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...

The world's first recommender system!

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: Using on known ratings, 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 7.

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 Tuning Parameters

The reader is likely familiar with histograms. Recall that the analyst must choose a bin width. Here there is a tradeoff:

- If the width is too small, some bins will have no data points, or very few. This will produce a very choppy appearance.
- But it's also bad to have too large a width. In the extreme, we have bins so wide that we have just one of them, totally uninformative.

The bin width is called a *tuning parameter* or a *hyperparameter*. Most machine learning algorithms, including most in recommender systems, have several tuning parameters. Choosing the values of those parameters is not easy, but there are methods for it.

Important note: Think of datasets A and B, similar but A having only 25 data points and B having 500. Then the problem of empty or nearly-empty bins is much less of an issue. In other words, we can afford to make the bin width smaller if we have dataset B, thus getting a more detailed picture.

1.4 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.

1.5 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.

1.6 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**.

1.7 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 or columns. The rank of M above is 2.

The reason we say "rows or columns" above is that it can be shown that the row rank and column rank are the same. Note that the implies that the rank of a matrix is less than or equal to the minimum of the number of rows and columns.

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: Matrix Times Vector

Consider the product of a matrix and a vector, Ax. This product is a linear combination of the columns of A. To see this, write

$$A = (A_1, A_2, A_3) \tag{2.14}$$

with A_i being the i^{th} column, and

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{2.15}$$

The point is that (2.14) looks like a row vector — it isn't, but we pretend it is — so that Ax looks like the "dot product of one vector with another. That would give us

$$Ax = x_1 A_1 + x_2 A_2 + x_3 A_3 \tag{2.16}$$

As noted, we then "unpretend," and find that (2.16) says that

> Ax is a linear combination of the columns of A. The coefficients in that linear combination are the elements of x.

2.2.3 Example: Matrix Factorization

Recall the relation

$$A \approx WH \tag{2.17}$$

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

$$A = \begin{pmatrix} a_1 \\ ... \\ a_r \end{pmatrix}, \tag{2.18}$$

and

$$H = \begin{pmatrix} h_1 \\ ... \\ h_m \end{pmatrix}, \tag{2.19}$$

so that for instance a_1 and h_1 are the first row in A and H, respectively.

Keep W unpartitiond:

$$W = \begin{pmatrix} w_{11} & ... & w_{1m} \\ ... & ... & ... \\ w_{r1} & ... & w_{rm} \end{pmatrix}, \tag{2.20}$$

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

$$WH = \begin{pmatrix} w_{11} & ... & w_{1m} \\ ... & ... & ... \\ w_{r1} & ... & w_{rm} \end{pmatrix} \begin{pmatrix} h_1 \\ ... \\ h_m \end{pmatrix} = \begin{pmatrix} w_{11}h_1 + ... + w_{1m}h_m \\ ... \\ w_{r1}h_1 + ... + w_{rm}h_m \end{pmatrix} \tag{2.21}$$

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

$$M = \begin{pmatrix} 1 & 5 & 1 & -2 \\ 8.02 & 2.99 & 2 & 8.2 \\ 9 & 8 & 3 & 6 \end{pmatrix} \tag{2.22}$$

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 (5.1).

(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?

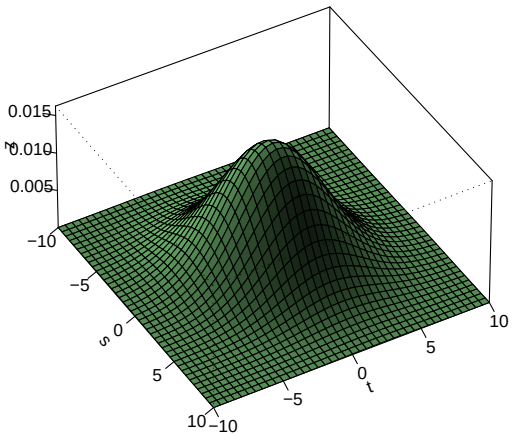
In the RS context, just think of, say, Amazon. The business has millions of users and millions of items. In other words, the ratings matrix has millions of rows and millions of columns, and even one million rows and columns would mean a total number of $(10^6)^2 = 10^{12}$ entries, about 8 terabytes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

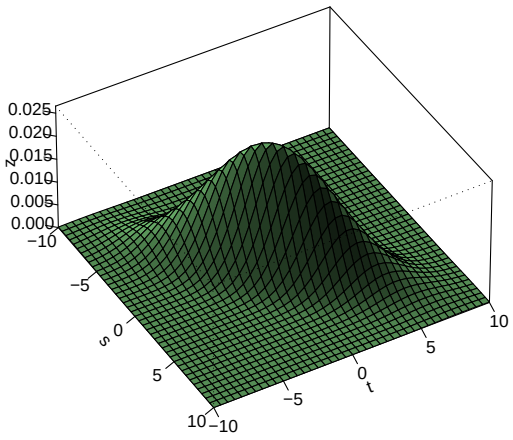
This is a core point in statistics/machine learning, the notion of *dimension reduction*. In complex applications, there is a pressing need to reduce the number of variables down to a manageable number — manageable not only in terms of computational time and space, but also the statistical problem of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

So we need methods to eliminate redundant or near-redundant data, such as row 3 in (2.22).

2.3.2 Exploiting Correlations

Statistically, the issue is one of correlation. In (2.22), 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, say human height and weight, 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 X_2 is largely redundant. So:

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

In the case of correlation 0.2, there really are two separate variables. The probability that $X_2 \approx -X_1$ is lower here.

Note one more time, though, the approximate nature of the approach we are developing. There really *are* two variables even in that correlation 0.8 example. 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

$$Y_1 = X_1 + X_2 \tag{2.23}$$

and

$$Y_2 = X_1 - X_2 \tag{2.24}$$

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. Remember, we want to reduce the number of variables we'll work with, and redundancy would say that we might be able to reduce further.

With that in mind, now suppose we have p variables, $X_1, X_2, ..., X_p$, not just two. If our data is on people, these variables may be height, weight, age, gender and so on. 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) They are ordered in terms of variance:

$$Var(Y_1) > Var(Y_2) > ... > Var(Y_p) \tag{2.25}$$

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 since they are essentially constants, uninformative. 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

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{s.d.(x)\ s.d.(y)} \tag{2.26}$$

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 — i.e. the set of all $\text{corr}(X_i, X_j)$ — 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*:

$$C' = C \tag{2.27}$$

where $'$ denotes matrix transpose.

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

$$Lx = \lambda x \tag{2.28}$$

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*,

$$U_i' U_j = 0, \ i \neq j \tag{2.29}$$

We always take the U_i to have length 1: Just divide the vector by its length, so it now has length 1, and is still an eigenvector.

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 again that since there are p variables, the correlation matrix will be $p \times p$.

2. Compute its eigenvalues and eigenvectors. It can be shown that the eigenvalues are the variances of our new variables.

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 $p \times p$.

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 $U^{(s)}$.

6. Form a new data matrix,

$$R = QU^{(s)} \tag{2.30}$$

R will be of size $n \times s$. So, we've replaced our original p variables, the p columns of Q, by s new variables, the columns of R. Column j of R is called the j^{th} principal component of the original data.

As mentioned, it can be shown that the variance of the j^{th} principal component is λ_j. The sum of all p eigenvalues 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 Applying Matrix Partitioning

Using the approach of Section 2.2, partition $U^{(s)}$ into its columns,

$$U^{(s)} = (U_1, ..., U_s) \tag{2.31}$$

and thus write

$$R = QU^{(s)} = Q(U_1, ..., U_s) \tag{2.32}$$

After that second equality, pretend that Q and the U_i are "numbers." Then the last expression in (2.32),

$$Q(U_1, ..., U_s) \tag{2.33}$$

is a "scalar" times a "vector," and is thus equal to

$$(QU_1, ..., QU_s) \tag{2.34}$$

So, the i^{th} column of R is QU_i. The latter quantity is of the Ax, matrix-times-vector form of Section 2.2.3, so it 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 Choosing the Number of Principal Components

The number of components we use, s, is called a *tuning parameter* or *hyperparameter*. 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.

2.3.7 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.

2.3.8 More on the PC Coefficients

There is more to consider.

Do the PC coefficients have any interpretation? The answer is probably no for ordinary people, but for the *domain experts*, very possibly yes. In the teaching evaluation example above, a specialist in survey design or teaching methods may well be able to interpret the dominance of Q1 in the second PC. A method called *factor analysis*, an extension of PCA, is popular in social science research.

For the rest of us, PCA is just a handy way to do dimension reduction.

But there is geometric terminology that will be helpful, as follows. Let's look at the **mlb** dataset from the **regtools** package. This is data on Major League baseball players.

```
              Name Team        Position Height Weight   Age
1   Adam_Donachie  BAL         Catcher     74    180 22.99
2       Paul_Bako  BAL         Catcher     74    215 34.69
3 Ramon_Hernandez  BAL         Catcher     72    210 30.78
4    Kevin_Millar  BAL  First_Baseman     72    210 35.43
5     Chris_Gomez  BAL  First_Baseman     73    188 35.71
6   Brian_Roberts  BAL Second_Baseman     69    176 29.39
  PosCategory
1     Catcher
2     Catcher
3     Catcher
4   Infielder
5   Infielder
6   Infielder
```

Let's apply PCA:

```
> hw <- as.matrix(mlb[,4:5])
> pcout <- prcomp(hw)
> pcout$rotation
                PC1          PC2
Height -0.05948695  0.99822908
Weight -0.99822908 -0.05948695
```

If we were to plot **hw**, we would put **hw[1,]** at the point (74,180) on our graph. Recall from high school math that 74 and 180 are called the *coordinates* of **hw2[1,]**, with respect to our "H axis" and "W axis."

But in doing PCA, we are creating new axes, PC1 and PC2, which are rotated versions of the H and W axes. (Hence the naming of the U matrix as "rotation" in the **prcomp()** return value.) Let's find the coordinates of **hw[1,]** with respect to the new axes:

```
> hw[1,] %*% pcout$rotation
           PC1     PC2
[1,] -184.0833 63.1613
```

So (74,180) has become (-184.1,63.2) under the new coordinate system. Let's see what the angle of rotation is. We can do that by seeing where a point on the H axis rotates to.

```
> pc10 <- c(1,0) %*% pcout$rotation
> pc10
             PC1       PC2
[1,] -0.05948695 0.9982291
> (atan(pc10[2] / pc10[1])) * 180/pi
[1] -86.58964
```

Almost 90 degrees clockwise.

