A Tour of Recommender Systems

Norm Matloff

University of California, Davis

Ancient "Yelp"

About This Book

The author has striven to minimize the number of errors, but no guarantee is made as to accuracy of the contents of this book.

The cover is from the British Museum onine site,

`https://www.britishmuseum.org/research/collection_online/collection_object_details/collection_image_gallery.aspx?partid=1&assetid=1613004116&objectid=277770`

with descripition:

> Clay tablet; letter from Nanni to Ea-nasir complaining that the wrong grade of copper ore has been delivered after a gulf voyage and about misdirection and delay of a further delivery...

Author's Biographical Sketch

Dr. Norm Matloff is a professor of computer science at the University of California at Davis, and was formerly a professor of statistics at that university. He is a former database software developer in Silicon Valley, and has been a statistical consultant for firms such as the Kaiser Permanente Health Plan.

Prof. Matloff's recently published books include'*Parallel Computation for Data Science* (CRC, 2015) and *Statistical Regression and Classification: from Linear Models to Machine Learning* (CRC 2017). The latter book was selected for the Ziegal Award in 2017.

Dr. Matloff was born in Los Angeles, and grew up in East Los Angeles and the San Gabriel Valley. He has a PhD in pure mathematics from UCLA, specializing in probability theory and statistics. He has published numerous papers in computer science and statistics, with current research interests in parallel processing, statistical computing, and regression methodology.

Prof. Matloff is a former appointed member of IFIP Working Group 11.3, an international committee concerned with database software security, established under UNESCO. He was a founding member of the UC Davis Department of Statistics, and participated in the formation of the UCD Computer Science Department as well. He is a recipient of the campuswide Distinguished Teaching Award and Distinguished Public Service Award at UC Davis.

Contents

Chapter 1

Setting the Stage

Let's first get an overview of the topic and the nature of this book. Keep in mind, this is just an overview; many questions should come to your mind, hopefully whetting your appetite for the succeeding chapters!

In this chapter, we will mainly describe *collaborative filtering*.

1.1 What Are Recommender Systems?

What is a recommender system (RS)? We're all familiar with the obvious ones — Amazon suggesting books for us to buy, Twitter suggesting whom we may wish to follow, even OK Cupid suggesting potential dates.

But many applications are less obvious. The University of Minnesota, for instance, has developed an RS to aid its students in selection of courses. The tool not only predicts whether a student would like a certain course, but also even predicts the grade she would get!

In discussing RS systems, we use the terms *users* and *items*, with the numerical outcome being termed the *rating*. In the famous MovieLens dataset, which we'll use a lot, users provide their ratings of films.

Systems that combine user and item data as above are said to perform *collaborative filtering*. The first part of this book will focus on this type of RS. *Content-based* RS systems work by learning a user's tastes, say by text analysis.

Ratings can be on an ordinal scale, e.g. 1-5 in the movie case. Or they can be binary, such as a user clicking a Like symbol in Twitter, 1 for a click, 0 for no click.

But ratings in RSs are much more than just the question, "How much do you like it?" The Minnesota grade prediction example above is an instance of this.

In another example, we may wish to try to predict bad reactions to presciption drugs among patients in a medical organization. Here the user is a patient, the item is a drug, and the rating may be 1 for reaction, 0 if not.

More generally, any setting suitable for what in statistics is called a *crossed heirachical model* fits into RS. The word *crossed* here means that each user is paired with multiple items, and vice versa. The hierarchy refers to the fact that we can group users within items or vice versa. There would be two levels of hierarchy here, but there could be more.

Say we are looking at elementary school students rating story books. We could add more levels to the analysis, e.g. kids within schools within school districts. It could be, for instance, that kids in different schools like different books, and we should take that into account in our analysis. The results may help a school select textbooks that are especially motivational for their students.

Note that in RS data, most users have not rated most items. If we form a matrix of ratings, with rows representing users and columns indicating items, most of the elements of the matrix will be unknown. We are trying to predict the missing values. Note carefully that these are not the same as 0s.

1.2 How Is It Done?

Putting aside possible privacy issues that arise in some of the above RS applications,[1] we ask here, How do they do this? In this prologue, we'll discuss a few of the major methods.

1.2.1 Nearest-Neighbor Methods

This is probably the oldest class of RS methodology, still popular today. It can be explained very simply.

Say there is a movie spoofing superheroes called *Batman Goes Batty* (BGB). Maria hasn't seen it, and wonders whether she would like it. To form a predicted rating for her, we could search in our dataset for the k users most similar to Maria in movie ratings and who have rated BGB. We would then average their ratings in order to derive a predicted rating of BGB for Maria. We'll treat the issues of choosing the value of k and defining "similar" later, but this is the overview.

The above approach is called *user-based*, with a corresponding *item-based* method. In general, these

[1] I used to be mildy troubled by Amazon's suggestions, but with the general demise of browsable bricks-and-mortar bookstores, I now tend to view it as "a feature rather than a bug."

are called k-NN methods, for "k-nearest neighbor." (We'll shorten it to kNN.)

1.2.2 Latent Factor Approach: Matrix Factorization

Let A denote the matrix of ratings described earlier, with A_{ij} denoting the rating user i gives to item j. Keep in mind, as noted, that most of the entries in A are unknown; we'll refer to them as NA, the R-language notation for missing values. Matrix factorization (MF) methods then estimate all of A as follows.

Let r and s denote the numbers of rows and colums of A, respectively. In the smallest version of the MovieLens data, for example, $r = 943$ and $s = 1682$. The idea is to find a *low-rank approximation* to A: We find matrices W and H, of dimensions $r \times m$ and $m \times s$, each of rank m, such that

$$A \approx WH \tag{1.1}$$

Typically $m << \min(r, s)$. Software libraries typically take 10 as the default.

We will review the concept of matrix rank later, but for now the key is that W and H are known matrices, no NA values. Thus we can form the product WH, thus obtaining estimates for all the missing elements of A.

1.2.3 Latent Factor Approach: Statistical Models

As noted, collaborative-filtering RS applications form a special case of crossed random-effects models, a statistical methodology. In that way, a useful model for Y_{ij}, the rating user i gives item j, is

$$Y_{ij} = \mu + \alpha_i + \beta_j + \epsilon_{ij} \tag{1.2}$$

a sum of an overall mean, an effect for user i, an effect for item j, and an "all other effects" term (often called the "error term," which is rather misleading).

In the MovieLens setting, μ would be the mean rating given to all movies (in the "population" of all movies, past, present and future), α_i would be a measure of the tendency of user i to give ratings more liberal or harsher than the average user, and β_j would a measure of the popularity of movie j, relative to the average movie.

What assumptions are made here? First, μ is a fixed but unknown constant to be estimated. As to α_i and β_j, one could on the one hand treat them as fixed constants to be estimated. On the other

hand, there are some advantages to treating them as random variables, we will be seen in Chapter 6.

1.2.3.1 Comparison

Why so many methods? There is no perfect solution, and each has advantagages and disadvantages.

First, note that in the kNN and MF methods, the user must choose the value of a design parameter. In kNN, we must choose k, the number of nearest neighbors, and while in MF, we must choose m, the rank.

Parameters such as k and m are known as *tuning parameters* in statistics and *hyperparameters* in machine learning (ML). Many ML methods have multiple hyperparameters, sometimes 10 or more. This can be quite a drawback, as choosing their values is quite difficult. By contrast, the statistical model described above has no tuning parameters.

A problem with both MF and the statistical models is that one is limited to prediction only of ratings for users and items that are already in our dataset. We could not predict a new user, for instance, without recomputing an updated fit. With kNN, there is no such restriction.

1.3 Covariates/Side Information

In predicting the rating for a given (user,item) pair, we may for example have demographic information on the user, such as age and gender. Incorporating such information — called *covariates* in statistics and *side information* in machine learning — may enhance our predictive ability, especially if this user has not rated many items to date.

1.4 Prerequisites

What is background is needed for this book?

- A calculus-based probability course.
- Some facility in programming.
- Good mathematical intuition.

We will be using the R programming language. No prior experience with R is assumed. A Quickstart in R is available in Appendix A.

We will also use some machine learning techniques, but no prior background is assumed.

1.5 Software

A number of libraries are available for RS methods. We will use the R package **rectools**, available in my GitHub repo, **github.com/matloff**.

1.6 What You Should Gain from This Book

- A solid understanding of RS fundamentals: You'll be able to build simple but effective RS systems, and will be able to read books and research on advanced RS methods.
- Greatly enhanced understanding of the basics of probability/statistics, machine learning and linear algebra.

Chapter 2

Some Infrastructure: Linear Algebra

There are some issues that will come up frequently. We'll first cover them briefly here, more later as the need arises. This chapter will review linear algebra, while the following one will review/extend the reader's knowledge of probability and statistics.

RS methods, as with other machine learning (ML) techniques, often make use of linear algebra, well beyond mere matrix multiplication.

2.1 Matrix Rank and Vector Linear Independence

Consider the matrix

$$M = \begin{pmatrix} 1 & 5 & 1 & -2 \\ 8 & 3 & 2 & 8 \\ 9 & 8 & 3 & 6 \end{pmatrix} \tag{2.1}$$

Note that the third row is the sum of the first two. In many contexts, this would imply that there are really only two "independent" rows in M in some sense related to the application.

Denote the rows of M by $r_i,\ i = 1, 2, 3$. Recall that we say they are *linearly independent* if it is not possible to find scalars a_i, at least one of them nonzero, such that the *linear combination* $a_1 r_1 + a_2 r_2 + a_3 r_3$ is equal to 0. In this case $a_1 = a_2 = 1$ and $a_3 = -1$ gives us 0, so the rows of M are linearly dependent.

Recall that the *rank* of a matrix is its maximal number of linearly independent rows and colums. (It can be shown that the row rank and column rank are the same.) The rank of M above is 2.

Recall too the notion of the *basis* of a vector space $\mathcal{V}$. It is a linearly independent set of vectors whose linear combinations collectively form all of $\mathcal{V}$. Here r_1 and r_2 form a basis for the *row space* of M. Alternatively, r_1 and r_3 also form a basis, as do r_2 and r_3.

The rank of an $r \times s$ matrix is thus at most $\min(r, s)$. In the case of equality, we say the matrix has *full rank*. A ratings matrix, such as A in Section 1.2.2, should be of full rank, since there presumably are no exact dependencies among users or items.

2.2 Partitioned Matrices

It is often helpful to partition a matrix into *blocks* (often called *tiles* in the parallel computation community).

2.2.1 How It Works

Consider matrices A, B and C,

$$A = \begin{pmatrix} 1 & 5 & 12 \\ 0 & 3 & 6 \\ 4 & 8 & 2 \end{pmatrix} \tag{2.2}$$

and

$$B = \begin{pmatrix} 0 & 2 & 5 \\ 0 & 9 & 10 \\ 1 & 1 & 2 \end{pmatrix}, \tag{2.3}$$

so that

$$C = AB = \begin{pmatrix} 12 & 59 & 79 \\ 6 & 33 & 42 \\ 2 & 82 & 104 \end{pmatrix}. \tag{2.4}$$

We could partition A as, say,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{2.5}$$

where

$$A_{11} = \begin{pmatrix} 1 & 5 \\ 0 & 3 \end{pmatrix}, \tag{2.6}$$

$$A_{12} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}, \tag{2.7}$$

$$A_{21} = \begin{pmatrix} 4 & 8 \end{pmatrix} \tag{2.8}$$

and

$$A_{22} = \begin{pmatrix} 2 \end{pmatrix}. \tag{2.9}$$

Similarly we would partition B and C into blocks of a compatible size to A,

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{2.10}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{2.11}$$

so that for example

$$B_{21} = \begin{pmatrix} 1 & 1 \end{pmatrix}. \tag{2.12}$$

The key point is that multiplication still works if we pretend that those submatrices are numbers! For example, pretending like that would give the relation

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \tag{2.13}$$

which the reader should verify really is correct as matrices, i.e. the computation on the right side really does yield a matrix equal to C_{11}.

