CHAPTER 5

Recommender Systems in R

Saurabh Bhatnagar
New York, USA
http://sanealytics.wordpress.com/

5.1 Introduction

Recommender systems are pervasive. You have encountered them while buying a book on www.barnesandnoble (if you like this. . .), renting a movie on Netflix, listening to music on Pandora, finding the bar to visit (FourSquare), to possibly even finding love at match.com. In this chapter, I show you how easy it is to write a recommendation engine in *R* and more importantly, test it out with real-world data to make sure it works. Recommendation systems help users find the right choices in an increasingly complex domain. But there are places where they fall short (black sheep, cold start). Later in the chapter, we talk about how to address those issues.

5.2 Business Case

Many retailers carry a myriad of products. It becomes an issue of matching the right product to the right person. Imagine going to a small wine shop, where the sommelier knew what you had purchased and liked earlier, and could recommend other wines for your taste. He might ask you a few questions (white or red today? fish or meat? meal or table?) but he will quickly try to get you that perfect bottle for the evening. Now no sommelier can keep track of thousands of transactions per day from millions of customers on a website. This expert system is a recommender engine.

5.3 Evaluation

Before we talk about recommendation systems, we have to talk about what qualities we are looking for. This will help us judge which recommendation system works better than the others given our data. A few metrics used commonly are:

RMSE (Root Mean Squared Error): Here, we measure how far real ratings are from the ones we predicted. Mathematically, we can write it out as

Data Mining Applications with R

$$\text{RMSE} = \sqrt{\frac{\sum_{(i,j)\in\kappa}\left(r_{(i,j)} - \hat{r}_{(i,j)}\right)^2}{|\kappa|}}$$

where k is the set of all user-item pairings (i, j) for which we have a predicted rating $\hat{r}_{(i,j)}$ and a known rating $r_{(i,j)}$, which was not used to learn the recommendation model.

Precision/Recall/f-value/AUC: Precision tells us how good the predictions are. In other words, how many were a hit.

$$\text{precision} = \left|\frac{\{\text{relevantdocuments}\} \cap \{\text{retrieveddocuments}\}}{\{\text{retrieveddocuments}\}}\right.$$

Recall tells us how many of the hits were accounted for, or the coverage of the desirable outcome.

$$\text{recall} = \left|\frac{\{\text{relevantdocuments}\} \cap \{\text{retrieveddocuments}\}}{\{\text{relevantdocuments}\}}\right.$$

Precision and recall usually have an inverse relationship. This becomes an even bigger issue for rare issue phenomenon like recommendations. To tackle this problem, we will use *f*-measure. This is nothing but the harmonic mean of precision and recall.

$$f = \text{value} = \frac{2 * \text{precision.recall}}{\text{precision} + \text{recall}}$$

Another popular measure is AUC. This is roughly analogous. The higher the AUC, the better we did at guessing what the user actually picked. It does not take ratings into account. Since this is more common, we will use it for our recommendation effectiveness. We will plot ROC on true positive rate vs. false positive rate. A true positive is when the recommended item was picked by the user. A false positive is when a recommended item was not picked by the user.

ARHR (Hit Rate): When returning a ranked list

$$\text{ARHR} = \frac{1}{\#\text{users}}\sum_{i=1}^{\#\text{hits}}\frac{1}{p_i}$$

where p is the position of the item in a ranked list.

There are other factors to consider besides just these metrics, like speed of recommendation, precalculation or post. Others are more esoteric like serendipity. These will make more sense as we go over some of these methods.

5.4 Collaborative Filtering Methods

Now for the fun part. If my friend Jimmy tells me that he liked the movie "Drive," I might like it too as we have similar tastes. However, if Paula tells me she liked "The Notebook," I might avoid it because we usually don't like the same movies. This is called UBCF (User-based

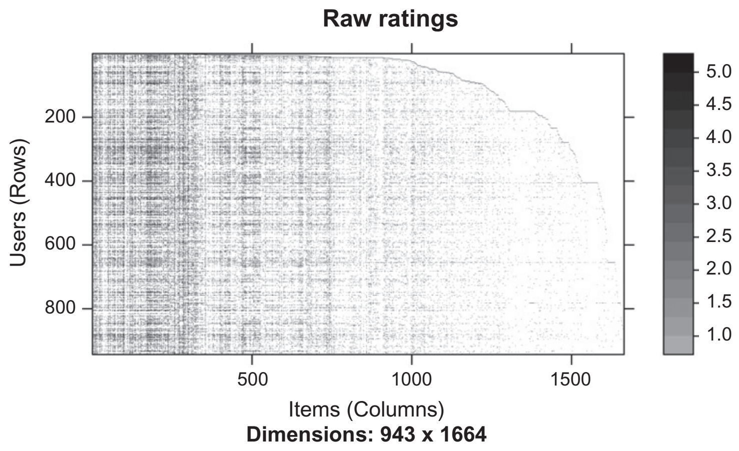

Figure 5.1
Raw ratings of MovieLense database.

collaborative filtering). Another way to think about it is soft-clustering. We find Users with similar tastes (neighborhood) and use their preferences to build yours (Figure 5.1).

Another flavor of this is IBCF (Item-Based Collaborative Filtering). If I watched "Darjeeling Limited," I might be inclined to watch "The Royal Tannenbaums" but not necessarily "Die Hard." This is because the first two are more similar in the users who have watched/rated them. This is rather simple to compute as all we need is the covariance between products to find out what this might be. You have seen this at Amazon (if you like this).

Let's compare both approaches on some real data (thanks R). We will use the packages recommenderlab for evaluation (Hahsler, 2011) and ggplot2 for graphics (Wickham, 2009).

```
> # Load required library
> library(recommenderlab)
> library(ggplot2) # For plots
> # Load the data we are going to work with
> data(MovieLense)
> MovieLense

943 × 1664 rating matrix of class âĂŸrealRatingMatrixâĂŹ with 99392 ratings.

>
> # Visualizing a sample of this
> image(MovieLense, main = "Raw ratings")
```

It appears that some movies were not rated at all by first few users. Maybe they were released later (Figures 5.2 and 5.3).

```
> summary(getRatings(MovieLense)) # Skewed to the right

    Min.      1st Qu.     Median      Mean      3rd Qu.     Max.
    1.00      3.00        4.00        3.53      4.00        5.00
```

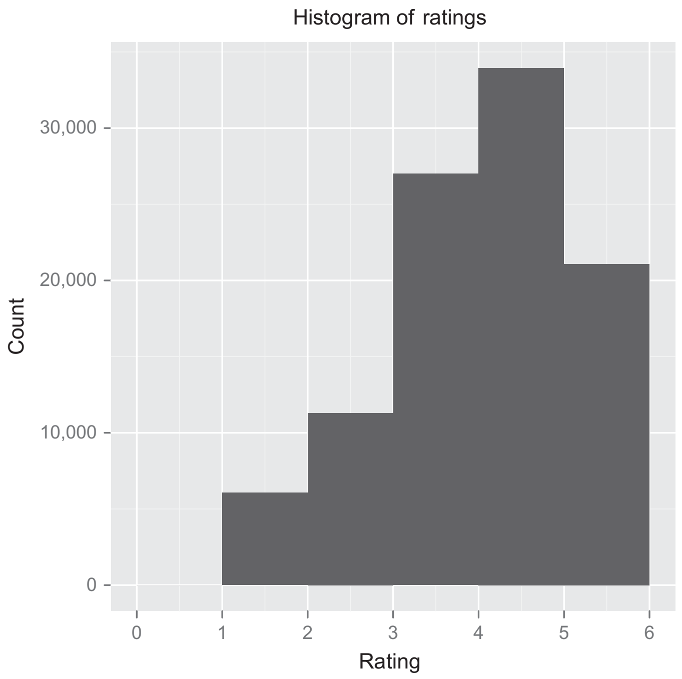

Figure 5.2
Ratings of MovieLense.

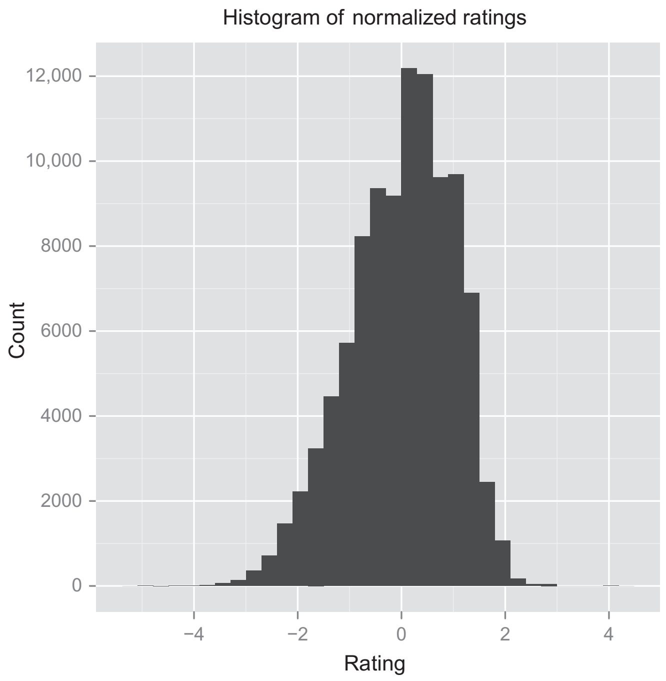

Figure 5.3
Normalized ratings of MovieLense.

```
> # Visualizing ratings
> qplot(getRatings(MovieLense), binwidth = 1,
+       main = "Histogram of ratings", xlab = "Rating")
```

Looks skewed to the right. What about after normalization?

```
> qplot(getRatings(normalize(MovieLense, method = "Z-score")),
+       main = "Histogram of normalized ratings", xlab = "Rating")
> summary(getRatings(normalize(MovieLense, method = "Z-score")))
```

```
     Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
  -4.8520   -0.6466   0.1084   0.0000   0.7506    4.1280
```

It seems better. Here is why normalization works. It adjusts for the bias of individual raters. For example, Sam might rate movies at 4 more often than Robin who usually rates them at 2. The normalization will take care of that user bias.

```
> # How many movies did people rate on average
> qplot(rowCounts(MovieLense), binwidth = 10,
+       main = "Movies Rated on average",
+       xlab = "# of users",
+       ylab = "# of movies rated")
```

Seems people get tired of rating movies at a logarithmic pace. But most rate some (Figure 5.4).

