Prediction of ratings on the MovieLens dataset with limited resource

Introduction.

As per September 2019, the MovieLens dataset contains 10M ratings and 95580 tags applied to 10681 movies by 71567 users of the online movie recommender service MovieLens. The whole dataset is split to two: first part is to create an algorithm with parameters to increase a likelihood of calculating a right rating value. And second part is a validation dataset to predict rating values for each record which is unknown.

The goal is to create an algorithm with high accuracy predictions. However, as I am going to show further on, there are obvious time and memory limitations that we should account for.

Steps I took include generation of two datasets: edx and validation. Afterwards, I considered different approaches to solve the outlined supervised problem and decided to keep one of them to submit my predictions. I will explain my choice in the Analysis section.

Methods and Analysis.

Step I. Conventional algorithms: Linear Regression, SVM and Random Forest.

My first approach was to try conventional algorithms such as Linear Regression, Support Vector Machine (SVM) and Random Forest. To assess their performance and choose one, I have decided to utilize repeated cross-validation with 10 folds and 3 repeats, a standart approach to evaluate machine learning algorithms. Unfortunately, I could only achieve my goal by significantly reducing in size the training dataset 'edx'. The fragment of code is presented below:

```
# lets take a subset of edx dataset
edx_sub <- edx[1:250,]

# prepare training scheme
control <- trainControl(method="repeatedcv", number=10, repeats=3)
# Naive Bayes
#set.seed(7)
#fit.nb <- train(rating~ as.factor(movieId) + as.factor(userId), data=edx_sub, meth
# Linear Regression
```

```
set.seed(7)
fit.lm <- train(rating~ as.factor(movieId) + as.factor(userId), data=edx_sub, metho
# SVM
set.seed(7)
fit.svm <- train(rating~ as.factor(movieId) + as.factor(userId), data=edx_sub, meth
# Random Forest
set.seed(7)
fit.rf <- train(rating~ as.factor(movieId) + as.factor(userId), data=edx_sub, metho
# collect resamples
results <- resamples(list(LM=fit.lm, SVM=fit.svm, RF=fit.rf))

# summarize differences between modes
summary(results)
```

Based on this approach, Random Forest delivered better performance (lowest root-mean-square deviation).
I did not proceed with this approach, as I have certain RAM limits. Although I could utilize cloud-computing resource (renting EC2 instance on AWS and run my analysis there), I am not sure my peer-group evaluators will have the same option. So, I moved to the next method.

Step II. RecommenderLab.

The second approach was to utilize a RecommenderLab package available for R.
This approach requires a data preparation, reshaping to prepare a special data structure so-called recommenderlab sparse-matrix.
As the data is reshaped and preprocessed, recommenderlab allows us to run analysis based on different methods and parameters:

- UBCF: User-based collaborative filtering
- IBCF: Item-based collaborative filtering
  Parameter 'method' decides similarity measure: Cosine or Jaccard.
  The fragment of code is presented as followed:

```
# If not installed, first install following packages in R:
install.packages('recommenderlab')
library(recommenderlab)
library(reshape2)

# Let's leave only columns of interest: movieId, userId and rating:
tr <- edx[,1:3]

# Use acast to convert above data as follows
# rows - columns matrix with rating filled as values:
g<-acast(tr, userId ~ movieId)
```

```r
# Convert it to matrix format:
R<-as.matrix(g)

# Convert R into realRatingMatrix data structure
# realRatingMatrix is a recommenderlab sparse-matrix like data-structure
r <- as(R, "realRatingMatrix")

# normalize the rating matrix
r_m <- normalize(r)
as(r_m, "list")

# Can also turn the matrix into a 0-1 binary matrix
r_b <- binarize(r, minRating=1)
as(r_b, "matrix")

# Create a recommender object (model)
# I use IBCF (Item-based collaborative filtering) with Jaccard-similarity parameter

rec=Recommender(r[1:nrow(r)],method="IBCF", param=list(normalize = "Z-score",method
```

Unfortunately, algorithm is stuck on the point of reshaping to recommenderlab sparse-matrix. Then, I receive an error constituting the problem of memory shortage.

Step III. Least squared - calculate it yourself.

After attempting conventional algorithms and RecommenderLab package with no luck, receiving RAM shortage error messages, I have decided to return to where we started it all: least squared calculated manually. At least there is a chance that this simple algorithm will be able to parse and analyze million of records. At this point I am not expecting a great accuracy, but I would like to see how it all will work.

