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

2.3.8 More on the PC Coefficients

There is more to consider.

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

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

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

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

Let's apply PCA:

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

If we were to plot **hw**, we would put **hw[1,]** at the point (74,1.0) on our graph. Recall from high school math that 74 and 180 are called the *coordinates* of **co2[1,]**, with respect to our "H axis" and "W axis."

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

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

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

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

Almost 90 degrees clockwise.

2.4 Vector Norms

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

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

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

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

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

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

3.4.1 What Is Overfitting?

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

```
> x <- rnorm(6)
> y <- rnorm(6) # unrelated to x!
> df <- data.frame(x,y)
> df$x2 <- x^2
> df$x3 <- x^3
> df$x4 <- x^4
> df$x5 <- x^5
> df
           x           y          x2            x3
1 -1.1855131  0.2881291 1.40544120 -1.666168894
2 -1.7838769 -2.0741740 3.18221664 -5.676682627
3 -0.7124510 -0.4253678 0.50758640 -0.361630431
4  0.1676111 -0.1949265 0.02809348  0.004708779
5  1.2462926 -0.7348481 1.55324535  1.935798245
6  0.3741604  1.9521667 0.13999601  0.052380963
            x4            x5
1 1.975265e+00  -2.341702414
2 1.012650e+01 -18.064433938
3 2.576440e-01  -0.183558689
4 7.892437e-04   0.000132286
5 2.412571e+00   3.006769615
6 1.959888e-02   0.007333126
> lmo <- lm(y ~ .,data=df)
> lmo

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

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

```
          6
 1.9521667
> y
[1]  0.2881291 -2.0741740 -0.4253678 -0.1949265 -0.7348481
[6]  1.9521667
```

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

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

On the other hand, we need to worry about sampling variance. Consider the case of opinion polls during an election campaign, in which the goal is to estimate p, the proportion of voters who will vote for Candidate Jones. If we use too small a sample size, say 50, our results will probably be inaccurate. This is due to sampling instability: Two pollsters, each randomly sampling 50 people, will sample different sets of people, thus each having different values of $\widehat{p}$, their sample estimates of p. For a sample of size 50, it is likely that their two values of $\widehat{p}$ will be substantially different from each other, whereas if the sample size were 5000, the two estimates would likely be close to each other. In other words, the variance of $\widehat{p}$ is too high if the sample size is just 50.[11]

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

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

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

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

3.4.2 Can Anything Be Done about It?

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

Rough Rule of Thumb:

One quick rule, backed up by mathematical theory, is that one should have $p < \sqrt{n}$, where p is the number of predictors, including polynomial and interaction terms (not to be confused with the quantity of the same name in our polling example above), and n is the number of cases in our sample. But this is certainly not a firm rule by any means, and I find it tends to be overly conservative.

Cross-Validation:

From the polynomial-ftting example in Section 3.4.1, we see the following key point:

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

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

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

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

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

Regularization:

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

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

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

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

- If $p \leq 2$, then $\overline{X}$ is the optimal estimator of μ.[13]
- If $p \geq 3$, then the optimal estimator is $c\overline{X}$ for some $0 < c < 1$.

  h

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

This was later applied to linear regression models, PCA and so on. Instead of finding b that minimizes (3.5), we minimize

$$r = \sum_{i=1}^{n} [W_i - (b_0 + b_1 H_i + b_2 A_i)]^2 + \lambda ||b||_1 \tag{3.19}$$

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

$$||b|| = \sum_{i=1}^{p} |b_i| \tag{3.20}$$

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

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

[13] Under Mean Squared Error loss.

3.4.3 The Problem of P-hacking

The (rather recent) term *p-hacking* refers to the following abuse of statistics.[14]

Say we have 250 pennies, and we wish to determine whether any are unbalanced, i.e. have probability of heads different from 0.5. We do so by tossing each coin 100 times. If we get fewer than 40 heads or more than 60, we decide this coin is unbalanced.[15] The problem is that, even if all the coins are perfectly balanced, we eventually will have one that has fewer than 40 or greater than 60 heads, just by accident. **We will then falsely declare this coin to be unbalanced.**

Or, to give a somewhat frivolous example that still will make the point, say we are investigating whether there is any genetic component to a person's sense of humor. Is there a Humor gene? There are many, many genes to consider. Testing each one for relation to sense of humor is like checking each penny for being unbalanced: Even if there is no Humor gene, then eventually, just by accident, we'll stumble upon one that seems to be related to humor.[16]

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

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

3.5 Extended Example

Let's illustrate this on the dataset **prgeng**, assembled from the 2000 US census. It consists of wage and other information on 20090 programmers and engineers in Silicon Valley. This dataset

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

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

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

is included in the R **polyreg** package, which fits polynomial models as we saw in Section 3.3.5.1 above.[17]

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

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

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