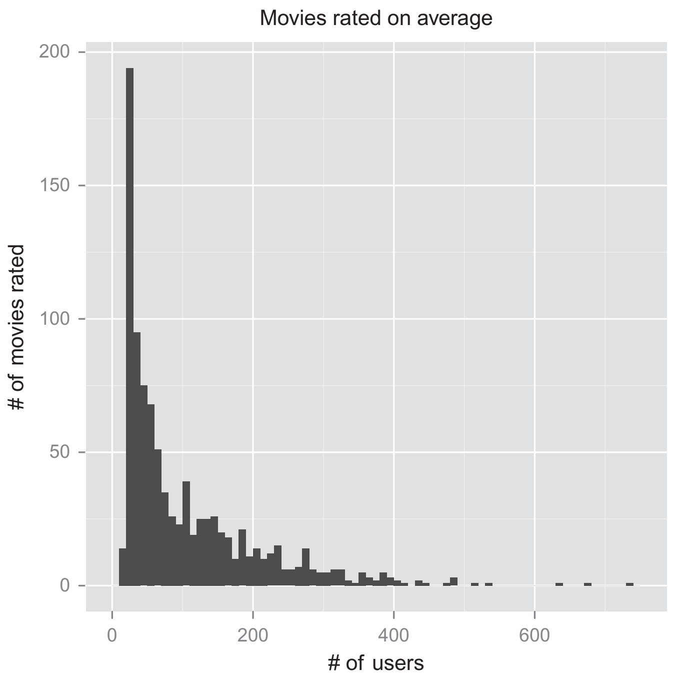

Figure 5.4
Distribution of movie raters.

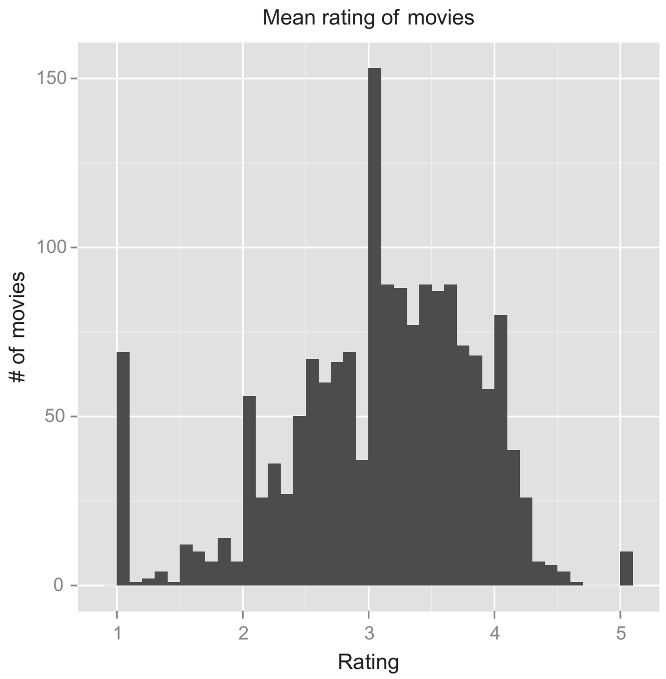

Figure 5.5
Mean rating of movies.

```
> # What is the mean rating of each movie
> qplot(colMeans(MovieLense), binwidth = .1,
+       main = "Mean rating of Movies",
+       xlab = "Rating",
+       ylab = "# of movies")
```

The big spike on 1 suggests that this could also be interpreted as binary. In other words, some people don't want to see certain movies at all. Same on 5 and on 3. We will give it the binary treatment later and see why that makes sense (Figure 5.5).

To evaluate recommender systems, we will split the data into test and training sets. We will use the training set to build our model. We will hold out some items from the test set, then make predictions using the model. Then we will compare our predictions against the holdout. If we predicted correctly, it is a true positive. If we predicted incorrectly, it is a false positive.

For production, I suggest using cross-validation. This is the same as above, except you slice it again and again. This makes sure that there was no bias in the slicing of data.

Let's see what algorithms can we test against?

```
> recommenderRegistry$get_entries(dataType = "realRatingMatrix")
```

```
$IBCF_realRatingMatrix
Recommender method: IBCF
```

```
Description: Recommender based on item-based collaborative filtering (real data).

$POPULAR_realRatingMatrix
Recommender method: POPULAR
Description: Recommender based on item popularity (real data).

$RANDOM_realRatingMatrix
Recommender method: RANDOM
Description: Produce random recommendations (real ratings).

$UBCF_realRatingMatrix
Recommender method: UBCF
Description: Recommender based on user-based collaborative filtering (real data).

> # We have a few options
>
> # Split the data into train and test. Here, train is 90%.
> # For testing it will take any 10 movie ratings by the user
> # and predict n others. Then compare to see if they match.
>
> scheme <- evaluationScheme(MovieLense, method = "split", train = .9,
+                            k = 1, given = 10, goodRating = 4)
> scheme

Evaluation scheme with 10 items given
Method: âĂŸsplitâĂŹ with 1 run(s).
Training set proportion: 0.900
Good ratings: >=4.000000
Data set: 943 × 1664 rating matrix of class âĂŸrealRatingMatrixâĂŹ with 99392 ratings.

> # Here we are using split, but other schemes are also available
> # For production testing, I STRONGLY recommend using cross-validation scheme
>
> # Let's check some algorithms against each other
> algorithms <- list(
+   "random items" = list(name="RANDOM", param=list(normalize = "Z-score")),
+   "popular items" = list(name="POPULAR", param=list(normalize = "Z-score")),
+   "user-based CF" = list(name="UBCF", param=list(normalize = "Z-score",
+                                                  method="Cosine",
+                                                  nn=50, minRating=3)),
+   "item-based CF" = list(name="IBCF", param=list(normalize = "Z-score"
+                                                  ))
+
+ )
>
```

You probably noticed normalize = "Z-score," but by now you would have expected that. The parameter that might be perplexing is method = "Cosine" for UBCF. Remember when we said that both user-based and item-based are working on similarity of neighborhoods. To define a neighborhood, we need a measure to tell them what similarity is. Cosine is most widely used. There are many others, like Jaccard, karypis, conditional. To go into them would be beyond the scope of this chapter.

nn is simply suggesting how many neighbors to consider. Make the neighborhood too small, and you might get recommendations that are all over the place. Make it too big and the recommendations might be too general. In the extreme case, too small might result in just one neighbor (one who has seen the same movie), so no useful recommendations can be generated. Too large in extreme will be the entire population, so it will result in a general popularity index, no nice user specific recommendations that we want. I am going to put that to 50 for now (Figures 5.6 and 5.7).

```
> # run algorithms, predict next n movies
> results <- evaluate(scheme, algorithms, n=c(1, 3, 5, 10, 15, 20))

RANDOM run
          1 [0.004 sec/0.506 sec]
POPULAR run
          1 [0.054 sec/0.1 sec]
UBCF run
          1 [0.048 sec/2.55 sec]
IBCF run
          1 [55.679 sec/0.508 sec]
> # Draw ROC curve
> plot(results, annotate = 1:4, legend="topleft")
> # See precision / recall
> plot(results, "prec/rec", annotate=3)
```

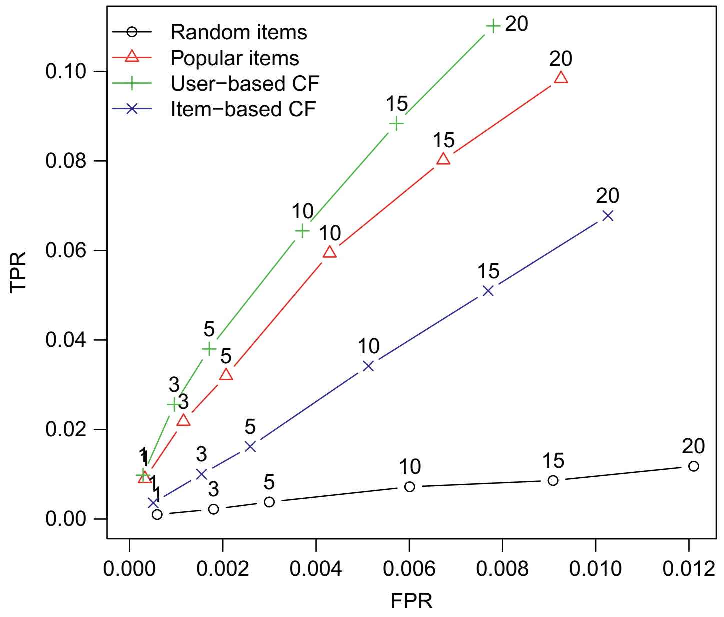

Figure 5.6
Evaluation of standard recommendation algorithms against Movie-lense.

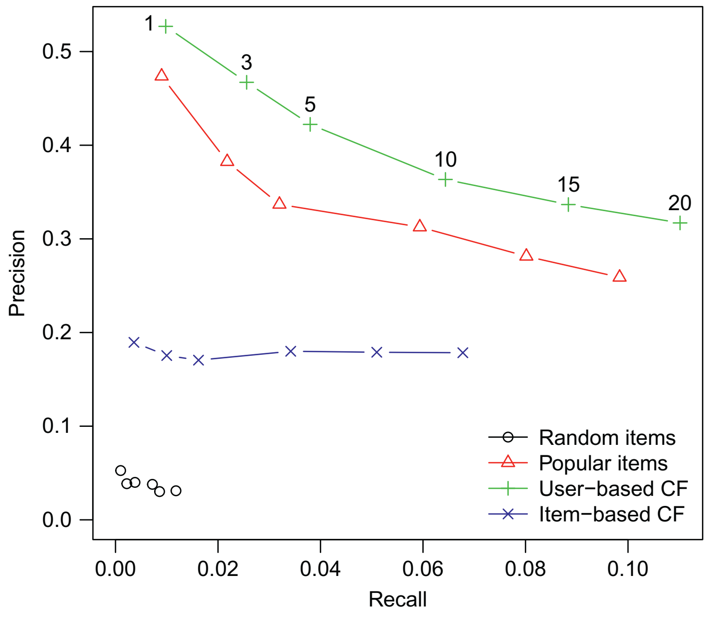

Figure 5.7
Evaluation of standard recommendation algorithms against Movie-lense.