2.3.9 Scaling

Some analysts prefer to *scale* the data before applying PCA. For each column, we would subtract the column mean and divide by the column standard deviation. The column would now have mean 0.0 and variance 1.0.

The rationale for doing this is that if PCA is applied to the original data, variables with large variance will dominate. And then units would play a role; e.g. a distance variable would have more impact if it were measured in kilometers than miles.

Scaling does solve this problem, but its propriety is questionable. Consider a setting with two features, A and B, with variances 500 and 2, respectively, and with mean 100 for both. Let A' and B' denote these features after centering and scaling.

As noted, PCA is all about removing features with small variance, as they are essentially constant. If we work with A and B, we would of course use only A. But if we work with A' and B', we would use both of them, as they both have variance 1.0.

So, dealing with the disparate-variance problem (e.g. miles vs. kilometers) shouldn't generally be solved by ordinary scaling, i.e. by dividing by the standard deviation. An alternative is to divide each column by its mean. This addresses the miles-vs.-kilometers problem, and makes sense in that a variance is large or small in relation to its mean.

2.4 Vector Norms

In math, the l_p *norm* of a vector in n-dimensional space is defined by

$$||x||_p = \left(\sum_{i=1}^{n} [x_i|^p \right)^{1/p} \tag{2.36}$$

This is actually a family of norms, for $1 \leq p \leq \infty$.[1] You are familiar with the Euclidean norm, $p = 2$. In statistics and machine learning, we are often minimizing the norm of some vector, usually with either $p = 1$ or $p = 2$.

You will often see the notation L_p instead of l_p. However, in math the former is used for function spaces, while the latter designates n-dimensional vectors, and we use the latter here.

You will also see references to the *Frobenius* norm of an $r \times s$ matrix. That is actually just the l_2 norm, treating the matrix as an rs-dimensional vector.

2.5 Handy Facts

- For conformable matrices A and B,

$$(AB)' = B'A' \tag{2.37}$$

- For invertible and conformable matrices A and B,

$$(AB)^{-1} = B^{-1}A^{-1} \tag{2.38}$$

- The R functions **t()** and **solve()** find the transpose and inverse of their matrix arguments. A more numerically stable function for inversion is **qr()**.

[1] $||x||_\infty = \max_i |x_i|$

Chapter 3

Some Infrastructure: Probability, Statistics and Regression Analysis

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.

This chapter will lay some groundwork, after which we will devote the following chapter to the crucial concept 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.

The data $X_1, ..., X_n$ is considered a *random sample* from a finite population if (a) each X_i has the same distribution as the population, i.e. every element of the population has the same chance of being chosen and (b) the X_i are independent.

We will usually think in terms of sample data here.

3.2 Probability, Expected Value and Variance

3.2.1 Long-Run View

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.

3.2.2 Expected Value and Variance

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.

3.2.3 Important Properties of Expected Value and Variance

- For random variables U and V,

$$E(U + V) = EU + EV \tag{3.1}$$

  even if U and V are not independent.

- If U and V *are* independent, then

$$Var(U + V) = Var(U) + Var(V) \tag{3.2}$$

- For any constant c, $E(cU) = cEU$ and $Var(cu) = c^2 Var(U)$.

- Let $\overline{X} = (X_1 + ... + X_n)/n$ be the sample mean from a random sample from a population having mean μ and variance σ^2. Then

$$E(\overline{X} = \mu \tag{3.3}$$

  and

$$Var(\overline{X} = \frac{\sigma^2}{n} \tag{3.4}$$

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,

$$E(W \mid H = t_1, A = t_2) \tag{3.5}$$

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 (3.5), 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 6.

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.

$$\text{mean weight} = \beta_0 + \beta_1 \text{ height} + \beta_2 \text{ age} \tag{3.6}$$

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 3.4.5.

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 **??**.

3.3.3.3 Comparison

Consider (3.6), our model for the function of t_1 and t_2

$$E(\text{weight} \mid \text{height} = t_1, \text{age} = t_2) \tag{3.7}$$

With the linear assumption (3.6), 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 (3.6). 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.

3.4 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.

Warning: There are various different versions of the MovieLens data. Your version may yield different results than what you see in this book.

3.4.1 A First Look

We'll define gender as 1 for male, 0 for female. We find (details below) that our estimated regression function of rating on age and gender is

$$\text{estimated mean rating} = 3.3599 + 0.005311 \text{ age} - 0.0069 \text{ gender} \tag{3.8}$$

(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:

```
# 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]
```

[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.

[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.

```
[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.

3.4.2 Polynomial Terms

People tend to gain weight during middle age, but often they lose weight when they become elderly. So (3.6), 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:

$$\text{mean weight} = \beta_0 + \beta_1 \text{ height} + \beta_2 \text{ age} + \beta_3 \text{ age}^2 \tag{3.9}$$

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 (3.9) 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 $\times$ 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 Chapter 4.

We'll have a long example in Section 4.2.

3.4.3 Dummy Variables

Ofter our variables will be *categorical*. Say we have data on US residents, including a variable for state of residence. In R, that would be coded as a *factor*. But in prediction that ID code for a state has no numerical meaning. A state with code 12, say is not "twice as good" as a state with ID 6. So, we would typically break that single variable, State, into 50 *dummy* variables, one for each state.[5] The dummy for California, say, would have the value 1 for those living in the state, and 0 otherwise.

In many R functios, including **lm()**, R automatically converts factors to dummies.

[5] We'd account for DC etc. as well.

3.4.4 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:

$$\text{mean rating} = \beta_0 + \beta_1 \text{ age} + \beta_2 \text{ male} + \beta_3 \text{ age} \times \text{male} \tag{3.10}$$

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:

$$\text{mean rating} = (\beta_0 + \beta_2) + (\beta_1 + \beta_3) \text{ age} \tag{3.11}$$

women:

$$\text{mean rating} = \beta_0 + \beta_1 \text{ age} \tag{3.12}$$

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.[6]

By the way, note *how* we would fit this model to our data. Our data frame has columns for rating, age and gender. We would then add a new column, computed as the product of the age and gender columns.

The term age $\times$ 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.

On the other hand, interaction terms don't make sense in some contexts. In the MovieLens data, we have a column for User 12 and one for User 39 (and many others). How about modeling the interaction between those two users?

On one level, one might immediately say No. Probably these two users don't even know each other. But even more, think of the mechanics. The product of the User 12 and User 39 columns will be

[6] 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.

all 0s! Intuitively, that would be useless, and mathematically the matrix inversion in (3.20) would be impossible.

3.4.5 Details of Linear Regression Estimation

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

$$r = \sum_{i=1}^{n} [W_i - (b_0 + b_1 H_i + b_2 A_i)]^2 \tag{3.13}$$

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, set them to 0, then set $\widehat{\beta}_k$ to the minimizing b_k. For example, we set

$$0 = \frac{\partial r}{\partial b_1} = -2 \sum_{i=1}^{n} [W_i - (b_0 + b_1 H_i + b_2 A_i)](-H_i) \tag{3.14}$$

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

$$r = (D - Ab)'(D - Ab) \tag{3.15}$$

(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,

$$\frac{\partial r}{\partial b} = (\frac{\partial r}{\partial b_0}, \frac{\partial r}{\partial b_1}, ..., \frac{\partial r}{\partial b_p})' \tag{3.16}$$

where $p+1$ is the number of predictor variables.[7]

It can be shown that for a vector u,

$$\frac{\partial u'u}{\partial u} = 2u \tag{3.17}$$

(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

$$\frac{\partial r}{\partial b} = \frac{\partial r}{\partial u}\frac{\partial u}{\partial b} = 2(D - Ab)\ \frac{\partial (D - Ab)}{\partial b} = 2(-A')(D - Ab) \tag{3.18}$$

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

$$0 = A'D - A'Ab \tag{3.19}$$

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

$$b = (A'A)^{-1}A'D \tag{3.20}$$

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

3.4.6 Linear Dependence Issues

Note that the solution in (3.20) exists and is unique, providing we do not have linearly dependent rows in our data, which would cause A to be 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

$$A = \begin{pmatrix} 1 & Age_1 & Male_1 & Female_1 \\ 1 & Age_2 & Male_2 & Female_2 \\ ... & & & \\ 1 & Age_{100000} & Male_{100000} & Female_{100000} \end{pmatrix} \tag{3.21}$$

[7] 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.

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 (3.20). In other words, not only would the Female column be unnecessary, it would be problematic.

But in many cases, the columns of A will be *approximately* linearly dependent, a situation called *multicollinearity*. In such cases, the computation of $(A'A)^{-1}$ may produce substantial roundoff error, causing unreliable answers. The **lm()** function issues a warning of a "rank-deficient" solution.

And in our RS setting, a subtle but vital problem arises, in terms of covariates. Consider for instance the call

```
lm(rating ~ userID + age)
```

If we know the user's ID, we know her age, so there is a perfect dependency. Consider a small example, with two users, of ages 28.8 an 39.0, and a dummy variable for the first. Then we would have

age column = 39.0 × the 1s column - 10.2 × user1 column

For **lm()** and **glm()** (which solves an equation like (3.20), there is not much that can be done here.

3.5 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

$$\text{probability of click} = \beta_0 + \beta_1 \text{ age} + \beta_2 \text{ gender} \tag{3.22}$$

One problem, though, is that a probability should be in [0,1] yet the right-hand side of (3.22) 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

$$P(Y = 1 \mid X_1 = t_1, ..., X_p = t_p) = \frac{1}{1 + \exp -(\beta_0 + \beta_1 t_1 + ... + \beta_p t_p)} \tag{3.23}$$

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$rating >= 4)  # a binary value, 1 or 0
> u$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]

3.5.1 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()**,

[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'.**

[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.

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)
    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.

3.5.2 Full Example

```
> rats <- read.table('u.data')
> head(rats)
   V1  V2 V3        V4
1 196 242  3 881250949
2 186 302  3 891717742
3  22 377  1 878887116
4 244  51  2 880606923
5 166 346  1 886397596
6 298 474  4 884182806
> class(rats$V1)
[1] "integer"
```

