Recommender Systems

David Stern

May 17, 2016

In this project, I experimented with a few techniques for building a recommender system for beers. The model I followed was the *yhat* recommender built on beer reviews provided by *Beer Advocate*.

First, we load the packages we used for this recommender:

```
suppressWarnings(suppressMessages(library(recommenderlab)))
suppressWarnings(suppressMessages(library(reshape2)))
suppressWarnings(suppressMessages(library(ggplot2)))
suppressWarnings(suppressMessages(library(Matrix)))
suppressWarnings(suppressMessages(library(irlba)))
suppressWarnings(suppressMessages(library(repmis)))
```

I then loaded the beer data into R and narrowed the dataframe to the three columns: *reviewer*,*beer*, and *overall rating*. The beers are rated on a scale from 0-5 with half-point increments. The beer data contains nearly 1.6 M ratings.

After loading the data, I cast it to wide-format so that each column represented a beer and each row a reviewer. In creating the user-movie rating matrix, I aggregated by mean, so that if a user rated a beer multiple times, we would have the average of those ratings. When we look at the top left corner of the user-beer ratings matrix, we see that it is pretty sparse:

```
## Downloading data from: https://www.dropbox.com/s/csx9iwuzu7exig8/beer_reviews.csv
##
## SHA-1 hash of the downloaded data file is:
## 5ff3cfd3cb00b777f220d81daca4d3bda5cb7050
```

```
##
Read 30.9% of 1586614 rows
Read 73.7% of 1586614 rows
Read 1586614 rows and 13 (of 13) columns from 0.168 GB file in 00:00:04
```

```
beer_trunc <- beer_reviews[1:30000,c(7,11,4)]
colnames(beer_trunc) <- c("reviewer","beer","rating")
beer_sparse <- dcast(beer_trunc,reviewer~beer,value.var="rating",fill=0,fun.aggregate=mean)
rownames(beer_sparse) <- beer_sparse$reviewer
beer_sparse <- beer_sparse[,-1]
beer_sparse <- as.matrix(beer_sparse)
beer_sparse[1:5,1:5]
```

```
##          '99 Wee Heavy Scotch Ale 'Pooya Porter 'Sconnie Pale Ale
##                                 0             0                 0
## 0110x011                        0             0                 0
## 05Harley                        0             0                 0
## 12vUnion                        0             0                 0
## 1759Girl                        0             0                 0
##          'Sconnie Rustic Trail Amber 'Sconnie Tall Blonde Ale
```

```
##                                0                    0
## 0110x011                       0                    0
## 05Harley                       0                    0
## 12vUnion                       0                    0
## 1759Girl                       0                    0
```

I calculated how sparse the matrix was, and it revealed that the ratings only constitute 0.03% of the possible user-beer combinations.

```
beerInter <- beer_sparse
is.na(beerInter) <- beerInter==0
sum(is.na(beerInter))/(nrow(beerInter)*ncol(beerInter))
```

```
## [1] 0.997352
```

I first calculated the following:

1. the mean rating given by each user
2. the mean rating assigned to each beer
3. the number of ratings given by each user
4. the number of ratings given to each beer
5. user bias - the difference in a user's mean rating from the global mean rating
6. beer bias - the difference in a beer's mean rating from the global mean rating
7. the global mean rating for all movies

```
beerMeans <- apply(beerInter, 2, mean, na.rm=T)
userMeans <- apply(beerInter, 1, mean, na.rm=T)
numRatingsBeer <- apply(beerInter, 2, function(x) length(which(!is.na(x))))
numRatingsUser <- apply(beerInter, 1, function(x) length(which(!is.na(x))))
universalMean <- mean(beerInter,na.rm=T)
userBias <- userMeans - universalMean
beerBias <- beerMeans - universalMean
universalMean
```

```
## [1] 3.881724
```