It seems like UBCF did better than IBCF. Then why would you use IBCF? The answer lies in when and how are you generating recommendations. UBCF saves the whole matrix and then generates the recommendation at predict by finding the closest user. For large number of users-items, this becomes an issue. IBCF saves only *k* closest items in the matrix and doesn't have to save everything. It is precalculated and predict simply reads off the closest items.

Predictably, RANDOM is the worst but perhaps surprisingly it seems, it's hard to beat POPULAR. All this means that the movies rated high are usually liked by everyone and are safe recommendations. This might be different for other datasets.

Before we talk about other methods, I would like to draw your attention about what the ROC curves aren't telling us. UBCF does better but is more expensive to generate recommendations at recommend time, as it uses the whole matrix. On the other hand, IBCF finds what's good between items, but loses the nuances of different user groups. So the recommendations are not that surprising. If you want serendipity, UBCF does a better job. A user who is like you is more likely to tell you about a "cult" classic that might be lost on the general population.

We are gong to implement a few collaborative filtering algorithms. So let’s see what realRatingMatrix class really is:

```
> # Let’s start with a regular matrix of 5 users, 10 items
> set.seed(2358)
> my.mat <- matrix(sample(c(as.numeric(−2:2), NA), 50,
+                   replace=TRUE,
+                   prob=c(rep(.4/5,5), .6)), ncol=10,
+             dimnames=list(user=paste(“u”, 1:5, sep=‘’),
+                          item=paste(“i”, 1:10, sep=‘’)))
> my.mat

       item
  user  i1   i2   i3   i4   i5   i6   i7   i8   i9   i10
    u1  2    NA   NA   NA   2    NA   −1   NA   −1   NA
    u2  NA   2    0    NA   NA   NA   2    2    NA   NA
    u3  NA   NA   NA   NA   −1   NA   1    NA   −1   NA
    u4  NA   NA   1    NA   −1   NA   NA   NA   NA   NA
    u5  NA   NA   −1   2    NA   NA   0    −2   NA   NA
> # Here user u2 has rated i2 as 2 and i3 as 0.
> # Please note that 0 could be a valid value
> # All unrated values are NA
>
> # Convert this to realRatingMatrix
> (my.realM <- as(my.mat, “realRatingMatrix”))

5 × 10 rating matrix of class âĂŸrealRatingMatrixâĂŹ with 17 ratings.

> str(my.realM)

Formal class ‘realRatingMatrix’ [package “recommenderlab”] with 2 slots
  ..@ data     :Formal class ‘dgCMatrix’ [package “Matrix”] with 6 slots
  .. .. ..@ i        : int [1:17] 0 1 1 3 4 4 0 2 3 0 ...
  .. .. ..@ p        : int [1:11] 0 1 2 5 6 9 9 13 15 17 ...
  .. .. ..@ Dim      : int [1:2] 5 10
  .. .. ..@ Dimnames:List of 2
  .. .. .. ..$ user:  chr [1:5] “u1” “u2” “u3” “u4” ...
  .. .. .. ..$ item:  chr [1:10] “i1” “i2” “i3” “i4” ...
  .. .. ..@ x        : num [1:17] 2 2 0 1 −1 2 2 −1 −1 −1 ...
  .. .. ..@ factors  : list()
  ..@ normalize: NULL

> # Hmm, can we look at the underlying object?
> rating.obj <- my.realM@data
> # This is the class called sparse Matrix (notice the uppercase M)
> # By default all 0 s in Matrix are dropped to save space.
> # Since we expect mostly NAs, it has taken our input mat,
> # and converted it to 0 s. We can do this another way
>
> dropNA(my.mat)
5 × 10 sparse Matrix of class “dgCMatrix”
```

```
u1  2  .    .  .   2  .  −1   .  −1  .
u2  .  2    0  .   .  .   2   2   .  .
u3  .  .    .  .  −1  .   1   .  −1  .
u4  .  .    1  .  −1  .   .   .   .  .
u5  .  .  −1  2    .  .   0  −2   .  .
> identical (rating.obj, dropNA(my.mat))

[1] TRUE

> # OK, let's convert it back
> as.matrix(rating.obj)
        item
   User   i1   i2   i3   i4   i5   i6   i7   i8   i9   i10
     u1   2    0    0    0    2    0   −1    0   −1    0
     u2   0    2    0    0    0    0    2    2    0    0
     u3   0    0    0    0   −1    0    1    0   −1    0
     u4   0    0    1    0   −1    0    0    0    0    0
     u5   0    0   −1    2    0    0    0   −2    0    0
> # This is wrong. We had NAs!
> # What happened here is as.matrix applied to Matrix class,
> # and so it translated it zeroes instead of NAs.
> # For the right translation, we need -
> as (my.realM, "matrix")

         item
   User   i1   i2   i3   i4   i5   i6   i7   i8   i9   i10
     u1   2    NA   NA   NA   2    NA   −1   NA   −1   NA
     u2   NA   2    0    NA   NA   NA   2    2    NA   NA
     u3   NA   NA   NA   NA   −1   NA   1    NA   −1   NA
     u4   NA   NA   1    NA   −1   NA   NA   NA   NA   NA
     u5   NA   NA   −1   2    NA   NA   0    −2   NA   NA
>
```

So this means that if we are to run a standard function against realRatingMatrix, we have to be careful to what the conversion really is. Keeping this in mind, let's move on. Sparse Matrices (Bates and Maechler, 2012) do not store zeroes and thus save a lot of space. Most recommendation system datasets are pretty sparse (very few items rated from a large catalog), so this makes sense.

5.5 Latent Factor Collaborative Filtering

Now on to some recent algorithms that have worked well (à la Netflix prize). SVD (Singular value decomposition) is a way to decompose a matrix. So if we take a matrix R, we can reduce it to

$$R = U.S.V'$$

where S is the diagonal.

Why is this relevant? Imagine there are k categories of items. So you can create a kxi matrix from i items by putting i items in k categories. Similarly you can put all users in these k categories knowing how much they like each category. That will be a uxk matrix.

So now you have A_{uxk} and B_{kxi}. And if you were to multiply these together you would get S_{uxi}, your original matrix back. Pretty neat.

But how do you take this large matrix and figure out those A and B matrices? One way is SVD decomposition. Again, we can think of recommendation systems as soft-clustering. If you now take only k elements from U, S, and V, you can multiply them and get something close to the original matrix, but not the same. This can be represented as

$$R_k = U_k.S_k.V_k'$$

S is a diagonal, so taking k elements from that is trivial. This approximate matrix can be saved for recommendations. So if you have to, give recommendations for user u_l. You can simply read off the row from R_k for him.

Another nice thing about the diagonal is that it is ordered from the most variance to the least. So taking the first k elements works just fine.

This method is very efficient for large sparse matrixes and for Netflix prize, and has shown some nice results. See Sarwar et al. (2000).

Let's go ahead and implement this idea in R. We are going to use Recommenderlab's framework so that we can check our work against other results.

```
> # At the time of writing this, you still have to create function .get_parameters,
> # but you can take the code from AAA.R so it is trivial.
>
> ## helper functions and registry
> # From AAA.R
> .get_parameters <- function(p, parameter) {
+     if(!is.null(parameter) && length(parameter) != 0) {
+         o <- pmatch(names(parameter), names(p))
+
+         if(any(is.na(o)))
+         stop(sprintf(ngettext(length(is.na(o)),
+                         "Unknown option: %s",
+                         "Unknown options: %s"),
+                   paste(names(parameter)[is.na(o)],
+                         collapse = " ")))
+
+         p[o] <- parameter
+     }
+
+     p
+  }
> # Now our new method using SVD
> REAL_SVD <- function(data, parameter= NULL) {
+
+   p <- .get_parameters(list(
+     categories = 50,
+     method="Cosine",
+     normalize = "center",
+     normalize_sim_matrix = FALSE,
+     alpha = 0.5,
+     treat_na = "0",
+     minRating = NA
+     ), parameter)
+
+   # Do we need to normalize data?
+   if(!is.null(p$normalize))
+     data <- normalize(data, method=p$normalize)
+
+   # Just save everything for now.
+   model <- c(list(
+     description = "full matrix",
+     data = data
+     ), p)
+
+   predict <- function(model, newdata, n = 10,
+                       type=c("topNList", "ratings"), ...) {
+
+     type <- match.arg(type)
+     n <- as.integer(n)
+
+     # Do we need to denormalize?
+     if(!is.null(model$normalize))
+        newdata <- normalize(newdata, method=model$normalize)
+
+     # Get the old data
+     data <- model$data@data
+     # Add new data to it to create combined matrix
+     data <- rBind(data, newdata@data)
+
+     ### svd does as.matrix which sets all missing values to 0!
+     # So we have to treat missing values before we pass it to svd (fix by Michael Hahsler)
+     data <- as(data, "matrix")
+
+     if(model$treat_na=="min") data[is.na(data)] <- min(data, na.rm=TRUE)
+     else if(model$treat_na=="mean") data[is.na(data)] <- mean(data, na.rm=TRUE)
+     else if(model$treat_na=="median") data[is.na(data)] <- median(data, na.rm=TRUE)
+     else if(model$treat_na=="max") data[is.na(data)] <- max(data, na.rm=TRUE)
+     else if(model$treat_na=="0") data[is.na(data)] <- 0
+     else stop("No valid way to treat NAs specified (treat_na)!")
+
+     # Calculate SVD using available function
+     s<-svd(data)
+
+     # Get Diag but only of p elements
+     S <- diag(s$d[1:p$categories])
+
+     # Multiply it back up, but only using p elements
+     ratings <- s$u[,1:p$categories] %*% S %*% t(s$v[,1:p$categories])
+
+     # Put back correct names
+     rownames(ratings) <- rownames(data)
+     colnames(ratings) <- colnames(data)
+
+     # Only need to give back new users
+     ratings <- ratings[(dim(model$data@data)[1]+1):dim(ratings)[1],]
+
+     # Convert to right type
+     ratings <- new("realRatingMatrix", data=dropNA(ratings))
+     ## prediction done
+
+     ratings <- removeKnownRatings(ratings, newdata)
+
+     if(!is.null(model$normalize))
+        ratings <- denormalize(ratings)
+
+     if(type=="ratings") return(ratings)
+
+     getTopNLists(ratings, n=n, minRating=model$minRating)
+
+   }
+
+   ## construct recommender object
+   new("Recommender", method = "SVD", dataType = class(data),
+       ntrain = nrow(data), model = model, predict = predict)
+ }
> # Add it to registry
> recommenderRegistry$set_entry(
+   method="SVD", dataType = "realRatingMatrix", fun=REAL_SVD,
+   description="Recommender based on SVD approximation (real data).")
>
```

