MovieLens Capstone Project

Astrid Melhado Dyer

1/2/2020

Contents

The purpose of the present assignment is to create a recommendation system using a reduced version of the MovieLens data set, and if possible to attain a RMSE <= 0.8649. The approach taken is rather opposite than the one used in the submission of the second capstone project of the course: Choose Your Own! Instead of conducting a spot-check algorithm or ensembling predictions, I rely heavily on the **recommenderlab** package to design a Collaborative Filtering Recommender Model.

Due to the pedagogic nature of the report, R codes are enclosed except when they are deemed repetitive, or do not contribute to show R proficiency. However RMD and R file maybe consulted to evaluate the entire code.

0.1 Training and Validation Set

Data Partition has been already taken care by the assignment prompt, and since this is a curated data set, there is little left to do regarding data cleaning. Thus, the real challenge of the assignment is how to deal with a large data set in a computer wise efficient manner.

The dimensions of the **edx** data set evidence the size of the call:

```
dim(edx)
```

```
[1] 9000055       6
```

Dimensions of the validation set as follows:

```
dim(validation)
```

```
[1] 999999      6
```

0.2 Data Preparation

In order to utilize the functions provided by the recommenderlab package, it is necessary to convert the data in a **realRatingMatrix"**. An S4 object that contains sparse matrices; hence, the first step is to transform the **edx** data set in a sparse matrix, and thereafter coerce them into a realRatingMatrix.

```
ratings_m <- sparseMatrix(i = as.integer(as.factor(edx$userId)),
                          j = as.integer(as.factor(edx$movieId)),
                          x = edx$rating)

dimnames(ratings_m) <- list(
  user=paste('u', 1:length(unique(edx$userId)), sep=''),
  item=paste('m', 1:length(unique(edx$movieId)), sep=''))

class(ratings_m)
```

```
[1] "dgCMatrix"
attr(,"package")
[1] "Matrix"
```

```
rRM <- as(ratings_m, "realRatingMatrix")
class(rRM)
```

```
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

0.2.1 Data Exploration

A glimpse of the matrix structure:

```
slotNames(rRM)
```

```
[1] "data"      "normalize"
```

```
dim(rRM@data)
```

```
[1] 69878 10677
```

0.2.2 Selecting Data that Matter

As it will be shown later, there is a large variation across users. Measures must be taken to avoid the potential bias caused by the lack of data generated by movies viewed only a couple of times. Likewise, distortion could

come from users who have rated a few movies only. It is wise to avoid both cases by appointing a threshold. In early stages, the following rule of thumb might be used and polished thereupon: **select movies that have been watched at least 100 times and users who have given their rating to at least 50 movies.**

```
rRM_mod <- rRM[rowCounts(rRM) > 50, colCounts(rRM) > 100]
rRM_mod
```

```
40151 x 5693 rating matrix of class 'realRatingMatrix' with 7923338 ratings.
```

Or we directly proceed to employ the quantile function.

```
min_movies1 <- quantile(rowCounts(rRM), 0.90)
print(min_movies1)
```

```
90%
301
```

```
min_users1 <- quantile(colCounts(rRM), 0.90)
print(min_users1)
```

```
   90%
2150.2
```

Either way, the idea is to keep only the relevant data.

```
rRM_mod2 <- rRM[rowCounts(rRM) > min_movies1,
                colCounts(rRM) > min_users1]
rRM_mod2
```

```
6978 x 1068 rating matrix of class 'realRatingMatrix' with 2313148 ratings.
```

0.2.3 Normalization

What about if there are users that just give high or low ratings to all their movies? This source of bias could be easily removed by normalizing the data aka making the average rating for each user 0 by means of the built-in normalize function.

```
#Normalizing the data
rRM_mod2_norm <- normalize(rRM_mod2)
sum(rowMeans(rRM_mod2_norm) > 0.00001) #Testing for Normality
```

```
[1] 0
```

Originally, the rating was an integer 1 to 5. After the normalization, we are dealing with continuous data, and the rating could be any number between -5 and 5.

Normalize is a default parameter within the functions to come; thus, there is no need to run the process except for illustration's puposes as we have done here.

0.3 Understanding Ratings

By momentarily, converting the matrix into a vector, we are able to explore the ratings.

```
v_ratings <- as.vector(rRM_mod2@data)
unique(v_ratings)
```

```
 [1] 0.0 3.0 4.0 4.5 5.0 2.0 2.5 3.5 1.0 1.5 0.5
```

```
#count the occurrences
t_ratings <- table(v_ratings)
kable(t_ratings)
```

v_ratings	Freq
0	5139356
0.5	19155
1	70164
1.5	30949
2	181345
2.5	109019
3	501465
3.5	261430
4	672960
4.5	157772
5	308889

Let us remove the ratings equal to 0 since they represent missing values according to the documentation, and convert the remaining ones into factors in order to build a frequency plot via the ggplot2 package.

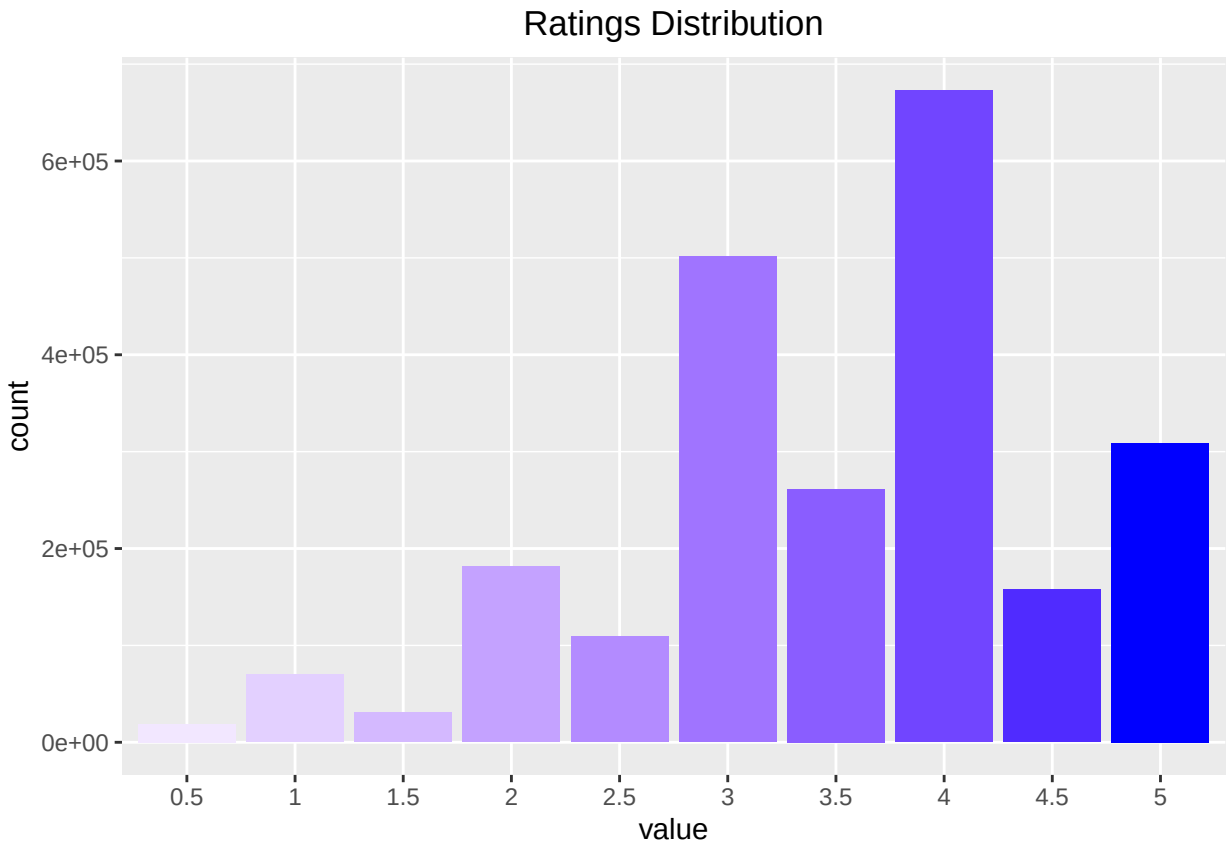
```
#0 Means no rating, turn into factors to plot
v_ratings <- v_ratings[v_ratings != 0]
v_ratings <- factor(v_ratings)
head(v_ratings)
```