2.2.2 Important Special Case

Recall the relation

$$A \approx WH \tag{2.14}$$

in Section 1.2.2, where A is $r \times s$, W is $r \times m$ and H is $m \times s$. Partition the first and third matrices into rows, i.e. write

$$A = \begin{pmatrix} a_1 \\ ... \\ a_r \end{pmatrix}, \tag{2.15}$$

and

$$H = \begin{pmatrix} h_1 \\ ... \\ h_m \end{pmatrix}, \tag{2.16}$$

Keep W unpartitiond:

$$W = \begin{pmatrix} w_{11} & ... & w_{1m} \\ ... & ... & ... \\ w_{r1} & ... & w_{rm} \end{pmatrix}, \tag{2.17}$$

Using the partitioning idea, write WH as a "matrix-vector product":

$$WH = \begin{pmatrix} w_{11} & ... & w_{1m} \\ ... & ... & ... \\ w_{r1} & ... & w_{rm} \end{pmatrix} \begin{pmatrix} h_1 \\ ... \\ h_m \end{pmatrix} = \begin{pmatrix} w_{11}h_1 + ... + w_{1m}h_m \\ ... \\ w_{r1}h_1 + ... + w_{rm}h_m \end{pmatrix} \tag{2.18}$$

Look at that! What it says is row i of WH, and thus approximately row i of A, is a linear combination of the rows of H. And with a different partitioning, we'd find that each column of WH is a linear combination of the columns of H. We'll see in Chapter 5 that this has big implications for the matrix factorization method of RS, a topic we lay the groundwork for next.

2.3 The Notion of Approximate Rank: Principal Components Analysis

Suppose the matrix in (2.1) had been

$$M = \begin{pmatrix} 1 & 5 & 1 & -2 \\ 8.02 & 2.99 & 2 & 8.2 \\ 9 & 8 & 3 & 6 \end{pmatrix} \tag{2.19}$$

Intuitively, we still might say that the rank of M is "approximately" 2. Or better yet, row 3 still seems redundant, Let's formalize that, leading to one of the most common techniques in statistics/machine learning. By the way, this technique will also later provide a way to find W and H in (2.14).

(Warning: This section will be somewhat long, but quite central to RS/ML.)

2.3.1 Dimension Reduction

One of the major themes in computer science is *scale*, as in the common question, "Does it scale?" The concern is, does an algorithm, method or whatever work well in large-scale systems?

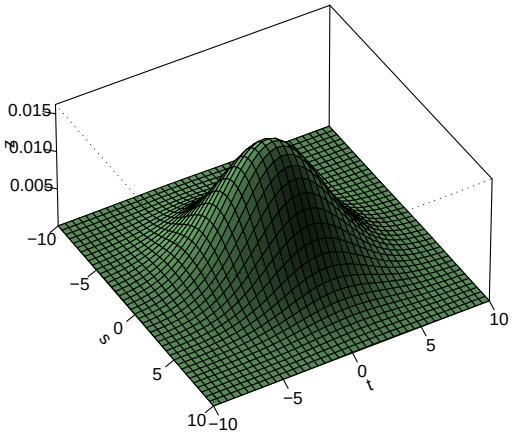
In the RS context, just think of, say, Amazon. The business has millions of users and millions of items. In other words, the ratings matrix has millions of rows and millions of columns, and even one million rows and columns would mean a total number of $(10^6)^2 = 10^{12}$ entries, about 8 terabytes of data.

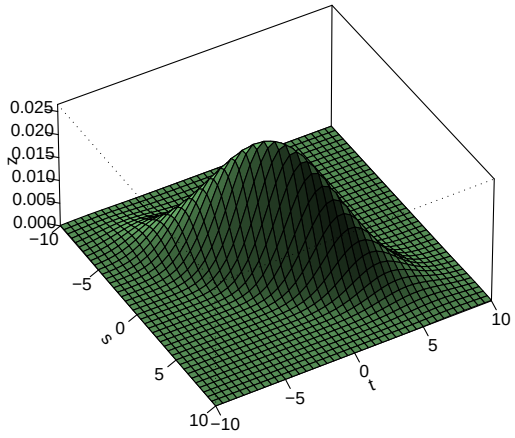
This is a core point in statistics/machine learning, the notion of *dimension reduction*. In complex applications, there is a pressing need to reduce the number of variables down to a manageable number — manageable not only in terms of computational time and space, but also the statistical problem of *overfitting* (Chapter 3).

So we need methods to eliminate redundant or near-redundant data, such as row 3 in (2.19).

2.3.2 Exploiting Correlations

Statistically, the issue is one of correlation. In (2.19), the third row is highly correlated with (the sum of) the first two rows. To explore the correlation idea further, here are two graphs of bivariate normal densities:





Let's call the two variables X_1 and X_2, with the corresponding axes in the graphs to be referred to as t_1 and t_2. The first graph was generated with a correlation of 0.2 between the two variables, while in the second one, the correlation is 0.8.

Not surprisingly due to the high correlation in the second graph, the "two-dimensional bell" is concentrated around a straight line, specifically the line $t_2 = -t_1$. In other words, there is high probability that $X_2 \approx -X_1$, so that:

> To a large extent, there is only one variable here, X_1 (or other choices, e.g. X_2), not two.

Note one more time, though, the approximate nature of the approach we are developing. There really *are* two variables above. By using only one of them, **we are relinquishing some information. But with the need to avoid overfitting, use of the approximation may be a net win for us.**

Well then, how can we determine a set of near-redundant variables, so that we can consider omitting them from our analysis? Let's look at those graphs a little more closely.

Any *level set* in the above graphs, i.e. a curve one obtains by slicing the bells parallel to the (t_1, t_2) plane can be shown to be an ellipse. As noted, the major axis of the ellipse will be the line $t_1 + t_2 = 0$. The minor axis will be the line perpendicular to that, $t_1 - t_2 = 0$. In turn, that means that standard probability methods can then be used to show that the variables

$$Y_1 = X_1 + X_2 \tag{2.20}$$

and

$$Y_2 = X_1 - X_2 \tag{2.21}$$

have 0 correlation. Then we have a good case for using only Y_1 in our data analysis, instead of using X_1 and X_2.

But why not use just X_1? As usual in statistics/ML, things get more complicated in higher dimensions. In choosing variables to retain in our analysis, it makes sense to require that they be uncorrelated, as Y_1 and Y_2 are above; if not, intuitively there is some redundancy among them, which of course is what we are hoping to avoid.

With that in mind, now suppose we have p variables, $X_1, X_2, ..., X_p$, not just two. We can no longer visualize in higher dimensions, but one can show that the level sets will be p-dimensional ellipsoids. These now have p axes rather than just two, and we can define p new variables, $Y_1, Y_2, ..., Y_p$ from the X_i, such that:

(a) The Y_i are uncorrelated.

(b) We can order them in terms of variance:

$$Var(Y_1) > Var(Y_2) > ... > Var(Y_p) \tag{2.22}$$

Now we have a promising solution to our dimension reduction problem. In [(b)] above, we can choose to use just the first few of the Y_i, omitting the ones with small variance. And again, since the Y_i will be uncorrelated, we are eliminating a source of possible redundancy among them.

PCA won't be a perfect solution — there is no such thing — as might be the case if the relations between variables is nonmonotonic. A common example is age, with mean income given age tending to be a quadratic (or higher degree) polynomial relation. But PCA is a very common "go to" method for dimension reduction, and may work well even in (mildly) nonmonotonic settings.

Now, how do we find these Y_i?

2.3.3 Eigenanalysis

Say I have a sample of n observations on two variables, $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_n$, say height and weight on $n = 100$ people. Then, formally, the *correlation* between the variables is

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{s.d.(x)\ s.d.(y)} \tag{2.23}$$

where the denominator is the product of the sample standard deviations of the two variables, and $\overline{x}$) and $\overline{y}$) are the sample means. The correlation is a number between -1 and 1.

The correlation matrix C of a set of p variables is $p \times p$, i.e. square. Moreover, since $\text{corr}(X_i, X_j) = \text{corr}(X_j, X_i)$, C is *symmetric*:

$$C' = C \tag{2.24}$$

where $'$ denotes matrix transpose.

Recall that for any square matrix L, if there is a scalar λ and a nonzero vector x such that

$$Lx = \lambda x \tag{2.25}$$

then we say that x is an *eigenvector* of L, with *eigenvalue* λ. (Note that the above implies that x is a column vector, $p \times 1$, a convention we will use throughout the book.)

It can be shown that any symmetric matrix has real (not complex) eigenvalues, and that the corresponding eigenvectors $U_1, ..., U_p$ are orthogonal,

$$U_i' U_j = 0, \ i \neq j \tag{2.26}$$

We always take the U_i to have length 1.

2.3.4 PCA

Typically we have many cases in our data, say n, arranged in an $n \times p$ matrix Q, with row i representing case i and column j representing the j^{th} variable.

Say our data is about people, 1000 of them, and we have data on height, weight, age, gender, years of schooling and income. Then $n = 1000$, $p = 6$.

So, finally, here is PCA:

1. Find the correlation matrix (or covariance matrix, a similar notion) from the data in Q. Note that since there are p, the correlation matrix will be $p \times p$.
2. Compute its eigenvalues and eigenvectors.
3. After ordering the eigenvalues from largest to smallest, let λ_i be the i^{th} largest, and let U_i be the corresponding eigenvector, scaled to have length 1.

4. Let U be the matrix whose i^{th} column is U_i. Its size will be $p \times s$.

5. Choose the first few eigenvalues, say s of them, using some criterion (see below). Denote the matrix of the first s columns of U by $U^{(s)}$. Note that U is

6. Form a new data matrix,

$$R = QU^{(s)} \tag{2.27}$$

   R will be of size $n \times s$. Column j of R is called the j^{th} principal component of the original data.

   It will be shown below that the variance of the j^{th} principal component is λ_j. The sum of all p values is the same as the sum of the variances of the original variables, an important point.

From this point onward, any data analysis we do will be with R, not Q. In R, row i is still data on the i^{th} case, e.g. the i^{th} person, but now with s new variables in place of the original p. Since typically $s << p$, we have achieved considerable dimension reduction.

2.3.5 Matrix View

Using the approach of Section 2.2, write

$$R = QU^{(s)} = Q(U_1, ..., U_s) = (QU_1, ..., QU_s) \tag{2.28}$$

where as before the U_i are the eigenvectors

So, the i^{th} column of R is QU_i. The latter quantity, again by Section 2.2, is a linear combination of the columns of Q, with the coefficients in that linear combination being the elements in U_i. Recall that each column of Q is one variable; e.g. for people, there may be an age column, a height column, a weight column and so on. Each column in R is one of our new variables. Therefore:

> The i^{th} new variable is equal to a linear combination of the old variables.

So, a new variable might be, say, 1.9 age + 0.3 height + 1.2 weight.

