Recommender Systems

6/27/2017

DATA643 Project 4

by Tulasi Ramarao

Description

This system recommends beer to beer enthusiasts. The goal of this system will be to recommend a new beer which they might like based on their own beer ratings. To accomplish this task, two collaborative filtering methods are used to compare two beer ratings entered by the reviewers. For example, When a user gives a similar rating for two beers, then these algorithms will consider these two beers to be similar to one another. These two recommendation algorithms are implemented and compared for accuracy. Two algorithms considered are: User based collaborative filtering (UBCF) and Item based collaborative filtering (IBCF). The algorithms are evaluated and compared using different algorithms, normalization techiques, similarity methods and neighborhood sizes. The recommenderlab library is used to accomplish this task.

These two collaborative methods are then combined with an attribute based filtering to recommend beer for customers.

Considering important metrics outside of accuracy like Serendipity in an algorithm is becoming popular to enhance user experience. Serendipity is defined as finding good things by accident. Unexpected surprise that will please a customer is the goal of adding Serendipity into a recommender. For example: A hiker will want to see recommendation of new exciting hiking places and not the same or the same kind he already hiked.

Regarding the beer purchase on beerAdvocate, their customers may want to try something new and exciting. So, the attributes of beer like the beer_style is added to the algorithm to improve the quality of prediction. Both usefulness and Unexpectedness is alo considered when creating these predictions. Usefulness is achieved by choosing overall scores above a certain value and unexpectedness is achieved by choosing another attribute of rating called the the rating_taste. That is because there are customers who rated low for aroma, but high for taste. So, a few rows(100) of beer with medium rating in beer aroma are added into the mix.

Dataset

The Dataset is downloaded from BeerAdvocate, a website for beer fans. The dataset consists of about 1.5 million reviews for years - 1999 to 2011. Each record is composed of a beers name, reviewers ratings on beer features like aroma, palate, appearance and taste. All ratings are on a scale from 1 to 5, 1 being the lowest. The picture below is an example screenshot of a user reviewing Guiness Golden Ale. The user has given an overall review of 3.13 out of 5, and has rated on individual beer features also.

Load the downloaded data from csv files.

```
# R function for graphs
source("displayData.R")

set.seed(3445) # to keep #s from the results the same

# Set the working directory
setwd("/Users/tulasiramarao/Documents/Tulasi/CUNYProjects/DATA643/RPrograms")

# load data from a local drive
beerratings.df <- read.csv("Data/beer_reviews.csv", header = TRUE,stringsAsFactors = FALSE,sep =",")
```

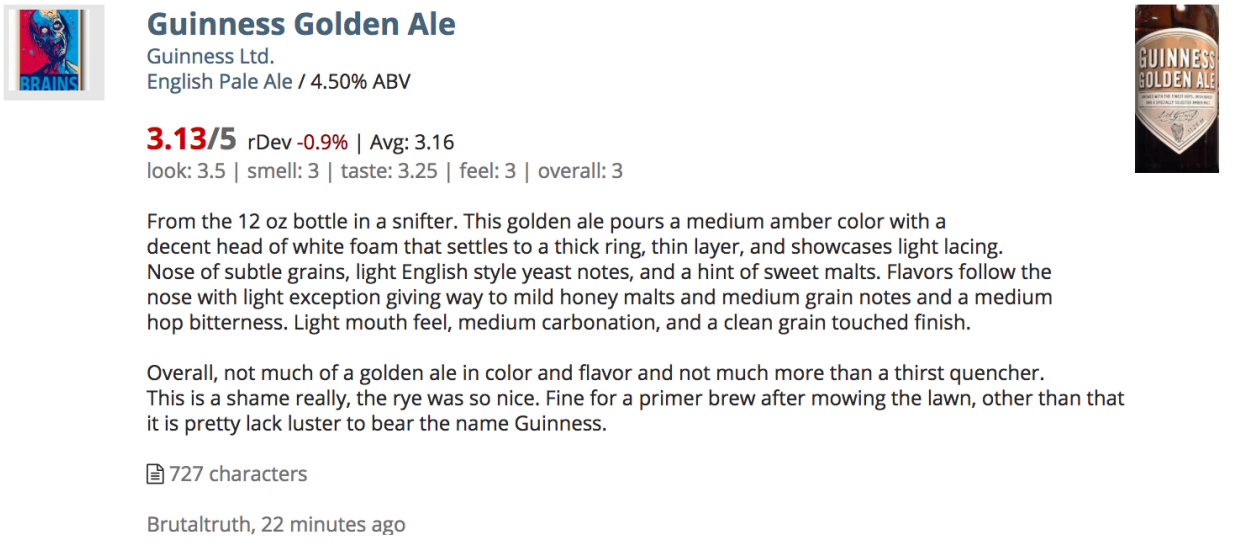


Figure 1:

```
#dim(beerratings.df)
#head(beerratings.df,2)
```

Data preparation:

Data is prepared based on instructions under Chapter 3 of *Building a recommendation system with R* - from Suresh Gorakala)

The beer dataset contains beers that have been rated only a few times. So the ratings are biased because of the lack of datapoints. So its necessary to have a required minimum of users who have rated beers and a minimum of beers that are rated by users. So we will define ratings containing the matrix - with users that have rated at least x beers and beers that have been rated at least y times.

```
#unique(beerratings.df$beer_style) # to possibly use beer_style for serepdipity
# Subsetting the ratings

# a copy for backup purpose
myratings <- beerratings.df
colnames(myratings)
```

```
##  [1] "brewery_id"         "brewery_name"       "review_time"
##  [4] "review_overall"     "review_aroma"       "review_appearance"
##  [7] "review_profilename" "beer_style"         "review_palate"
## [10] "review_taste"       "beer_name"          "beer_abv"
## [13] "beer_beerid"
```

```
myratings <- myratings %>%
  group_by(beer_beerid) %>%
  filter(n()>100)

myratings <- myratings %>%
  group_by(review_profilename) %>%
  filter(n()>50)
```

```
#head(myratings)
#colnames(myratings)

# drop unwanted columns
drops <- c("brewery_name","brewery_id","review_time")
myratings <- myratings[,!(names(myratings) %in% drops)]
#colnames(myratings)
#head(myratings)

#dim(myratings)

myratings.backup <- myratings
```

Serendipity was introduced into this process. Serendipity requires an act of balancing the degree of unexpectedness and the usefulness of the item. When observing the dataset, there were beers with a low score for review_aroma, but scored high for review_taste. So some of the low scoring review_aroma beers were chosen separately and then added to the main dataset and that can create some recommendations that can be a nice surprise for the user. To maintain the usefulness, beer had to have a minimum score of 1.5.

