 ***
      Girth     4.7082    0.2643  17.816 < 2e-16 ***
      Height     0.3393    0.1302   2.607  0.0145  *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.882 on 28 degrees of freedom
Multiple R-squared:  0.948, Adjusted R-squared:  0.9442
F-statistic: 255 on 2 and 28 DF, p-value: < 2.2e-16
```

Fitting a linear model, continued some more

There are some assumptions built into "lm". You need to know the fine print:

- The noise in the data, δ , is normally distributed about the true value.
- Homoscedasticity: the variance in the noise is constant throughout the data.

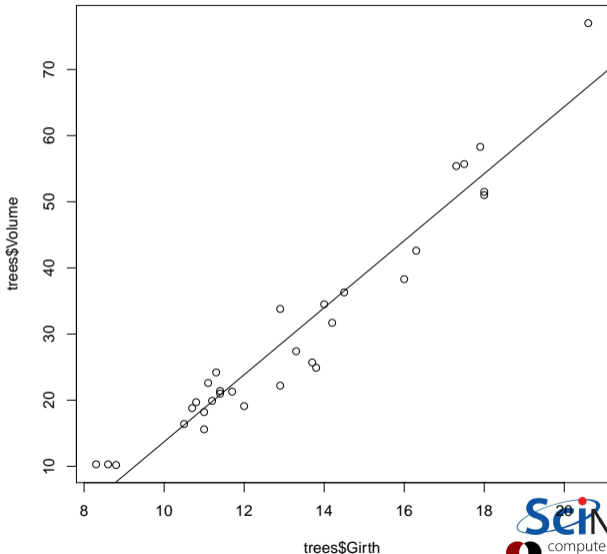
If these conditions are not met your model is not on a good statistical foundation.

Fitting a linear model, continued even more

It's always good to visualize your model once it's been made.

```
>
> model <- lm(Volume ~ Girth,
+           data = trees)
>
> plot(trees$Girth, trees$Volume)
> abline(model)
>
```

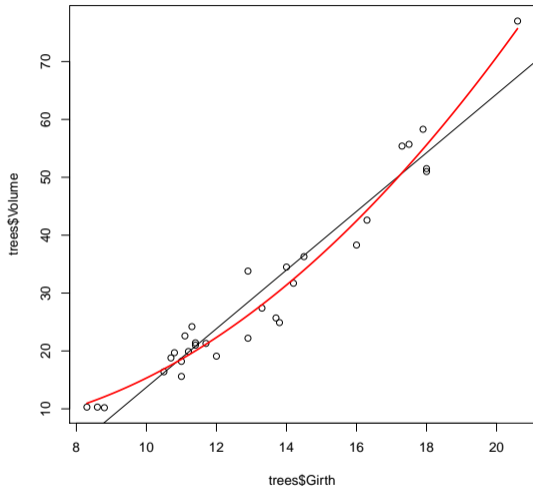
Not bad, but could be better.



Fitting a quadratic model

We can increase the order of the polynomial which we are fitting to the data.

```
> Girth2 <- trees$Girth**2
>
> model2 <- lm(Volume ~ Girth + Girth2,
+             data = trees)
>
> plot(trees$Girth, trees$Volume)
> abline(model)
>
> xx <- seq(min(trees$Girth), max(trees$Girth),
+          len = 100)
> yy <- model2$coef %*% rbind(1, xx, xx * xx)
>
> lines(xx, yy, lwd = 2, col = "red")
>
```



The "abline" command only works with linear fits.

Vector-matrix multiplication, an aside

What's up with that `%*%` symbol? That's the matrix-multiplication operator.

- Scalar-matrix multiplication gives element-by-element multiplication.
- Vector-matrix multiplication requires a special operator to be done the usual way.

```
> A <- matrix(1:9, nrow = 3)
> A
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> b <- 1:3
> b
[1] 1, 2, 3
> b %*% A
      [,1] [,2] [,3]
[1,]   14   32   50
```

$$\begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} * \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} b_1 * a_{11} + b_2 * a_{21} + b_3 * a_{31} \\ b_1 * a_{12} + b_2 * a_{22} + b_3 * a_{32} \\ b_1 * a_{13} + b_2 * a_{23} + b_3 * a_{33} \end{bmatrix}^T$$



Fitting a quadratic model, continued

What's up with that 'rbind' command?

The 'rbind' command ("row bind") will add rows to a data frame or array. Because there's only a single value in the first entry it's just repeated to fill in the row.

The 'seq' command ("sequence") will return a vector of values which range from the starting value to the ending value.

