Computing Primer for Applied Linear Regression, Third Edition

Using R

Sanford Weisberg

University of Minnesota

School of Statistics

July 29, 2011

©2005–2011, Sanford Weisberg

0

Introduction

> The alr3 package for use with R was substantially rewritten in 2010. In particular: (1) nearly all function names have changed, and most have been moved to a package called car; (2) the instructions for installing alr3 have changed slightly to load install car and load it whenever alr3 is installed or loaded; (3) the new alr3 package works with R but not with S-Plus; the old version of alr3 continues to work with S-Plus.

This computer primer supplements the book *Applied Linear Regression* (ALR), third edition, by Sanford Weisberg, published by John Wiley & Sons in 2005. It shows you how to do the analyses discussed in ALR using one of several general-purpose programs that are widely available throughout the world. All the programs have capabilities well beyond the uses described here. Different programs are likely to suit different users. We expect to update the primer periodically, so check www.stat.umn.edu/alr to see if you have the most recent version. The versions are indicated by the date shown on the cover page of the primer.

Our purpose is largely limited to using the packages with ALR, and we will not attempt to provide a complete introduction to the packages. If you are new to the package you are using you will probably need additional reference material.

There are a number of methods discussed in ALR that are not a standard part of statistical analysis, and some methods are not possible without writing

your own programs to supplement the package you choose. *The exceptions to this rule is R, for which packages are available to do everything in the book.*

Here are the programs for which primers are available.

R is a *command line* statistical package, which means that the user types a statement requesting a computation or a graph, and it is executed immediately. You will be able to use a package of functions for R that will let you use all the methods discussed in ALR; we used R when writing the book.

R also has a programming language that allows automating repetitive tasks. R is a favorite program among academic statisticians because it is free, works on Windows, Linux/Unix and Macintosh, and can be used in a great variety of problems. There is also a large literature developing on using R for statistical problems. The main website for R is `www.r-project.org`. From this website you can get to the page for downloading R by clicking on the link for `CRAN`, or, in the US, going to `cran.us.r-project.org`.

Documentation is available for R on-line, from the website, and in several books. We can strongly recommend two books. The book *An R Companion to Applied Regression* by Fox and Weisberg (2011) provides a fairly gentle introduction to R with emphasis on regression. The *Companion* also provides a comprehensive treatment of a package called `car` that implements most of the ideas in ALR. A more comprehensive though somewhat dated introduction to R is Venables and Ripley (2002), and we will use the notation VR[3.1], for example, to refer to Section 3.1 of that book. Venables and Ripley has more computerese than does the *Companion*, but its coverage is greater. Other books on R include Verzani (2005), Maindonald and Braun (2002), Venables and Smith (2002), and Dalgaard (2002).

We used R Version 2.10.1 on Windows and Linux to write update this primer. A new version of R is released twice a year, so the version you use will probably be newer. If you have a fast internet connection, downloading and upgrading R is easy, and you should do it regularly.

SAS is the largest and most widely distributed statistical package in both industry and education. SAS also has a GUI. While it is possible to do *some* data analysis using the SAS GUI, the strength of this program is in the ability to write SAS programs, in the editor window, and then submit them for execution, with output returned in an output window. We will therefore view SAS as a *batch* system, and concentrate mostly on writing SAS commands to be executed. The website for SAS is `www.sas.com`.

SAS is very widely documented, including hundreds of books available through amazon.com or from the SAS Institute, and extensive on-line documentation. Muller and Fetterman (2003) is dedicated particularly

to regression. We used Version 9.1 for Windows. We find the on-line documentation that accompanies the program to be invaluable, although learning to read and understand SAS documentation isn't easy.

Although SAS is a programming language, adding new functionality can be very awkward and require long, confusing programs. These programs could, however, be turned into SAS *macros* that could be reused over and over, so in principle SAS could be made as useful as R. We have not done this, but would be delighted if readers would take on the challenge of writing macros for methods that are awkward with SAS. Anyone who takes this challenge can send us the results (sandy@stat.umn.edu) for inclusion in later revisions of the primer.

We have, however, prepared *script files* that give the programs that will produce all the output discussed in this primer; you can get the scripts from `www.stat.umn.edu/alr`.

JMP is another product of SAS Institute, and was designed around a clever and useful GUI. A student version of JMP is available. The website is `www.jmp.com`. We used JMP Version 5.1 on Windows.

Documentation for the student version of JMP, called JMP-In, comes with the book written by Sall, Creighton and Lehman (2005), and we will write JMP-START[3] for Chapter 3 of that book, or JMP-START[P360] for page 360. The full version of JMP includes very extensive manuals; the manuals are available on CD only with JMP-In. Fruend, Littell and Creighton (2003) discusses JMP specifically for regression.

JMP has a scripting language that could be used to add functionality to the program. We have little experience using it, and would be happy to hear from readers on their experience using the scripting language to extend JMP to use some of the methods discussed in ALR that are not possible in JMP without scripting.

SPSS evolved from a batch program to have a very extensive graphical user interface. In the primer we use only the GUI for SPSS, which limits the methods that are available. Like SAS, SPSS has many sophisticated tools for data base management. A student version is available. The website for SPSS is `www.spss.com`. SPSS offers hundreds of pages of documentation, including SPSS (2003), with Chapter 26 dedicated to regression models. In mid-2004, amazon.com listed more than two thousand books for which "SPSS" was a keyword. We used SPSS Version 12.0 for Windows. A newer version is available.

This is hardly an exhaustive list of programs that could be used for regression analysis. If your favorite package is missing, please take this as a challenge: try to figure out how to do what is suggested in the text, and write your own primer! Send us a PDF file (sandy@stat.umn.edu) and we will add it to our website, or link to yours.

One program missing from the list of programs for regression analysis is Microsoft's spreadsheet program Excel. While *a few* of the methods described in the book can be computed or graphed in Excel, most would require great endurance and patience on the part of the user. There are many add-on statistics programs for Excel, and one of these may be useful for comprehensive regression analysis; we don't know. If something works for you, please let us know!

A final package for regression that we should mention is called Arc. Like R, Arc is free software. It is available from `www.stat.umn.edu/arc`. Like JMP and SPSS it is based around a graphical user interface, so most computations are done via point-and-click. Arc also includes access to a complete computer language, although the language, lisp, is considerably harder to learn than the S or SAS languages. Arc includes all the methods described in the book. The use of Arc is described in Cook and Weisberg (1999), so we will not discuss it further here; see also Weisberg (2005).

0.1 ORGANIZATION OF THIS PRIMER

The primer often refers to specific problems or sections in ALR using notation like ALR[3.2] or ALR[A.5], for a reference to Section 3.2 or Appendix A.5, ALR[P3.1] for Problem 3.1, ALR[F1.1] for Figure 1.1, ALR[E2.6] for an equation and ALR[T2.1] for a table. Reference to, for example, "Figure 7.1," would refer to a figure in this primer, not to ALR. Chapters, sections, and homework problems are numbered in this primer as they are in ALR. Consequently, the section headings in primer refers to the material in ALR, and not necessarily the material in the primer. Many of the sections in this primer don't have any material because that section doesn't introduce any new issues with regard to computing. The index should help you navigate through the primer.

There are four versions of this primer, one for R, and one for each of the other packages. All versions are available for free as PDF files at `www.stat.umn.edu/alr`.

Anything you need to type into the program will always be in `this font`. Output from a program depends on the program, but should be clear from context. We will write File to suggest selecting the menu called "File," and Transform → Recode to suggest selecting an item called "Recode" from a menu called "Transform." You will sometimes need to push a button in a dialog, and we will write "push OK" to mean "click on the button marked 'OK'." For non-English versions of some of the programs, the menus may have different names, and we apologize in advance for any confusion this causes.

R Most of the graphs and computer output in ALR were produced with R. The computer code we give in this primer may not reproduce the graphs exactly, since we have tweaked some of the graphs to make them look prettier for publication, and the tweaking arguments work a little differently in R. If you want to see the tweaks we used in R, look at the scripts, Section 0.4.

Table 0.1 The data file `htwt.txt`.

```
Ht Wt
169.6 71.2
166.8 58.2
157.1 56
181.1 64.5
158.4 53
165.6 52.4
166.7 56.8
156.5 49.2
168.1 55.6
165.3 77.8
```

0.2 DATA FILES

0.2.1 Documentation

Documentation for nearly all of the data files is contained in ALR; look in the index for the first reference to a data file. Separate documentation can be found in the file `alr3data.pdf` in PDF format at the web site `www.stat.umn.edu/alr`.

The data are available in a *package* for R, in a *library* for SAS, and as a directory of files in special format for JMP and SPSS. In addition, the files are available as plain text files that can be used with these, or any other, program. Table 0.1 shows a copy of one of the smallest data files called `htwt.txt`, and described in ALR[P3.1]. This file has two variables, named *Ht* and *Wt*, and ten cases, or rows in the data file. The largest file is `wm5.txt` with 62,040 cases and 14 variables. This latter file is so large that it is handled differently from the others; see Section 0.2.5.

A few of the data files have missing values, and these are generally indicated in the file by a place-holder in the place of the missing value. For example, for R, the placeholder is `NA`, while for SAS it is a period "." Different programs handle missing values a little differently; we will discuss this further when we get to the first data set with a missing value in Section 4.5.

0.2.2 R data files and a package

> The instructions for downloading the `alr3` package have changed as of 2010. Do not use the Packages → Install package(s) menu item as it will not load all the needed packages at once.

All the data files are collected into an R *package* named `alr3`. In addition, many functions you will find useful for the topics of ALR are now in a different package called `car`. When you install `alr3` on your computer using the following command, `car` will be downloaded as well:

```
> install.packages("alr3", dependencies=TRUE)
```

Follow the on-screen instructions to select a mirror site close to your home, and then select `alr3` from the list of available packages. By setting `dependencies=TRUE` `car` and several other packages will be obtained and installed.

Once you have installed the package, all you need to access a data file is to load the package,

```
> library(alr3)
```

(or `library(alr3,lib.loc="mylib/")` if you installed your own Linux/Unix library). The `alr3` package uses something called "lazy loading" of data sets, so all the data sets are immediately available for your use. For example, simply typing

```
> forbes

    Temp Pressure Lpres
1  194.5    20.79 131.8
2  194.3    20.79 131.8
3  197.9    22.40 135.0
4  198.4    22.67 135.6
5  199.4    23.15 136.5
6  199.9    23.35 136.8
7  200.9    23.89 137.8
8  201.1    23.99 138.0
9  201.4    24.02 138.1
10 201.3    24.01 138.0
11 203.6    25.14 140.0
12 204.6    26.57 142.4
13 209.5    28.49 145.5
14 208.6    27.76 144.3
15 210.7    29.04 146.3
16 211.9    29.88 147.5
17 212.2    30.06 147.8
```

prints the `forbes` data set. Data files in the text are always named with an ending of ".txt" which is not used in the `alr3` package. You can add a new variable to the data frame by typing, for example

```
> forbes$logTemp <- log2(forbes$Temp)
```

to add the base-two logarithm of Temperature to the data frame. There are similar functions `log` for natural logs and `log10` for base-ten logarithms

You can view the documentation for the data sets on-line. The most elegant method is to enter the command `help.start()` into R. This will start your web browser, if it is not already running. Click on "Packages," and then on "alr3." You will then get an alphabetical listing of all the data files and functions in the alr3 package, and you can select the one you want to see.

0.2.3 Two files missing from the alr3 package

The two datafiles anscombe and longley are available, with those names, in the datasets library that is part of R. For anscombe the variable names are as described in the textbook on page 12. For longley the variable names are slightly different from the names given on page 94; type ?longley to get the variable names.

0.2.4 Getting the data in text files

You can download the data as a directory of plain text files, or as individual files; see www.stat.umn.edu/alr/data. *Missing values on these files are indicated with NA. If your program does not use this missing value character, you may need to substitute a different character using an editor.*

0.2.5 An exceptional file

The file wm5.txt is not included in any of the compressed files, or in the libraries. This one file is nearly five megabytes long, requiring as much space as all the other files combined. If you need this file, for ALR[P10.12], you can download it separately from www.stat.umn.edu/alr/data.

0.3 A COMPANION TO APPLIED REGRESSION

The book *An R Companion to Applied Regression* by Fox and Weisberg (2011) provides a description of the car package, including all the functions that were in the early versions of alr3 that have been renamed, improved and moved to car. This book provides a considerably more comprehensive introduction to R than is available in this primer. We will refer to the *Companion* with references like COMPANION[3.1] for Section 3.1. Throughout this *Primer* you will be using functions in car. When you load the alr3 package car is automatically loaded as well so you don't need to do anything special to use car. Sample chapters of the *Companion* are available from the Sage website, http://www.sagepub.com/books/Book233899.

0.4 SCRIPTS

For R and SAS, we have prepared *script files* that can be used while reading this primer. For R, the scripts will reproduce nearly every computation shown in ALR; indeed, these scripts were used to do the calculations in the first place. For SAS, the scripts correspond to the discussion given in this primer, but

will not reproduce everything in ALR. The scripts can be downloaded from `www.stat.umn.edu/alr` for R or SAS.

For R users, scripts can be obtained while you are running R and also connected to the internet. To get the script for Chapter 2 for this primer, for example, you could type

```
> library(alr3)
> alrWeb(pscript = "chapter2")
```

To get the script for Chapter 2 of the text, use

```
> alrWeb(script = "chapter2")
```

In either case, the script will open in a window in your web browser, and you can then save or copy-and-paste the script.

Although both JMP and SPSS have scripting or programming languages, we have not prepared scripts for these programs. Some of the methods discussed in ALR are not possible in these programs without the use of scripts, and so we encourage readers to write scripts in these languages that implement these ideas. Topics that require scripts include bootstrapping and computer intensive methods, ALR[4.6]; partial one-dimensional models, ALR[6.4], inverse response plots, ALR[7.1, 7.3], multivariate Box-Cox transformations, ALR[7.2], Yeo-Johnson transformations, ALR[7.4], and heteroscedasticity tests, ALR[8.3.2]. There are several other places where usability could be improved with a script.

If you write scripts you would like to share with others, let me know (sandy@stat.umn.edu) and I'll make a link to them or add them to the website.

0.5 THE VERY BASICS

Before you can begin doing any useful computing, you need to be able to read data into the program, and after you are done you need to be able to save and print output and graphs. All the programs are a little different in how they handle input and output, and we give some of the details here.

0.5.1 Reading a data file

Reading data into a program is surprisingly difficult. We have tried to ease this burden for you, at least when using the data files supplied with ALR, by providing the data in a special format for each of the programs. There will come a time when you want to analyze real data, and then you will need to be able to get your data into the program. Here are some hints on how to do it.

R (See also COMPANION[2.1]) If you have installed the `alr3` package and want to read one of the data files described in ALR, you can follow the in-

structions in Section 0.2.2. If you have not installed the library, or you want to read a different file, use the command `read.table` to read a plain data file. The general form of this command is:

```
> d <- read.table("filename", header=TRUE, na.strings="?")
```

The filename is a quoted string, like `"C:/My Documents/data.txt"`, giving the name of the data file and its path[1]. In place of the file name you can use

```
> d <- read.table(file.choose(), header=TRUE, na.strings="?")
```

in which case a standard file dialog will open and you can select the file you want in the usual way.

The argument `header=TRUE` indicates that the first line of the file has variable names (you would say `header=FALSE` if this were not so, and then the program would assign variable names like `X1`, `X2` and so on), and the `na.strings="?"` indicates that missing values, if any, are indicated by a question mark rather than the default of `NA` used by R. `read.table` has many more options; type `help(read.table)` to learn about them. R has a package called `foreign` that can be used to read files of other types.

Suppose that the file `C:\My Documents\mydata\htwt.txt` gave the data in Table 0.1. This file is read by

```
> d <- read.table("C:/My Documents/mydata/htwt.txt", header=TRUE)
```

With Windows, always replace the backslashes \ *by forward slashes* /. While this replacement may not be necessary in all versions of R, the forward slashes always work. The `na.strings` argument can be omitted because this file has no missing data. As a result of this command, a `data.frame`, roughly like a matrix, is created named `d`. The two columns of `d` are called `d$Ht` and `d$Wt`, with the column names read from the first row of the file because `header=TRUE`.

In R, you can also read the data directly from the internet:

```
> d <-read.table("http://www.stat.umn.edu/alr/data/htwt.txt",
+       header=TRUE)
```

You can get any data file in the book in this way by substituting the file's name, and using the rest of the web address shown. Of course this is unnecessary because all the data files are part of the `alr3` package.

R is *case sensitive*, which means that a variable called `weight` is different from `Weight`, which in turn is different from `WEIGHT`. Also, the command `read.table` would be different from `READ.TABLE`. Path names are case-sensitive on Linux, but not on Windows.

[1] Getting the path right can be frustrating. If you are using R, select File → Change dir, and then use BROWSE to select the directory that includes your data file. In `read.table` you can then specify just the file name without the path.

0.5.2 Reading Excel Files

(See also COMPANION[2.1.3]) R is less friendly in working with Excel files[2], and you should start in Excel by *saving the file as a ".csv" file*, which is a plain text file, with the items in each row of the file separated by commas. You can then read the ".csv" file with the command `read.cvs`,

```
> data <- read.csv("ais.csv", header=TRUE)
```

where once again the complete path to the file is required. There are tools described in COMPANION[2.1.3] for reading Excel spreadsheets directly.

0.5.3 Saving text output and graphs

All the programs have many ways of saving text output and graphs. We will make no attempt to be comprehensive here.

R The easiest way to save printed output is to select it, copy to the clipboard, and paste it into an editor or word processing program. *Be sure to use a monospaced-font like Courier so columns line up properly.* In R on Windows, you can select File → Save to file to save the contents of the text window to a file.

The easiest way to save a graph in R on Windows or Macintosh is via a menu item. The plot has its own menus, and select File → Save as → filetype, where `filetype` is the format of the graphics file you want to use. You can copy a graph to the clipboard with a menu item, and then paste it into a word processing document.

In all versions of R, you can also save files using a relatively complex method of defining a *device*, COMPANION[7.4], for the graph. For example,

```
> pdf("myhist.pdf", horizontal=FALSE, height=5, width=5)
> hist(rnorm(100))
> dev.off()
```

defines a device of type `pdf` to be saved in the file "myhist.pdf" in the current directory. It will consist of a histogram of 100 normal random numbers. This device remains active until the `dev.off` command closes the device. The default with `pdf` is to save the file in "landscape," or horizontal mode to fill the page. The argument `horizontal=FALSE` orients the graph vertically, and `height` and `width` to 5 makes the plotting region a five inches by five inches square.

R has many devices available, including `pdf`, `postscript`, `jpeg` and others; see `help(Devices)` for a list of devices, and then, for example, `help(postscript)` for a list of the (many) arguments to the `pdf` command.

[2]Experienced users can read about, and install the the dcom package for using R with Excel; see cran.r-project.org/contrib/extra/dcom/.

0.5.4 Normal, F, t and χ^2 tables

ALR does not include tables for looking up critical values and significance levels for standard distributions like the t, F and χ^2. Although these values can be computed with any of the programs we discuss in the primers, doing so is easy only with R. Also, the computation is fairly easy with Microsoft Excel. Table 0.2 shows the functions you need using Excel.

Table 0.2 Functions for computing p-values and critical values using Microsoft Excel. The definitions for these functions are not consistent, sometimes corresponding to two-tailed tests, sometimes giving upper tails, and sometimes lower tails. Read the definitions carefully. The algorithms used to compute probability functions in Excel are of dubious quality, but for the purpose of determining p-values or critical values, they should be adequate; see Knsel (2005) for more discussion.

Function	What it does
`normsinv(p)`	Returns a value q such that the area to the left of q for a standard normal random variable is p.
`normsdist(q)`	The area to the left of q. For example, `normsdist(1.96)` equals 0.975 to three decimals.
`tinv(p,df)`	Returns a value q such that the area to the left of $-\|q\|$ *and* the area to the right of $+\|q\|$ for a $t(\mathrm{df})$ distribution equals q. This gives the critical value for a two-tailed test.
`tdist(q,df,tails)`	Returns p, the area to the left of q for a $t(df)$ distribution if $tails = 1$, and returns the sum of the areas to the left of $-\|q\|$ and to the right of $+\|q\|$ if $tails = 2$, corresponding to a two-tailed test.
`finv(p,df1,df2)`	Returns a value q such that the area to the *right* of q on a $F(\mathrm{df}_1, \mathrm{df}_2)$ distribution is p. For example, `finv(.05,3,20)` returns the 95% point of the $F(3, 20)$ distribution.
`fdist(q,df1,df2)`	Returns p, the area to the *right* of q on a $F(\mathrm{df}_1, \mathrm{df}_2)$ distribution.
`chiinv(p,df)`	Returns a value q such that the area to the *right* of q on a $\chi^2(\mathrm{df})$ distribution is p.
`chidist(q,df)`	Returns p, the area to the *right* of q on a $\chi^2(\mathrm{df})$ distribution.

R Table 0.3 lists the six commands that are used to compute significance levels and critical values for t, F and χ^2 random variables. For example, to find the significance level for a test with value -2.51 that has a $t(17)$ distribution, type into the text window

```
> pt(-2.51,17)
[1] 0.01124
```

which returns the area to the *left* of the first argument. Thus the lower-tailed p-value is 0.011241, the upper tailed p-value is $1 - .012241 = .98876$, and the two-tailed p-value is $2 \times .011241 = .022482$.

Table 0.3 Functions for computing p-values and critical values using R. These functions may have additional arguments useful for other purposes.

Function	What it does
`qnorm(p)`	Returns a value q such that the area to the left of q for a standard normal random variable is p.
`pnorm(q)`	Returns a value p such that the area to the left of q on a standard normal is p.
`qt(p,df)`	Returns a value q such that the area to the left of q on a $t(\mathrm{df})$ distribution equals q.
`pt(q,df)`	Returns p, the area to the left of q for a $t(df)$ distribution
`qf(p,df1,df2)`	Returns a value q such that the area to the left of q on a $F(\mathrm{df}_1, \mathrm{df}_2)$ distribution is p. For example, `qf(.95,3,20)` returns the 95% points of the $F(3, 20)$ distribution.
`pf(q,df1,df2)`	Returns p, the area to the left of q on a $F(\mathrm{df}_1, \mathrm{df}_2)$ distribution.
`qchisq(p,df)`	Returns a value q such that the area to the left of q on a $\chi^2(\mathrm{df})$ distribution is p.
`pchisq(q,df)`	Returns p, the area to the left of q on a $\chi^2(\mathrm{df})$ distribution.

0.6 ABBREVIATIONS TO REMEMBER

ALR refers to the textbook, Weisberg (2005). COMPANION refers to Fox and Weisberg (2011), and VR refers to Venables and Ripley (2002), our primary reference for R. JMP-START refers to Sall, Creighton and Lehman (2005), the primary reference for JMP. Information typed by the user looks like `this`. References to menu items looks like File or Transform → Recode. The name of a BUTTON to push in a dialog uses this font.

0.7 PACKAGES FOR R

The `alr3` package described includes data and a few functions that are used in the primer. In addition when you use the `alr3` package you will automatically have the functions in the `car` package and the `MASS` package available without

the necessity of loading them. We will also make use of a few other packages that others have written that will be helpful for working through ALR. The packages are all described in Table 0.4. Since all the packages are free, we recommend that you obtain and install them immediately. In particular, instructors should be sure the packages are installed for their students using a Unix/Linux system where packages need to be installed by a superuser.

Table 0.4 R packages that are useful with ALR. All the R packages are available from CRAN, `cran.us.r-project.org`.