Each beer belongs to a beer_style like Spiced beer, English Strong Ale, etc. The average score is replaced with a new score created by adding the weighted score from beer_style.

```
### SERENDIPITY ###

#myratings <- myratings.backup

#colnames(myratings)
#colnames(beerratings.df)
#head(beerratings.df,10)

# create overall mean based on beer_style ( to add usefulness)
genreMean <- ungroup(myratings) %>%
group_by(beer_style) %>%
summarize(meanScore=mean(review_overall))
#head(genreMean,10)

# merge counts and mean into data frame
myratings <- merge(myratings, genreMean, by.x='beer_style', by.y='beer_style', all.x=T)

myratings$SerenScore <- (as.numeric(myratings$review_overall) + as.numeric(myratings$meanScore))/2
#head(myratings,2)

aromaset <- ungroup(myratings) %>%
filter(review_aroma <= 3) %>% # low score for aroma
filter(review_aroma >= 1.5)  # not too low score for aroma
aromaset100 <- head(aromaset,100)
#colnames(aromaset100)


#min(mydata$review_overall)

# Serendipity surprise is to have review_aroma value below a certain level
# usefulness serenscore should be above a certain value and
scoreval <- 2.5
```

```
aromaval <- 3
myratings <- ungroup(myratings) %>%
group_by(beer_style) %>%
filter(SerenScore > scoreval)

#dim(myratings) #930767 x 12
#dim(aromaset100) #100 x 12 of low aroma
#head(myratings,3)
#head(aromaset100,3)
#str(myratings)
#str(aromaset100)
myratings <- as.data.frame(myratings)

myratingsNew <- rbind(aromaset100,myratings)  # add the dataset created for low review_aroma
dim(myratingsNew) # 930867 x 12
```

```
## [1] 930867     12
```

```
drops <- c("review_overall","meanScore","review_aroma","review_appearance","review_palate")
myratings <- myratings[,!(names(myratings) %in% drops)]
#head(myratings,2)

#dim(myratings)
```

The dimension of the dataset is now 52472 rows and 7 columns.

Using dplyr to reformat - making rating to be the values in the matrix; users as rows, items as columns. Also converting the dataframe to a matrix to feed to the recommenderlab library.

```
beer.reshaped <- dcast(myratings,review_profilename ~ beer_name, value.var = "SerenScore", fill=0, fun.a

# Filling in rownames
rownames(beer.reshaped) = beer.reshaped$review_profilename

# Removing the first column and converting to a matrix
beer.reshaped <- as.matrix(beer.reshaped[,-1])
#head(beer.reshaped)

#beer.reshaped[1:6, 1:6]
```

The matrix size is reduced by converting the matrix to a readRating matrix (from the recommenderlab) coerce into a realRatingMAtrix. This will greatly reduce the file size as shown below.

```
object.size(beer.reshaped)  #  ~ 97.3 MB
```

```
## 102053184 bytes
```

```
beerRealRating <- as(beer.reshaped, "realRatingMatrix")
object.size(beerRealRating) # ~ 145.8 MB
```

```
## 152847344 bytes
```

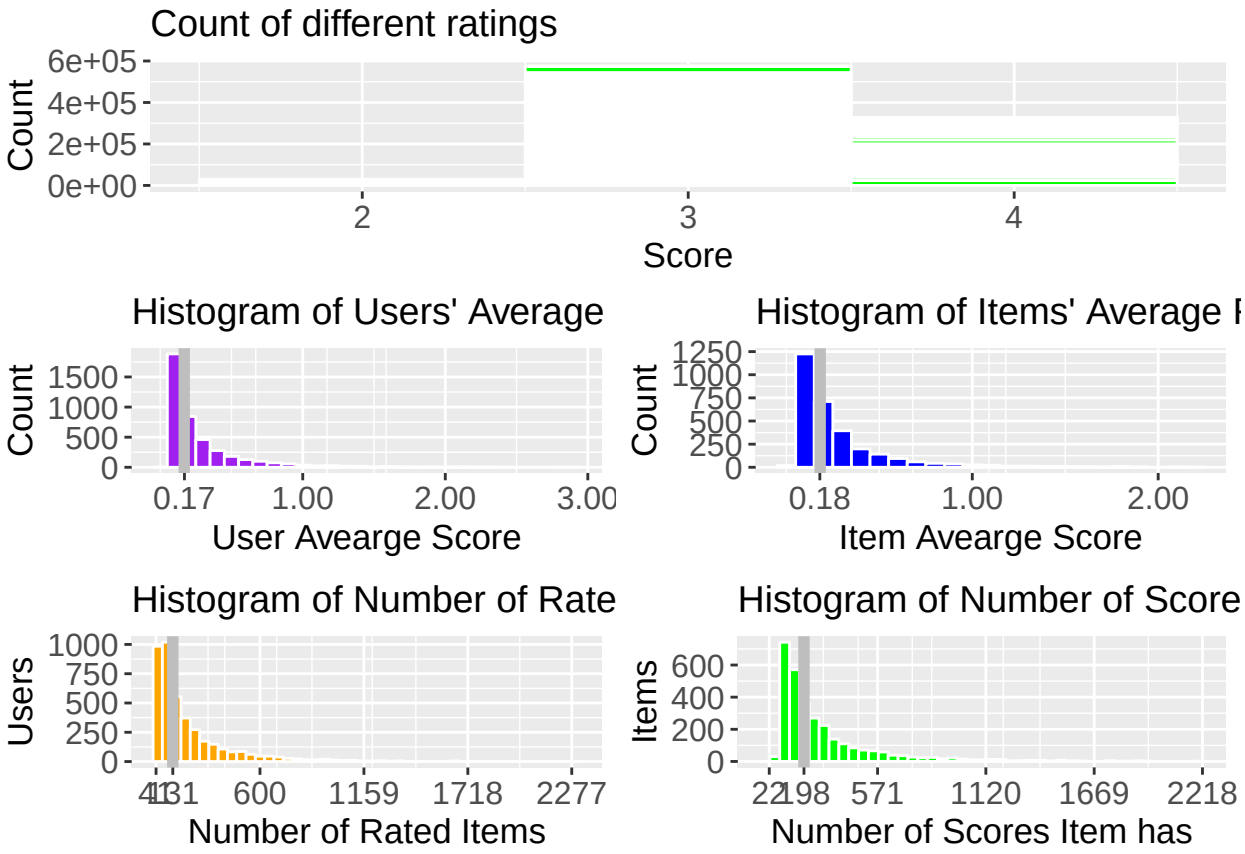
```
# Turn the matrix into a 0 -1 binary matrix
matrix_ones <- binarize(beerRealRating , minRating=1)

master_df <- myratings
master_df$SerenScore <- floor(myratings$SerenScore)
#str(master_df)
```

```
master_df <- myratings %>%
             group_by(SerenScore) %>%
             summarise(n = n())
```

Statistics of the dataset is displayed below.

```
displayData(matrix_sparse = beerRealRating,matrix_ones=matrix_ones,master_df = master_df)
```

```
##
## Attaching package: 'gridExtra'
## The following object is masked from 'package:dplyr':
##
##     combine
```



To increase the computational time, the minimum number for purchase is set and filtered and that will reduce the size of the matrix.

```
#matrix_ones
matrix_ones.filtered <- matrix_ones[,colCounts(matrix_ones) >= 200]
#matrix_ones.filtered
```

Data preparation:

Data is prepared based on instructions under Chapter 3 of Building a *recommendation system with R* - from Suresh Gorakala)

The beer dataset contains beers that have been rated only a few times. So the ratings are biased because of the lack of datapoints. So its necessary to have a required minimum of users who have rated beers and a

minimum of beers that are rated by users. So we will define ratings containing the matrix - with users that have rated at least x beers and beers that have been rated at least y times.