```
library(regtools)
library(polyreg)
data(prgeng)
pe <- prgeng[,c(1:4,6,5)]
head(pe)
tstidxs <- sample(1:nrow(prgeng),1000)
petrn <- pe[-tstidxs,]
petst <- pe[tstidxs,]
for (i in 1:4) {
   pfout <- polyFit(petrn,deg=i)
   preds <- predict(pfout,petst)
   print(mean(abs(preds-petst$wageinc)))
   print(length(pfout$fit$coefficients))
}
```

And the resulting Mean Absolute Prediction Errors and p Values :

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

Remember, p is the number of predictors, including all the dummies. When we have a degree 2

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

model, we have all the squared terms (except for the dummies), and the cross-product terms, e.g. interaction between age and gender. So p increase pretty rapidly with degree.

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

Chapter 4

Matrix Factorization Methods

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

$$A \approx WH \tag{4.1}$$

Again, most of the elements of A are unknown. But because it is typically the case that similar users have similar ratings patterns, we hope to obtain good estimates of all of A even though only a small part of it is known.

This is a form of dimension reduction, with the rank k controlling the bias-variance tradeoff. Typically a good approximation can be achieved with

$$k \ll \text{ rank}(A) \tag{4.2}$$

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.

4.1 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

4.2 Synthetic, Representative RS Users

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

$$(WH)_{i\cdot} = \sum_{m=1}^{k} W_{im} H_{m\cdot} \tag{4.3}$$

In other words, in (4.1), we have that:

(a) The entire vector of predicted ratings by user i can be expressed as a linear combination of the rows of H.

(b) Thus the rows of H can be thought of as an "approximate basis" for the rows of A.[1]

(c) 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.[2]

[1] Recall that the term *basis* in linear algebra 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.

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

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.

4.3 The Case of Entirely Known A

This notion of *nonnegative matrix factorization* has become widely used in a variety of ML applications in which the matrix A entirely known.

4.3.1 Image Classification

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. Thus the 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 classses of new images. Denote the class of image j in our original data by C_j.

We form a matrix A with n rows and $w = rc$ columns, where the i^{th} row, $A_{i.}$ stores the data for the i^{th} image, say in row-major order:[3] $A_{i.}$ would first store row 1 of that image, then store row 2 of the image, and so on.[4]

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.

So, just as in the PCA case, we transform our training data, via $A \to W$. We apply our favorite classification method, say 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 $S, i.e.$

$$\arg\min_l ||S - l'H|| \tag{4.4}$$

[3] Make sure not to confuse the rows of A with the rows of an image.

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

When then set λ to the minimizing l (we'll see how, shortly), and this is the coordinate vector for the new case.

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

$$k \ll \text{ rank}(A) \tag{4.5}$$

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 4.6.

4.3.2 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.

4.3.3 Relation to Recommender Systems

Many RS methods are text-based or even image-based. Say there is a new movie, not user ratings yet at all. One might compare the movie studio's synopsis of the film with those of flims for which we have ratings data, and try to predict how well each user would like this film.

A more ambiitious approach would be to do the same for images in the film's ad trailer.

4.3.4 How Do We Minimize (4.4)?

To answer this question, 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 70000x0;
- H is 50x784;
- S is 1x784; and

- l is 50x1.

Keeping these numbers in mind as concrete examples, now note that in (4.4),

$$||S - l'H||^2 = (S - l'H)(S - l'H)' \tag{4.6}$$

This looks pretty close to (3.6). But recall that S and $l'H$ are row vectors, so (4.6) looks slightly different. Now, using the fact from linear algebra that $(UV)' = V'U'$, (4.6) says

$$||S - l'H||^2 = (S' - H'l)(S' - H'l) \tag{4.7}$$

Now it's in the form of (3.6), so our minimization problem is solved! In (3.11), just set D, A and b to S', H' and l, respectively. This gives us

$$\lambda = (HH')^{-1}HS' \tag{4.8}$$

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
```