`alr3`	Contains all the data files used in the book, and a few additional functions that implement ideas in the book that are not already part of these programs. The R version is available from CRAN.
`MASS`	MASS is a companion to Venables and Ripley (2002), and is automatically loaded with `alr3`.
`car`	The companion to Fox and Weisberg (2011), and is automatically loaded with `alr3`.
`nlme`	This library fits linear and nonlinear mixed-effects models, which are briefly discussed in Section 6.5. This package is included with the base distribution of both R. This library is described by Pinheiro and Bates (2000).
`sm`	This is a library for local linear smoothing, and used only in Chapter 12. The R version is available from CRAN.

0.8 COPYRIGHT AND PRINTING THIS PRIMER

Copyright © 2005, 2011 by Sanford Weisberg. Permission is granted to download and print this primer. Bookstores, educational institutions, and instructors are granted permission to download and print this document for student use. Printed versions of this primer may be sold to students for cost plus a reasonable profit. The website reference for this primer is `www.stat.umn.edu/alr`. Newer versions may be available from time to time.

1 Scatterplots and Regression

1.1 SCATTERPLOTS

A principal tool in regression analysis is the two-dimensional scatterplot. All statistical packages can draw these plots. We concentrate mostly on the basics of drawing the plot. Most programs have options for modifying the appearance of the plot. For these, you should consult documentation for the program you are using.

R (See also COMPANION[1.1]) R scatterplots can be drawn using the function `plot`. A simple example is:

```
> library(alr3)
> plot(Dheight ~ Mheight, data=heights)
```

The data frame `heights` is loaded automatically as long as the `alr3` library has been loaded. The first argument to the `plot` function is a `formula`, with the vertical or y-axis variable to the left of the ~ and the horizontal or x-axis variable to the right. The second argument is the name of the data frame that includes the variables to be plotted. The resulting graph is shown in Figure 1.1 This same graph can be obtained with other sets of arguments to `plot`:

```
> plot(heights$Mheight, heights$Dheight)
> with(heights, plot(Mheight, Dheight))
```

The first of these doesn't require naming the data frame as an argument, but does require naming each variable completely including its data frame name.

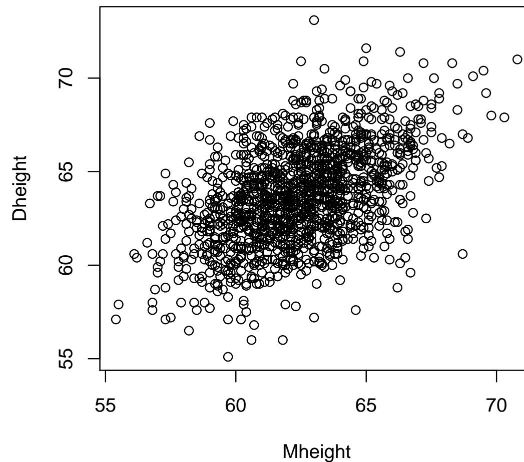


Fig. 1.1 A simple version of ALR[F1.1].

The second uses the `with` function (COMPANION[2.2.2]), to specify the data frame to be used in the following `plot` command.

Figure 1.1 doesn't match ALR[F1.1] exactly, but that can be done using some of the myriad optional arguments to `plot`.

```
> with(heights, plot(Mheight, Dheight, xlim=c(55, 75),
+                    ylim=c(55,75), pch=20))
```

The arguments `xlim` and `ylim` give the limits on the horizontal and vertical axes, respectively. In the example, both have been set to range from 55 to 75. The statement `c(55,75)` should be read as *collect the values 55 and 75 into a vector*. The argument `pch` selects the plotting character. Here we have use character 20, which is a small filled circle in R. VR[4.2] give a listing of all the characters available. You can also plot a letter, for example `pch="f"` would use a lower-case "f" as the plotting character. The argument `cex=.3` controls the size of the plotting symbol, here .3 times the default size. This graph has many points, so a very small symbol is desirable. VR[4.2] discusses most of the other settings for a scatterplot; we will introduce others as we use them. The documentation in R for the function `par` describes many of the arguments that control the appearance of a plot.

The figure ALR[F1.2] is obtained from ALR[F1.1] by deleting most of the points. Some packages may allow the user to remove/restore points interactively. While this is possible in R, see, for example, COMPANION[3.5], it

is reasonably tedious. You can simply redraw the figure, after selecting the points you want to appear:

```
> sel <- with(heights,
+   (57.5 < Mheight) & (Mheight <= 58.5) |
+   (62.5 < Mheight) & (Mheight <= 63.5) |
+   (67.5 < Mheight) & (Mheight <= 68.5))
> with(heights, plot(Mheight[sel], Dheight[sel], xlim=c(55,75),
+                    ylim=c(55,75), pty="s", pch=20, cex=.3,
+                    xlab="Mheight", ylab="Dheight"))
```

The variable `sel` that results from the first calculation is a vector of values equal to either TRUE and FALSE. It is equal to TRUE if either $57.5 < Mheight \leq 58.5$, *or* $62.5 < Mheight \leq 63.5$ *or* $67.5 < Mheight \leq 68.5$. The vertical bar `|` is the logical "or" operator; type `help("|")` to get the syntax for other logical operators. `Mheight[sel]` selects the elements of `Mheight` with `sel` equal to TRUE[1]. The `xlab` and `ylab` labels are used to label the axes, or else the label for the x-axis would have been, for example, `Mheight[sel]`.

ALR[F1.3] adds several new features to scatterplots. First, two graphs are drawn in one window. This is done using the `par` function, which sets global parameters for graphical windows,

```
> oldpar <- par(mfrow=c(1, 2))
```

meaning that the graphical window will hold an array of graphs with one row and two columns. This choice for the window will be kept until either the graphical window is closed or it is reset using another `par` command.

To draw ALR[F1.3] we need to find the equation of the line to be drawn, and this is done using the `lm` command, the workhorse in R for fitting linear regression models. We will discuss this at much greater length in the next two chapters beginning in Section 2.4, but for now we will simply use the function.

```
> m0 <- lm(Pressure ~ Temp, data=forbes)
> plot(Pressure ~ Temp, data=forbes, xlab="Temperature",
+      ylab="Pressure")
> abline(m0)
> plot(residuals(m0) ~ Temp, forbes, xlab="Temperature",
+      ylab="Residuals")
> abline(a=0, b=0, lty=2)
> par(oldpar)
```

The OLS fit is computed using the `lm` function and assigned the name `m0`. The first `plot` draws the scatterplot of *Pressure* versus *Temp*. We add the regression line with the function `abline`. When the argument to `abline` is the

[1]Although not used in the example, `Mheight[!sel]` would select the elements of `Mheight` with `sel` equal to FALSE.

name of a regression model, the function gets the information needed from the model to draw the OLS line. The second `plot` draws the second graph, this time of the residuals from model `m0` on the vertical axis versus *Temp* on the horizontal axis. A horizontal dashed line is added to this graph by specifying intercept `a` equal to zero and slope `b` equal to zero. The argument `lty=2` specifies a dashed line; `lty=1` would have given a solid line. The last command is unnecessary for drawing the graph, but it returns the value of `mfrow` to its original state.

ALR[F1.5] has two lines added to it, the OLS line and the line joining the mean for each value of *Age*. Here are the commands that draw this graph:

```
> with(wblake, plot(Age, Length))
> abline(lm(Length ~ Age, data=wblake))
> with(wblake, lines(1:8, tapply(Length, Age, mean), lty=2))
```

There are a few new features here. First, in the call to `plot` we specified the x- and y-axes, in that order, rather than using a formula like `Length ~ Age`. The call to `lm` is combined with `abline` into a single command. The `lines` command adds lines to an existing plot, joining the points specified. The points have horizontal coordinates `1:8`, the integers from one to eight for the *Age* groups, and vertical coordinates given by the mean *Length* at each *Age*. The function `tapply` can be read "apply the function `mean` to the variable *Length* separately for each value of *Age*."

ALR[F1.7] adds two new features: a separate plotting character for each of the groups, and a legend or key. Here is how this was drawn in R:

```
> plot(Gain ~ A, data=turkey, pch=S,
+      xlab="Amount(percent of diet)",
+      col=S, ylab="Weight gain, g")
> legend("topleft", legend=c("1", "2", "3"), pch=1:3,
+      cex=.6, inset=0.02)
```

S is a variable in the data frame with the values 1, 2, or 3, so setting `pch=S` sets the plotting character to be a "1," "2" or "3" depending on the value of *S*. Similarly, setting `col=S` will set the color of the plotted points to be different for each of the three groups. The first two arguments to the `legend` set its upper left corner, inset by 2% as set by the `inset` argument. The argument `legend=` is a list of text to show in the legend, and the `pch` argument tells to show the plotting characters. There are many other options; see the help page for `legend` or VR[4.3].

1.2 MEAN FUNCTIONS

1.3 VARIANCE FUNCTIONS

1.4 SUMMARY GRAPH

1.5 TOOLS FOR LOOKING AT SCATTERPLOTS

R ALR[F1.10] adds a smoother to ALR[F1.1]. R has a wide variety of smoothers available, and all can be added to a scatterplot. Here is how to add a *loess* smooth using the `lowess` function.

```
> plot(Dheight ~ Mheight, heights, cex=.1,pch=20)
> abline(lm(Dheight ~ Mheight, heights), lty=1)
> with(heights, lines(lowess(Dheight ~ Mheight, f=6/10,
+                     iter=1), lty=2))
```

The `lowess` specifies the response and predictor in the same way as `lm`. It also requires a smoothing parameter, which we have set to 0.6. The argument `iter` also controls the behavior of the smoother; we prefer to set `iter=1`, not the default value of 3.

1.6 SCATTERPLOT MATRICES

R The R function `pairs` draws scatterplot matrices. To reproduce an approximation to ALR[F1.11], we must first transform the data to get the variables that are to be plotted, and then draw the plot.

```
> fuel2001 <- transform(fuel2001,
+              Dlic=1000 * Drivers/Pop,
+              Fuel=1000 * FuelC/Pop,
+              Income = Income/1000,
+              logMiles = log2(Miles))
> names(fuel2001)

 [1] "Drivers"  "FuelC"    "Income"   "Miles"    "MPC"      "Pop"
 [7] "Tax"      "Dlic"     "Fuel"     "logMiles"

> pairs(~ Tax + Dlic + Income + logMiles + Fuel, data=fuel2001)
```

We used the `transform` to add the transformed variables to the data frame. Variables that reuse existing names, like *Income*, overwrite the existing variable. Variables with new names, like *Dlic*, are new columns added to the right of the data frame. The syntax for the transformations is similar to the language C. The variable $Dlic = 1000 \times Drivers/Pop$ is the fraction of

the population that has a driver's license times 1000. The variable *logMiles* is the base-two logarithm of *Miles*. The `log2` function in R can be used to compute logarithms to the base 2, while `log` and `log10` compute natural logs and base-ten logs, respectively. We could have alternatively computed the transformations one at a time, for example

```
> fuel2001$FuelPerDriver <- fuel2001$FuelC / fuel2001$Drivers
```

Typing `names(f)` gives variable names in the order they appear in the data frame. We specify the variables we want to appear in the plot using a one-sided formula, which consists of a "∼" followed by the variable names separated by + signs. In place of the one-sided formula, you could also use a two-sided formula, putting the response *Fuel* to the left of the ∼, but not on the right. Finally, you can replace the formula by a matrix or data frame, so

```
> pairs(fuel2001[, c(7,9,3,11,10)])
```

would give the scatterplot matrix for columns $7, 9, 3, 11, 10$ of the data frame `f`, resulting in the same plot as before.

The function `scatterplotMatrix` in the `car` package provides a more elaborate version of scatterplot matrices. The help file for the function, or COMPANION[3.3.2], can provide information.

Problems

1.1. Boxplots would be useful in a problem like this because they display level (median) and variability (distance between the quartiles) simultaneously. These can be drawn in R with the `boxplot` command, COMPANION[3.1.4].

R The `tapply` can be used to get the standard deviations for each value of *Age*. In R, the function `sd` computes the standard deviation.

In R, the command `plot(Length ~ as.factor(Age), data=wblake)` will produce the boxplot, while `plot(Length ~ Age, data=wblake)` produces a scatterplot. The `as.factor` command converts a numeric variable into a categorical variable.

1.2.

R You can resize a graph without redrawing it by using the mouse to move the lower right corner on the graph.

1.3.

R Remember that either `logb(UN1$Fertility,2)` or `log2(UN1$Fertility)` gives the base-two logarithm, COMPANION[3.4.1].

2
Simple Linear Regression

2.1 ORDINARY LEAST SQUARES ESTIMATION

All the computations for simple regression depend on only a few summary statistics; the formulas are given in the text, and in this section we show how to do the computations step–by-step. All computer packages will do these computations automatically, as we show in Section 2.6.

2.2 LEAST SQUARES CRITERION

R All the sample computations shown in ALR are easily reproduced in R. First, get the right variables from the data frame, and compute means. The computation is simpler in R:

```
> forbes1 <- forbes[, c(1, 3)]
> print(fmeans <- colMeans(forbes1))
```

```
 Temp Lpres
203.0 139.6
```

The `colMeans` function computes the mean of each column of a matrix or data frame; the `rowMeans` function does the same for row means.

Next, we need the sums of squares and cross-products. Since the sample covariance matrix is just $(n-1)$ times the matrix of sums of squares and cross-products, we can use the function `cov`:

```
> fcov <- (17 - 1) * cov(forbes1)
```

All the regression summaries depend only on the sample means and this matrix. Assign names to the components, and do the computations in the book:

```
> xbar <- fmeans[1]
> ybar <- fmeans[2]
> SXX  <- fcov[1,1]
> SXY  <- fcov[1,2]
> SYY  <- fcov[2,2]
> betahat1 <- SXY/SXX
> print(round(betahat1, 3))

[1] 0.895

> betahat0 <- ybar - betahat1 * xbar
> print(round(betahat0,3))

 Lpres
-42.14
```

2.3 ESTIMATING σ^2

R We can use the summary statistics computed previously to get the *RSS* and the estimate of σ^2:

```
> print(RSS <- SYY - SXY^2/SXX)

[1] 2.155

> print(sigmahat2 <- RSS/15)

[1] 0.1437

> print(sigmahat <- sqrt(sigmahat2))

[1] 0.379
```

2.4 PROPERTIES OF LEAST SQUARES ESTIMATES

2.5 ESTIMATED VARIANCES

The estimated variances of coefficient estimates are computed using the summary statistics we have already obtained. These will also be computed automatically linear regression fitting methods, as shown in the next section.

2.6 COMPARING MODELS: THE ANALYSIS OF VARIANCE

Computing the analysis of variance and F test by hand requires only the value of RSS and of $SSreg = SYY - RSS$. We can then follow the outline given in ALR[2.6].

R We show how the function `lm` can be used for getting all the summaries in a simple regression model, and then the analysis of variance and other summaries. `lm` is also described in COMPANION[3].

The basic form of the `lm` function is

```
> lm(formula, data)
```

The argument `formula` describes the mean function to be used in fitting, and is the only required argument. The `data` argument gives the name of the data frame that contains the variables in the formula, and may be ignored if the data used in the formula is explicitly available when `lm` is called. Other possible arguments will be described as we need them.

The formula is no more than a simplified version of the mean function. For the Forbes example, the mean function is $\mathrm{E}(Lpres|Temp) = \beta_0 + \beta_1 Temp$, and the corresponding formula is

```
> Lpres ~ Temp
```

The left-hand side of the formula is the response variable. The "`=`" sign in the mean function is replaced by "`~`". In this and other formulas used with `lm`, the *terms* in the mean function are specified, but the *parameters* are omitted. Also, the intercept is included without being specified. You could put the intercept in explicitly by typing

```
> Lpres ~ 1 + Temp
```

Although not relevant for Forbes data, you could fit regression through the origin by explicitly removing the term for the intercept,

```
> Lpres ~ Temp - 1
```

You can also replace the variables in the mean function by transformations of them. For example,

```
> log10(Pressure) ~ Temp
```

is exactly the same mean function because *Lpres* is the base-ten logarithm of *Pressure*. See COMPANION[4.2.1] for a discussion of formulas.

The usual procedure for fitting a simple regression via OLS is to fit using `lm`, and assign the result a name:

```
> m1 <- lm(Lpres ~ Temp, data=forbes)
```

The object `m1` contains all the information about the regression that was just fit. There are a number of helper functions that let you extract the information, either for printing, plotting, or other computations. The `print` function displays a brief summary of the fitted model. The `summary` function displays a more complete summary:

```
> summary(m1)

Call:
lm(formula = Lpres ~ Temp, data = forbes)

Residuals:
    Min      1Q  Median      3Q     Max
-0.3222 -0.1447 -0.0666  0.0218  1.3598

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -42.1378     3.3402   -12.6  2.2e-09
Temp          0.8955     0.0165    54.4  < 2e-16

Residual standard error: 0.379 on 15 degrees of freedom
Multiple R-squared: 0.995,        Adjusted R-squared: 0.995
F-statistic: 2.96e+03 on 1 and 15 DF,  p-value: <2e-16
```

You can verify that the values shown here are the same as those obtained by "hand" previously. The output contains a few items never discussed is in ALR, and these are edited out of the output shown in ALR. These include the quantiles of the residuals, and the "Adjusted R-Squared." Neither of these seem very useful. Be sure that you understand all the remaining numbers in this output.

The helper function `anova` has several uses in R, and the simplest is to get the analysis of variance described in ALR[2.6],

```
> anova(m1)

Analysis of Variance Table

Response: Lpres
          Df Sum Sq Mean Sq F value Pr(>F)
Temp       1    426     426    2963 <2e-16
Residuals 15      2       0
```

We have just scratched the surface of the options for `lm` and for specifying formulas. Stay tuned for more details later in this primer.

2.7 THE COEFFICIENT OF DETERMINATION, R^2

R For simple regression, R^2 can be computed as shown in the book. Alternatively, the correlation can be computed directly using the `cor` command:

```
> SSreg <- SYY - RSS
> print(R2 <- SSreg/SYY)

[1] 0.995

> cor(forbes1)

        Temp  Lpres
Temp  1.0000 0.9975
Lpres 0.9975 1.0000
```

and $0.99747817^2 = 0.9949627$. R^2 is also printed by the summary method shown in the last section.

2.8 CONFIDENCE INTERVALS AND TESTS

Confidence intervals and tests can be computed using the formulas in ALR[2.8], in much the same way as the previous computations were done.

R To use `lm` to compute confidence intervals, we need access to the estimates, their standard errors (for predictions we might also need the covariances between the estimates), and we also need to be able to compute multipliers from the appropriate t distribution.

To get access to the coefficient estimates, use the helper function `coef`,

```
> print(betahat <- coef(m1))

(Intercept)        Temp
   -42.1378      0.8955
```

The command `vcov` can be used to get the matrix of variances and covariances of the estimates:

```
> print(var <- vcov(m1))

            (Intercept)       Temp
(Intercept)    11.15693 -0.0549313
Temp           -0.05493  0.0002707
```

The correlation matrix can be computed with `cov2cor`

```
> cov2cor(var)
```

```
            (Intercept)    Temp
(Intercept)      1.0000 -0.9996
Temp            -0.9996  1.0000
```

The estimated variance of the intercept $\hat{\beta}_0$, the square of the standard error, is 11.1569. The standard errors are the square roots of the diagonal elements of this matrix, given by `sqrt(diag(var))`. The covariance between $\hat{\beta}_0$ and $\hat{\beta}_1$ is -0.0549, and the correlation is -0.9996.

Oddly enough, getting the estimated variance $\hat{\sigma}$ is a little harder:

```
> summary(m1)$sigma

[1] 0.379
```

The function `sigmaHat` in the `car` package, which is automatically loaded whenever you load `alr3`, remembers this for you:

```
> sigmaHat(m1)

[1] 0.379
```

The last item we need to compute confidence intervals is the correct multiplier from the t distribution. For a 95% interval, the multiplier is

```
> tval <- qt(1-.05/2, m1$df)
> tval

[1] 2.131
```

The `qt` command computes quantiles of the t-distribution; similar function are available for the normal (`qnorm`), F (`qF`), and other distributions; see Section 0.5.4. The function `qt` finds the value *tval* so that the area to the left of *tval* is equal to the first argument to the function. The second argument is the degrees of freedom, which can be obtained from `m1` as shown above.

Finally, the confidence intervals for the two estimates are:

```
> data.frame(Est = betahat,
+            lower=betahat - tval * sqrt(diag(var)),
+            upper=betahat + tval * sqrt(diag(var)))

                Est    lower    upper
(Intercept) -42.1378 -49.2572 -35.0183
Temp          0.8955   0.8604   0.9306
```

By creating a data frame, the values get printed in a nice table.

The standard R includes a function called `confint` that computes the confidence intervals for you. This function

```
> confint(m1, level=.95)
```

```
                2.5 %   97.5 %
(Intercept) -49.2572 -35.0183
Temp          0.8604   0.9306
```

gives the same answer. The `confint` function works with many other models in R, and is the preferred way to compute confidence intervals. For other than linear regression, it uses a more sophisticated method for computing confidence intervals.

Prediction and fitted values

R The `predict` command is a very powerful helper function for getting fitted and predictions from a model fit with `lm`, or, indeed, most other models in R. Here are the important arguments in R:

```
> predict(object, newdata, se.fit = FALSE,
+               interval = c("none", "confidence", "prediction"),
+               level = 0.95)
```

The `object` argument is the name of the regression model that has already been computed. In the simplest case, we have

```
> predict(m1)

    1     2     3     4     5     6     7     8     9    10    11
132.0 131.9 135.1 135.5 136.4 136.9 137.8 137.9 138.2 138.1 140.2
   12    13    14    15    16    17
141.1 145.5 144.7 146.5 147.6 147.9
```

returning the fitted values (or predictions) for each observation. If you want predictions for particular values, say *Temp* = 210 and 220, use the command

```
> predict(m1, newdata=data.frame(Temp=c(210,220)),
+          interval="prediction", level=.95)

    fit   lwr   upr
1 145.9 145.0 146.8
2 154.9 153.8 155.9
```

The `newdata` argument is a powerful tool in using the `predict` command, as it allows computing predictions at arbitrary points. The argument must be a data frame, with variables having the same names as the variables used in the mean function. For the Forbes data, the only term beyond the intercept is for *Temp*, and so only values for *Temp* must be provided. The argument `intervals="prediction"` gives prediction intervals at the specified level in addition to the point predictions; other intervals are possible, such as for fitted values; see `help(predict.lm)` for more information.

Additional arguments to `predict` will compute additional quantities. For example, `se.fit` will also return the standard errors of the fitted values (not the standard error of prediction). For example,

```
> predvals <- predict(m1, newdata=data.frame(Temp=c(210, 220)),
+                     se.fit=TRUE)
> predvals

$fit
    1     2
145.9 154.9

$se.fit
     1      2
0.1480 0.2951

$df
[1] 15

$residual.scale
[1] 0.379
```

The result `predvals` is a list. You could get a more compact representation by typing

```
> as.data.frame(predvals)

    fit se.fit df residual.scale
1 145.9 0.1480 15          0.379
2 154.9 0.2951 15          0.379
```

You can do computations with these values. For example,

```
> (150 - predvals$fit)/predvals$se.fit

     1      2
 27.60 -16.50
```

computes the difference between 150 and the fitted values, and then divides each by its standard error of the fitted value. The `predict` helper function does not compute the standard error of prediction, but you can compute it using equation ALR[E2.26],

```
> se.pred <- sqrt(predvals$residual.scale^2 + predvals$se.fit^2)
```

2.9 THE RESIDUALS

R The command `residuals` computes the residuals for a model. A plot of residuals versus fitted values with a horizontal line at the origin is given by

```
> plot(predict(m1), residuals(m1))
> abline(h=0,lty=2)
```

We will have more elaborate uses of `residuals` later in the primer. If you apply the `plot` helper function to the regression model `m1` by typing `plot(m1)`, you will also get the plot of residuals versus fitted values, along with a few other plots. We will not use the plot methods for models in this primer because it often includes plots not discussed in ALR. Finally, the `car` function `residualPlots` can also be used:

```
> residualPlots(m1)
```

We will discuss this last function in later sections of this primer.

