MovieLens Project

Egar Garcia

2/11/2019

Contents

1 Introduction

1.1 Objective

The objective of the project considered in this report is to create a movie recommendation system using the MovieLens dataset, actually a small subset of MovieLens of 10M has been provided.

For the scope of this project, by recomendation we understand the action of predicting the rating that a particular user gives to a particular movie. I.e. the purpose of is not to retrieve a list of recomended movies given an user, instead is predicting the rating given a set of pairs movie-user.

1.2 Goal

As a part of this project the trininig and validation sets have been given, in the form of an R script that generates them. The goal is to reach an RMSE of `0.87750` or less for the system's predictions agains the groud thuth, applied to the validation set.

1.3 Key steps

In this report different different methods are being developed, tested and evaluated, in order to get a final solution, the steps followed to do this work are the following:

- Analysis of the data
- Creation of a metodology for testing and evaluation
- Evaluation of the deeloped methods
- Analisys of the results for the diffent methods
- Chosing one method as solution

1.4 Datasets generation

The following is the code that has been provided to generate the datasets to develop and tests the different methods used for this project. It generates a set called `edx` to be used for development/training and a validation set called `validation` not used at all for development/training and just used for the performance's evaluation of the methods.

```
##########################################################
# Create edx set, validation set, and submission file
##########################################################

# Note: this process could take a couple of minutes

if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")

# MovieLens 10M dataset:
# https://grouplens.org/datasets/movielens/10m/
# http://files.grouplens.org/datasets/movielens/ml-10m.zip

dl <- tempfile()
download.file("http://files.grouplens.org/datasets/movielens/ml-10m.zip", dl)

ratings <- read.table(text = gsub("::", "\t", readLines(unzip(dl, "ml-10M100K/ratings.dat"))),
                      col.names = c("userId", "movieId", "rating", "timestamp"))

movies <- str_split_fixed(readLines(unzip(dl, "ml-10M100K/movies.dat")), "\\::", 3)
colnames(movies) <- c("movieId", "title", "genres")
movies <- as.data.frame(movies) %>% mutate(movieId = as.numeric(levels(movieId))[movieId],
                                           title = as.character(title),
                                           genres = as.character(genres))

movielens <- left_join(ratings, movies, by = "movieId")

# Validation set will be 10% of MovieLens data
```

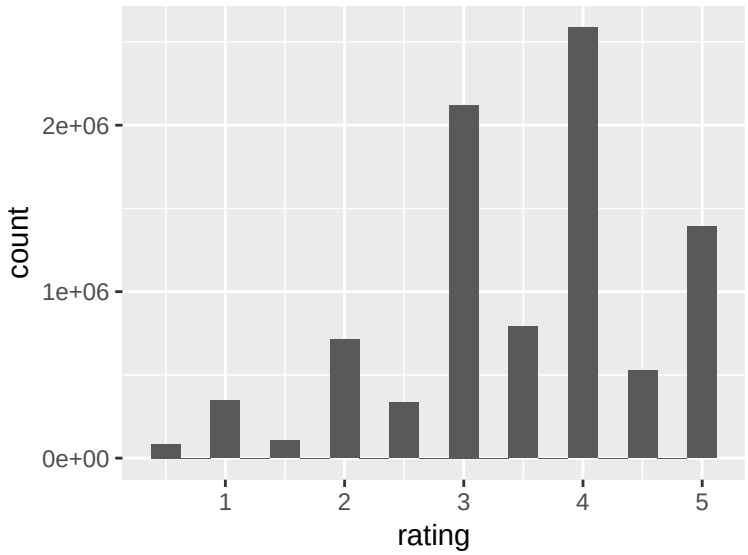
```r
set.seed(1)
test_index <- createDataPartition(y = movielens$rating, times = 1, p = 0.1, list = FALSE)
edx <- movielens[-test_index,]
temp <- movielens[test_index,]

# Make sure userId and movieId in validation set are also in edx set

validation <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

# Add rows removed from validation set back into edx set

removed <- anti_join(temp, validation)
edx <- rbind(edx, removed)

rm(dl, ratings, movies, test_index, temp, movielens, removed)
```

