2.1 SUMMARY

This chapter covers both the practical details and the broader philosophy of (1) reading data into R and (2) doing exploratory data analysis, in particular graphical analysis. To get the most out of the chapter you should already have some basic knowledge of R's syntax and commands (see the R supplement of the previous chapter).

2.2 INTRODUCTION

One of the basic tensions in all data analysis and modeling is how much you have all your questions framed before you begin to look at your data. In the classical statistical framework, you're supposed to lay out all your hypotheses before you start, run your experiments, come back to your office and test those (and only those) hypotheses. Allowing your data to suggest new statistical tests raises the risk of "fishing expeditions" or "data-dredging" — indiscriminately scanning your data for patterns*. Data-dredging is a serious problem. Humans are notoriously good at detecting apparent patterns even when they don't exist. Strictly speaking, interesting patterns that you find in your data after the fact should not be treated statistically, only used as input for the next round of observations and experiments†. Most statisticians are leery of procedures like stepwise regression that search for the best predictors or combinations of predictors from among a large range of options, even though some have elaborate safeguards to avoid overestimating the significance of observed patterns (Whittingham et al., 2006). The worst part about using such techniques is that in order to use them you must be conservative and discard real patterns, patterns that you originally had in mind, because you are screening your data indiscriminately (Nakagawa, 2004).

But these injunctions may be too strict for ecologists. Unexpected patterns in the data can inspire you to ask new questions, and it is foolish not to explore your hard-earned data. *Exploratory data analysis* (EDA: Tukey, 1977; Hoaglin et al., 2000, 2006; Cleveland, 1993) is a set of graphical techniques for finding interesting patterns in data. EDA was developed in the late 1970s when computer graphics first became widely available. It emphasizes *robust* and *nonparametric* methods, which make fewer assumptions about the shapes of curves and the distributions of the data and hence

*"Bible Codes", where people find hidden messages in the Bible, illustrate an extreme form of data-dredging. Critics have pointed out that similar procedures will also detect hidden messages in *War and Peace* or *Moby Dick* (McKay et al., 1999).

†Or you should apply a *post hoc* procedure [see `?TukeyHSD` and the `multcomp` package in R] that corrects for the fact that you are testing a pattern that was not suggested in advance — however, even these procedures only apply corrections for a specific set of possible comparisons, not all possible patterns that you could have found in your data.

are less sensitive to nonlinearity and outliers. Most of the rest of this book will focus on models that, in contrast to EDA, are parametric (i.e., they specify particular distributions and curve shapes) and mechanistic. These methods are more powerful and give more ecologically meaningful answers, but are also susceptible to being misled by unusual patterns in the data.

The big advantages of EDA are that it gets you looking at and thinking about your data (whereas stepwise approaches are often substitutes for thought), and that it may reveal patterns that standard statistical tests would overlook because of their emphasis on specific models. However, EDA isn't a magic formula for interpreting your data without the risk of data dredging. Only common sense and caution can keep you in the zone between ignoring interesting patterns and over-interpreting them. It's useful to write down a list of the ecological patterns you're looking for and how they relate your ecological questions *before* you start to explore your data, so that you can distinguish among (1) patterns you were initially looking for, (2) unanticipated patterns that answer the same questions in different ways, and (3) interesting (but possibly spurious) patterns that suggest new questions.

The rest of this chapter describes how to get your data into R and how to make some basic graphs in order to search for expected and unexpected patterns. The text covers both philosophy and some nitty-gritty details. The supplement at the end of the chapter gives a sample session and more technical details.

2.3 GETTING DATA INTO R

2.3.1 Preliminaries

Electronic format

Before you can analyze your data you have to get them into R. Data come in a variety of formats — in ecology, most are either plain text files (space- or comma-delimited) or Excel files.* R prefers plain text files with "white space" (arbitrary numbers of tabs or spaces) or commas between columns. Text files are less structured and may take up more disk space than more specialized formats, but they are the lowest common denominator of file formats and so can be read by almost anything (and, if necessary, examined and adjusted in any text editor). Since plain text formats are readable with a wide variety of text editors, they are unlikely to be made obsolete

*Your computer may be set up to open comma-delimited (`.csv`) files in Excel, but underneath they are just text files.

by changes in technology (you could say they're already obsolete), and less likely to be made unusable by corruption of a few bits of the file; only hard copy is better[†].