Chapter 4 uses the evaluationScheme to automatically split dataset into Testing and Training sets. So, using this tool to split ratings into 80% and 20%.

```
items_to_keep <- 5
percentage_training <- 0.8
rating_threshold <- 3
n_eval <- 1

#runs for a while
eval_sets <- evaluationScheme(data = beerRealRating, method ="split", train = percentage_training, give
#eval_sets

size_sets <- sapply(eval_sets@runsTrain, length)
size_sets
```

```
## [1] 3340
```

```
#3340 - size
```

Splitting data

To make prediction of ratings, we need to build a recommender. The following sets were extracted by using getData:

Train: The training set
Known: Test set with the item to build the recommendation
Unknown: Test set to test the recommendation Its a 3340 x 3040 realRatingMatrix object, and so nrows and ncolumns can be applied to it.

```
(nr <- nrow(getData(eval_sets,"train"))/nrow(matrix_ones.filtered))
```

```
## [1] 0.7998084
```

80% of data is in the training set as expected.

```
(nr <- nrow(getData(eval_sets,"known"))/nrow(matrix_ones.filtered))
```

```
## [1] 0.2001916
```

They have about the rest of 20% data in the test set.

Recommendation Algorithms:

Two recommendation algorithms are considered here - Used based and the Item based colloborative filtering.

I. User-Based Colloborative Filtering:

This algorithm groups users according to their history of ratings and recommends an item that a user similar to this user (in the same group) liked. So, if user A liked beer 1,2 and 3 and user B liked beer 1 and 2, then beer 3 is a good one to recommend to user B.The assumption of UBCF is that similar users will rate beers similarly. So, the ratings are predicted by first finding a neighborhood of similar users and then aggregating the user ratings to form a prediction.

Popular measures used are Pearson and cosine distance similarity.

a) Optimizing a numeric parameter (Neighborhood size):

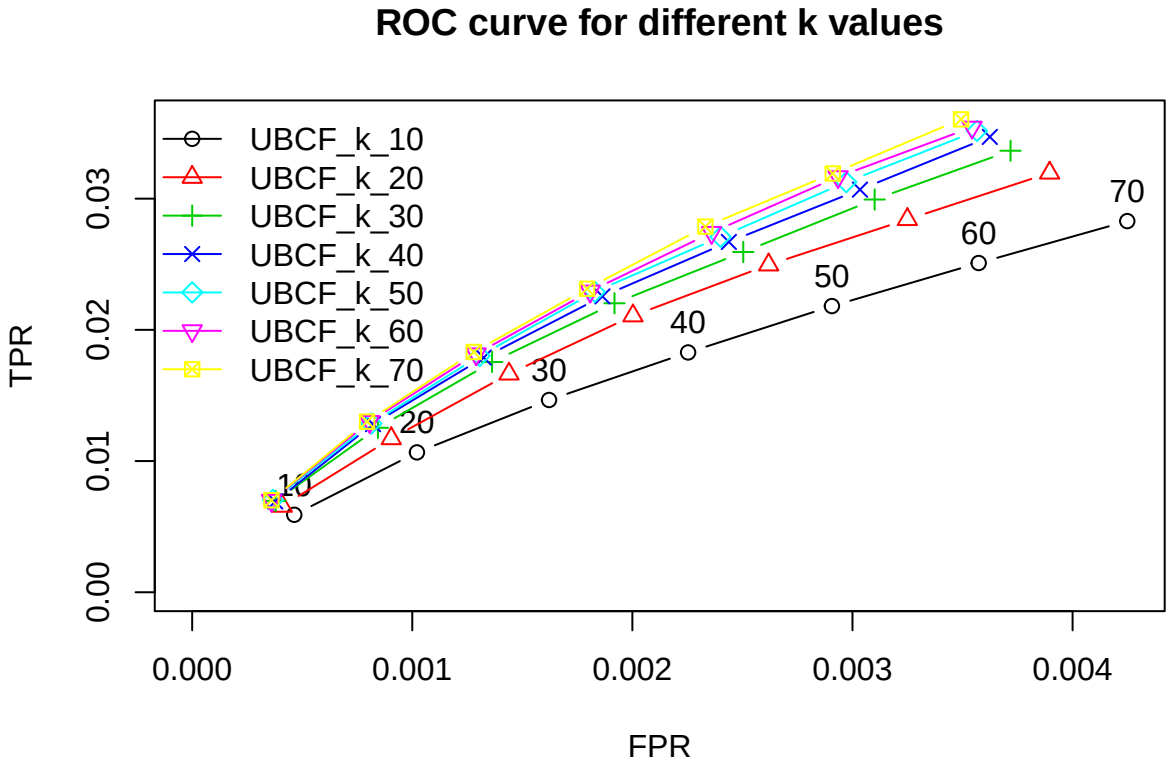
Recommendation models contain a numeric parameter that takes account of the k-closest users/items. We can optimize k, by testing different values of a numeric parameter. So, we can get the value we want to proceed testing with. Default k value is 30. We can explore ranges from 10 and 70. Building and evaluating the models:

```
vector_k <- c(10, 20, 30,40,50,60,70)
records <- c(5, 10, 15, 20, 25)
model_name <- "UBCF"
method_name <- "Cosine"

#define a list of models to evaluate by using lapply( distance metric is cosine )
models_to_evaluate <- lapply(vector_k, function(k) {
  list(name= model_name, param = list(normalize = "Z-score", method = method_name,nn=k))

})

# name the models
names(models_to_evaluate) <- paste0(("UBCF_k_"),vector_k)

# Takes a long time to run...
list_results <- evaluate(x=eval_sets,method = models_to_evaluate, n = vector_k,progress = FALSE)
```

```
## UBCF run fold/sample [model time/prediction time]
##   1  [2.797sec/115.25sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [3.075sec/119.987sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [3.012sec/118.445sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [2.899sec/119.694sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [2.9sec/122.29sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [3.226sec/128.793sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [3.118sec/122.889sec]
```

This evaluation took about 3.25 seconds for each iteration.

```
plot(list_results, annotate = 1, legend ="topleft")
title("ROC curve for different k values")
```



The best performing k can be identified by building a chart for these values with the ROC curve. ROC curve has the best performance for K = 70. So this value will be used in the neighborhood is ideal for all calculations.

Now a similarity matrix is calculated containing all user-to-user similarities using Pearson and Cosine similary measures.

```
model_to_evaluate <- "UBCF"
model_parameters <- list(method="Cosine", nn=70)

model_cosine <- Recommender(getData(eval_sets,"train"),model_to_evaluate,param=model_parameters)
```

Create predictions: This prediction does not predict the beer ratings for test. But this fills up the user 'X' item matrix so that for any userid/beerid, the predicted r can be obtained. Type parameter decides whether its for the ratings or the top-n items.

```
prediction_cosine <- predict(model_cosine,getData(eval_sets,"known"),type="ratings")
as(prediction_cosine,"matrix")[1:5]
```

```
## [1] NA NA NA NA NA
```

```
rmse_cosine <- calcPredictionAccuracy(prediction_cosine, getData(eval_sets, "unknown"))[1]
rmse_cosine
```

```
##     RMSE
## 1.351029
```