We had to take care of missing values for this implementation because the default SVD does as.matrix(). As shown earlier, this can be a problem with the object realRatingMatrix. So we have to convert it to a matrix properly. Now that we have added our method, let's run it and see what we get. We will continue with our previous example.

```
> algorithms <- list(
+   "random items" = list(name="RANDOM", param=list(normalize = "Z-score")),
+   "popular items" = list(name="POPULAR", param=list(normalize = "Z-score")),
+   "user-based CF" = list(name="UBCF", param=list(normalize = "Z-score",
+                                                  method="Cosine",
+                                                  nn=50, minRating=3)),
+   "item-based CF" = list(name="IBCF", param=list(normalize = "Z-score"
+                                                  )),
+   "SVD CF" = list(name="SVD", param=list(normalize = "Z-score",
+                                                  treat_na = "0"
+                                                  ))
+   )
```

```
> # run algorithms, predict next n movies
> results <- evaluate(scheme, algorithms, n=c(1, 3, 5, 10, 15, 20))

RANDOM run
         1 [0.002 sec/0.506 sec]
POPULAR run
         1 [0.051 sec/0.093 sec]
UBCF run
         1 [0.047 sec/1.703 sec]
IBCF run
         1 [55.831 sec/0.486 sec]
SVD run
         1 [0.048 sec/8.542 sec]
> # Draw ROC curve
> plot(results, annotate = 1:4, legend="topleft")
> # See precision / recall
> plot(results, "prec/rec", annotate=3)
```

Looks like it doesn't do quite as well as UBCF and the prediction time seems to be longer. Please note that there are other ways to approximate SVD as well that are more efficient. It still beats IBCF for this dataset (Figures 5.8 and 5.9).

Another take on this is PCA. If you have more items than users, you can reduce them to categories as before. But we will do it slightly differently. We can do PCA to decompose the original matrix into a matrix of its principal components.

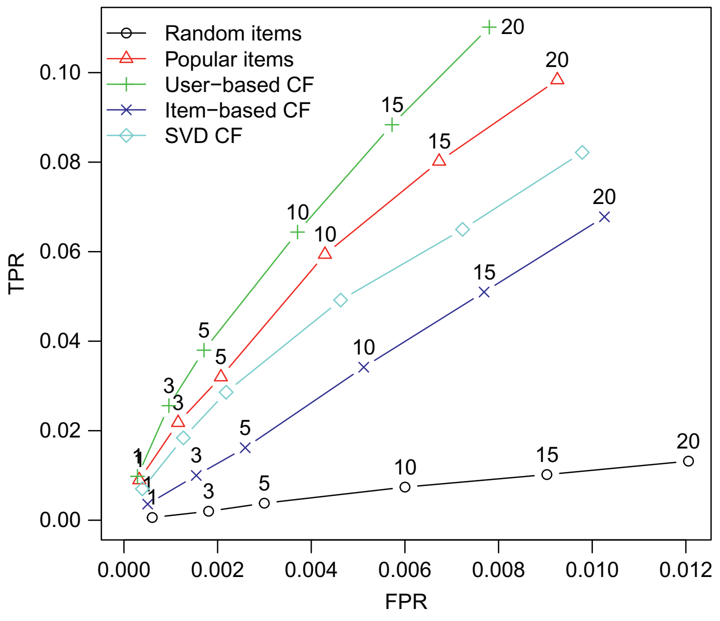

Figure 5.8
Performance of SVD CF against MovieLense.

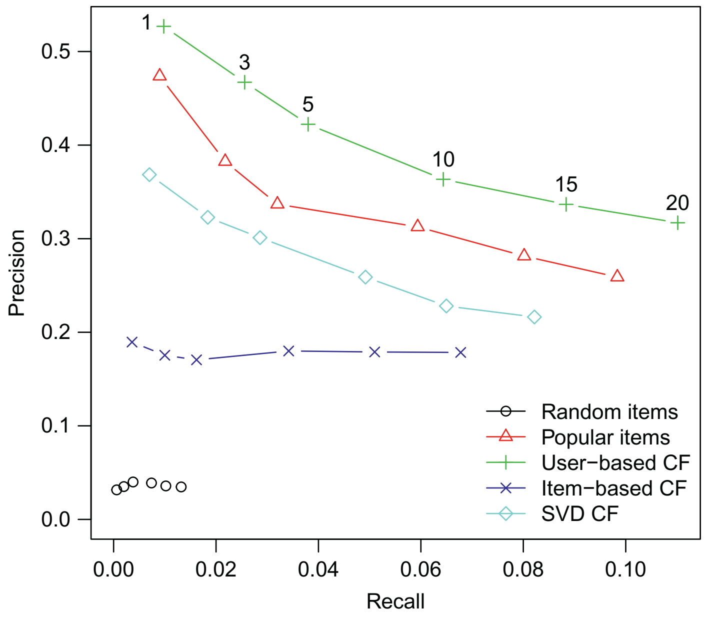

Figure 5.9
Performance of SVD CF against MovieLense.

$$R = W_1 x W_2 x W_3 \ldots x W_n$$

Multiplying these together will give us the same matrix back. You can think of each eigenvector representing a category by itself. Since the vectors are arranged from ones with most variability to the ones with least, you can take the first few vectors as a good indication of capturing most representative categories and approximate R. Again, the soft-clustering idea appears.

$$R_p = W_1 x W_2 x W_3 \ldots x W_n \,(where\, k < n)$$

For datasets with more users than items, this method is faster than regular SVD and works almost as well (Goldberg et al., 2001). Don't take my word for it, let's try it out and see how it works.

```
> REAL_PCA <- function(data, parameter = NULL) {
+
+   p <- .get_parameters(list(
+     categories = 20,
+     method="Cosine",
+     normalize = "center",
+     normalize_sim_matrix = FALSE,
+     alpha = 0.5,
+     na_as_zero = FALSE,
+     minRating = NA
+     ), parameter)
+
+
+   if(!is.null(p$normalize))
+     data <- normalize(data, method=p$normalize)
+
+   # Perform PCA
+   data <- data@data
+
+   # We will use princomp function, there are other methods available as well in R
+   # princomp does an as.matrix as well but it does not matter in this case
+   pcv<-princomp(data, cor=TRUE)
+
+   # Get the loadings
+   lpcv<-loadings(pcv)
+
+   # Total number of categories
+   cats <- min(dim(lpcv)[2], p$categories)
+
+   # det(lpcv[,1:99] %*% t(lpcv[, 1:99]))
+   # This is just a check. If this is close to 1 that means we did well.
+
+   # Convert to right type
+   itemcat <- new("realRatingMatrix",
+                              data = as(lpcv[,1:cats], "dgCMatrix"))
+
+   # Save the model
+   model <- c(list(
+      description = "PCA: Reduced item-category matrix",
+      itemcat = itemcat
+      ), p)
+
+   predict <- function(model, newdata, n = 10,
+                        type=c("topNList", "ratings"), ...) {
+
+     type <- match.arg(type)
+     n <- as.integer(n)
+
+     if(!is.null(model$normalize))
+        newdata <- normalize(newdata, method=model$normalize)
+
+     ## predict all ratings
+     u <- as(newdata, "dgCMatrix")
+     itemcat <- as(model$itemcat, "dgCMatrix")
+     ratings <- u %*% itemcat %*% t(itemcat)
+
+     ratings <- new("realRatingMatrix", data=dropNA(ratings),
+                   normalize = getNormalize(newdata))
+     ## prediction done
+
+     ratings <- removeKnownRatings(ratings, newdata)
```

```
+
+     if(!is.null(model$normalize))
+        ratings <- denormalize(ratings)
+
+     if(type=="ratings") return(ratings)
+
+     getTopNLists(ratings, n=n, minRating=model$minRating)
+
+   }
+
+   ## construct recommender object
+   new("Recommender", method = "PCA", dataType = class(data),
+       ntrain = nrow(data), model = model, predict = predict)
+ }
> # Add to registry
> recommenderRegistry$set_entry(
+   method="PCA", dataType = "realRatingMatrix", fun=REAL_PCA,
+   description="Recommender based on PCA approximation (real data).")
>
```

Now that we have implemented PCA as well, we can go ahead and compare it to what comes out of the box. We will have to use a different dataset since PCA won't work with more items than users. So we will try with Jester5k dataset. This has only 100 jokes but is evaluated by 5000 users. They are rated on a scale of −10 to 10 (Figure 5.10).

