Lab 7. Evaluating the Recommender Systems

Preparing the data to evaluate models

We need two sets - trainig set and testing set - to evaluate the model.

Splitting the data

I will split the data so that th training set consists of 80% of the data, and the test set - 20%.

```
percentage_training <- 0.8
```

For each user in the test set, we need to define how many items to use to generate recommendations. For this, I will first check the minimum number of items rated by users to be sure there will be no users with no items to test.

```
min(rowCounts(ratings_movies))
## [1] 18
items_to_keep <- 15 #number of items to generate recommendations
rating_threshold <- 3 # threshold with the minimum rating that is considered good
n_eval <- 1 #number of times to run evaluation

eval_sets <- evaluationScheme(data = ratings_movies,
                              method = "split",
                              train = percentage_training,
                              given = items_to_keep,
                              goodRating = rating_threshold,
                              k = n_eval)
eval_sets
## Evaluation scheme with 15 items given
## Method: 'split' with 1 run(s).
## Training set proportion: 0.800
## Good ratings: >=3.000000
## Data set: 560 x 332 rating matrix of class 'realRatingMatrix' with 55298 ratings.
getData(eval_sets, "train") # training set
## 448 x 332 rating matrix of class 'realRatingMatrix' with 44470 ratings.
getData(eval_sets, "known") # set with the items used to build the recommendations
## 112 x 332 rating matrix of class 'realRatingMatrix' with 1680 ratings.
getData(eval_sets, "unknown") # set with the items used to test the recommendations
## 112 x 332 rating matrix of class 'realRatingMatrix' with 9148 ratings.
qplot(rowCounts(getData(eval_sets, "unknown"))) +
```

```
  geom_histogram(binwidth = 10) +
  ggtitle("unknown items by the users")
```

Bootstrapping the data

Bootrstrapping is another approach to split the data. The same user can be sampled more than once and, if the training set has the same size as it did earlier, there will be more users in the test set.

```
eval_sets <- evaluationScheme(data = ratings_movies,
                              method = "bootstrap",
                              train = percentage_training,
                              given = items_to_keep,
                              goodRating = rating_threshold,
                              k = n_eval)

table_train <- table(eval_sets@runsTrain[[1]])
n_repetitions <- factor(as.vector(table_train))
qplot(n_repetitions) +
  ggtitle("Number of repetitions in the training set")
```

Using k-fold to validate models

This approach is the most accurate one, although it's computationally heavier.

```
n_fold <- 4
eval_sets <- evaluationScheme(data = ratings_movies,
                              method = "cross-validation",
                              k = n_fold,
                              given = items_to_keep,
                              goodRating = rating_threshold)
size_sets <- sapply(eval_sets@runsTrain, length)
size_sets
## [1] 420 420 420 420
```

Evavluating the ratings

I will use the k-fold approach for evaluation. In the following code, I re-define the evaluation sets, build IBCF model and a matrix with predicted ratings.

```
eval_sets <- evaluationScheme(data = ratings_movies,
                              method = "cross-validation",
                              k = n_fold,
                              given = items_to_keep,
                              goodRating = rating_threshold)

model_to_evaluate <- "IBCF"
model_parameters <- NULL

eval_recommender <- Recommender(data = getData(eval_sets, "train"),
                                method = model_to_evaluate,
                                parameter = model_parameters)

items_to_recommend <- 10
eval_prediction <- predict(object = eval_recommender,
                           newdata = getData(eval_sets, "known"),
                           n = items_to_recommend,
                           type = "ratings")

qplot(rowCounts(eval_prediction)) +
  geom_histogram(binwidth = 10) +
  ggtitle("Distribution of movies per user")
```