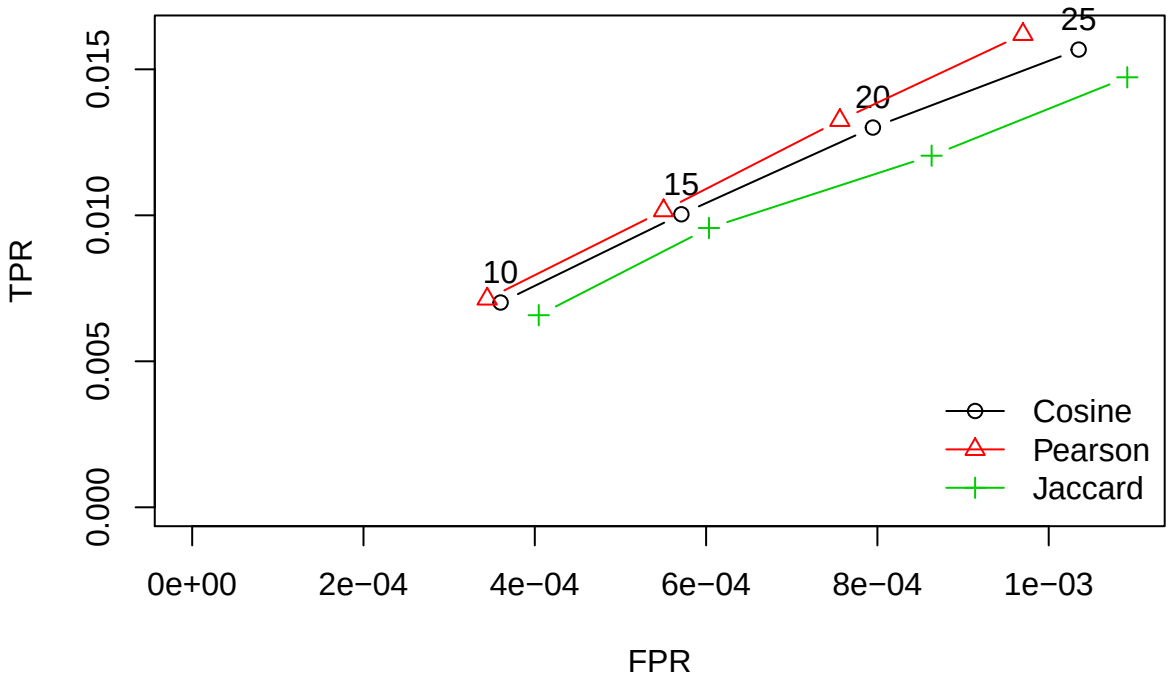
b. Distance methods:

This method gives measurement of the similarity between users/items based on the distance between them. Popular models are pearson, jaccard and cosine.

```
model_to_evaluate <- "UBCF"
kval <- 70
valList <- c(10, 15, 20, 25)

model_parameters1 <- list(normalize = "Z-score",method="Cosine",nn=kval)
model_parameters2 <- list(normalize = "Z-score",method="Pearson",nn=kval)
model_parameters3 <- list(normalize = "Z-score",method="jaccard",nn=kval)


distItem <- list(
   "Cosine" = list(name=model_to_evaluate, param=model_parameters1),
   "Pearson" = list(name=model_to_evaluate, param=model_parameters2),
   "Jaccard" = list(name=model_to_evaluate, param=model_parameters3)

)

# confusion matrix construction
dist_resultsUBCF <- evaluate(eval_sets, distItem, n=valList)
```

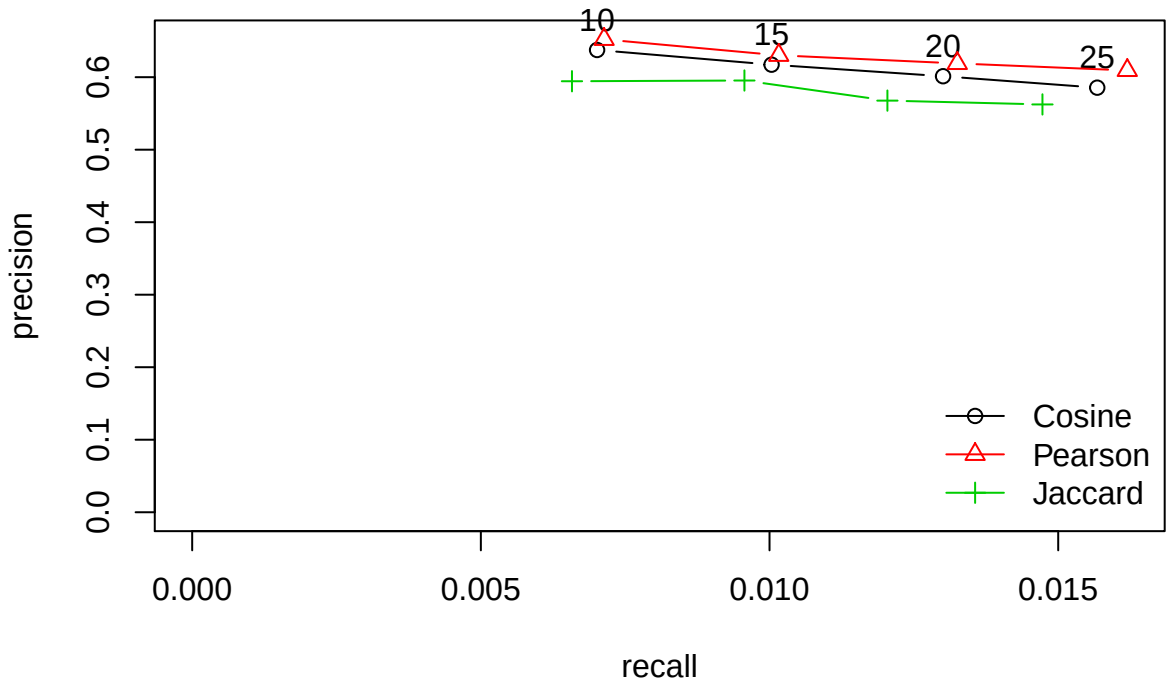
```
## UBCF run fold/sample [model time/prediction time]
##   1  [3.161sec/118.564sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [3.503sec/71.335sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [3.345sec/108.835sec]
```

```
plot(x=dist_resultsUBCF, y ="ROC",annotate=TRUE)
```



From the ROC curve, it can be seen that the performance was best when using the Pearson algorithm (the top most curve).

```
# Draw the precision/Recall curve
plot(x = dist_resultsUBCF, y = "prec/rec", annotate=TRUE)
```



Pearson performed best again in the Precision/Recall curve (the top most curve).

c) Normalization method:

Data needs to be normalized before applying any algorithm. (normalization is done here by taking users averages - which is the mean ratings of every user subtracted from known ratings)