2.3.6 Why It Works

Recall that the new variables have the λ_i for variances and are uncorrelated. Here's why:

It is common to *center* and **scale** one's data, meaning to subtract from each variable its mean, and divide by its standard deviation. The new mean and standard deviation are 0 and 1. In that case, (2.23) becomes

$$\frac{1}{n}\sum_{i=1}^{n} x_i y_i \tag{2.29}$$

The reader should verify that with the matrices Q and R above (old and new data), their correlation matrices are $Q'Q$ and $R'R$. Let's see how that works out:

$$\begin{aligned} R'R &= (QU^{(s)})'(QU^{(s)}) && (2.30)\\ &= U^{(s)\prime}Q'\ QU^{(s)} && (2.31)\\ & && (2.32) \end{aligned}$$

using the fact that $(AB)' = B'A'$.

But remember that the U_i are eigenvectors of the correlation matrix, in this case $Q'Q$! So

$$Q'QU^{(s)} = (\lambda_1 U_1, ..., \lambda_s U_s) \tag{2.33}$$

Finally, recall that the U_i are orthogonal with length 1. Substituting (2.33) in $rrqq$, we then get

$$R'R = \text{diag}(\lambda_1, ..., \lambda_s) \tag{2.34}$$

the matrix having the λ_i on the diagonal and 0s elsewhere. The latter imply that the new variables are uncorrelated, and the former fact says that the variances of the new variables are the λ_i, as claimed.

2.3.7 Choosing the Number of Principal Components

So, how do we choose s? This is the hard part, and there is no universal good method. Typically s is chosen so that

$$\sum_{j=1}^{s} \lambda_j \tag{2.35}$$

is "most" of total variance (again, that total is the above expression with p instead of s), but even this is usually done informally.

In ML/RS settings, though, s is typically chosen by a technique called *cross validation*, to be discussed in Chapter 3.

2.3.8 Software and Example

The most commonly used R function for PCA is **prcomp()**. As with many R functions, it has many optional arguments; we'll take the default values here.

For our example, let's use the Turkish Teaching Evaluation data, available from the UC Irvine Machine Learning Data Repository. It consists of 5820 student evaluations of university instructors. Each student evaluation consists of answers to 28 questions, each calling for a rating of 1-5, plus some other variables we won't consider here.

```
> turk <- read.csv('turkiye-student-evaluation.csv',header=T)
> head(turk)
  instr class nb.repeat attendance difficulty Q1 Q2 Q3 Q4
1     1     2          1          0          4  3  3  3  3
2     1     2          1          1          3  3  3  3  3
3     1     2          1          2          4  5  5  5  5
4     1     2          1          1          3  3  3  3  3
5     1     2          1          0          1  1  1  1  1
6     1     2          1          3          3  4  4  4  4
  Q5 Q6 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15 Q16 Q17 Q18 Q19
1  3  3  3  3  3   3   3   3   3   3   3   3   3   3   3
2  3  3  3  3  3   3   3   3   3   3   3   3   3   3   3
3  5  5  5  5  5   5   5   5   5   5   5   5   5   5   5
4  3  3  3  3  3   3   3   3   3   3   3   3   3   3   3
5  1  1  1  1  1   1   1   1   1   1   1   1   1   1   1
6  4  4  4  4  4   4   4   4   4   4   4   4   4   4   4
  Q20 Q21 Q22 Q23 Q24 Q25 Q26 Q27 Q28
1   3   3   3   3   3   3   3   3   3
2   3   3   3   3   3   3   3   3   3
3   5   5   5   5   5   5   5   5   5
4   3   3   3   3   3   3   3   3   3
5   1   1   1   1   1   1   1   1   1
6   4   4   4   4   4   4   4   4   4
> tpca <- prcomp(turk[,-(1:5)]
```

Let's explore the output. First, the standard deviations of the new variables:

```
> tpca$sdev
 [1] 6.1294752 1.4366581 0.8169210 0.7663429 0.6881709
 [6] 0.6528149 0.5776757 0.5460676 0.5270327 0.4827412
[11] 0.4776421 0.4714887 0.4449105 0.4364215 0.4327540
[16] 0.4236855 0.4182859 0.4053242 0.3937768 0.3895587
[21] 0.3707312 0.3674430 0.3618074 0.3527829 0.3379096
[26] 0.3312691 0.2979928 0.2888057
> tmp <- cumsum(tpca$sdev^2)
> tmp / tmp[28]
 [1] 0.8219815 0.8671382 0.8817389 0.8945877 0.9049489
 [6] 0.9142727 0.9215737 0.9280977 0.9341747 0.9392732
[11] 0.9442646 0.9491282 0.9534589 0.9576259 0.9617232
[16] 0.9656506 0.9694785 0.9730729 0.9764653 0.9797855
[21] 0.9827925 0.9857464 0.9886104 0.9913333 0.9938314
[26] 0.9962324 0.9981752 1.0000000
```

This is striking, The first principal component (PC) already accounts for 82% of the total variance among all 28 questions. The first five PCs cover over 90%. This suggests that the designer of the evaluation survey could have written a much more concise survey instrument with almost the same utility.

Now keep in mind that each PC here is essentially a "super-question" capturing student opinion via a weighted sum of the original 28 questions. Let's look at the first two PCs' weights:

```
> tpca$rotation[,1]
        Q1         Q2         Q3         Q4         Q5
-0.1787291 -0.1869604 -0.1821853 -0.1841701 -0.1902141
        Q6         Q7         Q8         Q9        Q10
-0.1870812 -0.1878324 -0.1867865 -0.1823915 -0.1923626
       Q11        Q12        Q13        Q14        Q15
-0.1866948 -0.1862382 -0.1922729 -0.1911814 -0.1902380
       Q16        Q17        Q18        Q19        Q20
-0.1962885 -0.1808833 -0.1935788 -0.1927359 -0.1931985
       Q21        Q22        Q23        Q24        Q25
-0.1911060 -0.1908591 -0.1948393 -0.1931334 -0.1888957
       Q26        Q27        Q28
-0.1908694 -0.1897555 -0.1886699


> tpca$rotation[,2]
         Q1          Q2          Q3          Q4          Q5
 0.35645673  0.23223504  0.11551155  0.24533527  0.20717759
```

```
         Q6          Q7          Q8          Q9         Q10
 0.20075314  0.24290761  0.24901577  0.12919618  0.18911720
        Q11         Q12         Q13         Q14         Q15
 0.11051480  0.21203229 -0.10616030 -0.15629705 -0.15533847
        Q16         Q17         Q18         Q19         Q20
-0.04865706 -0.26259518 -0.12905840 -0.15363392 -0.19670071
        Q21         Q22         Q23         Q24         Q25
-0.22007368 -0.22347198 -0.10278122 -0.06210583 -0.20787213
        Q26         Q27         Q28
-0.12045026 -0.07204024 -0.21401477
```

The first PC turned out to place approximately equal weights on all 28 questions. The second PC, though, placed its heaviest weight on Q1, with substantially varying weights on the other questions.

While we are here, let's check that the columns of U are orthogonal.

```
> t(tpca$rotation[,1]) %*% tpca$rotation[,2]
              [,1]
[1,] -2.012279e-16
```

Yes, 0 (with roundoff error). As an exercise in matrix partitioning, the reader should run

```
t(tpca$rotation) %*% tpca$rotation
```

then check that it produces the identity matrix I, then ponder why this should be the case.

Chapter 3

Some Infrastructure: Probability and Statistics

Many RS methods are probabilistic in nature, so we will lay out some infrastructure. It is assumed the reader has a background in calculus-based probability structures, such as expected value and density functions. Background in statistics (as opposed to probability) and machine learning is *not* assumed.[1]

Note that while we will develop some statistical methods here, notably regression and classification models, we will not cover *inferential* statistical methods such as confidence intervals and significance tests. For readers familiar with such topics, occasional footnotes will be provided.

Other than laying some groundwork, e.g. introducing the term *dummy variables*, the primary goal of this chapter will be to discuss the issue of *overfitting*.

3.1 Data as a Sample

In statistics, the data are usually considered a sample from a population. For instance, during an election campaign pollsters will take a sample, typically of 1200 people, from the population of all voters. Say they are interested in p, the population proportion of voters favoring Candidate Jones. They calculate their estimate of p, denoted $\widehat{p}$, to be the proportion of voters in the sample who like Jones.

[1]The reader may wish to consult my open source book on probability and statistics, N. Matloff, *From Algorithms to Z-Scores: Probability and Statistical Modeling in Computer Science*, `http://heather.cs.ucdavis.edu/probstatbook`.

Sometimes the notion of sampling is merely conceptual. Say for instance we are studying hypertension, on data involving 1000 patients. We think of them as a sample from the population of all sufferers of hypertension, even though we did not go through an actual sampling process.

In RS contexts, this means that we treat the users in our dataset as a sample from the conceptual population of all potential users. We may even treat the items as a sample from a conceptual population of all possible items.

In machine learning circles, it is not customary to think explicitly in terms of populations, samples and estimates. Nevertheless, it's implied, as ML people do talk about predicting new data from the model fitted on the original data. For the model to be useful, the new data needs to come from the same source as the original — what statisticians call a population.

We will usually think in terms of sample data here.

3.2 Probability, Expected Value and Variance

We will speak in terms of a *repeatable experiment*, which again could be physical or conceptual.

We think of probability as the long-run proportion of the time some event occurs. Say we toss a fair coin. What do we mean by $P(\text{heads} = 0.5)$? Here our repeatable experiment is tossing the coin. If we were to perform that experiment many times — ideally, infinitely many times — then in the long run, 50% of the repetitions would give us heads.

Now suppose our experiment, say a game, is to keep tossing a coin until we get three consecutive heads. Let X denote the number of tosses needed. Then for instance $P(X = 4) = 0.5^4 = 0.0625$ (we get a tail then three heads). Imagine doing this experiment infinitely many times: We toss the coin until we get three consecutive heads, and record X; we toss the coin until we get three consecutive heads, and record X; we toss the coin until we get three consecutive heads, and record X; and so on. This would result in infinitely many X values. Then in the long run, 6.25% of the X values would be 4.

The *expected value* $E(X)$ of a random variable X is its long-run average value over infinitely many repetitions of the experiment. In that 3-consecutive heads game above, it can be shown that $E(X) = 14.7$. In other words, if we were to play the game infinitely many times, yielding infinitely X values, their long-run average would be 14.7.

If there is no danger of ambiguity, we usually omit the parentheses, writing EX instead of $E(X)$.

The *variance* of a random variable is a measure of its dispersion, i.e. how much it varies from one repetition to the next. It is defined as $Var(X) = E[(X - EX)^2]$.

Say we have a population of people and our our experiment is to randomly draw one person from

the population, denoting that person's height by H. Then intuitively, EH will be the mean height of all the people in the population, traditionally written as μ.

3.3 Regression Models

Regression models, both *parametric* and *nonparametric*, **form the very core of statistics and machine learning (ML)**. Their importance cannot be overemphasized.[2]

3.3.1 Definition

Suppose we are predicting a variable Y from a vector X of variables, say predicting human weight from the vector (height,age). The *regression function* at $t = (t_1, t_2)$ of Y on X is defined to be the mean weight of all people in the subpopulation consisting of all people of height t_1 and age t_2.

Let W, H and A denote weight, height and age. We write the regression function as the *conditional expectation* of W given H and A,

$$E(W \mid H = t_1, A = t_2) \tag{3.1}$$

If, say $E(W \mid H = 70, A = 28) = 162$, it means that the mean weight of all people in the subpopulation consisting of 28-year-olds of height 70 is 162.

Note that in (3.1), the expression has a different value for each (t_1, t_2) pair. So it is a function of t_1 and t_2. This is why it is called the *regression function* of W on H and A.

