Introduction to R for Statisticians

Ko-Kang Wang
MSc Student
Department of Statistics
New Zealand

20 June 2003

Contents

Chapter 1

Introduction

1.1 What is R?

In the late 1970s to early 1980s, John Chambers from the then Bell Laboratory, with the assistance of Rick Becker, Allan Wilks and Duncan Temple Lang, developed a language called S. The development of the language was with collaboration of many distinguished researchers from AT&T and a wide groups of academics. In 1980, the first version of S was released to the outside world. It is a "language and computational environment designed specifically for carrying out statistical computations".

S version 2 was released in 1987 with the added facility for user-defined extensions. This means that end-users can develop their own extensions to the language. Then in 1990 S version 3 was released with some basic feature of object-orientation which was considered essential for providing good modelling facilities.

Now, S is in its version 4 (released in 2000). It now has sophisticated object-oriented features and a number of improvements in its development process.

There are many *dialects* of the S language. The two main and most well known are R and Insightful's Splus. Splus is a commercial implementation of S, whereas R is an open-sourced language like the original S. R was originally developed by two eminent statisticians, Dr. Ross Ihaka and Dr. Robert Gentleman, from the Department of Statistics at the University of Auckland in the early 1990s (Ihaka and Gentleman, 1996). Throughout the last decade, R has matured and grew into a wide range of disciplines, not only restricted in academia but also in the commercial world and research institutions. Since 1997, there has been a group of people called **The R Core Team**, who has write access to the master R source codes (which is currently stored at a server in a nice little room at the University of Wisconsin), There are currently 17 members in the team, from all over the world.

1.2 Online Resources

CRAN http://cran.r-project.org

R Homepage www.r-project.org

1.3 The R Foundation

Chapter 2

Syntax and Data Structure

2.1 Syntax

To be able to do anything in R, one must first understand the basic syntax R uses. Each line begins with either `>` or `+`. The former is a *prompt* sign, while the later is a *continuation* sign. The prompt sign means that you need to tell R to do something, while the continuation sign means your previous command has not been finished. For example:

```
Beginning of code
> 3 + 5
End of code
```

after the prompt sign, you asked R to calculate `3 + 5`.
Sometimes, as we will see later on, it is very useful to break a command into two or more lines. A trivial example is shown below:

```
Beginning of code
> 3 +
+ 5
End of code
```

as you can see, if I pressed `Enter` after `3 +`, the second line begins with a `+` sign, showing that you have not finished the first line.
Basic arithematic operators in R are the same as in most other statistical packages. That is:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `^` power operation

For example:

```
Beginning of code
> 3 + 5
[1] 8
> 3 - 5
[1] -2
```

```
> 3 * 5
[1] 15
> 3 / 5
[1] 0.6
> 3 ^ 5
[1] 243
```
End of code

Arithematic operations on matrices is slightly different, as you have to do it element-wise. We will look at it later.

All the functions in R are extremely well documented. To read the documentation, you can use either `?` or `help()`. For example, suppose you want to learn how to use the `plot()` function, then either of the following lines will do:

Beginning of code
```
> ?plot
> help(plot)
```
End of code

It is strongly recommended that you make use of the documentations. You will find that almost all of your questions can be answered there.

2.2 Vectors

The most basic type in R is the atomic vector. A vector contains indexed set of values of the same type:

- logical
- numeric – can be broken down into *integer*, *single* and *double* types. It is not important in R but is very important if you want to write code in C or Fortran, and call them from within R.
- complex
- character

There are several ways to create a vector in R. If you want to create vectors that have patterns, then you can use either `seq()` or `:` if it is a simple sequence; or `rep()` if a more complicated sequence. If there are no patterns, then the `c()` function can be used. For example:

Beginning of code
```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10

> seq(1, 10, by = 0.5)
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0
[12]  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0

> seq(1, 10, length = 21)
 [1]  1.00  1.45  1.90  2.35  2.80  3.25  3.70  4.15  4.60
```

```
[10]  5.05  5.50  5.95  6.40  6.85  7.30  7.75  8.20  8.65
[19]  9.10  9.55 10.00

> rep(2:5, 2)
[1] 2 3 4 5 2 3 4 5

> rep(2:5, rep(2, 4))
[1] 2 2 3 3 4 4 5 5

> x = c(42, 7, 64, 9)

> length(x)
[1] 4
```