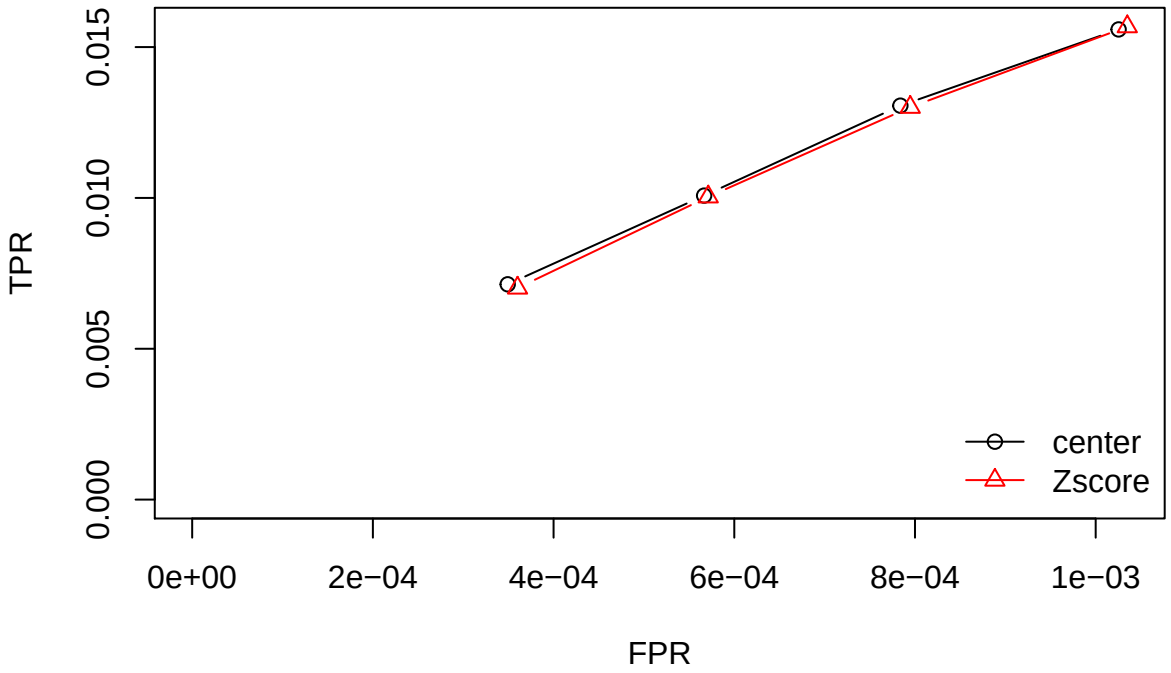
The normalization method is used for Z score using center and z-score parameters to feed the recommenderlab.

```
alg_dist <- list(
   "center" = list(name="UBCF", param=list(normalize = "center",method="Cosine",nn=70)),
   "Zscore" = list(name="UBCF", param=list(normalize = "Z-score",method="Cosine",nn=70))
)

dist_resultsUBCF <- evaluate(eval_sets, alg_dist, n=c(10, 15, 20, 25))
```

```
## UBCF run fold/sample [model time/prediction time]
##   1  [2.144sec/129.228sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [3.628sec/134.464sec]
```

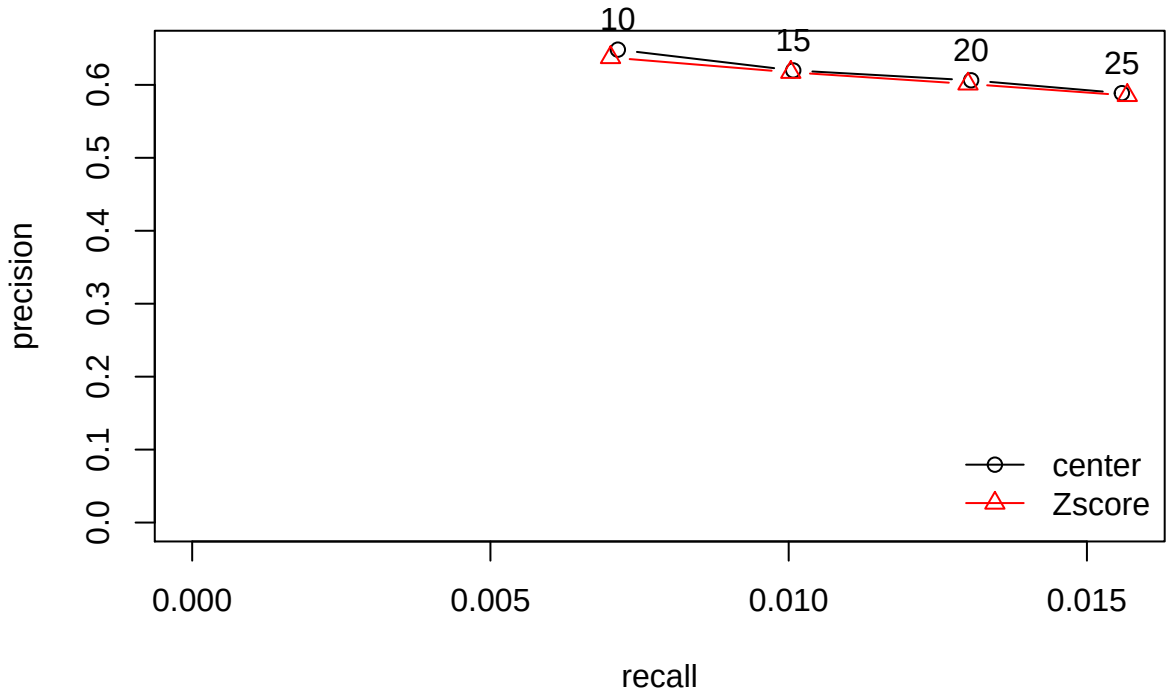
Now the ROC and the precision curve are plotted to compare and choose the best method between center and z-score methods. Pearson is the best in ROC and cosine in Precision/Recall.

```
#plot ROC
plot(x = dist_resultsUBCF, y = "ROC")
```



Center did best in ROC.

```
# Draw the precision curve
plot(x = dist_resultsUBCF, y = "prec", annotate = 1)
```



Center is still the best among these levels of recommendations.

II. Item-Based Collaborative-Filtering Recommender(IBCF):

This is a model based recommender based on the relationship between items inferred from the rating matrix. This model assumes that users prefer beer/items that are similar to other items they like.

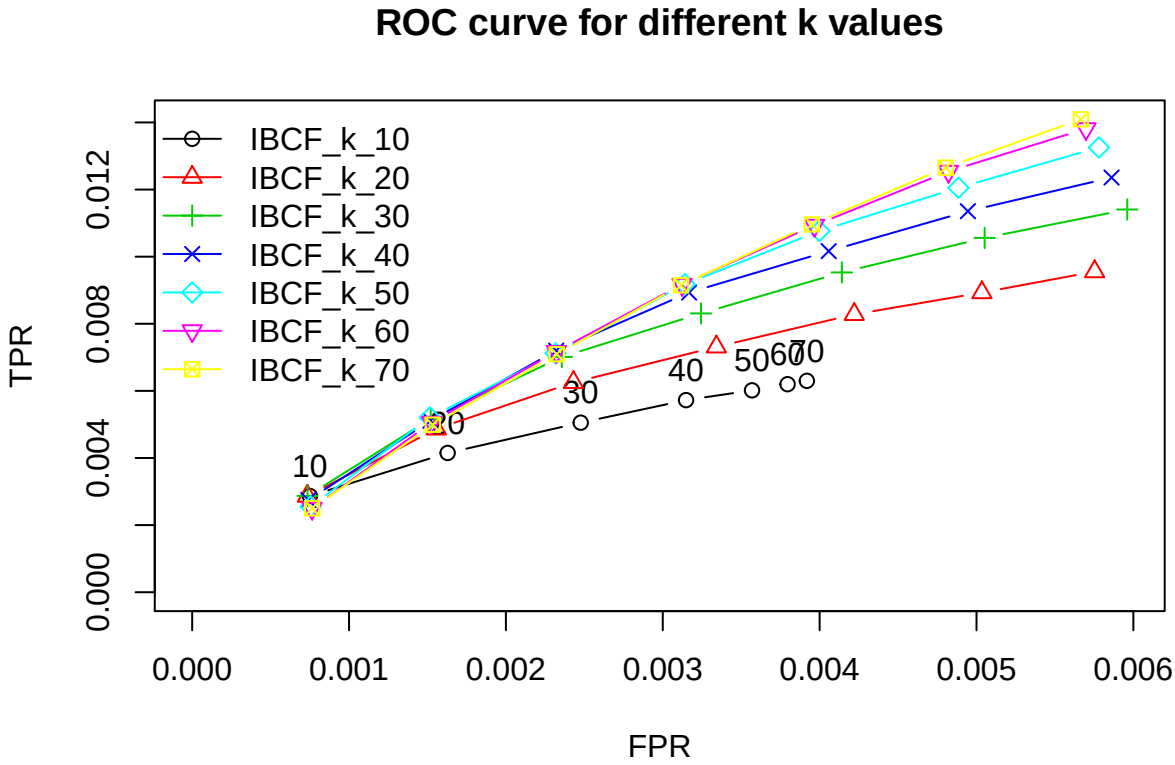
a) Optimizing a numeric paramter((Neighborhood size)):