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

4.4 The R Package NMF

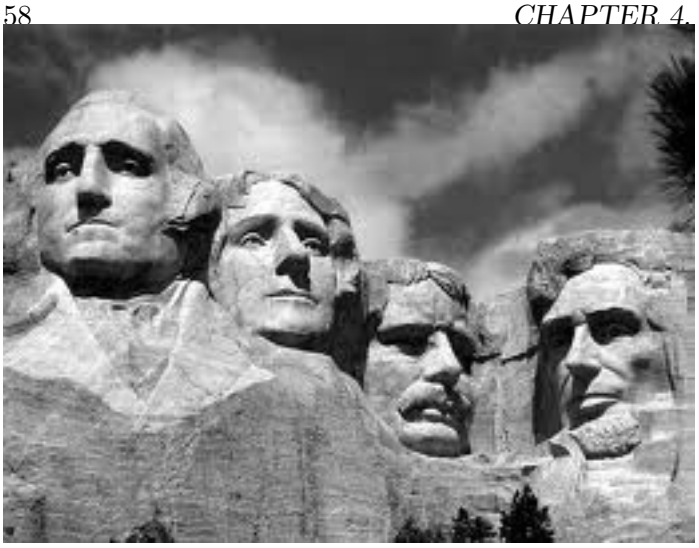
The R package **NMF** is quite extensive, with many, many options. In its simplest form, though, it is quite easy to use. For a matrix **a** and desired rank **k**, we simply run

```
> 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:



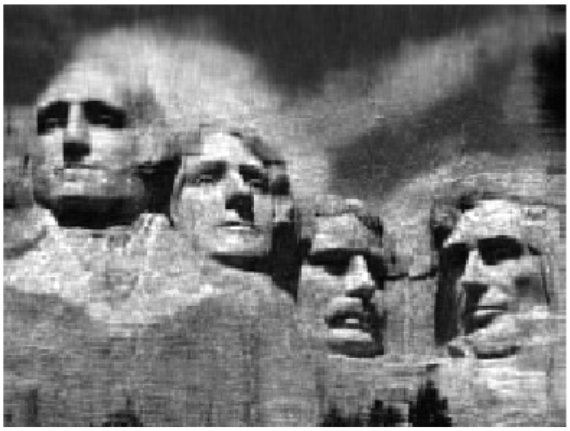
Here we have only one image, and we'll store it as a matrix A (rows of the matrix corresponding to rows of A). we'll use NMF to compress it, not do classification. First obtain A:

```
> 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.

4.5 Computation

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

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

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.

4.5.1 Objective Function

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

$$\|Q\|_2 = \sqrt{\sum_{i,j} Q_{ij}^2} \tag{4.9}$$

So our criterion for error of approximation will be

$$\|A - WH\|_2 \tag{4.10}$$

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

4.5.2 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

$$\|A_{\cdot j} - WH_{\cdot j}\|_2 \tag{4.11}$$

We are seeking to find $H_{\cdot j}$ that minimizes (4.9), 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 4.3.4. We are "predicting" $A_{\cdot j}$ from W,

So again in the notation of Section 3.3.5:

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

- 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,

$$A' = H'W' \tag{4.12}$$

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.

4.5.3 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. In (4.11), simply replace $A_{\cdot j}$ by the known elements of that vector, and replace W by the corresponding rows. Then proceed as before.

4.5.4 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 (4.10).[8] In fact, one can see this immediately: Doubling W while having H leaves

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

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

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.

4.6 How Do We Choose the Rank?

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

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

4.7 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.

4.8 Functions in rectools

The **recoSystem** package by Chih-Jen Li *et al* has a good implementation of NMF for recommender systems (i.e. it handles the missing values). However, it uses R6 classes (and is complicated in other

ways), which are unfamiliar to many R users, so the **rectools** package provides wrappers. Basic call forms are:

```
trainReco(ratingsIn,rnk=10,nmf=FALSE)
predict.RecoS3(recoObj,testSet)
```

Here **ratingsIn** is the usual three-column (userID,itemID,rating) (plus covariates, if any) data frame; **rnk** is the desired rank; **nmf** is false for SVD, true for NMF; **recoObj** is the return value from **trainReco()**; and **testSet** is the test set, in the same format as **ratingsIn**, minus the ratings column.

The return value from **trainReco()** is an R S3 object of class 'RecoS3', with components P and Q, corresponding to W and H' in our notation. In the case of covariates (see below), some R **attributes** are also included.

4.9 Dealing with Covariates

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

Here are relevant code excerpts:

trainReco():

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

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

```
    attr(result, "minResid") <- minResid
}
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.

4.10 Regularization

Recall Section 3.4.2, 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 (4.10), we minimize

$$||A - WH||_2 + \gamma_1 ||W||_2^2 + \gamma_2 ||H||_2^2 \tag{4.13}$$

Both γ_1 and γ_2 are tuning parameters.

4.11 "Bias" Removal

As noted in Chapter 5, 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

$$A_{uv} \leftarrow A_{uv} - R_u - S_v \tag{4.14}$$

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

$$A_{uv} \leftarrow A_{uv} + R_u + S_v \tag{4.15}$$