End of code

Chapter 3

Basic Statistics and Exploratory Data Analyses

3.1 Input Data

3.2 Summary Statistics

3.3 Basic Analyses and Plots

3.4 Analysis of Variance (ANOVA)

3.5 Dynamic Exploratory Data Analyses

Wang (2003b)

Chapter 4

Linear Regression

4.1 Simple Linear Regression

4.2 Multiple Linear Regression

4.3 Logistic Regression

4.4 Multivariate Linear Regression

Chapter 5

R Graphics

Chapter 6

Smoothing and Kernel Density Estimation

Chapter 7

Multivariate Analyses and Data Mining

7.1 Dimensional Reduction Techniques

7.1.1 Principle Componenet Analysis (PCA)

7.1.2 Biplots

7.1.3 Factor Analysis

7.2 Multivariate Analysis of Variance (MANOVA)

7.3 Cluster Analysis

7.4 Tree-Based Models

Tree-based models have been in existence since the early 1960s. At first they were used extensively by the decision makers in medical fields, botanists for classification purposes and by computer scientists for machine learning. The main advantage of tree-based models is that they are extremely easy to understand and interpret for non-experts, provided the tree itself is not too large.

There are two types of trees:

Classification Trees for categorical response variable

Regression Trees for continuous response variable

7.4.1 Terminology

Throughout Section 7.4.1–7.4.3, the Fisher's Iris data set (Fisher, 1936) will be used to explain the concepts of a classification tree. There are three sub-species of the Iris flower in the data: Iris setosa, Iris versicolor, and Iris virginica. They are denoted by `s`, `c`, and `v` respectively.

We use the following terminologies for tree-based models.

- If we let $\mathcal{T}$ denote a tree, and $|\mathcal{T}|$ denotes the number of nodes. The number of leaves, or the *size* of the tree, is denoted by $|\widetilde{\mathcal{T}}|$. In Fig. 7.1, $|\mathcal{T}| = 5$, and $|\widetilde{\mathcal{T}}| = 3$.
- A *Node* is denoted by τ, which may be an internal node or a terminal node (leaf). In Fig. 7.1, the internal nodes are `Petal.Length>=2.45` and `Petal.Width<1.75`. The leaves in the figure are `c`, `v`, and `s`.
- The root is considered to be an internal node. The root in Fig. 7.1 is `Petal.Length>=2.45`.
- The *depth* of a node τ is the length of the path from the root to τ.
- The number of *layers*/*levels* is the maximum depth of $\mathcal{T}$ + 1. So, in Fig. 7.1, it is 4. The first layer consists of the root only.
- Subtree $\mathcal{T}^*$ is a tree with a node in $\mathcal{T}$ as its root.

7.4.2 Univariate-Threshold-Split Binary Tree

A tree is called a *binary tree* if each node is only allowed to have 2 branches, an example of a binary tree is shown in Fig. 7.1.

A tree is a *univariate-split* tree, if only one variable is permitted to be split on at any given node. It is a *univariate-threshold-split* tree, if all the continuous variables in a univariate-split binary tree depend solely on whether the variable is less or greater than a given value, it is of the form $x_k > t_k$ where t_k is the threshold. Fig. 7.1 is an example of a univariate-threshold-split tree. One can look at it in another form, as shown in Fig. 7.2. The first split is on `Petal Length` at 2.45, while the second split is on `Petal Width` at 1.75.

A *surrogate split rule* is a "back-up" for a main splitting rule. For example, in Fig. 7.1, the first split is on `Petal Length`. If it is missing from an observation, an alternative measurement may be used, for example.

7.4.3 Under- and Over-Fitting

When fitting a tree, it is very important not to over-fit or under-fit it. If a tree is under-fitted, then it will not be flexible enough and may overlook the important structure in the data; on other other hand, if a tree is over-fitted, the resulting model becomes too complex and captures too much noise, and will not be representative for new samples.

The concepts of under-fitting versus over-fitting is best understood by a visual example. The top figure in Fig. 7.3 shows the correct classification of the Iris data, the bottom two figures show under-fitting and over-fitting.