Problems

2.2.

R You need to fit the *same* model to both the `forbes` and `hooker` data sets. You can then use the `predict` command to get the predictions for Hooker's data from the fit to Forbes' data, and *viceversa*.
2.7.

R The formula for regression through the origin explicitly removes the intercept, `y ~ x - 1`.
2.10.

3 Multiple Regression

3.1 ADDING A TERM TO A SIMPLE LINEAR REGRESSION MODEL

R The avPlots function in the car package can be used to draw added-variable plots. Introducing this function here is a little awkward because it requires fitting a regression model first, a topic covered later in this chapter. Nevertheless, we consider the data file UN3 that includes the three variables *Purban*, *PPgdp* and *Fertility* discussed in ALR[3.1]. The resulting plots are shown in Figure 3.1.

```
> m1 <- lm(log2(Fertility) ~ log2(PPgdp) + Purban, data=UN3)
> avPlots(m1, id.n=0)
```

The first command sets up the linear regression model using lm, as will be discussed in the remaining sections of this chapter. The formula is a little different because rather than first computing a variable like

```
> UN3$logFertility <- log2(UN3$Fertility)
```

we compute the logarithm on the fly inside the formula. This is a very useful practice in R that is generally not available in other computer programs. The avPlots takes the regression model as its first argument. The default of the function is to display the added-variable plot for each variable on the right side of the formula[1], so we get two plots here. The argument id.n=0 is used to

[1]Although we haven't yet discussed factors, an added-variable plot will also be displayed for each of the contrasts that makes up a factor, and these are probably not of much interest.

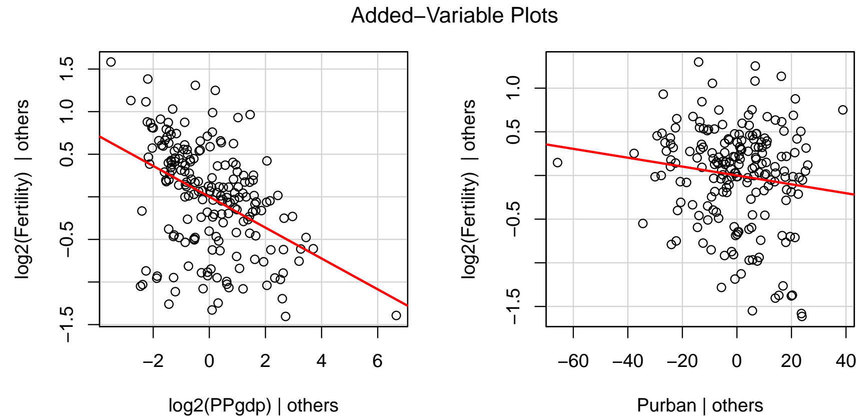


Fig. 3.1 Added-variable plots for each of the two predictors in the regression of log(*Fertility*) on log(*PPgdp*) and *Purban*, using base-two logarithms. The second of these two figures matches ALR[F3.1D].

suppress *point labeling*. By default `avPlots` will label several extreme points in the plot using row labels. In these plots they are rather distracting.

Finally, some of the values of the three variables in the regression model are not observed for some of the UN countries. These values are indicated by the symbol NA in the data file. By default R silently deletes all countries with missing values on at least one of the three variables, and does an analysis only on the fully observed cases.

Added-variable plots are discussed in COMPANION[6.2.3]. Point labeling is discussed in COMPANION[3.5]. Missing data is discussed in COMPANION[2.2.3].

3.2 THE MULTIPLE LINEAR REGRESSION MODEL

3.3 TERMS AND PREDICTORS

R Table 3.1 in ALR[3.3] gives the "usual" summary statistics for each of the interesting terms in a data frame. Oddly enough, the writers of R don't seem to think these are the standard summaries. Here is what you get from R:

```
> fuel2001 <- transform(fuel2001,
+               Dlic=1000 * Drivers/Pop,
```

You can choose the variables you want displayed using the `vars` argument, as discussed on the function's help page.

```
+             Fuel=1000 * FuelC/Pop,
+             Income = Income,
+             logMiles = log2(Miles))
> f <- fuel2001[,c(7, 8, 3, 10, 9)] # new data frame
> summary(f)
```

```
      Tax             Dlic          Income          logMiles
 Min.   : 7.5   Min.   : 700   Min.   :20993   Min.   :10.6
 1st Qu.:18.0   1st Qu.: 864   1st Qu.:25323   1st Qu.:15.2
 Median :20.0   Median : 909   Median :27871   Median :16.3
 Mean   :20.2   Mean   : 904   Mean   :28404   Mean   :15.7
 3rd Qu.:23.2   3rd Qu.: 943   3rd Qu.:31208   3rd Qu.:16.8
 Max.   :29.0   Max.   :1075   Max.   :40640   Max.   :18.2
      Fuel
 Min.   :317
 1st Qu.:575
 Median :626
 Mean   :613
 3rd Qu.:667
 Max.   :843
```

We created a new data frame `f` using only the variables of interest. Rather than giving the standard deviation for each variable, the `summary` command provides first and third quartiles. You can get the standard deviations easily enough, using

```
> apply(f, 2, sd)
```

```
     Tax      Dlic   Income logMiles      Fuel
   4.545   72.858 4451.637    1.487   88.960
```

The argument `2` in `apply` tells the program to apply the third argument, the `sd` function, to the second dimension, or columns, of the matrix or data frame given by the first argument. Sample correlations, ALR[T3.2], are computed using the `cor` command,

```
> round(cor(f),4)
```

```
             Tax    Dlic  Income logMiles    Fuel
Tax       1.0000 -0.0858 -0.0107  -0.0437 -0.2594
Dlic     -0.0858  1.0000 -0.1760   0.0306  0.4685
Income   -0.0107 -0.1760  1.0000  -0.2959 -0.4644
logMiles -0.0437  0.0306 -0.2959   1.0000  0.4220
Fuel     -0.2594  0.4685 -0.4644   0.4220  1.0000
```

3.4 ORDINARY LEAST SQUARES

R The sample covariance matrix is computed using either `var` or `cov`, so

```
> cov(f)
```

```
              Tax       Dlic     Income  logMiles       Fuel
Tax       20.6546    -28.425     -216.2   -0.2955    -104.89
Dlic     -28.4247   5308.259   -57070.5    3.3135    3036.59
Income  -216.1725 -57070.454 19817074.0 -1958.0367 -183912.57
logMiles  -0.2955      3.314    -1958.0    2.2103      55.82
Fuel    -104.8944   3036.591  -183912.6   55.8172    7913.88
```

We will compute the matrix $(\mathbf{X}'\mathbf{X})^{-1}$. To start we need the matrix $\mathbf{X}$, which has 51 rows, one column for each predictor, and one column for the intercept. Here are the computations:

```
> f$Intercept <- rep(1, 51)  # a column of ones added to f
> X <- as.matrix(f[, c(6, 1, 2, 3, 4)]) # reorder and drop fuel
> xtx <- t(X) %*% X
> xtxinv <- solve(xtx)
> xty <- t(X) %*% f$Fuel
> print(xtxinv, digits=4)
```

```
           Intercept        Tax       Dlic     Income   logMiles
Intercept  9.022e+00 -2.852e-02 -4.080e-03 -5.981e-05 -1.932e-01
Tax       -2.852e-02  9.788e-04  5.599e-06  4.263e-08  1.602e-04
Dlic      -4.080e-03  5.599e-06  3.922e-06  1.189e-08  5.402e-06
Income    -5.981e-05  4.263e-08  1.189e-08  1.143e-09  1.000e-06
logMiles  -1.932e-01  1.602e-04  5.402e-06  1.000e-06  9.948e-03
```

The first line added a column to the `f` data frame that consists of 51 copies of the number 1. The function `as.matrix` converted the reordered data frame into a matrix $\mathbf{X}$. The next line computed $\mathbf{X}'\mathbf{X}$, using the function `t` to get the transpose of a matrix, and `%*%` for matrix multiply. The `solve` function returns the inverse of is argument; it is also used to solve linear equations if the function has two arguments.

The estimates and other regression summaries can be be computed, based on these sufficient statistics:

```
> xty <- t(X) %*% f$Fuel
> betahat <- xtxinv %*% xty
> betahat
```

```
                [,1]
Intercept 154.192845
Tax        -4.227983
Dlic        0.471871
Income     -0.006135
logMiles   18.545275
```

As with simple regression the function `lm` is used to automate the fitting of a multiple linear regression mean function. The only difference between the simple and multiple regression is the formula:

```
> m1 <- lm(formula = Fuel ~ Tax + Dlic + Income + logMiles,
+          data = f)
> summary(m1)

Call:
lm(formula = Fuel ~ Tax + Dlic + Income + logMiles, data = f)

Residuals:
   Min     1Q Median     3Q    Max
-163.1  -33.0    5.9   32.0  183.5

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 154.19284  194.90616    0.79  0.43294
Tax          -4.22798    2.03012   -2.08  0.04287
Dlic          0.47187    0.12851    3.67  0.00063
Income       -0.00614    0.00219   -2.80  0.00751
logMiles     18.54527    6.47217    2.87  0.00626

Residual standard error: 64.9 on 46 degrees of freedom
Multiple R-squared: 0.51,       Adjusted R-squared: 0.468
F-statistic:   12 on 4 and 46 DF,  p-value: 9.33e-07
```

3.5 THE ANALYSIS OF VARIANCE

R The F-statistic printed at the bottom of the output for `summary(m1)` shown above corresponds to ALR[T3.4], testing all coefficients except the intercept equal to zero versus the alternative that they are not all zero.

Following the logic in ALR[3.5.2], we can get an F-test for the hypothesis that $\beta_1 = 0$ versus a general alternative by fitting two models, the larger one we have already fit for the mean function under the alternative hypothesis, and a smaller mean function under the null hypothesis. We can fit this second mean function using the `update` command:

```
> m2 <- update(m1, ~.-Tax)
> anova(m2,m1)

Analysis of Variance Table

Model 1: Fuel ~ Dlic + Income + logMiles
Model 2: Fuel ~ Tax + Dlic + Income + logMiles
```

```
  Res.Df    RSS Df Sum of Sq    F Pr(>F)
1     47 211964
2     46 193700  1     18264 4.34  0.043
```

The `update` function takes an existing object and updates it in some way, usually by changing the mean function or by changing the data. In this instance, we have updated the mean function on the right-hand side by taking the existing terms, indicated by the "." and then *removing Tax* by putting it in the mean function with a *negative* sign. We can then use the `anova` helper command to compare the two mean functions. The information is equivalent to what is shown in ALR[3.5.2] after equation ALR[E3.22], but in a somewhat different layout; I find the output from `anova` to be confusing.

If you use the `anova` helper command with just one argument, you will get a sequential Anova table, like ALR[T3.5]. The terms are fit in the order you specify them in the mean function, so `anova` applied to the following mean functions

```
> m1 <- lm(Fuel ~ Tax + Dlic + Income + logMiles, data = f)
> m2 <- update(m1, ~ Dlic + Tax + Income + logMiles)
> m3 <- update(m1, ~ Tax + Dlic + logMiles + Income)
```

all give different anova tables.

3.6 PREDICTIONS AND FITTED VALUES

R Predictions and fitted values for multiple regression use the `predict` command, just as for simple linear regression. If you want predictions for new data, you must specify values for *all* the terms in the mean function, apart from the intercept, so for example,

```
> predict(m1, newdata=data.frame(
+         Tax=c(20, 35), Dlic=c(909, 943), Income=c(16.3, 16.8),
+         logMiles=c(15, 17)))

    1     2
776.6 766.4
```

will produce two predictions of future values for the values of the predictors indicated.

Problems

R Several of these problems concern added-variable plots. These are easy enough to compute using the following method for the added-variable plot for a particular term X_2 in a mean function given in `m0` that you have already fit.

1. Fit the model `m1` that is the regression of the response on all terms *except for* X_2.
2. Fit the model `m2` that is the regression of X_2 on all the other predictors.
3. Draw the added-variable plot, `plot(residuals(m2), residuals(m1))`.

You can actually combine all these into a single very complex, statement:

```
> plot(residuals(update(m0,~.-X2)), residuals(update(m0, X2 ~ . -X2))
```

The plot above won't have labels or other identifying information. To automate printing added variable plots you can write your own function that will do the regressions and draw the plot, or you can use the function `avPlots`. For the fuel data,

```
> avPlots(m0, id.n=0)
```

will produce all added-variable plots in one window.

4 Drawing Conclusions

The first three sections of this chapter do not introduce any new computational methods; everything you need is based on what has been covered in previous chapters. The last two sections, on missing data and on computationally intensive methods introduce new computing issues.

4.1 UNDERSTANDING PARAMETER ESTIMATES

R In ALR[T4.1], Model 3 is overparameterized. R will print the missing value symbol `NA` for terms that are linear combinations of the terms already fit in the mean function, so they are easy to identify from the printed output.

If you are using the output of a regression as the input to some other computation, you may want to check to see if the model was overparameterized or not. If you have fit a model called, for example, `m2`, then the value of `m2$rank` will be the number of terms actually fit in the mean function, *including the intercept, if any*. It is also possible to determine which of the terms specified in the mean function were actually fit, but the command for this is obscure:

```
> m2$qr$pivot[1:m2$qr$rank]
```

will return the indices of the terms, starting with the intercept, that were estimated in fitting the model.

4.1.1 Rate of change

4.1.2 Sign of estimates

4.1.3 Interpretation depends on other terms in the mean function

4.1.4 Rank deficient and over-parameterized models

4.2 EXPERIMENTATION VERSUS OBSERVATION

4.3 SAMPLING FROM A NORMAL POPULATION

4.4 MORE ON R^2

4.5 MISSING DATA

The data files that are included with ALR use `NA` as a place holder for missing values. Some packages may not recognize this as a missing value indicator, and so you may need to change this character using an editor to the appropriate character for your program.

R R uses the symbol `NA` to incdicate missing values by default but you can you other missing-value indicators when you read data from a file. Suppose, for example, you have a text file `mydata.txt` that uses a period "." as the missing value indicator, with variable names in the first row. You could read this file as follows

```
> data <- read.table("mydata.txt", header=TRUE, na.strings=".")
```

This will read the file, and covert the "." to the missing value indicator.

There are several functions for working with missing value indicators. `is.na` serves two purposes, to set elements of a vector or array to missing, and to test each element of the vector or array to be missing, and return either TRUE or FALSE.

```
> a <- 1:5              # set a to be the vector (1,2,3,4,5)
> is.na(a) <- c(1,5)   # set elements 1 and 5 of a to NA
> a                     # print a

[1] NA  2  3  4 NA

> is.na(a)              # for each element of a is a[j] = NA?

[1]  TRUE FALSE FALSE FALSE  TRUE
```

R has a function called `complete.cases` that returns a vector with the value TRUE for all cases in a data frame with no missing values, and FALSE otherwise. For example,

```
> complete.cases(sleep1)
 [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[12]  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
[23]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
[34]  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
[45]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
[56] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

Many commands in R have an argument `na.action` that tells the command what to do if missing data indicators are present. The usual default is `na.action=na.omit`, which means *delete all cases with missing data.* Another option that is always available is `na.action=na.fail`, which would prevent the command from executing at all with missing data. In principle there could be other actions, but none are commonly used. Other methods will have an argument `na.omit` with default value FALSE. For example, the function `mean(X)` will return `NA` if any of the elements of X are missing, while `mean(X, na.omit=TRUE)` will return the mean of the values actually observed. As a warning, R functions are not always consistent on their use of arguments for missing data, or on the defaults for those arguments. You can always consult the help pages for the commands of interest to get the arguments right.

The `na.action` argument is available for `lm`. *The default is* `na.action=no.omit`. Explicitly including the `na.omit` argument makes clear what you are doing:

```
> m1 <- lm(log(SWS) ~ log(BodyWt) + log(Life) + log(GP),
+          data=sleep1, na.action=na.omit)
```

which will delete 20 cases with missing values. If you then fit

```
> m2 <- lm(log(SWS) ~ log(BodyWt) + log(GP),
+          data=sleep1, na.action=na.omit)
```

omitting log(*Life*), you can't actually compare the two fits via an analysis of variance because only 18 cases are missing the remaining variables.

You can guarantee that all fits are based on the same cases using the `subset` argument to `lm`.

```
> m3 <- lm(log(SWS) ~ log(BodyWt) + log(Life) + log(GP), data=sleep1,
+          subset = complete.cases(sleep1))
```

will fit this mean function, and any other, to the 42 fully observed cases in the data frame.

4.6 COMPUTATIONALLY INTENSIVE METHODS

R R is well suited for the bootstrap and other computationally intensive statistics. The books by Davison and Hinkley (1997) and by Efron and Tibshirani (1993) both provide comprehensive introductions to the bootstrap, and both have packages for R for general bootstrap calculations.

The case-resampling bootstrap is very easy. Using the transactions data discussed in ALR[4.6.1], here is the approach using R and the `bootCase` function in the `car` package:

```
> m1 <- lm(Time~ T1 + T2, data=transact)
> betahat.boot <- bootCase(m1,B=999)
```

The call to `bootCase` has three arguments. In the example above, we have used two of them, the name of the regression model, and the number of bootstraps, `B=999`. The third argument is called `f`, the name of a function to be evaluated on each bootstrap. The default is the generic function `coef` that will return the coefficient estimates, and since we did not specify a value for `f`, the default is used. On each bootstrap the coefficient estimates are saved.

At this point, `betahat.boot` is a matrix with $B = 999$ rows, one for each bootstrap replication. The number of columns depends on the argument `f` and for the default of the coefficient vector it has as many columns of regression coefficients in the model. Usual R tools can be used to summarize these bootstrap estimates:

```
> # bootstrap standard errors
> apply(betahat.boot,2,sd)

(Intercept)          T1          T2
  195.37481     0.68461     0.15242

> # bootstrap 95% confidence intervals
> cl <- function(x) quantile(x,c(.025,.975))
> apply(betahat.boot,2,cl)

      (Intercept)     T1     T2
2.5%      -238.99 4.1314 1.7494
97.5%      532.43 6.7594 2.3321

> coef(m1)

(Intercept)          T1          T2
   144.3694      5.4621      2.0345
```

We applied the `sd` command to each column to give the bootstrap standard errors. Given next is a short function we called `cl` that computes the 2.5- and 97.5-percentiles of a sample. We apply this to the columns of `betahat.boot` to get percentile based confidence intervals for each of the coefficients. Finally, we printed the coefficient estimates from the original data. We could have looked at histograms, scatterplots or other summaries.

The simulation outlined in ALR[4.6.3] is not based on resampling the observed data, but rather on modifying the observed data by adding normal random numbers with given variances. Once again, you need to write a function to do this simulation. Here is one approach:

```
> catchSim <- function(B=999){
+   ans <- NULL
+   for (i in 1:B) {
+     X <- npdata$Density + npdata$SEdens * rnorm(16)
+     Y <- npdata$CPUE + npdata$SECPUE * rnorm(16)
+     m0 <- lm(Y ~ X - 1)
+     ans <- c(ans, coef(m0))}
+   ans}
```

We then run this function:

```
> b0 <- catchSim(B=999)
> c(mean(b0),cl(b0))
```

```
           2.5%   97.5%
0.30704 0.22206 0.39437
```

For this simulation, we have written a one-off function for this particular problem. It has one argument, the number of simulations. In the `for` loop, we have explicitly referred to the data frame, even though we had previously attached the data. Variable names work a little differently inside a function, and so without the reference to the data frame, this function would not work. The command `rnorm` computes vectors of 16 standard normal random numbers; these are multiplied by the vectors of standard errors to get random numbers with the right standard deviations. The model `m0` is for regression through the origin, and only the estimated slope is kept on each replication. We have shown only a numeric summary of the mean and 95% confidence interval reusing the `cl` function we wrote previously, but graphical summaries could be used as well, probably using a histogram.

Problems

4.10. The datafile `longley` is not included in the `alr3` package, but it is included in the base `datasets` package in R and can be used as any other data set. The variable names are a little different, however:

GNP.deflator	=	GNP price deflator, in percent
GNP	=	GNP, in millions of dollars
Unemployed	=	Unemployment, in thousands of persons
Armed.Forces	=	Size of armed forces, in thousands
Population	=	Population 14 years of age and over, in thousands
Employed	=	Total derived employment in thousands the response
Year	=	Year

5 Weights, Lack of Fit, and More

5.1 WEIGHTED LEAST SQUARES

R Weighted least squares estimates are most easily obtained using the `weights` argument for the `lm` command. In the physics data in ALR[5.1], the weights are the inverse squares of the variable *SD* in the data frame. WLS is computed by

```
> m1 <- lm(y ~ x, data=physics, weights=1/SD^2)
> summary(m1)

Call:
lm(formula = y ~ x, data = physics, weights = 1/SD^2)

Residuals:
      Min        1Q    Median        3Q       Max
-2.32e+00 -8.84e-01  1.27e-06  1.39e+00  2.34e+00

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   148.47       8.08    18.4  7.9e-08
x             530.84      47.55    11.2  3.7e-06

Residual standard error: 1.66 on 8 degrees of freedom
Multiple R-squared: 0.94,       Adjusted R-squared: 0.932
F-statistic:  125 on 1 and 8 DF,  p-value: 3.71e-06
```

```
> anova(m1)

Analysis of Variance Table

Response: y
          Df Sum Sq Mean Sq F value  Pr(>F)
x          1    342     342     125 3.7e-06
Residuals  8     22       3
```

You can nearly "set and forget" weights and use `lm` for WLS just as you would for OLS. There are, however, a few exceptions:

1. The `residuals` helper function returns a vector of $y - \hat{y}$, as with OLS. As we will see later, with WLS a more reasonable set of residuals is given by $\sqrt{w}(y - \hat{y})$. R will return these correct residuals if you specify `residuals(m1, type="pearson")`. You can also used the function `weighted.residuals` that does the same thing. For OLS all the weights equal one, so the Pearson and ordinary residuals are identical. Consequently, if you always use `type="pearson"`, you will always be using the right residuals.

2. The `predict` helper function also works correctly for getting predictions and standard errors of fitted values, but it apparently does not always give the right answer for prediction intervals. The standard error of prediction is $(\hat{\sigma}^2 + \text{sefit}(y|X = \mathbf{x}_*)^2)^{1/2}$, rather than $(\hat{\sigma}^2/w_* + \text{sefit}(y|X = \mathbf{x}_*)^2)^{1/2}$. The R formula assumes that the variance of the future observation is σ^2 rather than σ^2/w_*, where w_* is the weight for the future value.

5.1.1 Applications of weighted least squares

R This subsection has our first application of polynomial regression. R attaches special meaning to the symbol `^` in formulas, so the formula `y ~ x + x^2` *won't work the way you expect it to work.* Instead, you need to use `y ~ x + I(x^2)`. The function `I` *inhibits* its argument, so the special meaning of `^` in formulas is not used, but the more general use as the exponentiation indicator is used.

R has alternative ways of specifying polynomial models, particularly using *orthogonal polyonmials.* The formula `y ~ poly(x,2)` also fits a quadratic polynomial, but rather than using x and x^2 as predictors, it uses x and the residuals from the regression of x^2 on x as predictors. This is numerically more stable, and highly recommended in large problems. Interpretation of coefficients is harder with orthogonal polynomials. The overall F test will be the same with either parameterization, and the t test for the quadratic term will be the same as well; the t-test for x will be different.

If you use `poly` with the argument `raw=TRUE`, then the orthogonal polynomials will be replaced by the raw polynomial terms. This alternative way of

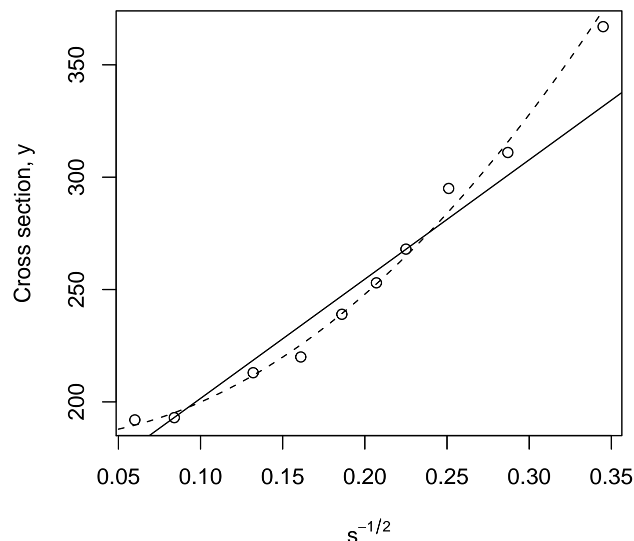
specifying a polynomial will produce the same coefficients as would be obtained entering each power-term in the formula, but the treatment of the terms in some help functions like `anova` is different. Try it and see!

5.1.2 Additional comments

5.2 TESTING FOR LACK OF FIT, VARIANCE KNOWN

R ALR[F5.1] contains data and two fitted curves, from, the linear and quadratic fits to the physics data. Here are the commands that generate this graph:

```
> m1 <- lm(y ~ x, data=physics, weights=1/SD^2)
> m2 <- update(m1, ~ . + I(x^2))
> plot(y ~ x, data=physics, xlab=expression(x=s^{-1/2}),
+           ylab="Cross section, y")
> abline(m1)
> a <- seq(.05,.35,length=50)
> lines(a, predict(m2, newdata=data.frame(x=a)), lty=2)
```



The models `m1` and `m2` are, respectively, the simple linear and quadratic WLS fits. The advantage to using `update` to define `m2` is that we do not need to specify the weights again. The `abline` command draws the fitted line for the simple linear fit. To get the fit for the quadratic is a little harder. The vector `a` includes 50 equally spaced values between .05 and .35, more or less the

minimum and maximum value of x. We then use the `lines` function to plot the predicted values at a versus a. In the `predict` helper function, we only had to specify values for x in the `newdata` argument. The program will use the value of x in both the linear and quadratic terms. Setting `lty=2` gives dashed lines. The use of the `expression` function allows typesetting math, in this case an exponent.

Getting the lack of fit test for known variance requires getting significance levels from the Chi-squared distribution. You can use the `pchisq` command for this purpose; see Section 0.5.4.

5.3 TESTING FOR LACK OF FIT, VARIANCE UNKNOWN

R The test of lack of fit based on replication is not a standard part of R. For problems with just one term x in the mean function beyond the intercept, it is easy to get the lack of fit test by adding `factor(x)` to the mean function:

```
> x <- c(1, 1, 1, 2, 3, 3, 4, 4, 4, 4)
> y <- c(2.55, 2.75, 2.57, 2.40, 4.19, 4.70, 3.81, 4.87,
+        2.93, 4.52)
> m1 <- lm(y ~ x)
> anova(lm(y ~ x + as.factor(x)))

Analysis of Variance Table

Response: y
             Df Sum Sq Mean Sq F value Pr(>F)
x             1   4.57    4.57   11.62  0.014
as.factor(x)  2   1.86    0.93    2.36  0.175
Residuals     6   2.36    0.39
```

The F-test on the line `as.factor(x)` is the F for lack-of-fit, and the line marked `Residuals` is the pure error.

With more than one term beyond the intercept finding the groups of repeated values to give pure error is harder. The `alr3` library includes a function `pureErrorAnova` that will do the trick for any number of terms:

```
> pureErrorAnova(m1)

Analysis of Variance Table

Response: y
             Df Sum Sq Mean Sq F value Pr(>F)
x             1   4.57    4.57   11.62  0.014
Residuals     8   4.22    0.53
 Lack of fit  2   1.86    0.93    2.36  0.175
 Pure Error   6   2.36    0.39
```

Except for the correct labeling of the lack-of-fit line, this gives the same answer as before.

5.4 GENERAL F TESTING

R The `anova` helper function is designed to compute the general F-test described in ALR[5.4]. For example, consider the fuel consumption data, and fit a sequence of models:

```
> fuel2001$Dlic <- 1000 * fuel2001$Drivers / fuel2001$Pop
> fuel2001$Fuel <- 1000*  fuel2001$FuelC / fuel2001$Pop
> fuel2001$Income <- fuel2001$Income / 1000
> fuel2001$logMiles <- log2(fuel2001$Miles)
> m1 <- lm(Fuel ~ Dlic + Income + logMiles + Tax, data=fuel2001)
> m2 <- update(m1, ~ . - Dlic - Income)
> anova(m2, m1)

Analysis of Variance Table

Model 1: Fuel ~ logMiles + Tax
Model 2: Fuel ~ Dlic + Income + logMiles + Tax
  Res.Df    RSS Df Sum of Sq    F  Pr(>F)
1     48 302192
2     46 193700  2    108492 12.9 3.6e-05
```

The `anova` helper lists the models to be compared from smallest model, the null hypothesis, to the largest model, the alternative hypothesis. The output gives the RSS and df under both hypotheses in the columns `Res.df` and `RSS`. The columns `Df` and `Sum of Sq` are the quantities for the numerator of ALR[E5.16]. The F and is p-value are then provided.

You can use `anova` for a longer sequence of nested models as well.

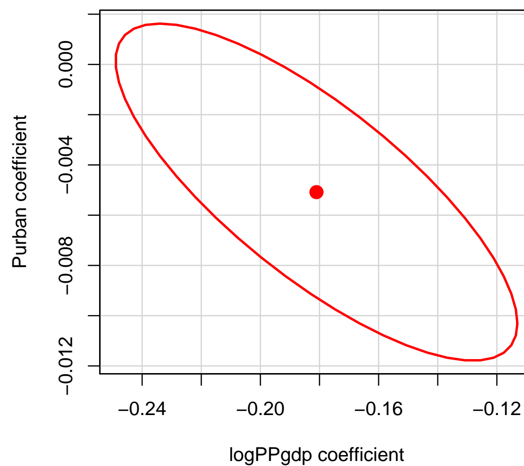
```
> m3 <- update(m2, ~ . - Tax)
> anova(m3, m2, m1)

Analysis of Variance Table

Model 1: Fuel ~ logMiles
Model 2: Fuel ~ logMiles + Tax
Model 3: Fuel ~ Dlic + Income + logMiles + Tax
  Res.Df    RSS Df Sum of Sq     F  Pr(>F)
1     49 325216
2     48 302192  1     23024  5.47   0.024
3     46 193700  2    108492 12.88 3.6e-05
```

The F on line number 2 is for the hypothesis `m3` versus `m2`, which may not be a hypothesis of interest. The F on line 3 tests `m2` versus `m3`, giving the same answer as before.

5.5 JOINT CONFIDENCE REGIONS

R ALR[F5.3] is one of the few figures in ALR that was not drawn in R. It is possible to draw this figure using the `confidenceEllipse` helper function in the `car` library.

```
> m1 <- lm(logFertility ~ logPPgdp + Purban, data = UN2)
> confidenceEllipse(m1, scheffe=TRUE)
```



Problems

5.3.

The bootstrap used in this problem is different from the bootstrap discussed in ALR[4.6] because rather than resampling *cases* we are resampling *residuals*. Here is the general outline of the method:

1. Fit the model of interest to the data. In this case, the model is just the simple linear regression of the response y on the predictor x. Compute the test statistic of interest, given by ALR[E5.23]. Save the fitted values $\hat{y}$ and the residuals $\hat{e}$.

2. A bootstrap sample will consist of the original x and a new y^*, where $y^* = \hat{y} + e^*$. The ith element of e^* is obtained by sampling from $\hat{e}$ with replacement. You can compute e^* by

```
> estar <- ehat[sample(1:n, replace=TRUE)]
```

3. Given the bootstrap data (x, y^*), compute and save ALR[E5.23].

4. Repeat steps 2 and 3 B times, and summarize results.

6
Polynomials and Factors

6.1 POLYNOMIAL REGRESSION

R ALR[F6.2] is a plot of the design points in the cakes data, with the values slightly jittered. R include a command `jitter` to make this easy. Here are the commands that will draw this figure:

```
> with(cakes, plot(jitter(X1), jitter(X2)))
```

`jitter` has arguments to control the magnitude of the jittering; type `help(jitter)` for information.

Polynomial models generally require creating many terms that are functions of a few base predictors. One way to do this is discussed in Section 5.1.1. For example, to fit the cubic polynomial with response y and predictor x, the formula `y ~ 1 + x + I(x^2) + I(x^3)` is appropriate.

An alternative procedure and generally preferred procedure is to use the `poly` to generate the polynomial terms in one or more variables. For example, suppose we have $n = 5$ cases, and $x = (1, 2, 3, 4, 5)'$.

```
> x <- 1:5
> (xmat <- cbind(x^0, x, x^2, x^3))

       x
[1,] 1 1  1   1
[2,] 1 2  4   8
[3,] 1 3  9  27
[4,] 1 4 16  64
[5,] 1 5 25 125
```

```
> as.data.frame(poly(x, 3, raw=TRUE))

  1  2   3
1 1  1   1
2 2  4   8
3 3  9  27
4 4 16  64
5 5 25 125
```

Following ALR, we would use the second to fourth columns of `xmat` for the linear, quadratic, and cubic terms in x. Adding the argument `raw=TRUE` created the same three predictors.

The `poly` command can also be used to get orthogonal polynomials, which are equivalent the ordinary polynomials but the variables created are orthogonal to each other.

```
> as.data.frame(poly(x, 3))

        1       2          3
1 -0.6325  0.5345 -3.162e-01
2 -0.3162 -0.2673  6.325e-01
3  0.0000 -0.5345 -4.096e-16
4  0.3162 -0.2673 -6.325e-01
5  0.6325  0.5345  3.162e-01
```

These are the same variables you would be obtained by applying the QR factorization of `xmat`,

```
> qr.Q(qr(xmat))

        [,1]       [,2]    [,3]       [,4]
[1,] -0.4472 -6.325e-01  0.5345 -3.162e-01
[2,] -0.4472 -3.162e-01 -0.2673  6.325e-01
[3,] -0.4472  1.110e-16 -0.5345  4.025e-15
[4,] -0.4472  3.162e-01 -0.2673 -6.325e-01
[5,] -0.4472  6.325e-01  0.5345  3.162e-01
```

These vectors are uncorrelated, making regression calculations potentially more accurate, but make interpretation of coefficients more difficult. See the help page for `poly`.

6.1.1 Polynomials with several predictors

R (See also COMPANION[SEC. 4.6.3]) With two predictors, and generalizing to more than two predictors, the full second-order mean function ALR[E6.4] can be fit to the cakes data by

```
> m1 <- lm(Y ~ X1 + X2 + I(X1^2) + I(X2^2) + X1:X2, data=cakes)
```

The new feature here is the interaction term `X1:X2`, which is obtained by multiplying `X1` by `X2` elementwise. Other programs often write an interaction as `X1*X2`, but in R, `X1*X2` = `X1+X2+X1:X2`.

As an alternative, you could pre-compute the polynomial terms and even the interactions and add them to the data frame:

```
> cakes$X1sq <- cakes$X1^2
> cakes$X2sq <- cakes$X2^2
> cakes$X1X2 <- cakes$X1 * cakes$X2
> m2 <- lm(Y ~ X1 + X2 + X1sq + X2sq + X1X2, data=cakes)
```

The models `m1` and `m2` are identical, but `m1` will be a little easier to use if you want to get predictions since for `m1` you need only supply new values for `X1` and `X2`, while for the latter you need to supply all five variables.

The third alternative is to use the command `poly` that fits polynomials in several variables.

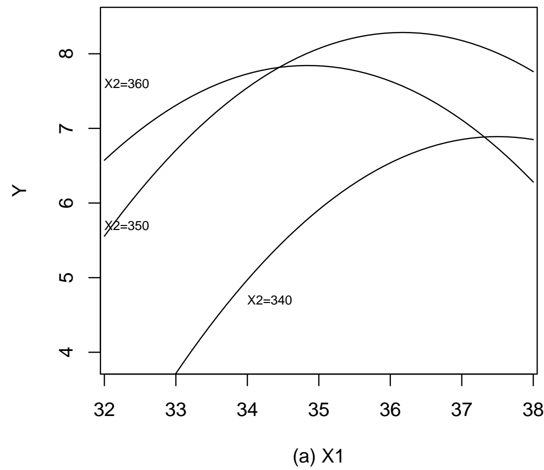
```
> m3 <- lm(Y ~ poly(X1, X2, degree=2, raw=TRUE), data=cakes)
```

which also fits the full second-order model. All of these approaches have advantages and disadvantages. For example, you can't use `update` to change the formula in `m3`, and using `predict` with `m2` is not very straightforward.

The plots shown in ALR[F6.3] are tedious to draw in R[1]. Here are the commands that will draw ALR[F6.3A]:

```
> with(cakes, plot(X1, Y, type="n", xlab="(a) X1"))
> X1new <- seq(32, 38, len=50)
> lines(X1new, predict(m1,
+      newdata=data.frame(X1=X1new, X2=rep(340, 50))))
> lines(X1new, predict(m1,
+      newdata=data.frame(X1=X1new, X2=rep(350, 50))))
> lines(X1new, predict(m1,
+      newdata=data.frame(X1=X1new, X2=rep(360, 50))))
> text(34, 4.7, "X2=340", adj=0, cex=0.7)
> text(32, 5.7, "X2=350" ,adj=0, cex=0.7)
> text(32, 7.6, "X2=360", adj=0, cex=0.7)
```

[1]As with many things in R, there are many ways to do this plot, and I would not be surprised to learn that there is a much easier way.



The code for ALR[F6.3B] is similar. The `par` command sets global graphical parameters, in this case resetting the graphics window to have one row of plots and two columns of plots. The first `plot` sets the values plotted on each axis and the labels for the axes. The argument `type="n"` means *draw the plot but not the points.* The other options are `type="p"` to draw points, the default; `type="l"` to draw lines, and `type="b"` for both points and lines. To the plot we then add three lines of fitted values versus X_1 for X_2 fixed at either 340, 350 or 360. Finally, the `text` helper is used to add text to the plot. The first two argument give the upper left corner of the text, then the text to be put on the plot. The `adj` argument can be used to adjust the placement of the text, and `cex=0.7` multiplies the standard height of the text by 0.7. This choice of `cex` works well when plotting using R on Windows, but it might be a poor choice using R on Linux or Macintosh. Part b is drawn in much the same way. The graphs you get are likely to be tall and thin, but you can rescale them to be more useful using the mouse by dragging the lower right corner of the window. Finally, `par` is used again to reset to one graph in a plot window.

6.1.2 Using the delta method to estimate a minimum or a maximum

R (See also COMPANION[SEC. 4.4.6]) The command `D` can be used for symbolic differentiation of an expression, and so the delta method can be implemented while allowing the program to do the differentiation; Venables and Ripley (2000, p. 167) discuss this at length. We will use the example from ALR[6.1.2]. The mean function we use is $\mathrm{E}(y|x) = \beta_0 + \beta_1 x + \beta_2 x^2$, and the nonlinear function of interest is $x_m = -2\beta_1/(2\beta_2)$. For data, we will use the cakes data, but ignore the variable X_1.

```
> m4 <- lm(Y ~ X2 + I(X2^2), data=cakes)
> Var <- vcov(m4)
> b0 <- coef(m4)[1]
> b1 <- coef(m4)[2]
> b2 <- coef(m4)[3]
```

We have assigned `b0`, `b1` and `b2` to be, respectively, the estimates of the intercept, linear and quadratic terms, and `Var` is the estimated variance of the coefficients. The `vcov` helper function recovers the matrix $\hat{\sigma}^2(\mathbf{X}'\mathbf{X})^{-1}$ from the fitted linear model.

To differentiate x_m requires several steps. First, define x_m as a string, and then convert it to an expression:

```
> xm <- "-b1/(2*b2)"
> xm.expr <- parse(text=xm)
> xm.expr

expression(-b1/(2 * b2))
```

Since the `b`'s have been assigned values, we can evaluate `xm.expr` to get a point estimate.

```
> eval(xm.expr)

   X2
354.2
```

The point estimate of x_m is 354.2 degrees.

Next, compute the derivatives with respect to each of the three parameters in the mean function, and collect them into a vector.

```
> expr <- expression(-b2/(2*b3))
> derivs <- c(D(xm.expr, "b0"), D(xm.expr,"b1"), D(xm.expr, "b2"))
> derivs

[[1]]
[1] 0

[[2]]
-(1/(2 * b2))

[[3]]
b1 * 2/(2 * b2)^2
```

which are the expressions given before ALR[E6.13]. The derivative with respect to `b0` is zero because `b0` does not appear in the expression. Since the `b`'s have been defined, we can evaluate the vector of derivatives:

```
> eval.derivs<-c(eval(D(xm.expr, "b0")), eval(D(xm.expr, "b1")),
+                eval(D(xm.expr, "b2")))
> eval.derivs
```

```
        I(X2^2)      X2
    0.0    43.6 30888.1
```

The labels printed from the evaluated derivatives are wrong, but we won't use them anyway. Finally, we can compute the estimated standard error of $\hat{x}_m$:

```
> sqrt(t(eval.derivs) %*% Var %*% eval.derivs)
```

```
      [,1]
[1,] 2.089
```

and the standard error is 2.0893.

If this seems excessively difficult, well, I agree. This discussion amounts to an algorithm for computing the delta method, and a command `deltaMethod` has been added to the `car` package that essentially follows this outline.

```
> deltaMethod(m4,"-b1/(2*b2)", parameterNames= c("b0", "b1", "b2"))
```

```
           Estimate    SE
-b1/(2*b2)    354.2 2.089
```

The key arguments to `deltaMethod` are the name of the regression model and a quoted string corresponding to the function of parameters of interest. The names of the parameters are assigned by the `parameterNames` argument. In most regression models you can simply use the names produced by `names(coef(model))` that are printed in the `summary` output for the regression as the names of the parameters. That won't work in this example because the name for the quadratic term `I(X2^2)` includes the exponentiation character and will confuse the `D` function used for differentiation. The `deltaMethod` function will work with linear, generalized linear, nonlinear, and many other regression models.

6.1.3 Fractional polynomials

6.2 FACTORS

(See also COMPANION[SEC. 4.2.3 AND SEC. 4.6]) Factors are a slippery topic because different computer programs will handle them in different ways. In particular, while SAS and SPSS use the same default for defining factors, JMP, R all used different defaults. A factor represents a qualitative variable with say a levels by $a - 1$ (or, if no intercept is in the model, possibly a) dummy variables. ALR[E6.16] describes one method for defining the dummy variables, using the following rules:

1. If a factor A has a levels, create a dummy variables $U_1, \ldots, U_a$, such that U_j has the value one when the level of A is j, and value zero everywhere else.

2. Obtain a set of $a-1$ dummy variables to represent factor A by dropping one of the dummy variables. For example, using the default coding in R, the first dummy variable U_1 is dropped, while in SAS and SPSS the last dummy variable is dropped.

3. JMP uses a completely different method.

Most of the discussion in ALR assumes the R default for defining dummy variables.

R The `factor` command is used to convert a numeric variable into a factor. The command will all its arguments is

```
> args(factor)

function (x = character(), levels, labels = levels, exclude = NA,
    ordered = is.ordered(x))
NULL
```

The argument `x` is the name of the variable to be turned into a factor, and is the only required argument. The argument `levels` is a vector of the names of the levels of `x`; if you don't specify the levels, then the program will use all the unique values of `x`, *sorted in alphabetical order*. Thus you will want to use the `levels` argument if the levels of a factor are, for example, "Low", "Medium" and "High" because by default `factor` will reorder them as "High," "Low" and "Medium." You can avoid this by setting `levels=c("Low","Medium","High")`. The next optional argument `labels` allows you to change the labels for the levels, so, for example, if the levels of the factor are 1, 2 and 3, you can change them with the argument `labels=c("Low","Medium","High")`.

The next argument is `ordered`. If this is set to TRUE, then an *ordered factor* is created. Ordered factors compute the dummy variables as if the levels of the factor were qualitative, not quantitative. Ordered factors are never required for the methodology in ALR, and we recommend that you don't use this argument when creating a factor. Additional arguments concern the handling of missing values; see the help page for more information.

For the sleep data discussed in ALR[6.2], the variable D is a qualitative variable with five levels, given by the integers 1, 2, 3 4 and 5. It can be declared a factor using

```
> sleep1$D <- factor(sleep1$D)
```

The terms for the factor created by R corresponds to our preferred method, dropping the dummy variable for the first level, and then using the remaining U_j.

If you use different definitions for the dummy variables for a factor:

1. Coefficient estimates are different because the terms in the mean function have different meaning. Consequently, t-statistics for individual terms are also different.

2. F-tests for the factor or for other terms in the mean function are the same regardless of the way the factor is defined.

6.2.1 No other predictors

R ALR[T6.1] is obtained by

```
> sleep1$D <- factor(sleep1$D)
> a1 <- lm(TS ~ D, sleep1)
> a0 <- update(a1, ~ . -1)
> summary(a0)

Call:
lm(formula = TS ~ D - 1, data = sleep1)

Residuals:
   Min     1Q Median     3Q    Max 
-7.010 -2.250 -0.341  2.610  6.817 

Coefficients:
   Estimate Std. Error t value Pr(>|t|)
D1   13.083      0.888   14.73  < 2e-16
D2   11.750      1.007   11.67  2.9e-16
D3   10.310      1.191    8.65  1.0e-11
D4    8.811      1.256    7.02  4.3e-09
D5    4.071      1.424    2.86   0.0061

Residual standard error: 3.77 on 53 degrees of freedom
  (4 observations deleted due to missingness)
Multiple R-squared: 0.902,       Adjusted R-squared: 0.892 
F-statistic: 97.1 on 5 and 53 DF,  p-value: <2e-16 

> summary(a1)

Call:
lm(formula = TS ~ D, data = sleep1)

Residuals:
   Min     1Q Median     3Q    Max 
-7.010 -2.250 -0.341  2.610  6.817 
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   13.083      0.888   14.73  < 2e-16
D2            -1.333      1.343   -0.99   0.3252
D3            -2.773      1.486   -1.87   0.0675
D4            -4.272      1.538   -2.78   0.0076
D5            -9.012      1.678   -5.37  1.8e-06

Residual standard error: 3.77 on 53 degrees of freedom
  (4 observations deleted due to missingness)
Multiple R-squared: 0.378,       Adjusted R-squared: 0.331
F-statistic: 8.05 on 4 and 53 DF,  p-value: 3.78e-05
```

Since the data frame `sleep1` has missing values, we used `na.action=na.omit` to delete species with missing data. The dummy variables are labeled according to the name of the factor and the level for that dummy variable. For example, D2 corresponds to the variable U_2 in ALR[T6.1B]. If the factor is defined using different variables then the labels for the terms are also different.

To produce the "pretty" output shown in ALR[T6.1] we used the R package `xtable` that takes standard R output and gives a table that can be inserted into a LaTeX document for printing. If this is of interest to you, see the script for this chapter for the exact commands used.

6.2.2 Adding a predictor: Comparing regression lines

R The discussion in ALR[6.2.2] uses the appropriate R formulas for the models discussed.

6.3 MANY FACTORS

6.4 PARTIAL ONE-DIMENSIONAL MEAN FUNCTIONS

ALR[F6.8] is much more compelling in color, and is shown here as Figure 6.1.

R The `alr3` package includes a command for fitting the partial one-dimensional or POD models described in ALR[6.4]. We illustrate the use of this command with the Australian Athletes data, and reproduce ALR[F6.8] in color.