Now, I compute the accuracy measures for each user. Most of the RMSEs are in the range of 0.8 to 1.4:

```
eval_accuracy <- calcPredictionAccuracy(x = eval_prediction,
                                        data = getData(eval_sets, "unknown"),
                                        byUser = TRUE)
head(eval_accuracy)
##          RMSE       MSE       MAE
## 5   1.3176685 1.7362504 1.1023861
## 6   0.9123219 0.8323312 0.7007555
## 11  1.0053254 1.0106793 0.7444377
## 13  1.3473600 1.8153791 1.0902174
## 15  1.9850663 3.9404883 1.5941144
## 21  1.1488383 1.3198294 0.8546132
qplot(eval_accuracy[, "RMSE"]) +
  geom_histogram(binwidth = 0.1) +
```

```
  ggtitle("Distribution of the RMSE by user")
```

In order to have a performance index for the whole model, I specify *byUser* as FALSE and compute the average indices:

```
eval_accuracy <- calcPredictionAccuracy(x = eval_prediction,
                                        data = getData(eval_sets, "unknown"),
                                        byUser = FALSE)
eval_accuracy
```

```
##      RMSE       MSE       MAE
## 1.1289428 1.2745119 0.8348615
```

Evaluating the recommendations

Another way to measure accuracies is by comparing the recommendations with the purchases having a positive rating.

```
results <- evaluate(x = eval_sets,
                    method = model_to_evaluate,
                    n = seq(10, 100, 10))
```

```
## IBCF run fold/sample [model time/prediction time]
##   1  [7.47sec/0.46sec]
##   2  [7.09sec/0.48sec]
##   3  [7.32sec/0.45sec]
##   4  [8.34sec/0.47sec]
```

```
head(getConfusionMatrix(results)[[1]])
```

```
##           TP        FP       FN       TN precision     recall        TPR
## 10  3.142857  6.857143 67.02857 239.9714 0.3142857 0.04466012 0.04466012
## 20  6.321429 13.678571 63.85000 233.1500 0.3160714 0.08891903 0.08891903
## 30  9.192857 20.807143 60.97857 226.0214 0.3064286 0.13193468 0.13193468
## 40 11.921429 28.078571 58.25000 218.7500 0.2980357 0.17033031 0.17033031
## 50 14.678571 35.321429 55.49286 211.5071 0.2935714 0.20963292 0.20963292
## 60 17.321429 42.678571 52.85000 204.1500 0.2886905 0.24824396 0.24824396
##           FPR
## 10 0.02719801
## 20 0.05426466
## 30 0.08295660
## 40 0.11213301
## 50 0.14119344
```

```
## 60 0.17070228
```

In order to have a look at all the splits at the same time, I sum up the indices:

```
columns_to_sum <- c("TP", "FP", "FN", "TN")
indices_summed <- Reduce("+", getConfusionMatrix(results))[, columns_to_sum]
head(indices_summed)
```

```
##           TP         FP       FN       TN
## 10 12.75714  27.17143 279.9857 948.0857
## 20 25.07143  54.78571 267.6714 920.4714
## 30 36.95000  82.83571 255.7929 892.4214
## 40 49.04286 110.67143 243.7000 864.5857
## 50 61.27143 138.37143 231.4714 836.8857
## 60 73.42857 166.14286 219.3143 809.1143
```

Finally, I plot the ROC and the precision/recall curves:

```
plot(results, annotate = TRUE, main = "ROC curve")
```

```
plot(results, "prec/rec", annotate = TRUE, main = "Precision-recall")
```

Comparing models

In order to compare different models, I define them as a following list: - Item-based collaborative filtering, using the Cosine as the distance function - Item-based collaborative filtering, using the Pearson correlation as the distance function - User-based collaborative filtering, using the Cosine as the distance function - User-based collaborative filtering, using the Pearson correlation as the distance function - Random recommendations to have a base line

```
models_to_evaluate <- list(
IBCF_cos = list(name = "IBCF",
                param = list(method = "cosine")),
IBCF_cor = list(name = "IBCF",
                param = list(method = "pearson")),
UBCF_cos = list(name = "UBCF",
                param = list(method = "cosine")),
UBCF_cor = list(name = "UBCF",
                param = list(method = "pearson")),
random = list(name = "RANDOM", param=NULL)
```

```
)
```