In order to build a recommender system that could predict ratings, I looked at collaborative filtering techniques. The first matrix factorization technique I explored was Singular Value Decomposition (SVD), as it will provide a very good low-rank approximation to our original ratings matrix. This method also allows us flexibility in design, since we can choose to add a user's bias or a beer's bias when we recompose our ratings matrix. In this case, I experimented with several methods of adding the mean to the ratings predictions. After Using the standard for the Netflix prize, we will look at the Root Mean Squared Error (RMSE) for our predictions based on the ratings data we have.

```
timeit <- proc.time()
decomp <- svd(beer_sparse,nu=3,nv=3)
decomp$s <- diag(decomp$d[1:3])
svdPredict <- userMeans + (decomp$u %*% sqrt(decomp$s)) %*% (sqrt(decomp$s) %*% t(decomp$v))
colnames(svdPredict) <- colnames(beer_sparse)
rownames(svdPredict) <- rownames(beer_sparse)
proc.time() - timeit
```

```
##    user  system elapsed
##  90.361   2.584 105.984
```

As we see here, one of the main issues with SVD is the amount of time and processing power it requires. The RMSE is also higher than we would like it to be.

```
RMSE <- function(predictionMatrix){
  sqrt(mean((predictionMatrix-beerInter)^2,na.rm=T))
}
RMSE(svdPredict)
```

```
## [1] 1.987307
```

The *irlba* package performs a quicker, more efficient form of singular value decomposition by using a method called *implicitly restarted Lanczos bi-diagonalization*. This is less computationally expensive and uses similar syntax to the *svd* function. We indicate the number of number of left and right singular vectors to compute withthe parameters *nu* and *nv*. The function will return an estimated number of singular values equal to $max(nu, nv)$. The first thing we notice is that the *irlba* function operates far more quickly and efficiently than the *svd* function. It also returns an improved RMSE over the SVD with the same number of left and right singular vectors.

```
timeit <- proc.time()
decomp2 <- irlba(beer_sparse,nu=3,nv=3)
irlbaPredict <- userMeans + (decomp2$u * sqrt(decomp2$d)) %*% (sqrt(decomp2$d) * t(decomp2$v))
colnames(irlbaPredict) <- colnames(beer_sparse)
rownames(irlbaPredict) <- rownames(beer_sparse)
proc.time() - timeit
```

```
##    user  system elapsed
##   1.463   0.169   1.736
```

```
RMSE(irlbaPredict)
```

```
## [1] 1.80105
```

One method of using collaborating filtering to predicting ratings that avoids imputation for missing value is Alternating Least Squares (ALS). I adapted this portion from Bugra Akyildiz's example for Python.

This technique regularizes the data by taking the factor matrices X (for users) and Y (for movies) and alternately estimates each factor matrix using the other until the change in the matrices becomes insignificant. Instead of using a binary matrix W that has a "1" for each user-beer pair that has a rating and a "0" for each pair to exclude predicted ratings in the calculated of the error rate, we will use the above *RMSE* function to calculate the error. Following Akyildiz's example, I started with factor matrices X and Y, each with 100 factors, and each composed simply of uniform random variables. After 20 iterations, we get the best RMSE so far: 1.43.

```
lambda <- 0.1
n_factors <- 100
m <- nrow(beer_sparse)
n <- ncol(beer_sparse)
n_iterations <- 20
```

```
timeit <- proc.time()
X <- matrix(5*runif(m*n_factors),nrow=m,ncol=n_factors)
Y <- matrix(5*runif(n_factors*n),nrow=n_factors,ncol=n)

errors = c()
for (i in 1:n_iterations){
    X = t(solve(Y %*% t(Y) + lambda * diag(n_factors),  Y%*%t(beer_sparse)))
    Y = solve(t(X) %*% X  + lambda * diag(n_factors), t(X)%*%beer_sparse)
    errors[i] <- RMSE(X%*%Y)
}
ALS <- X%*%Y
proc.time() - timeit
```