R is platform-agnostic. While text files do have very slightly different formats on Unix, Microsoft Windows, and Macintosh operating systems, R handles these differences. If you later save data sets or functions in R's own format (using `save` to save and `load` to load them), you will be able to exchange them freely across platforms.

Many ecologists keep their data in Excel spreadsheets. The `read.xls` function in the `gdata` package allows R to read Excel files directly, but the best thing to do with an Excel file (if you have access to a copy of Excel, or if you can open it in an alternative spreadsheet program) is to save the worksheet you want as a `.csv` (comma-separated values) file. Saving as a `.csv` file will also force you to go into the worksheet and clean up any random cells that are outside of the main data table — R won't like these. If your data are in some more exotic form (e.g. within a GIS or database system), you'll have to figure out how to extract them from that particular system into a text file. There are ways of connecting R directly with databases or GIS systems, but they're beyond the scope of this book. If you have trouble exporting data or you expect to have large quantities of data (e.g. more than tens of thousands of observations) in one of these exotic forms, you should look for advice at the R *Data Import/Export manual*, which is accessible through Help in the R menus.

Metadata

Metadata is the information that describes the properties of a data set: the names of the variables, the units they were measured in, when and where the data were collected, etc.. R does not have a structured system for maintaining metadata, but it does allow you to include a good deal of this metadata within your data file, and it is good practice to keep as much of this information as possible associated with the data file. Some tips on metadata in R:

- Column names are the first row of the data set. Choose names that compromise between convenience (you will be typing these names a lot) and clarity; `larval_density` or `larvdens` are better than either `x` or `larval_density_per_m3_in_ponds`. Use underscores or dots to separate words in variable names, not spaces.

[†]Unless your data are truly voluminous, you should also save a hard-copy, archival version of your data (Gotelli and Ellison, 2004).

- R will ignore any information on a line following a #. I usually use this comment character to include general metadata at the beginning of my data file, such as where and when the data were collected, what units it is measured in, and so forth — anything that can't easily be encoded in the variable names. I also use comments before, or at the ends of, particular lines in the data set that might need annotation, such as the circumstances surrounding questionable data points. You can't use # to make a comment in the middle of a line: use a comment like `# pH calibration failed` at the end of the line to indicate that a particular field in that line is suspect.
- if you have other metadata that can't easily be represented in plain-text format (such as a map), you'll have to keep it separately. You can reference the file in your comments, keep a separate file that lists the location of data and metadata, or use a system like Morpho (from `ecoinformatics.org`) to organize it.

Whatever you do, make sure that you have some workable system for maintaining your metadata. Eventually, your R scripts — which document how you read in your data, transformed it, and drew conclusions from it — will also become a part of your metadata. As mentioned in Chapter 1, this is one of the advantages of R over (say) Excel: after you've done your analysis, *if you were careful to document your work sufficiently as you went along*, you will be left with a set of scripts that will allow you to verify what you did; make minor modifications and re-run the analysis; and apply the same or similar analyses to future data sets.

Shape

Just as important as electronic or paper format is the organization or *shape* of your data. Most of the time, R prefers that your data have a single *record* (typically a line of data values) for each individual observation. This basically means that your data should usually be in "long" (or "indexed") format. For example, the first few lines of the seed removal data set look like this, with a line giving the number of seeds present for each station/date combination:

```
  station       date dist species seeds
1       1 1999-03-23   25     psd     5
2       1 1999-03-27   25     psd     5
3       1 1999-04-03   25     psd     5
4       2 1999-03-23   25     uva     5
```

```
5       2 1999-03-27   25     uva     5
6       2 1999-04-03   25     uva     5
```

Because each station has seeds of only one species and can only be at a single distance from the forest, these values are repeated for every date. During the first two weeks of the experiment no seeds of `psd` or `uva` were taken by predators, so the number of seeds remained at the initial value of 5.

Alternatively, you will often come across data sets in "wide" format, like this:

```
  station species dist seeds.1999-03-23 seeds.1999-03-27
1       1     psd   25                5                5
2       2     uva   25                5                5
3       3     pol   25                5                4
4       4     dio   25                5                5
5       5     cor   25                5                4
6       6     abz   25                5                5
```

(I kept only the first two date columns in order to make this example narrow enough to fit on the page.)

