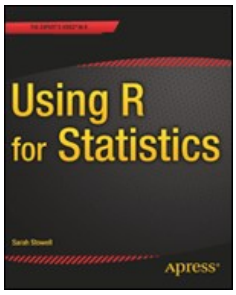


Using R for Statistics

by Sarah Stowell

Chapter 11: Regression and General Linear Models

Overview

Model building helps you to understand the relationships between variables and to make predictions about future observations. This chapter explains how to build regression models and other models in the general linear model family.

You will learn how to:

- build simple linear regression, multiple linear regression and polynomial regression models
- include interaction terms, transformed variables, and factor variables in a model
- add or remove terms from an existing model
- perform the stepwise, forward, and backward model selection procedures
- assess how well a model fits the data
- interpret model coefficients
- represent a model graphically with a line or curve of best fit
- check the validity of a model using diagnostics such as the residuals, deviance, and Cook's distances
- use your model to make predictions about new data

This chapter uses the `trees` dataset (included with R) and the `powerplant`, `concrete` and `people2` datasets (which are available with the downloads for this book and described in Appendix C).

General Linear Models

A general linear model is used to predict the value of a continuous variable (known as the *response* variable) from one or more explanatory variables. A general linear model takes the form:

$$y=\beta_0+\beta_1x_1+\beta_2x_2+\ldots\ \ldots+\beta_nx_n+\varepsilon$$

where y is the response variable, x_i are the explanatory variables, β_i are coefficients to be estimated and ε represents the random error.

The explanatory variables can be either continuous or categorical, and they can include cross products, polynomials and transformations of other variables.

The random errors are assumed to be independent, to follow a normal distribution with a mean of zero, and to have the same variance for all values of the explanatory variables.

Simple linear regression, multiple linear regression, polynomial regression, analysis of variance, two-way analysis of variance, analysis of covariance, and experimental design models are all types of general linear model.

Building the Model

You can build a regression model or general linear model with either the `lm` function or the g`lm` function. When used for this purpose, these functions do the same thing and give similar output. In this book we will use the `lm` function for building general linear models, but be aware that you may also see the g`lm` function in use.

Simple Linear Regression

To build a simple linear regression model with an explanatory variable named `var1` and a response variable named `resp`, use the command:

```
> lm(resp~var1, dataset)
```

You don't need to specify an intercept term (or constant term) in your model because R includes one automatically. To build a model without an intercept term, use the command:

```
> lm(resp~-1+var1, dataset)
```

When you build a model with the `lm` function, R displays the coefficient estimates. However, there are more components to the output that are not displayed, such as summary statistics, residuals, and fitted values. You can save the all of the output to an object, as shown here. Later in the chapter, you will learn how to access the various components of the model output.

```
> modelname<-lm(resp~var1, dataset)
```

Example 11-1: Simple Linear Regression Using the Tres Data

In the "Pearson's Correlation Coefficient" section in Chapter 5, we saw that there is a correlation between tree girth and tree volume. Suppose that you want to build a simple linear regression model to predict a tree's volume from its girth. To build the model, use the command:

```
> lm(Volume~Girth, trees)
```

```
   Call:
   lm(formula = Volume ~ Girth, data = trees)

   Coefficients:
   (Intercept)        Girth
       -36.943        5.066
```

From the output, you can see that the model formula is:

Volume=-36.943+5.066×Girth

This means that a tree with a girth of 10 inches has an expected volume of –36.943+5.066×10=13.717 cubic feet. For every additional inch of girth, the expected volume increases by 5.066 cubic feet.

To save the model output as an object, use the command:

```
> treemodel<-lm(Volume~Girth, trees)
```

The `treemodel` object will be used later in the chapter.

Multiple Linear Regression

To include several explanatory variables in a model, separate them with the plus sign:

```
> modelname<-lm(resp~var1+var2+var3, dataset)
```

Example 11-2: Multiple Linear Regression Using the Powerplant Data

Consider the `powerplant` dataset, which is described in Appendix C and is available with the downloads for this book. The dataset has three variables. The `Output` variable gives the output of a gas electrical turbine in megawatts. The `Pressure` and `Temp` variables give temperature and pressure measurements inside the turbine.

To build a multiple linear regression model that predicts the output from the pressure and temperature, use the command:

```
> lm(Output~Pressure+Temp, powerplant)
```

```
   Call:
   lm(formula = Output ~ Pressure + Temp, data = powerplant)

   Coefficients:
   (Intercept)     Pressure          Temp
      -32.8620       0.1858       -0.9916
```

From the output you can see that the model is:

Output = -32.8620 + 0.1858×Pressure - 0.9916×Temperature

This tells us that for an increase in pressure of one millibar, the expected output of the turbine increases by 0.1858 megawatts. For an increase in temperature of one centigrade, the expected output decreases by 0.9916 megawatts.

Interaction Terms

To add an interaction term to a model, use a colon (`:`). For example, `var1:var2` denotes the interaction between the two variables `var1` and `var2`. This command builds a model with terms for two variables and their interaction:

```
> modelname<-lm(resp~var1+var2+var1:var2, dataset)
```

You can also use a colon to express third-order and higher interactions:

```
> modelname<-lm(resp~var1+var2+var3+var1:var2+var1:var3+var2:var3+var1:var2:var3, dataset)
```

As you can see, this notation becomes lengthy for models with many variables. R has two useful shorthand notations for interaction terms, which are the asterisk (*) notation and the hat (^) notation.