We discussed that setting our unknown rating to simple average is much better than any random number (remember, RMSE of 1.05 vs RMSE of 1.49, dsbook section "Recommendation systems"). Adding movie and user effects draws error even lower (0.885). It means, our prediction model should include mean rating value, movieId and userId to account for those effects. Moreover, we utilize regularization technique by adding a penalization term to account for noisy estimates with either very high rating or low rating and very small number of ratings overall. So, when our b is far from zero we should penalize our estimated value.

Out of curiosity I run cross-validation with different lambda ranging from 0 to 25 with 0.25 step. By drawing a qplot, I saw that with larger value of lambda I receive lower estimate for my accuracy. I've decided to set lambda at 0.25 as it was a local maximum on my drawing and small enough number different from zero. I would not give up on penalization term to still decrease the effect of those noisy estimates.

My predicted rating was calculated as following (fragment of the code):

```
# Calculate predicted rating:
predicted_ratings <- validation %>%
    left_join(b_i, by = "movieId") %>%
    left_join(b_u, by = "userId") %>%
    mutate(rating = mu + b_i + b_u)
```

Another interesting consideration was pertained to rounding problem. My predicted ratings were digits with floating point. I had to figure out a rounding rule to deliver predictions such that it's possible to calculate Accuracy for the following grading. I assumed that accuracy might be calculated as something as

```
 mean(predicted_rating == real_rating)
```

Thus, I decided to round my predictions. At first, I thought rounding predictions to the closest halves digits. One way of doing would be:

```
mutate(rating = (ceiling(rating*2) /2))
```

But then, I remembered Question # 8 from Quiz, solving which implied that most of ratings are whole stars ratings. This matter gives me a reason to reconsider my rounding rule to the following:

```
mutate(rating = (round(rating, digits = 0)))
```

I, then, added my predictions to validation set and wrote it to the submission file:

```
# Ratings will go into the CSV submission file below:
write.csv(validation %>% select(userId, movieId, rating),
          "submission.csv", na = "", row.names=FALSE)
```

Results.

My goal was to predict rating values on validation set (10k rows) within limited computational resources I had. I added a rounding rule to approximate floating numbers to the closest wholes as, based on Quiz analysis we performed, we saw that there are more whole numbers then halves. I did run cross-validation tackling different lambda values on separate file to compute accuracy of my predictions. My estimation showed that Accuracy should be no higher than 37%. With the simple approach I adopted within memory limitations, I consider it is a fair value. However, it won't bring me extra points for Accuracy achievements.

In my R-file I left only the approach I used to generate predictions for submission file, namely algorithm I described in Step III. The full working code won't be a part of the EDX-submission, but can be found in my github public repository.

Conclusion.

My goal was to achieve highest accuracy of prediction for rating values on large MovieLens 10M dataset with resource limitation, memory per se. In the present report I outlined different approaches I tackled. My approaches can be divided to three different steps:

First, I run cross-validation on three classical algorithms: linear regression, SVM and Random Forest.
Random Forest demonstrated a promising performance based on small subset of training data. Unfortunately, I wasn't able to solve a problem of memory shortage. I also need to be mindful of resources my peer-evaluators possess.

Second, I utilized RecommenderLab package in R, based on item-collaborative filtering technique. As it's been shown in the Analysis section, preprocessing appeared to be resource-demanding as well. While reshaping matrix to recommenderlab's sparce-matrix, I received an error warning me about shortage of memory allocated to the process.

Third, I tried to proceed with classical dsbook technique - linear squared optimization. For that, I utilized regularization technique involving rating mean, user and movie effects with penalization term. This approach was successfully implemented even on the large MovieLens dataset with total 10M ratings. I used data from edx to generate predictions for validation set.

The latest approach doesn't incite high accuracy, but it provides with opportunity to run analysis on most computers my peer group may have considering large number of records.

Further considerations and possible extensions.

As I briefly mentioned before, this task can be solved by running more powerful algorithms (Step I) with help of cloud-computational power. One way of doing so is to rent a, say, EC2 instance on AWS and run R-file there.

Moreover, the nature of the problem sparks some ideas to experiment with Deep Learning - approaches. The models can be built based on LSTM or MLP neural networks to see if there could be even better accuracy achieved.

Thank you!