Terminology: It is common to refer to W here at the *response variable* and H and A as the *predictor variables*. The latter may also be called *explanatory variables* (in economics and other social sciences) or *features* (in ML).

3.3.2 Prediction

Say we have a person whose height and age are 70 and 28, but with unknown weight. It can be shown that the optimal (under a certain criterion) predictor of her weight is the value of the regression function at (70,28), $E(W \mid H = 70, A = 28) = 162$. It is optimal in the sense of minimizing expected squared prediction error.

[2]For many, the term *regression analysis* connotes a linear parametric model. But actually the term is much more general, defined to be the conditional mean as discussed below. Most ML techniques are nonparametric, as explained below, but are still regression methods.

3.3.3 Estimation

The regression function is an attribute of the population. Yet all we have is sample data. How do we estimate the regression function from our data?

3.3.3.1 Nonparametric

Intuitively, we could use a nearest-neighbor approach. To estimate $E(W \mid H = 70, A = 28)$, we could find, say, the 25 people in our sample for whom (H, A) is closest to (70,28), and average their weights to produce our estimate of $E(W \mid H = 70, A = 28)$.

This kind of approach is common in ML. The famous *random forests* method is basically a more complex form of kNN, as we will see in Chapter 7.

Statisticians also use methods like kNN. In fact, kNN and random forests were invented by statisticians. But more commonly, statistics uses *parametric* methods, as follows.

3.3.3.2 Parametric

The basic idea is to assume the regression function is linear in parameters β_i, e.g.

$$\text{mean weight} = \beta_0 + \beta_1 \text{ height} + \beta_2 \text{ age} \tag{3.2}$$

for some unknown values of the β_i.

Make sure to take careful note of the word "mean"! Clearly, the weights of individual people are not linear functions of their height and age.

As noted, the β_i are unknown, and need to be estimated from our sample data. The estimates will be denoted $\widehat{\beta}_i$. They are obtained by minimizing a certain sum of squares, to be discussed in Section 3.3.5.

By the way, if the reader is familiar with the ML methodology known as *neural networks*, she may be surprised that this technique is also parametric. Again, more in Chapter 7.

3.3.3.3 Comparison

Consider (3.2), our model for the function of t_1 and t_2

$$E(\text{weight} \mid \text{height} = t_1, \text{age} = t_2) \tag{3.3}$$

With the linear assumption (3.2), we will be estimating three quantities, the β_i. But with a nonparametric approach, we are estimating infinitely many quantities, one for each value of the (t_1, t_2) pair.

In other words, **parametric methods are a form of dimension reduction**. On the other hand, this reduction comes at the expense of relying on the assumption of linearity in (3.2). However, this is not so restrictive as it may seem, because:

- There are ways to assess the validity of the assumption. This is covered in almost any book on regression, such as mine (N. Matloff, *Statistical Regression and Classification: from Linear Models to Machine Learning*, CRC, 2017).
- One can add polynomial terms, as seen in the next section.
- Assumptions tend to be less important in prediction contexts than in estimation. In the RS context, for instance, a rough model may be fine if we wish to take into account gender in predicting ratings, but might be insufficient if we want to estimate the actual magnitude of gender effect.

3.3.4 The lm() Function in R

In R, the workhorse regression estimator is the **lm()** function. Let's apply this to the MovieLens data, predicting rating from age and gender. We'll define the latter as 1 for male, 0 for female. We find that our estimated regression function of rating on age and gender is

$$\text{estimated mean rating} = 3.3599 + 0.005311 \text{ age} - 0.0069 \text{ gender} \tag{3.4}$$

(Note the word *estimated*! These are not the true unknown population values, just estimates based on sample data.)

Actually, this shows that age and gender are pretty weak predictors of movie rating, which you will recall is on a scale of 1 to 5. A 10-year difference in age raises the predicted rating only by about 0.05! The effect of gender is small too. And while it is interesting to see that older people tend to give slighly higher ratings, as do women, we must keep in mind that the magnitude of the effect here is small.[3] Of course, the gender effect may be large in other RS datasets.

Here is the annotated code:

[3]You may be familiar with the term *statistically significant*. It is generally recognized today that this term can be quite misleading. This is beyond the scope of this book, but suffice it to say that although age and gender are statistically significant above (details available via adding the call **summary(lmout)** to the code below), their practical importance as predictors here is essentially nil. See R. Wasserstein and N. Lazar, The ASA's Statement on p-Values: Context, Process, and Purpose, *The American Statistician*, June 2016.

```
# read (user,item,rating,transID) data; name the columns
ratings <- read.table('u.data')
names(ratings) <- c('usernum','movienum','rating','transID')
# read demographic data
demog <- read.table('u.user',sep='|')
names(demog) <- c('usernum','age','gender','occ','ZIP')
# merge (database 'join' op)
u.big <- merge(ratings,demog,by.x=1,by.y=1)
u <- u.big[,c(1,2,3,5,6)]
# fit linear model
lmout <- lm(rating ~ age+gender,data=u)
```

Here's the output:

```
> lmout

Call:
lm(formula = rating ~ age + gender, data = u)

Coefficients:
(Intercept)          age       genderM
3.359894        0.005311    -0.006904
```

Let's take a closer look at that **genderM** coefficient.[4] Take for instance 28-year-old men and women; what are their mean ratings, according to this model?

```
> lmout$coef %*% c(1,28,1)
         [,1]
[1,] 3.50169
> lmout$coef %*% c(1,28,0)
          [,1]
[1,] 3.508593
```

(Note that the first '1' is needed to pick up the 3.359894.)

So, on average, 28-year-old women give ratings 3.508593 - 3.50169 = 0.006903 higher than men of that age. And except for roundoff error, that is the -0.006904 value was see in the output above.

[4]The gender variable had been coded in the data as 'M' and 'F', and R chose the first one that showed up in the data, 'M', as its base.

3.3.5 Details of Linear Regression Estimation

In the weight-height-age example, say, we form

$$r = \sum_{i=1}^{n} [W_i - (b_0 + b_1 H_i + b_2 A_i)]^2 \tag{3.5}$$

where W_i is the weight of the ith person in our sample data and so on. This is the sum of squared prediction errors. We take derivatives with respect to the b_k to minimize, then set $\widehat{\beta}_k$ to the minimizing b_k.

Though R will do the minimizing for us, it is worth having an idea how it works, especially as more practice in following matrix-centric derivations. To get a glimpse of this, we look at a matrix formulation, as follows. Let A denote the matrix of predictor values, with a 1s column tacked on at the left. In the above example, row 12, say, of A would consist of a 1, followed by the height and age of the 12^{th} person in our sample. Let D denote the vector of weights, so that D_{12} is the weight of the 12^{th} person in our sample. Finally, let b denote the vector of the b_k. Say we have data on 100 people. Then A will have 100 rows, and D will have length 100.

Use the above as a concrete guide to your thinking, but keep in mind the general case: If we have p predictors and n data points, then A and D will have sizes $n \times (p+1)$ and n

Then

$$r = (D - Ab)'(D - Ab) \tag{3.6}$$

(Write it out to see this. Doing so will be crucial to understanding the material below and many points in the rest of the book.)

Write the *gradient* of r with respect to b,

$$\frac{\partial r}{\partial b} = (\frac{\partial r}{\partial b_0}, \frac{\partial r}{\partial b_1}, ..., \frac{\partial r}{\partial b_p})' \tag{3.7}$$

where $p+1$ is the number of predictor variables.[5]

It can be shown that for a vector u,

$$\frac{\partial u'u}{\partial u} = 2u \tag{3.8}$$

[5]Note the representation here of a column vector as the transpose of a row vector. We will often do this, in order to save space on the page. And, any reference to a *vector* will be to a column vector unless stated otherwise.

(analogous to the scalar relations $d(u^2)/du = 2u$; again, this is seen by writing the expressions out).

Setting $u = D - Ab$ and applying the Chain Rule (adapted for gradients), we get

$$\frac{\partial r}{\partial b} = \frac{\partial r}{\partial u}\frac{\partial u}{\partial b} = 2(D - Ab)\ \frac{\partial (D - Ab)}{\partial b} = 2(-A')(D - Ab) \tag{3.9}$$

Setting the gradient to 0 and solving for b, we have

$$0 = A'D - A'Ab \tag{3.10}$$

so that the minimizing b, giving us $\widehat{\beta}$, is

$$b = (A'A)^{-1}A'D \tag{3.11}$$

This famous formula is what **lm()** computes in finding the $\widehat{\beta}_k$.

Note that we cannot simply multiply both sides of (3.10) by $(A')^{-1}$, as A' is nonsquare and thus noninvertible.

Note too that in our age/gender MovieLens example above, we should not have variables for both male and female. If we did, we have

$$A = \begin{pmatrix} 1 & A_1 & M_1 & F_1 \\ 1 & A_2 & M_2 & F_2 \\ ... & & & \\ 1 & A_{100000} & M_{100000} & F_{100000} \end{pmatrix} \tag{3.12}$$

where A_i is the age of the i^{th} person in our data, and one of M_i and F_i is 1 and the other 0, according to the gender of this person. (Recall that there are 100000 data points in tis dataset.) The problem is this: The third and fourth columns of A would then sum to a vector of all 1s, the same as in the first column. So the columns of A will be linearly dependent, and the rank will be 3 instead of 4. The same will then be true for $A'A$, so that $(A'A)^{-1}$ will not exist in (3.11).

In other words, not only would the F column be unnecessary, it would be prolematic.

3.3.5.1 Polynomial Terms

People tend to gain weight during middle age, but often they lose weight when they become elderly. So (3.2), which is linear in the age variable, may be rather unrealistic; we might believe a quadratic

model for mean weight as a function of age is better:

$$\text{mean weight} = \beta_0 + \beta_1 \text{ height} + \beta_2 \text{ age} + \beta_3 \text{ age}^2 \tag{3.13}$$

A key point is that this is still a linear model! When we speak of a linear model — the 'l' in "lm()" – we mean linear in the β_i. If in (3.13) we, say, multiply all the β_i by 3, the entire sum grows by a factor of 3, hence the linearity in the β_i.

Of course we may wish to add a quadratic term for height as well, and for that matter, a product term height × age. And since any model is merely an approximation, we might consider using higher and higher order polynomials. We do have to worry about overfitting though; see Section 3.4.

We'll have a long example in Section 3.5.

3.3.6 Categorical Variables (Predictor, Response)

A *categorical* variable is one that codes categories. In our RS context, for instance, a user's postal code — ZIP Code, in the US — may be a helpful predictor of a rating. Yet it cannot be treated in **lm()**, say as a numeric variable. After all, the ZIP Code 90024, for example, is not "twice as good" as 45002; they are just ID codes.

3.3.6.1 Dummy Variables as Predictors

So, what do we do if we wish to use a categorical variable as a predictor? The answer is that we break the variable into a set of *indicator variables*, colloquially known as *dummy variables*. These have the values 1 and 0, with 1 indicating the trait in question, 0 meaning not.

Say for instance in RS we have the categorical variable State for users in a US state. We would define 50 dummy variables, one for each state. For instance, the one for California would have the value 1 if the user lives in California, 0 otherwise.

Note carefully though that we would only use 49 of the dummies, not 50. We could for instance exclude Wyoming. Why? Because if the other 49 dummies are all 0, then we know this user must be in Wyoming. The Wyoming dummy would then be redundant. Not only do we want to avoid redundancy on dimension reduction grounds, but also that redundancy would result in the matrix A in (3.11) being less than full rank, so $(A'A)^{-1}$ would not exist.