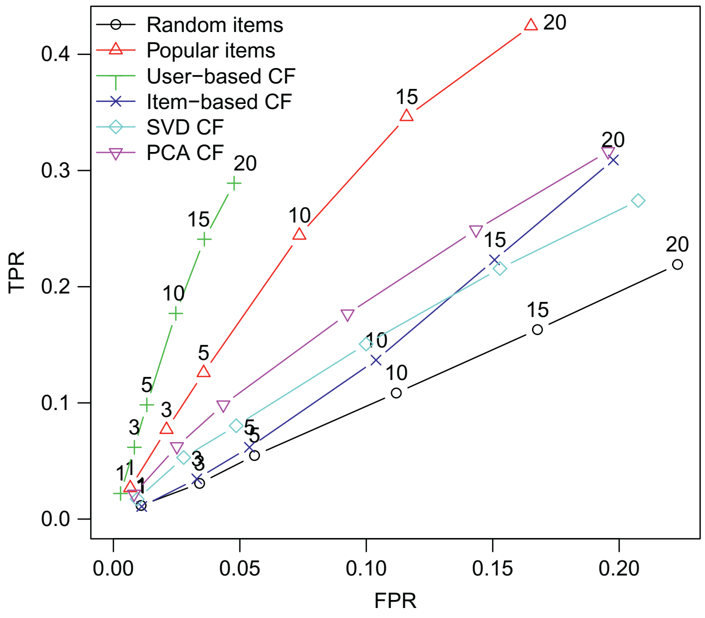

Figure 5.10
Performance of PCA CF against Jester5k.

```
> rm(MovieLense, scheme) # Clean up
> data(Jester5k) # Load another dataset
> scheme.jester <- evaluationScheme(Jester5k, method = “split”, train = .9,
+                            k = 1, given = 10, goodRating = 4)
> scheme.jester
Evaluation scheme with 10 items given
Method: âĂŸsplitâĂŹ with 1 run(s).
Training set proportion: 0.900
Good ratings: >=4.000000
Data set: 5000 × 100 rating matrix of class âĂŸrealRatingMatrixâĂŹ with 362106 ratings.

> algorithms <- list(
+   “random items” = list(name=“RANDOM”, param=list(normalize = “Z-score”)),
+   “popular items” = list(name=“POPULAR”, param=list(normalize = “Z-score”)),
+   “user-based CF” = list(name=“UBCF”, param=list(normalize = “Z-score”,
+                                                  method=“Cosine”,
+                                                  nn=50, minRating=3)),
+   “item-based CF” = list(name=“IBCF”, param=list(normalize = “Z-score”
+                                                  )),
+   “SVD CF” = list(name=“SVD”, param=list(normalize = “Z-score”,
+                                                  treat_na = “0”
+                                                  )),
+   “PCA CF” = list(name=“PCA”, param=list(normalize = “Z-score”
+                                                  ))
+   )
> # run algorithms, predict next n movies
> results <- evaluate(scheme.jester, algorithms, n=c(1, 3, 5, 10, 15, 20))


RANDOM run
         1 [0.005 sec/0.313 sec]
POPULAR run
         1 [0.222 sec/0.385 sec]
UBCF run
         1 [0.233 sec/4.909 sec]
IBCF run
         1 [0.57 sec/0.358 sec]
SVD run
         1 [0.227 sec/0.577 sec]
PCA run
         1 [0.377 sec/0.382 sec]
> # Draw ROC curve
> plot(results, annotate = 1:4, legend=“topleft”)
> # See precision / recall
> plot(results, “prec/rec”, annotate=3)
```

As you can see, this implementation of PCA does better than SVD for this dataset and is quite fast. But if I had all the time (and memory) in the world, I would still pick UBCF. It gets closest to the intuition “Ask your friends.” But as you can see from the timing, it takes a while to do so. SVD can be precomputed and results stored. That makes it very desirable. It doesn’t have to be done online if the user is known from before. PCA can be done on partial information and precomputed even earlier. It does require more items than users though (in this implementation). These two embody the intuition "What categories do you like, let me pick movies from those categories." And since the performance hit is small, these are good alternatives. In really large and sparse datasets, these might outperform UBCF. For Netflix prize, these methods were fundamental in getting a higher score (Figure 5.11).

Please note that there are many ways to decompose a matrix and many SVD implementations available in *R* (for example, irlba, model-based approximations, etc.) but I am showing here what can be done with a standard one. Feel free to switch out parts of this with better algorithms. But always test against the metric that is important to you.

Let's do one more with the same theme.

As before, imagine that for an *uxi* matrix, there are *k* categories. All items can be classified into *k* categories. For example, let's say all nontechnical books fall into four categories—romance, action, biography, misc. They can be in multiple categories with different rates. For example,

"The Godfather" = .9 action + 0 romance + .1 misc
"Lolita" = 0 action + .9 romance + .1 misc (bear with me here)
"Romeo and Juliet" = .1 action + .8 romance + .1 misc

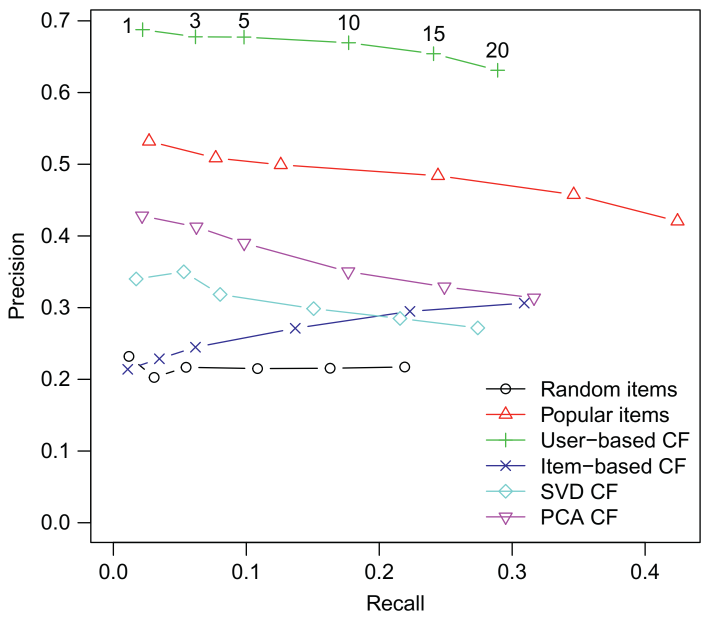

Figure 5.11
Performance of PCA CF against Jester5k.

Let’s go further and say that we have gone ahead and put all our nontechnical books in these ratings manually. So our entire library is cataloged by these three categories and how much they fall in each of them.

So now if Jim comes and asks me for a book recommendation, I know he likes .8 action + .1 romance + .1 misc. So I will recommend him “The Godfather.”

Let’s say Adele comes and asks me for a recommendation. I don’t know what she likes so I ask her for some books she has read. She tells me that she liked “Lolita.” I see that must mean that she likes .8 romance + .2 misc. The closest I have to that is “Romeo and Juliet.” So I will go ahead and recommend that. If I knew more books she liked, I could have a better idea of her taste breakdown, as I could simply average the categories of all the books she read.

So given a library of preclassified books, we can infer what the user’s taste is. Let’s say a User to category is an *uxk* matrix Θ. And Book to category is an *ixk* matrix X. To match books to users I can simply multiply these as before, $\Theta' X$.

There is one pesky detail. All books seem to have that misc category. All users will also have the same misc category. Since we don’t know what it is, we will go ahead and use the intercept of 1 there, by convention. So the matrix X truly becomes:

“The Godfather” = .9 action + 0 romance + 1
“Lolita” = .0 action + .9 romance + 1
“Romeo and Juliet” = .1 action + .8 romance + 1

To measure cost, we want to see how far our prediction ($\Theta' X$) is from reality (Y):

$$J(\Theta) = \frac{1}{2}\sum_{j=1}^{n_u}\sum\nolimits_{i:r(i,j)=1}\left(\Theta' X - Y_{i,j}\right)^2$$

We only do this for the items rated by the user as denoted by $i{:}r(i,j) = 1$. We would also have to regularize it to remove bias. Say our regularizing parameter is λ. We now have:

$$J(\Theta) = \frac{1}{2}\sum_{j=1}^{n_u}\sum\nolimits_{i:r(i,j)=1}\left(\Theta' X - Y_{i,j}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(\Theta^2\right)$$

We cannot apply the regularization parameter when $k = 1$ because we have an intercept. We will have to treat that case differently.
And taking the derivative of that with respect to Theta, our gradient is

$$\Theta_k^{(j)} = \sum\nolimits_{i:r(i,j)=1}\left(\left(\Theta^{(j)}\right)' X^{(i)} - Y^{(i,j)}\right)X_k^{(i)} + \lambda\left(\Theta_k^{(j)}\right)$$

(except for $k = 0$, our intercept, where there won’t be a regularization term).

This is called content based filtering. It's the class of algorithms that music services like Pandora typically use, thanks to an already classified music library—Music Genome Project.

But we could think of this problem the opposite way. If we knew the taste vectors of all users (*uxk* matrix X), then we could figure out what the Θ for each book is going the opposite direction. For example, if both Adele and Mike like "Lolita," I assume that the categories for "Lolita" are the average of the taste of those two users. Mathematically, the cost function now becomes

$$J\left(X^1 \ldots X^{n_m}\right) = \frac{1}{2}\sum_{i=1}^{n_m}\sum\nolimits_{j:r(i,j)=1}\left(\Theta' X^{(i)} - Y_{i,j}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(X^{(i)}\right)^2$$

So we could start with small initial guesses for both X and Θ. Then use X to estimate Θ, then Θ to estimate X, and so on. That way we will converge on appropriate values for X and Θ.

It turns out there is a way to combine both these equations. The final cost function is

$$J\left(X^1 \ldots X^{n_m}, \Theta^1 \ldots \Theta^{n_u}\right) = \frac{1}{2}\sum_{(i,j):r(i,j)=1}\left(\left(\Theta^{(j)}\right)' X^{(i)} - \left(Y^{(i,j)}\right)\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(X^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(\Theta^2\right)$$

And the gradients are:

$$\frac{d}{dX_k^{(i)}} = \sum_{i:r(i,j)=1}\left(\left(\Theta^{(j)}\right)' X^{(i)} - Y^{(i,j)}\right)\Theta_k^{(j)} + \lambda X_k^{(i)}$$

$$\frac{d}{d\Theta_k^{(j)}} = \sum\nolimits_{j:r(i,j)=1}\left(\left(\Theta^{(j)}\right)' X^{(i)} - Y^{(i,j)}\right)X_k^{(i)} + \lambda\Theta_k^{(j)}$$

We no longer have a special case for the intercept term because we are feature learning, and so it is superfluous.