Use the asterisk notation to include a group of variables and all their possible interactions. For example, the command:

```
> modelname<-lm(resp~var1*var2*var3, dataset)
```

builds a model with terms for the three variable main effects (`var1`, `var2`, `var3`), the three second-order interactions (`var1:var2`, `var1:var3`, `var2:var3`), and the third-order interaction (`var1:var2:var3`). This is equivalent to the previous very lengthy formula.

The hat notation includes a set of variables as well as all the possible interactions up to a given order. For example, to include all main effects and second-order interactions (but not the third-order interaction), use the command:

```
> modelname<-lm(resp~(var1+var2+var3)^2, dataset)
```

This is equivalent to this command:

```
> modelname<-lm(resp~var1+var2+var3+var1:var2+var1:var3+var1:var2, dataset)
```

Example 11-3: Factorial Experiment Using the Concrete Data

Consider the `concrete` dataset, which is shown in Figure 11-1 and available from the website. It gives the results of an experiment to determine the effect of cement type (I or II), additive type (A or B), and additive dose (0.3%, 0.4% or 0.5%) on the density of concrete.

	Cement	Additive	Additive.Dose	Density
1	I	A	0.003	2.43
2	II	A	0.003	2.431
3	I	B	0.003	2.43
4	II	B	0.003	2.418
5	I	A	0.004	2.419
6	II	A	0.004	2.436
7	I	B	0.004	2.435
8	II	B	0.004	2.414
9	I	A	0.005	2.419
10	II	A	0.005	2.425
11	I	B	0.005	2.422
12	II	B	0.005	2.41

Figure 11-1: The `concrete` dataset (see Appendix C for more details)

To build a model to predict density that includes terms for all of the explanatory variables (cement type, additive type, and additive dose) and their interactions (second and third order), use the command:

```
> concmodel<-lm(Density~Cement*Additive*Additive.Dose, concrete)
```

The `concmodel` object will be used later in this chapter.

Polynomial Terms

To build a polynomial regression model with terms for `var1`, $\texttt{var1}^2$, and $\texttt{var1}^3$, use the command:

```
> modelname<-lm(resp~var1+I(var1^2)+I(var1^3), dataset)
```

Notice that the terms `var1^2` and `var1^3` are nested inside an `I()`. This is because the symbols `^`, `*` and `+` have special meanings when used in a model formula, which can be confused with their usual arithmetic meanings of power, multiplication and addition. If you want to use these symbols for their usual arithmetic meanings, you must nest them inside the `I` function.

Example 11-4: Polynomial Regression Using the Tres Dataset

To build a model to predict tree volume that has terms for girth and girth^2, use the command:

```
> lm(Volume~Girth+I(Girth^2), trees)
```

```
Call:
lm(formula = Volume ~ Girth + I(Girth^2), data = trees)

Coefficients:
(Intercept)        Girth    I(Girth^2)
    10.7863      -2.0921        0.2545
```

From the result, you can see that the model formula is:

Volume = 10.7863 - 2.0921×Girth + 0.2545×Girth^2

To save the model as an object, use the command:

```
> polytrees<-lm(Volume~Girth+I(Girth^2), trees)
```

The `polytrees` object will be used later in this chapter.

Transformations

Simple variable transformations such as the log or square root transformations can be applied to the response or explanatory variables directly in the formula using the relevant function:

```
> modelname<-lm(log(resp)~var1, dataset)
```

For transformations that use the asterisk, hat, or plus symbols, nest them inside the `I` function:

```
> modelname<-lm(I(resp^2)~var1, dataset)
```

The Intercept Term

There may be occasions when you want to build a model without an intercept term. To do this, add `-1` as a term in the model, as shown here. This tells R not to include an intercept term, which would otherwise be included automatically:

```
> modelname<-lm(resp~-1+var1+var2+var3, dataset)
```

You can also create a model that contains only the intercept term, known as the *null model*:

```
> modelname<-lm(resp~1, dataset)
```

Including Factor Variables

You can include categorical variables in your model in the same way as continuous variables. Before including a categorical variable, use the `class` function to check that it has the `factor` variable class, as explained in Chapter 3 under "Working with Factor Variables."

When you include a factor variable in a model, R treats the first level of the factor as the reference level. So for a factor with n level, the model will have n-1 coefficients that express the effect of the remaining n-1 levels relative to the reference level.

To check which is the first level for a factor variable, use the `levels` function:

```
> levels(dataset$variable)
```

You can change the reference level for a factor variable with the `relevel` function:

```
> dataset$variable<-relevel(dataset$variable, "reflevel")
```

It is possible to change the way that R uses the coefficients to express the effect of the factor by changing the *contrasts* for the factor variable. Every factor variable has a set of contrasts associated with it, which you can view and change with the `contrasts` function. Enter `help(contrasts)` for more details on how to change the contrasts for a factor variable, and enter `help(contr.treatment)` for a list of contrast options.