Now, I define a different numbers of recommended movies and run and evaluate the models:

```
n_recommendations <- c(1, 5, seq(10, 100, 10))
list_results <- evaluate(x = eval_sets,
                         method = models_to_evaluate,
                         n = n_recommendations)
## IBCF run fold/sample [model time/prediction time]
##   1  [7.23sec/0.2sec]
##   2  [9.79sec/0.39sec]
##   3  [8sec/0.51sec]
##   4  [7.17sec/0.38sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [17.35sec/0.37sec]
##   2  [17.72sec/0.28sec]
##   3  [16.46sec/0.37sec]
##   4  [18.5sec/0.34sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [0.02sec/8.1sec]
##   2  [0sec/9.29sec]
##   3  [0.01sec/8.15sec]
##   4  [0.01sec/6.54sec]
## UBCF run fold/sample [model time/prediction time]
##   1  [0.02sec/17.52sec]
##   2  [0sec/17.32sec]
##   3  [0.02sec/15.24sec]
##   4  [0sec/9.46sec]
## RANDOM run fold/sample [model time/prediction time]
##   1  [0.01sec/0.21sec]
##   2  [0sec/0.26sec]
##   3  [0sec/0.19sec]
##   4  [0sec/0.33sec]
sapply(list_results, class) == "evaluationResults"
## IBCF_cos IBCF_cor UBCF_cos UBCF_cor   random
##     TRUE     TRUE     TRUE     TRUE     TRUE
avg_matrices <- lapply(list_results, avg)
head(avg_matrices$IBCF_cos[, 5:8])
##    precision      recall         TPR         FPR
```

```
## 1  0.3989979 0.005637331 0.005637331 0.00239457
## 5  0.3221017 0.020800675 0.020800675 0.01344330
## 10 0.3195863 0.043228764 0.043228764 0.02720080
## 20 0.3140114 0.085113290 0.085113290 0.05508793
## 30 0.3085196 0.127515994 0.127515994 0.08360939
## 40 0.3071197 0.171622620 0.171622620 0.11176470
```

Identifying the most suitable model

I compare the models by building a chart displaying their ROC curves and Precision/recall curves.

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



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


Optimizing a numeric parameter

IBCF takes account of the k-closest items. I will explore more values, ranging between 5 and 40:

```
vector_k <- c(5, 10, 20, 30, 40)
models_to_evaluate <- lapply(vector_k, function(k){
  list(name = "IBCF",
       param = list(method = "cosine", k = k))
})
names(models_to_evaluate) <- paste0("IBCF_k_", vector_k)
```

Now I will build and evaluate the same models:

```
n_recommendations <- c(1, 5, seq(10, 100, 10))
list_results <- evaluate(x = eval_sets,
                         method = models_to_evaluate,
                         n = n_recommendations)
## IBCF run fold/sample [model time/prediction time]
##   1  [8.16sec/0.41sec]
##   2  [7.08sec/0.19sec]
##   3  [8.11sec/0.39sec]
```

```
##   4  [10.01sec/0.47sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [9.34sec/0.28sec]
##   2  [9.31sec/0.39sec]
##   3  [10.56sec/0.37sec]
##   4  [9.18sec/0.3sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [4.7sec/0.17sec]
##   2  [5sec/0.34sec]
##   3  [5sec/0.19sec]
##   4  [4.35sec/0.23sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [5.72sec/0.2sec]
##   2  [8.15sec/0.39sec]
##   3  [7.86sec/0.2sec]
##   4  [8.45sec/0.47sec]
## IBCF run fold/sample [model time/prediction time]
##   1  [7.02sec/0.37sec]
##   2  [8sec/0.25sec]
##   3  [9.03sec/0.32sec]
##   4  [7.7sec/0.41sec]
```

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

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

Based on the ROC curve's plot, the k having the biggest AUC is 10. Another good candidate is 5, but it can never have a high TPR. This means that, even if we set a very high n value, the algorithm won't be able to recommend a big percentage of items that the user liked. The IBCF with k = 5 recommends only a few items similar to the purchases. Therefore, it can't be used to recommend many items.

Based on the precision/recall plot, k should be set to 10 to achieve the highest recall. If we are more interested in the precision, we set k to 5.