Recommendation models contain a numeric parameter that takes account of the k-closest users/items. k can be optimized by testing different values of a numeric parameter. So, the value of k can be obtained that will be used to test. Default value for k is 30. Ranges from 5 to 70 are explored.

```
vector_k <- c(10, 20, 30,40,50,60,70)
records <- c(5, 10, 15, 20, 25)
model_name <- "IBCF"
method_name <- "Cosine"

#define a list of models to evaluate by using lapply( distance metric is cosine )
models_to_evaluate <- lapply(vector_k, function(k) {
  list(name= model_name, param = list(normalize = "Z-score", method = method_name,k=k))
})

names(models_to_evaluate) <- paste0(("IBCF_k_"),vector_k)

# takes a long time
list_results2 <- evaluate(x=eval_sets,method = models_to_evaluate, n = vector_k,progress = FALSE)
```

```
## IBCF run fold/sample [model time/prediction time]
##   1  [398.921sec/0.584sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [378.029sec/0.449sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [379.676sec/0.612sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [378.406sec/0.502sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [379.892sec/0.475sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [394.599sec/0.69sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [391.906sec/0.723sec]
```

```
plot(list_results2, annotate = 1, legend ="topleft")
title("ROC curve for different k values")
```



This evaluation took a long time to run about 420 seconds for each iteration.
It can be seen from the ROC curves that the best performance is for k=70. So 70 items will be included in the neighborhood for all calculations.

Now a similarity matrix is calculated containing all item-to-item similarities using Pearson and Cosine similary measures.

```
# k = 70 determined from the ROC curve above for choosing the neighborhood
model_to_evaluate <- "IBCF"
model_parameters <- list(normalize = "Z-Score", method="Cosine", nn=70)

model_cosine <- Recommender(getData(eval_sets,"train"),model_to_evaluate,param=model_parameters)
```

```
## Warning: Unknown parameters: nn
```

```
## Available parameter (with default values):
## k     =  30
## method    =  Cosine
## normalize     =  center
## normalize_sim_matrix  =  FALSE
## alpha     =  0.5
## na_as_zero    =  FALSE
## verbose   =  FALSE
```

```
prediction_cosine <- predict(model_cosine,getData(eval_sets,"known"),type="ratings")

rmse_cosine <- calcPredictionAccuracy(prediction_cosine, getData(eval_sets, "unknown"))[1]
rmse_cosine
```

```
##     RMSE
## 2.088975
```

b) Distance methods:

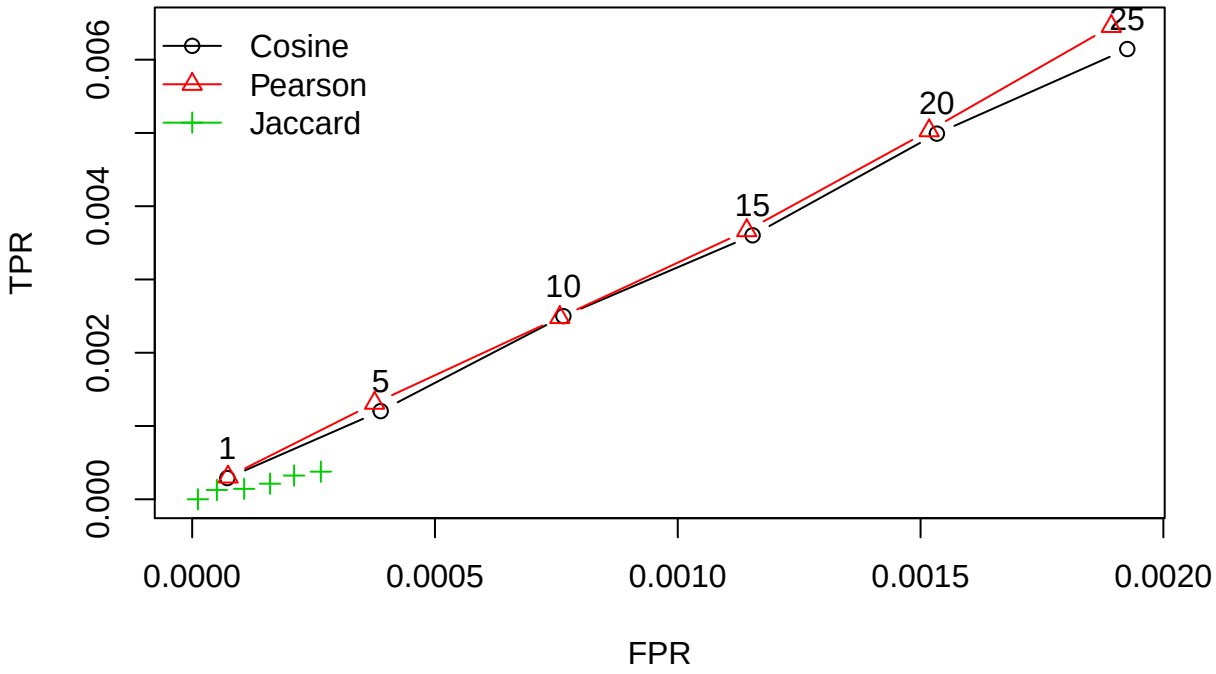
This method gives measurement of the similarity between users/items based on the distance between them. Popular models are pearson and cosine.

```
model_to_evaluate <- "IBCF"
valList <- c(1, 5, 10, 15, 20, 25)

model_parameters1 <- list(normalize = "Z-score",method="Cosine",k=70)
model_parameters2 <- list(normalize = "Z-score",method="Pearson",k=70)
model_parameters3 <- list(normalize = "Z-score",method="jaccard",k=70)


distItem <- list(
   "Cosine" = list(name=model_to_evaluate, param=model_parameters1),
   "Pearson" = list(name=model_to_evaluate, param=model_parameters2),
   "Jaccard" = list(name=model_to_evaluate, param=model_parameters3)

)

dist_resultsIBCF <- evaluate(eval_sets, distItem, n=valList)
```

```
## IBCF run fold/sample [model time/prediction time]
##   1  [424.663sec/0.605sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [273.892sec/0.645sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [325.853sec/0.58sec]
```