Example 11-5: Model with Factor Variable, Using the People2 Data Set

Suppose that you want to use the `people2` dataset to build a model to predict a person's height from their hand span and eye color.

You can build the model with the command:

```
> lm(Height~Hand.Span+Eye.Color, people2)
```

```
Call:
lm(formula = Height ~ Hand.Span + Eye.Color, data = people2)

Coefficients:
```

```
      (Intercept)        Hand.Span    Eye.ColorBrown   Eye.ColorGreen
          82.8902           0.4456           -3.6233          -4.1924
```

From the output, we can see that formula has two coefficients that express the effect of brown and green eyes on height relative to blue eyes. So for people with blue eyes, the model formula is:

Height = 82.8902+0.4456×Hand Span

The formula for people with brown eyes is:

Height = 82.8902 + 0.4456×Hand Span -3.6233

The formula for people with green eyes is:

Height = 82.8902 + 0.4456×Hand Span -4.1924

Updating a Model

Once you have built a model, you may want to add or remove a term from the model to see how the new model compares with the previous one. The `update` function allows you build a new model by adding or removing terms from an existing model. This is useful when working with models that have many terms, as it means that you don't have to retype the entire model specification every time you add or remove a term.

Suppose that you have built a model and saved it to an object named `model1`:

```
> model1<-lm(resp~var1+var2+var3+var4, dataset)
```

To create a new model named `model2` by adding an additional term to `model1` (such as the interaction `var1:var2`), use the command:

```
> model2<-update(model1, ~.+var1:var2)
```

Similarly, you can remove a term from the model:

```
> model2<-update(model1, ~.-var4)
```

To check that the new model has the formula you expect, use the `formula` function:

```
> formula(model2)
```

Example 11-6: Updating the Concmodel Model

In Example 11-3, we built the `concmodel` model with the command:

```
> concmodel<-lm(Density~Cement*Additive*Additive.Dose, concrete)
```

To create a new model by removing the three-way interaction from the `concmodel` model, use the command:

```
> concmodel2<-update(concmodel, ~.-Cement:Additive:Additive.Dose)
```

Check the model formula for the new model with the command:

```
> formula(concmodel2)
```

```
Density ~ Cement + Additive + Additive.Dose + Cement:Additive +
    Cement:Additive.Dose + Additive:Additive.Dose
```

You can see that the three-way interaction (`Cement:Additive:Additive.Dose`) has now been removed from the model.

Stepwise Model Selection Procedures

Stepwise model selection procedures are algorithms designed to simplify the process of finding a model that explains a large amount of variation while including as few terms as possible. They are useful when dealing which large models with many potential terms. Popular stepwise selection procedures include forward selection, backward elimination, and general stepwise selection.

The `step` function allows you to perform forward, backward, and stepwise model selection in R. The function takes an `lm` or `glm` model object as input, which should be the full model from which you want to select a subset of terms.

Suppose that you have created a large model such as the one shown here, which includes a total of fifteen terms: four main effects, six second-order interactions, four third-order interactions, and one fourth-order interaction:

```
> model1<-lm(resp~var1*var2*var3*var4, dataset)
```

Once you have created the model, perform the stepwise selection procedure with the command:

```
> model2<-step(model1)
```

By default, the `step` function uses the general stepwise selection method. To use the backward or forward methods, set the `direction` argument to `"backward"` or `"forward"`:

```
> model2<-step(model1, direction="backward")
```

The newly created model object can be used in just the same way as any other model object. To see which terms have been kept in the new model, use the `formula` function:

```
> formula(model2)
```

Example 11-7: Stepwise Selection Using the Concrete Data

In Example 11.3, we built the `concmod` model with the command:

```
> concmodel<-lm(Density~Cement*Additive*Additive.Dose, concrete)
```

To perform the stepwise selection procedure on this model, use the command:

```
> concmodel3<-step(concmodel)
```

To view the formula of the resulting model, use the command:

```
> formula(concmodel3)
```

```
Density ~ Cement + Additive + Additive.Dose + Cement:Additive
```

From the output, you can see that the stepwise selection procedure has removed the three-way interaction and two of the two-way interaction terms from the model, leaving the main effects of cement type, additive type, and additive dose, and the interaction between cement type and additive type.

Assessing the Fit of the Model

Before interpreting and using your model, you will need to determine whether it is a good fit to the data and includes a good combination of explanatory variables. You may also be considering several alternative models for your data and want to compare them.

Model Fit

The fit of a model is commonly measured in a few different ways. These include:

Coefficient of determination (R^2) gives an indication of how well the model is likely to predict future observations. It measures the portion of the total variation in the data that the model is able to explain. It takes values between 0 and 1. A value close to 1 suggests that the model will give good predictions, while a value close to 0 suggests that the model will make poor predictions.

Adjusted R-squared is similar to R^2, but makes an adjustment for the number of terms in the model.

Significance test for model coefficients tells you whether individual coefficient estimates are significantly different from 0, and hence whether the coefficients are contributing to the model. Consider removing coefficients with p-values greater than 0.05.

F-test tells you whether the model is significantly better at predicting compared with using the overall mean value as a prediction. For good models, the p-value will be less than 0.05. An F-test can also be used to compare two models. In this case, a p-value less than 0.05 tells you that the more complex model is significantly better than the simpler model.