```
> rats$V1 <- as.factor(rats$V1)
> rats$V2 <- as.factor(rats$V2)
> lmout <- lm(V3 ~ V1+V2,data=rats)  # runs about 10 mins
> coefs <- lmout$coefficients
> str(coefs)
 Named num [1:2624] 3.913 0.041 -0.529 0.88 -0.457 ...
 - attr(*, "names")= chr [1:2624] "(Intercept)" "V12" "V13" "V14" ...
# let's try predicting something
> newx <- rats[5,1:2]
> newx
   V1  V2
5 166 346
# how would user 166 like movie 8?
> newx$V2 <- '8'  # character, due to factor
# R factors are essentially character vectors with named levels
> newx
   V1 V2
5 166  8
> predict(lmout,newx)
       5
4.399462
```

A few comments:

- The V1 and V2 columns were numbers, but those "numbers" were user and movie IDs. We need to convert them to dummy variables. R will do that for us, provided we change them to factors.
- With over 900 users and 1600 movies, that's over 2500 dummies; 2624, to be exact.
- The **predict()** function is really handy, but its second argument needs to be a data frame (even if only one row) of the same structure as what went into **lm()**. The easiest way to do this is to start with one row of that data frame, then modify as needed.
- It's nice that we got a prediction for this user, but is it accurate? More on this later.

3.5.3 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[,c(1,3)])  # cat vs. all else
gfox <- glm(Fox ~ .,data=df[,c(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.

Chapter 4

Some Infrastructure: Model Selection and Overfitting

Model selection is hard, more of an art than a science. By far the most vexing issue in statistics and machine learning is that of *overfitting*.

4.1 Toy Example

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*.

4.2 Real Example

Let's illustrate this on the dataset **prgeng**, assembled from the 2000 US census. It consists of wage and other information on 20090 programmers and engineers in Silicon Valley. This dataset is included in **tegtools** package. We will also use the **polyreg** package, which fits polynomial models as we saw in Section 3.4.2 above.[1]

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

As usual, let's take a glance at the data:

```
> head(prgeng)
       age educ occ sex wageinc wkswrkd
1 50.30082   13 102   2   75000      52
2 41.10139    9 101   1   12300      20
3 24.67374    9 102   2   15400      52
4 50.19951   11 100   1       0      52
5 51.18112   11 100   2     160       1
6 57.70413   11 100   1       0       0
```

Note that education, occupation and sex are categorical variables, which R will convert to dummies for us. Here is the code:

```
library(regtools)
# polyreg does polynomial regression, forming the powers,
# cross products etc. and then calling lm()
library(polyreg)
data(prgeng)
pe <- prgeng[,c(1:4,6,5)]  # "Y" variable is assumed last column in polyFit()
head(pe)
tstidxs <- sample(1:nrow(prgeng),1000)
petrn <- pe[-tstidxs,]
petst <- pe[tstidxs,]
for (i in 1:4) {
   pfout <- polyFit(petrn,deg=i)
   preds <- predict(pfout,petst)
   print(mean(abs(preds-petst$wageinc)))
   print(length(pfout$fit$coefficients))
}
```

And the resulting Mean Absolute Prediction Errors and p Values :

degre	MAPE	p
1	24248.43	24
2	23468.56	164
3	24367.16	496
4	818934.9	1020

Remember, p is the number of predictors, including all the dummies. When we have a degree 2 model, we have all the squared terms (except for the dummies), and the cross-product terms, e.g. interaction between age and gender. So p increase pretty rapidly with degree.

In any event, though, the effects of overfitting are clear.

4.3 Bias vs. Variance

Let's take a closer look, in an RS context. Say we believe (3.10) 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.[2]

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 (postal) codes in the US; to have a dummy for each would almost certainly be overfitting. If we have only, say, 100,000 rows in our data, on average each ZIP code would have only about 2 rows, hardly enough for a good estimate of the effect of that code.

4.4 Mathematical Analysis of the Bias vs. Variance Tradeoff

Let's take a more precise look, employing a simple mathematical model.

[2]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.

4.4.1 The Setting

Suppose we have the samples of men's and women's heights, $X_1, ..., X_n$ and $Y_1, ..., Y_n$. Assume for simplicity that the population variance of height is the same for each gender, σ^2. The means of the two populations are designated by μ_1 and μ_2.

Say we wish to guess the height of a new person who we know to be a man but for whom we know nothing else. We do not see him, etc.

Suppose for just a moment that we actually know the distribution of X, i.e. the *population* distribution of male heights. What would be the best constant g to use as our guess for a person about whom we know nothing other than gender?

It is easily shown that the mean squared error MSE

$$E[(g - X)^2] \tag{4.1}$$

is minimized by setting $g = \mu_1$. Our best guess for this unseen man's height is the mean height of all men in the population. (Note that "mean" above averaged over all possible men in the population.)

Of course, we don't know μ_1, but we can do the next-best thing, i.e. use an estimate of it from our sample. The natural choice for that estimator would be

$$T_1 = \overline{X}, \tag{4.2}$$

the mean height of men in our sample.

4.4.2 Context of Interest: Very Small Sample

But what if our sample size n is really small, say $n = 5$? That's awfully small. We may wish to consider pooling the women's heights into our estimate, in order to get a larger sample. Then we would estimate μ_1 by incorporating the sample mean of women's heights, $\overline{Y}$:

$$T_2 = \frac{\overline{X} + \overline{Y}}{2}, \tag{4.3}$$

It may at first seem obvious that T_1 is the better estimator. Women tend to be shorter, after all, so pooling the data from the two genders would induce a bias, defined as

$$\text{bias} = \text{mean of the estimator - true population value} \tag{4.4}$$

Here "mean" refers to the average of the estimator over all possible samples from this population. It can be shown that for a sample mean M, drawn from a population with mean ν,

$$E(M) = \nu \tag{4.5}$$

In other words, M has 0 bias. Thus our T_1 here has 0 bias. But that is not the case for T_2:

$$E(T_2) = 0.5E(\overline{X}) + 0.5E(\overline{Y}) = (\mu_1 + \mu_2)/2 < \mu_1 \tag{4.6}$$

In other words. T_2 would have a negative bias.

For an estimator of T of some population quantity θ, its *mean square error* is defined to be

$$MSE = E[(T - \theta)^2] \tag{4.7}$$

One can derive that

$$\text{MSE} = \text{variance of the estimator} + \text{bias of the estimator}^2 \tag{4.8}$$

In other words, *some amount of bias may be tolerable*, if it will buy us a subtantial reduction in variance. After all, women are not that much shorter than men, so the bias might not be too bad. Meanwhile, the pooled estimate should have lower variance, as it is based on $2n$ data points, rather than n.

Before continuing, note first that T_2 is based on a simpler model than is T_1, as T_2 ignores gender. We thus refer to T_1 as being based on the more complex model.

So, the question becomes, which has the smaller MSE, T_1 or T_2? In other words:

> Which is smaller, $E[(T_1 - \mu_1)^2]$ or $E[(T_2 - \mu_1)^2]$?

4.4.3 Drawing Conclusions from This Example

After some elementary math stat operations, one can show that T_1 is a better predictor than T_2 if

$$\left(\frac{\mu_2 - \mu_1}{2}\right)^2 > \frac{\sigma^2}{2n} \tag{4.9}$$

Granted, we don't know the values of the μ_1 and σ^2, so in a real situation, we won't really know whether to use T_1 or T_2. But the above analysis makes the point that under some circumstances, it really is better to pool the data in spite of bias.

So you can see that T_1 is better only if either

- n is large enough, or
- the difference in population mean heights between men and women is large enough, or
- there is not much variation within each population, e.g. most men have very similar heights

In other words:

> A more complex model is more accurate than a simpler one only if either
>
> (a) we have enough data to support it, or
> (b) the complex model is sufficiently different from the simpler one

A very rough, intuitive way to view (a) is that our data is being "shared" by all the parameters to be estimated. In our example above, the simple model had one parameter, μ while the complex one had two, μ_1 and μ_2. Due to this "sharing," each parameter in the complex version has "a smaller piece of the pie."

In Section 4.2, we ran an **lm()** model with 2624 parameters, definitely a complex model. Was $n = 100000$ large enough to satisfy (a) above? We don't know, but again, it raises the issue of possible overfitting.

4.5 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.**

4.5.1 Rough Rule of Thumb

One quick rule, backed up by mathematical theory, 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, and I find it tends to be overly conservative.

4.5.2 Cross-Validation

From the polynomial-ftting example in Section 4.1, we see the following key point:

> An assessment of predictive ability, based on predicting the same data on which our model is fit, tends to be overly optimistic and may be meaningless or close to it.

This motivates the most common approach to dealing with the bias/variance tradeoff, *cross validation*. In the simplest version, one randomly splits the data into a *training set* and a *test set*.[3] 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.

For example, consider the analysis in Section 3.5.2. Say we are considering adding age and gender as predictors. Call the original model Model 1 and the one with predictors added Model 2. We would do the following:

(a) Randomly partition the 100,000 rows of the data frame into training and holdout tests of size 95,000 and 5000.

(b) Fit Model 1 to the training set, and use it on the predictor values in the holdout set to predict the ratings in that set. Compute some accuracy measure, say Mean Absolute Prediction Error (MAPE).

(c) Do as in (a), but with Model 2 instead of Model 1.

(d) Compare the two MAPE values, and choose the better model.

(e) For all (user,movie) pairs with unknown ratings, use the model from (d) to predict.

(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.)

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] The latter is also called a *holdout set* or a *validation set*. Note that there are many variants of this, e.g. something called *K-fold cross validation*.

4.5.3 Regularization

Suppose we are estimating a vector mean μ, using sample data on a vector X. For instance, we may have X equal to (height,weight,age,blood pressure). Following standard notation, let p denote the number of components of X, e.g. $p = 4$ in the above example.

The standard estimate is of course the sample mean, $\overline{X}$. In the above example, this would be the 4-vector consisting of the averages of height, weight, age and blood pressure in our sample.

Some years ago, mathematical statistician Charles Stein caused quite a stir by proving the following remarkable fact:

- If $p \leq 2$, then $\overline{X}$ is the optimal estimator of μ.[4]
- If $p \geq 3$, then the optimal estimator is $c\overline{X}$ for some $0 < c < 1$.

  h

So, in higher dimensions — remember, we are working with p in the dozens or even hundreds — we should shrink down our estimator. The intuition here is this: Occasionally sample data will contain some really extreme data points, and these skew our $\overline{X}$ estimator. By shrinking down the estimator, we reduce the influence of those extreme values. And with $p \geq 3$, extreme values happen often enough to make shrinkage (or *regularization*) a "win."

This was later applied to linear regression models, PCA and so on. Instead of finding b that minimizes (3.13), we minimize

$$r = \sum_{i=1}^{n} [W_i - (b_0 + b_1 H_i + b_2 A_i)]^2 + \lambda ||b||_1 \tag{4.10}$$

where $||b||_1$ is the l_1 vector norm of b:

$$||b|| = \sum_{i=1}^{p} |b_i| \tag{4.11}$$

This is not done directly out of concern for outliers so much as **is as a remedy to overfitting.** In the polynomial models we discussed earlier, higher-degree models at least have more components in b but also tend to be larger due to high variance. Of course, we have to choose λ, a tuning parameter (as s was for PCA); this is typically done by trying various values and assessing via cross-validation.

[4] Under Mean Squared Error loss.

This technique in linear regression is called the *LASSO*, the Least Absolute Shrinkage and Selection Operator. A popular implementation in R is the **lars** package.

4.6 The Problem of p-Hacking

The (rather recent) term *p-hacking* refers to the following abuse of statistics.[5]

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.[6] 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.[7]

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.

This also implies that cross-validation is no panacea either. If we compare a large number of models,

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

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

[7] 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.

there is a danger that one looks really good when it is not.

4.7 A Note on Covariates

Covariates can substantially enhance prediction power for users or items for whom we have rather little data, but their impact otherwise may be small.[8] The intuition underlying this is that if we have a lot of ratings from some user, they already tell us lots about her. Covariate information about this user may not add much to what we already know.[9]

Consider for instance the House Voting dataset at UCI.[10] For each member of the US House of Representatives, we have their votes, Yes or No, on some bills. One can treat this as a "recommender system," with the congresspeople as users and bills as items. We can thus try to predict how a given member of Congress would have voted on some bill on which she did not actually vote.

The data also tell us whether the politician is a Democrat or Republican, so party affiliation is a covariate. But if she has voted on many bills, that record already gives us a good idea as to her party, and thus this covariate may not be very helpful.

On the other hand, another possible covariate would be the congressional district that this person represents. If it is one that is heavily agricultural, for instance, she would likely support a farm bill, regardless of party. Note that in that case, the type of bill is a covariate for the item, so both covariates could be important.

[8] This rules out the MovieLens data, for instance, as its curator omitted users who had rated fewer than 25 movies.
[9] Note that with a linear model, we could not use the covariate anyway, as explained in Section 3.4.6.
[10] https://archive.ics.uci.edu/ml/datasets/congressional+voting+records

Chapter 5

Matrix Factorization Methods

This chapter covers one of the more popular methods for handling matrix factorization problems.

5.1 The Setting

Recall the brief introduction in Chapter 1: Let A denote the ratings matrix, with the element in row i, column j storing the rating of item j given by user i. Most of A is unknown, i.e. NA values in R. We wish to estimate the unknown ones.

Say the dimension of A is $u \times v$. We wish to find rank-k matrices W $(u \times k)$ and H $(k \times v)$ such that

$$A \approx WH \tag{5.1}$$

Note that the W and H that we find will be *fully known*, with values that will be derived somehow from the known elements of A.

Again, most of the elements of A are unknown. But it is typically the case that similar users have similar ratings patterns, and the matrix factorization will hopefully exploit that. We thus hope to obtain good estimates of all of A even though only a small part of it is known.

Note that k is a tuning parameter, chosen by the user. One might use cross-validation in making this choice, comparing performance of various values of k.

This is a form of dimension reduction, with the rank k controlling the bias-variance tradeoff.

Typically a good approximation can be achieved with

$$k \ll \text{rank}(A) \tag{5.2}$$

5.2 Finding W and H

There are two main approaches to matrix factorization in recommender systems and general machine learning:

- Singular Value Decomposition (SVD): This is a "cousin" of PCA, kind of a "square root" of the latter. There is a function in base R for this, **svd()**.
- Nonnegative Matrix Factorization (NMF): Here the matrix A has nonnegative elements, and one desires the same property for W and H. This may lead to sparsity in WH, and in some cases a helpful intepretability. There are several R packages for this; see below.

Since most of the issues are the same for both methods, we'll mainly stick to one, NMF.

5.3 Notation

We'll use the following notation for a matrix Q

- Q_{ij}: element in row i, column j
- $Q_{i\cdot}$: row i
- $Q_{\cdot j}$: column j

5.4 Synthetic, Representative Recommender Systems Users

Note the key relation, which we showed in Section 2.2:

$$(WH)_{i\cdot} = \sum_{m=1}^{k} W_{im} H_{m\cdot} \tag{5.3}$$

In other words, in (5.1), we have that:

- The entire vector of predicted ratings by user i can be expressed as a linear combination of the rows of H.
- The rows of H can thus be thought of as synthetic "users" who are representative of users in general. H_{rs} is the rating that synthetic user r gives item s.

In this manner, we can predict ratings for any user that is already in A. but what about an entirely new user? What we need is the coordinates of this new user with respect to the rows of H. We'll see how to get these later in the chapter.[1]

Of course, interchanging the roles of rows and columns above, we have that the columns of W serve as an approximate basis for the columns of A. In other words, the latter become synthetic, representative items, e.g. representative movies in the MovieLens data.

5.5 Vector Space View

Recall that the *span* of a set of vectors in a vector space is the set of all linear combinations of those vectors. The term *basis* in means a linearly independent set of vectors that spans the given subspace, i.e. any vector in the subspace can be expressed as a linear combinations of the basis vectors.

Thus the rows of H can be thought of as an "approximate basis" for the span of the rows of A.

5.6 The Case of Entirely Known A

In RS applications, the matrix A is only partially known. But it will be easier to understand the methods for finding the W and H matrices by looking at another class of applications. In fact, NMF has become widely used in a variety of ML applications in which the matrix A entirely known.

5.6.1 Image Classification

: **The setting:**

Say we have n image files, each of which has brightness data for r rows and c columns of pixels. We also know the class, i.e. the subject, of each image. The famous MNIST dataset, for instance, consists of 70,000 28x28 images of hand-drawn digits; here $n = 70000$ and $r = c = 28$. Thus the

[1] After accumlating enough new users, of course, we should update A, and thus W and H.

data for an image consists of $28^2 = 784$ pixel intensities, each in the range 0,1,2,...,255. (255 is fully black, 0 is fully white and the others are shades of gray.) We have 10 classes, '0' through '9'.

We wish to predict the classes of new images. Denote the class of image j in our original data by C_j.

We form a matrix A with n rows and $w = rc$ columns, where the i^{th} row, $A_{i.}$ stores the data for the i^{th} image, say in row-major order:[2] $A_{i.}$ would first store row 1 of that image, then store row 2 of the image, and so on.[3]

Dimension reduction:

Say we wish to use the logit approach to the MNIST data. That would mean that the features portion of our data is a 70000×784 matrix. With that many columns, computation may be extremely long, and since $\sqrt{70000} \approx 265$, we'd run a big risk of overfitting with 784 features. So, dimension reduction is imperative.

One form of dimension reduction would be PCA.[4] We could apply PCA to that 70000×784 matrix, and then use, say, the first 50 principal components. Our features matrix would now be of size 70000×50, much more manageable.

Approach is NMF. In the sense stated above, the rows of H serve as synthetic, representative images. Row i of A, i.e. the i^{th} image in our training data, is then approximately a linear combination of the rows of H, with the coordinates being the elements of row i of W.

Predicting new, unlabeled images:

So, just as in the PCA case, we transform our training data, via $A \to W$. We apply our favorite classification method — for concreteness, let's assume here that it is the logistic — to this new training data (together with the class vector for that data), then use it to classify new data vectors S.

To do the latter, we must find the coordinates of S with respect to the rows of H. This means finding the linear combination of rows of H that is closest to $S, i.e.$

$$\lambda = \arg\min_l ||S - l'H|| \tag{5.4}$$

(We'll see how to do the minimization shortly.) Then λ will be the coordinate vector for the new case. This is then plugged into our logit model, to predict the new case.

[2] Make sure not to confuse the rows of A with the rows of an image. One row of A contains the totality of rows and columns of one image.

[3] For simplicity here we will assume greyscale. For color, each row of A will consist of three pixel vectors, one for each primary color.

[4] *The "C" part of Convolutional Neural Networks also does dimension reduction, but that approach is not relevant to the discussion here.*

Thus we are going from rc variables to rk of them, where k is the chosen rank. If

$$k \ll \text{rank}(A) \tag{5.5}$$

we have a big dimension reduction.

Again, there is the issue of choosing k, as with choosing s in PCA, and so on. More on this in Section 5.10.

5.6.2 Text classification

Here A consists of, say, word counts. We have a list of k keywords, and d documents of known classes (politics, finance, sports etc.). Row i of A contains the counts of the various keywords (or maybe just a binary variable indicating presence or absence of the word). Otherwise, the situation is the same as for image recognition above.

5.7 The R Package NMF

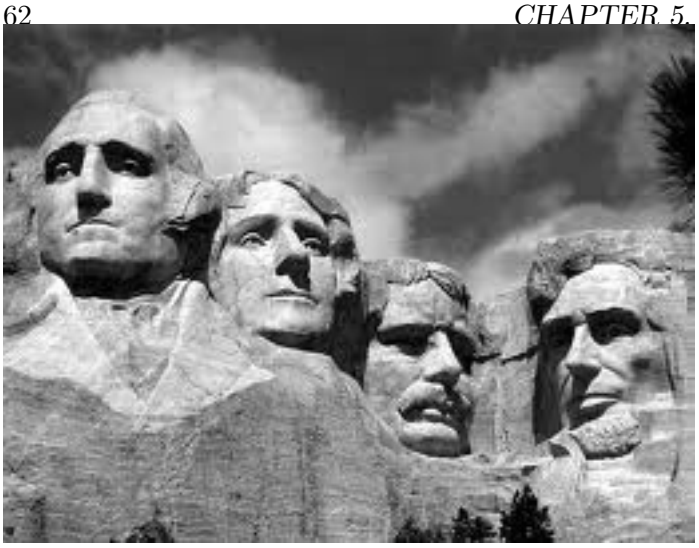
The R package **NMF** is quite extensive, with many, many options. In its simplest form, though, it is quite easy to use. For a matrix **a** and desired rank **k**, we simply run

```
> nout <- nmf(a,k)
```

Here the returned value **nout** is an object of class **"NMF"** defined in the package. It uses R's S4 class structure, with `@` as the delimiter denoting class membership, as opposed to $ as in the S3 case.

As is the case in many R packages, **"NMF"** objects contain classes within classes. The computed factors are in **nout@fit@W** and **nout@fit@H**.

Let's illustrate it in an image context, using the following:



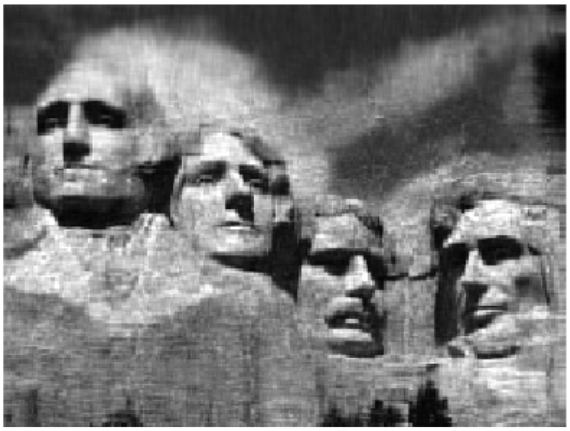
Here we have only one image, and we'll store it as a matrix A (rows of the matrix corresponding to rows of A). we'll use NMF to compress it, not do classification. First obtain A:

```
> library(pixmap)
# read file
> mtr <- read.pnm('MtRush.pgm')
> class(mtr)
[1] "pixmapGrey"
attr(,"package")
[1] "pixmap"
# mtr is an R S4 object of class "pixmapGrey"
# extract the pixels matrix
> a <- mtr@grey
```

Now, perform NMF with rank 50, find the approximation to A, and display it:

```
> aout <- nmf(a,50)
> w <- aout@fit@W
> h <- aout@fit@H
> approxa <- w %*% h
# brightness values must be in [0,1]
> approxa <- pmin(approxa,1)
> mtrnew <- mtr
> mtrnew@grey <- approxa
> plot(mtrnew)  # dispatched to plot.pixmapGrey()
```

Here is the result:



This is somewhat blurry. The original matrix has dimension 194×259, and thus presumably has rank 194.[5] We've approximated the matrix by one of rank only 50. We could use this for compression, and if we had millions of images and this amount of blurriness were acceptable, we could take this approach.

Actually, there are better ways to compress images, and this was just an illustration of the effect of the reduced rank. Getting back to our classification context, the point is:

- When we use the term *low-rank approximation*, it is indeed approximate, as can be seen by the blurriness.
- Applying NMF or PCA/SVD to a whole collection of images, e.g. MNIST, further heightens this approximate nature of the process.
- But we need to do something to avoid overfitting, i.e. some kind of dimension reduction, and finding a low-rank approximation does that.

5.8 The Bias vs. Variance Tradeoff

The blurriness in that second picture is really an issue of bias, as follows. Consider a given pixel, say in the 3rd row and 52nd column. That pixel's intensity in the second picture will be a weighted

[5] One could check this by finding the number of nonzero eigenvalues of $A'A$, say by running **prcomp()**.

average of various pixels in the first picture. Some of the latter may be in locations within the picture that are somewhat far away from the 3rd row and 52nd column. This biases the pixel in the second picture.

On the other hand, there definitely is a variance issue. Let's review what this entails.

Recall from Chapter 4 that an intuitive way to view the variance issue in overfitting is that our data are being "shared" by the various things we're estimating, so that in a rough sense, each of these things has less data to itself. Less data means more sample-to-sample variability, i.e. higher variance. In linear regression with p features, we are estimating $p + 1$ parameters (including β_0); the larger p is, the larger the variance of the estimated β_i. Thus in turn we get larger variance to our predicted values. For predicting a new case, different samples will give us different predictions, and larger p will give us higher variance in our predicted value for that case.

Let n and m denote the number of rows and columns in A. Then W and H will be of dimensions $n \times k$ and $k \times m$. Well, then, how many parameters are we estimating? It's

$$nk + km = k(n + m) \tag{5.6}$$

So, the larger we make k, the larger the variance.

In other words, in predicting a specific A_{ij}, our predicted value $\widehat{A}_{ij}$ will experience this tradeoff:

- Larger k means lesser bias in our estimate of $\widehat{A}_{ij}$.
- Larger k means greater variance in $\widehat{A}_{ij}$.

5.9 Computation

How are the NMF solutions found? What is **nmf()** doing internally?

Needless to say, the methods are all iterative, with one approach being that of the Alternating Least Squares algorithm AltLS). It's quite intuitive, builds on our previous material and provides insight into NMF itself.[6]

And most importantly — AltLS is easily adapted to the recommender systems setting. Remember, recommender systems differ fundamentally from, say, the image and text classification applications cited earlier, due to the fact that some, typically the vast majority, of elements of the A matrix are unknown.

So let's take a look, still assuming for now that A *is* competely known.

[6] By the way, Alt. Least Squares is not the default for **nmf()**. To select it, set **method = 'snmf/r'**.

5.9.1 Objective Function

We need an *objective function*, a criterion to optimize, in this case a criterion for goodness of approximation. Here we will take that to be the *Frobenius* norm (Section 2.4),

$$\|Q\|_2 = \sqrt{\sum_{i,j} Q_{ij}^2} \tag{5.7}$$

So our criterion for error of approximation will be

$$\|A - WH\|_2 \tag{5.8}$$

So, we choose W and H to be

$$\arg\min_{w,h} \|A - wh\|_2 \tag{5.9}$$

This measure is specified in **nmf()** by setting **objective = 'euclidean'**.

Note that we can write (5.7) as

$$\|Q\|_2 = \sqrt{\sum_j \left(\sum_i Q_{ij}^2 \right)} \tag{5.10}$$

This mean that if we can separately minimize that inner sum, for each j, we will have minimized the entire expression (5.10). Our strategy will depend on this.

5.9.2 Alternating Least Squares

So, how does Alternating Least Squares work? Suppose just for a moment that we know the exact value of W, with H unknown. Then for each j we could minimize

$$\|A_{\cdot j} - WH_{\cdot j}\|_2 \tag{5.11}$$

We are seeking to find $H_{\cdot j}$ that minimizes (5.7), with $A_{\cdot j}$ and W known. But since the Frobenius norm is just a sum of squares, that minimization is just a least-squares problem, i.e. linear regression, just as in Section 5.16. We are "predicting" $A_{\cdot j}$ from W,

So again in the notation of Section 3.4.5:

- The matrix A there is our W here, known.
- The vector D there is our $A_{\cdot j}$ here, known.
- The vector b there is our $H_{\cdot j}$ here, unknown and to be solved for.

So we compute

```
> h[,j] <- lm(a[,j] ~ w - 1)$coef
```

for each j.[7]

On the other hand, suppose we know H but not W. We could take transposes,

$$A' = H'W' \tag{5.12}$$

and then just interchange the roles of W and H above. Here a call to **lm()** gives us a column of W', thus a row of W, and we do this for all rows.

Putting all this together, we first choose initial guesses, say random numbers, for W and H; **nmf()** gives us various choices as to how to do this. Then we alternate: Compute the new guess for W assuming H is correct, then choose the new guess for H based on that new W, and so on.

During the above process, we may generate some negative values. If so, we simply truncate to 0.

5.9.3 Back to Recommender Systems: Dealing with the Missing Values

In our recommender systems setting, of course, most of A is missing. But we can easily adapt to that. Roughly speaking, in (5.11), do these replacements:

- replace $A_{\cdot j}$ by the known portion of $A_{\cdot j}$
- replace W by the corresponding rows of W

Then proceed as before.

Here is a little example. Say A ix 5×5 and we want rank 3. Then W and H are of sizes 5×3 and 3×5.

Note too that $(WH)_{\cdot j}$, thus column j of our approximation to A, is a linear combination of the columns of W, with coefficients being $H_{\cdot j}$.

[7] The -1 specifies that we do not want a constant term in the model.

Suppose

$$A_{.4} = \begin{pmatrix} NA \\ 3 \\ NA \\ 8 \\ 2 \end{pmatrix} \tag{5.13}$$

Then in (5.11) we replace $A_{.5}$ by

$$\begin{pmatrix} 3 \\ 8 \\ 2 \end{pmatrix} \tag{5.14}$$

Also, replace W by

$$\begin{pmatrix} w_{21} & w_{22} & w_{23} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix} \tag{5.15}$$

Remember, at this stage, W is assumed known. So, we just use **lm()**, "predicting" (5.14) from (5.15) to find $h_{.4}$.

5.9.4 Convergence and Uniqueness Issues

There are no panaceas for applications considered here. Every solution has potential problems. I like to call this the Pillow Theorem — pound down on one fluffy part and another part pops up.

Unlike the PCA case, one issue with NMF is uniqueness — there might not be a unique pair (W, H) that minimizes (5.8).[8] In fact, one can see this immediately: Doubling W while having H leaves the product WH unchanged. Of course, the product is all that really counts, but in turn, this may result in convergence problems. The NMF documentation recommends running **nmf()** multiple times; it will use a different seed for the random initial values each time.

The Alternating Least Squares method used here is considered by some to have better convergence properties, since the solution at each iteration is unique. This may come at the expense of slower convergence.

[8] See Donoho and Stodden, *When Does Non-Negative Matrix Factorization Give a Correct Decomposition into Parts?*, `https://web.stanford.edu/~vcs/papers/NMFCDP.pdf`.

5.10 How Do We Choose the Rank?

This is not an easy question. One approach would be to use **prcomp**, or for that matter **svd()**, to find the eigenvalues, then take our rank to be the number of "large" eigenvalues, as discussed in Chapter 2.

Of course, the typical way rank is chosen is cross validation.

5.11 Why Nonnegative?

In the applications we've mentioned here, we always have $A_{ij} \geq 0$. However, that doesn't necesarily mean that we need W and H to be nonnegative, and indeed if we were to use PCA, they may not so. (With PCA, even their product could have negative element, which we would truncate to 0.) Why use NMF, i.e. why insist that the factors W and H themselves be nonnegative?

There are a couple of reasons NMF may be preferable. First, truncation may be questionable if we have a lot of negative values. But the second reason is that NMF may be more useful, as follows:

In a facial image recognition case, say, there is hope that the vectors $W_{\cdot j}$ will be *sparse*, i.e. mostly 0s. Then we might have, say, the nonzero elements of $W_{\cdot 1}$ correspond to eyes, $W_{\cdot 2}$ correspond to nose and so on with other parts of the face. We are then "summing" to form a complete face. This may enable effective *parts-based recognition*, with helpful interpretations.

In our recommender systems setting, this parts-based effect, NMF would give us crisper distinction among the various synthetic users. This may reveal clusters of user behavior, which could be quite helpful to the analyst.

Another point is that the nonnegativity allows us to better fulfill the "synthetic users" idea from Section 5.4. To make these more realistic, they should be on the same level as real user ratings. We can arrange this by simply scaling down each $H_{i\cdot}$ to the ratings scale, e.g. 1 to 5 for the MovieLens data.

5.12 "Bias" Removal

As noted in Chapter 6, some users tend to give more liberal ratings, while others tend to be more cautious. Similarly, some items tend to be rated more highly than others. One way of dealing with that is to adjust our ratings matrix A accordingly: For each user i, let R_i denote the average of all known ratings from that user. Also, for each item j let S_j denote the average of all known ratings

for that item. These are termed *biases*. Then do the replacement

$$A_{uv} \leftarrow A_{uv} - R_u - S_v \tag{5.16}$$

and then form the matrix factorization as usual. In the end, after estimating the unknown entries in A, restore the subtracted quantities:

$$A_{uv} \leftarrow A_{uv} + R_u + S_v \tag{5.17}$$

5.13 Dealing with Covariates

Why stop with just removing "biases"? We can go a step further and account for user or item covariates.

The easiest approach to handling covariates is to simply "subtract them out" for the data via linear regression, then run NMF/SVD on the resulting *residuals*,[9] component in the S3 object returned by **lm().** and finally, at the prediction stage, add the regression values back in.

Here are relevant code excerpts:

trainReco():