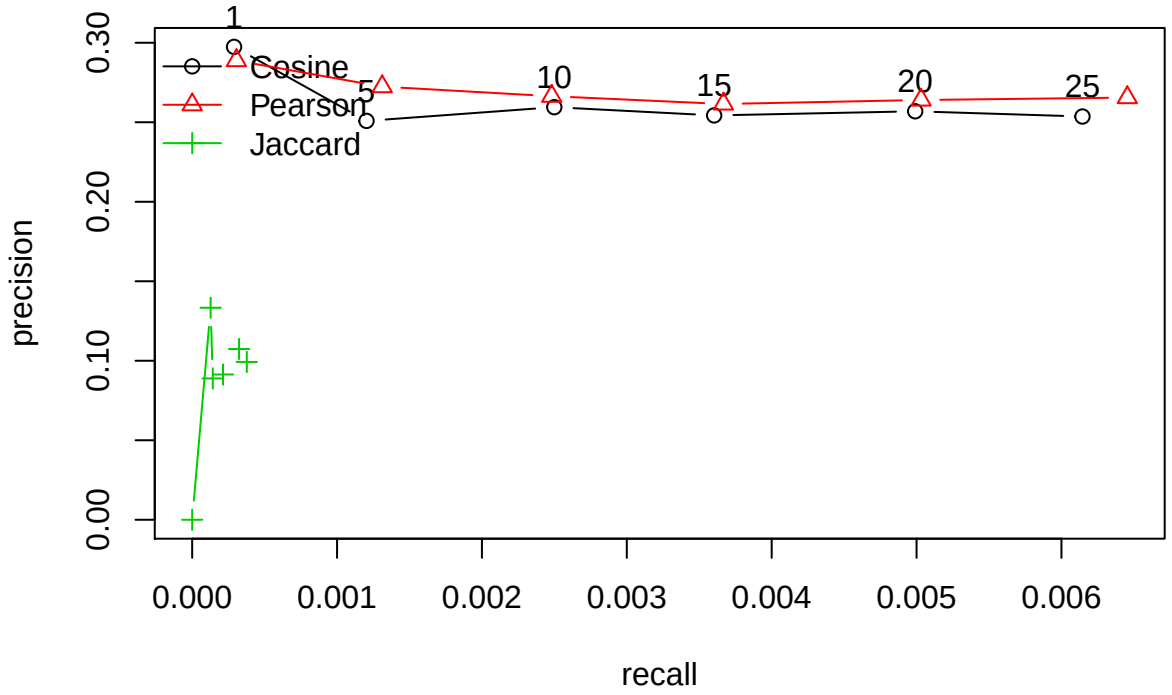
Plot the ROC and the Precision curves to compare the performances of Cosine, Pearson and Jaccard models.

```
#plot ROC
plot(x = dist_resultsIBCF, y = "ROC",legend ="topleft",annotate=1)
```



From the ROC curve, it can be seen that Pearson model did much better.

```
# Draw the precision curve
plot(x = dist_resultsIBCF, y = "prec", annotate = 1,legend ="topleft")
```



Pearson model performed better again.

c) Normalization method:

Data needs to be normalized before applying any algorithm. (normalization is done here by taking users averages - which is known ratings - mean rating of each user)

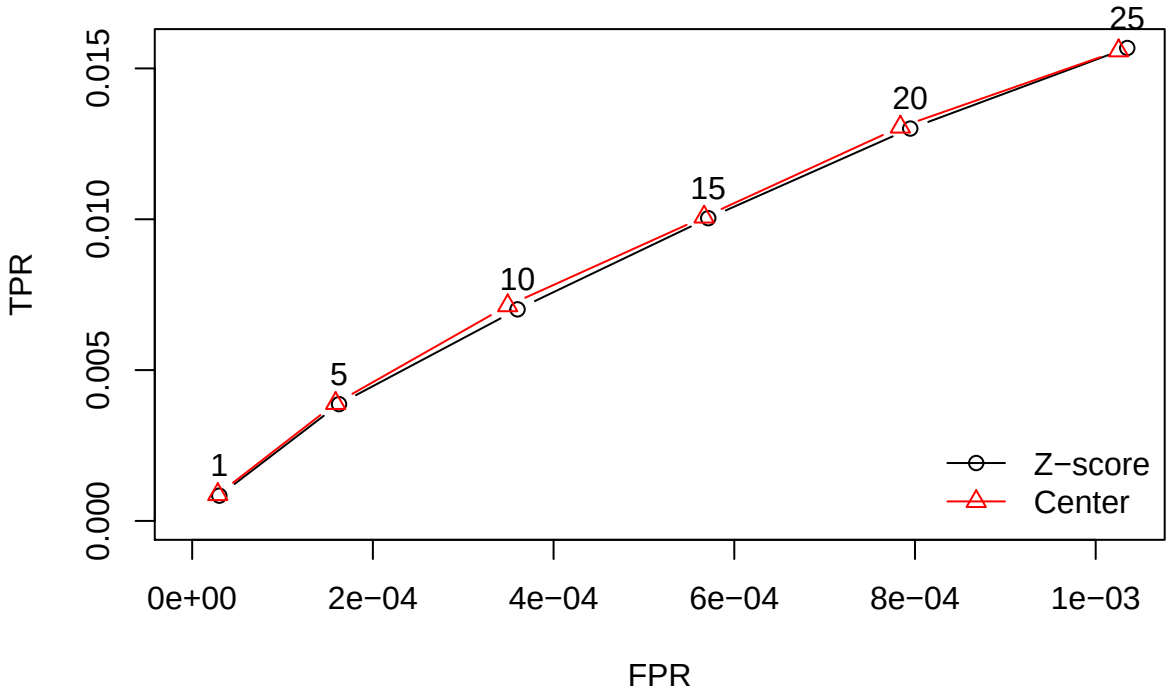
Using normalization method for Z score using center and z-score parameters to feed the recommenderlab.

```
# using k=70
algorithms <- list(
   "Z-score" = list(name="UBCF", param=list(normalize ="Z-score", method="Cosine",nn=70)),
   "Center" = list(name="UBCF", param=list(normalize ="center", method="Cosine",nn=70))
)

# run algorithms, predict next n beers
results <- evaluate(eval_sets, algorithms, n=c(1, 5, 10, 15, 20, 25))
```