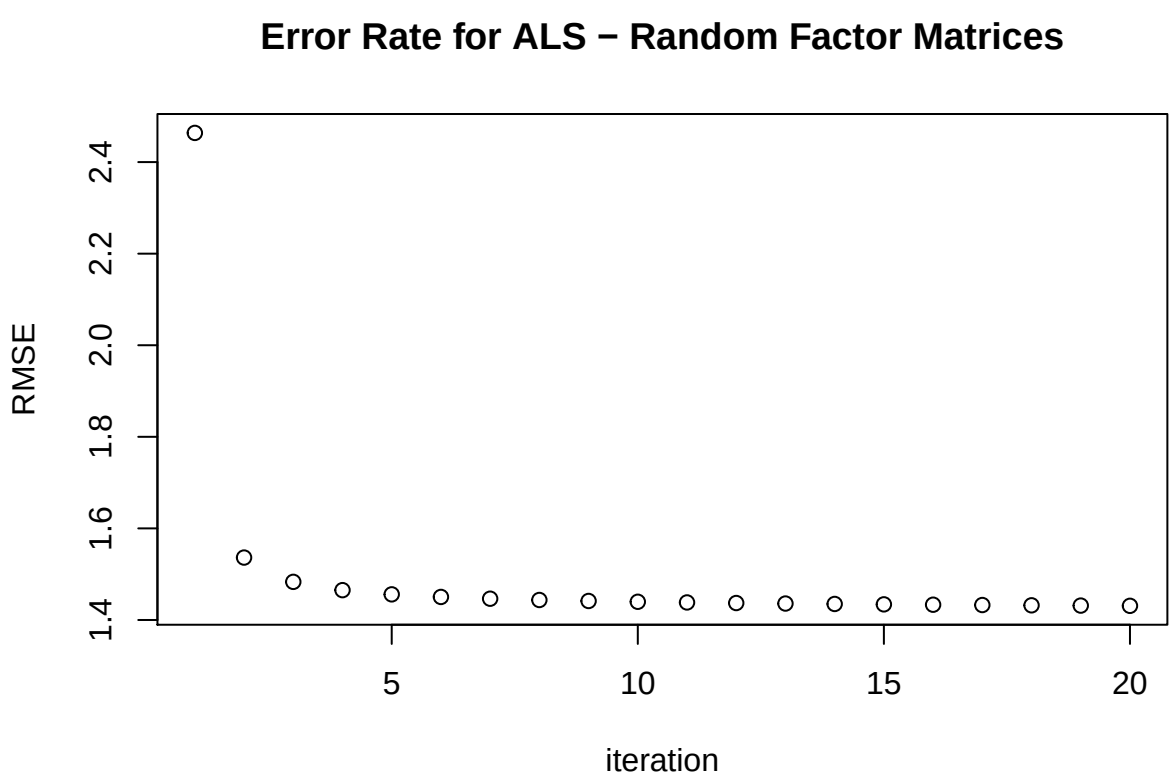
```
##    user  system elapsed
##  97.252   5.663 120.360
```

```
RMSE(ALS)
```

```
## [1] 1.43092
```

When we plot the RMSE by iteration, we see that the error rate drops sharply after the first iteration and converges on the minimum at the fifth.

```
plot(errors,xlab="iteration",ylab="RMSE",main="Error Rate for ALS - Random Factor Matrices")
```



I ran the same method using the factor matrices from an *irlba* decomposition of our beer ratings matrix with the same number of factors for X and Y.

```
timeit <- proc.time()
decomp3 <- irlba(beer_sparse,nu=100,nv=100)
X <- decomp3$u
Y <- t(decomp3$v)

errors2 = c()
for (i in 1:n_iterations){
    X = t(solve(Y %*% t(Y) + lambda * diag(n_factors),  Y%*%t(beer_sparse)))
    Y = solve(t(X) %*% X  + lambda * diag(n_factors), t(X)%*%beer_sparse)
    errors2[i] <- RMSE(X%*%Y)
}
ALS2 <- X%*%Y
proc.time() - timeit
```