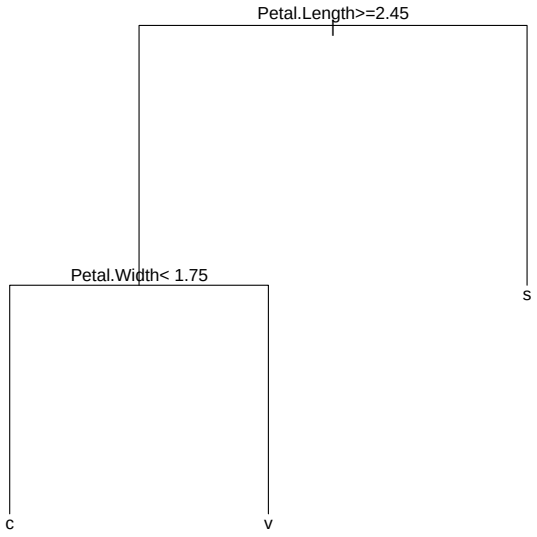

Figure 7.1: An Example of a binary classification tree. s = Iris setosa; c = Iris versicolor; v = Iris virginica.

7.4.4 Splitting Rules

One of the most important steps in the construction of a tree is when to split a node. Observations within a node need to be as homogeneous as possible, and observations between groups need to be as different as possible. In other words, one needs to increase the *distance* (difference) between nodes. A quantitative measure of this difference is called the **node impurity**.

Let $i(\tau)$ be the impurity function at τ, where τ is an internal node, and the *goodness of split*, S, be the decrease in impurity. Then:

$$\Delta i(S, \tau) \quad = \quad i(\tau) - [i(\tau_L) + i(\tau_R)]$$

One will want to maximise the value $\Delta i(S, \tau)$ over all possible splits S of node τ.

There are many measures of impurity, three of the most common are *Misclassification Error*, *Gini Index*, and *Entropy/Deviance*. As the response variable is binary, let p be the proportion in class 2. Then

Misclassification Error $1 - \max(p, 1-p)$

Gini Index $2p(1-p)$

Entropy/Deviance $-p \log p - (1-p) \log(1-p)$

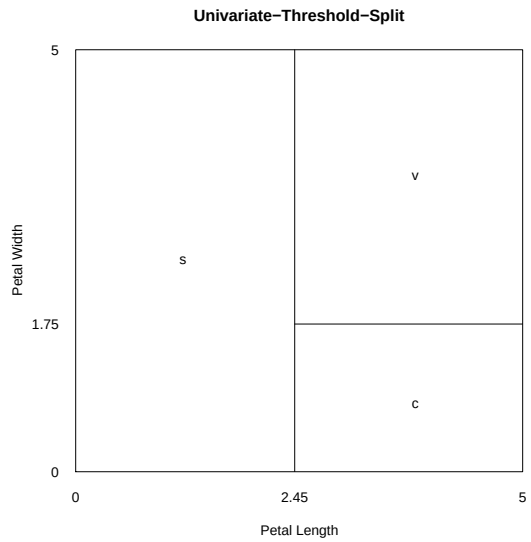

Figure 7.2: An Example of Univariate-Threshold-Split

A graphical visualisation of the three measurements is shown in Fig. 7.4.

Hastie et al. (2001) suggest that one should use either the entropy or the Gini Index. One of the reasons for this is clear in Fig. 7.4. Both entropy and Gini Index are differentiable, hence are better for numerical optimisation.

7.4.5 Doing in R

There are many R packages that allow you to get tree-based models: `knnTree`, `randomForest`, `rpart`, and `tree`. Of these, `rpart` is the most commonly used and the easiest one to learn. Venables and Ripley (2002) has a detailed explanation on the `tree` package. In this section I will be using `rpart` and the algorithm I use will be CART (Breiman et al., 1984). CART uses the Gini Index to split, and this is the default in `rpart()`.

By default, the `rpart()` function will attempt to contruct a tree with a minimum number of observation of 20 in a node, with a minimum number of observation of 7 in a leaf. The maximum number of surrogate rules used is defaulted to be 5. The complexity paramter (cp) is set to 0.01 Venables and Ripley (2002). These parameters can be modified in `rpart.control`.