2 Analysis of the data

2.1 A first look

Let's take an initial look at how the ratings are distributed in the given training set `edx`, by plotting a histogram to account the amount of the different ratings given by the customers.

```r
edx %>%
  ggplot() +
  geom_histogram(aes(x = rating), binwidth = 0.25)
```



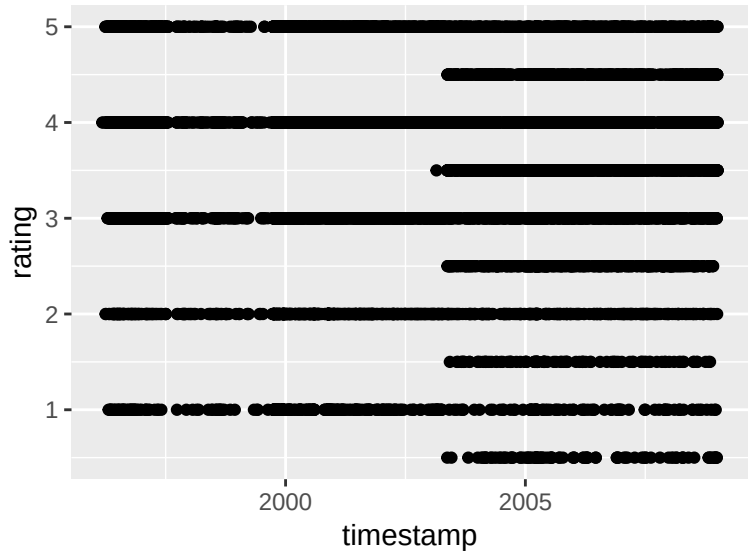
It can be observed that ratings can be interpreted as the number of starts from 1 to 5 that users give to a movie, however it can be seen that ratings ending with a half star are also used, at first impression looks like half start ratings are not very popular among users.

Let's visualize the data from another point of view, this time plotting the ratings against the timestamp, we can use `lubridate` to transform the timestamps to a more friendly format.

```
library(lubridate)
```

For exploratory purposes lets just plot using a small subset of the dataset, since using the whole one might take a lot of time and resources.

```
edx[createDataPartition(y = edx$rating, times = 1, p = 0.001, list = FALSE),] %>%
  ggplot(aes(x = as_datetime(timestamp), y = rating)) +
  geom_point() +
  labs(x = 'timestamp', y = 'rating')
```



Now something interesting can be observed, it looks like ratings ending in half-stars were permitted after certain point in time, and before that time just full-stars were allowed.

2.2 Identifying the point in time to split the data

To find the point in time were ratings ending in half star started to appear, the following code can be used. It basically gets the minimum timestamp in the dataset where a rating with half star is found.

```
half_stars_startpoint <- min(filter(edx, (rating * 2) %% 2 == 1)$timestamp)
```

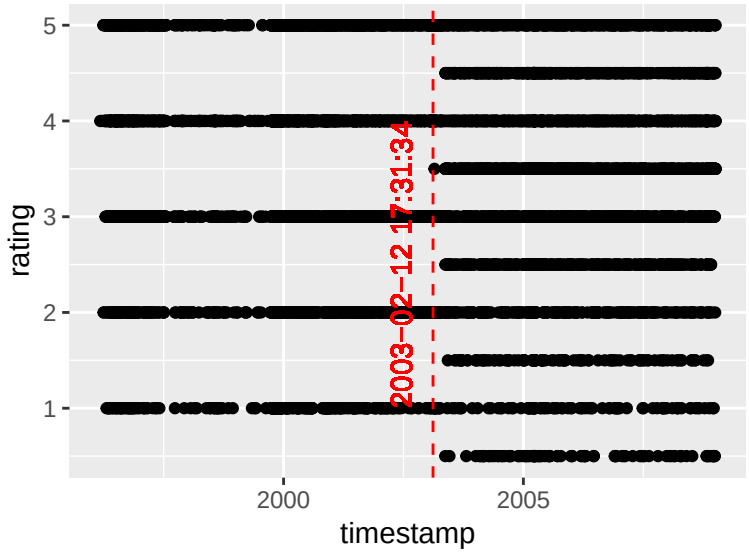
It can be seen that the point in time when half-star ratings started to appear was on **2003-02-12 17:31:34**. This can be done converting `half_stars_startpoint` to a more redable representation using the following code.

```
as_datetime(half_stars_startpoint)
```