Categorical variables in R:

In R, categorical variables are stored as objects of the class **'factor'**. The latter is a class designed

specifically for representing categorical data. It consists of integer codes for the categories, with character-string names. The various categories are called *levels*.

For example, say we are dealing with eye color, and that the only colors are brown, blue and green, and have data on four people:

```
> s <- c('brown','blue','green','blue')
> class(s)
[1] "character"
> sf <- as.factor(s)
> levels(sf)
[1] "blue"  "brown" "green"
> sf[2]
[1] blue
Levels: blue brown green
> as.numeric(sf)
[1] 2 1 3 1
```

In our MovieLens example above, let's take a look at the data frame **demog**:

```
> for (i in 1:5)
+    print(class(demog[,i]))
[1] "integer"
[1] "integer"
[1] "factor"
[1] "factor"
[1] "factor"
```

The last three columns are factors.[6] Let's see how many occupations there are:

```
> levels(demog$occ)
[1] "administrator" "artist"        "doctor"        "educator"
[5] "engineer"      "entertainment" "executive"     "healthcare"
[9] "homemaker"     "lawyer"        "librarian"     "marketing"
[13] "none"          "other"         "programmer"    "retired"
[17] "salesman"      "scientist"     "student"       "technician"
[21] "writer"
```

In a regression application, we'd form 21 dummies, but use only 20 of them (any 20).

However, the designers of R (and its predecessor S), in their wisdom, set things up to save us some time and trouble. We can just specify the factors we wish to use, and R will form the dummies for us, being careful to drop one of them.

[6] Even the first two could have been stored as factors, but were not coded as so.

This is what happened, for instance, in our example above in which we regressed rating against age and gender, with output

```
Coefficients:
(Intercept)           age       genderM
3.359894         0.005311     -0.006904
```

R noticed that gender was an R factor, with levels M and F. It created dummies for M and F, but just retained the former, as the first case in the data had gender as M.

So, the estimated coefficient -0.006904 meant that the "maleness" impact on mean rating has that value. Men give slightly lower ratings than women do, for fixed age.

In this manner, the estimated regression coefficient of a dummy variable is the effect, all other predictors fixed, of this trait *relative to the excluded trai*, say femaleness above.

3.3.6.2 Interaction Terms

Say we are in some RS context in which age and gender are substantial factors in predicting rating. Suppose also that we suspect men become more liberal raters as they age while women become more reserved in their ratings. Then a model like this might work well:

$$\text{mean rating} = \beta_0 + \beta_1 \text{ age} + \beta_2 \text{ male} + \beta_3 \text{ age} \times \text{male} \tag{3.14}$$

where *male* is a dummy variable. To see why this might be appropriate, consider what the above equation reduces to for men and women:

men:

$$\text{mean rating} = (\beta_0 + \beta_2) + (\beta_1 + \beta_3) \text{ age} \tag{3.15}$$

*women:*M

$$\text{mean rating} = \beta_0 + \beta_1 \text{ age} \tag{3.16}$$

So the male and female lines have different slopes (and different intercepts), allowing for the differential age effect we surmise. Of course, once we compute the $\widehat{\beta_i}$ from the data, it may well turn out that our differential aging trends may not be confirmed.[7]

[7]One must take sampling variability into account, say by forming confidence intervals for the β_i. As noted earlier, do not use significance testing for this. At any rate, these aspects are beyond the scope of this book.

The term age × male is called an *interaction term*. Note that interaction terms can be formed from any predictor, not just dummy variables. Also, one can form triple products for three-way interactions and so on, though this could greatly increase the complexity of the model and thus risk overfitting.

3.3.6.3 Dummy Variables as Response Variables

In many cases, the response variable may be categorical. In the RS context, for instance, a rating may simply be binary, i.e. like/dislike. Or even click/not click — does a user click on a Web page location? Let's use this as our example.

We are generally interested in the probability of a click. That actually fits a regression context, as follows. Code a click as 1 and nonclick as 0. Since the expected value of a variable of this type is the probability of a 1, and since a regression function by definition is an expected value, taking Click as our response variables does involve a regression function.

So, if our predictors were age and gender, say, we might entertain formulating our regression model as

$$\text{probability of click} = \beta_0 + \beta_1 \text{ age} + \beta_2 \text{ gender} \tag{3.17}$$

One problem, though, is that a probability should be in [0,1] yet the right-hand side of (3.17) can conceivably be anywhere in $(-\infty, \infty)$. For this and other reasons the usual parametric model for a binary response Y is the *logistic*: For p predictors X_i, our model is

$$P(Y = 1 \mid X_1 = t_1, ..., X_p = t_p) = \frac{1}{1 + \exp -(\beta_0 + \beta_1 t_1 + ... + \beta_p t_p)} \tag{3.18}$$

This is called a *generalized linear model*, as it has the linear form $\beta_0 + \beta_1 t_1 + ... + \beta_p t_p$ embedded inside another function, in this case the logistic function $g(s) = 1/(1 + e^{-s})$.

Note that the latter function, often called *logit* for short, has values only in [0,1], as desired, and is increasing in s, thus retaining the monotonic notion of linear models.[8]

The β_i are estimated by an R function **glm()**, similar to **lm()**.[9] Let's model a user giving a movie a rating of 4 or higher:

```
> r45 <- as.integer(u$rating >= 4)  # a binary value, 1 or 0
```

[8] These properties form the intuitive motivation for using logit models. Another motivation is this: Let X denote the vector of predictor variables, and let Y be the response variable, with the two classes 0 and 1. If within each class, X has a multivariate normal distribution, with the same covariance matrix in each class.

[9] The class of the return value is **'glm'**, which is a subclass of **'lm'.**

```
> u$r45 <- r45
> glmout <- glm(r45 ~ age+gender,data=u,family=binomial)
> glmout

Call:  glm(formula = r45 ~ age + gender, data = u)

Coefficients:
(Intercept)           age       genderM
-0.002510        0.006886     -0.011189
...
```

The argument **family = binomial** tells R that we want the logistic model, not some other generalized linear model, such a model known as *Poisson regression*.[10]

3.3.7 R's predict(), a Generic Function

A key aspect to R's object orientation is *generic* functions. Take **plot()**, for instance. Its action will depend on the class of object it is applied to. If we call the function on a vector, we get a histogram. But if we call it on a two-column matrix, we get a scatter diagram.

What happens is that when **plot()** is called, R will check what class of object the caller supplied as an argument. If the object is of class **"x"**, then the original call will be *dispatched* to **plot.x()**, a plotting function tailored to that class. (Of course, that means one needs to have been written and available.)

R's **predict()** is another example of a generic function, used to predict new cases. In the MovieLens example above, say we want to predict the rating given by a 30-year-old man. We could simply plug 30 and 1 into the estimated regression function, say using **coef()** to get the $\widehat{\beta}_i$:

```
> coef(lmout)
(Intercept)           age       genderM
3.359894442  0.005310673 -0.006903502
> coef(lmout) %*% c(1,30,1)  # linear algebra-style matrix multiply
       [,1]
[1,] 3.512311
```

Alternatively (and in many settings, more conveniently):

```
> newdata <- data.frame(age=30,gender='M')
> predict(lmout,newdata)
```

[10] By the way, the argument **family** must be an object of class **'function'**. Inside **glm()**, there will be a call **family()**. R has a built-in function **binomial()**, which is called here.

```
    1
3.512311
```

Recall that we had assigned the output of **lm()** to **lmout**, which will have class **'lm'**. So, the call to **predict()** above was dispatched to **predict.lm()**.

What about **glm()**? There is a function **predict.glm()**, which normally should be called with the argument **type = 'response'**. The latter means we want the return values to be the estimated values of the regression function, i.e. the conditional probabilities of response 1, given the values of the predictors.

3.3.8 More Than Two Levels in Categorical Response

What if our response variable is categorical but with more than two levels? In the click/nonclick setting, suppose the user has a choice of five things to click, and must choose one. Then the response is categorical with five levels.

There are two major approaches. To explain, we'll use the following very simple example. Say there are dogs, cats and foxes on a field, and they sometimes step on a sensor, so we know their weights but do not see them. Say we have data on 10000 data points, in which we do know the species. Our data frame, **df**, has 10000 rows and 4 columns. In the columns, say the names are 'Weight', 'Dog', 'Cat' and 'Fox', with the last three being dummies. Say we have 5000 dogs, 2000 cats and 3000 foxes. Then for instance 2000 of the rows in **df** would be cats.

One-vs.All (OVA) Method

One would run run three logistic models:

```
gdog <- glm(Dog ~ .,data=df[,1:2])  # dog vs. all else
gcat <- glm(Cat ~ .,data=df[,1:3])  # cat vs. all else
gdog <- glm(Fox ~ .,data=df[,1:4])  # fox vs. all else
```

Then for each new animal we encounter of unknown species, we call **predict()** three times, yielding three estimated conditional probabilities. If the one for cat, say, is largest, we guess Cat.

All vs. All (AVA) Method

Here again we'd run multiple logit models, in pairs as follows:

```
gdogcat <-
   glm(Dog ~ .,data=df[df$dog+df$cat==1,1:2])  # dog vs. cat
gdogfox <-
   glm(Dog ~ .,data=df[df$dog+df$fox==1,1:2])  # dog vs. fox
gcatfox <-
```

```
glm(Cat ∼ .,data=df[df$cat+df$fox==1,1:3])  # cat vs. fox
```

Then for each new animal we encounter of unknown species, we call **predict()** three times, again yielding three estimated conditional probabilities. Say in the first one, Cat "wins," i.e. the conditional probability is less than 0.5. Say Dog wins in the second, and Cat wins in the third. Since Cat had the most wins, we predict Cat.

Comparison

At first, OVA seems much better than AVA. If we have m levels, that means running $C(m,2) = O(m^2)$ parwise logit models, rather than m for OVA. However, that is somewhat compensated by the fact that each pairwise model will be based on less data, and some analysts contend that AVA can have better accuracy. It remains a bit of a controversy.

3.4 Bias, Variance, Overfitting and p-Hacking

By far the most vexing issue in statistics and machine learning is that of *overfitting*.

3.4.1 What Is Overfitting?

Suppose we have just one predictor, and n data points. If we fit a polynomial model of degree $n-1$, the resulting curve will pass through all n points, a "perfect" fit. For instance:

```
> x <- rnorm(6)
> y <- rnorm(6) # unrelated to x!
> df <- data.frame(x,y)
> df$x2 <- x^2
> df$x3 <- x^3
> df$x4 <- x^4
> df$x5 <- x^5
> df
           x           y          x2           x3
1 -1.1855131  0.2881291 1.40544120 -1.666168894
2 -1.7838769 -2.0741740 3.18221664 -5.676682627
3 -0.7124510 -0.4253678 0.50758640 -0.361630431
4  0.1676111 -0.1949265 0.02809348  0.004708779
5  1.2462926 -0.7348481 1.55324535  1.935798245
6  0.3741604  1.9521667 0.13999601  0.052380963
             x4           x5
1 1.975265e+00  -2.341702414
```

```
2 1.012650e+01 -18.064433938
3 2.576440e-01  -0.183558689
4 7.892437e-04   0.000132286
5 2.412571e+00   3.006769615
6 1.959888e-02   0.007333126
> lmo <- lm(y ~ .,data=df)
> lmo

Call:
lm(formula = y ~ ., data = df)

Coefficients:
(Intercept)            x           x2           x3
    -1.3127       4.7632      11.4809       0.5781
         x4           x5
    -6.9685      -2.4938
> lmo$fitted.values
         1          2          3          4          5
 0.2881291 -2.0741740 -0.4253678 -0.1949265 -0.7348481
         6
 1.9521667
> y
[1]  0.2881291 -2.0741740 -0.4253678 -0.1949265 -0.7348481
[6]  1.9521667
```

Yes, we "predicted" **y** perfectly, **even though there was no relation between the response and predictor variables).** Clearly that "perfect fit" is illusory, "noise fitting." Our ability to predict future cases would not be good. This is *overfitting*.