For example, we can fit a default `rpart()` tree to the German credit data using:

```
> ger.rp <- rpart(credit ~ ., data = german, method = "class")
```

This gives us the following text-format tree:

```
n= 1000
```

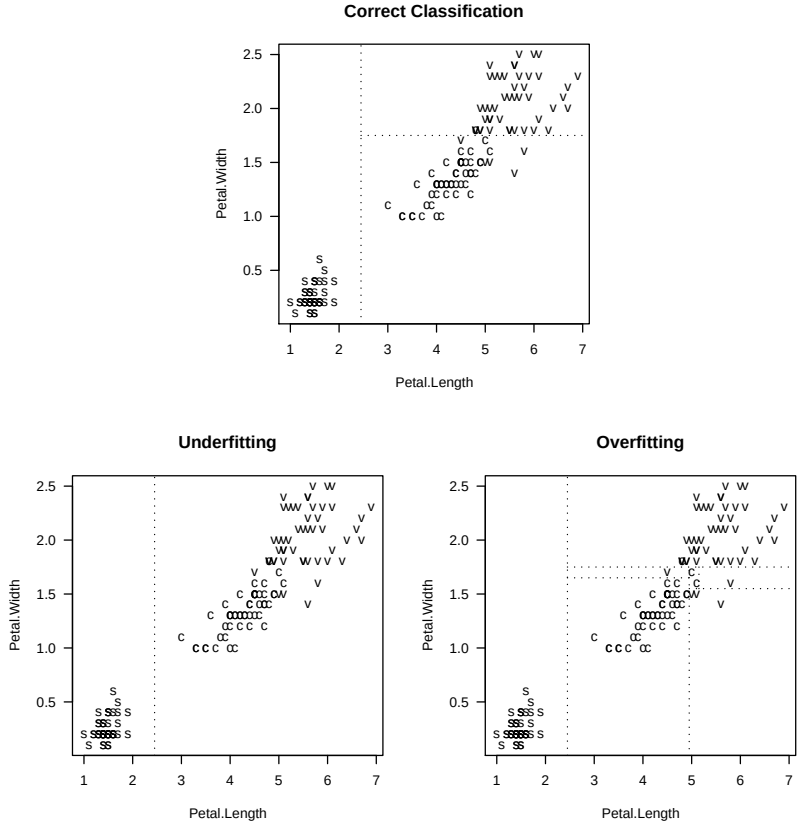

Figure 7.3: An Example of Under-fitting v.s. Over-fitting

```
node), split, n, loss, yval, (yprob)
      * denotes terminal node

  1) root 1000 300 1 (0.7000000 0.3000000)
    2) status=A13,A14 457  60 1 (0.8687090 0.1312910) *
    3) status=A11,A12 543 240 1 (0.5580110 0.4419890)
      6) duration< 22.5 306 106 1 (0.6535948 0.3464052)
       12) history=A32,A33,A34 278  85 1 (0.6942446 0.3057554)
         24) amount< 7491.5 271  79 1 (0.7084871 0.2915129)
           48) purpose=A40,A41,A410,A42,A43,A45,A48,A49 256  69 1 (0.7304688 0.2695312)
             96) duration< 11.5 73   9 1 (0.8767123 0.1232877) *
             97) duration>=11.5 183  60 1 (0.6721311 0.3278689)
              194) amount>=1387.5 118  29 1 (0.7542373 0.2457627) *
              195) amount< 1387.5 65  31 1 (0.5230769 0.4769231)
                390) real=A121,A122 45  14 1 (0.6888889 0.3111111) *
                391) real=A123,A124 20   3 2 (0.1500000 0.8500000) *
           49) purpose=A44,A46 15   5 2 (0.3333333 0.6666667) *
         25) amount>=7491.5 7   1 2 (0.1428571 0.8571429) *
       13) history=A30,A31 28   7 2 (0.2500000 0.7500000) *
```