To view some summary statistics for a model, use the `summary` function:

```
> summary(lmobject)
```

The `summary` function displays:

- the model formula
- quantiles for the residuals
- coefficient estimates with the standard error and a significance test for each
- the residual standard error and degrees of freedom
- the R^2 (multiple and adjusted)
- an F-test for model fit

Note that if you built your model with the g`lm` function instead of the `lm` function, the output will be slightly different and will use the generalized linear model terminology.

To view an ANOVA table for the model, which shows the calculations behind the F-test, use the `anova` function:

```
> anova(lmobject)
```

You can also use the `anova` function to perform an F-test to compare a more complex model to a simpler model (the order of the models is not important):

```
> anova(model1, model2)
```

This only works for nested models. That is, all of the terms in the simpler model must also be found in the more complex model.

Example 11-8: Model Summary Statistics for the Tremodel Model

In Example 11-1, we created a simple linear regression model to predict tree volume from tree girth and saved the output to an object named `treemodel`.

To view summary statistics for the model, use the command:

```
> summary(treemodel)
```

This gives the following output:

```
   Call:
   lm(formula = Volume ~ Girth, data = trees)

   Residuals:
      Min      1Q  Median      3Q     Max
   -8.065 -3.107  0.152   3.495   9.587

   Coefficients:
               Estimate Std. Error t value Pr(>|t|)
   (Intercept) -36.9435     3.3651 -10.98 7.62e-12 ***
   Girth         5.0659     0.2474  20.48 < 2e-16 ***
   ---
   Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

   Residual standard error: 4.252 on 29 degrees of freedom
   Multiple R-squared: 0.9353,     Adjusted R-squared: 0.9331
   F-statistic: 419.4 on 1 and 29 DF, p-value: < 2.2e-16
```

The R^2 value of 0.9353 tells us that the model explains approximately 94% of the variation in tree volume. This suggests the model would be very good at predicting tree volume.

The hypothesis tests for the model coefficients tell us that the intercept and girth coefficients are significantly different from 0.

The p-value for the F-test is less than 0.05, which tells us that the model explains a significant amount of the variation in tree volume.

To view the ANOVA table, use the command:

```
> anova(treemodel)
```

```
Analysis of Variance Table

Response: Volume
          Df Sum Sq Mean Sq F value   Pr(>F)
Girth      1 7581.8 7581.8 419.36 < 2.2e-16 ***
Residuals 29 524.3     18.1
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The output shows the calculations behind the F-test.

Example 11-9: Comparing the Tremodel and Polytres Models

Suppose that you want to compare `treemodel` (the simple linear regression model created in Example 11-1) and `polytrees` (the polynomial regression model created in Example 11-4) to determine whether the additional polynomial term in the `polytrees` model

significantly improves the fit of the model.

To perform an F-test to compare the two models, enter the command:

```
> anova(treemodel, polytrees)
```

```
Analysis of Variance Table

Model 1: Volume ~ Girth
Model 2: Volume ~ Girth + I(Girth^2)
  Res.Df    RSS Df Sum of Sq      F   Pr(>F)
1     29 524.30
2     28 311.38  1    212.92 19.146 0.0001524 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

From the results, you can see that the p-value of 0.0001524 is less than 0.05. This means that the `polytrees` model fits the data significantly better than the `treemodel` model.

Coefficient Estimates

Coefficient estimates (or parameter estimates) are provided with the summary output, but you can also view them with the `coef` function, or the identical `coefficients` function:

```
> coef(lmobject)
```

view confidence intervals for the coefficient estimates, use the `confint` function:

```
> confint(lmobject)
```

The function gives 95% intervals by default, but you can adjust this with the `level` argument:

```
> confint(lmobject, level=0.99)
```

Example 11-10: Coefficients Estimates for the Tremodel Model

To view coefficient estimates for the `treemodel`, use the command:

```
> coef(treemodel)
```

```
(Intercept)          Girth
 -36.943459       5.065856
```

To view confidence intervals for the coefficient estimates, use the command:

```
> confint(treemodel)
```

```
                2.5 %     97.5 %
(Intercept) -43.825953 -30.060965
Girth         4.559914   5.571799
```

From the output, you can see that the 95% confidence interval for the intercept is –43.83 to –30.06, and the 95% confidence interval for the `Girth` coefficient is 4.56 to 5.57.

Plotting the Line of Best Fit

For simple linear regression models, you can create a scatter plot with a line of best fit superimposed over the data to help visualize the model. Use the `plot` and `abline` functions:

```
> plot(resp~var1, dataset)
> abline(coef(lmobject))
```

See the "Scatter Plots" section in Chapter 8 for more details about creating scatter plots with the `plot` function, and the "Adding straight Lines" section in Chapter 9 for more about adding straight lines with the `abline` function.

For polynomial regression models, you can superimpose a polynomial curve over the data. Use the `curve` function (see "Plotting a Function" in Chapter 8).

Example 11-11: Scatter Plot with Line of Best Fit for the Tremodel Model

To create a scatter plot with line of best fit for the `treemodel`, use the commands:

```
> plot(Volume~Girth, trees)
> abline(coef(treemodel))
```

The result is shown in Figure 11-2.