Again, let's plot the ratings against the timestamp, but this time adding a vertical line indicating the point in time where half-star ratings were allowed.

```
edx[createDataPartition(y = edx$rating, times = 1, p = 0.001, list = FALSE),] %>%
  ggplot(aes(x = as_datetime(timestamp), y = rating)) +
  geom_point() +
  geom_vline(aes(xintercept = as_datetime(half_stars_startpoint)),
             color = "red", linetype = "dashed") +
  geom_text(aes(x = as_datetime(half_stars_startpoint),
                label = as_datetime(half_stars_startpoint),
                y = 2.5),
```

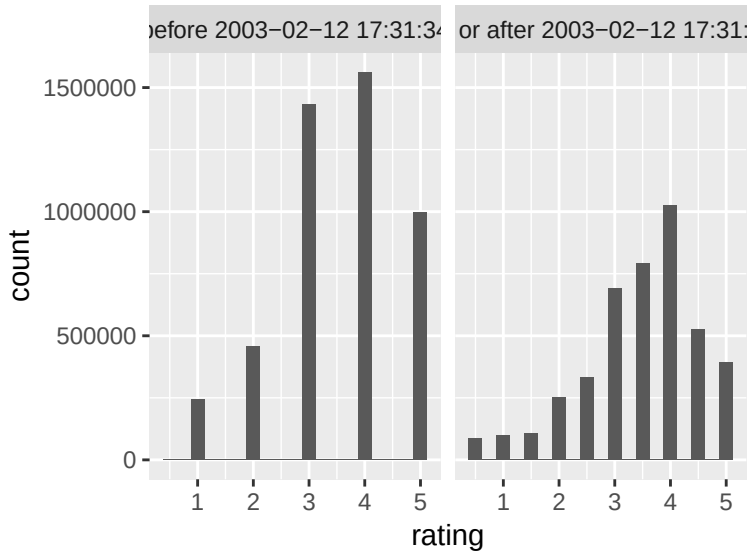
```
            color = "red", vjust = -1, angle = 90) +
  labs(x = 'timestamp', y = 'rating')
```



A clear partition of the dataset can be observed, the first one for ratings before 2003-02-12 17:31:34 where only full-stars were allowed, and a second one for ratings after that point in time where half-stars were allowed.

Let's create another plot about the distribution of ratings in each one of these partitions.

```
partition_names = c(paste('before', as_datetime(half_stars_startpoint)),
                    paste('on or after', as_datetime(half_stars_startpoint)))

edx %>%
  mutate(partition = factor(ifelse(timestamp < half_stars_startpoint,
                                   partition_names[1], partition_names[2]),
                            levels = partition_names)) %>%
  ggplot() +
  geom_histogram(aes(x = rating), binwidth = 0.25) +
  facet_grid(~ partition)
```



Now the distribution of ratings looks very different, it seems that ratings using half-stars were also popular among users when they were allowed (i.e. in the second partition).

Having a point in time to partition the dataset seems to be an important aspect to consider, since it could mean a different users behavior from one partition to another. For example, it might be that a partucular user had a tendency to rate movies with 4 stars before 2003-02-12 17:31:34, but then after that date the same user might have changed its tendency to rate 3.5 instead, since now half-stars were allowed.

3 Methodology

3.1 Representing ML methods/models as objects

To evaluate the different (machine learning) methods tested in this report, an object-oriented approach is used, each method is going to be represented by a model which would be generated through the construction of an object.

To generate a model, the constructor function receives a dataset (the training set) which is used to fit the model an construct its object. The model's object includes a `predict` function used to perform a prediction for another dataset (which could be the one for testing, validation, production, etc.).

