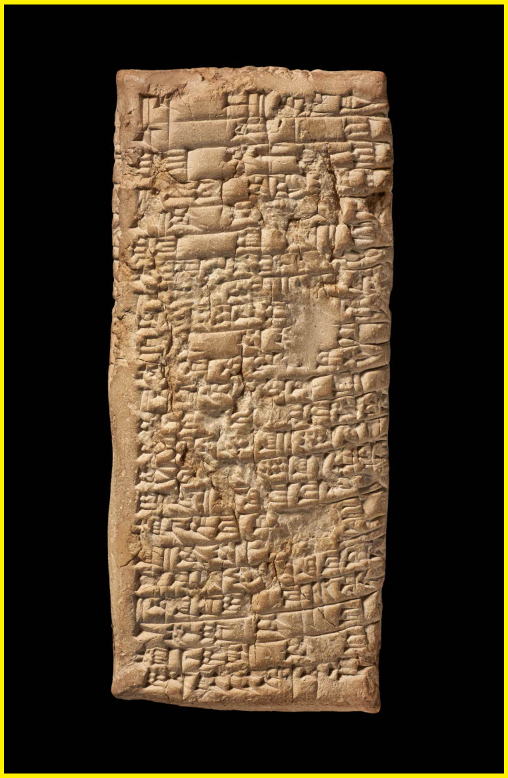
A Tour of Recommender Systems

Norm Matloff

University of California, Davis



Ancient "Yelp"

About 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), *Statistical Regression and Classification: from Linear Models to Machine Learning* (CRC 2017), and *Probability and Statistics for Data Science: Math+R+Data* (CRC 2019). The second book was the recipient of the Ziegal Award in 2017. His book with NSP, *The Art of Machine Learning: Algoorithms+Data+R*, will be published in 2022.

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 machine learning, fair learning, missing data methods, and data privacy.

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*, one of several common approaches to recommender systems (RTSs).

1.1 What Are Recommender Systems?

What is an 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 prescription 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 The "Hello World" of RS: MovieLens

This dataset is the standard introduction to RSs, and indeed is a standard example in RS research papers. It's available from the GroupLens project at the University of Minnesota, `grouplens.org`. In this book, we will mainly use the 100K version, which contains 100,000 rates in the format

```
userID  movieID  rating1to5  timestamp
```

Let's take a look:

```
> download.file('http://files.grouplens.org/datasets/movielens/ml-100k.zip',
>    'ml-100k.zip')
> unzip('ml-100k.zip')
> ml100 <- read.csv('ml-100k/u.data',header=FALSE,sep='\t')
> head(ml100)
   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
```

(Note that the data comes in TAB-separated fields.)

We see for instance user 22 rated movie 377 as 1. Let’s explore some more:

```
> length(unique(ml100$V1))
[1] 943
> length(unique(ml100$V2))
[1] 1682
> table(ml100$V3)

    1     2     3     4     5
 6110 11370 27145 34174 21201
```

So there were 943 users and 1682 films. Users seemed to be pretty liberal in their ratings, with the most popular rating being a 4.

We can break that down by user:

```
> tmp <- tapply(ml100$V3,ml100$V1,mean)
> tmp[22]
      22
3.351562
```

Here we grouped ratings according to age, computing the mean rating for each age value.

So for instance user 22 gave an average rating of about 3.35 to the ones he/she rated. And how many films was that?

```
> tmp <- tapply(ml100$V3,ml100$V1,length)
> tmp[22]
 22
128
```

Did user 22 rate movie 88?

```
> ml100[ml100$V1 == 22 & ml100$V2 == 88,]
[1] V1 V2 V3 V4
<0 rows> (or 0-length row.names)
```

No. And that is the essence of RS: How can we predict what rating user 22 would give to that movie?

We can represent our data as a matrix. Let's illustrate that with some functions from the `rectools` package, which we will be using in this book:

```
> ml100UD <- formUserData(ml100[,1:3])
> ml100A <- asMatrix(ml100UD)
> dim(ml100A)
[1]  943 1682
> ml100A[1:5,1:5]
     [,1] [,2] [,3] [,4] [,5]
[1,]    5    3    4    3    3
[2,]    4   NA   NA   NA   NA
[3,]   NA   NA   NA   NA   NA
[4,]   NA   NA   NA   NA   NA
[5,]    4    3   NA   NA   NA
```

The `formUserData()` function inputs data in the (userID, itemID, rating) format we've seen, and outputs an R list, one element for each distinct user. Then **asMatrix()** forms the ratings matrix from that list. Don't worry about the full call forms now, but just become familiar with the structure of the matrix. We see, for instance, that user 2 rated item 1 as a 4, but didn't rated movies 2-5.

1.3 NA Values Are Not So Inocuous

There are lots of ways to deal with *missing values*, i.e. NAs. In fact, many full books have been written. Some terminology has developed over years in terms of modeling how NAs arisew.

The simplest model is *Missing Completely at Random*, MCAR. As the name implies, this simply means that nature (or the data collection process) has randomly sprinkled NA values among the data.

Another, *Missing at Random*, MAR, is more complicated. Here's what one of the leading authors says (*https://stefvanbuuren.name/fimd/sec-MCAR.html*):

> ...While convenient, MCAR is often unrealistic for the data at hand.
>
> If the probability of being missing is the same only within groups defined by the observed data, then the data are missing at random (MAR). MAR is a much broader class than MCAR. For example, when placed on a soft surface, a weighing scale may produce more missing values than when placed on a hard surface. Such data are thus not MCAR. If, however, we know surface type and if we can assume MCAR within the type of surface, then the data are MAR. Another example of MAR is when we take a sample from

a population, where the probability to be included depends on some known property. MAR is more general and more realistic than MCAR. Modern missing data methods generally start from the MAR assumption.

RS data is in fact usually *not* MCAR. There may be a good reason a user hasn't rated a movie — she knows she wouldn't like it. Handling missing values is hard enough in general, with no easy answers, but this should be kept in mind.

1.4 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 for collaborative filtering (CF)..

1.4.1 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.

In general, methods like this are called k-NN methods, for "k-nearest neighbor." (We'll shorten it to kNN.) Actually, kNN was one of the earliest methods in machine learning in general.

1.4.2 Latent Factor Approach: Matrix Factorization

This one is less intuitive, but is probably the most popular CF method.

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; following R convention, we'll refer to them as NA, the R-language notation for missing values. So for our movie example above, `bf{A[22,377] = 1` and `A[22,88] = NA`.

[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."

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 our data above, for example, $r = 943$ and $s = 1682$. The idea is to find a *low-rank approximation* to A: Using our 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.

For our example above, our predicted value for user 22's rating of movie 88 would be

$$(WH)_{22,88} \tag{1.2}$$

1.4.3 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

$$Y_{ij} = \mu + \alpha_i + \beta_j + \epsilon_{ij} \tag{1.3}$$

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").

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 12.

It is customary to use the "hat" notation$\widehat{\ }$to mean "estimate of." After finding estimates of the above model quantities from our data, our predicted value for user 22 and movie 88 would then be

$$\widehat{A}[22,88] = \widehat{\mu} + \widehat{\alpha}_{22} + \widehat{\beta}_{88} \tag{1.4}$$

1.4.4 Variety Is Good

Why so many methods? There is no perfect solution, and each has advantagages and disadvantages. Some methods may do better than others on specific datasets. Some methods take more computation time than others. Some methods are hard to explain to nontechnical people.

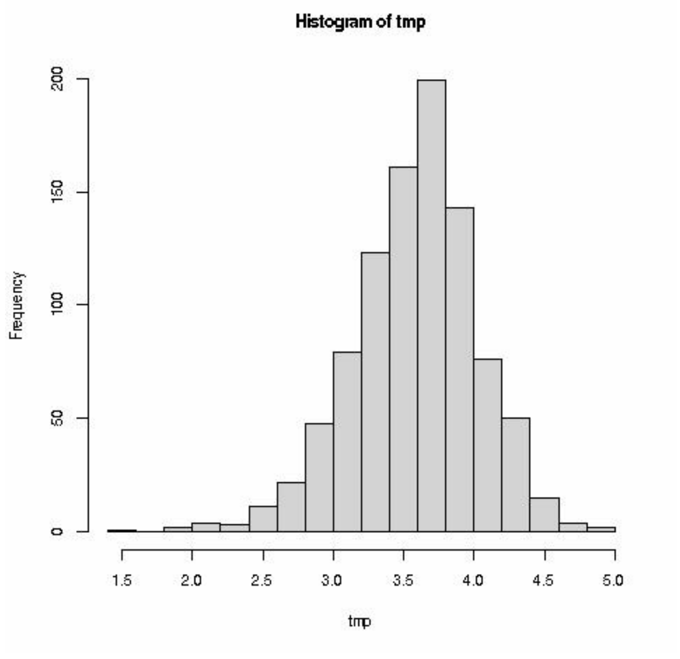
It's good, therefore, to have a variety of methods in our toolbox.

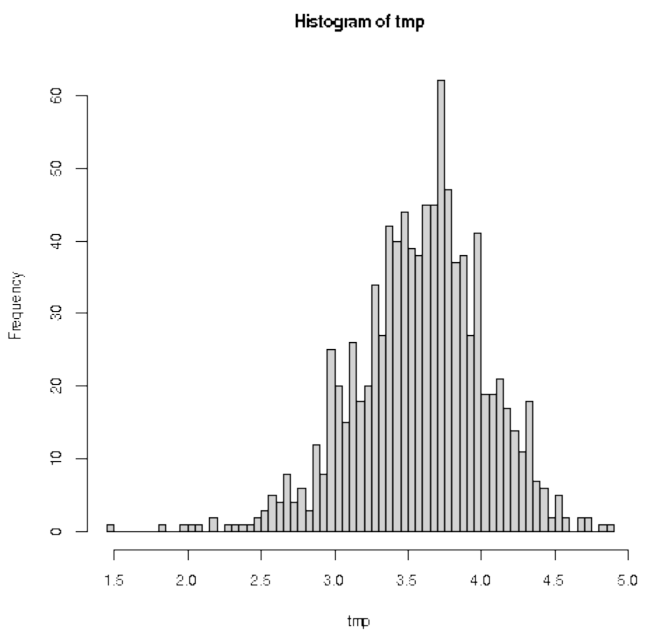
1.5 Tuning Parameters

The reader is likely familiar with histograms. Recall that the analyst must choose a bin width, or similarly, the number of bins. Here there is a tradeoff:

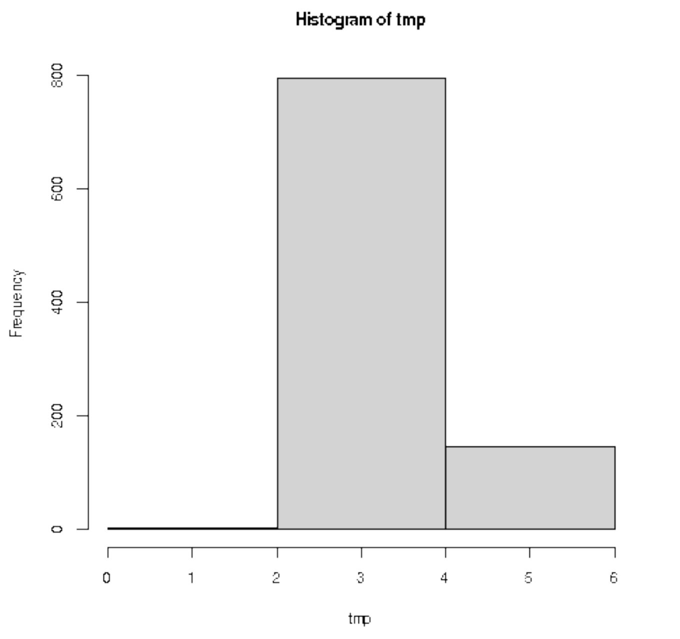
- If the width is too small, some bins will have no data points, or very few. We intuitively feel that small samples are not likely to be accurate, and we will get a histograms that has a very choppy appearance. For instance,

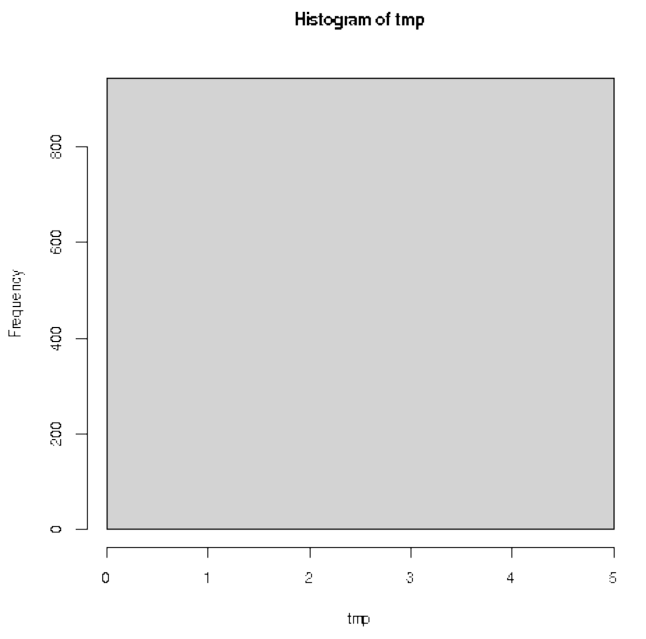
  ```
  > tmp <- tapply(ml100$V3,ml100$V1,mean)
  > hist(tmp,breaks=15)
  > hist(tmp,breaks=50)
  ```





- But it's also bad to have too large a width. In the extreme, we have bins so wide that we have just one or two of them; a 2-bin plot would be of very limited usefulness, and a 1-bin plot would be totally uninformative:





The bin width is called a *tuning parameter* or a *hyperparameter*. In kNN, the number of neighbors k is the tuning parameter, and in MF applications, it's the matrix rank m. Most machine learning algorithms, including most in recommender systems, have tuning parameters, even 10 or more. Choosing the values of those parameters is not easy, but there are methods for it, as we will see.