```
[1] 3   4   4.5 3   4   3
Levels: 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5
```

```
str(v_ratings)
```

```
 Factor w/ 10 levels "0.5","1","1.5",..: 6 8 9 6 8 6 10 10 6 4 ...
```

```
#plot
pl_v_ratings <- ggplot(as_tibble(v_ratings), aes(x=value, fill = ..x..))+
  geom_histogram(stat = "count")+
  ggtitle("Ratings Distribution")+
  theme(plot.title = element_text(hjust = 0.5))+
  scale_fill_gradient2(low = "green", high = "blue")+
  guides(fill=FALSE)
pl_v_ratings
```



Most common rating is 4, and most of them occur above 2.

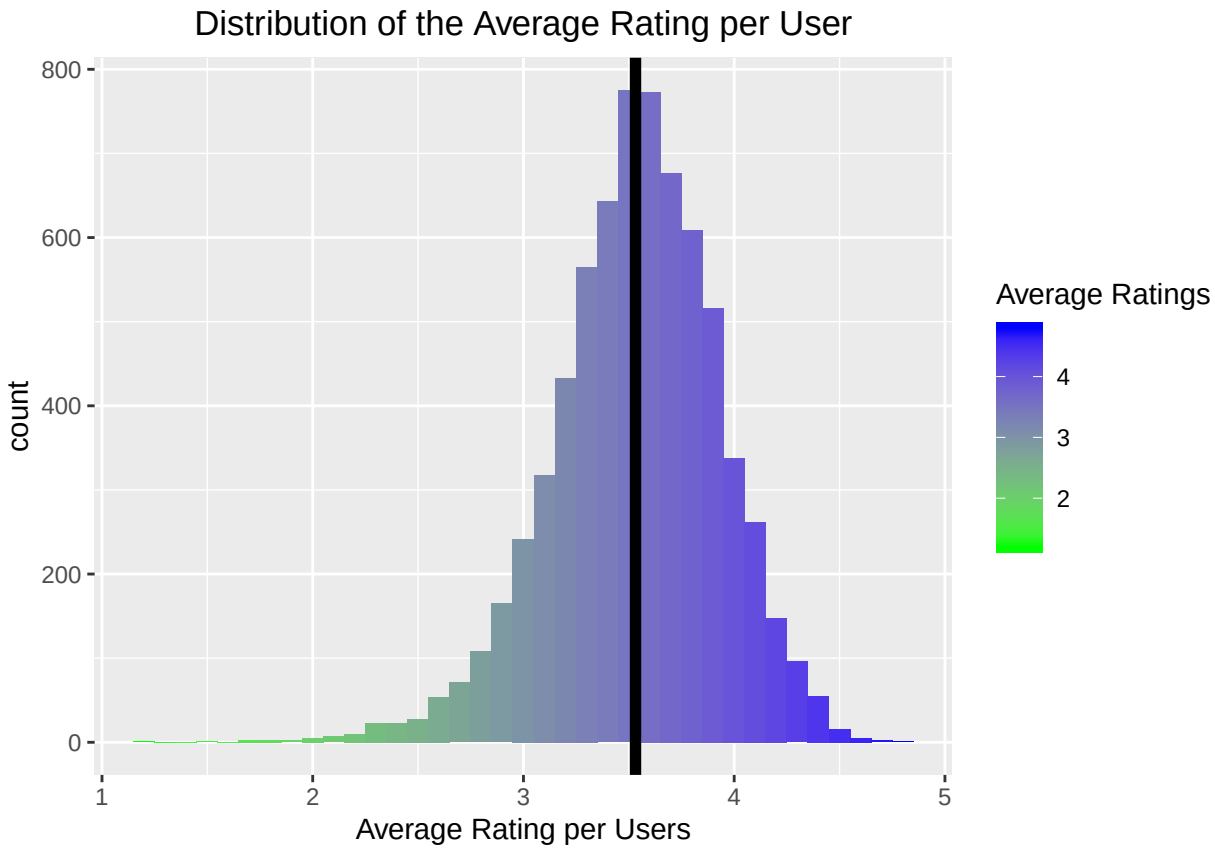
0.3.1 Ratings Distribution

A plot of the average rating distribution might provide an overall view that enables the decision making process. The highest bar gravitates around 3.5, and there are few ratings that are either 1 or 5. So, removing them as we did in the section **selecting data that matter**, it is the way to go.

```
avg_ratings_per_user <- as_tibble(rowMeans(rRM_mod2))
#head(avg_ratings_per_user)

pl_avg_ratings_per_user <- ggplot(avg_ratings_per_user, aes(x=value, fill= ..x..))+
  geom_histogram(binwidth = 0.10)+
  ggtitle("Distribution of the Average Rating per User") +
  theme(plot.title = element_text(hjust = 0.5))+
  xlab("Average Rating per Users")+
  scale_fill_gradient("Average Ratings", low = "green", high = "blue")+
  geom_vline(xintercept=mean(avg_ratings_per_user$value), color="black", size=2)

pl_avg_ratings_per_user
```



	value
	Min. :1.198
	1st Qu.:3.298
	Median :3.551
	Mean :3.532
	3rd Qu.:3.795
	Max. :4.780

0.4 The Notion of Distance

In plain words, the basic notion that lays at the foundations of **Collaborative Filtering Recommender Systems" is that if two users share the same interest on a given item in the past, they will, most likely, have similar taste in the future, Hence, the key question is how to calculate the similarity between users and items in order to generate predictions for new users and new items whose data is unknown.

Similarity measures like Euclidian distance, Cosine and Pearson correlation become a capital notion for this type of model.

See the following two matrices depicting distance between users.

0.4.1 Cosine similarity