For example, the following function creates and object to represent a model that always gives the most common rate (the mode) of the training set as prediction.

```
#' This object-constructor function is used to generate a model that returns
#' a as prediction the most common rating in the dataset used to fit it.
#' @param dataset The dataset used to fit the model
#' @return The model
RModeModel <- function(dataset) {
  model <- list()

  model$ratings <- unique(dataset$rating)
  model$mode <- model$ratings[which.max(tabulate(match(dataset$rating, model$ratings)))]

  #' The prediction function
  #' @param s The dataset used to perform the prediction of
  #' @return A vector containing the prediction for the given dataset
  model$predict <- function(s) {
    model$mode
  }

  model
}
```

Using the constructor function, an object can be created to fit the particular model, p.e:

```
model <- RModeModel(edx)
```

Then this model can be used to make predictions, p.e. the following code makes predictions using the training and the validation sets:

```
training_pred <- model$predict(edx)
validation_pred <- model$predict(validation)
```

And the predictions are helpful to messure the performance of the model in terms of RMSE and/or accuracy, applied to the training and validation sets, p.e:

```r
sprintf("Train-RMSE: %f, Train-Acc: %f, Val-RMSE: %f, Val-Acc: %f",
        RMSE(training_pred, edx$rating),
        mean(training_pred == edx$rating),
        RMSE(validation_pred, validation$rating),
        mean(validation_pred == validation$rating))
```

```
## [1] "Train-RMSE: 1.167044, Train-Acc: 0.287602, Val-RMSE: 1.168016, Val-Acc: 0.287420"
```

3.2 Refining the metodology to use partitioned models

Given the previous observation that the dataset can be significantly different in two partitions (before 2003-02-12 17:31:34 and on-or-after that), it would be convenient to train and predict a particular model in the two separate partitions and then merging the prediction results for the whole set. The following function creates an object in charge of doing that, for a given method it fits a model for each one of the partitions, and it has a prediction function that merges the predictions for the first and second models according to the timestamp.

```r
#' This object-constructor function is used to generate a metamodel
#' that contains two models,
#' one fitted for data before the startpoint when half stars were allowed in the
#' ratings, and the other one fitted for data on or after that startpoint.
#' The predictions are performed by choosing the appropriate model according to the
#' data's timestamp.
#'
#' @param dataset The dataset used to fit both models,
#'    it should contain a column called 'timestamp'
#' @param base_model_generator The function used to generate the base models,
#'    it should receive a dataset to fit the model and have a prediction function
#' @return The created metamodel
PartitionedModel <- function(dataset, base_model_generator) {
  partitioned_model <- list()

  # Spliting the dataset in 2,
  # one set for data before the startpoint when half stars were allowed
  dataset1 <- dataset %>% filter(timestamp < half_stars_startpoint)
  # the other one for the data on or after the startpoint when half stars were allowed
  dataset2 <- dataset %>% filter(timestamp >= half_stars_startpoint)

  # Generating a model for each dataset
  partitioned_model$model1 <- base_model_generator(dataset1)
  partitioned_model$model2 <- base_model_generator(dataset2)

  #' Performs a prediction with the combined fitted models,
  #' it tries to do the prediction with the respective model based on the timestamp.
  #' @param s The dataset used to perform the prediction of
  #' @return A vector containing the prediction for each row of the dataset
  partitioned_model$predict <- function(s) {
    # Performing the predictions on the whole dataset for each one of the models
    pred1 <- partitioned_model$model1$predict(s)
    pred2 <- partitioned_model$model2$predict(s)

    # Selecting the prediction to use according to the data's timestamp.
    s %>%
```

```
      mutate(pred = ifelse(timestamp < half_stars_startpoint, pred1, pred2)) %>%
      .$pred
  }

  partitioned_model
}
```