Let's take a closer look, in an RS context. Say we believe (3.14) is a good model for the setting described in that section, i.e. men becoming more liberal raters as they age but women becoming more conservative. If we omit the interaction term, than we will underpredict older men and overpredict older women. This biases our ratings.

On the other hand, we need to worry about sampling variance. Consider the case of opinion polls during an election campaign, in which the goal is to estimate p, the proportion of voters who will vote for Candidate Jones. If we use too small a sample size, say 50, our results will probably be inaccurate. This is due to sampling instability: Two pollsters, each randomly sampling 50 people, will sample different sets of people, thus each having different values of $\widehat{p}$, their sample estimates of p. For a sample of size 50, it is likely that their two values of $\widehat{p}$ will be substantially different from each other, whereas if the sample size were 5000, the two estimates would likely be close to

each other. In other words, the variance of $\widehat{p}$ is too high if the sample size is just 50.[11]

In a parametric regression setting, increasing the number of terms roughly means that the sampling variance of the $\widehat{\beta}_i$ will increase.

So we have the famous *bias/variance tradeoff*: As we use more and more terms in our regression model (predictors, polynomials, interaction terms), the bias decreases but the variance increases. This "tug of war" between these decreasing and increasing quantities typically yields a U-shaped curve: As we increase the number of terms from 1, mean absolute prediction error will at first decrease but eventually will increase. Once we get to the point at which it increases, we are *overfitting*.

This is particularly a problem when one has many dummy variables. For instance, there are more than 42,000 ZIP Codes in the US; to have a dummy for each would almost certainly be overfitting.

3.4.2 Can Anything Be Done about It?

So, where is the "happy medium," the model that is rich enough to capture most of the dynamics of the variables at hand, but simple enough to avoid variance issues? Unfortunately, **there is no good answer to this question.**

One quick rule of thumb is that one should have $p < \sqrt{n}$, where p is the number of predictors, including polynomial and interaction terms (not to be confused with the quantity of the same name in our polling example above), and n is the number of cases in our sample. But this is certainly not a firm rule by any means. One may be able to get good prediction with a considerably larger value of p, especially if *regularization* methods are used (Chapter 5).

The most common approach to dealing with the bias/variance tradeoff is *cross validation*. In the simplest version, one randomly splits the data into a *training set* and a *test set*.[12] We fit the model to the training set and then, pretending we don't know the "Y" (i.e. response) values in the test set, predict those values from our fitted model and the "X" values (i.e. the predictors) in the test set. We then "unpretend," and check how well those predictions worked.

The test set is "fresh, new" data, since we called **lm()** or whatever only on the training set. Thus we are avoiding the "noise fitting" problem. We can try several candidate models, then choose the one that best predicts the test data.

(Note carefully that after fitting the model via cross-validation, we then use the full data for later prediction. Splitting the data for cross-validation was just a temporary device for model selection.)

[11] The repeatable experiment here is randomly choosing 50 people. Each time we perform this experiment, we get a different set of 50 people, thus a different value of $\widehat{p}$. The latter is a random variable, and thus has a variance.

[12] The latter is also called a *holdout set* or a *validation set*.

Cross-validation is essentially the standard for model selection, and it works well if we only try a few models. Problems can occur if we try many models, as seen in the next section.

3.4.3 The Problem of P-hacking

The (rather recent) term *p-hacking* refers to the following abuse of statistics.[13]

Say we have 250 pennies, and we wish to determine whether any are unbalanced, i.e. have probability of heads different from 0.5. We do so by tossing each coin 100 times. If we get fewer than 40 heads or more than 60, we decide this coin is unbalanced.[14] The problem is that, even if all the coins are perfectly balanced, we eventually will have one that has fewer than 40 or greater than 60 heads, just by accident. **We will then falsely declare this coin to be unbalanced.**

Or, to give a somewhat frivolous example that still will make the point, say we are investigating whether there is any genetic component to a person's sense of humor. Is there a Humor gene? There are many, many genes to consider. Testing each one for relation to sense of humor is like checking each penny for being unbalanced: Even if there is no Humor gene, then eventually, just by accident, we'll stumble upon one that seems to be related to humor.[15]

Though the above is not about prediction, it has big implications for the prediction realm. In ML there are various datasets on which analysts engage in contests, vying for the honor of developing the model with the highest prediction accuracy, say for classification of images. If there is a large number of analysts competing for the prize, then even if all the analysts have models of equal accuracy, it is likely that one is substantially higher than the others, just due to an accident of sampling variation. This is true in spite of the fact that they all are using the same sample; it may be that the "winning" analyst' model happens to do especially well in the given data, and may not be so good on another sample from the same population. So, when some researcher sets a new record on a famous ML dataset, it may be that the research really has found a better prediction model — or it may be that it merely looks better, due to p-hacking.

The same is true for your own analyses. If you try a large number of models, the "winning" one may actually not be better than all the others.

[13] The term *abuse* here will not necessarily connote intent. It may occur out of ignorance of the problem.

[14] For those who know statistics: This gives us a Type I error rate of about 0.05, the standard used by most people.

[15] For those with background in statistics, the reason this is called "p-hacking" is that the researcher may form a significance test for each gene, computing a p-value for each test. Since under the null hypothesis we have a 5% chance of getting a "significant" p-value for any given gene, the probability of having at least one significant result out of the thousands of tests is quite high, even if the null hypothesis is true in all cases. There are techniques called *multiple inference* or *multiple comparison* methods, to avoid p-hacking in performing statistical inference. See for example *Multiple Comparisons: Theory and Methods*, Jason Hsu, 1996, CRC.

3.5 Extended Example

Let's illustrate this on the dataset **prgeng**, assembled from the 2000 US census. It consists of wage and other information on programmers and engineers in Silicon Valley. This dataset is included in the R **polyreg** package, which fits polynomial models as we saw in Section 3.3.5.1 above.[16]

```
> getPE()  # produces data frame pe
> pe1 <- pe[,c(1,2,4,6,7,12:16,3)]  # choose some predictors
> head(pe1)
     age sex wkswrkd ms phd occ1 occ2 occ3 occ4 occ5
1 50.30082   0      52  0   0    0    0    1    0    0
2 41.10139   1      20  0   0    0    1    0    0    0
3 24.67374   0      52  0   0    0    0    1    0    0
4 50.19951   1      52  0   0    1    0    0    0    0
5 51.18112   0       1  0   0    1    0    0    0    0
6 57.70413   1       0  0   0    1    0    0    0    0
wageinc
1   75000
2   12300
3   15400
4       0
5     160
6       0
```

By the way, note the dummy variables. We have just two levels for education, so anyone with just a bachelor's degree or less, or a professional degree, with be "other," coded by **ms** and **phd** both having the values 0. Similarly, there are actually six occupations, hence five dummies; the sixth occupation is coded by having 0s for the five occupation dummies.

Well, then, let's see how well we can predict wage income. First, split the data:

```
> set.seed(9999)
> testidxs <- sample(1:nrow(pe1),1000)
> testset <- pe1[testidxs,]
> trainset <- pe1[-testidxs,]
```

Then fit the model and predict:

```
> lmout <- lm(wageinc ~ .,data=pe1)  # predict wage from all others
> predvals <- predict(lmout,testset[,-11])
> mean(abs(predvals - testset[,11]))
```

[16] Available from `github.com/matloff`.

```
[1] 26494.62
```

An average, our prediction is off by more than $26,000, not so good, but not the main point here.

Let's try a quadratic model. Again, we could form the squared terms ourselves, but **polyreg** makes it convenient.

```
> pfout <- polyFit(pe1,deg=2)
> mean(abs(predict(pfout,testset[,-11]) - testset[,11]))
[1] 25885.43
```

Ah, some improvement, with a second-degree model. Note that this includes the squares of all predictors, and their products, e.g. age times weeks worked. Altogether there are now $p = 46$ predictors, up from our original 10:

```
> dim(pfout$poly.xy)
[1] 20090    47
```

How about a cubic model?

```
> pfout <- polyFit(pe1,deg=3)
> mean(abs(predict(pfout,testset[,-11]) - testset[,11]))
[1] 25410.9
```

Even better, though it may be largely a difference due to sampling variation. We are now up to $p = 118$.

```
> pfout <- polyFit(pe1,deg=4)
getPoly time:  1.12 0.056 1.177 0 0
lm() time:  0.88 0.024 0.906 0 0
> mean(abs(predict(pfout,testset[,-11]) - testset[,11]))
[1] 25380.65
Warning message:
In predict.lm(object$fit, plm.newdata) :
  prediction from a rank-deficient fit may be misleading
```

About the same, but with an ominous warning. R found that the matrix $A'A$ in (3.11) was close to nonfull-rank, thus nearly singular (noninvertible). Now $p = 226$.

Remarkably, this improvement continued:

deg	MAPE
6	25245.83
10	24679.86
15	24123.59

And at degree 15, p was a whopping 3791. By the way, in that last computation, it took more than 2 minutes to simply generate the polynomials — those 3791 columns — and almost 20 minutes to process the call to **lm()**.

Yet, even with this large model, we have not yet reached the point of overfitting. Eventually, though, MAPE will begin to rise.

Chapter 4

Nearest-Neighbor Methods

SHOW THE CODE FOR USING IT IN OUR 3 METHODS

GIVE A SIMPLE MATH DERIVATION OF VALUE OF IT, E.G. FOR SMALL N-I

Chapter 5

Matrix Factorization Methods

SHOW THE CODE FOR USING IT IN OUR 3 METHODS

GIVE A SIMPLE MATH DERIVATION OF VALUE OF IT, E.G. FOR SMALL N-I

Chapter 6

Statistical Models

SHOW THE CODE FOR USING IT IN OUR 3 METHODS

GIVE A SIMPLE MATH DERIVATION OF VALUE OF IT, E.G. FOR SMALL N-I

Chapter 7

Use of Side Information

SHOW THE CODE FOR USING IT IN OUR 3 METHODS

GIVE A SIMPLE MATH DERIVATION OF VALUE OF IT, E.G. FOR SMALL N-I

Chapter 8

Bias, Variance and Overfitting

GIVE EX FROM 132.TEX ON DENSITY ESTIMATION AT A POINT OF RISING DENSITY, MAYBE AT ENDPOINT OF SUPPORT, SO HAVE A CLEAR BIAS; MATH DERIVATION

APPLY TO K IN KNN, RANK OF MATRIX, FIXED- VS. RANDOM-EFFECTS IN STAT MODEL, VALUE OR NOT TO USING COVARS

Appendix A

R Quick Start

Here we present a quick introduction to the R data/statistical programming language. Further learning resources are listed at `http://heather.cs.ucdavis.edu//r.html`.

R syntax is similar to that of C. It is object-oriented (in the sense of encapsulation, polymorphism and everything being an object) and is a functional language (i.e. almost no side effects, every action is a function call, etc.).