```
similarity_cosine <- similarity(rRM_mod2[1:5, ], method = "cosine", which = "users")
#class(similarity_cosine)
kable(as.matrix(similarity_cosine))
```

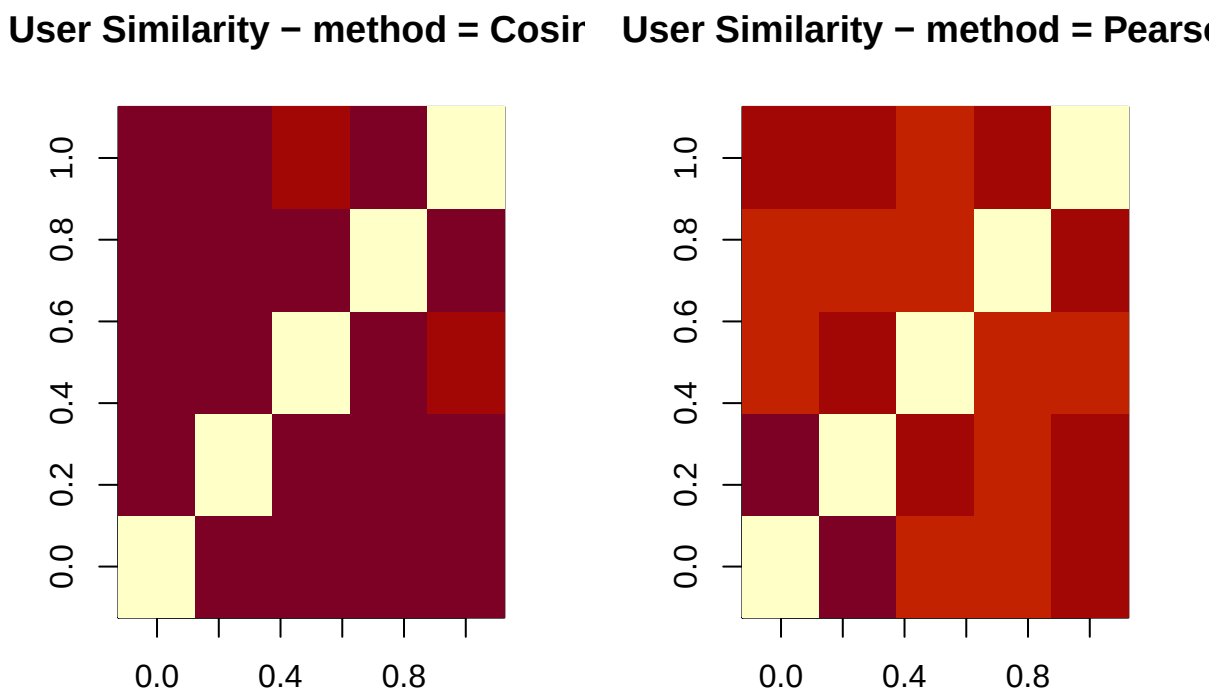
	u8	u17	u28	u30	u43
u8	0.0000000	0.9751670	0.9250941	0.9475547	0.9311529
u17	0.9751670	0.0000000	0.9460828	0.9505584	0.9364177
u28	0.9250941	0.9460828	0.0000000	0.9183983	0.8644848
u30	0.9475547	0.9505584	0.9183983	0.0000000	0.9626945
u43	0.9311529	0.9364177	0.8644848	0.9626945	0.0000000

0.4.2 Pearson Similiraty

```
similarity_pearson <- similarity(rRM_mod2_norm[1:5, ], method = "pearson", which = "users")
#class(similarity_pearson)
kable(as.matrix(similarity_pearson))
```

	u8	u17	u28	u30	u43
u8	0.0000000	0.6442279	0.5084680	0.5208026	0.5535391
u17	0.6442279	0.0000000	0.5441986	0.5186064	0.5460681
u28	0.5084680	0.5441986	0.0000000	0.5000000	0.5000000
u30	0.5208026	0.5186064	0.5000000	0.0000000	0.5789801
u43	0.5535391	0.5460681	0.5000000	0.5789801	0.0000000

```
par(mfrow = c(1, 2))
im_cosine <- image(as.matrix(similarity_cosine), main = "User Similarity - method = Cosine")
im_pearson <- image(as.matrix(similarity_pearson), main = "User Similarity - method = Pearson")
```



The deeper the color, the farther the distance. Of course, the light diagonal signals 0 distance between same user.

So far, the Pearson method seems more accurate in indicating similarities between users. Let us see, if that

hint is being confirmed by further testing.

0.5 Constructing the Model

To tackle the assignment, I will start by building a simple individual model to grasp understanding, and depart from there. I have selected an **IBCF** model or Item-Based Collaborative Filtering meaning: given two items, an index is calculated by dividing the number of users purchasing both items by the number of users purchasing at least one of them.

0.5.1 Defining Parameters

Although training and data set are given by the assignment prompt, each of them, when required, is partitioned according the guidelines of the **recommenderlab** package by means of the **evaluationScheme()** function to stay true to the selected package. In this first attempt, the split method will be used.

Furthermore, for each user in the test set, it is necessary to indicate how many items to use in otder to generate recommendations. The remaining ones will be employed to test model accuracy. This parameter should be lower than the minimum number purchased by any user; otherwise we might end up with users without items to test the model. That is the function of the piece of code: **min(rowCounts())**

Also, we need to define what constitutes good and bad items. This brain-teaser is easily solved by setting a threshold by selecting the rating's media.

Finally, I set to 1 the numbers of times to run the evaluation.

```
percentage_training <- 0.8
min(rowCounts(rRM_mod2))#37
```

```
[1] 37
```

```
items_to_keep <- 25
rating_threshold <- 3
n_eval <- 1
```

0.5.2 Splitting the Data

```
# split
eval_sets_split <- evaluationScheme(data = rRM_mod2, method = "split",
                                    train = percentage_training, given = items_to_keep,
                                    goodRating = rating_threshold, k = n_eval)
```

For good order's sake, we check the different outcomes.

```
##Checking eval_sets
getData(eval_sets_split, "train")
```

```
5582 x 1068 rating matrix of class 'realRatingMatrix' with 1848249 ratings.
```

```
#same number of users
getData(eval_sets_split, "known")#1396 x 1068 rating matrix of class 'realRatingMatrix' with 34900 rati
```

```
1396 x 1068 rating matrix of class 'realRatingMatrix' with 34900 ratings.
```

```
getData(eval_sets_split, "unknown")#1396 x 1068 rating matrix of class 'realRatingMatrix' with 427043 r
```

```
1396 x 1068 rating matrix of class 'realRatingMatrix' with 429999 ratings.
```

We make sure that around 20% of the total data is in the test sets

```
#There should be about 20 percent of data in the test set:
nrow(getData(eval_sets_split, "known")) / nrow(rRM_mod2)#0.2000573
```

```
[1] 0.2000573
```

We check that items to keeps is equal amount of items for each user.

```
#Let's see how many items we have for each user in the known set. It should be equal to items_to_keep
unique(rowCounts(getData(eval_sets_split, "known")))#25
```

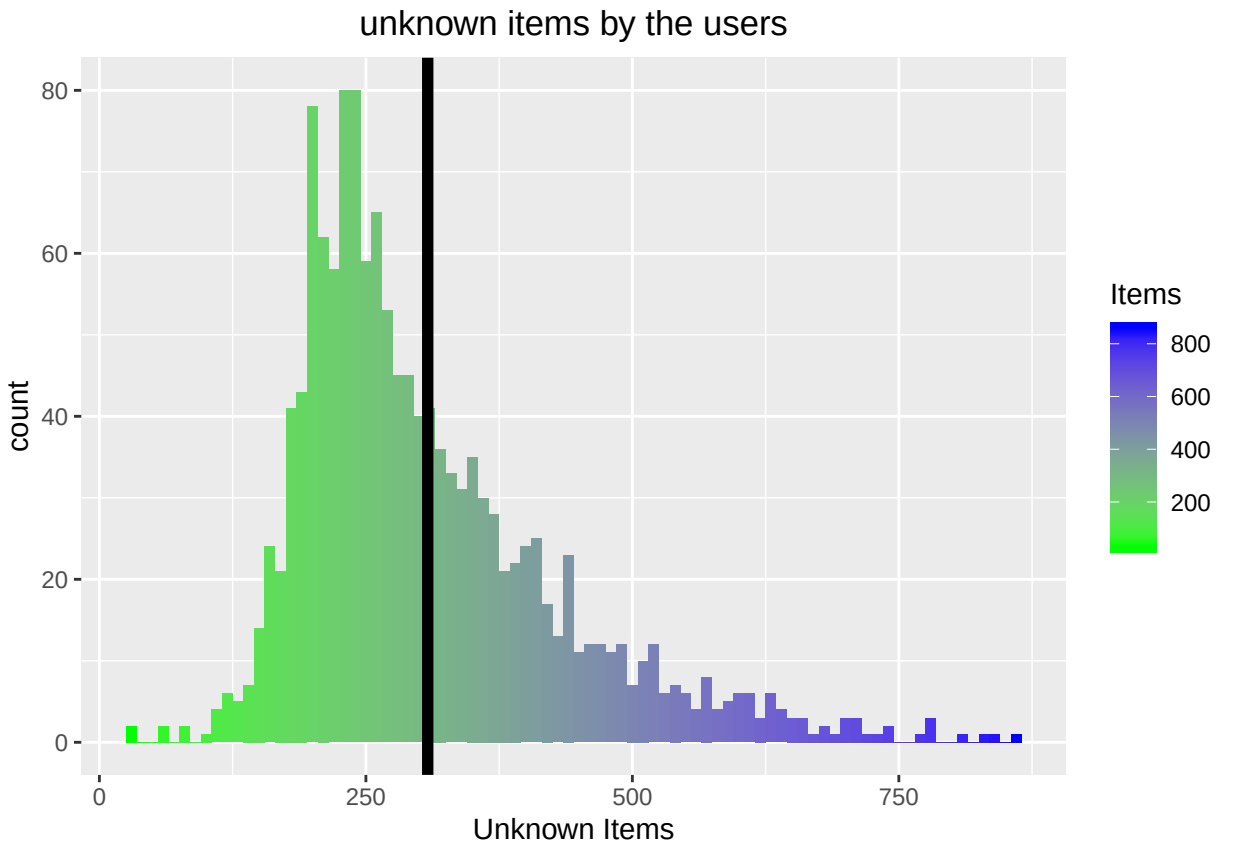
```
[1] 25
```

As expected, the number of items by users varies a lot. The distribution is right skewed. The black line indicates the mean. Normalization is a must.