To meassure the accuracy it also would be convenient to have a function that rounds the predictions to the actual number to represent stars, either full or half. That means that according to the timestamp, before the half-star starpoint only values in $\{1, 2, 3, 4, 5\}$ are allowed, and on or after the startpoint values in $\{1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5\}$ are allowed. The folowing function performs such rounding.

```
#' Converts a prediction (which is a floating point number) to a one used to
#' represent ratings given by stars,
#' i.e. {1, 2, 3, 4, 5} if the timestamp is before the half start startpoint
#' or {1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5} if the timestamp is on or after.
pred2stars <- function(timestamp, pred) {
  # Rounds the prediction either to be full-stars or having a half-star
  # according to the timestamp
  rounded_pred <- ifelse(timestamp < half_stars_startpoint,
                         round(pred),
                         round(pred * 2)/2)

  # Making sure the rating is not smaller that 1 or bigger than 5
  rounded_pred <- ifelse(rounded_pred >= 1, rounded_pred, 1)
  rounded_pred <- ifelse(rounded_pred <= 5, rounded_pred, 5)
}
```

To test a particular method if would be very helpful to have a function that fits the respective model to either to the whole training set or the partitions given by the half-star startpoint, and then meassures the prediction performance against the training and validation sets. The following function does exaclty that and returns the results in a dataset, which can be included as a table in this report.

```
#' This function is used to report the performance of a model in terms of
#' RMSE and Accuracy for the training and validation sets.
#' It evaluates the performance in two modes:
#' 1) using the whole training set to fit the model and
#' 2) partitioning the training set before and on-or-after
#' the startpoint when half stars were allowed.
#' @param  method_name The name of the method to evaluate
#' @param training_set The dataset used to fit the models
#' @param validation_set The dataset used as validation set
#' @param model_generator The constructor function to generate the model
#' @returns A dataset reporting the performance results
get_performance_metrics <- function(method_name, training_set, validation_set,
                                    model_generator) {

  result <- data.frame('METHOD' = character(), 'SET_MODEL' = character(),
                       'TRAIN_TIME' = integer(),
                       'PRED_TRAIN_TIME' = integer(), 'PRED_VAL_TIME' = integer(),
                       'TRAIN_RMSE' = double(), 'TRAIN_ACC' = double(),
                       'VAL_RMSE' = double(), 'VAL_ACC' = double(),
                       stringsAsFactors = FALSE)

  counter <- 0
```

```
  for (is_partitioned in c(FALSE, TRUE)) {
    counter <- counter + 1
    result[counter,] <- list(method_name, NA, NA, NA, NA, NA, NA)

    train_start <- Sys.time() # recording start o the training

    # Chosing the set type: partitioned or whole
    if (is_partitioned) {
      result[counter, 'SET_MODEL'] <- 'partitioned'
      model <- PartitionedModel(training_set, model_generator)
    } else {
      result[counter, 'SET_MODEL'] <- 'whole'
      model <- model_generator(training_set)
    }

    train_end <- Sys.time() # recording the end of the training

    # Recording the time spent on training
    result[counter, 'TRAIN_TIME'] <- train_end - train_start

    for (is_training in c(TRUE, FALSE)) {
      # Chosing the dataset to evaluate
      if (is_training) {
        ds <- training_set
      } else {
        ds <- validation_set
      }

      pred_start <- Sys.time() # recording the start of the prediction

      # Getting the prediction for the chosen dataset
      pred <- model$predict(ds)

      pred_end <- Sys.time() # recording the end of the prediction

      # Recording the time spent on the prediction
      result[counter, ifelse(is_training, 'PRED_TRAIN_TIME', 'PRED_VAL_TIME')] <-
        pred_end - pred_start

      # Calculating the RMSE
      result[counter, ifelse(is_training, 'TRAIN_RMSE', 'VAL_RMSE')] <-
        RMSE(pred, ds$rating)
      # Calculating the accuracy
      result[counter, ifelse(is_training, 'TRAIN_ACC', 'VAL_ACC')] <-
        mean(pred2stars(ds$timestamp, pred) == ds$rating)
    }
  }

  result
}
```

3.3 Putting all together

To get the performance of a specific (machine learning) method, the following command is useful, where `model_generator` is the constructor function of the respective model:

```
get_performance_metrics(edx, validation, model_generator)
```