```
hasCovs <- (ncol(ratingsIn) > 3)
if (hasCovs) {
    covs <- as.matrix(ratingsIn[, -(1:3)])
    lmout <- lm(ratingsIn[, 3] ~ covs)
    # now make fake ratings; for NMF, must be >= 0
    minResid <- min(lmout$residuals)
    ratingsIn[, 3] <- lmout$residuals - minResid
}
...
r$train(train_set, opts = list(dim = rnk, nmf = nmf))
result <- r$output(out_memory(), out_memory())
attr(result, "hasCovs") <- hasCovs
if (hasCovs) {
    attr(result, "covCoefs") <- coef(lmout)
    # add the residuals back in
    attr(result, "minResid") <- minResid
```

[9] In regression modeling, the values of true Y minus the fitted model are called "residuals." They are often used to assess quality of model fit. There is a **residuals**

```
}
class(result) <- "RecoS3"
result
```

predict.RecoS3():

```
if (hasCovs) {
     tmp <- c(1, testCovs[i, ]) %*% covCoefs + minResid
}
pred[i] <- p[j, ] %*% q[k, ] + tmp
```

In the covariates case, **trainReco()** replaces the ratings by the residuals, so the product WH approximates them rather than the original A. Then in **predict.RecoS3()**, after multiplying $W_{i.}$ and $H_{.j}$, the estimated regression function value for the given case is added back in.

The role of **minResid** is this: If we are using NMF, fitting to the residuals, which are both positive and negative, makes no sense. So, we subtract the (algebraically) smallest residual, resulting in only nonnegative values. Again, this is added back in later on.

5.14 Regularization

Recall Section 4.5, where we introduced the idea of shrinkage as a guard against overfitting.

For NMF (or SVD), we probably don't want to force some predicted ratings to 0, the l_2 norm is a popular choice. Thus, instead of choosing W and H to minimize (5.8), we minimize

$$||A - WH||_2 + \gamma_1 ||W||_2^2 + \gamma_2 ||H||_2^2 \tag{5.18}$$

Both γ_1 and γ_2 are tuning parameters.

The **NMF** package offers regularization as an option.

5.15 The recosystem Package

The **recosystem** package does matrix factorization specifically for recommender systems, i.e. specifically for settings in which the matrix A has many missing values. It's written by experts in numerical matrix factorization, and features a number of useful options.

Below is a **recosystem** session using the small MovieLens data. Let's suppose we've already decided on rank $k = 20$, say by cross validation, and now we'll go back to using the full dataset for

our predictions.

```
> library(recosystem)
# all action will take place within r;
# typically the output of a function will be stored as a new component in r
> r <- Reco()
> ml <- read.table('u.data',header=F)
# need to create an object of class 'DataSource', specifying which
# columns are user IDs, item IDs and ratings
> ml.dm <- data_memory(ml[,1],ml[,2],ml[,3],index1=TRUE)

# do the factorization, with rank 20; do use NMF
> r$train(ml.dm,opts=list(dim=20,nmf=TRUE))
iter      tr_rmse          obj
   0       2.0381   5.0056e+05
   1       1.0296   1.7402e+05
   2       0.9529   1.6028e+05
   3       0.9449   1.5868e+05
   4       0.9418   1.5811e+05
   5       0.9397   1.5774e+05
   6       0.9382   1.5749e+05
   7       0.9371   1.5729e+05
   8       0.9362   1.5713e+05
   9       0.9355   1.5701e+05
  10       0.9348   1.5690e+05
  11       0.9343   1.5681e+05
  12       0.9338   1.5673e+05
  13       0.9334   1.5666e+05
  14       0.9330   1.5660e+05
  15       0.9327   1.5654e+05
  16       0.9324   1.5649e+05
  17       0.9321   1.5645e+05
  18       0.9318   1.5641e+05
  19       0.9316   1.5637e+05
# training went for 20 iterations; RMSE is the square root
#    of mean squared error
# for large data, write to disk, otherwise in memory
> result <- r$output(out_memory(),out_memory())
> str(result)
List of 2
 $ P: num [1:943, 1:20] 0.676 0.677 0.574 0.836 0.574 ...
```

```
 $ Q: num [1:1682, 1:20] 0.712 0.614 0.568 0.645 0.612 ...
# P and Q are W and H'
> w <- result$P
> h <- t(result$Q)
# let's try a prediction, with a known rating
> head(ml)
   V1  V2 V3        V4
1 196 242  3 881250949
2 186 302  3 891717742
3  22 377  1 878887116
...
> w[22,] %*% h[,377]
         [,1]
[1,] 2.196976
# or just have recosystem do it for us
> preds <- r$predict(ml.dm,out_memory())
> head(preds)
[1] 3.979107 4.212397 2.196976 3.601082 3.900878 4.467487
```

5.16 How Do We Minimize (5.4)?

The answer, as it was in Section 5.9, is to convert the question to one of linear regresson.

It will be helpful to keep some concrete numbers in mind, say with the MNIST data. Say we wish a rank of 50. Then

- A is 70000x784;
- W is 70000x50;
- H is 50x784;
- S is 1x784; and
- l is 50x1.

Keeping these numbers in mind as concrete examples, now note that in (5.4),

$$||S - l'H||^2 = (S - l'H)(S - l'H)' \tag{5.19}$$

This looks pretty close to (3.15). But recall that S and $l'H$ are row vectors, so (5.19) looks slightly different. Now, using the fact from linear algebra that $(UV)' = V'U'$, (5.19) says

$$||S - l'H||^2 = (S' - H'l)(S' - H'l) \tag{5.20}$$

Now it's in the form of (3.15), so our minimization problem is solved! In (3.20), just set D, A and b to S', H' and l, respectively. This gives us

$$\lambda = (HH')^{-1}HS' \tag{5.21}$$

We could compute this directly, using R's matrix operations (for matrix inversion, we can use **solve()**, or for better numerical accuracy, **solve.qr()**), but it's easier to send it to **lm()**, e.g.

```
lm(s ∼ t(h) - 1)$coef
```

Again, the '-1" tells **lm()**, "Don't add a column of 1s to H'."

Chapter 6

Neighborhood-Based Methods

One of the simplest and yet often most effective recommender system methods is based on this natural principle:

> Say we have a user U, for whom we want to predict the rating of an item I. We find the users in our existing data D who are most similar to U and who have seen rating I, and take our predicted value to be the average of those users' ratings of I.

Note that here the user U might be in D or might be new. As long as I is in D, we are in business.

Of course, we must define "similar." There are two common ways to do this. Recall our notation p denoting our number of predctors/features. This would include our user and item IDs, and possible covariates. Then consider these approaches:

- Define some distance function, and then find the k closest people in D to U.
- Develop a system of rectangles — hyperrectangles in p-dimensional space — and determine which one U falls in.

The first is basically *k-Nearest Neighbor regression* (kNN), a classic statistics/machine learning technique, though note that a major difference here is that we only consider users in D who have rated the same products as N.

The second can be viewed as something called *kernel regression*, but we will add a tree-based structure, with the result termed *Classification and Regression Analysis* (CART) for a single tree and *random forests* for a collection of trees. In the latter, we create a collection of systems of rectangles and average over them.[1]

[1]The terms "CART" and "random forests" have many variations, but we take the terms as generic.

6.1 kNN

Let's see how kNN works.

6.1.1 Notation

As before, let A denote the ratings matrix. The element a_{ij} in row i, column j, is the rating that user i has given/would give to item j. In the latter case, a_{ij} is unknown, and its predicted value will be denoted by $\widehat{a}_{ij}$. Following R notation, we will refer to the unknown values as NAs.

Note that for large applications, the matrix A is far too large to store in memory. One could resort to storage schemes for *sparse* matrices, e.g. *Compresed Row Storage*, but here we will simply use A to help explain concepts. In the **rectools** package,[2] the input data is run through **formUserData()** and algorithms use that instead of A. This function organizes the data into an R list, one element per user. Each such element records the ratings made by that user.

Let's refer to a new case to be predicted as NC, i.e. from above, predicting how a user U would rate an item I.

6.1.2 User-Based Filtering

In predicting how a given user would rate a given item, we first find all users that have rated the given item, then determine which of those users are most similar to the given user. Our prediction is then the average of the ratings of the given item among such "similar" users. A corresponding approach based on similar items, *item-based filtering*, is used as well. We focus on such methods in this chapter.

6.1.3 (One) Implementation

Below is code from **rectools** (somewhat simplified).[3] The arguments are:

- **origData:** The original dataset, after having been run through **formUserData()**.
- **newData:** The element of **origData** for NC.[4]
- **newItem:** ID number of the item to be predicted for NC.

[2] https://github.com/matloff/rectools

[3] This function was written largely by Vishal Chakraborti.

[4] If NC is new, not in the database (called *cold start*), we synthesize a list element for it, assuming NC has rated at least one item.

- k: The number(s) of nearest neighbors. Can be a vector.

Here is an example of using **formUserData()** on the MovieLens data:[5]

```
> head(ml)
V1  V2 V3
1 196 242  3
2 186 302  3
3  22 377  1
4 244  51  2
5 166 346  1
6 298 474  4
> mlud <- formUserData(ml)
> mlud[[3]]
$userID
[1] "3"

$itms
[1] 335 245 337 343 323 331 294 332 328 334 350 341 318 300
[15] 345 299 324 348 351 330 327 307 272 354 264 349 321 260
[29] 268 288 355 320 258 339 342 303 329 317 181 338 302 322
[43] 352 271 333 344 326 319 325 347 336 353 340 346

$ratings
335 245 337 343 323 331 294 332 328 334 350 341 318 300 345
1   1   1   3   2   4   2   1   5   3   3   1   4   2   3
299 324 348 351 330 327 307 272 354 264 349 321 260 268 288
3   2   4   3   2   4   3   2   3   2   3   5   4   3   2
355 320 258 339 342 303 329 317 181 338 302 322 352 271 333
3   5   2   3   4   3   4   2   4   2   2   3   2   3   2
344 326 319 325 347 336 353 340 346
4   2   2   1   5   1   1   5   5

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

So, for any given user, **mlud** will show the items rating by this user and the ratings the user has given to those items. Here we see that user 3 has rated items 335, 245,, 337, 343,..., with ratings 1,1,1,3,...