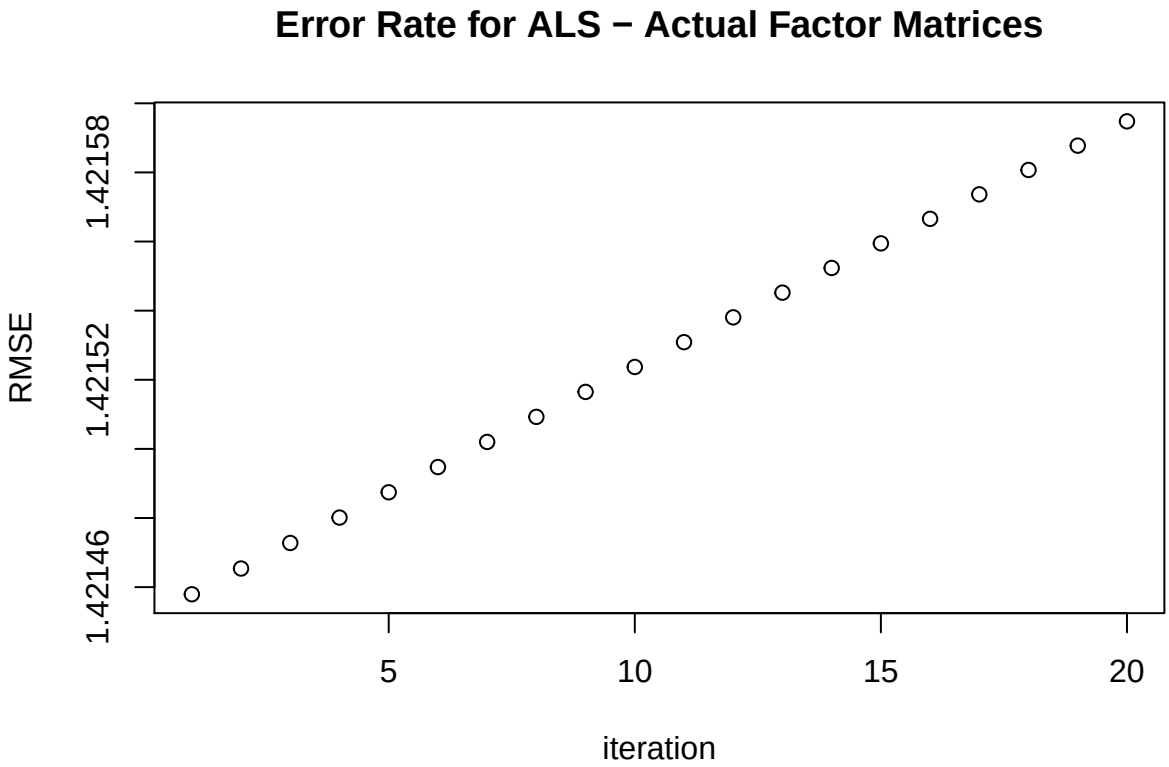
```
##    user  system elapsed
## 118.265   6.074 131.769
```

```
RMSE(ALS2)
```

```
## [1] 1.421595
```

Remarkably, this strategy performs only slightly better than using the random factors matrices, demonstrating that we can use this method without performing any kind of decomposition on our original ratings matrix. The error seems to increase very marginally for each iteration, demonstrating that the benefit to using the real factor matrices for X and Y is that you have a very low error after the first iteration. After experimenting with different values of the regularization parameter λ, I observed that the RMSE could also increase with the number of iterations, but in each case the change was very marginal and the RMSE for each approximated 1.42. Overall, this method seems to have a significant advantage for sparse matrices because it is efficient at learning the factor matrices and avoids the pitfall of overfitting data when we impute missing ratings.

```
plot(errors2,xlab="iteration",ylab="RMSE",main="Error Rate for ALS - Actual Factor Matrices")
```