For example, to get the performance of the previosly defined model that always predict the most common rating of the training set, the following code can be used to retrieve the performance results and then to includes them in two tables, the first one related to the results of the RMSE/accuracy, and the second one related to the time it takes for training/prediction.

```
results_RModeModel <-
    get_performance_metrics('R Mode', edx, validation, RModeModel)

knitr::kable(results_RModeModel %>%
               select('METHOD', 'SET_MODEL',
                      'TRAIN_RMSE', 'TRAIN_ACC', 'VAL_RMSE', 'VAL_ACC'))

knitr::kable(results_RModeModel %>%
               select('METHOD', 'SET_MODEL',
                      'TRAIN_TIME', 'PRED_TRAIN_TIME', 'PRED_VAL_TIME'))
```

METHOD	SET_MODEL	TRAIN_RMSE	TRAIN_ACC	VAL_RMSE	VAL_ACC
R Mode	whole	1.167044	0.2876016	1.16801599486538	0.2874203
R Mode	partitioned	1.167044	0.2876016	1.16801599486538	0.2874203

METHOD	SET_MODEL	TRAIN_TIME	PRED_TRAIN_TIME	PRED_VAL_TIME
R Mode	whole	0.662899	0.0000050	0.0000300
R Mode	partitioned	1.047990	0.9342439	0.1864691

The previously described mechanism is how the differemt methods are going to be tested in this report, i.e. an object-model is generated then the performance's results are calculated and displayed.

The performance is evaluated in two ways for each model, the first one fitting the model with the whole training set, and the second one splitting the training set in two, one partition for observations before 2003-02-12 17:31:34 and the other one for observations on of after that point in time.

The displaying of the performance's results woule be done in two tables (for visual pursposes) per model:

- The first table includes:
  - METHOD: The method's name
  - SET_MODEL: The type of model in regards to the dataset, which can be `whole` meaning the whole dataset is used to fit the model, or `partitioned` meaning that a partition given by the point in time 2003-02-12 17:31:34 is used for the fitting
  - TRAIN_RMSE: The RMSE of the prediction against the groud thuth applied to the training set
  - TRAIN_ACC: The accuracy of the prediction against the groud thuth applied to the training set
  - VAL_RMSE: The RMSE of the prediction against the groud thuth applied to the validation set
  - VAL_ACC: The accuracy of the prediction against the groud thuth applied to the valisation set
- The second table includes:
  - METHOD: The method's name
  - SET_MODEL: As described above

- TRAIN_TIME: The time in seconds it took for the training to complete
- PRED_TRAIN_TIME: The time in seconds it took for the prediction applied to the training set to complete
- PRED_VAL_TIME: The time in seconds it took for the prediction applied to the validation set to complete

4 Methods

4.1 Model based on the average of ratings

Let's first take a simple model that always returns as prediction the average of the ratings observed in the training set.

This model is described by the following equation:

$$r_{u,m} = \mu + \varepsilon_{u,m}$$

where:

- $r_{u,m}$ is the rating given by the user u to the movie m
- μ is the average of the observed ratings
- $\varepsilon_{u,m}$ is the independent error (variability) of the prediction of the rating for the user u to the movie m

The prediction of the rating that an user gives to a movie is just the value of mu, which is described by the formula:

$$\hat{r}_{u,m} = \mu$$

The following is the constructor function in charge of creating the model for this method.

```
#' This object-constructor function is used to generate a model
#' that always returns as prediction the average of the rating in the
#' given dataset used to fit the model.
#' @param dataset The dataset used to fit the model
#' @return The model
RAvgModel <- function(dataset) {
  model <- list()

  # The average of ratings
  model$mu <- mean(dataset$rating)

  #' The prediction function
  #' @param s The dataset used to perform the prediction of
  #' @return A vector containing the prediction
  model$predict <- function(s) {
    model$mu
  }

  model
}
```

The performance for this model is displayed in the following tables.

METHOD	SET_MODEL	TRAIN_RMSE	TRAIN_ACC	VAL_RMSE	VAL_ACC
R Average	whole	1.060331	0.261445	1.06120181029262	0.2619273
R Average	partitioned	1.059362	0.261445	1.06022101747849	0.2619273

METHOD	SET_MODEL	TRAIN_TIME	PRED_TRAIN_TIME	PRED_VAL_TIME
R Average	whole	0.0237150	0.0000100	0.0000150
R Average	partitioned	0.5551729	0.8766851	0.1952541

As a general observation can be seen that the performance in terms of RMSE has been improved in comparison to the model based on the ratings' mode, however the accuracy went worse, either way the RMSE's performance is still bellow the one desired for this project.