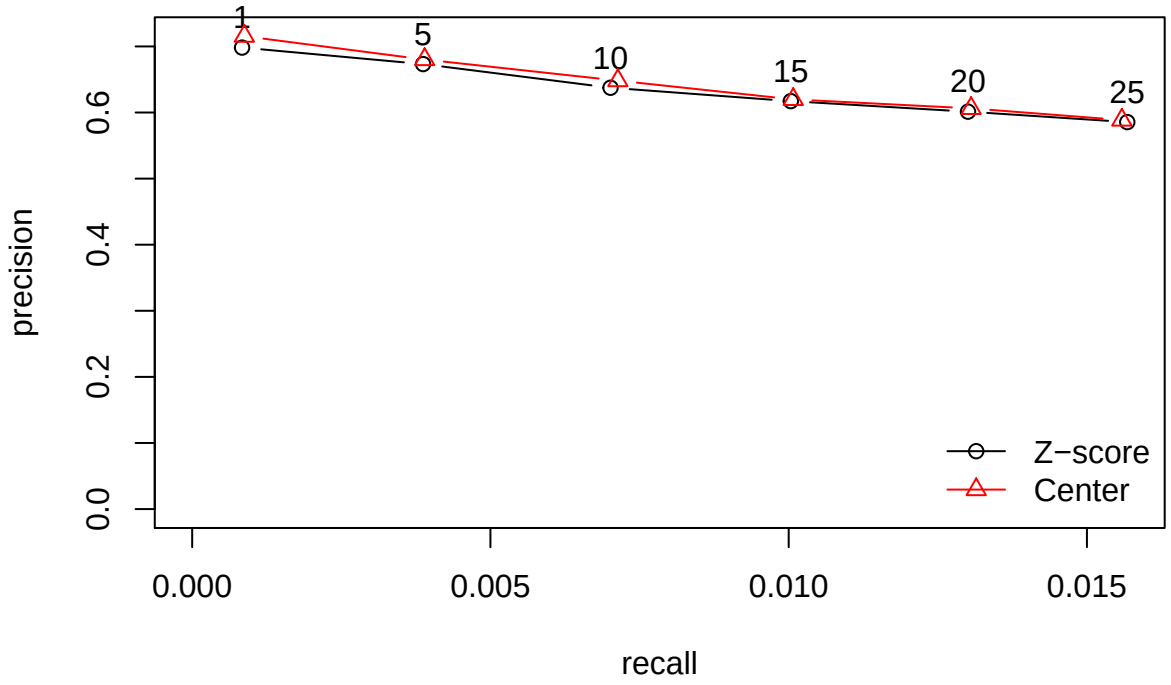
```
## UBCF run fold/sample [model time/prediction time]
##   1  [2.91sec/115.752sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [2.001sec/117.154sec]
```

Comparing the Z-score and the center methods for k=70.

```
#plot ROC
plot(results, y = "ROC", annotate = 1)
```




```
# Draw the precision curve
plot(x = results, y = "prec", annotate = 1)
```



From the above plots, it can be seen that Center peforms well for normalization method.

Findings and Recommendations

```
library(knitr)
ubcf <- c('3.4','70','Pearson','1.351029', 'Pearson','Center')
```

```
ibcf <-c('420','70','Pearson','2.088975', 'Pearson','Pearson')


myresults.df <- data.frame(ubcf,ibcf)


colnames(myresults.df) <- c("UBCF","IBCF")
rownames(myresults.df) <- c("Compilation time in seconds/iteration:","Nearest Neighborhood:", "Best sim

kable(myresults.df,  type = "html",caption="Results")
```

Table 1: Results

	UBCF	IBCF
Compilation time in seconds/iteration:	3.4	420
Nearest Neighborhood:	70	70
Best similarity using:	Pearson	Pearson
RMSE -Distance	1.351029	2.088975
Precision	Pearson	Pearson
Normalized using:	Center	Pearson

The User based Collaborative Filtering(UBCF) model produced a smaller prediction error than the Item based Collaborative filtering(IBCF). UBCF is recommended over IBCF because of the following reasons.

When calculating:
a) Distance methods - UBCF had lower RMSE that indicated that UBCF was a better fit. Distance had good reading for both Pearson and Jaccard models in the case of UBCF, however, the Pearson model had a better ROC and a precision curve. For IBCF, the Jaccard model performed better.
b) Normalization - Z-Score performed well in both IBCF and UBCF.
c) The neightborhood size - UBCF had the lowest RMSE for 70 and IBCF had the lowest RMSE for 30.
d) Compilation time - UBCF also ran much faster than IBCF.

Limitations:

The collaborative systems need history of data to function, so a new item or a new user may not get a rating because they haven't had a history of rating.
Large datasets

2. Implement at least one change to your algorithm, to promote a "business goal" such as greater serendipity, novelty, or diversity.

Serendipity was implemented by downgrading the most popular items to promote less known(medium rated attribute - rating_aroma). The items were also evaluated and chosen because they were unexpected and useful for the user.

3. As part of your textual conclusion, discuss one or more additional experiments that could be performed and/or metrics that could be evaluated only if online evaluation was possible. Also, briefly propose how you would design a reasonable online evaluation environment.

Several research papers have concluded that offline evaluations could not reliably predict the CTR(click through rate) of a user. There were very few times that offline predicted CTR accurately.These conclusions make sense because of several factors that cast doubt on the way offline validation is conducted.

Human factor is a big consideration and is not considered for offline validation. Human factors include age, bias, behaviour and other issues like privacy concerns of users.

Age - Younger users may like recommended beers than older users. The offline dataset does not have the age of the user. So, collecting age and using that feature in the algorithm may help in creating a complete dataset.

Amateur - The ratings from amateur beer consumers may think that aroma or taste is the best they ever tasted and give the highest rating, though the beer may not be that good. If the dataset contained a large number of datapoints of novice beer consumers, then that dataset is not considered optimal.

Bias - User may have a bias towards a particular genres of beer and so may give a high rating to a beer belonging to that genre, not really tasting it. In reality, that particular beer may not be good or the one the user would have liked. This leads to a bias in dataset collection. So, if one suspects that the dataset is baised or incomplete, then avoiding offline evaluation altogether may be a better option.

Behaviour - Users may not receive recommendations on time, or they may not like the description of item, or the name of the beer for some reason. These features that affect human behaviours are not accounted for and the predictive power of the offline validation without considering these unpredictable behavour is hard to predict.

Privacy - The website has registered as well as unregistered users. Unregistered users may have privacy concerns and hence may not be accounted for in the CTR numbers. So they will have a lower CTR. But when the recommender system is tested offline, even the registered users would have had lower rates, since human factors are not considered.

Imperfect dataset - The offline dataset may not give a comprehensive view of all types of users or account for any imperfections in data. So, basically its a dataset without any data collected for imperfections in human behaviour. Its impossible to determine how and when these behaviors happen. The dataset is incomplete and so may not always have the predictive power.

So, the Online evaluation is better, if two or more offline recommender systems, when compared performed equally good for the dataset. Offline is suitable to recommend a highly similar beer but not for a diverse kind. Offline evaluation is perferred only if the dataset represents the real world use cases.

Challenges:

The program had to be compiled several times (several hours each time) because of the program not knitting in RStudio. Other options like cache = TRUE, to speed up compiling and opts_knit$set(verbose=TRUE) to view where the code is executing were added. That showed that the program compiled fully but errored in the end with an error in knitr. In order to meet the project deadline, the plan was to submit a pdf document (combined from 10+ pieces of separetely compiled pdf documents). All the pdfs were created and were ready to be combined. Meanwhile, the night before the deadline, after researching this problem for hours led to a possible solution: an option called - latex_engine: xelatex. And when that was added before knitting, the program knitted successfully and a pdf document was generated to be submitted.

References:

The websites and the book below were used primarily to understand the material in order to create the recommenders.

1. https://github.com/ChicagoBoothML/MachineLearning_Fall2015/blob/master/Programming%20Scripts/MovieLens%20Movie%20Recommendation/R/MovieLens_MovieRecommendation.Rmd
2. https://rpubs.com/tarashnot/recommender_comparison
3. *Buiding a recommendation System with R - Suresh Gorakala,Michele Usuelli*
4. https://rpubs.com/tarashnot/recommender_comparison
5. http://docear.org/papers/a_comparative_analysis_of_offline_and_online_evaluations.pdf

6. http://digitalcommons.uri.edu/cgi/viewcontent.cgi?article=1455&context=theses