```
dat_unknown <- as.tibble(rowCounts(getData(eval_sets_split, "unknown")))
head(dat_unknown)
```

```
# A tibble: 6 x 1
  value
  <int>
1   342
2   273
3   269
4   202
5   348
6   373
```

```
pl_unknown <- ggplot(dat_unknown, aes(x=value, fill=..x..))+
  geom_histogram(binwidth = 10) +
  theme(plot.title = element_text(hjust = 0.5))+
  ggtitle("unknown items by the users")+
  xlab("Unknown Items")+
  scale_fill_gradient("Items", low = "green", high = "blue")+
  geom_vline(xintercept=mean(dat_unknown$value), color="black", size=2)
pl_unknown
```



As an alternative to the **split** method, we use **k-folds** which seems to perform more accurately; although, it takes more computer time.

```
n_fold <- 4
eval_sets_kfold <- evaluationScheme(data = rRM_mod2, method = "cross-validation",
                                    k = n_fold, given = items_to_keep, goodRating = rating_threshold)
```

0.5.3 Setting the Model

```
##single model
model_parameter <- NULL
```

Please note that when parameter are set to null,default parameters of the function will be applied.

0.5.4 Creating the Recommender

```
eval_recommender <- Recommender(data = getData(eval_sets_kfold, "train"),
                                method = "IBCF", parameter = model_parameter)
items_to_recommend <- 10
```

0.5.5 Predicting

```
eval_prediction <- predict(object = eval_recommender, newdata = getData(eval_sets_kfold, "known"),
                           n = items_to_recommend, type = "ratings")
class(eval_prediction)
```

```
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

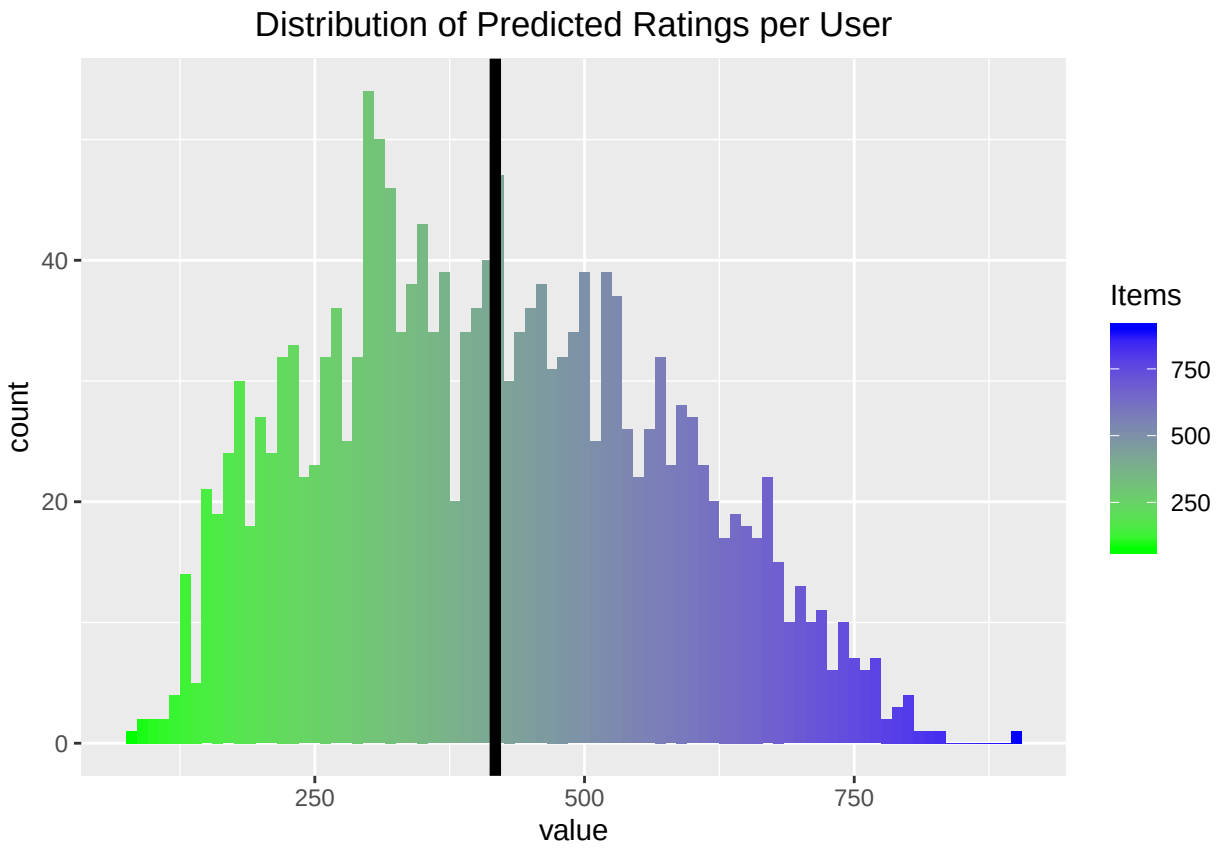
0.5.6 Predictions Distribution

```
d_prediction <- as.tibble(rowCounts(eval_prediction))
kable(head(d_prediction))
```

value
601
163
552
453
587
574

So far, so good, our model predicts 4 as the most common rating which is consistent with our findings so far.

```
pl_d_prediction <- ggplot(d_prediction, aes(x =value, fill = ..x..))+
  geom_histogram(binwidth = 10) +
  ggtitle("Distribution of Predicted Ratings per User")+
  theme(plot.title = element_text(hjust = 0.5))+
    scale_fill_gradient("Items", low = "green", high = "blue")+
  geom_vline(xintercept=mean(d_prediction$value), color="black", size=2)
pl_d_prediction
```



0.6 Accuracy

First, we evaluate the accuracy per user before evaluating the model accuracy via the RMSE as requested by the assignment prompt. To this end is necessary to have in mind the concepts summarized hereunder. Also, do note that in this context **items** equate to **movies**:

- **(TP) True Positives**: recommended items that have been purchased.
- **(FP) False Positives**: recommended items that have not been purchased.
- **(FN) False Negatives**: not recommended items that have been purchased.
- **(TN) True Negatives**: not recommended items that have not been purchased.
- **(TPR) True Positive Rate**: percentage of purchase items that have been recommended =TP/(TP+FN)
- **(FPR) False Positive Rate**: percentage of non-purchase items that have been recommended = FP/(FP+TN)

0.6.1 Accuracy per User

```
eval_accuracy <- calcPredictionAccuracy(
  x = eval_prediction, data = getData(eval_sets_kfold, "unknown"), byUser =
    TRUE)
#head(eval_accuracy)
tail(eval_accuracy)
```

```
            RMSE       MSE       MAE
u69797 1.2323707 1.5187375 0.9409140
u69832 1.2797260 1.6376986 1.0164041
u69836 1.2755689 1.6270759 0.9139312
u69862 0.9025523 0.8146007 0.7142934
u69865 1.3711328 1.8800052 1.1014652
u69873 1.0065708 1.0131848 0.5931742
```

Out of the 6 displayed observations, just 2 approximates to the RMSE target which implies testing alternative models and optimizing numeric parameters.

0.6.2 Distribution of RMSE

A well-depicted graph is always helpful for getting the bigger picture. The mean gravitates between 1.30 and 1.40 RMSE, far from desired value.

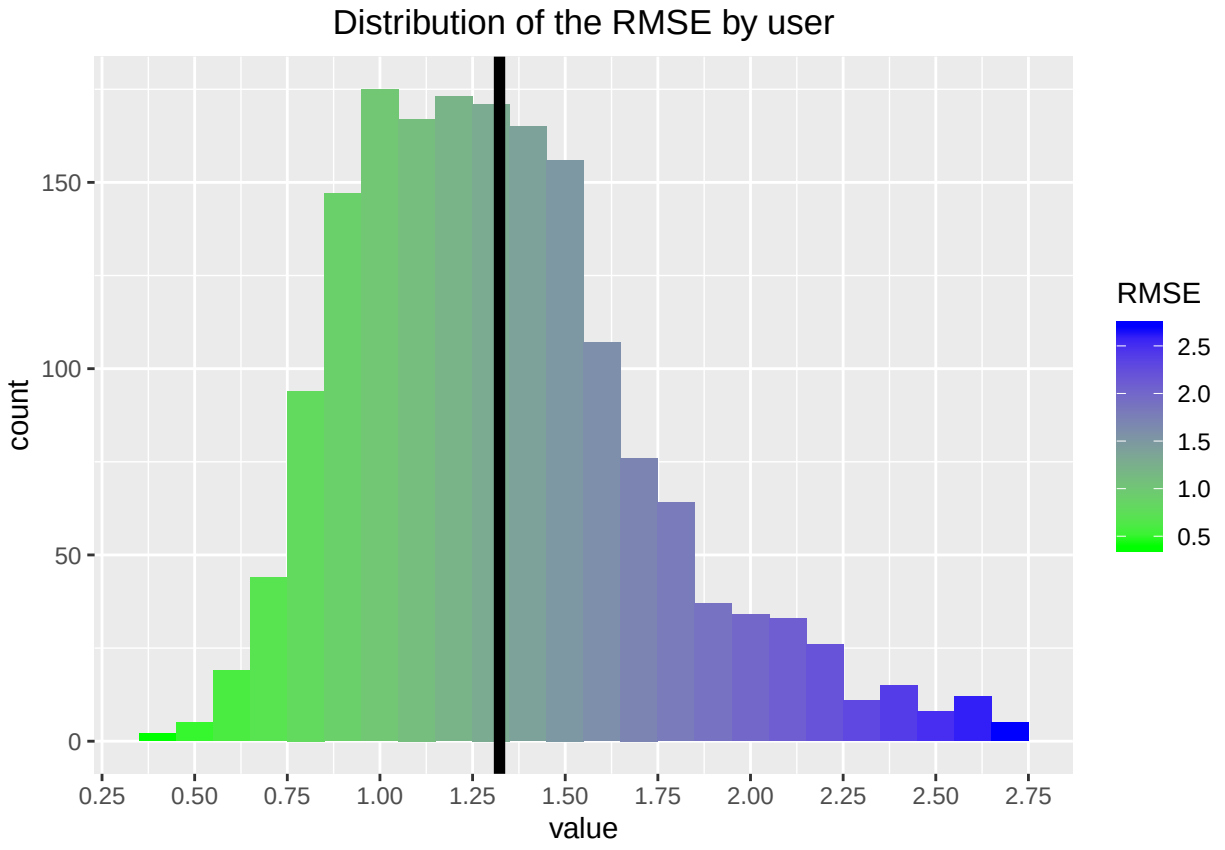
```
d_accuracy <- as.tibble(eval_accuracy[, "RMSE"])
kable(head(d_accuracy, 10))
```

value
1.2042237
0.9615225
0.8559455
1.5358653
2.6631713
1.5051945
1.2743572
1.1521117
1.5112400
1.1446369

```
pl_d_accuracy <- ggplot(d_accuracy, aes(x=value, fill = ..x..))+
  geom_histogram(binwidth = 0.1)+
  ggtitle("Distribution of the RMSE by user")+
```

```
  theme(plot.title = element_text(hjust = 0.5))+
  scale_fill_gradient("RMSE", low = "green", high = "blue")+
  geom_vline(xintercept=mean(d_accuracy$value), color="black", size=2)+
  scale_x_continuous(breaks = seq(0, 4, 0.25))

pl_d_accuracy
```



0.6.3 Accuracy per Model

Already warned by the previous model, it does not come as a surprise the RSME when evaluating the model. Sufficed to set the parameter **byuser** to false.

```
#Accuracy per model

eval_accuracy_model <- calcPredictionAccuracy(x = eval_prediction, data = getData(eval_sets_kfold, "unk
                                              byUser = FALSE)
eval_accuracy_model
```

```
    RMSE      MSE      MAE
1.461570 2.136187 1.101230
```

0.6.4 Evaluate the Recommendations

Our model performs well when dealing with true negatives, but behaves poorly when it comes to true positives.

```
#Evaluate the recommendations
results <- evaluate(x = eval_sets_kfold, method = "IBCF", n = seq(10, 100, 10))
```

```
IBCF run fold/sample [model time/prediction time]
     1  [54.3sec/1.7sec]
     2  [55.1sec/1.79sec]
     3  [53.72sec/2.71sec]
     4  [52.89sec/1.75sec]
```

```
#class(results)
kable(head(getConfusionMatrix(results)[[1]]))
```

	TP	FP	FN	TN	precision	recall	TPR	FPR
10	2.785223	7.214777	247.0218	785.9782	0.2785223	0.0114228	0.0114228	0.0090949
20	5.543528	14.456472	244.2635	778.7365	0.2771764	0.0228336	0.0228336	0.0182278
30	8.317869	21.682131	241.4891	771.5109	0.2772623	0.0342941	0.0342941	0.0273512
40	11.060710	28.939290	238.7463	764.2537	0.2765178	0.0453821	0.0453821	0.0364842
50	13.777205	36.222795	236.0298	756.9702	0.2755441	0.0563983	0.0563983	0.0456492
60	16.574456	43.425544	233.2325	749.7675	0.2762409	0.0678451	0.0678451	0.0547176

0.6.5 Aggregated Values

By summarizing the values, we get the full picture.

```
col_to_sum <- c("TP", "FP", "FN", "TN")
aggregated_values <- Reduce("+", getConfusionMatrix(results))[, col_to_sum]
kable(head(aggregated_values))
```