While the dot product of the factor matrices X and Y are useful in explaining some of the interaction between a user and an item, it doesn't predict a full rating a drinker might give a beer. To predict the full rating for each user-beer combination, we will add the global mean rating and the biases for each beer and user. This allows us to account for the typical deviation from the mean a particular user assigns beers, or the typical deviation a beer's ratings might deviate from the average of all beers. The rating for a particular drinker-beer combination will be calculated:

$$r_{ui} = \mu + bias_{beer} + b_{user} + X \cdot Y$$

The function below takes in a user, beer, and the prediction matrix of any of the above methods.

```
getRating <- function(user,beer,method){
  if(beer_sparse[user,beer]!=0){
    paste("Already rated: ",round(beer_sparse[user,beer],1))
  }
  else{
    predicted <- method[user,beer]
    predicted <- predicted + universalMean + userBias[user] + beerBias[beer]
    paste("Predicted rating: ",round(predicted,1))
  }
}
```

We can see how the three perform:

```
getRating("12vUnion","'Sconnie Pale Ale",irlbaPredict)
```

```
## [1] "Predicted rating:  7.6"
```

```
getRating("12vUnion","'Sconnie Pale Ale",svdPredict)
```

```
## [1] "Predicted rating:  7.6"
```

```
getRating("12vUnion","'Sconnie Pale Ale",ALS)
```

```
## [1] "Predicted rating:  2.9"
```

```
getRating("12vUnion","'Sconnie Pale Ale",ALS2)
```

```
## [1] "Predicted rating:  2.9"
```

I also wanted to explore how implementing a weighted rating rather than the global mean, so I adapted IMDB's method for movies. The technique used by IMDB to regularize and rank movies is:

$$Weighted\ Rating = WR = \frac{V \cdot R + M \cdot C}{V + M}$$
$$R = average\ for\ the\ movie\ (mean)$$
$$V = number\ of\ votes\ for\ the\ movie$$
$$M = minimum\ votes\ required\ to\ be\ listed\ in\ the\ Top\ 250\ (currently\ 25,000)$$
$$C = the\ mean\ vote\ across\ the\ whole\ report$$

I decided to weigh the ratings based on the data for all users who rated more than five beers and all beers that had at least five ratings. Calling the ratings for each of the methods above shows us that the weight rating provides a significantly different prediction (-0.3) for each of the methods above. This demonstrates that how we choose to supplant collaborative filtering with corrections for bias has a significant effect on predictions that are independent of our factorization techniques.

```
M <- 5
beer_min_5 <- beer_sparse[rowSums(beer_sparse)>M,colSums(beer_sparse)>M]
R <- apply(beerInter, 2, mean, na.rm=T)
V <- apply(beerInter, 2, function(x) length(which(!is.na(x))))
C <- mean(beerInter,na.rm=T)

weightBeerRating <- function(beerName){
  (V[beerName]*R[beerName] + M*C)/(V[beerName]+M)
}

getRating2 <- function(user,beer,method){
  if(beer_sparse[user,beer]!=0){
    paste("Already rated: ",round(beer_sparse[user,beer],1))
  }
  else{
    predicted <- method[user,beer]
    predicted <- predicted + weightBeerRating(beer) + userBias[user] + beerBias[beer]
    paste("Predicted rating: ",round(predicted,1))
  }
}

getRating2("12vUnion","'Sconnie Pale Ale",irlbaPredict)
```

```
## [1] "Predicted rating:  7.3"
```

```
getRating2("12vUnion","'Sconnie Pale Ale",svdPredict)
```

```
## [1] "Predicted rating:  7.3"
```

```
getRating2("12vUnion","'Sconnie Pale Ale",ALS)
```

```
## [1] "Predicted rating:  2.6"
```

```
getRating2("12vUnion","'Sconnie Pale Ale",ALS2)
```

```
## [1] "Predicted rating:  2.6"
```