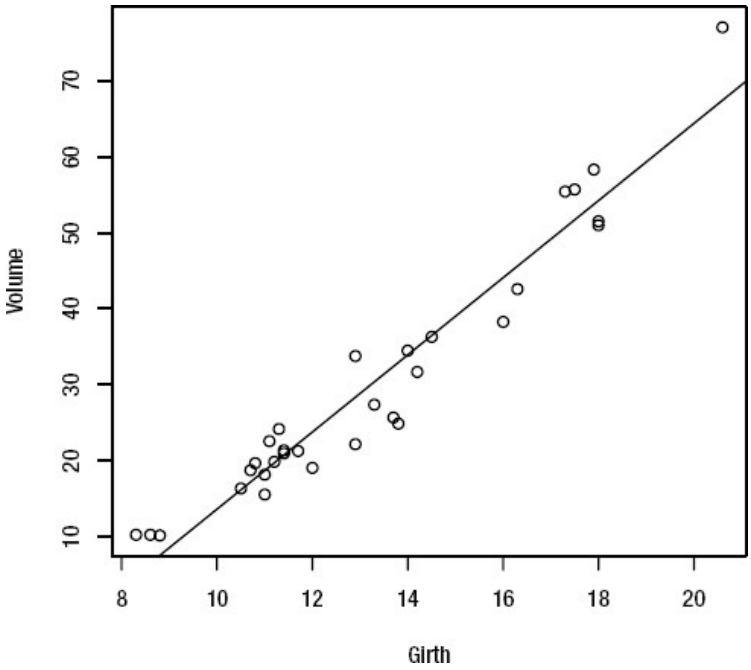


Figure 11-2: Scatter plots with simple linear regression model superimposed

Example 11-12: Scatter Plot with Polynomial Curve for the Polytres Model

In Example 11-4, we created a polynomial regression model named `polytrees` to predict tree volume from tree girth. To view the model coefficients for the `polytrees` model, use the `coef` function:

```
> coef(polytrees)
```

```
(Intercept)            Girth   I(Girth^2)
 10.7862655       -2.0921396    0.2545376
```

To create a scatter plot with curve superimposed, use the `curve` function:

```
> plot(Volume~Girth, trees)
> curve(10.7863-2.0921*x+0.2545*x^2, add=T)
```

Figure 11-3 shows the result.

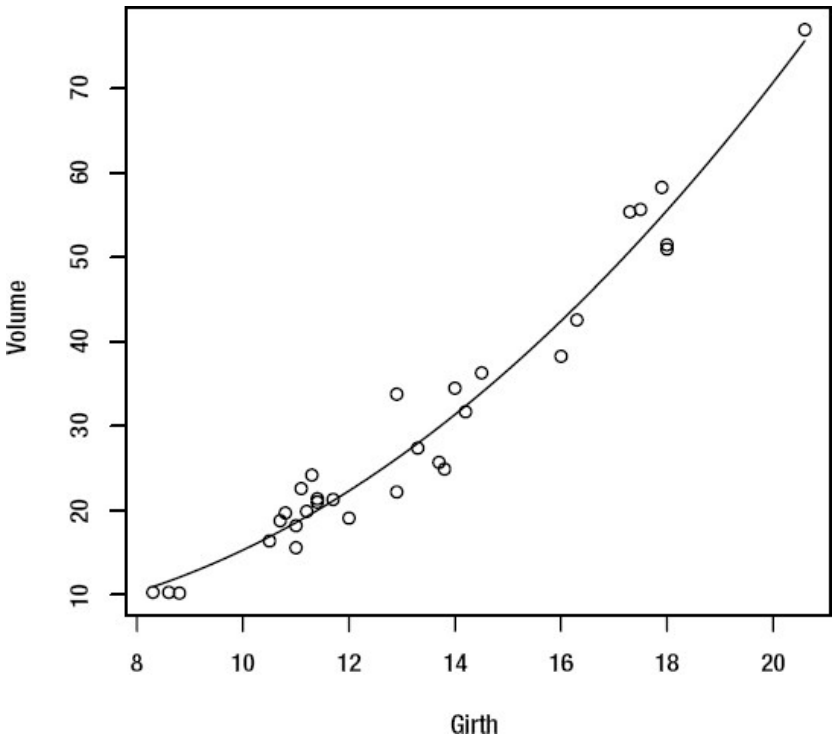


Figure 11-3: Scatter plots with polynomial regression model superimposed

---

Model Diagnostics

This section discusses the methods used to check the suitability of the model for the data and the reliability of the coefficient estimates. These include examining the model residuals and also measures of influence such as the leverage and Cook's distances for each of the observations.

Residual Analysis

The residuals for a given model are the set of differences between the observed values of the response variable, and the values predicted by the model (the fitted values). Standardized residuals and studentized residuals are types of residuals that have been adjusted to have a variance of one.

Examining the set of residuals for a model helps you to determine whether the model is appropriate. If the assumptions of a general linear model are met, then the residuals will be normally distributed and have constant variance. They will also be independent and will not follow any observable patterns. Residuals also help you to identify any observations that heavily influence the model.