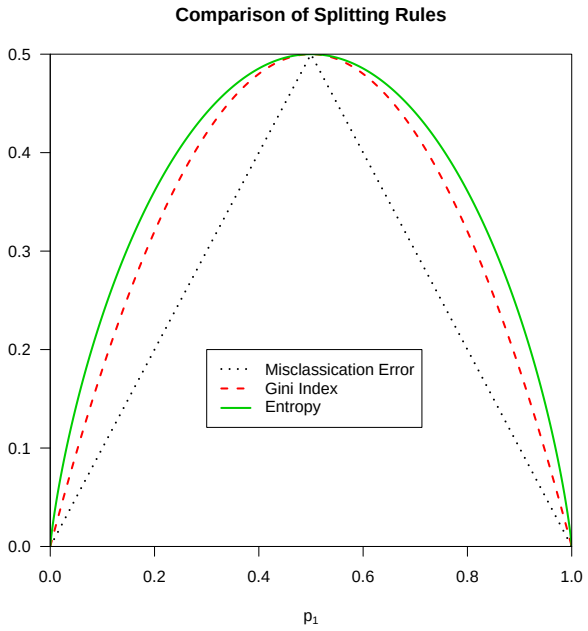

Figure 7.4: Measure of Node Impurity for Binary Response. Entropy has been scaled to pass through (0.5, 0.5)

```
7) duration>=22.5 237 103 2 (0.4345992 0.5654008)
  14) savings=A64,A65 41  12 1 (0.7073171 0.2926829) *
  15) savings=A61,A62,A63 196  74 2 (0.3775510 0.6224490)
    30) duration< 47.5 160  69 2 (0.4312500 0.5687500)
      60) purpose=A41 23   6 1 (0.7391304 0.2608696) *
      61) purpose=A40,A410,A42,A43,A45,A46,A49 137  52 2 (0.3795620 0.6204380) *
    31) duration>=47.5 36   5 2 (0.1388889 0.8611111) *
```

Of course, it is easier to visualise a picture of the tree. This can be done using the `plot()` and `text()` commands:

```
> plot(ger.rp)
> text(ger.rp, cex = .7)
```

and this gives us Fig 7.5

```
n= 1000

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 1000 300 1 (0.70000000 0.30000000)
   2) status=A13,A14 457  60 1 (0.86870897 0.13129103) *
   3) status=A11,A12 543 240 1 (0.55801105 0.44198895)
     6) duration< 31.5 434 170 1 (0.60829493 0.39170507)
      12) amount< 10975.5 426 162 1 (0.61971831 0.38028169)
```

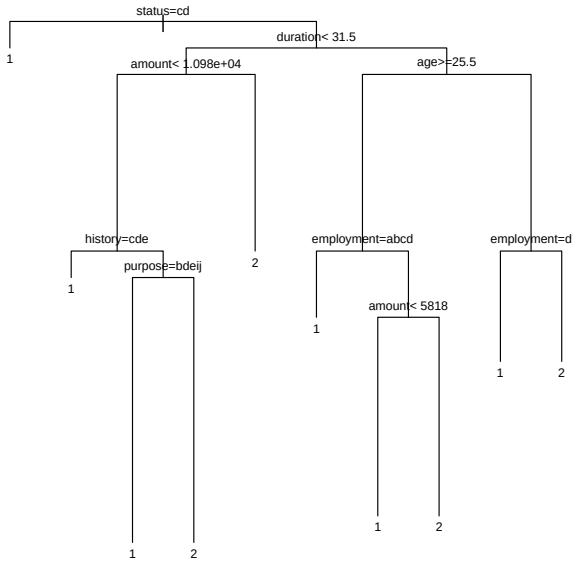

Figure 7.5: Default `rpart()` on the German Credit Data

```
   24) history=A32,A33,A34 377 128 1 (0.66047745 0.33952255) *
   25) history=A30,A31 49  34 1 (0.30612245 0.69387755)
     50) purpose=A41,A42,A43,A48,A49 31  17 1 (0.45161290 0.54838710) *
     51) purpose=A40,A410,A45,A46 18   5 2 (0.05555556 0.94444444) *
 13) amount>=10975.5 8   0 2 (0.00000000 1.00000000) *
7) duration>=31.5 109  70 1 (0.35779817 0.64220183)
 14) age>=25.5 89  52 1 (0.41573034 0.58426966)
   28) employment=A71,A72,A73,A74 68  34 1 (0.50000000 0.50000000) *
   29) employment=A75 21  15 2 (0.14285714 0.85714286)
     58) amount< 5818 9   6 1 (0.33333333 0.66666667) *
     59) amount>=5818 12   0 2 (0.00000000 1.00000000) *
 15) age< 25.5 20  10 2 (0.10000000 0.90000000)
   30) employment=A74 7   5 1 (0.28571429 0.71428571) *
   31) employment=A72,A73,A75 13   0 2 (0.00000000 1.00000000) *
```

7.5 Discriminant Function Analyses

7.6 Neural Networks

In the early 1980s, computer scientists actively pursued research with a *black box non-linear* technique called **neural networks**. It can be conceptualised to

be mimicking a human brain, and is intended to be the *ultimate* model for everything. Fig. 7.6 shows a **single-layer feed-forward neural network** model.