Let's go ahead and implement this low rank matrix factorization using stochastic gradient descent (Funk, 2006; Koren et al., 2009; Koren, 2008).

```
> REAL_LRMF <- function(data, parameter= NULL) {
+
+   p <- .get_parameters(list(
+     categories = min(100, round(dim(data@data)[2]/2)),
+     method="Cosine",
+     normalize = "Z-score",
+     normalize_sim_matrix = FALSE,
+     minRating = NA,
+     lambda = 1.5, # regularization
+      maxit = 2000 # Number of iterations for optim
+    ), parameter)
+
+    if(!is.null(p$normalize))
+      data <- normalize(data, method=p$normalize)
+
+    model <- c(list(
+      description = "full matrix",
+      data = data
+    ), p)
+
+    predict <- function(model, newdata, n = 10,
+                        type=c("topNList", "ratings"), ...) {
+
+      type <- match.arg(type)
+      n <- as.integer(n)
+
+      if(!is.null(model$normalize))
+         newdata <- normalize(newdata, method=model$normalize)
+
+      # Get new data, make one Matrix object
+      data <- model$data@data
+      data <- rBind(data, newdata@data)
+
+      Y <- t(data)
+
+      # initialization
+      # Users
+      theta <- Matrix(runif(p$categories * dim(Y)[2]), ncol = p$categories)
+      # Items
+      X <- Matrix(runif(dim(Y)[1] * p$categories), ncol = p$categories)
+
+      # We are going to scale the data so that optim converges quickly
+      scale.fctr <- max(abs(Y@x))
+      Y@x <- Y@x / scale.fctr
+
+      # Let's optimize
+      system.time(
+        res <- optim(c(as.vector(X), as.vector(theta)),
+                     fn = J_cost_full, gr = grad,
+                     Y=Y, lambda = model$lambda,
+                     num_users = dim(theta)[1], num_books = dim(X)[1],
+                     num_cats = model$categories,
+                     method = "CG", # Slow method, faster methods available
+                     control = list(maxit=model$maxit, factr = 1e-2)
+                     )
+      )
+
+      print(paste("final cost: ", res$value, " convergence: ", res$convergence,
+                  res$message, " counts: ", res$counts))
+
+     X_final <- unroll(res$par, num_users = dim(theta)[1],
+                          num_books = dim(X)[1], num_cats = p$categories)[[1]]
+     theta_final <- unroll(res$par, num_users = dim(theta)[1],
+                               num_books = dim(X) [1], num_cats = p$categories) [[2]]
+
+     Y_final <- (X_final %*% t(theta_final) )
+     Y_final <- Y_final * scale.fctr
+     dimnames(Y_final) = dimnames(Y)
+
+     ratings <- t(Y_final)
+
+     # Only need to give back new users
+     ratings <- ratings[(dim(model$data@data)[1]+1):dim(ratings)[1],]
+
+     ratings <- new("realRatingMatrix", data=dropNA(as.matrix(ratings)))
+     ## prediction done
+
+     ratings <- removeKnownRatings(ratings, newdata)
+
+     if(!is.null(model$normalize))
+       ratings <- denormalize(ratings)
+
+     if(type=="ratings") return(ratings)
+
+     getTopNLists(ratings, n=n, minRating=model$minRating)
+
+   }
+
+   ## construct recommender object
+   new("Recommender", method = "LRMF", dataType = class(data),
+       ntrain = nrow(data), model = model, predict = predict)
+ }
> # Helper functions
> unroll <- function (Vec, num_users, num_books, num_cats) {
+   # Unroll the vector
+   endIdx <- num_books * num_cats
+   X <- Matrix(Vec[1:endIdx], nrow = num_books)
+   theta <- Matrix(Vec[(endIdx + 1): (endIdx + (num_users * num_cats))],
+                   nrow = num_users)
+
+   return (list(X, theta))
+ }
> J_cost_full <- function (Vec, Y, lambda, num_users, num_books, num_cats) {
+   # Calculate the cost
+   # Unroll the vector
+   Vec.unrolled <- unroll(Vec, num_users, num_books, num_cats)
+   X <- Vec.unrolled[[1]]
+   theta <- Vec.unrolled[[2]]
+
+   R          <- as(Y, "nsparseMatrix") * 1 # Creates binary matrix
+   Y_dash          <- (X %*% t(theta) ) * R #(Y!=0)
+   J_cost          <- .5 * (sum ((Y_dash - Y)^2)
+                           + lambda/2 * sum(X^2)
+                           + lambda/2 * sum (theta^2) )
+
+   return (J_cost)
+ }
> grad <- function (Vec, Y, lambda, num_users, num_books, num_cats) {
+   # Unroll the vector
+   Vec.unrolled <- unroll(Vec, num_users, num_books, num_cats)
+   X <- Vec.unrolled[[1]]
+   theta <- Vec.unrolled[[2]]
+
+   # Calculate gradients
+   R          <- as(Y, "nsparseMatrix") * 1 # Creates binary matrix
+   Y_dash     <- (X %*% t(theta) ) * R #(Y!=0)
+   X_gr       <- (    (Y_dash - Y) * R ) %*% theta + lambda * X
+   theta_grad <- (  t((Y_dash - Y) * R ) %*% X     + lambda * theta)
+
+   return (c(as.vector(X_gr), as.vector(theta_grad)))
+ }
> recommenderRegistry$set_entry(
+   method="LRMF", dataType = "realRatingMatrix", fun=REAL_LRMF,
+   description="Recommender based on Low Rank Matrix Factorization (real data).")
>
```

Please note that I am using optim with gradient descent because it is standard but there are many very fast optimizers available in R (e.g., L-BFGS-B). Feel free to try them.

We have to be careful though, if optim does not converge our estimates won't be very good. We can scale our Matrix to small numbers before we call optim to take care of that. Let's test it out (Figure 5.12).

```
> algorithms <- list(
+   "random items" = list(name="RANDOM", param=list(normalize = "Z-score")),
+   "popular items" = list(name="POPULAR", param=list(normalize = "Z-score")),
+   "user-based CF" = list(name="UBCF", param=list(normalize = "Z-score",
+                                                   method="Cosine",
+                                                   nn=50, minRating=3)),
+   "item-based CF" = list(name="IBCF", param=list(normalize = "Z-score"
+                                                   )),
+   "SVD CF" = list(name="SVD", param=list(normalize = "Z-score",
+                                                   treat_na = "0"
+                                                   )),
+   "PCA CF" = list(name="PCA", param=list(normalize = "Z-score"
+                                                   )),
+   "LRMF" = list(name="LRMF", param=list(normalize = "Z-score",
+                                         maxit = 5000
+                                                    ))
+   )
> # run algorithms, predict next n movies
> results <- evaluate(scheme.jester, algorithms, n=c(1, 3, 5, 10))
```

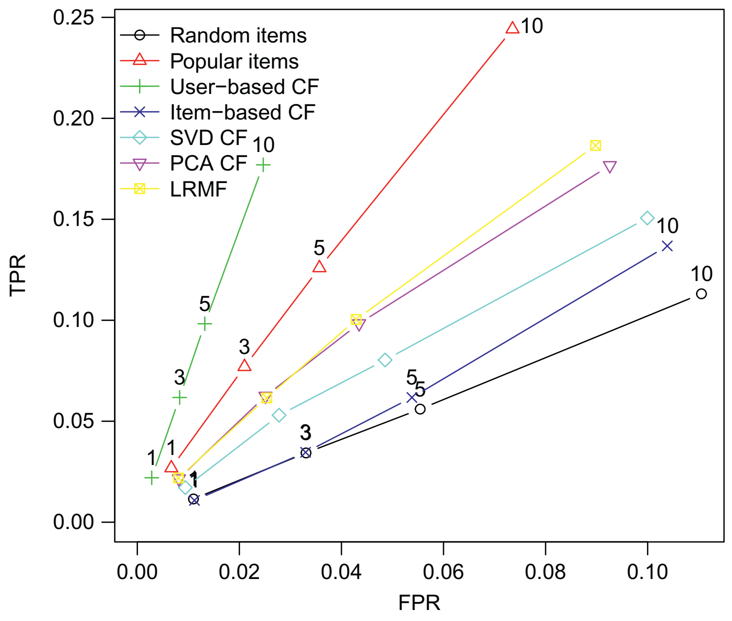

Figure 5.12
Performance of LRMF, 50 categories, 1.5 lambda against Jester5k.

```
RANDOM run
         1 [0.006 sec/0.311 sec]
POPULAR run
         1 [0.219 sec/0.383 sec]
UBCF run
         1 [0.227 sec/4.961 sec]
IBCF run
         1 [0.569 sec/0.358 sec]
SVD run
         1 [0.225 sec/0.578 sec]
PCA run
         1 [0.359 sec/0.363 sec]
LRMF run
         1 [1] "final cost: 774.063103603801 convergence: 0 counts: 967"
[2] "final cost: 774.063103603801 convergence: 0 counts: 373"
[0.227 sec/272.949 sec]
> # Draw ROC curve
> plot(results, annotate = 1:4, legend="topleft")
```

This method is especially powerful because we end up with X and Θ, which are reduced matrices of users and items in k categories. This method also takes care of worrying about NAs, as we only use the items that were rated. SVD and PCA also give roughly the same matrices, some would say the same. For example X can be thought of as the $U_k.\sqrt{S_k}$ and Θ as $\sqrt{S_k}V_k'$. You can go ahead and do user based collaborative filtering or item based with these reduced matrices. You can read more about user-based and item-based collaborative filtering in the vignette for recommenderlab. For any of these, we don't have to know what the categories are ahead of time, the model figures that out. Finally, for LRMF we can do two passes. We can create the model and figure out X in the first pass. Then use that to figure out Θ_u for a given user. This means that we can do online modeling. I will leave the implementation as an exercise for you, dear reader. Watch out for the intercept term. A similar way to incrementally update SVD is also available, see (Sarwar et al., 2002). More advanced matrix decomposition methods have also been tried out, see (Abernethy et al., 2006).