To calculate raw residuals for a model, use the `residuals` function. There is also an identical function called `resid`. To calculate standardized residuals, use the `rstandard` function, and for studentized residuals, use the `rstudent` function:

```
> residuals(lmobject)
```

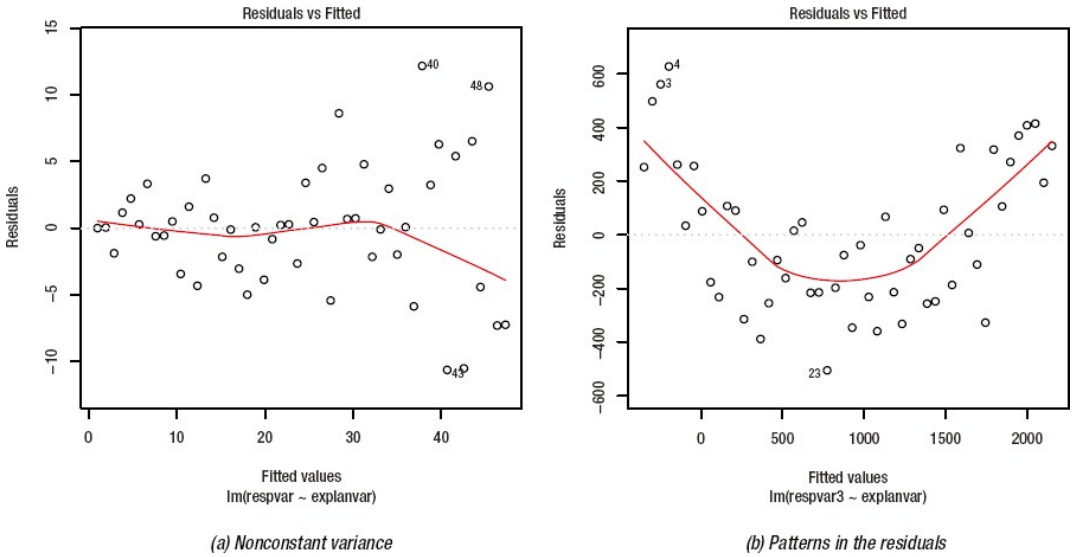
You can save the residuals to a new variable in your dataset, as shown here. This is useful if you want to plot the residuals against the response or explanatory variables:

```
> dataset$resids<-rstudent(lmobject)
```

Similarly, you can create a new variable containing fitted values with the `fitted` function:

```
> dataset$fittedvals<-fitted(lmobject)
```

Plotting the residuals is the easiest way to check for violations of the model assumptions. A normal probability plot or histogram of the residuals will help you to determine whether the residuals are normally distributed. Plots of the residuals against fitted values and residuals against explanatory values will allow you to check whether the variance is constant. If it is not, you will see a funnel shape like the one shown in in Figure 11-4a. These plots also help you to spot patterns in the residuals, like the one shown in Figure 11-4b.



(a) Nonconstant variance *(b) Patterns in the residuals*

Figure 11-4: Residuals plots warning of issues with the model

You can use the `plot` function to create residual plots for a model object:

```
> plot(lmobject, which=1)
```

Use the `which` argument to select from the following plots:

1. Residuals against fitted values
2. Normal probability plot of residuals
3. Scale-location plot
4. Residuals against leverage

I've omitted the numbers 4 and 6 which select plots of influence measures here, as you will learn about them in the next section.

Some other useful plots of residuals can be created:

- A histogram of residuals for checking the assumption of normality:
  ```
  > hist(dataset$resids)
  ```
- A plot of residuals against the response variable:
  ```
  > plot(resids~resp, dataset)
  ```
- A plot of the residuals against the explanatory variable:
  ```
  > plot(resids~var1, dataset)
  ```

Example 11-13: Residual Plots for the Tremodel Model

Suppose that you wish to analyze the residuals for the `treemodel` model, to check that the assumptions of the model are met.

First, save the studentized residuals to the `trees` dataset:

```
> trees$Resids<-rstudent(treemodel)
```

Next, set up the graphics device to display six plots, as explained in Chapter 9 in the "Multiple Plots in the Plotting Area" section:

```
> par(mfrow=c(3,2))
```

Next, use the relevant commands to create the following plots: histogram of residuals; normal probability plot of the residuals; residuals against fitted values; residuals against response (`Volume`); residuals against explanatory variable (`Girth`); residuals against other variable of interest (`Height`):

```
> hist(trees$Resids)
> plot(treemodel, which=2)
> plot(treemodel, which=1)
> plot(Resids~Volume, trees)
> plot(Resids~Girth, trees)
> plot(Resids~Height, trees)
```

Figure 11-5 shows the result.