Important note: Think of plotting a histogram for datasets A and B, similar but A having only 25 data points and B having 500. With dataset B, can afford to make the bin width smaller than with dataset A, as the problem of having empty or nearly-empty bins is much less of an issue. Of course, at a certain point, the number bins will be too large even with B, but the point is that optimal values of tuning parameters depend on the size of our dataset (as well as various other factors).

By the way, the latent-variable statistical model we introduced earlier has no tuning parameters. That may seem tempting, and the model certainly has various advantages. But a tuning parameter gives an opportunity to *tune*, to seek the most accurate model possible.

1.6 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 yet.

However, making good use of side information may not be so easy, for a couple of reasons:

- The exact nature of a relationship may not be very clear. In the movie example, for instance, we might think that age is an important factor. Let's take a quick look, using an extended version of `ml100` obtained from others in the downloaded data (details not show for now):

  ```
  > w <- tapply(ml100kpluscovs$rating,ml100kpluscovs$age,mean)
  > plot(w)
  ```

  Yes, there does seem to be an upward age trend. But is it turning downward at the older ages?

- And that relationship may not even be useful. After all, if some user has rated a large number of films, this data already tells us a lot about this user's taste in movies. For that reason, information about this user's age may be redundant.

No easy answers in predictive data modeling!

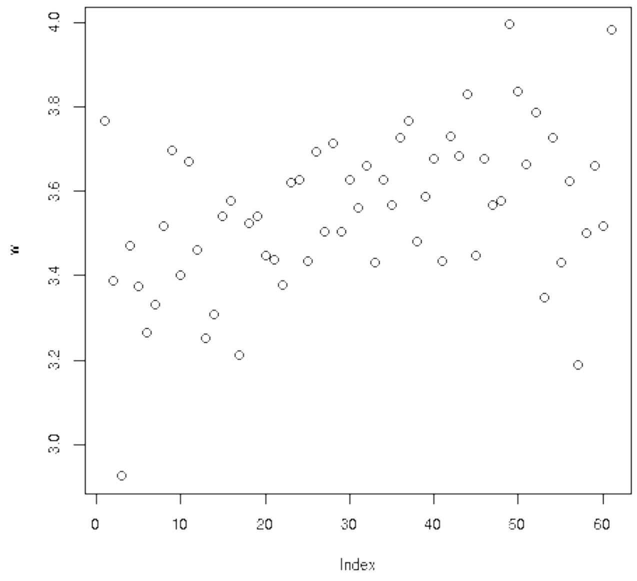


Figure 1.1: Rating vs. Age

1.7 Overfitting, Holdout Sets and Cross-Validation

A major issue—some would say THE major issue—in predictive data modeling is *overfitting.* Professor Yaser Abu-Mostafa of Caltech, a prominent ML figure, once summed it up: "The ability to avoid overfitting is what separates professionals from amateurs in ML."[2] And my Google query on "overfitting" yielded 6,560,000 results!

The concept refers to fitting an overly-complex model to our data. Consider Figure 1.1, for example. We might fit a straight line, and do fairly well. But noting a possible dip in trend near the right side, we could fit a parabola, i.e. a quadratic model, And why stop there? We could try polynomials of degree 3, 4 and so on.

In fact, a polynomial of degree 60 would give an exact fit, with all 61 points lying on the curve.[3] But clearly, it would be absurd to do this, as bad as our example above of a histogram with just one or two bins.

And indeed, the analogy to the histogram case is strong. Our polynomial degree d here is a tuning

[2]https://www.youtube.com/watch?v=EQWr3GGCdzw

[3]A degree-60 model has 62 parameters, including the constant term.

parameter. Too large a value, or one that is too small, won't work well.

In prediction applications, the definition of "won't work well" is that our predictions of new cases in the future won't be very accurate. Referring to our dataset as the *training data* ("train" as in "learn" in "machine learning"), we say that we overfit our model to the training data.

So, we need some new data. If we had some, we would take each of our candidate models—one model for each degree we try—and see which one predicts best on the new data.

Well, we often don't have new data yet, so we create some. Before fitting our models, we set aside some of our data. We call this the *holdout set*, and take the rest of our data as the training set. We fit our various candidate models on this training set, then use each of the fitted models to predict the holdout set. Whichever candidate model does best will then be our choice for prediction on further cases in the future.

But...by coincidence, the particular holdout set that we form could be biaed in favor of one model or another. So, we try many holdout sets, randomly chosen. This of course also means many training sets. For each training/holdout set pair, we fit all of our candidate models. In the end, we take as our final choice whichever model does best over all the holdout sets.

1.8 "P-Hacking," Roundoff Error Etc.

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. Say we have data on 100 people, with some kind of measurement on "humorosity" of each one, plus genetic data on each person.

There are tons of genes to check. For each gene, we could compare the humorosity of people with the gene and people not having the gene (say, having a particular mutation). Since we have a random sample of people, our data is random. The key point is that *just be accident, by coincidence, there may be one gene that seems to separate the humorous people from the dull ones*, when in fact there is no Humor gene. Conducting a statistical study involving very large numbers of variables without concern regarding the above possible scenario is called *p-hacking.*[4]

In the ML context, we (should) worry about p-hacking if we have a large number of potential predictor variables (textitfeatures, in ML parlance). In order to avoid overfitting, we may wish to pare down our feature set, e.g. retain age as a predictor but discard gender. Denote the number of predictors by p. (Standard notation, not related to the first letter in "p-values.") We have 2^p possible subsets to retain. Even with modest value of p, say 10, that's a lot of subsets! By accident,

[4]The etymology need not concern us here, but for the curious: Statistical hypothesis testing—itself methdology that is the subject of much criticism, rightly so—leads to something called a p-value. In the genetics example, the analyst would look for small p-values, and declare an important discovery if they find one for some gene.

one might seem to predict better on the holdout sets.

There is no magic solution to this problem (or lots of other problems in predictive modeling). Granted, 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. But these are difficult to apply, and generally have retrictive assumptions.

Similarly, many ML methods involve computations involving huge matrices, with, say, millions of rows and thousands of columns. Say we wish to find the inverse (or similar entity) of a huge matrix. Not only will the computation take a long time, but also the cumulative roundoff error could be quite substantial. Can we trust the result?

Again, there are no magic solutions. However, with experience one will develop ways to assess possible problems like these, and deal with them. Note that "dealing with them" may simply mean treating our results as useful but not infallible—a good viewpoint in any case.

Chapter 2

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.

2.1 kNN

Let's see how kNN works.

2.1.1 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,[1] 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.

2.1.2 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.

2.1.3 (One) Implementation

Below is code from **rectools** (somewhat simplified).[2] The arguments are:

- **origData:** The original dataset, after having been run through **formUserData()**.
- **newData:** The element of **origData** for NC.[3]
- **newItem:** ID number of the item to be predicted for NC.
- k: The number(s) of nearest neighbors. Can be a vector.

Here is an example of using **formUserData()** on the MovieLens data:[4]

[1]https://github.com/matloff/rectools
[2]This function was written largely by Vishal Chakraborti.
[3]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.
[4]The data have been read from disk without converting to R factors.

```
> 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,...

```
predict.usrData <- function(origData,newData,newItem,k)
{
# we first need to narrow origData down to the users who
# have rated newItem

# here oneUsr is one user record in origData; the function will look for a
# j such that element j in the items list for this user matches the item
# of interest, newItem; (j,rating) will be returned

checkNewItem <- function(oneUsr) {
   whichOne <- which(oneUsr$itms == newItem)
   if (length(whichOne) == 0) {
      return(c(NA,NA))
   } else return(c(whichOne,oneUsr$ratings[whichOne]))
}

found <- as.matrix(sapply(origData,checkNewItem))
# description of 'found':
# found is of dimensions 2 by number of users in training set
# found[1,i] = j means origData[[i]]$itms[j] = newItem;
# found[1,i] = NA means newItem wasn't rated by user i
# found[2,i] = rating in the non-NA case

# we need to get rid of the users who didn't rate newItem
whoHasIt <- which(!is.na(found[1,]))
origDataRatedNI <- origData[whoHasIt]
# now origDataRatedNI only has the relevant users, the ones who
# have rated newItem, so select only those columns of the found matrix
found <- found[,whoHasIt,drop=FALSE]

# find the distance from newData to one user y of origData; defined for
# use in sapply() below
onecos <- function(y) cosDist(newData,y,wtcovs,wtcats)
cosines <- sapply(origDataRatedNI,onecos)
# the vector cosines contains the distances from newData to all the
# original data points who rated newItem