[5]The data have been read from disk without converting to R factors.

```
1  predict.usrData <- function(origData,newData,newItem,k)
2  {
3  # we first need to narrow origData down to the users who
4  # have rated newItem
5
6  # here oneUsr is one user record in origData; the function will look for a
7  # j such that element j in the items list for this user matches the item
8  # of interest, newItem; (j,rating) will be returned
9
10 checkNewItem <- function(oneUsr) {
11    whichOne <- which(oneUsr$itms == newItem)
12    if (length(whichOne) == 0) {
13       return(c(NA,NA))
14    } else return(c(whichOne,oneUsr$ratings[whichOne]))
15 }
16
17 found <- as.matrix(sapply(origData,checkNewItem))
18 # description of 'found':
19 # found is of dimensions 2 by number of users in training set
20 # found[1,i] = j means origData[[i]]$itms[j] = newItem;
21 # found[1,i] = NA means newItem wasn't rated by user i
22 # found[2,i] = rating in the non-NA case
23
24 # we need to get rid of the users who didn't rate newItem
25 whoHasIt <- which(!is.na(found[1,]))
26 origDataRatedNI <- origData[whoHasIt]
27 # now origDataRatedNI only has the relevant users, the ones who
28 # have rated newItem, so select only those columns of the found matrix
29 found <- found[,whoHasIt,drop=FALSE]
30
31 # find the distance from newData to one user y of origData; defined for
32 # use in sapply() below
33 onecos <- function(y) cosDist(newData,y,wtcovs,wtcats)
34 cosines <- sapply(origDataRatedNI,onecos)
35 # the vector cosines contains the distances from newData to all the
36 # original data points who rated newItem
37
38 # action of findKnghbourRtng(): find the mean rating of newItem in
39 # origDataRatedNI, for ki (= k[i]) neighbors
40 #
```

```
41 # if ki > neighbours present in the dataset, then the
42 # number of neighbours is used
43 findKnghbourRtng <- function(ki){
44   ki <- min(ki, length(cosines))
45   # nearby is a vector containing the indices of the ki closest neighbours
46   nearby <- order(cosines,decreasing=FALSE)[1:ki]
47   mean(as.numeric(found[2, nearby]))
48 }
49 sapply(k, findKnghbourRtng)
50 }
```

6.1.4 Not Really a Distance

Note that the distances were computed by the function **cosDist()**, which computes a "cosine" similarity:

```
find cosine distance between x and y, objects
# of 'usrData' class
#
# only items rated in both x and y are used; if none
# exist, then return NaN
#
#  wtcovs: weight to put on covariates; NULL if no covs
#  wtcats: weight to put on item categories; NULL if no cats

cosDist <- function(x,y,wtcovs=NULL,wtcats=NULL)
{
# rated items in common
commItms <- intersect(x$itms,y$itms)
if (length(commItms)==0) return(NaN)
# where are those common items in x and y?
xwhere <- which(!is.na(match(x$itms,commItms)))
ywhere <- which(!is.na(match(y$itms,commItms)))
xvec <- x$ratings[xwhere]
yvec <- y$ratings[ywhere]
if (!is.null(wtcovs)) {
   xvec <- c(xvec,wtcovs*x$cvrs)
   yvec <- c(yvec,wtcovs*y$cvrs)
}
if (!is.null(wtcats)) {
```

```
   xvec <- c(xvec,wtcats*x$cats)
   yvec <- c(yvec,wtcats*y$cats)
}

xvec %*% yvec / (l2a(xvec) * l2a(yvec))
}

l2a <- function(x) sqrt(x %*% x)
```

Basically, the "distance" between two rows u and v of A is defined by

$$\frac{u'v}{||u||_2 \; ||v||_2} \tag{6.1}$$

This not really a distance,[6] but it is a common measure of similarity between two vectors in machine learning. In two or three dimensions, it really is the cosine of the angle between u and v.

Note that larger cosines mean the vectors are more similar. We find the k most similar rows in D to U, and average their ratings of the given item.

6.1.5 Regression Analog

Recall the method of k-nearest neighbor (kNN) regression estimation from Chapter 3, involving prediction of weight from height and age:

> 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 $E(W \mid H = 70, A = 28)$.

So kNN RS is really the same as kNN regression

6.1.6 Choosing k

As we have already seen with RS, regression and machine learning methods, the typical way to choose a model is to use cross-validation. This is true for kNN RS as well; we can choose the value of k via cross-validation.

[6] IN math terms, it's not a *metric*.

6.1.7 Item-Based Filtering

Consider again our setting in which we wish to predict the rating user U would give to item I. We could switch the above procedure, trading rows for columns. We would find the columns corresponding to items U has rated, then find the closest k of those columns to column I. The ratings given by U in those closest column would then be averaged to yield our prediction.

6.1.8 Covariates

To accommodate covariates, we simply add covariate columns to the input matrix, say now with columns 'userId', 'itemId', 'rating' and age'. Note that they figure into the distance measure, just like the user and item I dummies.

6.2 CART and Random Forests

CART is based on forming a *recursive partitioning* of the data space. First the space is split in two, then each of the two parts is split in two, and so on, forming a binary tree. Splitting along a branch stops when some criterion is no longer met.

6.2.1 Motivating Example

Let's illustrate CART with the R package **partykit**.[7] We'll predict weight from height and age in the baseball player data.

```
> library(regtools)
> data(mlb)
> mlb <- mlb[,c(4:6)]
> head(mlb)
  Height Weight   Age
1     74    180 22.99
2     74    215 34.69
3     72    210 30.78
4     72    210 35.43
5     73    188 35.71
6     69    176 29.39
> ctout <- ctree(Weight ~ .,data=mlb)
```

[7] The name is an allusion to the recursive partioning nature of CART.

Here is the tree that is produced:

```
> node_party(ctout)
[1] root
|   [2] V2 <= 73
|   |   [3] V2 <= 70
|   |   |   [4] V3 <= 29.56 *
|   |   |   [5] V3 > 29.56 *
|   |   [6] V2 > 70
|   |   |   [7] V3 <= 32.51
|   |   |   |   [8] V2 <= 72 *
|   |   |   |   [9] V2 > 72 *
|   |   |   [10] V3 > 32.51 *
|   [11] V2 > 73
|   |   [12] V2 <= 75
|   |   |   [13] V3 <= 27.55
|   |   |   |   [14] V2 <= 74 *
|   |   |   |   [15] V2 > 74 *
|   |   |   [16] V3 > 27.55 *
|   |   [17] V2 > 75
|   |   |   [18] V2 <= 79
|   |   |   |   [19] V3 <= 26.03 *
|   |   |   |   [20] V3 > 26.03 *
|   |   |   [21] V2 > 79 *
```

We'll look more at the tree shortly, but first suppose we wish to predict the weight of a player who is 72 inches tall, age 31.

```
> predict(ctout,data.frame(Height=72,Age=31))
       1
189.9091
```

Where did that come from? Look at the tree. Height $\leq$ 73? Yes, so go to node 3. Height $\leq$ 70? No, go to node 6, etc., winding up at node 8. The asterisk indicates a leaf node, so we're done traversing the tree.‘ No, what is the mean weight in that node?

```
> ht <- mlb$Height
> age <- mlb$Age
> mean(mlb[70 < ht & ht <= 72 & age <= 32.51,]$Weight)
[1] 189.9091
```

6.2.2 Use in Recommender Systems

Clearly, it would be difficult to use CART directly on, say, the MovieLens data. Since the user and item IDs are not *ordinal*, i.e. do not have an inherent underlying ordering, we'd need to form dummy variables, and thus test for one of them at a time. The tree would be greatly unbalanced, to the right, and we'd have a computational problem, at the least.

But actually, it's a lot worse than that. Think of a node "User ID = 168?" In cases where we take the left branch, i.e. the data point is indeed for user 168, the next question might be, say, Item ID = 12?" The key point is that **there will be at most 1 data point satisfying both conditions.** That's not enough to make a good tree. And indeed, CART software typically sets a minimum node size as a hyperparameter; for **ctree(), it's minsplit**.

An alternative is to do an *embedding* of the user and item ID data, a form of dimension reduction. What we could do here is replace each user ID by the mean rating given by that user, and do the same for the movies:

```
> userMeans <- tapply(ml$V3,ml$V1,mean)
> itemMeans <- tapply(ml$V3,ml$V2,mean)
> mlemb <- mlb
> mlemb$V1 <- userMeans[ml$V2]
> mlemb$V2 <- itemMeans[ml$V2]
> mlemb$V1 <- as.vector(mlemb$V1)
> mlemb$V2 <- itemMeans[ml$V2]
> mlemb$V2 <- as.vector(mlemb$V2)
> ctout <- ctree(V3 ~ .,data=mlemb)
```

We could then use **predict()** as before.

6.2.3 Tuning Parameters

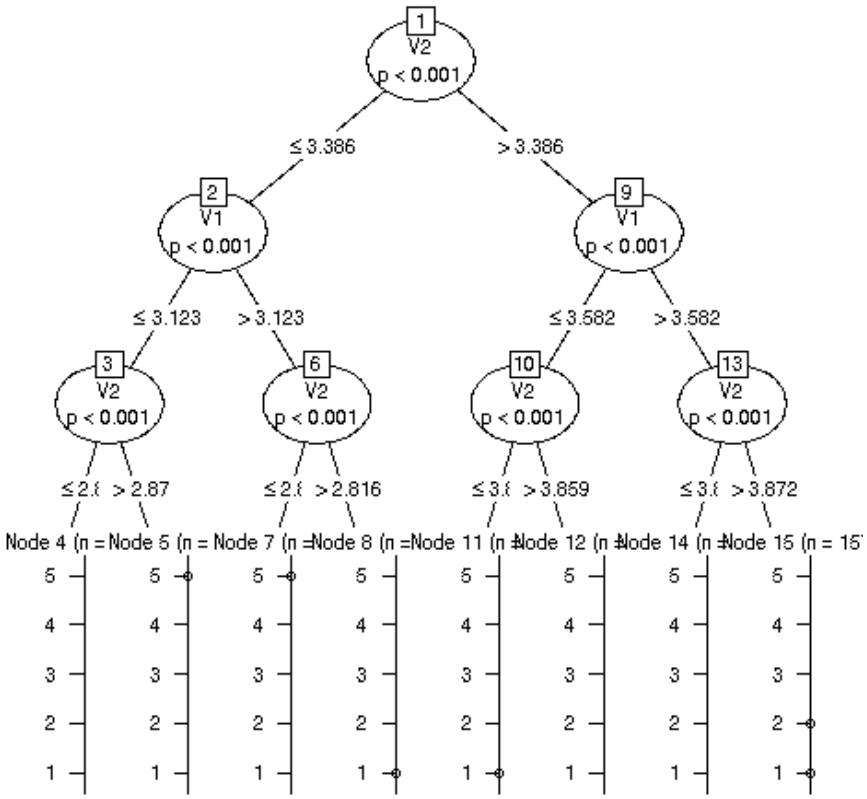
The **ctree()** function has various tuning parameters. We saw one of them above, **minsplit**. Another is **maxdepth**, the maximum number of levels we wish to allow in the tree. The default value, 0, means no limit. You can see in the output above the **ctree()** built us a 5-level tree. The tuning parameters must be set via the function **ctree_control()**.