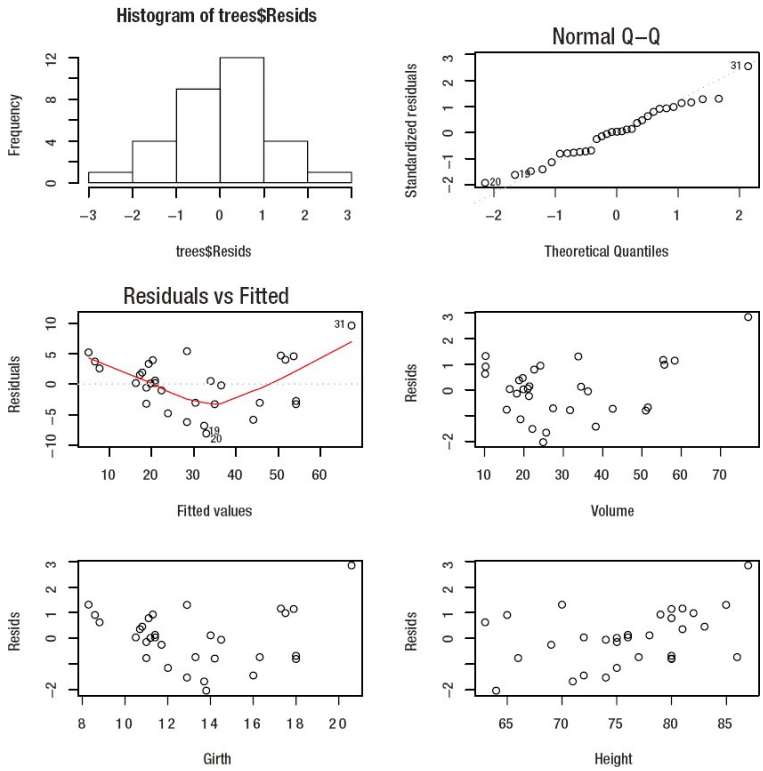


Figure 11-5: Residual plots for the `treemodel` model

From the histogram and normal probability plot, we can see that the residuals are approximately normally distributed.

In the plot of residuals against fitted values and the plot of residuals against girth, we can see that there is a slight pattern in the residuals. The residuals tend to be negative for trees with medium girth, and positive for trees with very small or very large girth. This suggests that adding polynomial terms to the model may improve the fit.

There are no obvious patterns in the plots of residual against volume and height. There are also no obvious outliers in any of the plots.

Leverage

The leverage helps to identify observations that have outlying values or unusual combinations for the explanatory variables. A large leverage value indicates that the observation may have a big influence on the model.

To calculate the leverage of each observation for a given model, use the `hatvalues` function:

```
> hatvalues(lmobject)
```

To create a plot of the residuals against the leverage, use the command:

```
> plot(lmobject, which=5)
```

To create a plot of the Cook's distances against the leverage, use the command:

```
> plot(lmobject, which=6)
```

Cook's Distances

The Cook's distance for an observation is a measure of how much the model parameters change if the observation is removed before estimation. Large values indicate that the observation has a big influence on the model.

To calculate the Cook's distances for a model, use the `cooks.distance` function:

```
> cooks.distance(lmobject)
```

You can create a plot of Cook's distance against observation number with the command:

```
> plot(lmobject, which=4)
```

and a plot of Cook's distance against leverage with the command:

```
> plot(lmobject, which=6)
```

Example 11-14: Leverage and Cook'S Distances for the Tremodel

Suppose that you want to create a set of plots for the `treemodel` regression model, to help identify any observations that may have a large influence on the model.

First set up the graphics device to hold four plots (see "Multiple Plots in the Plotting Area" in Chapter 9):

```
> par(mfrow=c(2,2))
```

Then create the three plots showing leverage and Cook's distances:

```
> plot(treemodel, which=4)
> plot(treemodel, which=5)
> plot(treemodel, which=6)
```

The result is shown in Figure 11-6.