# action of findKnghbourRtng(): find the mean rating of newItem in
# origDataRatedNI, for ki (= k[i]) neighbors
#
# if ki > neighbours present in the dataset, then the
# number of neighbours is used
findKnghbourRtng <- function(ki){
  ki <- min(ki, length(cosines))
  # nearby is a vector containing the indices of the ki closest neighbours
  nearby <- order(cosines,decreasing=FALSE)[1:ki]
  mean(as.numeric(found[2, nearby]))
}
sapply(k, findKnghbourRtng)
}
```

2.1.4 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

$$\frac{u'v}{||u||_2 \; ||v||_2} \tag{2.1}$$

This not really a distance,[5] 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.

2.1.5 Regression Analog

Recall the method of k-nearest neighbor (kNN) regression estimation from Chapter **??**, 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

2.1.6 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.

2.1.7 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.

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

2.1.8 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.

Chapter 3

Some Infrastructure: Linear Algebra

RS methods, as with other machine learning (ML) techniques, often make use of linear algebra, well beyond mere matrix multiplication. Here we will review/extend some issues of particular importance.

3.1 Matrix Rank and Vector Linear Independence

Consider the matrix

$$M = \begin{pmatrix} 1 & 5 & 1 & -2 \\ 8 & 3 & 2 & 8 \\ 9 & 8 & 3 & 6 \end{pmatrix} \tag{3.1}$$

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* $a_1r_1 + a_2r_2 + a_3r_3$ 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: For an $r \times s$ matrix W we have

$$rk(W) \leq \min(r, s) \tag{3.2}$$

where $rk()$ means "rank of." In the case of equality, we say the matrix has *full rank*. A ratings matrix, such as A in Section 1.4.2, should be of full rank, since there presumably are no exact dependencies among users or items.

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 $\mathcal{V}$. And the *dimension* of $\mathcal{V}$ is the number of vectors in a basis (which can be shown to be the same for all bases).

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.

3.2 Partitioned Matrices

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

3.2.1 How It Works

Consider matrices A, B and C,

$$A = \begin{pmatrix} 1 & 5 & 12 \\ 0 & 3 & 6 \\ 4 & 8 & 2 \end{pmatrix} \tag{3.3}$$

and

$$B = \begin{pmatrix} 0 & 2 & 5 \\ 0 & 9 & 10 \\ 1 & 1 & 2 \end{pmatrix}, \tag{3.4}$$

so that

$$C = AB = \begin{pmatrix} 12 & 59 & 79 \\ 6 & 33 & 42 \\ 2 & 82 & 104 \end{pmatrix}. \tag{3.5}$$

We could partition A as, say,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{3.6}$$

where

$$A_{11} = \begin{pmatrix} 1 & 5 \\ 0 & 3 \end{pmatrix}, \tag{3.7}$$

$$A_{12} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}, \tag{3.8}$$

$$A_{21} = \begin{pmatrix} 4 & 8 \end{pmatrix} \tag{3.9}$$

and

$$A_{22} = \begin{pmatrix} 2 \end{pmatrix}. \tag{3.10}$$

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

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{3.11}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{3.12}$$

so that for example

$$B_{21} = \begin{pmatrix} 1 & 1 \end{pmatrix}. \tag{3.13}$$

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

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \tag{3.14}$$

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}.

3.2.2 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

$$A = (A_1, A_2, A_3) \tag{3.15}$$

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

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{3.16}$$

The point is that (3.15) 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

$$Ax = x_1 A_1 + x_2 A_2 + x_3 A_3 \tag{3.17}$$

As noted, we then "unpretend," and find that (3.17) says that

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

Similarly:

> Let y be a row vector of length equal to the number of rows of A. Then yA is a linear combination of the rows of A, with the coefficients being the elements of y.

Taking this a step further:

> Each row of AB is a linear combination of the rows of B, with the coefficients for a given B row being the elements of the corresponding row of A.
>
> Similarly, each column of AB is a linear combination of the columns of A, with the coefficients being the elements of the corresponding column of B.

3.2.2.1 Application of Partitioning: Rank of a Matrix Product

Say we have conformable matrices A and B, and set $C = AB$. We have:

- The quantity $rk(C)$ is the maximal number of linearly independent rows of C, among *all* possible coefficients.
- The quantity $rk(B)$ is the maximal number of linearly independent rows of B, among *all* possible coefficients.
- Each row of $C = AB$ is a linear combination of rows of B, i.e. each row of C is *some* linear combination of the rows of B.
- Thus we see that

  $$rk(B) \geq rk(C) \tag{3.18}$$

  Similarly,

  $$rk(A) \geq rk(C) \tag{3.19}$$

- In other words,

  $$rk(C) \leq min(rk(A), rk(B)) \tag{3.20}$$

3.2.3 Application of Partitioning: Approximate Matrix Factorization

Recall the relation

$$A \approx WH \tag{3.21}$$

in Section 1.4.2, where A is $r \times s$, W is $r \times m$ and H is $m \times s$.

The material in the last section then says:

- Row i of A is *approximately* equal to a linear combination of the rows of H.
- Column j of A is *approximately* equal to a linear combination of the rows of W.

We'll see in Chapter 10 that this has big implications for the matrix factorization method of RS. We'll see that in a sense, W will contain information about "typical" users, and H will contain information about "typical" items.

3.3 Vector Norms

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

$$||x||_p = \left(\sum_{i=1}^{n} [x_i|^p \right)^{1/p} \tag{3.22}$$

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 a length-rs vector.

3.4 Handy Facts

- For conformable matrices A and B,

$$(AB)' = B'A' \tag{3.23}$$

  where $'$ denotes matrix transpose.

- For invertible and conformable matrices A and B,

$$(AB)^{-1} = B^{-1}A^{-1} \tag{3.24}$$

- 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 4

Probability

It is assumed the reader has background in calculus-based probability constructs, e.g. density functions and distributions involving infinite series. Here we treat some advanced topics used in the sequel.

4.1 Multivariate Distributions

4.1.1 Multivariate Probability Mass Functions

Recall that for a single discrete random variable X, the distribution of X is defined to be a list of all the values of X, together with the probabilities of those values. We encapsulate those in the *probability mass function*:

$$p_X(i) = P(X = i) \tag{4.1}$$

Here the argument is i.

So, if X is the number of heads in two flips of a coin, $p_X(0) = 1/4$, $p_X(1) = 1/2$ and $p_X(2) = 1/4$.

This is extended to a pair of discrete random variables U and V as

$$p_{U,V}(i,j) = P(U = i, V = j) \tag{4.2}$$

with arguments i and j.

For example, suppose we have a bag containing two yellow marbles, three blue ones and four green ones. We choose four marbles from the bag at random, without replacement. Let Y and B denote the number of yellow and blue marbles that we get. Then

$$p_{Y,B}(i,j) = P(Y = i \text{ and } B = j) = \frac{\binom{2}{i}\binom{3}{j}\binom{4}{4-i-j}}{\binom{9}{4}} \tag{4.3}$$

Here is a table displaying all the values of $p_{Y,B}(i,j)$

i ↓, j →	0	1	2	3
0	0.0079	0.0952	0.1429	0.0317
1	0.0635	0.2857	0.1905	0.1587
2	0.0476	0.0952	0.0238	0.000

So in our marble example above, $p_{Y,B}(1,2) = 0.048$, $p_{Y,B}(2,0) = 0.012$ and so on.

4.1.2 Multivariate Density Functions

Recall that for a continuous random variable X, we can find probabilities involving X by integrating its density:

$$P(X \text{ in } A) = \int_A f_X(t)\ dt \tag{4.4}$$

for regions A in the real line.

This extends to multiple dimensions. For sets A in the plane,

$$P[(X,Y) \text{ in } A] = \int_A \int f_{X,Y}(s,t)\ ds\ dt \tag{4.5}$$

The density for k random variables $X_1, ..., X_k$ is likewise a function whose k-fold integrals are probabilities.

We almost never compute such integrals. Instead, we rely on approximations, including simulation. The R package **mvtnorm** can be used for these purposes.

4.2 Relations Between Variables

4.2.1 Covariance

You have seen the *variance* of a single random variable before,

$$Var(X) = E[(X - EX)^2] \tag{4.6}$$

It is a measure of *dispersion*, i.e. how much X varies in repeated realizations.[1]

In the example above in which X is the number of heads obtained in two flips of a coin, one can show that $EX = 1$ and $Var(X) = 2(0.5)(1 - 0.5) = 0.5$. So, if we look at many, many realizations of X, the long-run average value of X will be 1, and the long-run average of $(X - 1)^2$ will be 1/2.

The *standard deviation* of X, denoted here by $\sigma(X)$, is defned at $\sqrt{Var(X)}$. Recall that for a constant, i.e. a nonrandom quantity, c, we have

$$\sigma(cX) = c\sigma(X) \tag{4.7}$$

and

$$\sigma(X + c) = \sigma(X) \tag{4.8}$$

The reader should make sure both of these make intuitive sense. For instance, in the second case, the reasoning is: Set $Y = X + c$. In repeated realizations, Y will vary from its mean exactly as X varies from its mean.

How does this extend? The answer is the *covariance*:

$$Cov(X, Y) = E[(X - EX)(Y - EY)] \tag{4.9}$$

Just as $Var(X)$ is a measure of how much X varies, the covariance is a measure of how X and Y vary *together*. Note that it is a signed quantity.

Think of height and weight in a human population, X and Y. People who are taller than average tend to be heavier than average, making

$$(X - EX)(Y - EY) > 0 \tag{4.10}$$

[1] If we look at many instances of a random variable X, they are called *realizations*. By the way, EX means $E(X($; it is customary to drop the parentheses if there is no danger of ambiguiity.

Shorter people tend to be lighter than average, but again

$$(X - EX)(Y - EY) > 0 \tag{4.11}$$

Now,

$$E[(X - EX)(Y - EY)] \tag{4.12}$$

is the average value of that product, as we range through various people in the population, so the covariance will be positive. $(X - EX)(Y - EY)$ won't be positive for all people, but if the relation is typical enough, the average will be positive.

Note that $Cov(X, X) = Var(X)$.

4.2.2 Covariance Matrices

Say we have p random variables X_1,..., X_p. Their *covariance matrix* is $p \times p$, with the (i, j) element being $Cov(X_i, X_j)$. The diagonal elements are the variances of the individual variables.

4.3 Correlation

Roughly speaking, positively-related variable will have positive covariance, with a similar statement for negatively-related variables. In fact, correlation is just scaled-down variance:

$$\rho(X, Y) = \frac{Cov(X, Y)}{\sigma(X)\ \sigma(Y)} \tag{4.13}$$

where ρ and σ are standard symbols for correlation and standard deviation, respectively.

Correlation is unitless. E.g. if X and Y are measured in meters, the meter units cancel. One can show that

$$-1 \leq \rho \leq 1 \tag{4.14}$$

4.3.1 Proof of (4.14) by Vector Spaces

Let $\mathcal{W}$ be the set of all random variables with mean 0 and finite variance (in some probability space). We lose no generality in assume mean 0, since replacing a random variable U by $U - EU$

makes the mean 0 and does not change the correlation.

$\mathcal{W}$is clearly closed under linear combinations, and we can define an inner product ("dot product") by

$$< U, V >= Cov(U, V) \tag{4.15}$$

The vector norm will be

$$||U|| = (< U, U >)^{0.5} = \sigma(U) \tag{4.16}$$

To qualify as an inner product, it must be symmetric and linear in both arguments, and must be *positive definite*, i.e. $< U, U >\geq 0$, with equality if and only if $U = 0$. Covariance satisfies all these conditions.

On any inner product space, the Cauchy-Schwarz Inequality says that the absolute value of the inner product of two vectors, divided by the product of their norms, is at most 1. That gives us (4.14)!

4.3.2 Is a Correlation Large or Small?

So, is a correlation of, say, 0.4, large or small? It is customary to gauge this by the squared correlation, which arises in the theory of linear predictors (Chapter 5). The quantity ρ^2 can be shown to be the reduction in mean squared prediction error when we predict V using U (or vice versa).

Say we want to predict V knowing U. Intutively, the larger $\rho(U, V)$ is, the more accurately I can predict. This can be quantified.

Let's consider predictors of the form

$$\widehat{V} = c_1 + c_2 U \tag{4.17}$$

$\widehat{V}$ means the predicted value of V

There are optimal choices for the c_i, but let's not get into that for now.

Our *mean squared prediction error* is then

$$\text{MSPE}_1 = E[(V - \widehat{V})^2] \tag{4.18}$$

But what if we don't know U? Then we can't use (4.17). In that case, a reasonable guess for V would be EV. Then our MSPE would be

$$\text{MSPE}_2 = E[(V - EV)^2] \tag{4.19}$$

You'll recognize the latter as $Var(V)$, but the main point here is that it turns out that

$$\rho^2(U,V) = \frac{MSPE_2 - MSPE_1}{MSPE_2} \tag{4.20}$$

In other words, the squared correlation can be interpreted as the proportional reduction in MSPE that we attain by using U to predict V instead of using EV to do so. This is often described as "The proportion of variation in V that is explainable by U."

4.3.3 Example: MovieLens

Again, **ml100kpluscovs** is MovieLens plus some side information:

```
 head(ml100kpluscovs)
  user item rating timestamp age gender        occ   zip userMean Nuser G1 G2
1    1    1      5 874965758  24      M technician 85711 3.610294   272  0  0
2  117    1      4 880126083  20      M    student 16125 3.918605    86  0  0
3  429    1      3 882385785  27      M    student 29205 3.393720   414  0  0
4  919    1      4 875289321  25      M      other 14216 3.470046   217  0  0
5  457    1      4 882393244  33      F   salesman 30011 4.025271   277  0  0
6  468    1      5 875280395  28      M   engineer 02341 3.993007   143  0  0
  G3 G4 G5 G6 G7 G8 G9 G10 G11 G12 G13 G14 G15 G16 G17 G18 G19 itemMean Nitem
1  0  1  1  1  0  0  0   0   0   0   0   0   0   0   0   0   0 3.878319   452
2  0  1  1  1  0  0  0   0   0   0   0   0   0   0   0   0   0 3.878319   452
3  0  1  1  1  0  0  0   0   0   0   0   0   0   0   0   0   0 3.878319   452
4  0  1  1  1  0  0  0   0   0   0   0   0   0   0   0   0   0 3.878319   452
5  0  1  1  1  0  0  0   0   0   0   0   0   0   0   0   0   0 3.878319   452
6  0  1  1  1  0  0  0   0   0   0   0   0   0   0   0   0   0 3.878319   452
```

(The G's are genres.)

Let's find some correlations:[2]

[2]These are only *sample* correlations. Our data are regarded as a sample from a *population* of data from all possible users and all possible movies.

```
> cor(ml100kpluscovs[,c('age','userMean','Nuser')])
                 age   userMean       Nuser
age       1.00000000  0.1355457 -0.03908845
userMean  0.13554566  1.0000000 -0.30755257
Nuser    -0.03908845 -0.3075526  1.00000000
```

There seems to be rather little relation betwen age and the mean rating for a user, and between age and the number of ratings for a user. But the latter two variables seem to have a negative relation—still rather weak, with each variable explaining about 9% of the variation of the other; users who give higher ratings rate fewer movies.

We might ask, say, whether any genres are related to each other:

```
# some rows are all NAs
> cc <- complete.cases(ml100kpluscovs[,11:29])
> w <- ml100kpluscovs[cc,]
> cor(w[,11:19])
            G11          G12          G13          G14          G15         G16
G11  1.00000000 -0.031493550 -0.030273614  0.232039913 -0.055267116  0.01614678
G12 -0.03149355  1.000000000 -0.054022202  0.001783286 -0.076165496  0.03418726
G13 -0.03027361 -0.054022202  1.000000000 -0.053634780 -0.009765185 -0.08138982
G14  0.23203991  0.001783286 -0.053634780  1.000000000 -0.059997502 -0.03084474
G15 -0.05526712 -0.076165496 -0.009765185 -0.059997502  1.000000000 -0.06345353
G16  0.01614678  0.034187257 -0.081389822 -0.030844736 -0.063453529  1.00000000
G17  0.11025056  0.069788805 -0.111244337  0.230288448 -0.106256128  0.04679552
G18 -0.04280364 -0.076381585 -0.055381372 -0.075833813  0.126545414  0.16727458
G19 -0.01826573 -0.032594547 -0.031331963 -0.032360794 -0.052448355 -0.05253437
            G17         G18         G19
G11  0.11025056 -0.04280364 -0.01826573
G12  0.06978881 -0.07638159 -0.03259455
G13 -0.11124434 -0.05538137 -0.03133196
G14  0.23028845 -0.07583381 -0.03236079
G15 -0.10625613  0.12654541 -0.05244835
G16  0.04679552  0.16727458 -0.05253437
G17  1.00000000 -0.10041124 -0.07278201
G18 -0.10041124  1.00000000 -0.02372091
G19 -0.07278201 -0.02372091  1.00000000
```

There may be some slight relations.

4.3.4 An Important Matrix Property

Say we have a random *vector* X, length p, and a $k \times p$ nonrandom matrix A. Then AX is a new random vector, of length k. One can show that

$$Cov(AX) = A \; Cov(X) \; A' \tag{4.21}$$

where here $Cov()$ with a single argument means covariance matrix, and $'$ denotes transpose.

An important special case is that in which A is a row vector. Then AX is a number, and $Cov(AX)$ reduces to $Var(AX)$.

4.4 The Normal (Gaussian) Family of Distributions

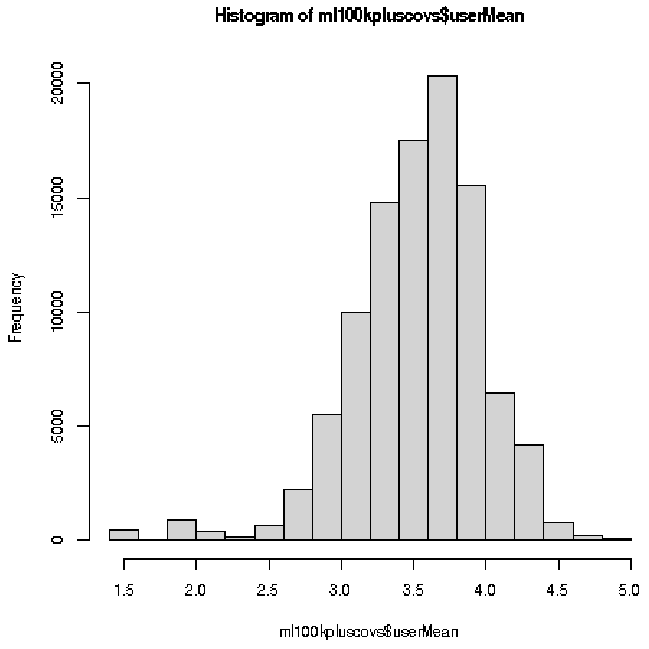
4.4.1 Univariate

The density function here is the famous "bell-shaped curve,"

$$f_X(t) = \frac{1}{\sqrt{2\pi}c} \exp\left(-0.5 \left[(t-b)/c\right]^2\right) \tag{4.22}$$

This is a parametric family of curves. Here b and c are parameters. They also turn out to be the mean and standard deviation of the distribution. Notation for this distribution is $\mathrm{N}(a, b^2)$.

This is a very common model, as a histogram of one's data often looks rather bell-shaped. Here is one from MovieLens:



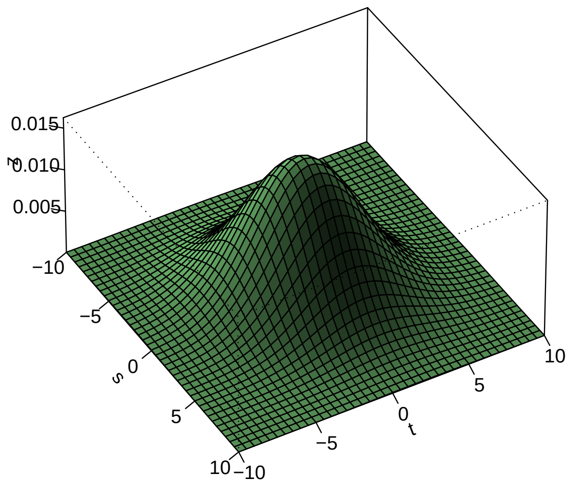
There are lots of issues here, e.g. "How close must the histogram be to bell-shaped in order for us to use it as a model?" More on this in Chapter 6.

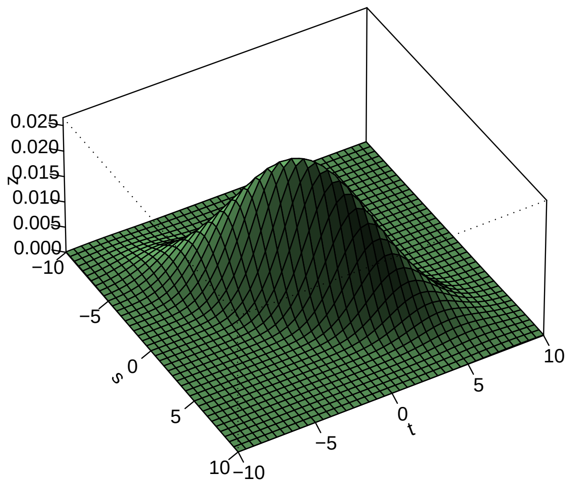
One reason that random variables in practice tend to be approximately normal is the *Central Limit Theorem*, which says that if a random variable X is a sum of many random variables, then its distribution is approximately normal even if the individual terms are not normal. An example is human height. Think of the body as made up of a number of chunks. The person's height is the sum of the heights of the chunks. This is a rough analysis, actually based on advanced versions of the theorem, but in fact human height *is* approximately normally distributed. (Try running a histogram on the height column in the `mlb` dataset that comes with `regtools`.)

4.4.2 Multivariate

If we have two random variables X and Y, they have a *bivariate normal distribution* if their density looks like a "three-dimensional bell." We will skip the exact density (for p variables).

Below are graphs of example bivariate normal densities, one with $\rho = 0.2$ and the other with $\rho = 0.8$.





You can see the effect of a higher correlation in the second graph; the bell is hugging a line at the base, so the two variables tend to be large together or small together.

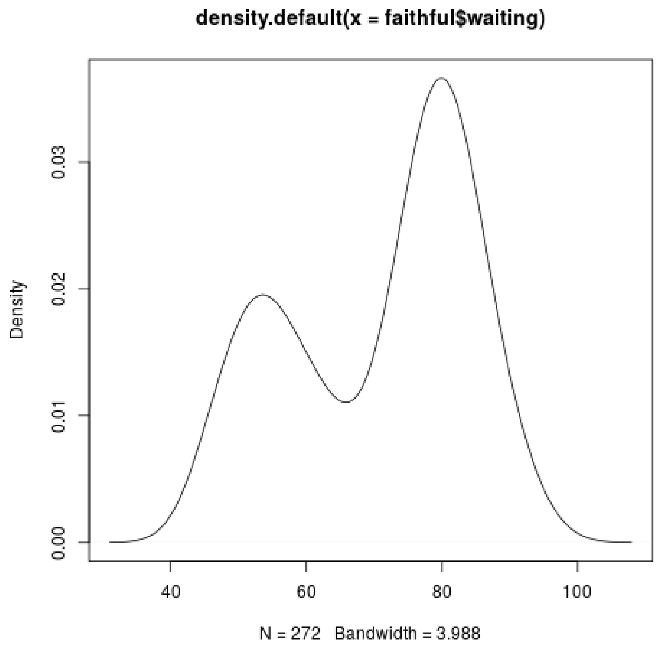
4.5 Mixture Distributions

4.5.1 Motivating Example

The `faithful` dataset included in R consists of data from the Old Faithful geyser in Yellowstone National Park, USA. Let's take a look at times between eruptions:

```
> plot(density(faithful$waiting))
```

Here we are using a more sophisticated density estimator than `hist()`, `density()`. It gives smooth curves.



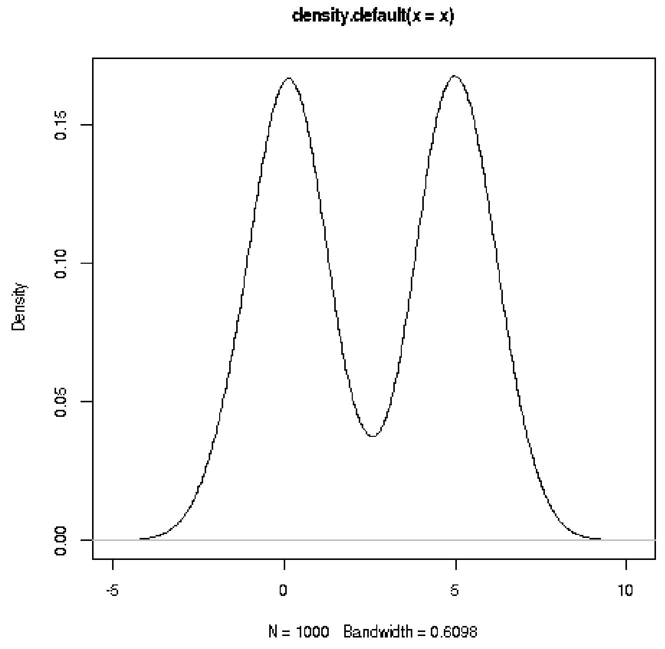
We might say it's a "double bell." What's at work here?

It looks like a *mixture* of two normals, meaning that possibly two underground mechanisms are causing eruptions, sometimes mechanism A and sometimes mechanism B. If the wait times between events is normal for each mechanism, we would get the "double bell" appearance.

To understand this better, consider this simulation experiment:

```
> x1 <- rnorm(2500)
> x2 <- rnorm(2500)+5
> i <- sample(c(TRUE,FALSE),1000,replace=TRUE)
> x <- ifelse(i,x1,x2)
> plot(density(x))
```

Here we generated 2500 random values each from N(0, 1) and N(5, 1), but formed `x` by randomly choosing from those two sets of 2500. To put more weight on one than the other, use the **prob** argument in **sample()**.



Sure enough, we got a double bell.

One can use a technique known as the *EM algorithm* to estimate the parameters of the two normals, plus the proportion of each. The R package `mixtools` does the computation.

4.5.2 Clustering Algorithms and Use in RS

Clustering methods are exploratory tools to find mixtures. In the Old Faithful example, it was rather clear that there was a mixture of two normals, but in general it's much, much harder, for a couple of reasons:

- We are usually in a multivariate setting, rather than the univariate one in the Old Faithful example.
- The number of mixing components, 2 in the above example, is not clear at all.

Various methods have been devised, and have been found useful in RS, such as in *market segmentation.* We may find that moviegoers, for instance, tend to fall into, say, 5 main groups. We will return to this later.

Chapter 5

Linear Models

In Section 4.3.2 we made reference to *linear models*. Directly or indirectly, they form the basis for much of ML.

5.1 Minimizing MSPE

Suppose we are predicting a random variable Y, based on a random vector X, say predicting weight from (height,age). Our guess will be some function of X, $g(X)$. What is the best g, in the sense of minimizing MSPE? In other words, how should we choose g to minimize

$$E\left([Y - g(X)]^2\right) \tag{5.1}$$

One can show that:

> We get minimum MSPE by taking $g(X)$ to be the conditional mean, $E(Y|X)$.

This should make good intuitive sense: To predict the weight of someone 70 inches tall, we take our guess to be the mean weight of all 70-inch-tall people.

We will then define the *regression function*

$$\mu(t) = E(Y|X = t) \tag{5.2}$$

Note that the argument $t = (t_1, ..., t_p)'$ is a vector. (Note: All vectors will be column vectors if not otherwise stated.) Also, prepending a 1, we will use the notation $\widetilde{t} = (1, t')'$.

This definition is *general*; it does not assume a linear model. Let's now see where the classical linear model comes from.

5.2 Motivation for the Classical Linear Model

Say we are predicting a variable Y from p features, $X_1, ..., X_p$. Let X denote the (column) vector of those features. Sometimes we will also add a constant 1 at the beginning, setting

$$\widetilde{X} = (1, X_1, ..., X_p)' \tag{5.3}$$

Here is the main point:

> Suppose (X, Y) has a $p + 1$ dimensional normal distribution. Then the conditional distribution of Y given X has the following properties:
>
> - **Linearity:**
>
> $$\mu(t) = \beta_0 + \beta_1 t_1 + ... + \beta_p t_p = \beta' \widetilde{t} \tag{5.4}$$
>
>   for a certain vector β.[1]
> - **Conditional normality:** For any t, the conditional distribution of Y given $X = t$ is normal.
> - **Homoskedasticity:** The conditional variance of Y given $X = t$ is the same for all t.

5.3 Coming Back Down to Earth

For those readers who have some background in linear regression modeling, the above three bullet items will sound familiar. They are the motivation for the classic assumptions of linear modeling: linearity, normality and homoskedasticity. *However, note the following:*

- The normality and homoskedasticity assumptions are used only for *statistical inference*, i.e. confidence intervals and tests. We will not be performing statistical inference in this book; ML is mainly about *prediction.*[2]

[1]One can show that $\beta = E(Y\widetilde{X})[E(\widetilde{X}\widetilde{X}')]^{-1}$.

[2]Sadly, in ML circles, the situation is confused by using the term *inference* for prediction.

Even for inference, those assumptions aren't really necessary. We get normality for our estimated β from the Central Limit Theorem, and we can deal with the lack of homoskedasticity by using the *sandwich estimator*, e.g. in the R **sandwich** package.

- Many regresssion functions in practice are approximately linear. And, as will be seen shortly, polynomial models, which may provide a good fit in nonlinear situations, are actually linear!

In other words:

- The classic linear model is motivated by settings in (X, Y) has a multivariate normal distribution.
- This assumption is highly restrictive, but *is not necessary* for our purposes.

For the remainder of this chapter, we assume

$$g(t) = \beta' \widetilde{t} \tag{5.5}$$

for some vector β to be estimated from our data.

5.4 Estimating β

Here we lay groundwork leading up to the famous *least-squares estimate.*

5.5 Sample vs. Population

Most readers have probably noticed that when the results of a survey are released, say during elections, a *margin of error* (MOE) is stated. For instance, "55% of those surveyed say they plan to vote for Candidate Jones, with a margin of error of 3.2%." The MOE is recognition of the fact that only a sample of voters were surveyed, not the entire population of voters.

Let p denote the population proportion, i.e. the proportion of voters across the population who favor Jones. The value of p is unknown, but our estimate is $\widehat{p} = 0.55$. The "hat" notation$\widehat{\ }$means "estimate of." (The MOE is the radius of a 95% confidence interval for p, though as noted, we do not make much use of statistical inference in this book.)

Every ML method is an estimator in some form or other. However, the terms *sample* and *population* are not used in the ML community. Instead, they speak a *probabilistic generative process* to mean the same thing as sampling data from a population.

So, β is a population value. Its estimator from the data is denoted $\widehat{\beta}$.

5.5.1 The Least Squares Estimator

Denote our data by $(X_{ij}, Y_j),\ j = 1, ..., n$. In other words, we have n data points, and in the j^{th} of them, X_{ij} and Y_j are the valus of X_i and Y, respectively. Also, define

$$X^{(j)} = (1, X_{1j}, ..., X_{pj})' \tag{5.6}$$

To keep a concrete example in mind, again suppose we are predicting human weight Y from height X_1 and age X_2. $X^{(3)}$, for instance, is then the vector (height,age) for the third person in our data (with a 1 prepended).

Putting together the fact that g minimizes (5.1) and the assumption (5.5), we have that β is the vector b that minimizes

$$E\left[(Y - b'\widetilde{X})^2\right] \tag{5.7}$$

The sample analog of this quantity is

$$\frac{1}{n}\sum_{j=1}^{n}\left[(Y_j - b'X^{(j)})^2\right] \tag{5.8}$$

Since $b = \beta$ minimizes (5.7), it is natural by analogy to take $\widehat{\beta}$ to be the value of b that minimizes (5.8).

Just a little bit more notation:

$$D = (Y_1, ..., Y_n)' \tag{5.9}$$

$$A = \begin{pmatrix} X^{(1)'} \\ ... \\ X^{(n)'} \end{pmatrix} \tag{5.10}$$

Then (5.8) (without the $1/n$ factor) is

$$(D - Ab)'(D - Ab) \tag{5.11}$$

To minimize this, we must set the derivative of (5.11) to 0. It can easily be verified that for a vector u,

$$\frac{d}{du} u'u = 2u \tag{5.12}$$

Applying this and the Chain Rule to (5.11), i.e.

$$\frac{d}{db} = \frac{d}{du} \frac{du}{db} \tag{5.13}$$

we have

$$0 = A(D - Ab) \tag{5.14}$$

Solve for b:

$$\widehat{\beta} = (A'A)^{-1} AD \tag{5.15}$$

This is the *least squares estimator* of β.

5.6 Computation

We certainly do not want to do this compution by hand. What are our choices?

5.6.1 lm()

R's **lm()** ("linear model") function does the computation for us. Here is an example using **mlb**, a dataset in the **regtools** package, involving Major League Baseball players:[3]