Long format takes up more room, especially if you have data (such as `dist` above, the distance of the station from the edge of the forest) that apply to each station independent of sample date or species (which therefore have to be repeated many times in the data set). However, you'll find that this format is typically what statistical packages request for analysis.

It is possible to read data into R in wide format and then convert it to long format. R has several different functions — `reshape` and `stack/unstack` in the base package, and `melt/cast/recast` in the `reshape` package[*] — that will let you switch data back and forth between wide and long formats. Because there are so many different ways to structure data, and so many different ways you might want to aggregate or rearrange them, software tools designed to reshape arbitrary data are necessarily complicated (Excel's pivot tables, which are also designed to restructure data, are as complicated as `reshape`).

- `stack` and `unstack` are simple but basic functions — `stack` converts from wide to long format and `unstack` from long to wide; they aren't

[*]If you don't know what a package is, go back and read about them in the R supplement for Chapter 1.

very flexible.

- `reshape` is very flexible and preserves more information than `stack`/`unstack`, but its syntax is tricky: if `long` and `wide` are variables holding the data in the examples above, then

  ```
  > reshape(wide, direction = "long", timevar = "date",
  +     varying = 4:5)
  > reshape(long, direction = "wide", timevar = "date",
  +     idvar = c("station", "dist", "species"))
  ```

  convert back and forth between them. In the first case (wide to long) we specify that the time variable in the new long-format data set should be `date` and that columns 4–5 are the variables to collapse. In the second case (long to wide) we specify that `date` is the variable to expand and that `station`, `dist` and `species` should be kept fixed as the identifiers for an observation.

- the `reshape` package contains the `melt`, `cast`, and `recast` functions, which are similar to `reshape` but sometimes easier to use: e.g.

  ```
  > library(reshape)
  > recast(wide, formula = ... ~ ., id.var = c("station",
  +     "dist", "species"))
  > recast(long, formula = station + dist + species ~
  +     ..., id.var = c("station", "dist", "species",
  +     "date"))
  ```

  in the formulas above, `...` denotes “all other variables” and `.` denotes “nothing”, so the formula `...~.` means “separate out by all variables” (long format) and `station+dist+species~...` means “separate out by station, distance, and species, put the values for each date on one line”

In general you will have to look carefully at the examples in the documentation and play around with subsets of your data until you get it reshaped exactly the way you want. Alternatively, you can manipulate your data in Excel, either with pivot tables or by brute force (cutting and pasting). In the long run, learning to reshape data will pay off, but for a single project it may be quicker to use brute force.

2.3.2 Reading in data

Basic R commands

The basic R commands for reading in a data set, once you have it in a long-format text file, are `read.table` for space-separated data and `read.csv` for comma-separated data. If there are no complications in your data, you should be simply be able to say (e.g.)

```
> data = read.table("mydata.dat", header = TRUE)
```

(if your file is actually called `mydata.dat` and includes a first row with the column names) to read your data in (as a *data frame* — see p. 50) and assign it to the variable `data`.

There are several potential complications to reading in files, which are more fully covered in the R supplement: (1) finding your data file on your computer system (i.e., telling R where to look for it); (2) checking that every line in the file has the same number of variables, or *fields* — R won't read it otherwise; and (3) making sure that R reads all your variables as the right data types (discussed in the next section).

2.4 DATA TYPES

When you read data into a computer, the computer stores those data as some particular data *type*. This is partly for efficiency — it's more efficient to store numbers as strings of bits rather than as human-readable character strings — but its main purpose is to maintain a sort of metadata about variables, so the computer knows what to do with them. Some operations only make sense with particular types — what should you get when you try to compute `2+"A"`? `"2A"`? If you try to do something like this in Excel you get an error code — `#VALUE!`; if you do it in R you get the message `Error ...non-numeric argument to binary operator`*.

Computer packages vary in how they deal with data. Some lower-level languages like C are *strongly typed*; they insist that you specify exactly what type every variable should be, and require you to convert variables between types (say integer and real, or floating-point) explicitly. Languages or packages like R or Excel are looser, and try to guess what you have in mind and convert variables between types (*coerce*) automatically as appropriate.

*the `+` symbol is called a "binary operator" because it is used to combine two values