```
> xx <- seq(min(trees$Girth), max(trees$Girth), len = 100)
>
> rbind(1)
      [,1]
[1,]    1
>
> rbind(1, xx, xx * xx)
      [,1]      [,2]      [,3]      [,4]      [,5] ...
      1.00  1.000000  1.000000  1.000000  1.000000 ...
xx  8.30  8.424242  8.548485  8.672727  8.79697 ...
      68.89 70.967860 73.076593 75.216198 77.38668 ...
>
> class(rbind(1, xx, xx * xx))
[1] "matrix" "array"
>
> dim(rbind(1, xx, xx * xx))
[1] 3 100
```

Vector-matrix multiplication, an aside

We do vector-matrix multiplication to calculate the predicted values for the model at the locations of xx . This gives us the predicted values

$$yy = a * 1 + b * xx + c * xx^2$$

```
> coef(model2)
(Intercept)      Girth      Girth2
 10.7862655   -2.0921396    0.2545376
-----
> rbind(1, xx, xx * xx)
  [,1]      [,2]      [,3]      [,4] ...
  1.00  1.000000  1.000000  1.000000 ...
xx  8.30  8.424242  8.548485  8.672727 ...
 68.89 70.967860 73.076593 75.216198 ...
-----
> yy <- model2$coef %*% rbind(1, xx, xx * xx)
```

We do vector-matrix multiplication between the model coefficients and the locations of our desired predicted values:

$$\begin{bmatrix} 10.79 & -2.09 & 0.25 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & \dots \\ 8.30 & 8.42 & 8.55 & \dots \\ 68.89 & 70.97 & 73.08 & \dots \end{bmatrix} = \begin{bmatrix} 10.96 & 11.23 & 11.50 & \dots \end{bmatrix}$$

R is column major, another aside

All arrays are stored linearly in memory. A 1D array is linear in memory, but so is a 2D array.

Did you notice how the matrix A was created? Did you notice how there's an ambiguity in how the matrix could be created?

R is "column major". This means that its memory is organized by columns. Other column major languages are Fortran and Matlab.

This is contrast to "row major" languages (C, C++, Python), which organize their arrays by row.

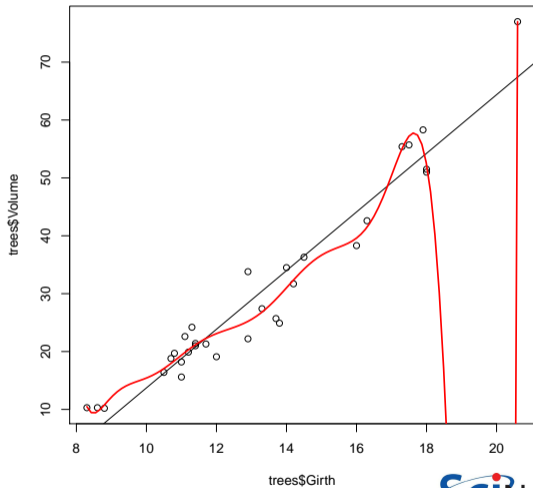
```
>
> A <- matrix(1:9, nrow = 3)
>
> A
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
>
```

This is important to remember if you convert vectors into arrays.

Fitting a higher-order model

There are several ways to have a higher-order fit. The best one is to use 'poly'.

```
>
> model10 <- lm(Volume ~ poly(Girth, 10),
+   data = trees)
>
> plot(trees$Girth, trees$Volume)
> abline(model)
>
> p.model10 <- predict(model10,
+   data.frame(Girth = xx))
>
> lines(xx, p.model10, lwd = 2, col = "red")
>
```



What's up with 'poly'?

We've seen that one way to do a non-linear fit is to use the commands

```
> xsq <- x * x  
-----  
> model3 <- lm(y ~ x + xsq)
```

or it can be written as

```
> model3 <- lm(y ~ x + I(x**2) + I(x**3))
```

Unfortunately, when used this way, x , x^2 and x^3 will be correlated with each other. This can lead to resolution problems, especially at higher orders.

The 'poly' function fixes this by generating a set of orthogonal polynomials evaluated at 'x'.

Summary

We've started looking at data, and fitting it. Things to remember:

- Plot your data!
- The standard error can be used to estimate confidence intervals for calculated quantities.
- Start with `lm`, both for linear and other polynomial fits.
- The summary function can give both confidence intervals on coefficient values, as well as information on the null hypothesis that the coefficient is zero.

We will examine model fitting further next class.