4.2 Model based on movie and user effects

This model is based on the one described in https://rafalab.github.io/dsbook/recommendation-systems.html which was motivated by some of the approaches taken by the winners of the Netflix challenges on October 2006.

It would be the equivalent of a linear model with the movie and user as independent variables and the rating as the dependent variable, which could be potrentially fit by the following code.

```
lm(rating ~ as.factor(movieId) + as.factor(userId))
```

However running the above code would take a lot of time and resources, then instead an approximation is done where the effects per user and movie are calculated.

This model is described by the following equation:

$$r_{u,m} = \mu + b_m + b_u + \varepsilon_{u,m}$$

where:

- $r_{u,m}$ is the rating given by the user u to the movie m
- μ is the average of the observed ratings
- b_m is the observed effect for a particular movie m (movie bias)
- b_u is the observed effect for a particular user u (user bias)
- $\varepsilon_{u,m}$ is the independent error (variability) of the prediction of the rating for the user u to the movie m

The movie effect b_m of a movie m is the average of $r_{u,m} - \mu$ for all the users u that rated the movie. It would be expected that good movies have a positive effect (bias), while bad movies have a negative effect (bias).

The user effect b_u of an user u is the average of $r_{u,m} - \mu - b_m$ for all the movies m that the user rated. It would be expected that optimistic users (that can rate well a really bad movie) have a positive effect (bias), while cranky users (that can reate bad a great movei) have a negative effect (bias).

The prediction of the rating that an user gives to a movie is the value described by the formula:

$$\hat{r}_{u,m} = \mu + b_m + b_u$$

The following constructor function creates an object to represent this model.

```
#' This object-constructor function is used to generate a model
#' of the form:
#'   r_u,m = mu + b_m + b_u + E_u,m
#' where:
#'  - 'r_u,m' is the rating given by an user 'u' to a movie 'm'
#'  - 'mu' is the average of all the observed ratings
#'  - 'b_m' is the movie effect (movie bias) of a movie 'm'
#'  - 'b_u' is the user effect (user bias) of an user 'u'
#'  - 'E_u,m' is the error in the prediction.
```

```r
#'
#' @param dataset The dataset used to fit the model
#' @return The model
MovieUserEffectModel <- function(dataset) {
  model <- list()

  # The average of all the ratings in the dataset
  model$mu <- mean(dataset$rating)

  # Getting the movie bias for each movie
  model$movie_info <- dataset %>%
    group_by(movieId) %>%
    summarise(movie_bias = mean(rating - model$mu))

  # Getting the user bias for each user
  model$user_info <- dataset %>%
    left_join(model$movie_info, by = 'movieId') %>%
    group_by(userId) %>%
    summarise(user_bias = mean(rating - movie_bias - model$mu))

  #' The prediction function, it retrieves as prediction:
  #'   mu + b_m + b_u
  #' where:
  #'  - 'mu' is the average of all the observed ratings during training
  #'  - 'b_m' is the movie effect (movie bias) observed during training for a movie 'm'
  #'  -  b_u' is the user effect (user bias) observed during training for an user 'u'
  #'
  #' @param s The dataset used to perform the prediction of
  #' @return A vector containing the prediction
  model$predict <- function(s) {
    s %>%
      left_join(model$movie_info, by = 'movieId') %>%
      left_join(model$user_info, by = 'userId') %>%
      mutate(pred = model$mu +
               ifelse(!is.na(movie_bias), movie_bias, 0) +
               ifelse(!is.na(user_bias), user_bias, 0)) %>%
      .$pred
  }

  model
}
```

The performance for this model is displayed in the following tables.