```
> head(mlb)  # take a look around
        Position Height Weight   Age
1        Catcher     74    180 22.99
2        Catcher     74    215 34.69
3        Catcher     72    210 30.78
4   First_Baseman     72    210 35.43
5   First_Baseman     73    188 35.71
```

[3]We trimmed down the number of columns here.

```
6 Second_Baseman      69     176  29.39
> lmout <- lm(Weight ~ Height + Age,mlb)
> coef(lmout)
 (Intercept)       Height         Age
-187.6381754    4.9235994    0.9115326
> predict(lmout,data.frame(Height=73,Age=25))
       1
194.
```

The call says, "Fit a linear model for predicting weight from height and age, using the dataset **mlb**." The **coef()** function extracts $\widehat{\beta}$ from the output object. We see that $\widehat{\beta} = (-187.64, 4.92, 0.91)'$.

We then predict a new case. If the player whose weight is to be predicted is of height 73 and age 25, we would predict wieght 194 (an integer here just by coincidence).

By the way, **coef()** and **predict()** are *generic* functions in R, meaning that their actions depend on the class of object they are called on. In this case, **lmout** is of class **'lm'**, so **coef()** is *dispatched* to a function **coef.lm()** tailored to such objects; **predict()** is dispatched to **predict.lm()**. There are many generic functions in R, notably **print()** and **plot()**.

5.6.2 qeLin()

In this book, we will use the **qeML** package. The name stands for "quick and easy machine learning," alluding to the goal of making things as quick and convennient as possible:

- The functions have a simple, *uniform* user interface.
- Cross-validation is automatically taken care of .

Most of the **qeML** functions are wrappers to functions in other R packages; **qeML** adds a simple, uniform interface to them.

Let's apply it to the above example:

```
> qeout <- qeLin(mlb[,-1],'Weight')
holdout set has  101 rows
```

The **qeML** functions all have the user specify the "Y" variable in the second argument, and "X" in the first argument. It is assumed that all of the "X" columns will be used to predict "Y," so to be consistent with the earlier example, we needed to exclude column 1.

We can predict with any of the **qeML** functions.

```
> predict(qeout,data.frame(Height=73,Age=25))
      11
194.2245
```

The functions do automatic cross-validation, with the size of the holdout being 10% of the size of the dataset.[4] The prediction accuracy on the holdout set is in the **testAcc** component of the return value:

```
> qeout$testAcc
[1] 14.13652
```

For continuous "Y," this is the Mean Absolute Prediction Error (MAPE). On average, our predictions are about 14 pounds off. Could this be improved by adding **Position** to our feature set?

```
> qeLin(mlb,'Weight')$testAcc
holdout set has  101 rows
[1] 12.16986
```

Ah, yes. Catchers tend to be stocky, pitchers lanky and so on, so there is valuable extra information there. By the way, **lm()** automatically converts R factors to dummy/one-hot form, as was the case for **Position** here.

As noted, the **qeML** functions offer the benefit of a uniform API. Let's try the kNN ("k-nearest neighbors" method:

```
> qeKNN(mlb,'Weight')$testAcc
holdout set has  101 rows
[1] 15.05624
```

Oh, not as good. If the true regression function is approximately linear, we generally will do better by exploiting that fact.

As wrappers, the class of a **qeML** return value is a subclass of the function being wrapped:

```
> class(qeout)
[1] "qeLin" "lm"
```

They thus inherit methods:

```
> coef(qeout)
 (Intercept)       Height          Age
-192.0942868    4.9726027    0.9327504
```

[4]This is the reason for the slight discrepancy above.

5.7 The Logistic Model

Say we are predicting a binary"Y." In the **mlb** data, say we are predicting whether a player is a catcher. Say we define Y to be 1 for catcher, 0 if not. In many ML circles, the coding is 1 and - 1, as opposed to the "statistical" 1,0. The latter has the advantage that its expectation is the probability of the $Y = 1$ class, which will be seen below is an integral part of the *logistic* model

$$\mu(t) = E(Y|X = t) = P(Y = 1|X = t) \tag{5.16}$$

The model is

$$P(Y = 1|X = t) = \frac{1}{1 + e^{-\beta' t}} \tag{5.17}$$

The reader will notice that in that expression we see the expression $\beta' t$ from the linear model. Accordingly, the logistic (often called "logit") is termed a *generalized linear model.* The R **glm()** function implements models such as this. (Another is *Poisson regression.*) In the logistic case, **qeLogit()** serves as a wrapper.

5.7.1 Again, a Multivariate Normal Motivation

The right-hand side of (5.16) is in (0,1), and thus a reasonable model for a probability. It is an increasing function of $\beta' t$, thus giving it an appealing quasi-linear nature. If say Y codes having diabetes, it is reasonable to postulate that the probability of having the disease is an increasing function of the age of the patient.

As with the linear model, though, there is a multivariate normal motivation:

> Suppose the distribution of X, given $Y = i$, is multivariate normal with mean vector different for each i but with covariance matrix being the same across values of $i, i = 0, 1$. Then (5.16) holds.

Let's see how this works for $p = 1$. Consider a short interval in the real line, $A = (t, t + \epsilon)$. We'll look at $P(Y = 1|X \text{ in } A)$, then let $\epsilon \to 0$ to get $P(Y = 1|X = t)$

Let $q = P(Y = 1)$, the *unconditional* probability of class 1. Let $w_i(t)$ denote the conditional density of X, given $Y = i$ (which by assumption is normal). Then by Bayes Rule,

$$
\begin{aligned}
P(Y=1|X \text{ in } A)[&= \frac{qP(X \text{ in } A \mid Y=1)}{qP(X \text{ in } A \mid Y=1) + (1-q)P(X \text{ in } A \mid Y=0)} && (5.18)\\
&\approx \frac{q\epsilon w_1(t)}{q\epsilon w_1(t) + (1-q)\epsilon w_0(t)} && (5.19)\\
&= \frac{1}{1 + \frac{1-q}{a}\frac{w_0(t)}{w_1(t)}} && (5.20)
\end{aligned}
$$

Now, since the w_i are normal, we have

$$
w_i(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-0.5\left(\frac{t-\mu_i}{\sigma}\right)^2} \tag{5.21}
$$

Substituting this in (5.19) does indeed simplify to (5.16) for the appropriate β.

5.7.2 Example: Predicting "Catcherness"

```
> catch <- mlb$Position == 'Catcher'
> mlb$catch <- as.factor(catch)
> qeo <- qeLogit(mlb[,-1],'catch')
> predict(qeo,data.frame(Height=73,Weight=225,Age=25))
$predClasses
[1] "FALSE"

$probs
         FALSE      TRUE
[1,] 0.8743196 0.1256804
```

The **qeML** functions sense that Y is binary (or more generally, categorical) by testing whether it is an R factor. So we needed to convert the logical vector to a factor. The prediction was FALSE, i.e. this player is guessed to not be a catcher, and in fact has only about a 13% chance of being a catcher.

5.7.3 Example: Vertebral Disease

As noted, we can also predict general categorical Y. Here we consider a vertebral dataset from the UCI Machine Learning Repository.

```
> head(vert)
```

```
     V1    V2    V3    V4     V5    V6 V7
1 63.03 22.55 39.61 40.48  98.67 -0.25 DH
2 39.06 10.06 25.02 29.00 114.41  4.56 DH
3 68.83 22.22 50.09 46.61 105.99 -3.53 DH
4 69.30 24.65 44.31 44.64 101.87 11.21 DH
5 49.71  9.65 28.32 40.06 108.17  7.92 DH
6 40.25 13.92 25.12 26.33 130.33  2.23 DH
```

There are three disease types, NO, DH and SL, where NO means normal. The other columns are various vertebral meansurements.

```
> qel <- qeLogit(vert,'V7')
holdout set has  31 rows
```

As an example of prediction, consider a patient like the one in **vert[1,]**; but with $V7 = 1.88$.

```
> newPatient <- vert[1,-7]
> newPatient$V6 <- 1.88
> predict(qel,newPatient)
$predClasses
[1] "DH"

$probs
            DH        NO         SL
[1,] 0.8594815 0.1190561 0.02146242
```

The prediction would be DH, with a probability of about 86%.

How well does logit predict on this dataset?

```
> qel$testAcc
[1] 0.1612903
> qel$baseAcc
[1] 0.5053763
```

For binary/categorical Y, the **qeML** functions report the overall misclassification rate. Here, we would predict correctly about 84% of the time.

Are the features very helpful in prediction? Consider this:

```
> table(vert$V7) / nrow(vert)

       DH        NO        SL
0.1935484 0.3225806 0.4838710
```

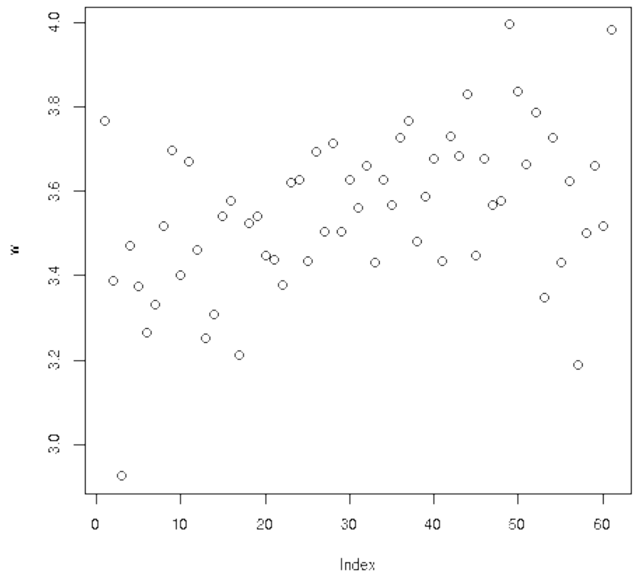
If we did not have the features to use for prediction, we would always guess SL, as it is the most common cateogory. We would be right 48% of the time, thus wrong 52% of the time. But actually we didn't need to make that calculation above; it's returned by **qeLogit()**:

```
> qel$baseAcc
[1] 0.5053763
```

(Again, The small discrepancy is due to using the full dataset versus using just the training set.)

5.8 Polynomial Models

Recall Figure 1.1, reproduced below for convenience:



The trend seemed to be vaguely linear, upward, but maybe even a downward trend near the end. The latter point suggests there may even be a quadratic trend, with the middle-aged given higher ratings while the young and old are less generous.

Though it sounds counterintuitive, quadratic models are actually linear! Here is why: The model is

$$E(\text{userMean} \mid \text{age} = t) = \beta_0 + \beta_1 \text{ age} + \beta_2 \text{ age}^2 \tag{5.22}$$

This is quadratic in age but linear in β. For instance, if we were to multiply each β_i by 2.4, the entire sum would grow by that factor.

Indeed, we should just consider age^2 as a new feature:

```
> z <- ml100kpluscovs[,c('age','userMean')]
> z$age2 <- z$age^2
> head(z)
  age userMean age2
1  24 3.610294  576
2  20 3.918605  400
3  27 3.393720  729
4  25 3.470046  625
5  33 4.025271 1089
6  28 3.993007  784
```

We would then run **qeLin()** as usual:

```
> qeLin(z,'userMean')$testAcc
holdout set has  1000 rows
[1] 0.3254251
> qeLin(z[,1:2],'userMean')$testAcc
holdout set has  1000 rows
[1] 0.3472223
```

There may be a slight improvement of the quadratic model over the linear one. But we must remember that since the holdout set is randomly chosen, there is some randomness to the above numbers. Let's try 10 replications of each, using the **replicMeans()** function from the **regtools** package:

```
> replicMeans(10,"qeLin(z[,1:2],'userMean')$testAcc")
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
[1] 0.3363623
attr(,"stderr")
```

```
[1] 0.003186719
> replicMeans(10,"qeLin(z,'userMean')$testAcc")
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
holdout set has  1000 rows
[1] 0.3392382
attr(,"stderr")
[1] 0.003183424
```

(The *standard error* is the estimated standard deviation of the reported mean.)

The quad model probably is not helpful here.

A better example is the **pef** dataset from **regtools**.

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

This is data from the 2000 US census, in Silicon Valley, over six different computer-related occupations. Let's try a linear model:

```
> qeLin(pef,'wageinc')$testAcc
holdout set has  1000 rows
[1] 25906.99
```

On average, our prediction is off by almost $26,000. But it's better than not using the features at all, i.e. just using mean income as our predictor:

```
> qeLin(pef,'wageinc')$baseAcc
holdout set has  1000 rows
[1] 30804.21
```

Let's try a quadratic model. But it would be harder than (5.21). We would have to add columns for squares of all the variables (first converting the categorical variables like **occ** do dummies/one-hots), *and* computer "cross-product terms, such as age times sex (representing a difference in effect of age, between men and wome).

But **qeML** does all that for us, via **qePolyLin()**. So, let's try that:

```
> replicMeans(50,"qeLin(pef,'wageinc')$testAcc")
[1] 25361
attr(,"stderr")
[1] 145.1841
> replicMeans(50,"qePolyLin(pef,'wageinc',deg=2)$testAcc")
[1] 24762.21
attr(,"stderr")
[1] 136.9662
```

So, a quadratic model yields a small, but likely worthy, improvement.

There is also **qePolyLog()**, for quadratic features in a logistic model.

5.9 Application to Collaborative Filtering

So, let's make our first attempt at collaborative filtering, on the MovieLens data. We'll just use a linear model, with the small 100K dataset.