The return value of **ctree()** has class, of course, **'party'**. It has various generic functions. We saw **predict()** above; **print()** gives output similar to, but a little more detailed than, **node_party()**.

Another is **plot()**, which due to screen space issues, is only usable for a few levels. Here we set **maxdepth** to 3,

```
> ctout <- ctree(V3 ~ .,data=mlemb,control = ctree_control(maxdepth=3))
> plot(ctout)
```

l



Actually, the plotting comes from the fact that **ctout** is also of class **'BinaryTree'**

6.2.4 Covariates

Covariates are easily handled, e.g.

```
ctout <- ctree(V3 ~ V1+V2+age+gender,data=mlemb)
```

6.2.5 Random Forests

Some years after usage of CART became widespread, there was concern that the procedure was too sensitive to small changes in the data. Consider the root node, for instance. If the data is changed slightly, that first split may change somewhat, with the change cascading down the entire tree. In other words, the tree is quite subject to sampling variation, meaning in turn that the predicted value of any particular future case will have a higher variance.

The solution was to generate many trees, and combine the results. New trees are generated by resampling from our original data: For a dataset of n data points, we randomly sample n times, *with replacement*, and run CART on that new sample. We then predict a new case by obtaining a prediction from each tree, then aggregating the predictions: In a regression setting, we average all the predicted values, while in a classification setting, our final prediction is whichever class was predicted most often aomong the various trees. The resampling is an example of the *bootstrap*, so the entire process is called *bootstrap aggregating*, or *bagging*.

The original idea for all this came from Tin Kam Ho. Leo Breiman, one of the developers of CART, refined it and coined the term *random forests*.

The **partykit** package includes the function **cforest()** for this technique.

Chapter 7

Statistical Models

Recommender systems is inherently statistical. Indeed, the very fact that we discuss the bias-variance tradeoff recognizes the fact that our data are subject to sampling variation, a core statistical notion. In this chapter, we will apply classical statistical estimation methods to a certain *latent variables* model.

7.1 The Basic Model

Again, for concreteness, we'll speak in terms of user ratings of movies. Let (U, I) denote a random (user ID, movie ID) pair. Let u and m denote the numbers of users and movies. Denote the user's rating by Y_{IJ}. The model is additive, postulating that

$$Y_{IJ} = \mu + \alpha_I + \beta_J + \epsilon_{IJ} \tag{7.1}$$

Here μ is an unknown constant, the overall population mean over all users and all movies. The numbers $\alpha_1, \alpha_2, ..., \alpha_u$ and $\beta_1, \beta_2, ..., \beta_m$ are also unknown constants; think of α_i to be the tendency of user i to give harsher ($\alpha_i < 0$) or more generous ($\alpha_i > 0$) ratings, relative to the general population of users, with a similar situation for the β_j and movies. The ϵ term is thought of as the combination of all other affects.

Note that what makes, e.g., α_I random above is that I is random, and similarly for the β_J and ϵ_{IJ}. The α, β and ϵ terms are assumed to be statistically independent, each with mean 0.

So, we model a user's rating of a movie as the sum of latent additive user and movie terms, plus a catch-all "everything else" term.[1] The question then becomes how to estimate μ, and $\alpha_1, \alpha_2, ..., \alpha_u$

[1] What does the word *latent* here mean? Why is μ not "latent"? The answer is that it is a tangible quantity;

and $\beta_1, \beta_2, ..., \beta_m$, where u and m are the numbers of users and movies in our data. We will present two methods.

7.2 Two General Statistical Methods for Parameter Estimation

We'll be using two famous estimation tools from statistics, the Method of Moments and Maximum Likelihood Estimation. We'll introduce those in this section.

7.2.1 Example: Guessing the Number of Coin Tosses

To aovid distracting complexity, consider the following game. I toss a coin until I accumulate a total of r heads. I don't tell you the value of r that I used, only informing you of K, the number of tosses I needed.

It can be shown that

$$P(K = u) = \binom{u-1}{r-1} 0.5^u, \ u = r, r+1, ... \tag{7.2}$$

Say I play the game 3 times, and I tell you $K = 7, 10$ and 9. What could you do to try to guess r?

Notation: We play the game n times, always with the same r, yielding $K_1, K_2, ..., K_n$.

7.2.2 The Method of Moments

The *moments* of a random variable X are the expected values of the powers. E.g. $E(X^3)$ is called the third moment of X.

If we are trying to estimate s parameters, $\theta_1, ..., \theta_s$, we need s moments. We find population expressions for the θ_i in terms of the first s moments of the random variable at hand, setting up s equations that match those expressions to the estimated parameters, $\widehat{\theta}_1, ..., \widehat{\theta}_s$, then solve for the latter, then solve for the latter

Here we have just one parameter, r. It can be shown that in the game example,

$$E(K) = \frac{r}{0.5} = 2r \tag{7.3}$$

we all can imagine finding the overall mean for all users and movies, given enough data. By contrast, the α values' existence depend on the validity of the model. It's similar to the NMF situation, where the postulate postulates existence of a set of "typical" users.

MM involves replacing both sides of an equation like (7.3) by sample estimates, in this case

$$\overline{K} = 2\widehat{r} \tag{7.4}$$

where

$$\overline{K} = \frac{K_1 + ... + K_n}{n} \tag{7.5}$$

and $\widehat{r}$ is our estimate of r.[2]

So the idea of MM is:

1. Find theoretical (i.e. population-level) equations for various expected values, enough to cover the number of parameters being estimated.
2. In those equations, replace expected values and parameters by sample estimates.
3. Solve for the sample estimates.

7.2.2.1 The Method of Maximum Likelihood

To guess r in the game, you might ask, "What value of r would make it most likely to need 7 tosses to get r heads?" You would then find the value of w that maximizes the *likelihood*, defined to be the probability of our observed data under a given value of the parameter(s), in this case

$$\Pi_{i=1}^{n} \binom{K_i}{w-1} 0.5^{K_i} \tag{7.6}$$

In this discrete case you could not use calculus, and simply would use trial-and-error to find the maximizing value of w, which will be our $\widehat{r}$.

7.2.2.2 Comparison: MM vs. MLE

If these two methods were nervous academics, MM would be quite envious of MLE:

- MLE is by far the more widely-used method.

[2]It is standard to use the "hat" symbol to mean "estimate of."

- MLE can be shown to be optimal in a certain sense. (Roughly, it has the smallest possible variance of all estimators, when n is large.)
- Various aspects of MLE and related topics are famous enough to be named after people, e.g. Fisher information (yes, the significance testing Fisher) and the Cramer-Rao lower bound.

On the other hand:

- Often MM makes fewer assumptions than MLE. That will be the case for us in the RS application below, a major point.
- MM is easier to explain. MLE has the same "What if...?" basis that p-values have, rather confusing.
- MM is actually the basis for the 2013 Nobel Prize in Economics! Lars Peter Hansen won the prize for his development of the Generalized Method of Moments estimation tool.

7.3 MM Applied to (7.1)

As you'll see, MM is arguably the more useful of the two methods in this particular setting.

7.3.1 Derivation of the Estimates

The expected values in Section 7.2.2 can be conditional. So, from (7.1), write

$$E(Y_{IJ} \mid I = k) = \mu + \alpha_k + E(\beta_J | I = k),\ k = 1, 2, ..., u \tag{7.7}$$

But since I and J are independent, we have

$$E(\beta_J | I = k) = E(\beta_J) = 0,\ k = 1, 2, ..., u \tag{7.8}$$

so

$$E(Y_{IJ} \mid I = k) = \mu + \alpha_k,\ k = 1, 2, ..., u \tag{7.9}$$

Now we must find our sample estimate of the left-hand side, and equate it to $\mu + \alpha_k$.

But the natural estimate of $E(Y_{IJ} \mid I = k)$ is simply the mean rating user k gave to all movies she rated.

Moreover, the natural estimate of μ is the average rating given to all movies in our data.

So we now have our $\widehat{\alpha}_k$. The derivation of the $\widehat{\beta}_l$ is similar.

7.3.2 Relation to Linear Model

For simplicity, consider the call

```
lm(rating ~ userID-1)
```

omitting the movies. Think of what will happen with the matrix A and the vector D in Section 3.4.5.

Recall that the -1 in the above call means we do not want an intercept term. In that case, **lm()** will produce u dummy variables rather than $u - 1$. This will help clarify the situation.

So, in the matrix A, column i will be the vector of 1s and 0s in the dummy for user i, $i = 1, ..., u$. Now consider the (i, i) element in $A'A$. It's the dot product of row i in A' and column i in A, thus the dot product of column i in A and column i in A. That will in turn be the sum of some 1s — actually, n_i 1s, where n_i is the number of ratings user i has made.

Meanwhile, the same reasoning says that for $i \neq j$, element (i, j) in $A'A$ is 0, since two dummy vectors coming from the same categorical variable will never have a 1 in the same position.

Putting all that together, we have that

$$(A'A)^{-1} = \text{diag}(\frac{1}{n_1}, ..., (\frac{1}{n_u}) \tag{7.10}$$

a diagonal matrix with the indicated elements.

What about $A'D$ in (3.4.5)? Similar reasoning shows that its m^{th} element is the sum of all the ratings given by user m.

Putting this all together, we find that the m^{th} estimated coefficient returned by **lm()** will be the average rating given by user m — exactly the same as MM gave us!

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=TRUE)
[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.