	TP	FP	FN	TN
10	11.24628	28.75372	996.5344	3135.466
20	22.45762	57.54238	985.3230	3106.677
30	33.64261	86.35739	974.1380	3077.862
40	44.85968	115.14032	962.9210	3049.079
50	56.00859	143.99141	951.7721	3020.228
60	67.21993	172.77606	940.5607	2991.443

0.7 Recommender Package

A quick view of some of the bounties of the *recommenderlab* package. Note that the package allow us to choose from several models.

```
Available_Models <- recommenderRegistry$get_entries(dataType = "realRatingMatrix")
kable(names(Available_Models))
```

x
ALS_realRatingMatrix
ALS_implicit_realRatingMatrix
IBCF_realRatingMatrix
LIBMF_realRatingMatrix
POPULAR_realRatingMatrix
RANDOM_realRatingMatrix
RERECOMMEND_realRatingMatrix
SVD_realRatingMatrix
SVDF_realRatingMatrix
UBCF_realRatingMatrix

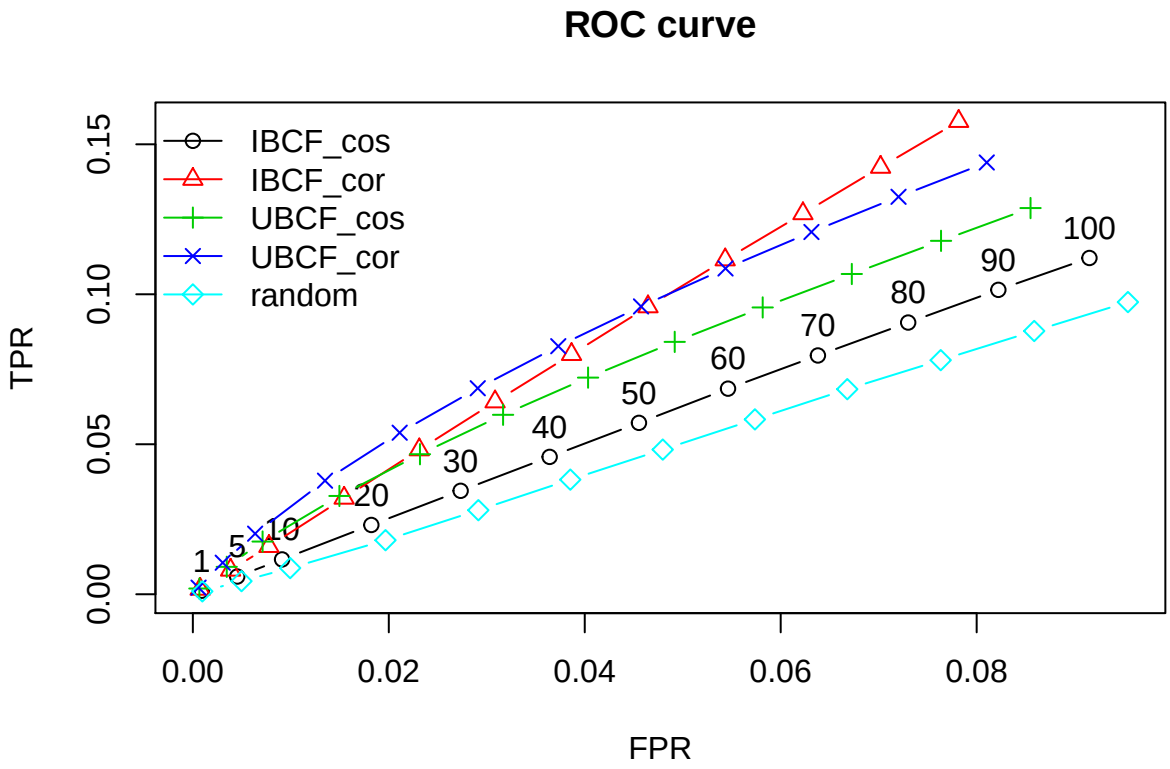
0.8 Selecting the Most Suitable Model

Due to the nature of the data we narrow the scope to User and Item Based Collaborative filtering recommender models.

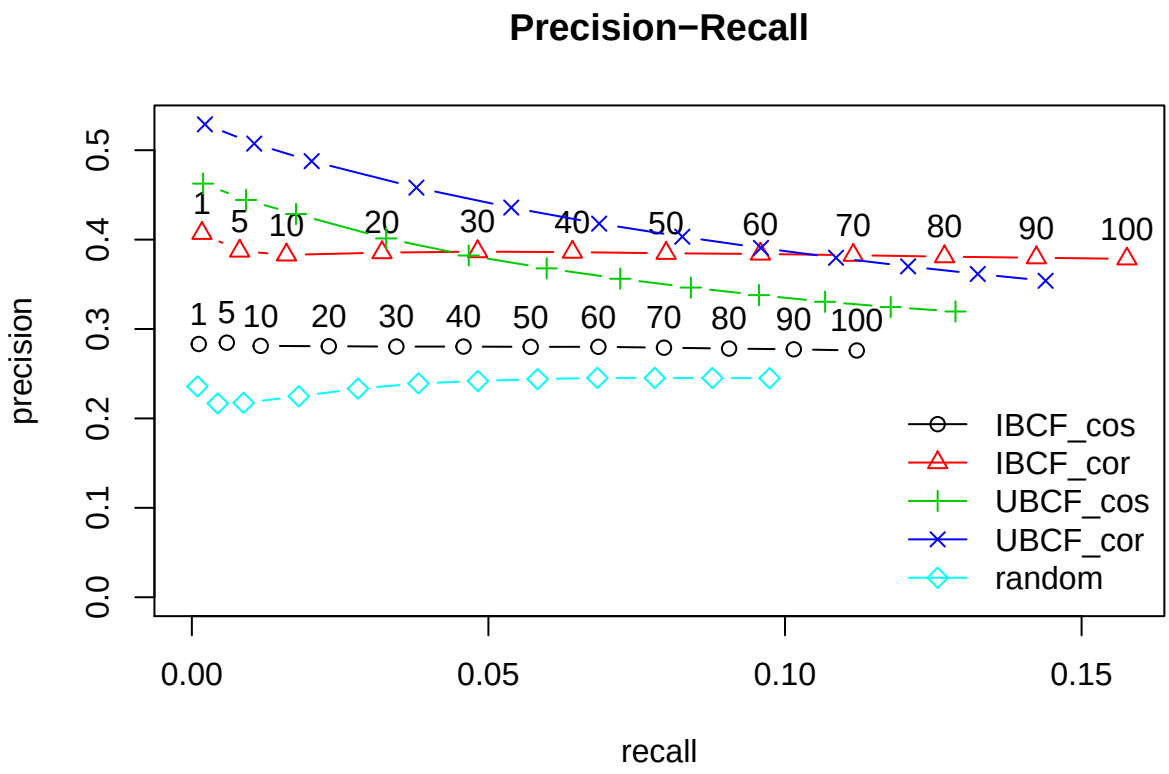
```
IBCF run fold/sample [model time/prediction time]
     1  [52.98sec/1.8sec]
     2  [54.55sec/1.8sec]
     3  [54.88sec/1.41sec]
     4  [53.07sec/1.45sec]
IBCF run fold/sample [model time/prediction time]
     1  [46.12sec/1.57sec]
     2  [45.96sec/1.57sec]
     3  [46.66sec/1.92sec]
     4  [47.63sec/1.95sec]
UBCF run fold/sample [model time/prediction time]
     1  [0.36sec/82.81sec]
     2  [0.32sec/79.97sec]
     3  [0.31sec/78.29sec]
     4  [0.34sec/80.17sec]
UBCF run fold/sample [model time/prediction time]
     1  [0.34sec/67.75sec]
     2  [0.32sec/67.53sec]
     3  [0.33sec/67.49sec]
     4  [0.32sec/67.55sec]
RANDOM run fold/sample [model time/prediction time]
     1  [0.01sec/3.27sec]
     2  [0.01sec/3.17sec]
     3  [0.01sec/3.19sec]
     4  [0.02sec/3.7sec]
```

	precision	recall	TPR	FPR
1	0.4626289	0.001905349	0.001905349	0.0006671568
5	0.4443872	0.009137389	0.009137389	0.0034592067
10	0.4286226	0.017563047	0.017563047	0.0071193809
20	0.4012958	0.032761960	0.032761960	0.0149605655
30	0.3822356	0.046679548	0.046679548	0.0231828819
40	0.3678515	0.059821718	0.059821718	0.0316653734

```
plot(list_results, annotate = 1, legend = "topleft")
title("ROC curve")
```




```
plot(list_results, "prec/rec", annotate = c(1,2), legend = "bottomright")
title("Precision-Recall")
```



Based on above plots, our first contestant is UBCF method = Pearson.