```
> qeLin(ml100kpluscovs[,c('user','item','rating')],'rating')$testAcc
holdout set has  1000 rows
[1] 0.9213201
> qeLin(ml100kpluscovs[,c('user','item','rating')],'rating')$baseAcc
holdout set has  1000 rows
[1] 0.9523251
> qeLin(ml100kpluscovs[,c('rating','userMean','itemMean')],'rating')$testAcc
holdout set has  1000 rows
[1] 0.7484943
```

Remember, **lm()**, and thus the wrapper **qeLin()**, will convert the **user** variable to dummies, and the same for **item**. With 943 users and 1682 items, that's 2624 dummies. A rough rule of thumb is that one should have at most $p < \sqrt{n}$, i.e. about 300 features. In other words, using the **user** and **item** columns is probably overfitting. Replacing them by the *embeddings*, i.e. replacing a variable by its summary or proxy, seems to be a good idea here; replacing 2624 columns by 2 seemed to pay off.

5.10 Regularization: the LASSO

A number of modern statistical methods “shrink” their classical counterparts. This is true for ML methods as well. In particular, the principle may be applied in:

- Boosting.
- Linear models.
- Support vector machines.
- Neural networks.

This is also applied to some RS models, such as matrix factorization.

5.10.1 Motivation

Suppose we have sample data on human height, weight and age. Denote the population means of these quantities by μ_{ht}, μ_{wt} and μ_{age}. We estimate them from our sample data as the corresponding sample means, $\overline{X}_{ht}$, $\overline{X}_{wt}$ and $\overline{X}_{age}$.

Just a bit more notation, giving names to vectors:

$$\mu = (\mu_{ht}, \mu_{wt}, \mu_{age}) \tag{5.23}$$

and

$$\overline{X} = (\overline{X}_{ht}, \overline{X}_{wt}, \overline{X}_{age}) \tag{5.24}$$

Amazingly, *James-Stein theory* says the best estimate of μ might NOT be $\overline{X}$.[5] It might be a shrunken-down version of $\overline{X}$, say $0.9\overline{X}$, i.e.

$$(0.9\overline{X}_{ht}, 0.9\overline{X}_{wt}, 0.9\overline{X}_{age}) \tag{5.25}$$

And the higher the dimension (3 here), the more shrinking needs to be done. (Though for 1 or 2 dimensions, shrinkage brings no improvement.) The intuition is this: For many samples, there are a few data points that are extreme, on the fringes of the distribution. These points skew our estimators, in the direction of being too large. So, it is optimal to shrink them.

[5] Here “best” means in terms of Mean Square Error. The MSE for the non-shrunken case is $E\left[||\overline{X} - \mu||^2\right]$.

How much shrinking should be done? In practice this is unclear, and typically decided by our usual approach, cross-validation.

Putting aside the mathematical theory — it's quite deep — the implication for us in this book is that, for instance, the least-squares estimator $\widehat{\beta}$ of the population coefficient vector β in the linear model is often too large, and should be shrunken. Most interestingly, **this turns out to be a possible remedy for overfitting**.

5.10.2 Shrinking $\widehat{\beta}$ in Linear Regression Models

Instead of finding the value of b that minimizes (5.11), we minimize

$$(D - Ab)'(D - Ab) + \lambda||b]]_1 \tag{5.26}$$

The value of the hyperparameter $\lambda > 0$ is up to the analyst, but is typically chosen by cross-validation. The resulting $\widehat{\beta}$ is called the Least Absolute Shrinkage and Selection Operator, LASSO.

Why does this result in shrinkage? In minimizing (5.11) we are given free rein, choosing any b that we want. But (**??**) penalizes us for using longer b. Actually, one can show that minimizing (**??**) is equivalent to minimizing (5.11) under the constraint that

$$||b||_1 \leq \eta \tag{5.27}$$

η is now the hyperparameter.

5.10.3 The Famous LASSO Picture

As mentioned, a key property of the LASSO is that it usually provides a *sparse* solution for $\widehat{\beta}$, meaning the many of the $\widehat{\beta}_i$ are 0. In other words, many features are discarded, thus providing a means of dimension reduction, thus an approach to avoiding overfitting. Figure **??** shows why. Here is how it works.

The figure is for the case of $p = 2$ predictors, whose coefficients are b_1 and b_2. (For simplicity, we assume there is no constant term b_0.) Let U and V denote the corresponding features. Write $b = (b_1, b_2)$ for the vector of the b_i.

Without shrinkage, we would choose b to minimize the sum of squared errors,

$$SSE = (Y_1 - b_1U_1 - b_2V_1)^2 + ... + (Y_n - b_1U_n - b_2V_n)^2 \tag{5.28}$$

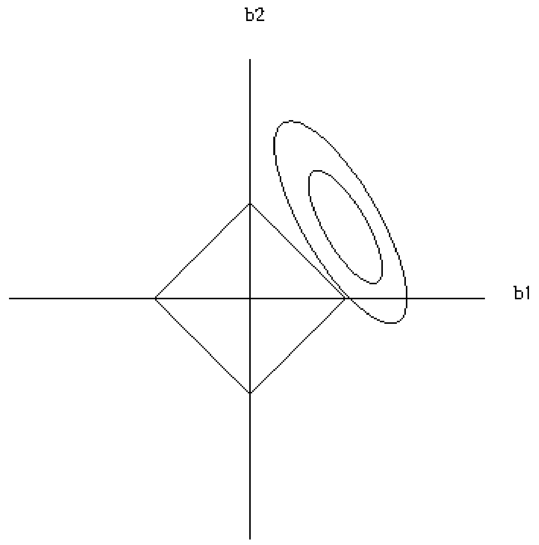


Figure 5.1: Feature subsetting nature of the LASSO

Recalling that the non-shrunken b is called the Ordinary Least Squares estimator, let's name that b_{OLS}, and name the corresponding SSE value SSE_{OLS}.

The horizontal and vertical axes are for b_1 and b_2, as shown. There is one ellipse for each possible value of SSE. For SSE equal to, say 16.8, the corresponding ellipse is the set of all points b that have SSE = 16.8. As we vary the SSE value, we get various concentric ellipses, two of which are shown in the picture. The smaller the ellipse, the smaller the value of SSE.

The OLS estimate is unique, so the ellipse for SSE = SSE_{OLS} is degenerate, consisting of the single point b_{OLS}.

The corners of the diamond are at $(\eta, 0)$, $(0, \eta)$ and so on. Due to the constraint (**??**), our LASSO estimator b_{LASSO} must be somewhere within the diamond.

Remember, we want SSE — the sum of square prediction errors — to be as small as possible, but at the same time we must stay within the diamond. So the solution is to choose our ellipse to just barely touch the diamond, as we see with the outer ellipse in the picture. The touchpoint is then our LASSO solution, b_{LASSO}.

But that touchpoint is sparse — $b2 = 0$! And you can see that, for almost any orientation and position of the ellipses, the eventual touchpoint will be one of the corners of the diamond, thus a sparse solution. Thus the LASSO will usually be sparse, which is the major reason for its popularity.

If we were to use the L2 norm in (**??**), that diamond would be a circle. The resulting $\widehat{\beta}$ is called the *ridge estimator*. It also shrinks, but is not sparse, since the touchpoint can be anywhere.

5.10.4 The qeLASSO Function

This wraps the **glmnet** package, which chooses λ through its own internal cross-validation.

Chapter 6

Statistics

It is assumed here that the reader has background in basic statistical inference, i.e. hypothesis testing and confidence intervals.

6.1 Sample vs. Population

Most readers have probably noticed that when the results of a survey are released, say during elections, a *margin of error* (MOE) is stated. For instance, "55% of those surveyed say they plan to vote for Candidate Jones, with a margin of error of 3.2%." The MOE is recognition of the fact that only a sample of voters were surveyed, not the entire population of voters.

Let p denote the population proportion, i.e. the proportion of voters across the population who favor Jones. The value of p is unknown, but our estimate is $\widehat{p} = 0.55$.

Assuming the voters were polled at random, $\widehat{p}$ is a random variable. As such, it has a standard deviation, which can be shown to be

$$[p(1-p)/n]^{0.5} \tag{6.1}$$

which we in turn estimate as

$$[\widehat{p}(1-\widehat{p})/n]^{0.5} \tag{6.2}$$

where n is th number of people polled. This is the *standard error* of $\widehat{p}$, and is a measure of how accurate $\widehat{p}$ is as an estimate of p.

What about the MOE? This is 1.96 times the standard error, and is the radius of an approximate 95% confidence interval for p.

Every ML method is an estimator in some form or other. However, the terms *sample* and *population* are not used in the ML community. Instead, they speak a *generative process* to mean the same thing as sampling data from a population.

6.2 How Do We Select an Estimator?

6.2.1 Sample Analogs

In our survey example above, our estimate $\widehat{p}$ is the sample analog of the population value p; the former is the proportion in our sample data and the latter is the proportion in the population.

Similarly, say we wish to estimate a population variance, say the variance of blood pressure $Var(B)$ in some patient population. The sample analog is

$$s^2 = \frac{1}{n} \sum_{i=1}^{n} (B_i - \overline{B})^2 \tag{6.3}$$

where the B_i are the blood pressures in our sample, and $\overline{B}$ is the average pressure in our sample. This is analogous to the population value,

$$Var(B) = E[(B - EB)^]\tag{6.4}$$

How about estimating a covariance matrix? Again, an extension of (6.3). Say our sample consists of n vectors $U_1, ..., U_n$. Then

$$\widehat{Cov}(U) = \frac{1}{n} \sum_{i=1}^{n} (U_i - \overline{U})(U_i - \overline{U})' \tag{6.5}$$

where here $\overline{U}$ is the average of the *vectors* U_i.

In many cases, the choice of estimator is less straightforward. For instance, say we wish to estimate the probability density of B, $f_B()$. Actually, you already know such an estimator—a histogram! As long as one scales to histogram to have total area 1.0 (in R's `hist()` function, set `probability=TRUE`), this provides an estimate of $f_B()$. Let's call the histogram $\widehat{f_B}()$.

6.2.2 Consistency of Estimators

What do we mean in saying that an estimator is an estimate of something? At the very least, we should require *consistent*, meaning that the estimator converges to the population quantity as the sample size grows.

For example, in the survey example, we have

$$\lim_{n \to \infty} \widehat{p} = p \tag{6.6}$$

But what about the histogram case? For the estimator to be consistent, we need not only $n \to \infty$ but also $h \to 0$, where h is the bin width.[1]

6.2.3 Unbiasedness of Estimators

Another classical criterion for goodness of an estimator is that it be *unbiasded*, which means the expected value of the estimator is the population quantity. In the survey example above, this criterion would ask that

$$E\widehat{p} = p \tag{6.7}$$

which in fact can be shown to be true.

But s^2 is more complicated. Books typically divide by $n-1$ instead of n in (6.3), as a "fudge factor" to make

$$E(s^2) = \sigma^2 \tag{6.8}$$

where the right-hand side denotes the population variance. But actually even with the fudge factor s (without the power 2) is biased:

$$0 < Var(s) = E(s^2) - (Es)^2 = \sigma^2 - (Es)^2 \tag{6.9}$$

so

$$Es < \sigma \tag{6.10}$$

[1]Technically, we cannot allow h to go to 0 "too quickly," but this issue is beyond the scope of this book.

rather than

$$Es = \sigma \tag{6.11}$$

At the time basic statistics was being developed—about 100 years ago—not much was known, so unbiasedness was considered important. It in fact is still useful in some settings, but today it's understood that we should not emphasize it so much.

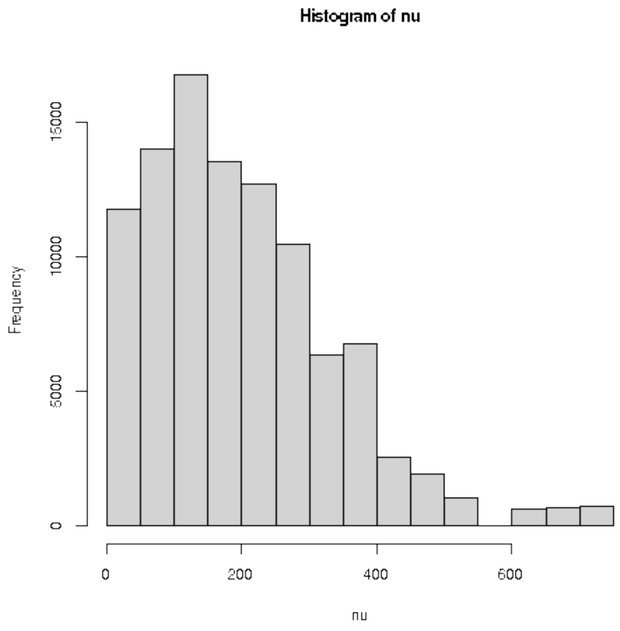
Indeed, a big issue in ML is the Bias-Variance Tradeoff. In the above context, dividing by $1/n$ instead of $1/(n-1)$ reduces variance, i.e. reduces $Var(s^2)$, which is good, at a cost of increasing bias, a tradeoff.

Actually, it can be shown that the best mean squared error for s^2 comes from dividing by $1/(n+1)$. We'll use n here to keep the analogy tight. Of course, for large n, the difference is negligible.

6.2.4 Fitting Parametric Models

It is often the case that one may try to fit a parametric family of densities to one's datat. Let's use MovieLens as an example.

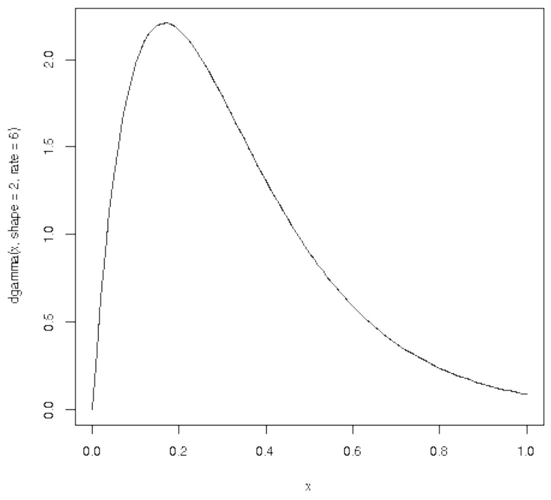
```
> nu <- ml100kpluscovs$Nuser
> hist(nu)
```



That shape seems to suggest a good fit from the *gamma* family of densities,

$$f_W(t) = \frac{1}{(r-1)!}\lambda^r t^{r-1} e^{-\lambda t}, \ t > 0 \tag{6.12}$$

A typical curve looks something like this:



The family has parameters r and λ.[2] For each value of the latter two quantities, we get a different curve, some flatter, some more peaked.

The question at hand is:

> Is there some (r, λ) pair such such that the curve (6.12) fits the data well?

The task is then to:

- Estimate r and λ from our data.
- Check the fit.

How do we do the estimation? Two standard methods are the Method of Moments (MM) and Method of Maximum Likelihood (MLE).

6.2.4.1 The Method of Moments

The idea is simple, at least in concept. Here's how it works in the gamma example above.

First, some terminology: For a random variable X, the k^{th} *moment* is $E(X^k),\ k = 1, 2, 3, ...$ A variant is the k^{th} *central moment*, $E[(X - EX)^k]$. Either the ordinary or central moments are fine.

Let N denote the number of ratings. Then for the gamma family,

$$EN = r/\lambda \tag{6.13}$$

and

$$Var(N) = r/\lambda^2 \tag{6.14}$$

Then replace everything by sample analogues:

$$\overline{X} = \widehat{r}/\widehat{\lambda} \tag{6.15}$$

and

$$s^2 = \widehat{r}/\widehat{\lambda}^2 \tag{6.16}$$

[2]Technically this is a subset of the gamma family, called *Erlang*, but the only restriction is that r be an integer.

Dividing the first equation by the second, we obtain

$$\widehat{\lambda} = \overline{N}/s^2 \tag{6.17}$$

and thus from the first equation,

$$\widehat{r} = \overline{N}\widehat{\lambda} = \overline{N}^2/s^2 \tag{6.18}$$

where $\overline{N}$ and s^2 are the sample mean and variance of $N_1, ..., N_n$ for n users.

Now, what about assessing the fit? A classic method is a *goodness of fit test*, but these days hypothesis testing is frowned upon, for good reason. Here's why:

Chapter 7

Introduction to Machine Learning Methods

Chapter 8

Clustering

Chapter 9

Principal Components Analysis

9.1 Multivariate Distributions, Covariance Matrices

9.2 An Approach to Approximate Rank: Principal Components Analysis

Suppose the matrix in (3.1) had been

$$M = \left(\begin{array}{rrrr} 1 & 5 & 1 & -2 \\ 8.02 & 2.99 & 2 & 8.2 \\ 9 & 8 & 3 & 6 \end{array} \right) \tag{9.1}$$

Intuitively, we still might say that the rank of M is "approximately" 2. And 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 (10.1).

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

9.2.1 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?

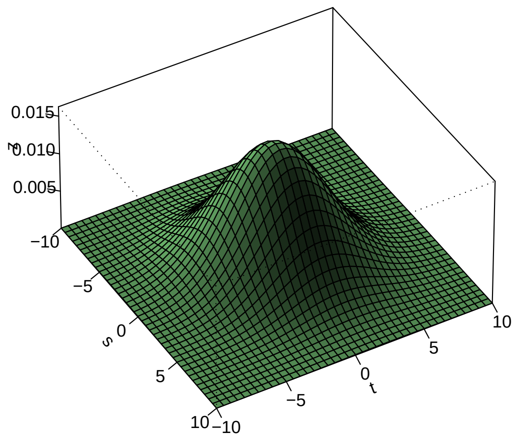
In the RS context, just think of, say, Amazon. The business has millions of users and millions of items. In other words, the ratings matrix has millions of rows and millions of columns, and even one million rows and columns would mean a total number of $(10^6)^2 = 10^{12}$ entries, about 8 terabytes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

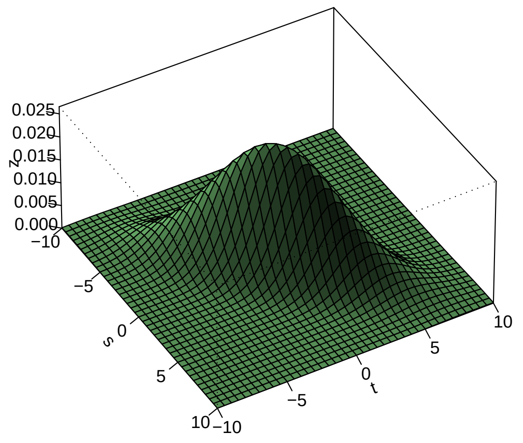
This is a core point in statistics/machine learning, the notion of *dimension reduction*. In complex applications, there is a pressing need to reduce the number of variables down to a manageable number — manageable not only in terms of computational time and space, but also the statistical problem of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

So we need methods to eliminate redundant or near-redundant data, such as row 3 in (9.1).

9.2.2 Exploiting Correlations

Statistically, the issue is one of correlation. In (9.1), 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. The graphs can be thought of as two-dimensional histograms, plotting frequency of occurrence of values (X_1, X_2) against those values.

Not surprisingly due to the high correlation in the second graph, the "two-dimensional bell" is concentrated around a straight line. Here we've generated the data so that the line is $t_2 = -t_1$.[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

$$Y_1 = X_1 + X_2 \tag{9.2}$$

and

$$Y_2 = X_1 - X_2 \tag{9.3}$$

have 0 correlation. We'll return to this point about correlation shortly, but if we are going to just user one variable instead of two, 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

[1]For height and weight in the `mlb` data in the `regtools` package, the line turns out to be about $t_2 = -151.133 + 4.783t_1$, but for simplicity, we're using $t_2 = -t_1$ as our example.

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:

$$Var(Y_1) > Var(Y_2) > ... > Var(Y_p) \tag{9.4}$$

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?

9.2.3 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

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{s.d.(x)\ s.d.(y)} \tag{9.5}$$

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*:

$$C' = C \tag{9.6}$$

where $'$ denotes matrix transpose.

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

$$Lx = \lambda x \tag{9.7}$$

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*,

$$U_i' U_j = 0, \ i \neq j \tag{9.8}$$

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.

9.2.4 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,

$$R = QU^{(s)} \tag{9.9}$$

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.

9.2.5 Applying Matrix Partitioning

Using the approach of Section 3.2, partition $U^{(s)}$ into its columns,

$$U^{(s)} = (U_1, ..., U_s) \tag{9.10}$$

and thus write

$$R = QU^{(s)} = Q(U_1, ..., U_s) \tag{9.11}$$

After that second equality, pretend that Q and the U_i are "numbers." Then the last expression in (9.11),

$$Q(U_1, ..., U_s) \tag{9.12}$$

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

$$(QU_1, ..., QU_s) \tag{9.13}$$

So, the i^{th} column of R is QU_i. The latter quantity is of the Ax, matrix-times-vector form of Section 3.2.3, 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.

9.2.6 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

$$\sum_{j=1}^{s} \lambda_j \tag{9.14}$$

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 **??**.

9.2.7 Software and UCI Repository 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.

9.2.8 MovieLens Example

9.2.9 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.

9.2.10 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.

Chapter 10

Matrix Factorization Methods

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

10.1 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

$$A \approx WH \tag{10.1}$$

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

$$k \ll \text{ rank}(A) \tag{10.2}$$

10.2 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.

10.3 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

10.4 Synthetic, Representative Recommender Systems Users

Note the key relation, which we showed in Section 3.2:

$$(WH)_{i \cdot} = \sum_{m=1}^{k} W_{im} H_{m \cdot} \tag{10.3}$$

In other words, in (10.1), 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.

10.5 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.

10.6 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.

10.6.1 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, $A_{i\cdot}$ stores the data for the i^{th} image, say in row-major order:[2] $A_{i\cdot}$ 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 $A \rightarrow W$. 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 $S, i.e.$

$$\lambda = \arg\min_l ||S - l'H|| \tag{10.4}$$

(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

$$k \ll \text{ rank}(A) \tag{10.5}$$

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 10.10.

10.6.2 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.

10.7 The R Package NMF

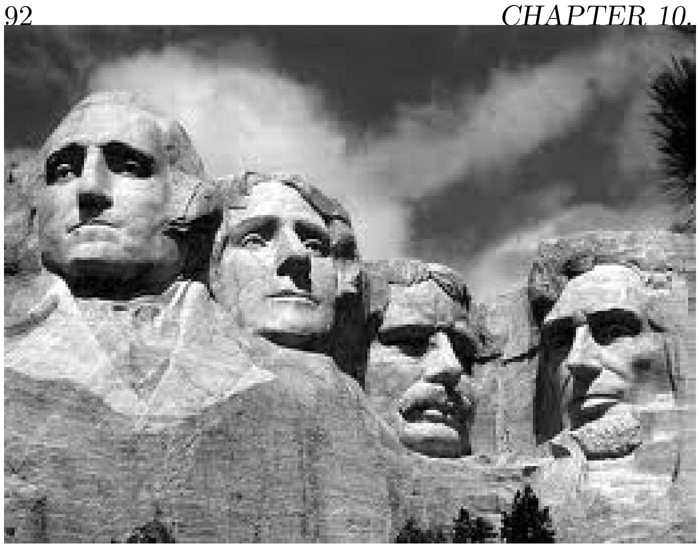
The R package **NMF** is quite extensive, with many, many options. In its simplest form, though, it is quite easy to use. For a matrix **a** and desired rank **k**, we simply run

```
> 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:



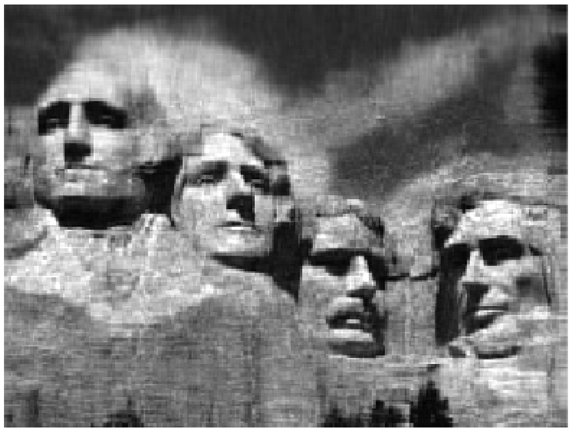
Here we have only one image, and we'll store it as a matrix A (rows of the matrix corresponding to rows of A). we'll use NMF to compress it, not do classification. First obtain A:

```
> 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.

10.8 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 **??** 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

$$nk + km = k(n+m) \tag{10.6}$$

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 $\widehat{A}_{ij}$.

10.9 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'**.

10.9.1 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 3.3),

$$\|Q\|_2 = \sqrt{\sum_{i,j} Q_{ij}^2} \tag{10.7}$$

So our criterion for error of approximation will be

$$\|A - WH\|_2 \tag{10.8}$$

So, we choose W and H to be

$$\arg\min_{w,h} \|A - wh\|_2 \tag{10.9}$$

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

Note that we can write (10.7) as

$$\|Q\|_2 = \sqrt{\sum_j \left(\sum_i Q_{ij}^2 \right)} \tag{10.10}$$

This mean that if we can separately minimize that inner sum, for each j, we will have minimized the entire expression (10.10). Our strategy will depend on this.

10.9.2 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

$$\|A_{\cdot j} - WH_{\cdot j}\|_2 \tag{10.11}$$

We are seeking to find $H_{\cdot j}$ that minimizes (10.7), 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 10.16. We are "predicting" $A_{\cdot j}$ from W,

So again in the notation of Section **??**:

- 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,

$$A' = H'W' \tag{10.12}$$

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.

10.9.3 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 (10.11), 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

$$A_{.4} = \begin{pmatrix} NA \\ 3 \\ NA \\ 8 \\ 2 \end{pmatrix} \tag{10.13}$$

Then in (10.11) we replace $A_{.5}$ by

$$\begin{pmatrix} 3 \\ 8 \\ 2 \end{pmatrix} \tag{10.14}$$

Also, replace W by

$$\begin{pmatrix} w_{21} & w_{22} & w_{23} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix} \tag{10.15}$$

Remember, at this stage, W is assumed known. So, we just use **lm()**, "predicting" (10.14) from (10.15) to find $h_{.4}$.

10.9.4 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 (10.8). 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.

10.10 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 3.

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

10.11 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 10.4. 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 $H_{i.}$ to the ratings scale, e.g. 1 to 5 for the MovieLens data.

10.12 "Bias" Removal

As noted in Chapter 11, 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

$$A_{uv} \leftarrow A_{uv} - R_u - S_v \tag{10.16}$$

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

$$A_{uv} \leftarrow A_{uv} + R_u + S_v \tag{10.17}$$

10.13 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*,[8] 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
```

[8]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 $W_{i\cdot}$ and $H_{\cdot j}$, 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.

10.14 Regularization

Recall Section **??**, 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 (10.8), we minimize

$$||A - WH||_2 + \gamma_1 ||W||_2^2 + \gamma_2 ||H||_2^2 \tag{10.18}$$

Both γ_1 and γ_2 are tuning parameters.

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

10.15 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
```

10.16 How Do We Minimize (10.4)?

The answer, as it was in Section 10.9, 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 (10.4),

$$||S - l'H||^2 = (S - l'H)(S - l'H)' \tag{10.19}$$

This looks pretty close to (**??**). But recall that S and $l'H$ are row vectors, so (10.19) looks slightly different. Now, using the fact from linear algebra that $(UV)' = V'U'$, (10.19) says

$$||S - l'H||^2 = (S' - H'l)(S' - H'l) \tag{10.20}$$

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

$$\lambda = (HH')^{-1}HS' \tag{10.21}$$

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 11

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.

11.1 kNN

Let's see how kNN works.

11.1.1 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,[1] 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.

11.1.2 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.

11.1.3 (One) Implementation

Below is code from **rectools** (somewhat simplified).[2] The arguments are:

- **origData:** The original dataset, after having been run through **formUserData()**.
- **newData:** The element of **origData** for NC.[3]
- **newItem:** ID number of the item to be predicted for NC.
- k: The number(s) of nearest neighbors. Can be a vector.

Here is an example of using **formUserData()** on the MovieLens data:[4]

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

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

[3]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.

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

```
> 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,...

```
predict.usrData <- function(origData,newData,newItem,k)
{
# we first need to narrow origData down to the users who
# have rated newItem

# here oneUsr is one user record in origData; the function will look for a
# j such that element j in the items list for this user matches the item
# of interest, newItem; (j,rating) will be returned

checkNewItem <- function(oneUsr) {
   whichOne <- which(oneUsr$itms == newItem)
   if (length(whichOne) == 0) {
      return(c(NA,NA))
   } else return(c(whichOne,oneUsr$ratings[whichOne]))
}

found <- as.matrix(sapply(origData,checkNewItem))
# description of 'found':
# found is of dimensions 2 by number of users in training set
# found[1,i] = j means origData[[i]]$itms[j] = newItem;
# found[1,i] = NA means newItem wasn't rated by user i
# found[2,i] = rating in the non-NA case

# we need to get rid of the users who didn't rate newItem
whoHasIt <- which(!is.na(found[1,]))
origDataRatedNI <- origData[whoHasIt]
# now origDataRatedNI only has the relevant users, the ones who
# have rated newItem, so select only those columns of the found matrix
found <- found[,whoHasIt,drop=FALSE]

# find the distance from newData to one user y of origData; defined for
# use in sapply() below
onecos <- function(y) cosDist(newData,y,wtcovs,wtcats)
cosines <- sapply(origDataRatedNI,onecos)
# the vector cosines contains the distances from newData to all the
# original data points who rated newItem

# action of findKnghbourRtng(): find the mean rating of newItem in
# origDataRatedNI, for ki (= k[i]) neighbors
#
# if ki > neighbours present in the dataset, then the
# number of neighbours is used
findKnghbourRtng <- function(ki){
  ki <- min(ki, length(cosines))
  # nearby is a vector containing the indices of the ki closest neighbours
  nearby <- order(cosines,decreasing=FALSE)[1:ki]
  mean(as.numeric(found[2, nearby]))
}
sapply(k, findKnghbourRtng)
}
```

11.1.4 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

$$\frac{u'v}{||u||_2 \ ||v||_2} \tag{11.1}$$

This not really a distance,[5] 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.

11.1.5 Regression Analog

Recall the method of k-nearest neighbor (kNN) regression estimation from Chapter **??**, 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

11.1.6 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.

11.1.7 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.

[5]IN math terms, it’s not a *metric*.

11.1.8 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.

Chapter 12

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.

12.1 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

$$Y_{IJ} = \mu + \alpha_I + \beta_J + \epsilon_{IJ} \tag{12.1}$$

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.

12.2 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.

12.2.1 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

$$P(K = u) = \binom{u-1}{r-1} 0.5^u, \ u = r, r+1, ... \tag{12.2}$$

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 $K_1, K_2, ..., K_n$.

12.2.2 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,

$$E(K) = \frac{r}{0.5} = 2r \tag{12.3}$$

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 (12.3) by sample estimates, in this case

$$\overline{K} = 2\widehat{r} \tag{12.4}$$

where

$$\overline{K} = \frac{K_1 + ... + K_n}{n} \tag{12.5}$$

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.

12.2.2.1 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

$$\Pi_{i=1}^n \binom{K_i}{w-1} 0.5^{K_i} \tag{12.6}$$

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 $\widehat{r}$.

12.2.2.2 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.

12.3 MM Applied to (12.1)

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

12.3.1 Derivation of the Estimates

The expected values in Section 12.2.2 can be conditional. So, from (12.1), write

$$E(Y_{IJ} \mid I = k) = \mu + \alpha_k + E(\beta_J | I = k), \; k = 1, 2, ..., u \tag{12.7}$$

But since I and J are independent, we have

$$E(\beta_J | I = k) = E(\beta_J) = 0, \; k = 1, 2, ..., u \tag{12.8}$$

so

$$E(Y_{IJ} \mid I = k) = \mu + \alpha_k, \; k = 1, 2, ..., u \tag{12.9}$$

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.

12.3.2 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 **??**.

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

$$(A'A)^{-1} = \text{diag}(\frac{1}{n_1}, ..., (\frac{1}{n_u}) \tag{12.10}$$

a diagonal matrix with the indicated elements.

What about $A'D$ in (**??**)? 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!

12.4 MLE Applied to (12.1)

Chapter 13

PageRank

Many attribute the early success of Google to its superior search engine, based on an algorithm they called PageRank. It was assigned US Patent US6285999B1, with the sole inventor listed as Larry Page, one of the two founders of the firm. However, Page credited co-founder Sergei Brin as contributing in conversations about the idea, as well as some Stanford University faculty and some previous researchers who had done related work. The patent belongs to Stanford. The name, *PageRank*, is a pun on both the inventor's surname and the fact that it indexes Web *pages*.

The goal is to rank all the pages of thw World Wide Web in order of importance. The key is defining that latter term.

13.1 Markov Model

The method models Web "surfing" as following a *Markov chain*, a very widely-used approach to stochastic modeling in many fields. Its main assumption is rather strict, but in this case that is not very important, as we are merely putting together a rough measure of the importance of each Web page.

13.1.1 Structure

One observes some process at times 1,2,3,... The *state* at time *m*, a random variable, is denoted by X_m. Different applications have different state spaces; it could be queue length in a network buffer, for instance.

The main assumption is that the system follows the *Markov property*, which in rough terms can be described as:

> The probabilities of future states, given the present state and the past states, depends only on the present state; the past is irrelevant.

We sometimes say the system is "memoryless" — in looking to the future, knowing the present, the past is "forgotten."

A famous example is *random walk*. Say at time 0 we are at position 0 on the real number line. We flip a coin, then go 1 step left or right, depending on whether the outcome is head or tails. Then we do that again, repeatedly. X_m is our position at time m. Clearly this system is Markovian.

In formal terms:

$$P(X_{t+1} = s_{t+1} | X_t = s_t, X_{t-1} = s_{t-1}, \ldots, X_0 = s_0) = P(X_{t+1} = s_{t+1} | X_t = s_t) \tag{13.1}$$

13.1.2 Matrix Formulation

We define p_{ij} to be the probability of going from state i to state j in one time step; note that this is a *conditional* probability, i.e. $P(X_{n+1} = j \mid X_n = i)$. These quantities form a matrix, denoted by P.

So, this matrix gives the probabilites of going from state i to state j in one time step:

$$P(X_{n+1} = s \mid X_n = r) = P_{rs} = p_{rs} \tag{13.2}$$

And the Markov property can be used to show that P^2 gives the probabilities of the various 2-time step transitions, i.e.

$$P(X_{n+2} = s \mid X_n = r) = (P^2)_{rs} \tag{13.3}$$

Similarly, P^k gives the k-step probabilities,

$$P(X_{n+k} = s \mid X_n = r) = (P^k)_{rs} \tag{13.4}$$

13.1.3 Long-Run Behavior

Suppose the chain is *irreducible*, meaning that one can reach every state from every other, in a finite number of steps.[1] Suppose also that the state space is finite, and *aperiodic*. The latter term

[1]Computer scientists, true to form, have their own separate term for this, *strongly connected*.

means that the GCD of return times to any state is 1; the random walk is periodic, since that number is 2. Then

$$\pi_r = \lim_{m\to\infty} P(X_m = r) \tag{13.5}$$

exists, and is independent of the starting position X_0.

Note that π is a vector, with element r defined as above.

13.1.4 Finding π

(13.5) and (13.4), and our by-now well-honed skill at using partitioned matrices imply that

$$\lim_{m\to\infty} P^m = \left(\begin{array}{c} \pi' \\ \pi' \\ ... \\ \pi' \end{array} \right) \tag{13.6}$$

Note that the fact the rows are identical here reflects the independence of (13.5) of the starting position X_0.

This gives us a way to find π: Simply raise P to a high power, and then each row is approximately π'. In fact, we could the get an even better approximation by averaging those rows. (Of course, this is for chains with finite state spaces.)

Alternatively, one can view the whole thing as an eigenvalue problem. The above material can be used to show that

$$\pi' = \pi P \tag{13.7}$$

so that

$$\pi' = P'\pi' \tag{13.8}$$

which makes π' an eigenvector of P' with eigenvalue 1. It's also possible to show that the other eigenvalues are between 0 and 1. So an alternate way to find π is to find the eigenvectors of P'. And in turn one way to do that is to use the famous Power Method for finding the eigenvector corresponding to the largest eigenvalue of a matrix. That method involves powers of P as well, so in the end it's similar to the above.

13.2 The PageRank Model

Here the state space consists of one state for every page on the Web, and "time" is hops from one page to another. X_3, say, is the third page you visit during a Web surfing session. What Page had to do was devise some clever formulation of the matrix P that would reasonably model Web surfing.

13.2.1 Details

The Markov property is questionable here, but again, we don't care much about that. It's just a mechanism for devising an importance measure.

Now, what is the matrix P here? For each Web page i, let n_i denote the number of outlinks from that page. The model is that after a visit to that page, the Web surfer will choose one of those links, with uniform probability among them, i.e. $1/n_i$ each. The π vector then becomes the vector of importance scores for the various pages.

Google also adds a *damping* factor to the model, which we will not go into here.

13.2.2 Computation of π

All of the above is fine, but there are literally billions of Web pages. Thus the matrix P has billions of rows and columns. Fortunately, though, it is a very sparse matrix, lots of 0s, so we can exploit that property to reduce computation.

Note too that there is a shortcut to find the matrix powers: We first square P, then square the result, yielding P^4, then square *that* result, yielding P^8 and so on.

There are many refinements of all this, of course.

13.3 Then What?

Clearly there is a lot more to what Google does to provide users with recommended Web page. What we saw here is only a first step.