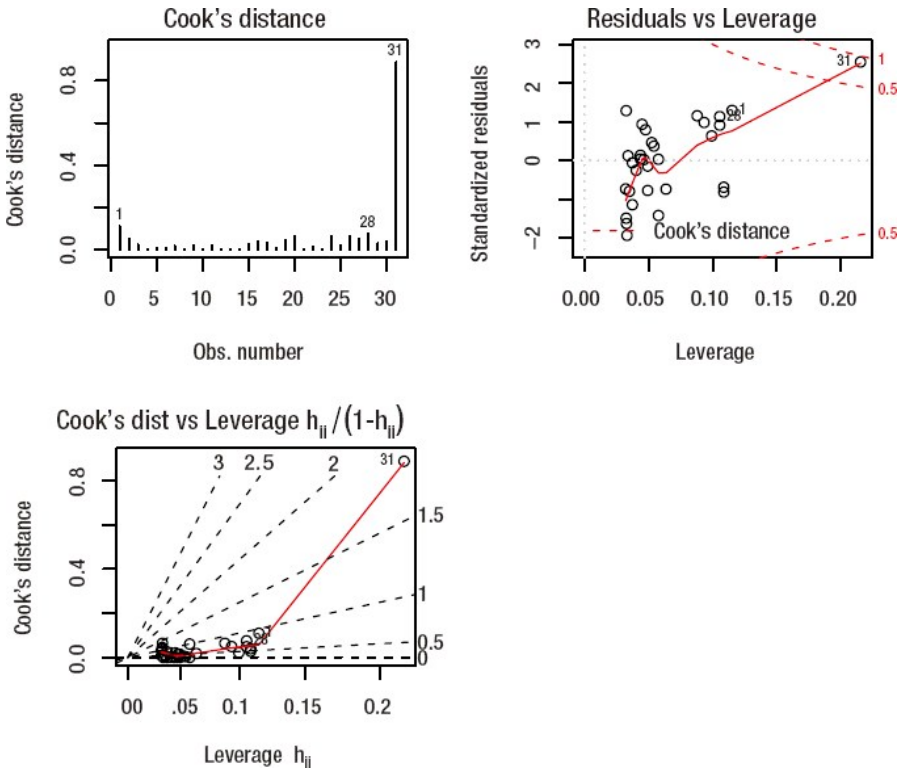


Figure 11-6: Influence plots for `treemodel` model

The contours on the second plot (residuals vs. leverage) divide areas with similar Cook's distances. The contours on the third plot (Cook's distance vs. leverage) divide areas with a similar standardized residual.

All three plots show that observation number 31 has a large Cook's distance and leverage, and also a fairly large standardized residual. This suggests that this observation has a big influence on the model and should be investigated further.

Making Predictions

Once you have built a model that you are happy with, you may want to use it to make predictions for new data. R has a convenient function for making predictions, called `predict`. To use this function, the new data must be arranged in a data frame (see "Entering Data Directly" in Chapter 2 for how to create data frames). The explanatory variables in the new dataset should be given identical names to those in the original dataset from which the model was built. It does not matter if the order of the variables is different, or if there are additional variables present.

Once your new data is arranged in a data frame, you can use the `predict` function:

```
> predict(lmobject, newdata)
```

The command creates a vector of predictions that correspond to the rows of the data frame. You can attach it to the data frame as a new variable:

```
> newdata$predictedvalues<-predict(lmobject, newdata)
```

You can also use the `predict` function to calculate confidence or prediction intervals for your predictions. Recall that confidence intervals only account for the uncertainty of the model estimation, while prediction interval also account for natural random variation in the response variable.

To calculate a confidence interval, set the `interval` argument to `"confidence"` and for a prediction interval, set it to `"prediction"`. You can adjust the size of the interval with the `level` argument:

```
> predict(lmobject, newdata, interval="confidence", level=0.99)
```

Example 11-15: Making Predictions Using Tremodel

Suppose that you want to use the `treemodel` regression model to estimate the volume of three trees with girths of 17.2, 12.0, and 11.4 inches.

First put the new data into a data frame:

```
> newtrees<-data.frame(Girth=c(17.2, 12.0, 11.4))
```

To make the predictions and add them to the `newtrees` data frame, use the command:

```
> newtrees$predictions<-predict(treemodel, newtrees, interval="prediction")
```

Then view the contents of the dataset:

```
> newtrees
```

```
     Girth predictions.fit predictions.lwr predictions.upr
1     17.2        50.18927        41.13045        59.24809
2     12.0        23.84682        14.98883        32.70481
3     11.4        20.80730        11.92251        29.69210
```

From the output, you can see that for a tree with a girth of 17.2 inches, the predicted volume is 50.2 cubic feet with a prediction interval of 41.1 to 59.2.

Summary

You should now be able to build a general linear model including factor variables, polynomial terms, and interaction terms where appropriate, and interpret the model coefficients. You should be able to add or remove terms from an existing model and apply the stepwise model selection procedure. You should be able to assess how well the model fits the data and use model diagnostics to determine whether the model is valid. For simple linear regression and polynomial regression models, you should be able to represent the model graphically. Finally, you should also be able to use your model to make predictions about future observations.

This table summarizes the main commands covered.

Task	Command
Build simple linear regression model	`lm(resp~var1},dataset)`
Build multiple regression model	`lm(resp~var1+var2+var3,dataset)`
Build model with interaction term	`lm(resp~var1+var2+var1:var2,dataset)`
Build model with interaction terms to a given order	`lm(resp~(var1+var2+var3)^2,dataset)`
Build factorial model	`lm(resp~var1*var2*var3,dataset)`
Build polynomial regression model	`lm(resp~var1+I(var1^2)+I(var1^3),dataset)`
Build model with log-transformed response variable	`lm(log(resp)~var1+var2+var3,dataset)`
Build model without intercept term	`lm(resp~-1+var1+var2+var3,dataset)`
Build null model	`lm(resp~1,dataset)`
Update a model	`update(lmobject, ~.+var4)`
Stepwise selection	`step(lmobject)`
Summarize a model	`summary(lmobject)`
Coefficient estimates	`coef(lmobject)`
Confidence interval for coefficient estimate	`confint(lmobject)`
Plot line of best fit	`plot(resp~var1,dataset) abline(coef(lmobject))`
Raw residuals	`residuals(lmobject)`
Standardized residuals	`rstandard(lmobject)`
Studentized residuals	`rstudent(lmobject)`
Fitted values	`fitted(lmobject)`

Residuals and influence plots	`plot(`*`lmobject`*`, which=1)`
Leverage	`hatvalues(`*`lmobject`*`)`
Cook's distances	`cooks.distance(`*`lmobject`*`)`
Predictions	`predict(`*`lmobject,newdata`*`)`

You have now covered everything you need to perform the most common types of statistical analysis. For the curious reader, Appendix A will explain how to access additional functionality with the use of add-on packages.