0.9 Optimizing Numeric Parameters

```
vector_k <- c(5, 10, 20, 30, 40)
models_to_evaluate_1 <- lapply(vector_k, function(k){
  list(name = "UBCF", param = list(method = "pearson", k = k))})
names(models_to_evaluate_1) <- paste0("UBCF_k_", vector_k)
n_recommendations <- 20
list_results_k <- evaluate(x = eval_sets_kfold, method = models_to_evaluate_1,
                           n = n_recommendations)
```

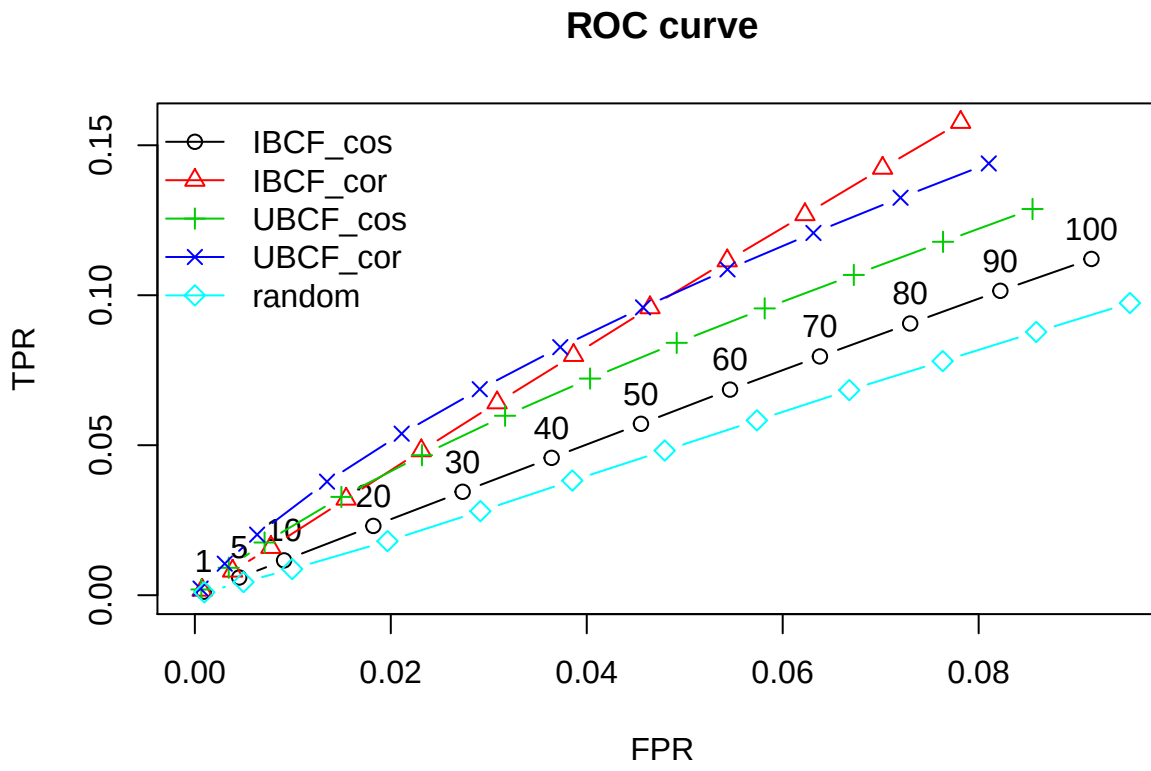
```
UBCF run fold/sample [model time/prediction time]
     1  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.34sec/66.37sec]
     2  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
```

```
[0.32sec/66.26sec]
     3  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.34sec/67.05sec]
     4  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.34sec/69.9sec]
UBCF run fold/sample [model time/prediction time]
     1  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.38sec/66.91sec]
     2  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.29sec/63.05sec]
     3  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.3sec/63.57sec]
     4  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.35sec/66.64sec]
UBCF run fold/sample [model time/prediction time]
     1  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.35sec/67.99sec]
     2  Available parameter (with default values):
method   =  cosine
```

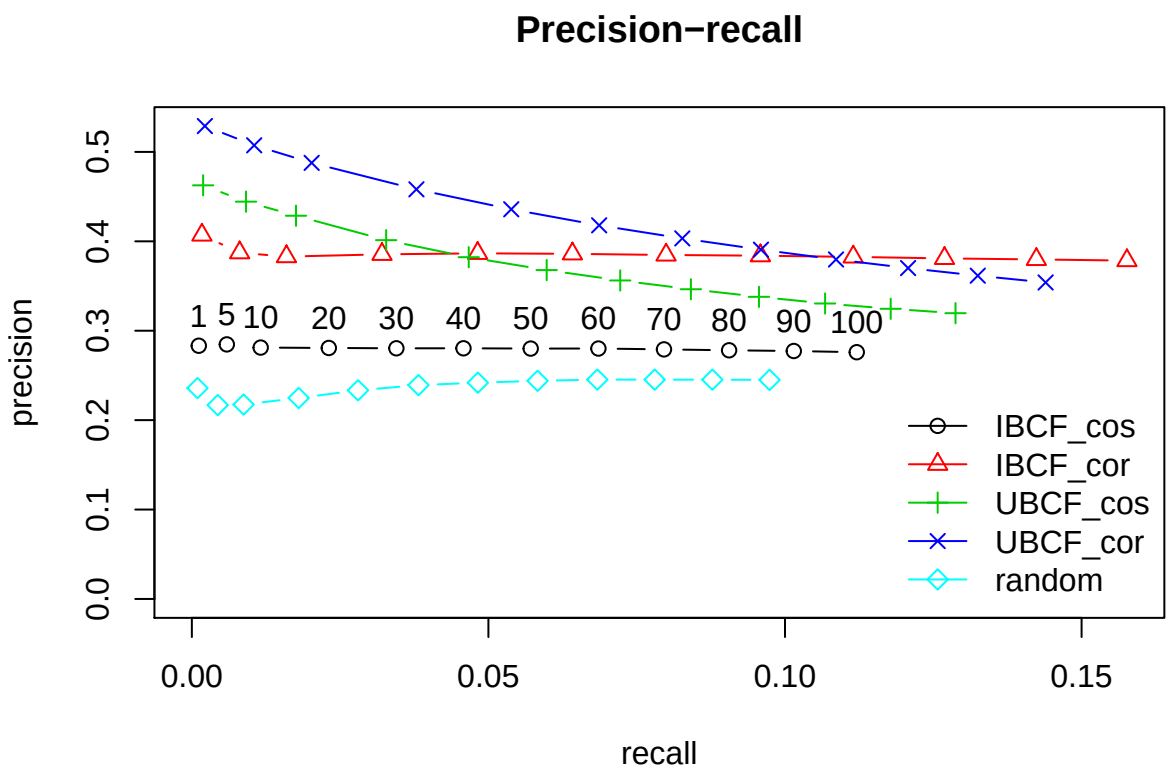
```
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.37sec/67.59sec]
     3  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.33sec/67.15sec]
     4  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.68sec/66.46sec]
UBCF run fold/sample [model time/prediction time]
     1  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.36sec/67.2sec]
     2  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.32sec/67.95sec]
     3  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.67sec/66.57sec]
     4  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.33sec/66.93sec]
UBCF run fold/sample [model time/prediction time]
     1  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
```

```
verbose  =  FALSE
[0.59sec/68.04sec]
     2  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.34sec/67.69sec]
     3  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.33sec/66.65sec]
     4  Available parameter (with default values):
method   =  cosine
nn   =  25
sample   =  FALSE
normalize    =  center
verbose  =  FALSE
[0.32sec/67.4sec]
```

```r
plot(list_results, annotate = 1,legend = "topleft")
title("ROC curve")
```



```
plot(list_results, "prec/rec", annotate =1, legend = "bottomright")
title("Precision-recall")
```