5.6 Simplified Approach

If you don't have to predict how the users will rate the item, sometimes a simplified approach will do. In this case, you don't have to rely on a user-rating matrix but rather a binary feedback. We might only know whether the user picked an item or not. We don't have to predict what the rank given to each movie was. We can test a simple version of this by binarizing our data. In addition to UBCF and IBCF, we will also try association rules that are already available in the package. Association rules find items that happen to show up together with a high level of confidence. They are a recursive algorithm and take a long time to compute all the rules. Nevertheless, we can try it for free here (Figures 5.13 and 5.14).

```
> rm(Jester5k, scheme.jester) # Clean up
> data(MovieLense) # Load data
> # Binarize
> MovieLense.bin <- binarize(MovieLense, minRating = 3)
> # What is available to us?
> recommenderRegistry$get_entries(dataType = "binaryRatingMatrix")

$AR_binaryRatingMatrix
Recommender method: AR
Description: Recommender based on association rules.

$IBCF_binaryRatingMatrix
Recommender method: IBCF
Description: Recommender based on item-based collaborative filtering (binary rating data).

$POPULAR_binaryRatingMatrix
Recommender method: POPULAR
Description: Recommender based on item popularity (binary data).

$RANDOM_binaryRatingMatrix
Recommender method: RANDOM
Description: Produce random recommendations (binary ratings).

$UBCF_binaryRatingMatrix
Recommender method: UBCF
Description: Recommender based on user-based collaborative filtering (binary data).

> # We have a few options
>
```

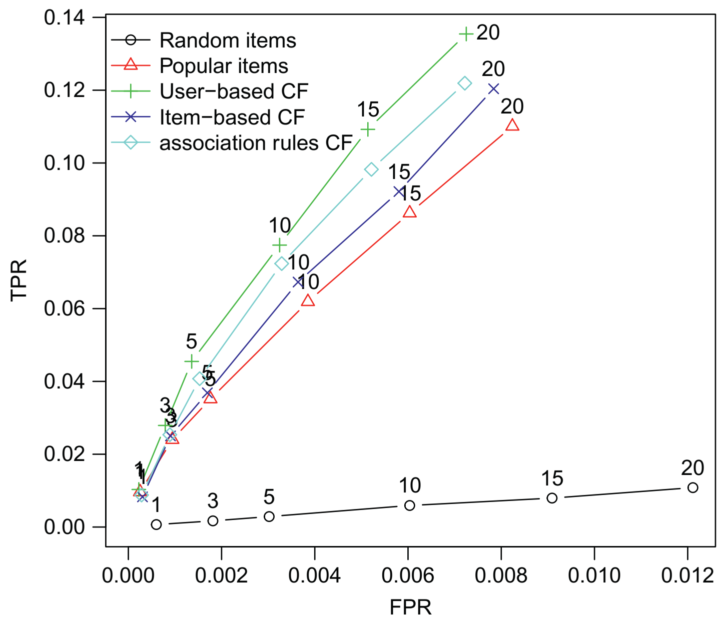

Figure 5.13
Performance of Association Rules CF against binarized Jester5k.

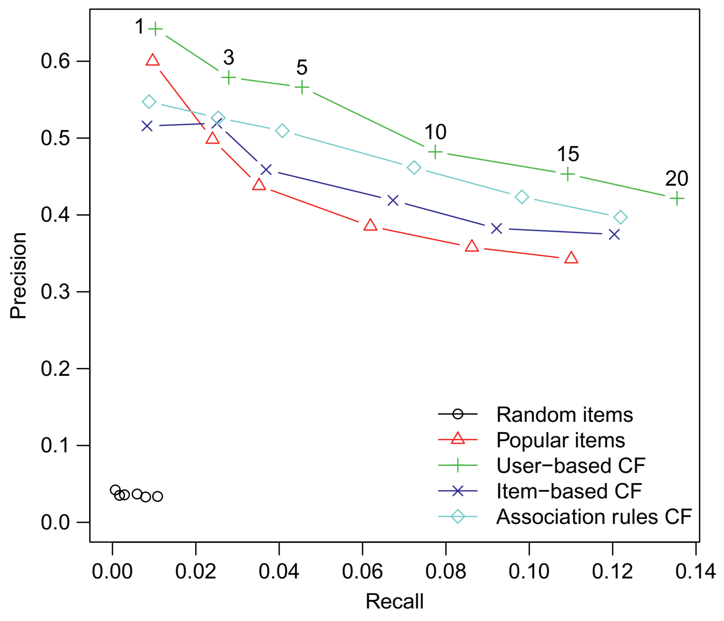

Figure 5.14
Performance of Association Rules CF against binarized Jester5k.

```
> # Let's check some algorithms against each other
> scheme.bin <- evaluationScheme(MovieLense.bin,
+                                method = "split", train = .9,
+                                k = 1, given = 6) # Had to decrease given
> scheme.bin
Evaluation scheme with 6 items given
Method: âĂŸsplitâĂŹ with 1 run(s).
Training set proportion: 0.900
Good ratings: >=NA
Data set: 943 × 1664 rating matrix of class âĂŸbinaryRatingMatrixâĂŹ with 82026 ratings.

> algorithms <- list(
+   "random items" = list(name="RANDOM", param=NULL),
+   "popular items" = list(name="POPULAR", param=NULL),
+   "user-based CF" = list(name="UBCF", param=NULL),
+   "item-based CF" = list(name="IBCF", param=NULL),
+   "association rules CF" = list(name="AR", param=NULL)
+ )
> # run algorithms, predict next n movies
> results.bin <- evaluate(scheme.bin, algorithms, n=c(1, 3, 5, 10, 15, 20))

RANDOM run
        1 [0.006 sec/0.545 sec]
POPULAR run
        1 [0.004 sec/0.584 sec]
UBCF run
        1 [0.002 sec/2.248 sec]
IBCF run
        1 [36.234 sec/0.608 sec]
AR run
        1 [0.095 sec/2.681 sec]
> # Draw ROC curve
> plot(results.bin, annotate = 1:4, legend="topleft")
> # See precision / recall
> plot(results.bin, "prec/rec", annotate=3)
```

The major lesson here is that even without rating data, we got a high AUC with just what we have. In a production system, a combination approach is often used—Ratings (if they are available), How long did the user watch the movie? Did she click on other movies as well? We can use information such as visitor's browser history, temporal data or other implicit feedback.

5.7 Roll Your Own

In the examples above, you have learned how to roll your own algorithms. You can implement even more complicated algorithms and test out your ideas against some real data. You don't have to bet that one method will work better over other, you can actually measure it. Very little work is required to convert your data into the right format (Figure 5.15).

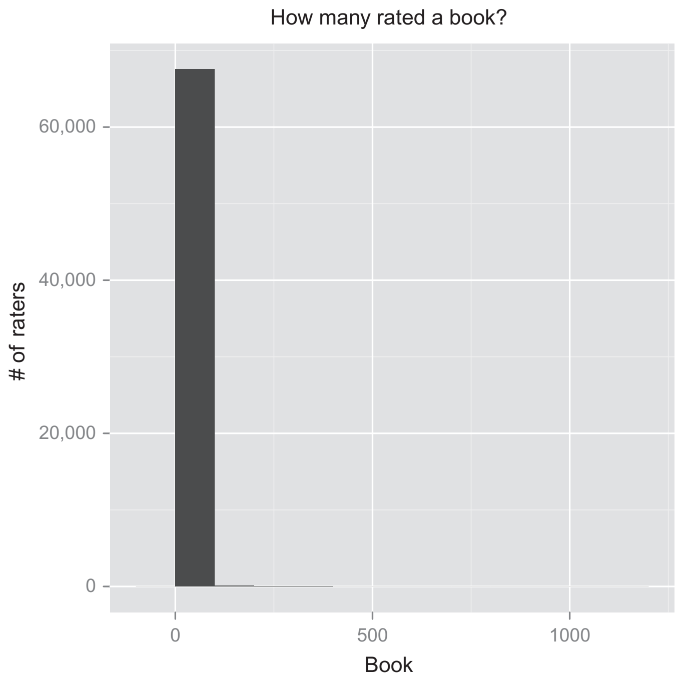

Figure 5.15
Distribution of Book Readers.

Let’s take a freely available dataset from (Ziegler et al., 2005) and run with that. This dataset has books rated by users.

```
> #######################
> # Data Aquisition
> #######################
>
> # Name of download file
> temp <- tempfile(fileext = ".zip")
> # Get the file from http://www.informatik.uni-freiburg.de/~cziegler/BX/
> download.file("http://www.informatik.uni-freiburg.de/~cziegler/BX/BX-CSV-Dump.
zip",                           temp)
> # Read in bookratings
> bookratings <- read.csv(unz(temp, "BX-Book-Ratings.csv"),
+                          header=FALSE, sep = ';',
+                          stringsAsFactors = FALSE, skip = 1,
+                          col.names = c("User.ID", "ISBN", "Book.Rating"))
> # Not used here, provided to peak your curiosity
> # users <- read.csv(unz(temp, "BX-Users.csv"),
> #                    header=FALSE, sep = ';',
> #                    stringsAsFactors = FALSE, skip = 1,
> #                    col.nam = c("User-ID", "Location", "Age"))
>
> # Are there any duplicates?
> bookratings[duplicated(bookratings)]

data frame with 0 columns and 493813 rows
> # No duplicate ratings
>
> # Read the book names
> books <- read.csv(unz(temp, "BX-Books.csv"),
+                   header=FALSE, sep = ';',
+                   stringsAsFactors = FALSE, skip = 1,
+                   col.names = c("ISBN", "Book-Title", "Book-Author",
+                                 "Year-Of-Publication", "Publisher",
+                                 "Image-URL-S", "Image-URL-M", "Image-URL-L"))
> #######################
> # Data Wrangling
> #######################
>
> # Merge the two datasets
> bookratings.dtl <- merge(bookratings, books, on = ISBN)
> bookratings.dtl$Book.detail <- with(bookratings.dtl, paste(ISBN, Book.Title, Book.
Author,
> # We only need these fields
> bookratings.dtl <- bookratings.dtl[, c("User.ID", "Book.detail", "Book.Rating")]
> # Convert it to a realRatingMatrix
> (bookratings.r <- as(bookratings.dtl, "realRatingMatrix"))

26137 × 67665 rating matrix of class âĂŸrealRatingMatrixâĂŹ with 199149 ratings.

>
> # Look at the distribution
>
> # Books were rated by how many users?
> qplot(as.vector(colCounts(bookratings.r)),
+        binwidth=100,
+        main = "How many rated a book?",
+        xlab = "Book",
+        ylab = "# of raters")
>
```