```
> m <- pod(LBM~Ht+Wt+RCC, data=ais, group=Sex, mean.function="pod")
> anova(m)

POD Analysis of Variance Table for LBM, grouped by Sex

1: LBM ~ Ht + Wt + RCC
```

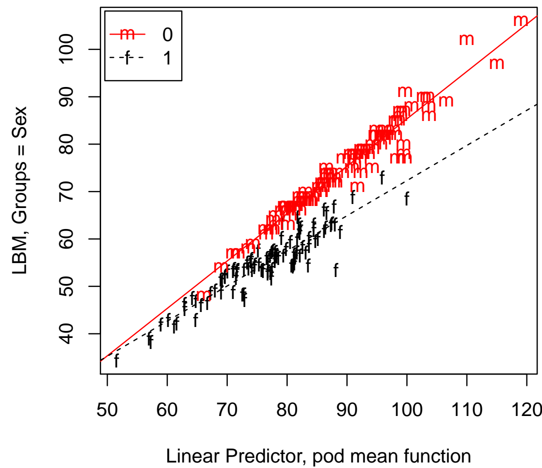


Fig. 6.1 Figure ALR[F6.8], in color.

```
2: LBM ~ Ht + Wt + RCC + Sex
3: LBM ~ eta0 + eta1 * Ht + eta2 * Wt + eta3 * RCC + Sex1 * (th02 +
3:     th12 * (eta1 * Ht + eta2 * Wt + eta3 * RCC))
4: LBM ~ Ht + Wt + RCC + Sex + Ht:Sex + Wt:Sex + RCC:Sex
             Res.Df  RSS Df Sum of Sq      F Pr(>F)
1: common       198 2937
2: parallel     197 1457  1      1479 245.65 <2e-16
3: pod          196 1186  1       272  45.09  2e-10
4: pod + 2fi    194 1168  2        18   1.47   0.23

> plot(m,pch=c("m","f"),colors=c("red","black"))
```

The `pod` command requires that you specify a formula, giving the response on the left and the variables on the right that make up the linear combination of terms. A new feature of this command is the required `group` argument that specifies the grouping variable. In this case, the linear predictor will be of the form $\beta_1 Ht + \beta_2 Wt + \beta_3 RCC$, and it will be fit for each level of *Sex*. The argument `mean.function` is used to select which form of the POD mean function is to be used. The default is `mean.function="pod"`, which fits the partial one-dimensional model to the groups, given for the example by

$$\begin{aligned}\mathrm{E}(LBM|\mathrm{pod}) &= \beta_0 + \beta_1 Ht + \beta_2 Wt + \beta_3 RCC + \\ &\quad \beta_4 Sex1 + \beta_5 Sex1 \times (\beta_1 Ht + \beta_2 Wt + \beta_3 RCC)\end{aligned}$$

where *Sex1* is a dummy variable equal to one when the grouping variable *Sex* is equal to one and zero otherwise.

Three other choices of mean function are available. `mean.function="common"` that ignores the grouping variable, essentially equivalent to Model 4 in ALR[6.2.2]. For the example, it is given by

$$\mathrm{E}(LBM|\text{common}) = \beta_0 + \beta_1 Ht + \beta_2 Wt + \beta_3 RCC$$

`mean.function="parallel"` that fits a mean function that is parallel within group, like Model 2 in ALR[6.2.2]. For the example, it is

$$\mathrm{E}(LBM|\text{parallel}) = \beta_0 + \beta_1 Ht + \beta_2 Wt + \beta_3 RCC + \beta_4 Sex1$$

Finally, `mean.function="general"` fits a separate coefficient for each predictor in each group. This represents a larger, and more complex, mean function than the POD mean function.

$$\begin{aligned}\mathrm{E}(LBM|\text{general}) &= \beta_0 + \beta_1 Ht + \beta_2 Wt + \beta_3 RCC + \beta_4 Sex1 + \\ &\quad \beta_5 Sex1 \times Ht + \beta_6 Sex1 \times Wt + \beta_7 Sex1 \times RCC\end{aligned}$$

Additional arguments to `pod` are the same as for `lm`, and include `data` to specify a data frame, `subset` to specify a subset of cases to use, and so on.

The `anova` helper command for POD models is a little different from the default method. Three F-tests are shown in the table; the first compares the first two mean functions; the second compares the parallel line mean function to the POD mean function and the third compares the POD mean function to the larger mean function with separate coefficients for each group. All F-tests use the error term from the largest mean function as the denominator. In this example, the POD mean function is a clear improvement over the parallel regression mean function, but the more general mean function is not an improvement over the POD mean function.

The `plot` method for POD models will produce the equivalent of Figure **??**. The full complement of arguments and their defaults are for R are given by:

```
> plot(x, colors = rainbow(nlevels(x$group)),
+      pch = 1:nlevels(x$group), key = FALSE,
+      xlab = "Linear Predictor",
+      ylab = as.character(formula(x)[[2]]), ...)
```

Any additional arguments are graphical parameters. In this command, the argument `x` is the result of a call to the `pod` command. If `key=TRUE`, then you can add a key (or legend) to the plot by clicking the mouse in the location you want the key to appear.

Finally, `pod` models have all the usual helper commands, like `residuals`, `fitted`, `predict` and `update` that work in the usual way.

6.5 RANDOM COEFFICIENT MODELS

R The random coefficient model can be fit using the `nlme` package. This package is part of the base distribution of R. Here are the R commands that give the computations discussed in ALR[6.5]; see Pinheiro and Bates (2000) for complete documentation. This example also uses lattice graphics; see VR[4.5].

```
> library(nlme)
> library(lattice)
> data(chloride)
> xyplot(Cl ~ Month|Type, group=Marsh, data=chloride,
+        ylab="Cl (mg/liter)",
+        panel.groups=function(x,y,...){
+               panel.linejoin(x,y,horizontal=FALSE,...)
+               }
+        )
> m1 <- lme(Cl ~ Month + Type, data=chloride,
+        random=~ 1 + Type|Marsh)
> m2 <- update(m1, random= ~ 1|Marsh)
```

The model `m1` fits a separate intercept for each *Type*, since it is a factor, and a single slope for *Month*. In the `random` arguement, we specify a variance term `1` for a separate random intercept for each marsh, and `Type`, which allows the variation between marshes to be different for the two types. Model `m2` is identical except the variation between marshes in a type is assumed to be the same. These two models can be compared by an analysis of variance:

```
> anova(m1,m2)

   Model df   AIC   BIC logLik   Test L.Ratio p-value
m1     1  7 228.1 237.7 -107.1
m2     2  5 229.7 236.5 -109.8 1 vs 2   5.554  0.0622

> intervals(m2)

Approximate 95% confidence intervals

 Fixed effects:
                lower   est.  upper
(Intercept)  -21.5084 -5.504 10.501
Month          0.7549  1.854  2.953
TypeRoadside  28.6917 50.572 72.452
attr(,"label")
[1] "Fixed effects:"

 Random Effects:
  Level: Marsh
```

```
                 lower  est. upper
sd((Intercept)) 7.576 13.34 23.47

 Within-group standard error:
lower  est. upper
4.750 6.387 8.588
```

According to the anova table, the p-value is about 0.06 for the hypothesis that the variance between marshes is the same for the two types of march, versus the alternative that it is different. Continuing with the simpler model `m2`, we get estimates and confidence intervals for the population slope and intercept, for the variation in intercepts between marshes, and for the within-march variance.

7 Transformations

R This chapter uses graphs and smoothers in ways that are uncommon in most most statistical packages, including R. Using these methods in R generally requires a sequence of commands that would be too tedious for most people to use on a regular basis.

Our solution to this problem is to automate this process with several commands that are included in the `car` package. These commands become available when you use the command `library(alr3)` to load the data files. In the main part of this chapter we show how to use these commands, along with an outline of what they do. Here is a description of coming attractions.

Transformation families We work with three families of transformations. The basic power family is computed with the function `basicPower`; the Box-Cox power family is computed with `bcPower`, and the Yeo-Johnson family, which can be used if the variable includes non-positive values, is computed with `yjPower`.

Transforming predictors The `invTranPlot` function provides a graphical method to transform a single predictor for linearity. The `powerTransform` function provides a method for transforming one or more variables for normality. It also permits conditioning on other, non-transformed variables.

Although these function are adequate for almost all applications, additional function are available in the `car` package that are not discussed in ALR, including `boxTidwell` which is essentially a numerical generalization of `invTranPlot`, `crPlots` for component plus residual plots.

Transforming the response The `boxcox` command in the `MASS` package implements the Box-Cox method of finding a power transformation. The function `boxCox` with a capital "C" extends the method to allow for using the Yeo-Johnson family of transformations. The `powerTransform` command provides numeric, rather than graphical, output, for the same method. The `invResPlot` is used to transform the response for linearity using a graph.

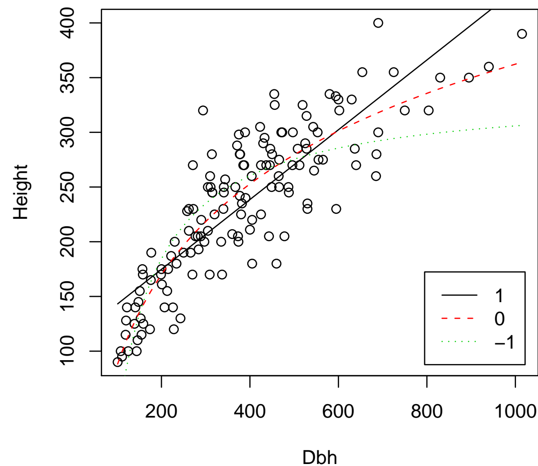
7.1 TRANSFORMATIONS AND SCATTERPLOTS

7.1.1 Power transformations

7.1.2 Transforming only the predictor variable

R To draw a graph like ALR[F7.3] with both points and several lines, you must first draw the graph with the `plot` command, and then add the lines with the `lines` helper command.

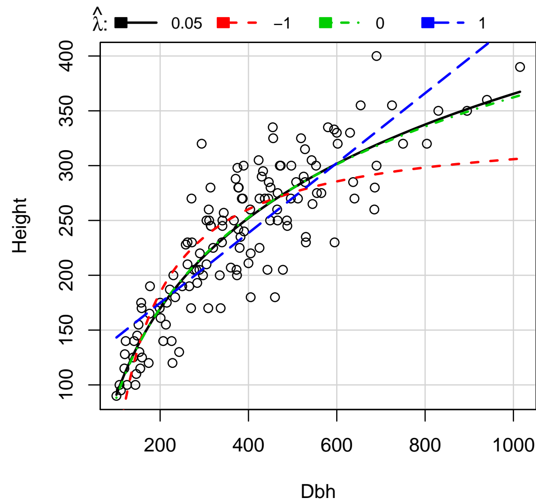
```
> with(ufcwc, plot(Dbh, Height))
> lam <-c(1, 0, -1)
> new.dbh <- with(ufcwc, seq(min(Dbh), max(Dbh), length=100))
> for (j in 1:3){
+    m1 <- lm(Height ~ bcPower(Dbh,lam[j]), data=ufcwc)
+    lines(new.dbh,
+       predict(m1, data.frame(Dbh=new.dbh)), lty=j, col=j)
+    }
> legend("bottomright", inset=.03, legend = as.character(lam),
+         lty=1:3, col=1:3)
```



We draw the plot and add three lines for $\lambda \in (-1, 0, 1)$. The variable `new.dbh` gives 100 equally spaced values between the minimum and maximum of *Dbh*, for plotting positions. The `for` loop draws the lines. First, we use `lm` to fit the model implied by the transformation. The command `bcPower`, from the `car` package, transforms the predictor. The `lines` helper function adds the line to the plot. Since `m1` fits the regression of the response on $\psi_S(Dbh, \lambda)$, in `predict` we need only specify values for *Dbh*. The `predict` helper will then figure out how to compute $\psi_S(Dbh, \lambda)$ from the values of *Dbh* provided. A `legend` (or key) was added to the plot. The helper command `locator` expects input from the user by clicking the mouse on the graph. This point will be the upper-left corner of the legend.

The `car` package includes a function called `invTranPlot` that automates this procedure.

```
> with(ufcwc, invTranPlot(Dbh, Height))

    lambda    RSS
1  0.04788 152123
2 -1.00000 197352
3  0.00000 152232
4  1.00000 193740
```



The first two arguments are the predictor and the response, respectively. After the graph is drawn, the residual sum of squares for each value of λ used in the plot is printed, including by default the value of λ that minimizes the residual sum of squared errors. The function `invTranEstimate` can be called directly to get the optimal λ without drawing the graph,

```
> unlist(with(ufcwc, invTranEstimate(Dbh,Height)))
```

```
  lambda  lowerCI  upperCI
 0.04788 -0.16395  0.25753
```

We get $\hat{\lambda} = 0.048$, close to the logarithmic transformation selected visually, with standard error 0.15. The minimum value of RSS =152,123. The `unlist` command made the output prettier.

The command `bcPower` is used by `invTranEstimate` and by `invTranPlot` to compute the transformation. The form of this command is

```
> bcPower(U, lambda, jacobian.adjusted=FALSE)
```

where `U` is a vector, matrix or data frame, `lambda` is the transformation parameter. If `U` is a vector, then `lambda` is just a number, usually in the range from -2 to 2; if `U` is a matrix or a data, frame, then `lambda` is a vector of numbers. The function returns $\psi_S(U, \lambda)$, as defined at ALR[E7.3]. If you set `jacobian.adjusted=TRUE` then $\psi_M(U, \lambda)$, from ALR[E7.6], is returned.

The `yjPower` command is similar but uses the Yeo-Johnson family $\psi_{YJ}(U, \lambda)$, ALR[7.4]. Finally, use `basicPower` for the power transformations $\psi(U, \lambda)$. For example, to plot a response Y versus $\psi_M(U, 0)$, type

```
> plot(bcPower(U, 0, jacobian.adjusted=FALSE), Y)
```

7.1.3 Transforming the response only

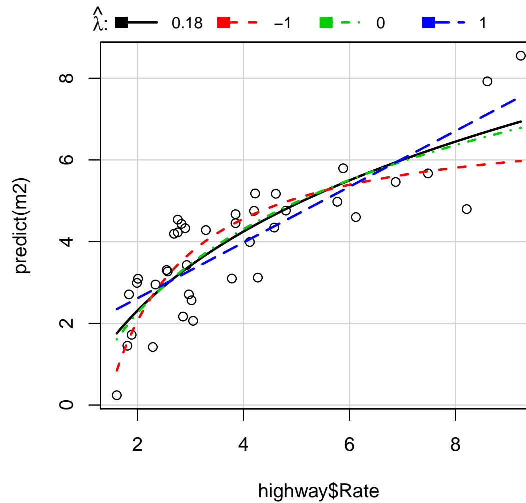
R The method described ALR[7.1.3] requires the steps: (1) fit the model with the response untransformed, and predictors transformed; (2) draw the inverse plot with fitted values on the horizontal axis, and the untransformed response on the vertical axis; (3) estimate the transformation by fitting a smoother to this graph. The only "smoother" we will discuss is fitting

$$\mathrm{E}(\hat{Y}|Y) = \alpha_0 + \alpha_1 \phi_S(Y, \lambda)$$

which parallels the discussion for a predictor transformation.

For an example, we will use the highway accident data. We will assume the transformations of predictors found in ALR[7.2.2]

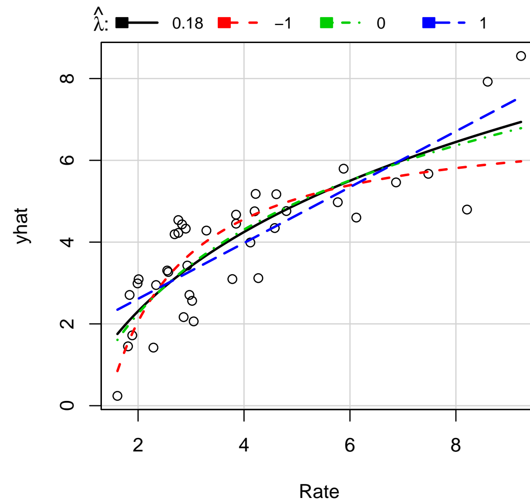
```
> highway <- transform(highway, logLen=log2(Len),
+            logADT=log2(ADT), logTrks=log2(Trks),
+            logSigs1=log2((Len*Sigs+1)/Len))
> m2 <- lm(Rate ~ logLen + logADT + logTrks + Slim + Shld + logSigs1,
+          data=highway)
> invTranPlot(highway$Rate,predict(m2))

   lambda   RSS
1  0.1846 30.62
2 -1.0000 34.72
3  0.0000 30.73
4  1.0000 32.46

> unlist(invTranEstimate(highway$Rate,predict(m2)))

 lambda lowerCI upperCI
 0.1846 -0.5273  0.9279
```



We used the **transform** to add new columns required in ALR[7.2.2] to the **highway** data frame. The first argument is the name of the data frame, and the remaining arguments will be the labels of the newly created columns. For example, **logLen** will be the base-two logarithm of **Len**. The regression is fit with the “wrong” response. We then used the **invTranPlot** command that we used in the last section when transforming the predictor to get ALR[F7.7]. The **invTranEstimate** was then used provide a confidence interval for the value of λ that minimizes the residual sum of squares. The numerical estimate agrees with the visual assessment of ALR[F7.7]: the best values of λ are close to zero, but there is lots of variation here, reflected in the large standard error.

This process is automated with the **invResPlot** command in the **car** package,

```
> invResPlot(m2)

   lambda   RSS
1  0.1846 30.62
2 -1.0000 34.72
3  0.0000 30.73
4  1.0000 32.46
```



This will produce the estimate from `invTranEstimate`, and the plot from `invTranPlot`. See the help page for all the options.

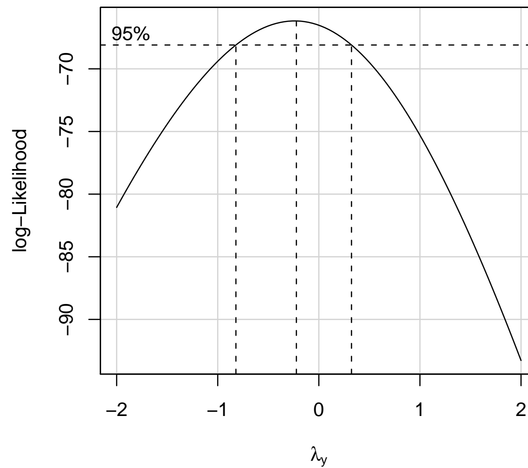
7.1.4 The Box and Cox method

R A very easy to use the Box and Cox method for automatically selecting λ is described by VR[6.8] and COMPANION[SEC. 6.4.1] using the `car` function `boxCox`.

Continuing with the highway data from the last section,

```
> boxCox(m2, xlab=expression(lambda[y]))
> summary(powerTransform(m2))

bcPower Transformation to Normality

   Est.Power Std.Err. Wald Lower Bound Wald Upper Bound
Y1   -0.2384   0.2891           -0.805           0.3282

Likelihood ratio tests about transformation parameters
                           LRT df       pval
LR test, lambda = (0)  0.6884  1 4.067e-01
LR test, lambda = (1) 18.2451  1 1.942e-05
```

The `powerTransform` command, with no arguments beyond the name of the regression model, gives a numeric summary corresponding to the graphical summary in `boxCox`.



7.2 TRANSFORMATIONS AND SCATTERPLOT MATRICES

The scatterplot matrix is the central graphical object in learning about regression models. You should draw them all the time; all problems with many continuous predictors should start with one.

R The primary command in R for drawing scatterplot matrices is the `pairs` command. The basic use of `pairs` requires that we specify the matrix or data frame to be plotted. For example, apart from a few statements added to improve formatting for the book, ALR[F7.5] is obtained from

```
> pairs(~ Rate + Len + ADT + Trks + Slim + Shld +
+         Sigs, data=highway)
```

The only required argument to `pairs` is a formula, here a *one-sided formula*, in which all the variables to be plotted appear to the right of the "~". You can replace the formula with a matrix or data-frame.

There are many enhancements possible for a scatterplot matrix, such as using colors or symbols for different groups of points, adding smoothers, for example, OLS as a dashed line and *loess* as a connected line, and putting histograms on the diagonal. In the `car`, there is a function called `scatterplotMatrix`, COMPANION[SEC. 3.3.2] that adds a dizzying array of arguments for fine control over a scatterplot matrix, including these smoothers and histograms by default. The defaults for `scatterplotMatrix` are generally sensible, and will produce the plot shown in Figure 7.1,

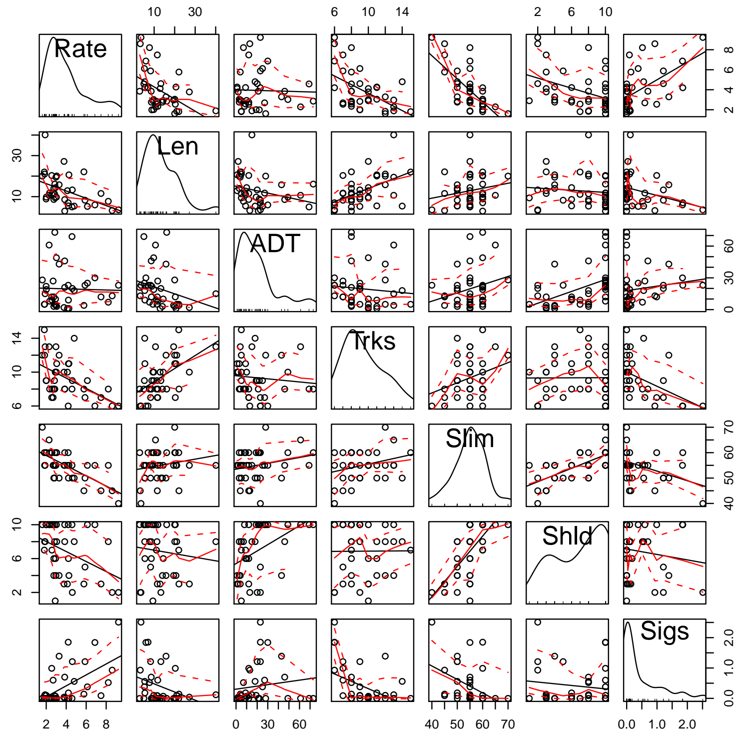


Fig. 7.1 Scatterplot matrix using the `scatterplotMatrix` function from `car`.