It seems tha setting n_fold to 40 is where the trade-off between precision and recall are maximized.

0.10 Testing on the Validation Set

Without further ado, we proceed to run the model on the validation set.

```
ratings_m_val <- sparseMatrix(i = as.integer(as.factor(validation$userId)),
                          j = as.integer(as.factor(validation$movieId)),
                          x = validation$rating)

dimnames(ratings_m_val) <- list(
  user=paste('u', 1:length(unique(validation$userId)), sep=''),
  item=paste('m', 1:length(unique(validation$movieId)), sep=''))

class(ratings_m_val)
```

```
[1] "dgCMatrix"
attr(,"package")
[1] "Matrix"
```

```
rRM_val <- as(ratings_m_val, "realRatingMatrix")
class(rRM_val)
```

```
[1] "realRatingMatrix"
attr(,"package")
```

```
[1] "recommenderlab"
```

```
dim(rRM_val)
```

```
[1] 68534  9809
```

```
rRM_mod2_val <- rRM[rowCounts(rRM_val) > 50, colCounts(rRM) > 100]
rRM_mod2
```

```
6978 x 1068 rating matrix of class 'realRatingMatrix' with 2313148 ratings.
```

```
min(rowCounts(rRM_mod2_val))
```

```
[1] 11
```

```
n_fold <- 40
items_to_keep <- 10
rating_threshold <- 3
n_eval <- 1

set.seed(13)
eval_sets_kfold_val <- evaluationScheme(data = rRM_mod2_val, method = "cross-validation",
                                        k = n_fold, given = items_to_keep, goodRating = rating_threshold)
```

```
eval_recommender_val <- Recommender(data = getData(eval_sets_kfold_val, "train"),
                                    method = "UBCF", parameter = "pearson")
```

```
model_detail <- getModel(eval_recommender_val)
model_detail$description
```

```
[1] "UBCF-Real data: contains full or sample of data set"
```

```
set.seed(13)
eval_prediction_val <- predict(object = eval_recommender_val, newdata = getData(eval_sets_kfold_val, "kn
                               n = items_to_recommend, type = "ratings")
class(eval_prediction_val)
```

```
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

```
set.seed(13)
eval_accuracy_val <- calcPredictionAccuracy(
  x = eval_prediction_val, data = getData(eval_sets_kfold_val, "unknown"))
#head(eval_accuracy)
eval_accuracy_val
```

```
     RMSE       MSE       MAE
0.9627589 0.9269046 0.7508625
```

Although maximizing the numerical parameters mainly k-fold, and switching from item to user based collaborative filtering model have greatly improve our RSME from 1.46 to 0.96, it does not yet hit the mark.

0.11 Why do I not achieve a more desirable RMSE?

Is it the model or the data reduction techniques employed where I have gone amiss?

In order to ascertain the cause of my demise, I run the model on the *MovieLense* data set encountered in the *recommenderlab* package. The data is already a realRating Matrix which implies that dimensionality reduction techniques have already wisely applied to the original data set.

```r
data("MovieLense")
dim(MovieLense)
```

```
[1]  943 1664
```

```r
class(MovieLense)
```

```
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

```r
slotNames(MovieLense)
```

```
[1] "data"      "normalize"
```

```r
min_movies_ML <- quantile(rowCounts(MovieLense), 0.90)
print(min_movies_ML)
```

```
90%
242
```

```r
min_users_ML <- quantile(colCounts(MovieLense), 0.90)
print(min_users_ML)
```

```
90%
169
```

```r
RM <- MovieLense[rowCounts(MovieLense) > min_movies_ML,
                 colCounts(MovieLense) > min_users_ML]
RM
```

```
94 x 166 rating matrix of class 'realRatingMatrix' with 10418 ratings.
```

```r
min(rowCounts(RM))
```

```
[1] 56
```

```r
n_fold <- 40
items_to_keep <- 50
rating_threshold <- 3
n_eval <- 1

set.seed(13)
eval_sets_RM <- evaluationScheme(data = RM, method = "cross-validation",
                                 k = n_fold, given = items_to_keep, goodRating = rating_threshold)
```

```r
set.seed(13)
eval_recommender_RM <- Recommender(data = getData(eval_sets_RM, "train"),
                                   method = "UBCF", parameter = "pearson")
```

```r
set.seed(13)
eval_prediction_RM <- predict(object = eval_recommender_RM, newdata = getData(eval_sets_RM, "known"),
                              n = items_to_recommend, type = "ratings")
class(eval_prediction_RM)
```

```
[1] "realRatingMatrix"
attr(,"package")
[1] "recommenderlab"
```

```r
set.seed(13)
eval_accuracy_RM <- calcPredictionAccuracy(x = eval_prediction_RM,
                                           data = getData(eval_sets_RM, "unknown"),
                                           byUser = FALSE)
eval_accuracy_RM
```

```
     RMSE       MSE       MAE
0.8647487 0.7477904 0.6745923
```

0.12 Conclusions

Attained RSME of 0.8647487 is within the indicated target, but the results were not obtained over the validation set as dimensionality reduction techniques were, most likely, not properly applied. Designed model proved to come across, but failure to effectively reduce the data was costly. Although, I consider that I successfully created a recommender, the picture is far from complete without mastering tools as **PCA**, **SVD**, and others. Learning such techniques is something to look forward in the nearby future.

0.13 Works Cited

Douglas Bates and Martin Maechler (2019). Matrix: Sparse and Dense Matrix Classes and Methods. R package version 1.2-18. https://CRAN.R-project.org/package=Matrix

Michael Hahsler (2019). recommenderlab: Lab for Developing and Testing Recommender Algorithms. R package version 0.2-5. https://CRAN.R-project.org/package=recommenderlab

Matt Dowle and Arun Srinivasan (2019). data.table: Extension of `data.frame`. R package version 1.12.2. https://CRAN.R-project.org/package=data.table

Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer, Brenton Kenkel, the R Core Team, Michael Benesty, Reynald Lescarbeau, Andrew Ziem, Luca Scrucca, Yuan Tang, Can Candan and Tyler Hunt. (2019). caret: Classification and Regression Training. R package version 6.0-84. https://CRAN.R-project.org/package=caret

H. Wickham. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2016.

Hao Zhu (2019). kableExtra: Construct Complex Table with 'kable' and Pipe Syntax. R package version 1.1.0. https://CRAN.R-project.org/package=kableExtra

Gorakala Suresh K, Michelle Usuelli.Building a Recommendation System with R.Packt Publishing Birmingham,h 2015.