A.1 Correspondences

aspect	C/C++	R
assignment	=	<- (or =)
array terminology	array	vector, matrix, array
subscripts	start at 0	start at 1
array notation	m[2][3]	m[2,3]
2-D array storage	row-major order	column-major order
mixed container	struct, members accessed by .	list, members acessed by $ or [[ ]]
return mechanism	return	return() or last value computed
primitive types	int, float, double, char, bool	integer, float, double, character, logical
logical values	true, false	TRUE, FALSE (abbreviated T, F)
mechanism for combining modules	include, link	library()
run method	batch	interactive, batch
comment symbol	//	#

A.2 Starting R

To invoke R, just type "R" into a terminal window, e.g. **xterm** in Linux or Macs, **CMD** in Windows.

If you prefer to run from an IDE, you may wish to consider ESS for Emacs, StatET for Eclipse or RStudio, all open source. ESS is the favorite among the "hard core coder" types, while the colorful, easy-to-use, RStudio is a big general crowd pleaser. If you are already an Eclipse user, StatET will be just what you need.[1]

R is normally run in interactive mode, with > as the prompt. Among other things, that makes it easy to try little experiments to learn from; remember my slogan, "When in doubt, try it out!" For batch work, use **Rscript**, which is in the R package.

A.3 First Sample Programming Session

Below is a commented R session, to introduce the concepts. I had a text editor open in another window, constantly changing my code, then loading it via R's **source()** command. The original contents of the file **odd.R** were:

```
1 oddcount <- function(x)  {
2    k <- 0  # assign 0 to k
3    for (n in x)  {
4       if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
5    }
6    return(k)
7 }
```

By the way, we could have written that last statement as simply

```
1    k
```

because the last computed value of an R function is returned automatically. This is actually preferred style in the R community.

The R session is shown below. You may wish to type it yourself as you go along, trying little experiments of your own along the way.[2]

[1] I personally use **vim**, as I want to have the same text editor no matter what kind of work I am doing. But I have my own macros to help with R work.

[2] The source code for this file is at `http://heather.cs.ucdavis.edu/~matloff/MiscPLN/R5MinIntro.tex`. You can download the file, and copy/paste the text from there.

```
> source("odd.R")  # load code from the given file
> ls()  # what objects do we have?
[1] "oddcount"
> # what kind of object is oddcount (well, we already know)?
> class(oddcount)
[1] "function"
> # while in interactive mode, and not inside a function, can print
> # any object by typing its name; otherwise use print(), e.g. print(x+y)
> oddcount  # a function is an object, so can print it
function(x)  {
   k <- 0  # assign 0 to k
   for (n in x)  {
      if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
   }
   return(k)
}

> # let's test oddcount(), but look at some properties of vectors first
> y <- c(5,12,13,8,88)  # c() is the concatenate function
> y
[1]  5 12 13  8 88
> y[2]  # R subscripts begin at 1, not 0
[1] 12
> y[2:4]  # extract elements 2, 3 and 4 of y
[1] 12 13  8
> y[c(1,3:5)]  # elements 1, 3, 4 and 5
[1]  5 13  8 88
> oddcount(y)  # should report 2 odd numbers
[1] 2

> # change code (in the other window) to vectorize the count operation,
> # for much faster execution
> source("odd.R")
> oddcount
function(x)  {
   x1 <- (x %% 2 == 1)  # x1 now a vector of TRUEs and FALSEs
   x2 <- x[x1]  # x2 now has the elements of x that were TRUE in x1
   return(length(x2))
}

```

```
> # try it on subset of y, elements 2 through 3
> oddcount(y[2:3])
[1] 1
> # try it on subset of y, elements 2, 4 and 5
> oddcount(y[c(2,4,5)])
[1] 0

> # further compactify the code
> source("odd.R")
> oddcount
function(x)  {
   length(x[x %% 2 == 1])  # last value computed is auto returned
}
> oddcount(y)  # test it
[1] 2

# and even more compactification, making use of the fact that TRUE and
# FALSE are treated as 1 and 0
> oddcount <- function(x) sum(x %% 2 == 1)
# make sure you understand the steps that that involves:  x is a vector,
# and thus x %% 2 is a new vector, the result of applying the mod 2
# operation to every element of x; then x %% 2 == 1 applies the == 1
# operation to each element of that result, yielding a new vector of TRUE
# and FALSE values; sum() then adds them (as 1s and 0s)

# we can also determine which elements are odd
> which(y %% 2 == 1)
[1] 1 3
```

Note that the function of the R function **function()** is to produce functions! Thus assignment is used. For example, here is what **odd.R** looked like at the end of the above session:

```
oddcount <- function(x)  {
   x1 <- x[x %% 2 == 1]
   return(list(odds=x1, numodds=length(x1)))
}
```

We created some code, and then used **function()** to create a function object, which we assigned to **oddcount**.

A.4 Vectorization

Note that we eventually **vectorized** our function **oddcount()**. This means taking advantage of the vector-based, functional language nature of R, exploiting R's built-in functions instead of loops. This changes the venue from interpreted R to C level, with a potentially large increase in speed. For example:

```
> x <- runif(1000000)  # 1000000 random numbers from the interval (0,1)
> system.time(sum(x))
   user  system elapsed
  0.008   0.000   0.006
> system.time({s <- 0; for (i in 1:1000000) s <- s + x[i]})
   user  system elapsed
  2.776   0.004   2.859
```

A.5 Second Sample Programming Session

A matrix is a special case of a vector, with added class attributes, the numbers of rows and columns.

```
> # "rowbind() function combines rows of matrices; there's a cbind() too
> m1 <- rbind(1:2,c(5,8))
> m1
     [,1] [,2]
[1,]    1    2
[2,]    5    8
> rbind(m1,c(6,-1))
     [,1] [,2]
[1,]    1    2
[2,]    5    8
[3,]    6   -1

> # form matrix from 1,2,3,4,5,6, in 2 rows; R uses column-major storage
> m2 <- matrix(1:6,nrow=2)
> m2
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> ncol(m2)
[1] 3
> nrow(m2)
```

```
[1] 2
> m2[2,3]  # extract element in row 2, col 3
[1] 6
# get submatrix of m2, cols 2 and 3, any row
> m3 <- m2[,2:3]
> m3
     [,1] [,2]
[1,]    3    5
[2,]    4    6

> m1 * m3  # elementwise multiplication
     [,1] [,2]
[1,]    3   10
[2,]   20   48
> 2.5 * m3  # scalar multiplication (but see below)
     [,1] [,2]
[1,]  7.5 12.5
[2,] 10.0 15.0
> m1 %*% m3  # linear algebra matrix multiplication
     [,1] [,2]
[1,]   11   17
[2,]   47   73

> # matrices are special cases of vectors, so can treat them as vectors
> sum(m1)
[1] 16
> ifelse(m2 %%3 == 1,0,m2) # (see below)
     [,1] [,2] [,3]
[1,]    0    3    5
[2,]    2    0    6
```

A.6 Recycling

The "scalar multiplication" above is not quite what you may think, even though the result may be. Here's why:

In R, scalars don't really exist; they are just one-element vectors. However, R usually uses **recycling**, i.e. replication, to make vector sizes match. In the example above in which we evaluated

the express `2.5 * m3`, the number 2.5 was recycled to the matrix

$$\begin{pmatrix} 2.5 & 2.5 \\ 2.5 & 2.5 \end{pmatrix} \tag{A.1}$$

in order to conform with **m3** for (elementwise) multiplication.

A.7 More on Vectorization

The **ifelse()** function is another example of vectorization. Its call has the form

```
ifelse(boolean vectorexpression1, vectorexpression2, vectorexpression3)
```

All three vector expressions must be the same length, though R will lengthen some via recycling. The action will be to return a vector of the same length (and if matrices are involved, then the result also has the same shape). Each element of the result will be set to its corresponding element in **vectorexpression2** or **vectorexpression3**, depending on whether the corresponding element in **vectorexpression1** is TRUE or FALSE.

In our example above,

```
> ifelse(m2 %%3 == 1,0,m2) # (see below)
```

the expression `m2 %%3 == 1` evaluated to the boolean matrix

$$\begin{pmatrix} T & F & F \\ F & T & F \end{pmatrix} \tag{A.2}$$

(TRUE and FALSE may be abbreviated to T and F.)

The 0 was recycled to the matrix

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{A.3}$$

while **vectorexpression3**, **m2**, evaluated to itself.

A.8 Third Sample Programming Session

This time, we focus on vectors and matrices.

```
> m <- rbind(1:3,c(5,12,13))  # "row bind," combine rows
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    5   12   13
> t(m)  # transpose
     [,1] [,2]
[1,]    1    5
[2,]    2   12
[3,]    3   13
> ma <- m[,1:2]
> ma
     [,1] [,2]
[1,]    1    2
[2,]    5   12
> rep(1,2)  # "repeat," make multiple copies
[1] 1 1
> ma %*% rep(1,2)  # matrix multiply
     [,1]
[1,]    3
[2,]   17
> solve(ma,c(3,17))  # solve linear system
[1] 1 1
> solve(ma)  # matrix inverse
     [,1] [,2]
[1,]  6.0 -1.0
[2,] -2.5  0.5
```

A.9 Default Argument Values

Consider the **sort()** function, which is built-in to R, though the following points hold for any function, including ones you write yourself.

The online help for this function, invoked by

```
> ?sort
```

shows that the call form (the simplest version) is

```
sort(x, decreasing = FALSE, ...)
```

Here is an example:

```
> x <- c(12,5,13)
> sort(x)
[1]  5 12 13
> sort(x,decreasing=FALSE)
[1] 13 12  5
```

So, the default is to sort in ascending order, i.e. the argument **decreasing** has TRUE as its default value. If we want the default, we need not specify this argument. If we want a descending-order sort, we must say so.

A.10 The R List Type

The R **list** type is, after vectors, the most important R construct. A list is like a vector, except that the components are generally of mixed types.

A.10.1 The Basics

Here is example usage:

```
> g <- list(x = 4:6, s = "abc")
> g
$x
[1] 4 5 6

$s
[1] "abc"

> g$x  # can reference by component name
[1] 4 5 6
> g$s
[1] "abc"
> g[[1]]  # can reference by index, but note double brackets
[1] 4 5 6
> g[[2]]
[1] "abc"
> for (i in 1:length(g)) print(g[[i]])
[1] 4 5 6
[1] "abc"
```

```
# now have ftn oddcount() return odd count AND the odd numbers themselves,
# using the R list type
> source("odd.R")
> oddcount
function(x)  {
   x1 <- x[x %% 2 == 1]
   return(list(odds=x1, numodds=length(x1)))
}
> # R's list type can contain any type; components delineated by $
> oddcount(y)
$odds
[1]  5 13

$numodds
[1] 2

> ocy <- oddcount(y)  # save the output in ocy, which will be a list
> ocy
$odds
[1]  5 13

$numodds
[1] 2

> ocy$odds
[1]  5 13
> ocy[[1]]  # can get list elements using [[ ]] instead of $
[1]  5 13
> ocy[[2]]
[1] 2
```

A.10.2 The Reduce() Function

One often needs to combine elements of a list in some way. One approach to this is to use **Reduce()**:

```
> x <- list(4:6,c(1,6,8))
> x
[[1]]
[1] 4 5 6
```

```
[[2]]
[1] 1 6 8

> sum(x)
Error in sum(x) : invalid 'type' (list) of argument
> Reduce(sum,x)
[1] 30
```

Here **Reduce()** cumulatively applied R's **sum()** to **x**. Of course, you can use it with functions you write yourself too.

Continuing the above example:

```
> Reduce(c,x)
[1] 4 5 6 1 6 8
```

A.10.3 S3 Classes

R is an object-oriented (and functional) language. It features two types of classes, S3 and S4. I'll introduce S3 here.