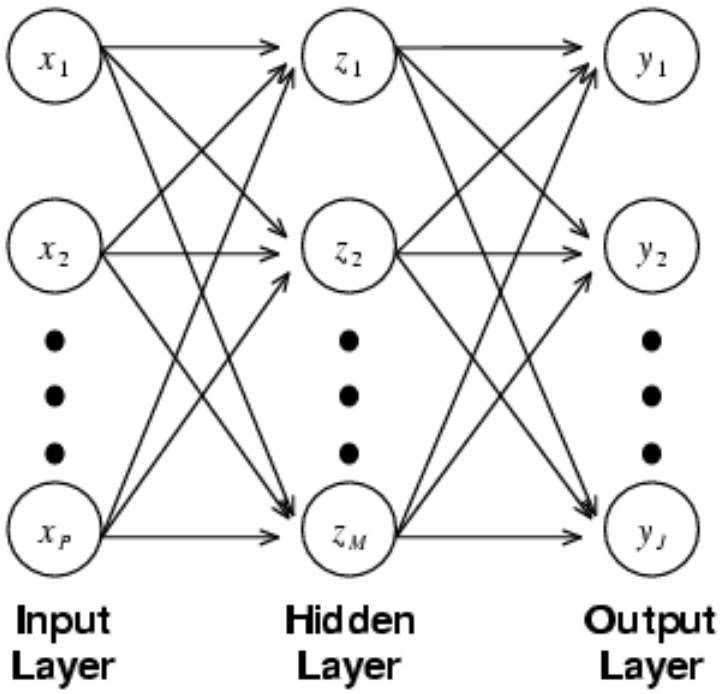

Figure 7.6: A Single-Layer Feed-Forward Neural Network

In Fig. 7.6, there are three *layers*. All technical terms will be explained in Section 7.6.1.

As this is a black box regression technique, it was at first shunned by statisticians. They declared it as a non-reliable technique that would not survive over time. However, to the statisticians' dismay, neural networks have not only survived throughout the last two decades, but also they flourished and are becoming more and more popular. Professor Trevor Hastie has stated that:

> While neural networks probably get more attention than they deserve in the scientific community at large, they in turn get less attention than they deserve from statisticians. (Hastie, 1996)

7.6.1 Terminology

The notation used in this section is based on Fig. 7.6.

Let **x** represent nodes in the input layer; **z** represent nodes in the hidden layer; and **y** represent nodes in the output layer. Then the model can be represented

mathematically as:

$$\begin{aligned} z_m &= \sigma\left(\alpha_{0m} + \boldsymbol{\alpha}_m^T \mathbf{x}\right), \qquad m = 1, \ldots, M, \\ f_j(x) &= g_j\left(\beta_{0j} + \boldsymbol{\beta}_j^T \mathbf{Z}\right), \qquad j = 1, \ldots, J \end{aligned} \tag{7.1}$$

The *activation function*, σ, is used to introduce non-linearities into the hidden layer. A function from the sigmoid family, $\sigma(z) = 1/\left(1 + e^{-z}\right)$, is often used for this. The *weights* (regression coefficients), $\boldsymbol{\alpha}_m$ and $\boldsymbol{\beta}_j$, are to be estimated. They define linear combinations of the input vector $\mathbf{x}$ and $\mathbf{z}$. The *biases* are α_{0m} and β_{0j}. The function g_j is a final transformation function of the output, which is either an identity, a linear function, or a non-linear function. We write $\mathbf{Z} = (z_1, \ldots, z_M)^T$, and each "node" in Fig. 7.6 is called a *unit*.

Fig. 7.6 only has one hidden layer—hence it is called *single-layer*. However, a neural network model can have several hidden layers, in which case it will be called a *multi-layer* model. In most cases, a single-layer model will be enough. In SAS Enterprise Miner, the term *multilayer perception* is used for both single-layer and multi-layer models. If one only specifies one hidden-layer in the multilayer perception model, then it is a single-layer neural network; if one specifies more than one hidden-layer, then one gets a multi-layer model.

The neural networks function is very similar to Project Pursuit Regression (PPR), which has the form:

$$Y_j(x) = \sum_{j=1}^{M} g_j\left(\boldsymbol{\alpha}_j^T \mathbf{x}\right) + \varepsilon$$

with the difference that in PPR the transformation function is usually smooth and non-parametric (usually some variant of splines), whereas in neural networks the function is known and much simpler.