This has a very long tail. For brevity, we will go ahead and pick the top few.

```
> # Taking the 1000 most rated books (approx.)
> colIdx <- colCounts(bookratings.r)
> # Seeing the cutoff value here
> sort(colIdx, decreasing = TRUE)[1000]

0553106341::Dust to Dust::Tami Hoag
                                  24

> # using cutoff threshold
> bookratings.r@data <- bookratings.r@data[, which(colIdx >= 24)]
> bookratings.r

26137 × 1016 rating matrix of class âĂŸrealRatingMatrixâĂŹ with 58978 ratings.

> # Let's also cut down on number of users
> # We are going to evaluate on users by giving the model 5 things they have rated
> # And ask to predict the next 5
> # So we need to have atleast 10 ratings per user
> summary(rowCounts(bookratings.r))

   Min.    1st Qu.   Median   Mean    3rd Qu.   Max.
   0.000   0.000     1.000    2.256   1.000     258.000

> rowIdx <- rowCounts(bookratings.r)
> qplot(rowIdx, binwidth = 10,
+       main = "Books Rated on average",
+       xlab = "# of users",
+       ylab = "# of books rated")
>
> # If 5 are given and 5 are predicted, need to remove <10
> length(id2remove <- which(rowIdx < 10))

[1] 25074

> bookratings.r <- bookratings.r[-1*id2remove]
> # Final realRatingMatrix
> bookratings.r

1063 × 1016 rating matrix of class âĂŸrealRatingMatrixâĂŹ with 34104 ratings.

> #######################
> # Model Evaluation
> #######################
>
> scheme <- evaluationScheme(bookratings.r, method="cross-validation", goodRating=5,
+                           k=2, given=10)
> algorithms <- list(
+   "random items" = list(name="RANDOM", param=NULL),
+   "popular items" = list(name="POPULAR", param=NULL),
+   "user-based CF" = list(name="UBCF", param=list(method="Cosine",
+                                                  nn=10, minRating=1)),
+   "Item-based CF" = list(name = "IBCF", param = list(normalize="Z-score")),
+   "LRMF (100 categories)" = list(name = "LRMF", param = list(categories=100,
+                                                             normalize="Z-score"))
+ )
> results <- evaluate(scheme, algorithms, n=c(1, 3, 5, 10))

RANDOM run
        1 [0.002 sec/2.454 sec]
        2 [0.002 sec/2.61 sec]
POPULAR run
        1 [0.008 sec/0.598 sec]
        2 [0.008 sec/0.587 sec]
UBCF run
        1 [0.005 sec/6.707 sec]
        2 [0.005 sec/6.882 sec]
IBCF run
        1 [10.043 sec/2.422 sec]
        2 [9.645 sec/2.587 sec]
```

```
LRMF run
         1 [1] "final cost: 103.409093016859 convergence: 0 counts: 137"
[2] "final cost: 103.409093016859 convergence: 0 counts: 39"
[0.022 sec/48.616 sec]
         2 [1] "final cost: 119.560837692644 convergence: 0 counts: 1469"
[2] "final cost: 119.560837692644 convergence: 0 counts: 442"
[0.021 sec/479.138 sec]
>

> plot (results, annotate=c(1,3), legend="topleft")
```

Very quickly, we were able to run our algorithms on this new dataset (Figures 5.16 and 5.17).

5.8 Final Thoughts

When I introduced metrics at the beginning, I said they are roughly analogous. But that is not the full story. Are you showing your recommendations ranked or are they jumbled? How many do you show? Do you have other business rules (certain items HAVE to be shown every × times, like at a dating site)? Do you have limited inventory (recommending a physical product)? The application of the recommender system drives the evaluation method. The right

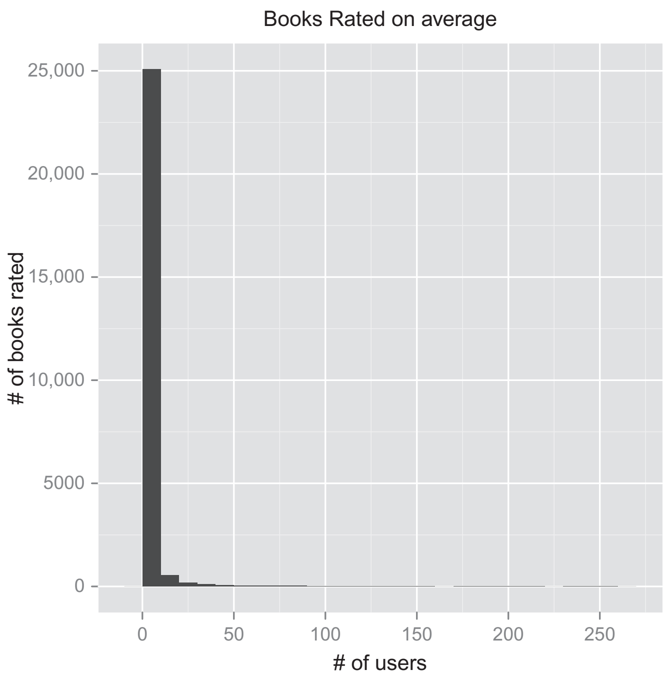

Figure 5.16
Books rated on average.

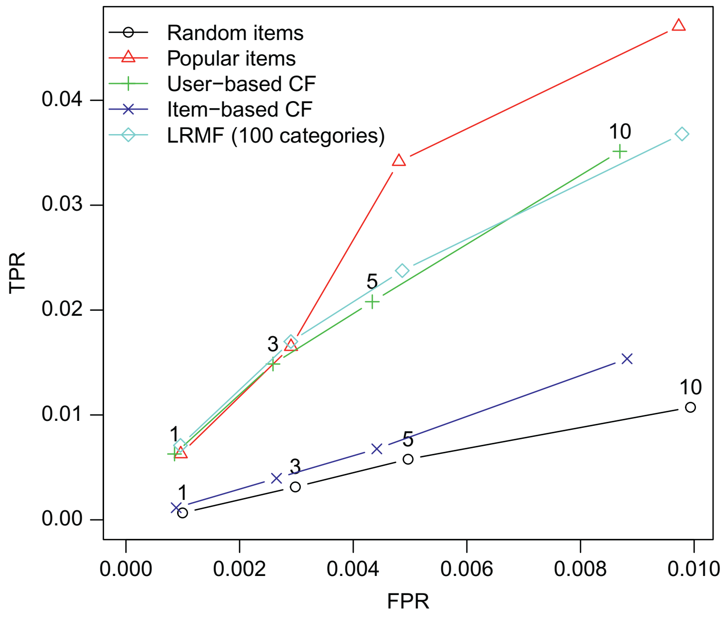

Figure 5.17
Performance of multiple CF algorithms against Book Ratings data.

metric should reflect what you want to measure. When you have that down you can use this framework to narrow down to the best answer quickly.

There are other concerns besides ROC curves. Speed of calculation could be important. UBCF gives the highest AUC for MovieLense but if speed is important, I might choose PCA over that for that dataset. If I know that I will not go over 20 items recommended to a single user, I might go with IBCF (fastest at predict time). But IBCF gives rather predictable recommendations. If serendipity is important and I want users to be surprised by the recommendations, I might do things differently. In real big data systems I might not have the luxury to get the SVD of a large sparse matrix, so I might go with model based methods or lower dimension representations. Maybe a combination is more suitable.

Then there are other issues you will hear about in the community. If we get a new user into the fold about whom we know nothing about, it would be hard to recommend the right items for him right away. This is called the cold start problem. One solution could be showing the most popular items to start with and switch to another method after you have enough history. In practice it is possible to get some data even if the user has not made any ratings, for example, geolocation, time spent, time of day, weekday, etc. This is called implicit data collection and is a fine approach.

Another dilemma is “black sheep” problem. These are users who are so unique that no other user in the system seems to follow a similar pattern. But this is an issue for real life recommenders as well, so it is OK to give reasonable recommendations to them even if they are not at par. And over time, hopefully you have enough of those kinds of users to form their own cluster/community. Then one of these methods will actually work for them. Yet another issue is how to build the user’s trust in the recommendation. One solution is to show why you are showing those recommendations, for example, for IBCF (because the users who liked item *A* and item *B* liked item *C*), or for UBCF (found users *U* and *V* just like you who like item *C*).

You now know how to test out of the box algorithms, roll your own, use your own data, and questions to ask yourself before deploying this in a real-world system. I hope this has given you a practical introduction to recommender systems and gotten you excited about testing your theories in R. Good luck and happy coding.

References

Abernethy, J., Bach, F., Evgeniou, T., Vert, J.-P. 2006. Low-rank matrix factorization with attributes. *CoRR*, abs/cs/0611124.

Bates, D., Maechler, M. 2012. Matrix: Sparse and Dense Matrix Classes and Methods. http://Matrix.R-forge.R-project.org/. R package version 1.0-6.

Funk, S., 2006. Netflix update: try this at home. http://sifter.org/simon/journal/20061211.html.

Goldberg, K., Roeder, T., Gupta, D., Perkins, C., 2001. Eigentaste: a constant time collaborative filtering algorithm. Information Retrieval. 1386-45644, 133–151. http://dx.doi.org/10.1023/A:1011419012209.

Hahsler, M., 2011. recommenderlab: lab for developing and testing recommender algorithms. http://CRAN.R-project.org/package=recommenderlab. R package version 0.1-3.

Koren, Y., 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ‘08. ACM, New York, NY, USA, pp. 426–434. http://dx.doi.org/10.1145/1401890.1401944. http://doi.acm.org/10.1145/1401890.1401944.

Koren, Y., Bell, R., Volinsky, C., 2009. Matrix factorization techniques for recommender systems. Computer. 0018-916242 (8), 30–37. http://dx.doi.org/10.1109/MC.2009.263. http://dx.doi.org/10.1109/MC.2009.263.

Sarwar, B.M., Karypis, G., Konstan, J.A., Riedl, J.T., 2000. Application of dimensionality reduction in recommender system—a case study. In: IN ACM WEBKDD WORKSHOP.

Sarwar, B., Karypis, G., Konstan, J., Riedl, J., 2002. Incremental singular value decomposition algorithms for highly scalable recommender systems. In: Fifth International Conference on Computer and Information Sciencepp. 27–28.

Wickham, H., 2009. ggplot2: Elegant Graphics for Data Analysis. Springer, New York.http://had.co.nz/ggplot2/book.

Ziegler, C.-N., McNee, S.M., Konstan, J.A., Lausen, G., 2005. Improving recommendation lists through topic diversification. In: Proceedings of the 14th international conference on World Wide Web, WWW ‘05. ACM, New York, NY, USAISBN 1-59593-046-9, pp. 22–32. http://dx.doi.org/10.1145/1060745.1060754.http://doi.acm.org/10.1145/1060745.1060754.