An S3 object is simply a list, with a class name added as an *attribute*:

```
> j <- list(name="Joe", salary=55000, union=T)
> class(j) <- "employee"
> m <- list(name="Joe", salary=55000, union=F)
> class(m) <- "employee"
```

So now we have two objects of a class we've chosen to name **"employee"**. Note the quotation marks.

We can write class *generic functions*:

```
> print.employee <- function(wrkr) {
+    cat(wrkr$name,"\n")
+    cat("salary",wrkr$salary,"\n")
+    cat("union␣member",wrkr$union,"\n")
+ }
> print(j)
Joe
salary 55000
union member TRUE
```

```
> j
Joe
salary 55000
union member TRUE
```

What just happened? Well, **print()** in R is a *generic* function, meaning that it is just a placeholder for a function specific to a given class. When we printed **j** above, the R interpreter searched for a function **print.employee()**, which we had indeed created, and that is what was executed. Lacking this, R would have used the print function for R lists, as before:

```
> rm(print.employee)  # remove the function, to see what happens with print
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

attr(,"class")
[1] "employee"
```

A.11 Some Workhorse Functions

```
> m <- matrix(sample(1:5,12,replace=TRUE),ncol=2)
> m
[,1] [,2]
[1,]    2    1
[2,]    2    5
[3,]    5    4
[4,]    5    1
[5,]    2    1
[6,]    1    3
# call sum() on each row
> apply(m,1,sum)
[1] 3 7 9 6 3 4
# call sum() on each column
> apply(m,2,sum)
```

```
[1] 17 15
> f <- function(x) sum(x[x >= 4])
# call f() on each row
> apply(m,1,f)
[1] 0 5 9 5 0 0
> l <- list(x = 5, y = 12, z = 13)
# apply the given funciton to each element of l, producing a new list
> lapply(l,function(a) a+1)
$x
[1] 6

$y
[1] 13

$z
[1] 14
# group the first column of m by the second
> sout <- split(m[,1],m[,2])
> sout
$`1`
[1] 2 5 2
$`3`
[1] 1
$`4`
[1] 5

[1] 2
# find the size of each group, by applying the length() function
> lapply(sout,length)
$`1`
[1] 3

[1] 1
$`4`
[1] 1

$`5`
[1] 1
# like lapply(), but sapply() attempts to make vector output
> sapply(sout,length)
```

```
1 3 4 5
3 1 1 1
```

A.12 Handy Utilities

R functions written by others, e.g. in base R or in the CRAN repository for user-contributed code, often return values which are class objects. It is common, for instance, to have lists within lists. In many cases these objects are quite intricate, and not thoroughly documented. In order to explore the contents of an object—even one you write yourself—here are some handy utilities:

- **names()**: Returns the names of a list.
- **str()**: Shows the first few elements of each component.
- **summary()**: General function. The author of a class **x** can write a version specific to **x**, i.e. **summary.x()**, to print out the important parts; otherwise the default will print some bare-bones information.

For example:

```
> z <- list(a = runif(50), b = list(u=sample(1:100,25), v="blue␣sky"))
> z
$a
 [1] 0.301676229 0.679918518 0.208713522 0.510032893 0.405027042
0.412388038
 [7] 0.900498062 0.119936222 0.154996457 0.251126218 0.928304164
0.979945937
[13] 0.902377363 0.941813898 0.027964137 0.992137908 0.207571134
0.049504986
[19] 0.092011899 0.564024424 0.247162004 0.730086786 0.530251779
0.562163986
[25] 0.360718988 0.392522242 0.830468427 0.883086752 0.009853107
0.148819125
[31] 0.381143870 0.027740959 0.173798926 0.338813042 0.371025885
0.417984331
[37] 0.777219084 0.588650413 0.916212011 0.181104510 0.377617399
0.856198893
[43] 0.629269146 0.921698394 0.878412398 0.771662408 0.595483477
0.940457376
[49] 0.228829858 0.700500359
```

```
$b
$b$u
 [1] 33 67 32 76 29  3 42 54 97 41 57 87 36 92 81 31 78 12 85 73 26 44
86 40 43

$b$v
[1] "blue␣sky"
> names(z)
[1] "a" "b"
> str(z)
List of 2
 $ a: num [1:50] 0.302 0.68 0.209 0.51 0.405 ...
 $ b:List of 2
  ..$ u: int [1:25] 33 67 32 76 29 3 42 54 97 41 ...
  ..$ v: chr "blue␣sky"
> names(z$b)
[1] "u" "v"
> summary(z)
  Length Class  Mode
a 50     -none- numeric
b  2     -none- list
```

A.13 Data Frames

Another workhorse in R is the *data frame*. A data frame works in many ways like a matrix, but differs from a matrix in that it can mix data of different modes. One column may consist of integers, while another can consist of character strings and so on. Within a column, though, all elements must be of the same mode, and all columns must have the same length.

We might have a 4-column data frame on people, for instance, with columns for height, weight, age and name—3 numeric columns and 1 character string column.

Technically, a data frame is an R list, with one list element per column; each column is a vector. Thus columns can be referred to by name, using the **$** symbol as with all lists, or by column number, as with matrices. The matrix **a[i,j]** notation for the element of **a** in row **i**, column **j**, applies to data frames. So do the **rbind()** and **cbind()** functions, and various other matrix operations, such as filtering.

Here is an example using the dataset **airquality**, built in to R for illustration purposes. You can learn about the data through R's online help, i.e.

```
> ?airquality
```

Let's try a few operations:

```
> names(airquality)
[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"
> head(airquality)  # look at the first few rows
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> airquality[5,3]  # wind on the 5th day
[1] 14.3
> airquality$Wind[3]  # same
[1] 12.6
> nrow(airquality)  # number of days observed
[1] 153
> ncol(airquality)  # number of variables
[1] 6
> airquality$Celsius <- (5/9) * (airquality[,4] - 32)  # new variable
> names(airquality)
[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"     "Celsius"
> ncol(airquality)
[1] 7
> airquality[1:3,]
  Ozone Solar.R Wind Temp Month Day  Celsius
1    41     190  7.4   67     5   1 19.44444
2    36     118  8.0   72     5   2 22.22222
3    12     149 12.6   74     5   3 23.33333
> aqjune <- airquality[airquality$Month == 6,]  # filter op
> nrow(aqjune)
[1] 30
> mean(aqjune$Temp)
[1] 79.1
> write.table(aqjune,"AQJune")  # write data frame to file
> aqj <- read.table("AQJune",header=T)  # read it in
```

A.14 Graphics

R excels at graphics, offering a rich set of capabilities, from beginning to advanced. In addition to the functions in base R, extensive graphics packages are available, such as **lattice** and **ggplot2**.

One point of confusion for beginners involves saving an R graph that is currently displayed on the screen to a file. Here is a function for this, which I include in my R startup file, **.Rprofile**, in my home directory:

```
pr2file
function (filename)
{
    origdev <- dev.cur()
    parts <- strsplit(filename, ".", fixed = TRUE)
    nparts <- length(parts[[1]])
    suff <- parts[[1]][nparts]
    if (suff == "pdf") {
        pdf(filename)
    }
    else if (suff == "png") {
        png(filename)
    }
    else jpeg(filename)
    devnum <- dev.cur()
    dev.set(origdev)
    dev.copy(which = devnum)
    dev.set(devnum)
    dev.off()
    dev.set(origdev)
}
```

The code, which I won't go into here, mostly involves manipulation of various R graphics devices. I've set it up so that you can save to a file of type either PDF, PNG or JPEG, implied by the file name you give.

A.15 Packages

The analog of a library in C/C++ in R is called a **package** (and often loosely referred to as a **library**). Some are already included in base R, while others can be downloaded, or written by yourself.

```
> library(parallel)  # load the package named 'parallel'
> ls(package:parallel)  # let's see what functions it gave us
 [1] "clusterApply"        "clusterApplyLB"      "clusterCall"
 [4] "clusterEvalQ"        "clusterExport"       "clusterMap"
 [7] "clusterSetRNGStream" "clusterSplit"        "detectCores"
[10] "makeCluster"         "makeForkCluster"     "makePSOCKcluster"
[13] "mc.reset.stream"     "mcaffinity"          "mccollect"
[16] "mclapply"            "mcMap"               "mcmapply"
[19] "mcparallel"          "nextRNGStream"       "nextRNGSubStream"
[22] "parApply"            "parCapply"           "parLapply"
[25] "parLapplyLB"         "parRapply"           "parSapply"
[28] "parSapplyLB"         "pvec"                "setDefaultCluster"
[31] "splitIndices"        "stopCluster"
> ?pvec  # let's see how one of them works
```

The CRAN repository of contributed R code has thousands of R packages available. It also includes a number of "tables of contents" for specific areas, say time series, in the form of CRAN Task Views. See the R home page, or simply Google "CRAN Task View."

```
> install.packages("cts","~/myr")  # download into desired directory
--- Please select a CRAN mirror for use in this session ---
...
downloaded 533 Kb


The downloaded binary packages are in
        /var/folders/jk/dh9zkds97sj23kjcfkr5v6q00000gn/T//RtmplkKzOU/downloaded
> ?library
> library(cts,lib.loc="~/myr")

Attaching package:    c t s
...
```

A.16 Other Sources for Learning R

There are tons of resources for R on the Web. You may wish to start with the links at `http://heather.cs.ucdavis.edu/~matloff/r.html`.

A.17 Online Help

R's **help()** function, which can be invoked also with a question mark, gives short descriptions of the R functions. For example, typing

```
> ?rep
```

will give you a description of R's **rep()** function.

An especially nice feature of R is its **example()** function, which gives nice examples of whatever function you wish to query. For instance, typing

```
> example(wireframe())
```

will show examples—R code and resulting pictures—of **wireframe()**, one of R's 3-dimensional graphics functions.

A.18 Debugging in R

The internal debugging tool in R, **debug()**, is usable but rather primitive. Here are some alternatives:

- The RStudio IDE has a built-in debugging tool.
- For Emacs users, there is **ess-tracebug**.
- The StatET IDE for R on Eclipse has a nice debugging tool. Works on all major platforms, but can be tricky to install.
- My own debugging tool, **debugR**, is extensive and easy to install, but for the time being is limited to Linux, Mac and other Unix-family systems. See *http://heather.cs.ucdavis.edu/debugR.html*.

A.19 Complex Numbers

If you have need for complex numbers, R does handle them. Here is a sample of use of the main functions of interest:

```
> za <- complex(real=2,imaginary=3.5)
> za
```

```
[1] 2+3.5i
> zb <- complex(real=1,imaginary=-5)
> zb
[1] 1-5i
> za * zb
[1] 19.5-6.5i
> Re(za)
[1] 2
> Im(za)
[1] 3.5
> za^2
[1] -8.25+14i
> abs(za)
[1] 4.031129
> exp(complex(real=0,imaginary=pi/4))
[1] 0.7071068+0.7071068i
> cos(pi/4)
[1] 0.7071068
> sin(pi/4)
[1] 0.7071068
```

Note that operations with complex-valued vectors and matrices work as usual; there are no special complex functions.

A.20 Further Reading

For further information about R as a programming language, there is my book, *The Art of R Programming: a Tour of Statistical Software Design*, NSP, 2011, as well as Hadley Wickham's *Advanced R*, Chapman and Hall, 2014.

For R's statistical functions, a plethora of excellent books is available. such as *The R Book* (2nd Ed.), Michael Crowley, Wiley, 2012. I also very much like *R in a Nutshell* (2nd Ed.), Joseph Adler, O'Reilly, 2012, and even Andrie de Vries' *R for Dummies*, 2012.