```
> scatterplotMatrix( ~ Rate + Len + ADT + Trks + Slim + Shld +
+                       Sigs, data=highway)
```

One very useful argument to `scatterplotMatrix` is `groups`. Setting `groups=Hwy` results in a different color and symbol used for each unique value of the grouping variable. A legend is also added to the plot, but it can be hard to read and can be suppressed using the `legend.plot=FALSE` argument. An alias of `scatterplotMatrix` is `spm`.

7.2.1 The 1D estimation result and linearly related predictors

7.2.2 Automatic choice of transformation of the predictors

R One important use of scatterplot matrices is to help select transformations toward normality, with the goal of getting approximately linearly related predictors, ALR[7.2.1]. The `car` package includes one basic command `powerTransform` for this purpose, along with several helper commands that

work with it. Continuing with the highway data, suppose we want to consider transforming the five variables *Len*, *ADT*, *Trks*, *Shld*, *Sigs1* discussed in ALR[7.2.2]. The first step is to draw the scatterplot matrix discussed above, and shown in Figure 7.1 and ALR[F7.5]. Since curvature is apparent in plots of predictors versus predictors, transformations can help. We can next use Velilla's multivariate extension of the Box-Cox method to find starting guesses for transformations. This is what `powerTransform` does:

```
> highway$Sigs1 <-
+      (round(highway$Sigs*highway$Len) + 1)/highway$Len
> ans <- powerTransform(cbind(Len, ADT, Trks, Shld, Sigs1) ~ 1,
+      data=highway)
> summary(ans)

bcPower Transformations to Multinormality

      Est.Power Std.Err. Wald Lower Bound Wald Upper Bound
Len      0.1429   0.2126          -0.2738           0.5596
ADT      0.0501   0.1204          -0.1860           0.2862
Trks    -0.7019   0.6178          -1.9129           0.5090
Shld     1.3455   0.3631           0.6339           2.0571
Sigs1   -0.2440   0.1498          -0.5377           0.0496

Likelihood ratio tests about transformation parameters
                                  LRT df      pval
LR test, lambda = (0 0 0 0 0)  23.373  5 0.0002864
LR test, lambda = (1 1 1 1 1) 133.179  5 0.0000000
LR test, lambda = (0 0 0 1 0)   6.143  5 0.2925204
```

In the first line, we created the variable *Sigs1* because it did not appear in the data frame. We used the `round` command to make sure that $Sigs \times Len$, which is the number if signals, is exactly an integer.

The syntax for transformating predictors has changed from older versions of `alr3`. The variables to be used in `powerTransform` are specified using a formula. On the left-hand side is a matrix giving all the variables that may be transformed. The right-hand side will generally just be `1`, indicating that we want marginal transformations toward normality without conditioning on any predictors. The right-side can be any valid linear model, and transformations will be done conditional on the linear model. *Only the predictors on the left-hand side are transformed.* The `data`, `subset` and `na.action` items are used as in `lm`. We used the `summary` helper to print the answer; this is almost the same as ALR[T7.2], but it now includes confidence intervals about the estimated transformations rather than Wald tests.

Examining the output, it appears that all the estimated powers are close to zero (for logarithms) except for *Shld*. Reasonable rounded values for the transformations are automatically tested by the function with a p-value close

to 0.3. You could expicity get a test for any value you like. For example, for $\boldsymbol{\lambda} = (0, 0, -1, 1, 0)$.

```
> testTransform(ans,lambda=c(0, 0, -1, 1, 0))

                                   LRT df   pval
LR test, lambda = (0 0 -1 1 0) 4.393  5 0.4944
```

The large p-value suggests that is value is also a plausible choice for the transformations.

You can add the transformed values to the data frame using the following rather obscure command:

```
> highway <- transform(highway,
+        basicPower(highway[,names(coef(ans))],
+              coef(ans, round=TRUE)))
```

This uses the `transform` command to add columns to `highway`, as the basic power transformations using the rounded powers. The `names(coef(ans))` gets the names of the columns of `highway` that are to be transformed. We used basic powers because these are easier to understand and are equivalent in modeling to the Box-Cox powers[1] You should examine the transformed predictors in a scatter plot matrix to make sure the transformations appear satisfactory.

7.3 TRANSFORMING THE RESPONSE

R See Sections 7.1.3-7.1.4 above for the examples in this section.

7.4 TRANSFORMATIONS OF NON-POSITIVE VARIABLES

R All the transformation functions in `car`car can be used with the Yeo-Johnson family, usually with the `family="jyPower"` argument. An alternative to the Yel-Johnson family is to add a constant to the response to make it strictly positive, but this latter method is relatively arbitrary, and little information is generally available in the data to help select the added constant.

[1] If some of the rounded powers had been negative, the Box-Cox powers might be preferred to avoid changing the sign of the corresponding regression coefficient.

8

Regression Diagnostics: Residuals

8.1 THE RESIDUALS

R Suppose you have fit a linear model using `lm`, and named it, for example, `m1`. The commands listed in Table 8.1 give helper functions to get the quantities described in this chapter or in the next chapter.

You can use these quantities in graphs or for any other purpose. For example,

```
> plot(predict(m1), residuals(m1, type="pearson"))
> plot(predict(m1), predict(m2))
> press <- residuals(m1) / (1 - hatvalues(m1))
```

The first is the usual plot of residuals versus fitted values, and the second is the plot of fitted values from `m2` versus the fitted values from `m1`, and the third computes and saves the *PRESS* residuals, ALR[P9.4].

8.1.1 Difference between $\hat{\mathrm{e}}$ and e

8.1.2 The hat matrix

R The hat-matrix $\mathbf{H}$ is rarely computed in full because it is an $n \times n$ matrix that can be very large. Its diagonal entries, the leverages, are computed simply by `hatvalues(m1)` for a model `m1`. Should you wish to compute $\mathbf{H}$ in full, you can use the following commands. First, suppose that X is an $n \times p$ matrix. Then

Table 8.1 R commands relating to residuals. For example, `residuals(m1)` will return the residuals from the model `m1`.

	Quantities described in ALR[8]
`predict`	Fitted values, ALR[E8.3] for OLS and WLS.
`residuals`	Residuals, ALR[8.4] for OLS only.
`residuals(m1,type="pearson")`	Residuals, ALR[E8.13], for WLS. For OLS, the Pearson residuals and the ordinary residuals are the same, so this option can be used for all least squares models.
`hatvalues`	Leverages, ALR[E8.11], for OLS and WLS. In S-Plus, use `lm.influence(m1)$hat`.
Quantities described in ALR[9]. All of these functions are available in the base version of R. If you are using S-Plus, we recommend you use the `car` library. These commands are then available for S-Plus as well.	
`rstandard`	Standardized residual, ALR[E9.3].
`rstudent`	Studentized residuals, ALR[E9.4].
`cooks.distance`	Cook's distance, ALR[E9.6].

```
> decomp <- qr(cbind(rep(1,n),X))
> Hat <- qr.Q(decomp) %*% t(qr.Q(decomp))
```

This is probably nonintuitive, so here is an explanation. First, append a column of ones to X for the intercept. Then, compute the QR decomposition, ALR[A.9], of the augmented matrix. The helper function `qr.Q` returns Q, and the second command is just ALR[EA.27].

8.1.3 Residuals and the hat matrix with weights

As pointed out in ALR[8.1.3], the residuals for WLS are $\sqrt{w_i} \times (y_i - \hat{y}_i)$. Whatever computer program you are using, you need to check to see how residuals are defined.

R In R, the correct residuals for WLS are given by `residuals(m1,type="pearson")`. These are also correct for OLS, and so these should always be used in graphical procedures.

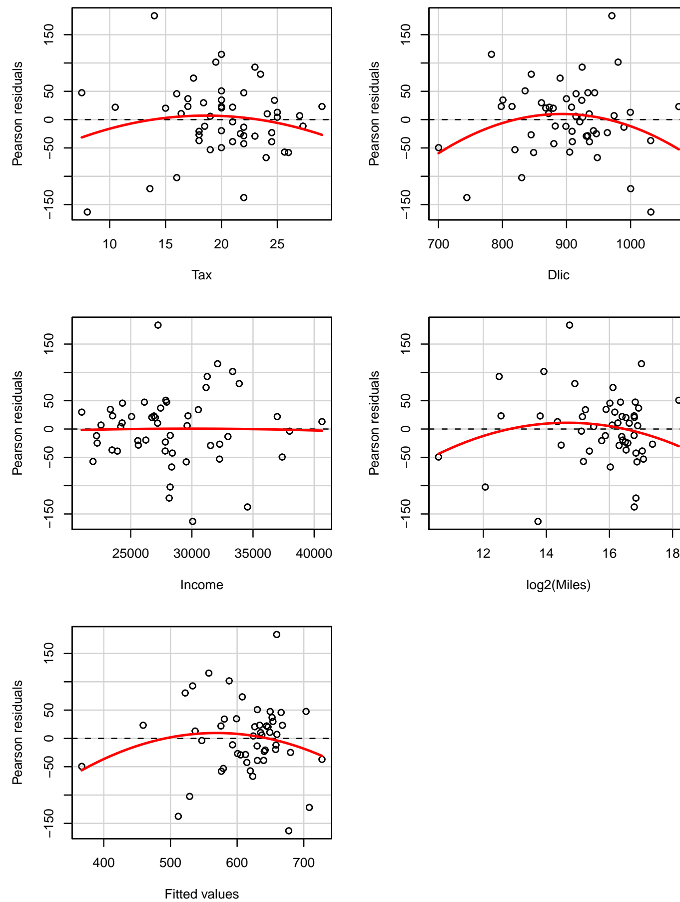
8.1.4 The residuals when the model is correct

8.1.5 The residuals when the model is not correct

8.1.6 Fuel consumption data

R The figure ALR[F8.5] has six separate graphs. The first four plots are plots of residuals versus each of the four predictors in turn, the fifth is a plot of residuals versus fitted values and the last a *summary graph* of the response versus fitted values. The first five of these plots are drawn automatically using the `plotResiduals`

```
> fuel2001 <- transform(fuel2001, Fuel = 1000 * FuelC / Pop,
+   Dlic = 1000 * Drivers / Pop)
> m1 <- lm(Fuel ~ Tax + Dlic + Income + log2(Miles), data=fuel2001)
> residualPlots(m1)
```



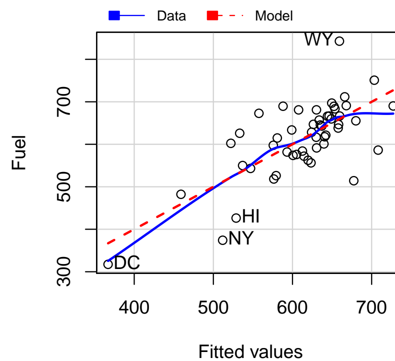
A new feature of this command in the `car` package is *automatic point marking*. In each of the graphs the labels of the `id.n` points with the largest absolute residuals and the `id.n` points with the most extreme values on the horizontal axis are labeled with their case labels. You can turn the labeling off with the argument `id.n=0` on the call to `plotResiduals`. You can also control which plots are shown if you don't want to see all of them, or control the layout; see the help page for `plotResiduals`. For example,

```
> residualPlots(m1, ~ log2(Miles), fitted=FALSE)
> residualPlots(m1, ~ 1, fitted=TRUE)
```

The first of these would plot only versus `log2(Miles)`, and the second would plot only against fitted values.

The last of the plots, of the response versus the fitted values, is the default plot produced by the `mmp` or its equivalent `marginalModelPlot` command:

```
> marginalModelPlot(m1, id.n=4)
```



In this case we set `id.n=4` to label more points.

R has a mechanism for identifying points that is fairly general, but clumsy and not very pleasant to use. The basic outline is:

1. Draw a graph. The method works best with a scatterplot, so that is the only case we consider. Let's suppose the plot was created by

   ```
   > plot(x, y)
   ```

2. While the graph is on the screen, use the `identify` helper to identify points. The basic command is

   ```
   > identify(x, y, labels)
   ```

   where `labels` is a vector of labels or numbers of the same length as `x` and `y` giving the labels that will be used to identify the points. Nothing seems to happen when you type this command. The program is waiting for you to click the mouse button on the graph. If you click close enough to a point, the label corresponding to the nearest point will be printed on the graph. This will continue until you press "escape," select Stop → Stop locator from the plot's menu, or select Stop after clicking the right mouse button.

The mechanism for labelling points noninteractively uses the `text` command; see the script for this chapter for an example.

We have added in the `car` package a function called `showLabels` that can be used to label points either interactively or automatically. This function is

called by all the `car` graphics functions, but you can use it with any graph you create with `plot`. See the on-line documentation for `showLabels`, or COMPANION[SEC. 3.5].

R has its own standard suite of residual plots produced by the `plot` helper. Some of these "default" plots are not discussed in ALR, and so we will not discuss them further here, either.

8.2 TESTING FOR CURVATURE

R The command `residualPlots` illustrated above also implements the curvature tests described in ALR[8.2].

```
> residualPlots(m1)
```

```
            Test stat Pr(>|t|)
Tax            -1.077    0.287
Dlic           -1.922    0.061
Income         -0.084    0.933
log2(Miles)    -1.347    0.185
Tukey test     -1.446    0.148
```

8.3 NONCONSTANT VARIANCE

8.3.1 Variance Stabilizing Transformations

8.3.2 A diagnostic for nonconstant variance

R Here is the complete outline of the computations for the test for heteroscedasticity for the snow geese data discussed in ALR[8.3.2]:

```
> m1 <- lm(photo ~ obs1, snowgeese)
> sig2 <- sum(residuals(m1, type="pearson")^2) /
+             length(snowgeese$obs1)
> U <- residuals(m1)^2 / sig2
> m2 <- update(m1, U ~ .)
> anova(m2)
```

```
Analysis of Variance Table

Response: U
          Df Sum Sq Mean Sq F value  Pr(>F)
obs1       1    163   162.8    50.8 8.5e-09
Residuals 43    138     3.2
```

After getting the data, the model with term `obs1` is fit. The quantity `sig2` is the estimate of σ^2, using n as a denominator rather than $n-p'$. Using `pearson`

residuals make this computation correct even for WLS. The score variable U is computed by the next command, and then `m2` is the regression of U on the terms in the original model, using the `update` helper. The score test is 1/2 the sum of squares for regression, for $(1/2)162.83 = 81.41$.

The `ncvTest` automates computing the score test for nonconstant variance. The command called `ncvTest` is part of the `car` package. For the snow geese data,

```
> ncvTest(m1)

Non-constant Variance Score Test
Variance formula: ~ fitted.values
Chisquare = 81.41    Df = 1     p = 1.831e-19
```

In the sniffer data, we would like to consider more variance functions. Some of the results in ALR[T8.4] can be obtained as follows:

```
> m1 <- lm(Y ~ TankTemp + GasTemp + TankPres + GasPres, sniffer)
> ncvTest(m1, ~ TankTemp, data=sniffer)

Non-constant Variance Score Test
Variance formula: ~ TankTemp
Chisquare = 9.706    Df = 1     p = 0.001837

> ncvTest(m1, ~ TankTemp + GasPres, data=sniffer)

Non-constant Variance Score Test
Variance formula: ~ TankTemp + GasPres
Chisquare = 11.78    Df = 2     p = 0.002769

> ncvTest(m1, ~ TankTemp + GasTemp + TankPres +
+                GasPres, data=sniffer)

Non-constant Variance Score Test
Variance formula: ~ TankTemp + GasTemp + TankPres + GasPres
Chisquare = 13.76    Df = 4     p = 0.008102

> ncvTest(m1)

Non-constant Variance Score Test
Variance formula: ~ fitted.values
Chisquare = 4.803    Df = 1     p = 0.02842
```

To use `ncvTest`, specify the regression model, then a one-sided formula that specifies the model for the variance, and then if necessary the data frame.

8.3.3 Additional comments

8.4 GRAPHS FOR MODEL ASSESSMENT

8.4.1 Checking mean functions

R The `car` package has two commands for drawing marginal model plots; `mmp` draws *one* marginal model plot, while `mmps` draws many such plots. You can also use the full names of these functions, `marginalModelPlot` and `marginalModelPlots`, respectively.

ALR[F8.11] is created by

```
> c1 <- lm(Height ~ Dbh, ufcwc)
> mmp(c1, ufcwc$Dbh, label="Diameter, Dbh",
+         col.line=c("blue", "red"))
```

The first argument to `mmp` is the only required argument, and is the name of a regression model. The second argument is the quantity to put on the horizontal axis of the plot, with `predict(c1)` as the default. We have shown two additiona arguments, for the horizontal axis label and the colors of the two smooths; neither argument is required, and indeed blue and red are the default colors.

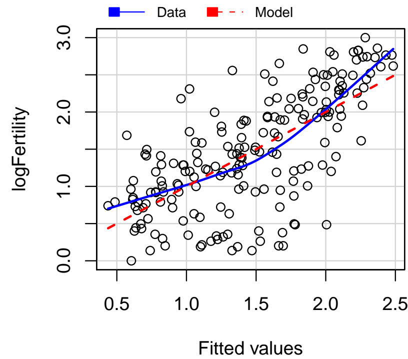
The command `mmp` uses the `loess` smoother, with defaults `degree=1, span=2/3`; both of these values can be changed when using the command. An interesting exercise would be to implement marginal model plots with other smoothers, and with non-deterministic methods for selecting a bandwidth parameter.

To draw marginal model plots versus each predictor and versus fitted values, use `mmps`. The following will draw three of the four plots in ALR[F8.13]:

```
> m1 <- lm(logFertility ~ logPPgdp + Purban, UN2)
> mmps(m1)
```

The fourth plot in ALR[F8.13] plots in a direction that is a random linear combination of the terms. You can get plots like this one using the `randomLinComb` command in the `alr3` package,

```
> mmp(m1, u=randomLinComb(m1))
```



Every time you draw this last plot, it will be different because the random numbers used will change. The list of names returned are the case names of identified points. You can suppress point labels the the `id.n=0` argument. If you want to reproduce a plot, use the `seed` argument:

```
> mmp(m1, u=randomLinComb(m1, seed=254346576))
```

8.4.2 Checking variance functions

R The commands `mmp` and `mmps` include an argument `sd` that if set to `sd=TRUE` will add the standard deviation smooths to the plot, as described in ALR[8.4.2].

9

Outliers and Influence

COMPANION[CHAPTER 6] covers much of the material in the chapter at greater length.

9.1 OUTLIERS

9.1.1 An outlier test

R The function `rstandard` computes the standardized residuals ALR[E9.3]. You can also use ALR[E9.3] to compute these directly. Given a model `m1`,

```
> ri <- residuals(m1, type="pearson") /
+            (sigma.hat(m1) * sqrt(1 - hatvalues(m1)))
```

Pearson residuals guarantee that the same formula will work with weighted or unweighted least squares.

The outlier test is computed with `rstudent`. These, too, can be computed directly. For a model `m1`, we use ALR[E9.4],

```
> ti <- ri * sqrt((m1$df-1)/(m1$df - ri^2))
```

The quantity `m1$df` is the df for residuals, equal to $n - p'$.

9.1.2 Weighted least squares

9.1.3 Significance levels for the outlier test

R Significance levels for the outlier test use the `pt` command. For example, if the largest in absolute value of $n = 65$ Studentized residuals is 2.85, with $p' = 5$, the df for error is $n - p' - 1 = 65 - 5 - 1 = 59$. The significance level, using the Bonferroni bound, is

```
> 65 * 2 * (1 - pt(2.85, 59))
```

```
[1] 0.3908
```

By subracting the result of the `pt` command from one, we get the upper tail, rather than the lower tail. We multiply by two to get a two-tailed test. We multiply by $n = 65$ for the Bonferroni inequality.

If you have a regression model `m`, the test for one outlier is.

```
> n * 2 * (1 - pt(max(abs(rstudent(m)), m$df - 1)))
```

The `outlierTest` automates the process of identifying outliers. Using the example from the help page for `outlierTest`, which uses a data file in the `car` package,

```
> outlierTest(lm(prestige ~ income + education, data=Duncan))
```

```
No Studentized residuals with Bonferonni p < 0.05
Largest |rstudent|:
         rstudent unadjusted p-value Bonferonni p
minister    3.135           0.003177       0.1430
```

9.1.4 Additional comments

9.2 INFLUENCE OF CASES

R Table 8.1 lists the fundamental commands for influence and outlier statistics. Each of the commands returns a vector.

In addition, R will compute and return the building blocks that are used in computing these statistics. In particular, the command `influence` in R returns a structure of these building blocks. For example, if you have a model `m1`, then type

```
> ans <- influence(m1)
```

`ans$coefficients` returns a matrix with n rows and p' columns whose ith row is $\hat{\beta}_{(i)}$, ALR[E9.5]. This latter quantity was used to draw ALR[F9.1]. `ans$sigma` returns a vector of the $\hat{\sigma}_{(i)}$, the square roots of the estimate of σ, each computed with one case deleted. Finally, `ans$hat` returns the leverage.

In linear models, all the leave-one-out statistics can be computed from the leverages and residuals, so neither of the first two quantities are computed in an influence analysis.

R make refitting a regression with one (or a few) cases deleted easy. Suppose your model `m1` was fit with $n = 96$ cases, and you want to refit, leaving out cases 2, 7, 16. You need only type

```
> m2 <- update(m1, subset=(1:96)[-c(2, 7, 16)])
```

which uses the `subset` argument to `update` to refit with the specified cases deleted. The value of the `subset` argument can be either a vector of case numbers, as is the case here, or a vector of logical values with `TRUE` for values to be included, and `FALSE` for values to be excluded.

9.2.1 Cook's distance

R The command `cooks.distance` in R computes Cook's distance. It can also be computed directly from ALR[E9.8] for a model `m1`,

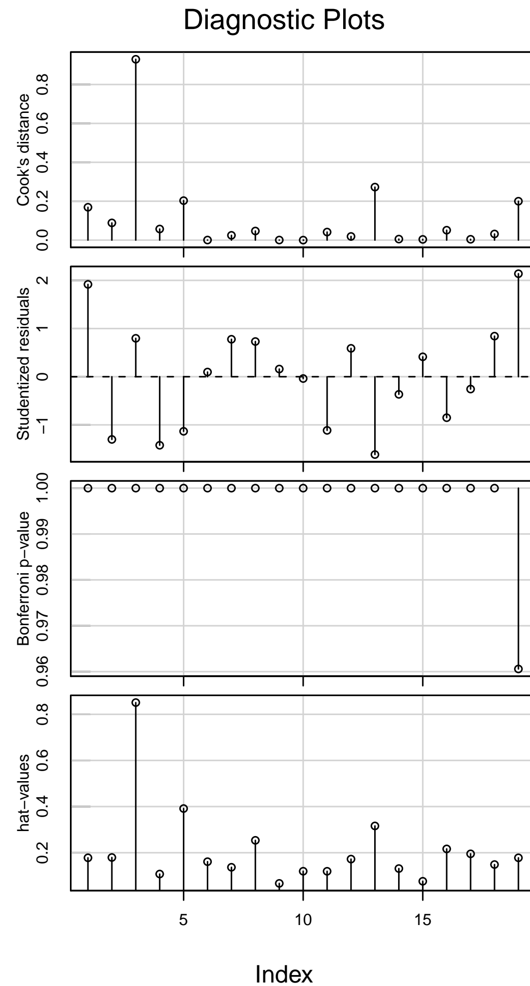
```
> Di <- (ri^2 /m1$rank) * (hatvalues(m1))/(1 - hatvalues(m1))
```

`m1$rank` counts up the number of coefficients estimated in the mean function, so it is equal to p'.

9.2.2 Magnitude of D_i

R The command `infIndexPlot` in the `car` package is used to draw index plots of diagnostic statistics, like ALR[F9.3]. Here are the commands to draw this figure. The third panel down, which in not in ALR[F9.3] gives the Bonferroni significance levels for testing each observation in turn to be an outlier. Since these are upper bounds for most cases the upper bound is equal to one.

```
> m1 <- lm(y ~ BodyWt + LiverWt + Dose, rat)
> infIndexPlot(m1)
```



This function also gives an index plot of the p-values for the (two-tailed) outlier test based on the Bonferroni inequality.

9.2.3 Computing D_i

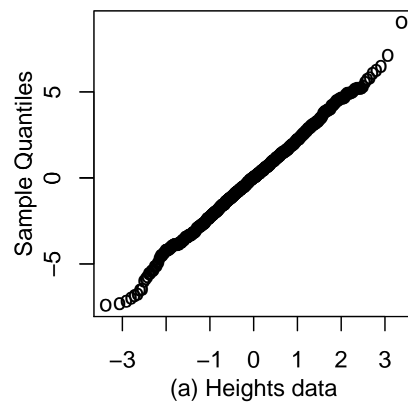
9.2.4 Other measures of influence

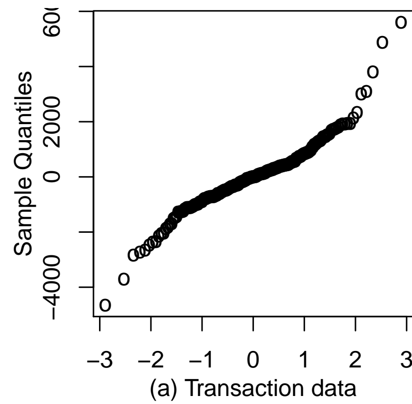
R Just a reminder: added-variable plots are available in the `car` package; see Section 3.1.

9.3 NORMALITY ASSUMPTION

R The `qqnorm` command is used to draw normal probability plots. Here are the commands to draw ALR[F9.5], which shows two normal probability plots.