7.6.2 Choosing a Model

It is very difficult to choose an appropriate neural network model to fit. There are many things one has to consider, for example:

- the number of hidden layers,
- the number of hidden units in each layer,
- what activation function to use (from input layer to hidden layer; and from hidden layer to output layer),
- what error function to use,
- what optimisation method to use.

Unfortunately, there are no good rules of thumb that one can use. The choice is particularly difficult when it comes to the number of hidden layers and units. Although in most cases a single-layer is enough, one still has to decide how many hidden units to put into the hidden layer. Two common methods to decide the number of hidden units are:

- Hastie et al. (2001) suggest using somewhere between 5 to 100 hidden units in a hidden layer, with the number increasing with the number of inputs and training cases.
- SAS (2003) recommends that one should start with no hidden units[1], add one at a time and compare the error rates and predictive power.

7.6.3 Doing in R

[1] In which case the model becomes either a simple linear regression or a logistic regression, depending on the response variable.

Chapter 8

Programming in R

The aim of this chapter is to provide a basic understanding of how one can program in R. It is assumed that you have some basic statistics and mathematics knowledge but may have never programmed before.

If you are have programmed in R or are an experienced R programmer, then this chapter may not be suitable for you. I recommend that you start from (Chambers, 1998) or (Venables and Ripley, 2000).

8.1 Importance of Programming

The word *programming* scares many people. Some of the people–statisticians–I have talked to are either scared of it or cannot see the point of it. They think that a good statistical package/software should incorporate everything, and allows one to perform all known statistical techniques under the sun.

This is a wrong perception. For one thing, a statistcal package that does everything will be very expensive to implement (both in terms of the monetary costs and person-hour costs). A classical example of this is the SAS software. The gold license include every component and will give one an extremely powerful statistical package, but it costs a fortune to buy.

The idea of learning basic programming is to allow one to implement algorithms that have not been defined in a package. One of the strengths of R is that not only it is free and open-sourced, but also it allows people to implement their own functions or packages to suit their needs.

Beginning of code

```
foo = read.table()
x = 1:3
```

End of code

8.2 Good Programming Habits

Before we start looking at the syntax structure in R, I want to emphasise on some good programming habits.

When you write a programme, no matter what language is used, the most important thing that you must bear in mind is **readability** and **understandabilitiy**. If another person wants to see your codes, he/she has to be able to at least read it. More importantly, when **YOU** comes back to read your codes in a few months, or even weeks, you want to be able to understand what you have done. The matters can be handled by having good habits when you write your codes.

Indentation is extremely important. Indentation means that you do not start all your lines at the same place. Nor should you write all your codes in one line. When you have an open braket at the end of a line, the next line should be *indented*—right-shift with a few spaces. For example[1]:

```
for (i in seq(along = nmstrata)) {
  select <- asgn.e == (i - 1)
  ni <- sum(select)
  if (!ni)
    next
  xi <- qtx[select, , drop = FALSE]
  cols <- colSums(xi^2) > 1e-05
  if (any(cols)) {
    xi <- xi[, cols, drop = FALSE]
    attr(xi, "assign") <- asgn.t[cols]
    fiti <- lm.fit(xi, qty[select, , drop = FALSE])
    fiti$terms <- Terms
  }
  else {
    y <- qty[select, , drop = FALSE]
    fiti <- list(coefficients = numeric(0), residuals = y,
                 fitted.values = 0 * y, weights = wts, rank = 0,
                 df.residual = NROW(y))
  }
  if (projections)
    fiti$projections <- proj(fiti)
  class(fiti) <- c(if (maov) "maov", "aov", oldClass(er.fit))
  result[[i]] <- fiti
}
```

is a good example of well-indented codes. It is certainly much more readable than the following unindented codes:

```
for (i in seq(along = nmstrata)) {
select <- asgn.e == (i - 1)
ni <- sum(select)
if (!ni)
next
xi <- qtx[select, , drop = FALSE]
```