```
> m1 <- lm(Dheight ~ Mheight, heights)
> t1 <- lm(Time~T1+T2,transact)
> par(mfrow=c(1,2), mar=c(4,3, 0, .5)+.1, mgp=c(2, 1, 0),pch="o")
> qqnorm(residuals(m1), xlab="(a) Heights data", main="")
> qqnorm(residuals(t1), xlab="(a) Transaction data", main="")
```





10
Variable Selection

10.1 THE ACTIVE TERMS

The first example in this chapter uses randomly generated data. This can be helpful in trying to understand issues against a background where we know the right answers. Generating random data is possible in most statistical packages, though doing so may not be easy or intuitive.

R Here is code in R for generating the random data with standard normal random variables, and then fitting the regression for the first case in ALR[10.1]

```
> set.seed(123456)
> case1 <- data.frame(x1=rnorm(100), x2=rnorm(100),
+                       x3=rnorm(100), x4=rnorm(100))
> e <- rnorm(100)
> case1$y <- 1 + case1$x1 + case1$x2 + e
> m1 <- lm(y ~ ., data=case1)
```

The `set.seed` function is used to initialize the random number generator to get the same answers every time I repeat this simulation in writing this primer. Setting the seed is a good idea in general, but you should use a different seed for every simulation. The `data.frame` command created a data frame, with each variable consisting of 100 standard normal pseudo-random numbers. The variable `case1$y` was added to the data frame to be the response, and then the model was fit. The formula `y ~.` uses y as the response and all other columns in the data frame as predictors.

Case 2 is more complicated because the random predictors are correlated. In R, you can use the function `rmvnorm` in the `mvtnorm` package.

```
> Var2 <- matrix(c(1,   0, .95,   0,
+                   0,   1,   0,-.95,
+                 .95,   0,   1,   0,
+                   0,-.95,   0,   1), ncol=4)
> library(mvtnorm)
> X <- rmvnorm(100, sigma=Var2)
> dimnames(X)[[2]] <- paste("x", 1:4, sep="")
> case2 <- data.frame(X)
> case2$y <- 1 + case2$x1 + case2$x2 + e
> m2 <- lm(y ~ x1 + x2 + x3 + x4, data=case2)
```

10.1.1 Collinearity

R The variance inflation factors, defined following ALR[E10.5], are computed by the command `vif` in the `car` package. Here are the variance inflation factors for the first two examples in ALR[10.1]

```
> vif(m1)

   x1    x2    x3    x4
1.052 1.063 1.034 1.041

> vif(m2)

   x1    x2    x3    x4
12.46 12.82 12.51 12.88
```

As expected, in the first case the variance inflation factors are all close to one, but in the second example they are all large.

10.1.2 Collinearity and variances

10.2 VARIABLE SELECTION

10.2.1 Information criteria

The information criteria ALR[E10.7]–ALR[E10.9] depend only on the *RSS*, p', and possibly an estimate of σ^2, and so if these are needed for a particular model, they can be computed from the usual summaries available in a fitted regression model.

R The `extractAIC` can be used to compute *AIC*, *BIC* and C_p. For a fitted subset model `m0` and a fitted larger model `m1`, the following commands extract these quantities:

Table 10.1 R commands relating to subset selection.

	Fit all possible subsets and find the best few
`leaps`	Uses the Furnival and Wilson algorithm to examine all possible regressions. The package `leaps` is loaded with the `alr3` package. If you choose to try this, we recommand using the `subsets` command in `car`, which will provide graphical output in something called a C_p plot. This command does not work well with factors and interactions and is of limited value.
	Functions based on stepwise selection
`step`	Use this function for both forward selection, backward elimination, or a combination of them, based on any of the information criteria except for PRESS.
`drop1`	This command will fit all models obtained by a current fitted model by dropping one term, and gives very easy to interpret output. With this command, you can do backward elimination "by hand," one step at a time. This is particularly helpful in problems with factors and interactions, since dropping a main effect from a mean function that includes interactions with that variable is not recommended.
`add1`	Adds one term from a potential pool of terms to the fitted model. This can be used for forward selection "by hand."

```
> extractAIC(m0, k=2) # give AIC
> extractAIC(m0, k=log(length(residuals(m0)))) # gives BIC
> extractAIC(m0, scale=sigmaHat(m1)^2) # gives Cp
```

10.2.2 Computationally intensive criteria

Computation of *PRESS*, ALR[E10.10], is not common in regression programs, but it is easy to obtain given the residuals and leverages from a fitted model.

R In R, given a model `m1`, *PRESS* can be computed as

```
> PRESS <- sum((residuals(m1, type="pearson") /
+               (1-ls.diag(m1)$hat))^2)
```

This simple computation works for linear models, but not for more complex models such as nonlinear or generalized linear models.

10.2.3 Using subject-matter knowledge

10.3 COMPUTATIONAL METHODS

R Four basic commands are included for use when examining many possible regression models, as listed in Table 10.1.

The primary command we recommend using is `step`. Using the highway data in ALR[T10.5], we start by fitting the largest and smallest models that we want to consider:

```
> highway <- transform(highway, logLen=log2(Len),
+            logADT=log2(ADT), logTrks=log2(Trks),
+            logSigs1=log2((Len*Sigs+1)/Len))
> m1 <- lm(log2(Rate) ~ logLen + logADT + logTrks + logSigs1 +
+          Slim + Shld + Lane + Acpt + Itg + Lwid + Hwy,
+          data=highway)
> m0 <- lm(log2(Rate) ~ logLen, data=highway)
```

As discussed in ALR[10.2.1], log(*Len*) will be in all subsets. One of the terms, *Hwy*, is a factor with 3 df. The first argument to `step` is the name of a model, and for forward selection, it is the smallest model considered. The argument `scope` gives explicity the the `lower` and `upper` models, via one-sided formulas. The `direction` is specified in quotation marks, and then the data frame must be named. The output of this command is very long:

```
> ansf1 <- step(m0,scope=list(lower=~ logLen,
+           upper=~logLen + logADT + logTrks + logSigs1 +
+                  Slim + Shld + Lane + Acpt + Itg +
+                  Lwid + Hwy),
+                  direction="forward", data=highway)

Start:  AIC=-43.92
log2(Rate) ~ logLen

           Df Sum of Sq   RSS   AIC
+ Slim      1      5.30  6.11 -66.3
+ Acpt      1      4.37  7.04 -60.8
+ Shld      1      3.55  7.86 -56.5
+ logSigs1  1      2.00  9.41 -49.4
+ logTrks   1      1.52  9.90 -47.5
+ Hwy       1      1.04 10.37 -45.7
+ logADT    1      0.89 10.52 -45.1
<none>                  11.41 -43.9
+ Lane      1      0.55 10.87 -43.8
+ Itg       1      0.45 10.96 -43.5
+ Lwid      1      0.39 11.03 -43.3

Step:  AIC=-66.28
log2(Rate) ~ logLen + Slim

           Df Sum of Sq  RSS   AIC
+ Acpt      1     0.600 5.51 -68.3
+ logTrks   1     0.548 5.56 -67.9
<none>                  6.11 -66.3
+ logSigs1  1     0.305 5.81 -66.3
+ Shld      1     0.068 6.04 -64.7
```

```
+ logADT    1     0.053 6.06 -64.6
+ Lwid      1     0.035 6.08 -64.5
+ Hwy       1     0.028 6.08 -64.5
+ Lane      1     0.007 6.11 -64.3
+ Itg       1     0.006 6.11 -64.3

Step:  AIC=-68.31
log2(Rate) ~ logLen + Slim + Acpt

           Df Sum of Sq  RSS   AIC
+ logTrks   1     0.360 5.15 -68.9
<none>                  5.51 -68.3
+ logSigs1  1     0.250 5.26 -68.1
+ Hwy       1     0.139 5.37 -67.3
+ Shld      1     0.072 5.44 -66.8
+ logADT    1     0.032 5.48 -66.5
+ Lane      1     0.031 5.48 -66.5
+ Itg       1     0.028 5.48 -66.5
+ Lwid      1     0.026 5.49 -66.5

Step:  AIC=-68.94
log2(Rate) ~ logLen + Slim + Acpt + logTrks

           Df Sum of Sq  RSS   AIC
<none>                  5.15 -68.9
+ Hwy       1    0.1782 4.97 -68.3
+ Shld      1    0.1359 5.02 -68.0
+ logSigs1  1    0.1053 5.05 -67.7
+ logADT    1    0.0650 5.09 -67.4
+ Lwid      1    0.0396 5.11 -67.2
+ Itg       1    0.0228 5.13 -67.1
+ Lane      1    0.0069 5.14 -67.0

> ansf1

Call:
lm(formula = log2(Rate) ~ logLen + Slim + Acpt + logTrks, data = highway)

Coefficients:
(Intercept)       logLen         Slim         Acpt      logTrks
     6.0110      -0.2357      -0.0460       0.0159      -0.3290
```

The default criterion for fitting is *AIC*. Step 1 gives the summary for each mean function obtained by adding one term to the base model given by `m0`. The models are ordered, so the one that minimizes *AIC* appears first.

At Step 2, *Slim* is added to the current model, and all the mean functions add one term to log(*Len*) and *Slim*. Steps continue until the model at the previous step, marked `<none>` has smaller *AIC* than adding the next term; in this example this occurs at Step 4, so the base model for this step is taken as the minimizer of *AIC*. The object `ansf1` is the fitted model that minimizes *AIC*.

Backward elimination is similar, except the specified model is the largest one. The other arguments are similar to the arguments for forward selection. In this example, we have set `scale=sigmaHat(m1)^2`, in which case the selection criterion is C_p, not *AIC*. If you set `k=log(length(residuals$m1))`, then *BIC* will be used. You can't use *PRESS* with this command.

```
> ansf2 <- step(m1, scope=list(lower=~ logLen,
+          upper=~logLen + logADT + logTrks + logSigs1 + Slim +
+                 Shld + Lane + Acpt + Itg + Lwid + Hwy),
+                 direction="backward", data=highway,
+                 scale=sigmaHat(m1)^2)

Start:  AIC=12
log2(Rate) ~ logLen + logADT + logTrks + logSigs1 + Slim + Shld +
    Lane + Acpt + Itg + Lwid + Hwy

           Df Sum of Sq  RSS   Cp
- Lane      1     0.000 4.05 10.0
- Lwid      1     0.032 4.08 10.2
- Hwy       1     0.110 4.16 10.7
- Shld      1     0.114 4.17 10.8
- Slim      1     0.137 4.19 10.9
- Itg       1     0.204 4.26 11.4
- logTrks   1     0.226 4.28 11.5
- logADT    1     0.253 4.31 11.7
<none>                  4.05 12.0
- logSigs1  1     0.458 4.51 13.1
- Acpt      1     0.477 4.53 13.2

Step:  AIC=10
log2(Rate) ~ logLen + logADT + logTrks + logSigs1 + Slim + Shld +
    Acpt + Itg + Lwid + Hwy

           Df Sum of Sq  RSS    Cp
- Lwid      1     0.034 4.09  8.23
- Shld      1     0.115 4.17  8.77
- Hwy       1     0.120 4.17  8.81
- Slim      1     0.137 4.19  8.92
- Itg       1     0.215 4.27  9.44
- logTrks   1     0.232 4.28  9.55
```

```
- logADT    1     0.280 4.33  9.87
<none>                  4.05 10.00
- logSigs1  1     0.472 4.53 11.15
- Acpt      1     0.477 4.53 11.18

Step:  AIC=8.23
log2(Rate) ~ logLen + logADT + logTrks + logSigs1 + Slim + Shld +
    Acpt + Itg + Hwy

           Df Sum of Sq  RSS    Cp
- Shld      1     0.081 4.17  6.77
- Hwy       1     0.092 4.18  6.84
- Slim      1     0.200 4.29  7.56
- logTrks   1     0.205 4.29  7.59
<none>                  4.09  8.23
- Itg       1     0.334 4.42  8.46
- logADT    1     0.469 4.56  9.35
- Acpt      1     0.470 4.56  9.36
- logSigs1  1     0.585 4.67 10.13

Step:  AIC=6.77
log2(Rate) ~ logLen + logADT + logTrks + logSigs1 + Slim + Acpt +
    Itg + Hwy

           Df Sum of Sq  RSS   Cp
- Hwy       1     0.065 4.23 5.21
- logTrks   1     0.179 4.35 5.96
<none>                  4.17 6.77
- Itg       1     0.417 4.58 7.55
- Acpt      1     0.453 4.62 7.79
- Slim      1     0.579 4.75 8.63
- logSigs1  1     0.583 4.75 8.65
- logADT    1     0.716 4.88 9.54

Step:  AIC=5.21
log2(Rate) ~ logLen + logADT + logTrks + logSigs1 + Slim + Acpt +
    Itg

           Df Sum of Sq  RSS   Cp
- logTrks   1     0.156 4.39 4.25
<none>                  4.23 5.21
- Acpt      1     0.402 4.64 5.88
- Slim      1     0.517 4.75 6.65
- Itg       1     0.602 4.84 7.22
- logSigs1  1     0.644 4.88 7.50
```

```
- logADT    1     0.794 5.03 8.49

Step:  AIC=4.25
log2(Rate) ~ logLen + logADT + logSigs1 + Slim + Acpt + Itg

           Df Sum of Sq  RSS   Cp
<none>                  4.39 4.25
- Slim      1     0.483 4.87 5.47
- Acpt      1     0.515 4.90 5.68
- Itg       1     0.646 5.04 6.55
- logADT    1     0.850 5.24 7.91
- logSigs1  1     0.927 5.32 8.43
```

The commands `drop1` and `add1` do one step of these methods, so you can do the stepwise fitted "by hand."

10.3.1 Subset selection overstates significance

10.4 WINDMILLS

10.4.1 Six mean functions

10.4.2 A computationally intensive approach

The data for the windmill example in ALR[10.4.2] is not included with the `alr3` library, and must be downloaded separately from `www.stat.umn.edu/alr`. If you are connected to the internet, you can load the data file with this command

```
> loc <- "http://www.stat.umn.edu/alr/data/wm5.txt"
> wm5 <- read.table(loc, header=TRUE)
```

The R commands used to compute ALR[F10.1] are given in the script for this chapter. The process requires many statements, but the method is straightforward. The command to compute the simulation in the script for this chapter has a comment marker in front of it, as the simulation can take several hours to complete. The results from the simulation discussed in ALR[10.4.2] are available from the data page at `www.stat.umn.edu/alr` in the file `sim1.out`.

11

Nonlinear Regression

11.1 ESTIMATION FOR NONLINEAR MEAN FUNCTIONS

11.2 INFERENCE ASSUMING LARGE SAMPLES

R *and* **S-Plus** The command `nls` is used to obtain OLS and WLS estimates for nonlinear regression models. `nls` differs from `lm` in a few key respects:

1. The formula for defining a model is different. In `lm`, a typical formula is `Y~ X1 + X2 + X3 + X4`. This formula specifies the names of the response and the *terms*, but not the names of the *parameters* because to each term there is an associated parameter, or parameters for a factor. For a `nls` model, the formula specifies *both* terms and parameters. A typical example might be `Y~th1 + th2 * (1 - exp(-th3 * x))`. In this case there are three parameters, `th1`, `th2` and `th3`, but only one term, `x`. The right-hand side formula should be an expression of parameters, terms, and numbers that can be evaluated by the computer language C.

2. Factors are generally not used with `nls`.

3. A named list `start` of *starting values* must be specified. This serves the dual purpose of telling the algorithm where to start, and also to name the parameters. For the example, `start=list(th1=620, th2=200, th3=10)` indicates that `th1`, `th2` and `th3` are parameters, and so by implication the remaining quantity, `x`, must be a predictor variable.

4. Most iterative algorithms for nonlinear least squares require computation of derivatives, and many programs require the user to provide formulas for the derivatives. This is not the case in `nls`; all derivatives are computed using numerical differentiation.

Here is the input leading to ALR[T11.2]

```
> n1 <- nls(Gain ~ th1 + th2*(1-exp(-th3 * A)), data=turk0,
+         start=list(th1=620, th2=200, th3=10))
> summary(n1)

Formula: Gain ~ th1 + th2 * (1 - exp(-th3 * A))

Parameters:
    Estimate Std. Error t value Pr(>|t|)
th1   622.96       5.90  105.57  < 2e-16
th2   178.25      11.64   15.32  2.7e-16
th3     7.12       1.21    5.91  1.4e-06

Residual standard error: 19.7 on 32 degrees of freedom

Number of iterations to convergence: 4
Achieved convergence tolerance: 1.48e-06
```

After loading the data, we called `nls` with the required arguments of a formula and starting values. In addition, we specified a data frame. R has many other arguments to `nls`, including several that are identical with arguments of the same name in `lm`, including `subset` for selecting cases, and `na.action` for setting the missing value action. A few arguments are used to change the details of the computing algorithm; see the help page for `nls`.

The printed output from the `summary` method is also similar to the output from `ls`, as described in ALR[11.2]. The "correlation of parameter estimates" is the matrix ALR[E11.14] rescaled as a correlation matrix.

Similar to an object created by the `lm` command, objects created by `nls` have `summary`, `plot` and `predict` methods, and these are used in a way that is similar to linear regression models. For example, ALR[F11.2] is obtained by

```
> with(turk0, plot(A, Gain, xlab="Amount (percent of diet)",
+        ylab="Weight gain, g"))
> x <- (0:44)/100
> lines(x,predict(n1,data.frame(A=x)))
```

Lack-of-fit testing is possible if there are repeated observations. The idea is to compare the nonlinear fit to the one-way analysis of variance, using the levels of the predictor as a grouping variable:

```
> m1 <- lm(Gain ~ as.factor(A), turk0)
> anova(n1,m1)
```

```
Analysis of Variance Table

Model 1: Gain ~ th1 + th2 * (1 - exp(-th3 * A))
Model 2: Gain ~ as.factor(A)
  Res.Df Res.Sum Sq Df Sum Sq F value Pr(>F)
1     32      12367
2     29       9824  3   2544     2.5  0.079
```

The model `m1` is a linear model, while `n1` is a nonlinear model. Even so, the `anova` will correctly compare them, giving the F test for lack of fit.

Unlike `lm`, `nls` does not allow factors in a formula. To use factors, you need to create dummy variables for the various levels of the factor and use these.

For example, consider fitting the models ALR[11.17]–ALR[11.19]. The data frame `turkey` includes a variable `S` that is a factor with three levels for the source of the dietary addtive. Weighted least squares is required. We can apply ALR[E5.8] to get `nls` to get WLS estimates. We have $y = g(\theta, x) + e/\sqrt{w}$ where the e's have constant variance, so the y's have variance σ^2/w. Multiply both sides of the mean function by $\sqrt{w}$ to get $\sqrt{w}y = \sqrt{w}g(\theta, x) + e$ so we can get WLS estimates in the original problem by getting OLS estimates with $\sqrt{w}g(\theta, x)$ as the kernel mean function, and $\sqrt{w}y$ as the response.

The code for fitting the four models using WLS is given next. The vector `m` is the number of pens at each treatment combination, and is the vector of weights for this problem, `S` is the factor, and `Gain` is the response variable.

```
> # create the indicators for the categories of S
> turkey$S1 <- turkey$S2 <- turkey$S3 <- rep(0,dim(turkey)[1])
> turkey$S1[turkey$S==1] <- 1
> turkey$S2[turkey$S==2] <- 1
> turkey$S3[turkey$S==3] <- 1
> # compute the weighted response
> turkey$wGain <- sqrt(turkey$m) * turkey$Gain
> # fit the models
> # common regressions
> m4 <- nls(wGain ~ sqrt(m) * (th1 + th2*(1-exp(-th3 * A))),
+             data=turkey, start=list(th1=620, th2=200, th3=10))
> # most general
> m1 <- nls( wGain ~ sqrt(m) * (S1 * (th11 + th21 *
+                     (1 - exp(-th31 * A))) +
+                     S2 * (th12 + th22 * (1 - exp(-th32 * A)))+
+                     S3 * (th13 + th23*(1 - exp(-th33 * A)))),
+             data=turkey, start= list(th11=620, th12=620, th13=620,
+                                      th21=200, th22=200, th23=200,
+                                      th31=10, th32=10, th33=10))
> # common intercept
> m2 <- nls(wGain ~ sqrt(m) * (th1 +
+                    S1 * (th21 * (1 - exp(-th31 * A)))+
```

```
+                   S2 * (th22 * (1 - exp(-th32 * A)))+
+                   S3 * (th23 * (1 - exp(-th33 * A)))),
+           data=turkey, start= list(th1=620,
+                               th21=200, th22=200, th23=200,
+                               th31=10, th32=10, th33=10))
> # common intercept and asymptote
> m3 <- nls( wGain ~ sqrt(m) * (th1 + th2  * (
+                    S1 * (1 - exp(-th31 * A))+
+                    S2 * (1 - exp(-th32 * A))+
+                    S3 * (1 - exp(-th33 * A)))),
+           data=turkey,start= list(th1=620, th2=200,
+                                  th31=10, th32=10, th33=10))
```

We have now created the dummy variables, weighted response, and the models with different mean functions. If there were no weights, we could leave off the `sqrt(w)` in the above statements, and use `Gain`, not `wGain`, as the response. In each of the models we had to specify starting values for all the parameters. The starting values we use essentially assume no group differences. Here are the `anova` tables to compare the models:

```
> anova(m4,m2,m1)
Analysis of Variance Table

Model 1: wGain ~ sqrt(m) * (th1 + th2 * (1 - exp(-th3 * A)))
Model 2: wGain ~ sqrt(m) * (th1 + S1 * (th21 * (1 - exp(-th31 * A)))
         + S2 * (th22 * (1 - exp(-th32 * A)))
         + S3 * (th23 * (1 - exp(-th33 * A))))
Model 3: wGain ~ sqrt(m) * (S1 * (th11 + th21 * (1 - exp(-th31 * A)))
         + S2 * (th12 + th22 * (1 - exp(-th32 * A)))
         + S3 * (th13 + th23 * (1 - exp(-th33 * A))))
  Res.Df Res.Sum Sq Df Sum Sq F value Pr(>F)
1     10       4326
2      6       2040  4   2286    1.68   0.27
3      4       1151  2    889    1.54   0.32
> anova(m4,m3,m1)
Analysis of Variance Table

Model 1: wGain ~ sqrt(m) * (th1 + th2 * (1 - exp(-th3 * A)))
Model 2: wGain ~ sqrt(m) * (th1 + th2 * (S1 * (1 - exp(-th31 * A))
         + S2 * (1 - exp(-th32 * A)) + S3 * (1 - exp(-th33 * A))))
Model 3: wGain ~ sqrt(m) * (S1 * (th11 + th21 * (1 - exp(-th31 * A)))
         + S2 * (th12 + th22 * (1 - exp(-th32 * A)))
         + S3 * (th13 + th23 * (1 - exp(-th33 * A))))
  Res.Df Res.Sum Sq Df Sum Sq F value Pr(>F)
1     10       4326
2      8       2568  2   1758    2.74   0.12
3      4       1151  4   1417    1.23   0.42
```

These tests are interpreted as in ALR[6.2.2].

11.3 BOOTSTRAP INFERENCE

R The bootstrap can be done using the same `bootCase` command used for linear models. For example, to get the bootstrap leading to ALR[F11.5], using R

```
> smod <- C ~ th0 + th1 * (pmax(0,Temp-gamma))
> s1 <- nls(smod, data=segreg, data = segreg,
+           start=list(th0=70, th1=.5, gamma=40))
> set.seed(10131985)
> s1.boot <- bootCase(s1, B=999)
```

`s1.boot` will consist only of a matrix with 999 rows and 3 columns, one row for each replication, and one column for each parameter, and summarizing the bootstrap is your responsibility. The summary used in ALR uses the `scatterplot.matrix` command in the `car` package.