[1] Extracted from the `aov()` function in R. You do not need to understand them, it is just used to show indentation.

```
cols <- colSums(xi^2) > 1e-05
if (any(cols)) {
xi <- xi[, cols, drop = FALSE]
attr(xi, "assign") <- asgn.t[cols]
fiti <- lm.fit(xi, qty[select, , drop = FALSE])
fiti$terms <- Terms
}
else {
y <- qty[select, , drop = FALSE]
fiti <- list(coefficients = numeric(0), residuals = y,
fitted.values = 0 * y, weights = wts, rank = 0,
df.residual = NROW(y))
}
if (projections)
fiti$projections <- proj(fiti)
class(fiti) <- c(if (maov) "maov", "aov", oldClass(er.fit))
result[[i]] <- fiti
}
```

Programming languages do not care whether you indent, as the spaces in front of a line are usually ignored. The purpose of indentation is for humans. The number of spaces in an indented line varies, but is usually taken to be either 2 or 4 character-space and no more than 8. If you have more than 8 spaces then you will quickly run out of page margins. Any good text editor allows you to do indentation easily. To name a few: (X)Emacs (available for both Linux/Unix and Windows), UltraEdit (Windows), vi (both Linux/Unix and Windows), Crimson Editor (Windows). Some of them are free while others are shareware. A google search will allow you to find more editors.

Documenting/Commenting you codes is also vitally important. Most, if not all, of us will forget why we write a particular piece of codes or the logical flow of the codes, in a few weeks time. Putting *comments* will allow you to remember. In R, a line start with `#` is treated as comments and any words after this character are ignored.

Variable naming is also important. It is a very bad habbit to assign a variable with the name of your favourite person/animal/pet/fruit...etc. It tells nothing! For example, if you are fitting a simple linear regression model using:

```
fit <- lm(y ~ x)
```

will be much more meaningful than:

```
fish <- lm(y ~ x)
```

(unless of course your response variable or the data has got something to do with fish...)

It is very important to have a good habits when you start learning how to programme, as once you have developed a habit it is very hard to change at a later time.

8.3 Writting Simple Functions

In R, if you write a block of codes that does something and you put them together, it is called a **function**. Here is an example of an R function:

```
# Using the Gamma property to compute n!
factorial <- function(n) {
  gamma(n + 1)
}
```

The example computes the factorial of a given number, n, using the property of the Gamma function: $\Gamma(n+1) = n!$. It can be used like:

```
> factorial(3)
[1] 6
> factorial(4)
[1] 24
```

If you want to print out a statement directly, you can use the `print()` function in R. The function can also be very useful for debugging purposes, as we will see in Section 8.4. An example of using the function is shown below:

```
# Hello World!
hello <- function() {
  print("Hello World!")
}
```

8.4 More complicated functions

Of course, it is highly unlikely that your function will only consists of one line of code. Even if it is a simple function like factorial, you should consider putting some debugging statement. You need to consider the possibility that a user may provide an incorrect input to your function. In the factorial example, if the user provide a non-numeric input, or a number that is negative, then the function should provide some sensible error messages. The above example can be re-written as:

8.5 Conditional Statement

8.6 Loops

8.7 Recursion

8.8 Object-Oridented Programming

Bibliography

Breiman, L., Friedman, J., Olshen, R., Stone, C., 1984. Classification and Regression Trees. Wadsworth.

Chambers, J., 1998. Programming with Data. Springer, New York.
URL http://cm.bell-labs.com/cm/ms/departments/sia/Sbook/

Fisher, R. A., 7 1936. The use of multiple measurements in taxonomic problem. Annals of Eugenic Part II, 179–188.

Hastie, T., 1996. Neural Networks. Wiley, Ch. 1, pp. 1–7.

Hastie, T., Tibshirani, R., Friedman, J., 2001. The Elements of Statistcal Learning: Data Mining, Inference, and Prediction. Springer.

Ihaka, R., Gentleman, R., 1996. R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics 5 (3), 299–314.

SAS, 2003. SAS Enterprise Miner Reference. SAS.

Venables, W., Ripley, B., 2000. S Programming. Springer, iSBN 0-387-98966-8.
URL http://www.stats.ox.ac.uk/pub/MASS3/Sprog/

Venables, W. N., Ripley, B. D., 2002. Modern Applied Statistics with S. Fourth Edition. Springer, iSBN 0-387-95457-0.
URL http://www.stats.ox.ac.uk/pub/MASS4/

Wang, K., 7 2003a. A data mining analysis of actuarial data. Master's thesis, University of Auckland.

Wang, K., 2003b. Ggobi for dummies under windows.
URL http://www.stat.auckland.ac.nz/~kwan022/rinfo.php