```
> scatterplot.matrix(s1.boot, diagonal="histogram",
+    lwd=0.7, pch=".",
+    labels=c(expression(theta[1]), expression(theta[2]),
+              expression(gamma)),
+    ellipse=FALSE, smooth=TRUE, level=c(.90))
```

Numerical summaries can also be appropriate, for example,

```
> s1.boot.summary <-
+   data.frame(rbind(
+    apply(s1.boot, 2, mean),
+    apply(s1.boot, 2, sd),
+    apply(s1.boot, 2, function(x){quantile(x, c(.025, .975))})))
> row.names(s1.boot.summary)<-c("Mean", "SD","2.5%", "97.5%")
> s1.boot.summary

          th0     th1   gamma
Mean  74.8407 0.60701 43.1235
SD     1.4453 0.11833  4.6156
2.5%  72.0214 0.46359 36.8698
97.5% 77.5957 0.97129 55.0744
```

Here we combined the summaries into a data frame so they would be pretty when printed. The `apply` command is used repeatedly to apply functions to the columns of `s1.boot`, and we summarize with the mean, SD, and 95% percentile-based confidence interval for each parameter.

11.4 REFERENCES

12

Logistic Regression

Both logistic regression and the normal linear models that we have discussed in earlier chapters are examples of *generalized linear models*. Many programs, including SAS and R have procedures that can be applied to any generalized linear model. Both JMP and SPSS seem to have separate procedures for logistic regression. There is a possible source of confusion in the name. Both SPSS and SAS use the name *general linear model* to indicate a relatively complex linear model, possibly with continuous terms, covariates, interactions, and possibly even random effects, but with normal errors. Thus the general linear model is a special case of the generalized linear models.

12.1 BINOMIAL REGRESSION

12.1.1 Mean Functions for Binomial Regression

12.2 FITTING LOGISTIC REGRESSION

R The key command in fitting a generalized linear model is `glm`. We will only consider the logistic regression case, but only minor changes are required for other cases.

Here is a general form for the `glm` command, illustrating the basic arguments available for binomial regression:

```
> glm(formula, family = binomial(link="logit"), data, weights,
+     subset, na.action, start = NULL)
```

The arguments `data`, `weights`, `subset` and `na.action` are used in the same way they are used in `lm`. The `family` argument is new, and the `formula` argument is a little different than its use in `lm` or `nls`. There are a few other arguments, mostly modifying the computational algorithm, that will probably be rarely needed.

As in ALR[12.1.1], let $\theta(\mathbf{x})$ be the probability of success given the value of the terms X in the mean function equal $\mathbf{x}$. According to the logistic regression model, we have from ALR[E12.7]

$$\log\left(\frac{\theta(\mathbf{x})}{1-\theta(\mathbf{x})}\right) = \boldsymbol{\beta}'\mathbf{x} \tag{12.1}$$

The quantity on the right of this equation is called the *linear predictor*. The quantity on the left depends on the link function (or equivalently, the kernel mean function) used in fitting, which in this case is the *logit link*, the inverse of the logistic kernel mean function discussed in ALR[10.1.1].

The `formula` for a `glm` is a two-sided formula. The right-hand side of this formula is the linear predictor, equivalent to $\boldsymbol{\beta}'\mathbf{x}$ in (12.1). The left-hand side is either: (1) if the number of trials is always equal to one for each case, then the left-hand side is just the the response variable, a vector of zeroes and ones. (2) If the number of trials is not always equal to one, then the left-hand side is a matrix with two columns, the first column giving the number of successes and the second giving the number of failures.

The `family` argument is the name of an *error distribution*, which is `binomial` for binomial regression. Other choices like `Poisson` and `gaussian` are used for Poisson errors and normal errors, respectively. As an argument to the family, you set the `link` function. The only link discussed in ALR is the logit link for logistic regression. Since this is the default, it need not be specified. See VR[7] for a more general discussion.

12.2.1 One-predictor example

R For ALR[T12.1], here are the computations:

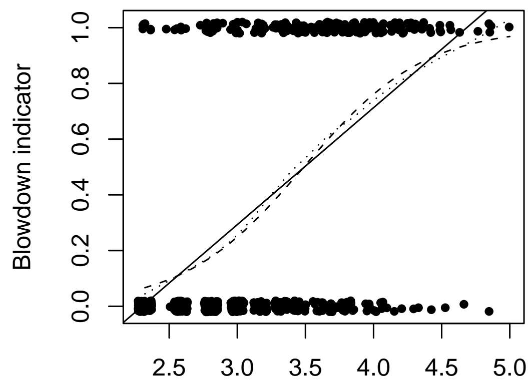
```
> m1 <- glm(y ~ logb(D, 2), family=binomial, data=blowBF)
> summary(m1)

Call:
glm(formula = y ~ logb(D, 2), family = binomial, data = blowBF)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-2.507  -0.757  -0.494   0.810   2.327

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
```

```
(Intercept)   -7.892      0.633   -12.5   <2e-16
logb(D, 2)     2.263      0.191    11.8   <2e-16

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 856.21  on 658  degrees of freedom
Residual deviance: 655.24  on 657  degrees of freedom
AIC: 659.2

Number of Fisher Scoring iterations: 4
```

The linear predictor used consists of the base-two logarithm of D. The `family` must be specified, because the default is `family=gaussian`, but the link can be skipped because the default is `"logit"`.

Here are the commands to draw ALR[F12.1A].

```
> with(blowBF, {
+   plot(jitter(log2(D), amount=.05),
+       jitter(y, amount=0.02), pch=20,
+       xlab=expression(paste("(a) ",Log[2](Diameter))),
+       ylab="Blowdown indicator")
+   xx <- seq(min(D), max(D), length=100)
+   abline(lm(y~ log2(D)), lty=1)
+   lo.fit <- loess(y ~ log2(D), degree=1)
+   lines(log2(xx), predict(lo.fit, data.frame(D=xx)), lty=3)
+   lines(log2(xx), predict(m1, data.frame(D=xx), type="response"), lty=2)
+ })
```

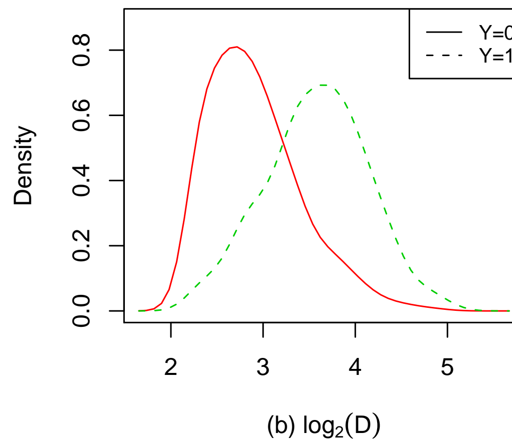


(a) Log_2(Diameter)

The data are jittered in `plot` to avoid over-plotting. The `with` function is used so variable names need not specify the name of the data frame. We have used it here with many commands by surrounding them with { and }. The vector `xx` will be used in plotting fitted values. First, the OLS line is added to the plot. Then the `loess` function was used to get a smoother, and it is added to the plot. Then, the fitted values from the model `m1` are added to the plot.

ALR[F12.1B] uses the `sm.density` command in the `sm` package.

```
> library(sm)
> with(blowBF, {
+   sm.density.compare(log2(D), y, lty=c(1,2),
+     xlab=expression(paste("(b) ", log[2](D))))
+   legend("topright",legend=c("Y=0", "Y=1"), lty=c(1,2),cex=0.8)
+ })
```



12.2.2 Many Terms

R Here are the statements to get the models shown in ALR[T12.2]:

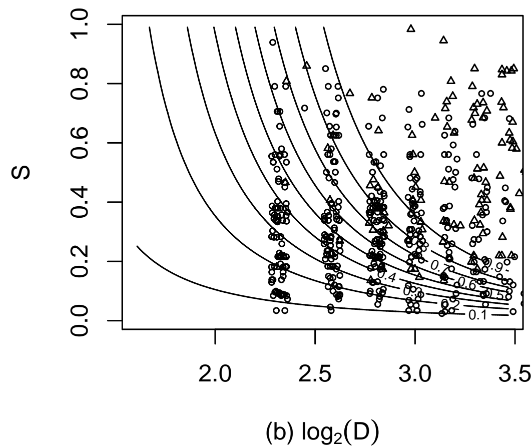
```
> m2 <- glm(y ~ log2(D) + S, family=binomial, data=blowBF)
> m3 <- update(m2,~ . + log(D):S)
> summary(m2)

Call:
glm(formula = y ~ log2(D) + S, family = binomial, data = blowBF)

Deviance Residuals:
```

```
   Min      1Q  Median      3Q     Max
-2.663  -0.659  -0.351   0.547   2.693

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   -9.562      0.750  -12.75   <2e-16
log2(D)        2.216      0.208   10.66   <2e-16
S              4.509      0.516    8.74   <2e-16

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 856.2  on 658  degrees of freedom
Residual deviance: 563.9  on 656  degrees of freedom
AIC: 569.9

Number of Fisher Scoring iterations: 5

> summary(m3)

Call:
glm(formula = y ~ log2(D) + S + S:log(D), family = binomial,
    data = blowBF)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-3.344  -0.614  -0.412   0.382   2.356

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   -3.678      1.425   -2.58   0.0099
log2(D)        0.401      0.439    0.91   0.3614
S            -11.205      3.638   -3.08   0.0021
S:log(D)       7.085      1.645    4.31  1.7e-05

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 856.21  on 658  degrees of freedom
Residual deviance: 541.75  on 655  degrees of freedom
AIC: 549.7

Number of Fisher Scoring iterations: 5
```

ALR[12.2.2] also includes several non-standard graphics. ALR[F12.2A] uses `sm.density`, as illustrated previously, and ALR[F12.2B] is just an ordinary scatterplot, but uses jittering. ALR[F12.3] shows contours on constant esti-

mated probability superimposed on the scatterplot of the data. Here are the commands for ALR[F12.3B].

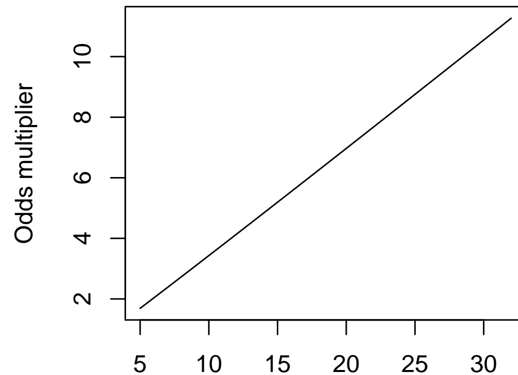
```
> with(blowBF, {
+   xa <- seq(min(D), max(D), len=99)
+   ya <- seq(.01, .99, len=99)
+   za <- matrix(nrow=99, ncol=99)
+   for (i in 1:99) {
+     za[,i] <- predict(m3, data.frame(D=rep(xa[i], 99), S=ya),
+              type="response")}
+   contour(log(xa), ya, za,
+       xlab=expression(paste("(b) ", log[2](D))), ylab="S")
+   points(jitter(log2(D), amount=.04), S,pch=y + 1, cex=.5)
+   })
```



We first defined a grid `xa` for the horizontal axis for D, `ya` for the vertical axis for S. `za` is then defined as a (very large) matrix giving the predicted value for each combination (`xa`, `ya`). The `contour` command is then used to draw the probability contours based on this matrix. We needed to adjust the horizontal axis to be the base-two logarithm of `xa`. The jittered points were then added.

The plots in ALR[F12.5] are considerably simpler. Here is the second one.

```
> xx <- with(blowBF, seq(min(log2(D)), max(log2(D)), length=100))
> plot(2^(xx), exp(coef(m3)[3]/10 + coef(m3)[4] * xx/10), type="l",
+         xlab="(b) D", ylab="Odds multiplier")
```



(b) D

12.2.3 Deviance

R Comparing models uses the `anova` just as with `lm` models. For the three models fit to the Balsam Fir blowdown data,

```
> anova(m1, m2, m3, test="Chisq")

Analysis of Deviance Table

Model 1: y ~ logb(D, 2)
Model 2: y ~ log2(D) + S
Model 3: y ~ log2(D) + S + S:log(D)
  Resid. Df Resid. Dev Df Deviance P(>|Chi|)
1       657        655
2       656        564  1     91.3   < 2e-16
3       655        542  1     22.2   2.5e-06
```

We specified `test="Chisq")` to compare the change in deviance to the Chi-squared distribution to get significance levels. The default is no significance levels.

12.2.4 Goodness of Fit Tests

R The Titanic data uses binomial, rather than binary regression, since the number of trails exceeds one. Here is the syntax:

```
> m1 <- glm(cbind(Surv, N-Surv) ~ Class + Age + Sex, data=titanic,
+               family=binomial)
```

The response is specified as a matrix whose first column is the number of survivors, and whose second column is the number of deaths.

12.3 BINOMIAL RANDOM VARIABLES

12.3.1 Maximum likelihood estimation

12.3.2 The Log-likelihood for Logistic Regression

12.4 GENERALIZED LINEAR MODELS

Problems

R 12.5.1. The `pairs` will accept a matrix (or data frame) for its first argument, and the argument `col` can be used to set the colors of the points. You can also use `scatterplotMatrix` in the `car` package.

Appendix A

A.1 WEB SITE

A.2 MEANS AND VARIANCES OF RANDOM VARIABLES

A.2.1 E notation

A.2.2 Var notation

A.2.3 Cov notation

A.2.4 Conditional moments

A.3 LEAST SQUARES FOR SIMPLE REGRESSION

A.4 MEANS AND VARIANCES OF LEAST SQUARES ESTIMATES

A.5 ESTIMATING $\mathrm{E}(Y|X)$ USING A SMOOTHER

R A bewildering array of options are available for smoothing, both in the base programs, and in packages available from others. In ALR, we have almost

always used the `loess` smoother or an older version of it called `lowess`[1]. There is no particular reason to prefer this smoother over other smoothers. For the purposes to which we put smoothers, mainly to help us look at scatterplots, the choice of smoother is probably not very important.

The important problem of choosing a smoothing parameter is generally ignored in ALR; one of the nice features of `loess` is that choosing the smoothing parameter of about 2/3 usually gives good answers. R have several methods for selecting a smoothing parameter; see VR[8.7], and the references given in ALR[A.5], for more information.

The `lowess` is used to draw ALR[FA.1]

```
> with(ufcwc, {
+   plot(Height ~ Dbh)
+   abline(lm(Height ~ Dbh),lty=3)
+   lines(lowess(Dbh, Height, iter=1, f=.1), lty=4)
+   lines(lowess(Dbh, Height, iter=1, f=2/3), lty=1)
+   lines(lowess(Dbh, Height, iter=1, f=.95), lty=2)
+   legend(700,200, legend=c("f=.1","f=2/3", "f=.95", "OLS"),
+     lty=c(4, 1, 2, 3))
+   })
```

We have used the `lines` helper to add the `lowess` fits to the graph, with different values of the smoothing argument, `f`. We set the argument `iter` to one rather than the default value of three.

More or less the same figure can be obtained using `loess`.

```
> with(ufcwc, {
+   plot(Dbh, Height)
+   abline(lm(Height ~ Dbh), lty=3)
+   new <- seq(100, 1000, length=100)
+   m1 <- loess(Height ~ Dbh, degree=1, span=.1)
+   m2 <- loess(Height ~ Dbh, degree=1, span=2/3)
+   m3 <- loess(Height ~ Dbh, degree=1, span=.95)
+   lines(new, predict(m1, data.frame(Dbh=new)), lty=4, col="cyan")
+   lines(new, predict(m2, data.frame(Dbh=new)), lty=1, col="red")
+   lines(new, predict(m3, data.frame(Dbh=new)), lty=2, col="blue")
+   legend(700, 200, legend=c("f=.1", "f=2/3", "f=.95", "OLS"),
+     lty=c(4, 1, 2, 3))
+   })
```

`loess` expects a formula. `degree=1` uses local linear, rather than quadratic, smoothing, to match `lowess`. The `span` argument is like the `f` argument for `lowess`.

[1] `loess` and `lowess` have different defaults, and so they will give the same answers only if they are set to use the same tuning parameters.

ALR[FA.2] with the standard deviation smooths was drawn using the following commands.

```
> with(ufcwc, {
+   plot(Dbh, Height)
+   loess.fit <- loess(Height ~ Dbh, degree=1, span=2/3) # gives same answers
+   sqres <- residuals(loess.fit)^2
+   loess.var.fit <- loess(sqres ~ Dbh, degree=1, span=2/3)
+   new <- seq(min(Dbh), max(Dbh), length=100)
+   lines(new, predict(loess.fit, data.frame(Dbh=new)))
+   lines(new, predict(loess.fit, data.frame(Dbh=new)) +
+               sqrt(predict(loess.var.fit, data.frame(Dbh=new))), lty=2)
+   lines(new, predict(loess.fit,data.frame(Dbh=new)) -
+               sqrt(predict(loess.var.fit, data.frame(Dbh=new))), lty=2)
+   })
```

We used the `loess` command rather than `lowess` because `loess` has a method for the `residuals` helper function.

A.6 A BRIEF INTRODUCTION TO MATRICES AND VECTORS

R VR[3.9] gives a good summary of the most commonly used matrix commands. These are also discussed in the documentation that comes with the two programs, or in various help pages available in the program.

A.6.1 Addition and subtraction

A.6.2 Multiplication by a scalar

A.6.3 Matrix multiplication

A.6.4 Transpose of a matrix

A.6.5 Inverse of a matrix

A.6.6 Orthogonality

A.6.7 Linear dependence and rank of a matrix

A.7 RANDOM VECTORS

A.8 LEAST SQUARES USING MATRICES

A.8.1 Properties of estimates

A.8.2 The residual sum of squares

A.8.3 Estimate of variance

A.9 THE QR FACTORIZATION

R These programs compute the QR factorization using Lapack routines, or the slightly older but equally good Linpack routines. We illustrate the use of these routines with a very small example.

```
> X <- matrix(c(1, 1, 1, 1, 2, 1, 3, 8, 1, 5, 4, 6), ncol=3,
+      byrow=FALSE)
> y <- c(2, 3, -2, 0)
> X

     [,1] [,2] [,3]
[1,]    1    2    1
[2,]    1    1    5
[3,]    1    3    4
[4,]    1    8    6

> y

[1]  2  3 -2  0

> QR <- qr(X)
```

Here X is a 4×3 matrix, and y is a 4×1 vector. To use the QR factorization, we start with the command `qr`. This does not find Q or R explicitly but it

does return a structure that contains all the information needed to compute these quantities, and there are helper functions to return Q and R:

```
> qr.Q(QR)

     [,1]     [,2]    [,3]
[1,] -0.5 -0.27854  0.7755
[2,] -0.5 -0.46424 -0.6215
[3,] -0.5 -0.09285 -0.0605
[4,] -0.5  0.83563 -0.0935
```

```
> qr.R(QR)

     [,1]   [,2]   [,3]
[1,]   -2 -7.000 -8.000
[2,]    0  5.385  2.043
[3,]    0  0.000 -3.135
```

Q is a 4×3 matrix with orthogonal columns, and R is is a 3×3 upper triangular matrix. These matrices are rarely computed in full like this. Rather, other helper functions are used.

In the regression context, there are three helpers that return quantities related to linear regression, using ALR[EA.25–EA.27].

```
> qr.coef(QR,y)

[1]  1.7325 -0.3509  0.0614
```

```
> qr.fitted(QR,y)

[1]  1.0921  1.6886  0.9254 -0.7061
```

```
> qr.resid(QR,y)

[1]  0.9079  1.3114 -2.9254  0.7061
```

`qr.coef` computes ALR[EA.26], `qr.fitted` computes the fitted values $QQ'y$, and `qr.resid` computes residuals $y - QQ'y$.

The `backsolve` command is used to solve triangular systems of equations, as described in ALR[A.9].

The basic linear regression routine `lm` uses the QR factorization to obtain estimates, standard errors, residuals and fitted values.

A.10 MAXIMUM LIKELIHOOD ESTIMATES

A.11 THE BOX-COX METHOD FOR TRANSFORMATIONS

A.11.1 Univariate case

A.11.2 Multivariate case

A.12 CASE DELETION IN LINEAR REGRESSION

References

1. Chambers, J. and Hastie, T. (eds.) (1993). *Statistical Models in S.* Boca Raton, FL: CRC Press.

2. Cook, R. D. and Weisberg, S. (1982). *Residuals and Influence in Regression.* London: Chapman & Hall.

3. Cook, R. D. and Weisberg, S. (1999). *Applied Regression Including Computing and Graphics.* New York: Wiley.

4. Cook, R. D. and Weisberg, S. (2004). Partial One-Dimensional Regression Models.

5. Dalgaard, Peter (2002). *Introductory Statistics with R.* New York: Springer.

6. Davison, A. and Hinkley, D. (1997). *Bootstrap Methods and their Application.* Cambridge: Cambridge University Press.

7. Efron, B. and Tibshirani, R. (1993). *An Introduction to the Bootstrap.* Boca Raton: Chapman & Hall.

8. Fox, John, and Weisberg, S. (2011). *An R Companion to Applied Regression*, second edition. Thousand Oaks, CA: Sage.

9. Fruend, R., Littell, R. and Creighton, L. (2003). *Regression Using JMP.* Cary, NC: SAS Institute, Inc., and New York: Wiley.

10. Furnival, G. and Wilson, R. (1974). Regression by leaps and bounds. *Technometrics*, 16, 499-511.

11. Knüsel, Leo (2005). On the accuracy of statistical distributions in Microsoft Excel 2003. *Computational Statistics and Data Analysis*, 48, 445–449.

12. Maindonald, J. and Braun, J. (2003). *Data Analysis and Graphics Using R.* Cambridge: Cambridge University Press.

13. Muller, K. and Fetterman, B. (2003). *Regression and ANOVA: An Integrated Approach using SAS Software.* Cary, NC: SAS Institute, Inc., and New York: Wiley.

14. Nelder, J. (1977). A reformulation of linear models. *Journal of the Royal Statistical Society*, A140, 48–77.

15. Pinheiro, J. and Bates, D. (2000). *Mixed-Effects Models in S and S-plus.* New York: Springer.

16. Rubin, D. B. (1987). *Multiple Imputation for Nonresponse in Surveys.* New York: John Wiley & Sons, Inc.

17. Sall, J., Creighton, L. and Lehman, A. (2005). *JMP Start Statistics*, third edition. Cary, NC: SAS Institite, and Pacific Grove, CA: Duxbury. **Referred to as** JMP-START**.**

18. SPSS (2003). *SPSS Base 12.0 User's Guide.* Chicago, IL: SPSS, Inc.

19. Thisted, R. (1988). *Elements of Statistical Computing.* New York: Chapman & Hall.

20. Venables, W. and Ripley, B. (2000). *S Programming.* New York: Springer.

21. Venables, W. and Ripley, B. (2002). *Modern Applied Statistics with S*, 4th edition. New York: Springer. **referred to as** VR**.**

22. Venables, W. and Smith, D. (2002). *An Introduction to R.* Network Theory, Ltd.

23. Verzani, John (2005). *Using R for Introductory Statistics.* Boca Raton: Chapman & Hall.

24. Weisberg, S. (2005). *Applied Linear Regression*, third edition. New York: Wiley. **referred to as** ALR.

25. Weisberg, S. (2005). Lost opportunities: Why we need a variety of statistical languages. *Journal of Statistical Software*, 13(1), www.jstatsoft.org.