METHOD	SET_MODEL	TRAIN_RMSE	TRAIN_ACC	VAL_RMSE	VAL_ACC
Movie and User Effect	whole	0.8567039	0.3590048	0.865348824577316	0.3559134
Movie and User Effect	partitioned	0.8524909	0.3615478	0.861984562840588	0.3581904

METHOD	SET_MODEL	TRAIN_TIME	PRED_TRAIN_TIME	PRED_VAL_TIME
Movie and User Effect	whole	6.066686	4.188021	0.4283781
Movie and User Effect	partitioned	7.967980	9.449213	1.2714341

It can be observed that the performance in terms of RMSE and accuracy was improved in comparison to the previous models (based on mode and average). In fact, the values obtained in the RMSEs applied to the validation set (either fitting the whole set or in partitions) are enought to reach the objective of this project.

4.3 Model based on Naive-Bayes applied to the frequency of ratings

Models based on frequencies try to chose one of the existing ratings when doing the predictions, instead of calculating a numeric approximation. This has the purpose of increasing the accuracy rater than minimazing the global error given by metrics like the RSMSE.

This particular model uses a Naive-Bayes approach applied to the frequeancy of the existing ratings per user and movie. The basis of this model is to find the probability that an user gives a specific rating r, denoted $p_u(r)$; and similarly, the probability that a movie is rated as r, denoted $p_m(r)$. These probabilities can be estimated from the training set just by counting the number of observations for each one of the ratings (the rating frequency) either per user or per customer, and then dividing them by the total of observations per user or per customer respectively.

The prediction of the rating that an user gives to a movie is the value described by the following formula, which basically takes the rating which product of the probabilities related to the user and movie is the maximum:

$$\hat{r}_{u,m} = \arg\max_{r \in R} \Big\{ p_u(r) \cdot p_m(r) \Big\}$$

where:

- $\hat{r}_{u,m}$ is the predicted rating given by the user u to the movie m
- R is the set of all posible ratings
- $p_u(r)$ is the probability of the user u to give a rating r
- $p_m(r)$ is the probability that a movie m receives a rating r

The following constructor function creates an object which represents this model.

```
#' This object-constructor function is used to generate a model
#' to estimate the probability that an user gives a rating 'r',
#' and the probability that a movie is given a rating 'r',
#' for each one of the existing ratings.
#' Then using those probabilities to estimate the prediction of the
#' rating that an user gives to a movie, by getting the rating which maximizes
#' the product of those probabilities.
#'
#' @param dataset The dataset used to fit the model
#' @return The model
RFNaiveBayesModel <- function(dataset) {
  model <- list()

  # Getting the set of all the existing ratings
  model$ratings <- sort(unique(dataset$rating))

  # Names of the columns to be used to store the probabilities that
  # an user gives a specific rating, there would be as many columns as
  # existing ratings
  model$rating_movie_cols <- paste('rating_movie', model$ratings, sep = '_')

  # Names of the columns to be used to store the probabilities that
  # a movie is given a specific rating, there would be as many columns as
  # existing ratings
  model$rating_user_cols <- paste('rating_user', model$ratings, sep = '_')
```

```r
# Information of the movies, including the probability for each movie
# to be given each one of the existing ratings
model$movie_info <- dataset %>%
  group_by(movieId, rating) %>%
  summarise(freq = n()) %>%
  spread(rating, freq, sep = '_movie_', fill = 0) %>%
  left_join(dataset %>% group_by(movieId) %>% summarise(num_ratings = n()),
            by = 'movieId') %>%
  group_by(movieId) %>%
  summarise_at(model$rating_movie_cols, funs(sum(.) / num_ratings))

# Information of the user, including the probability for each user
# to give each one of the existing ratings
model$user_info <- dataset %>%
  group_by(userId, rating) %>%
  summarise(freq = n()) %>%
  spread(rating, freq, sep = '_user_', fill = 0) %>%
  left_join(dataset %>% group_by(userId) %>% summarise(num_ratings = n()),
            by = 'userId') %>%
  group_by(userId) %>%
  summarise_at(model$rating_user_cols, funs(sum(.) / num_ratings))

#' The prediction function, it retrieves as prediction the rating 'r'
#' that gives the maximun of the products:
#'    p(r|u) * p(r|m)
#' where:
#'  - 'p(r|u)' is the probability that the user 'u' gives a rating 'r'
#'  - 'p(r|m)' is the probability that the movie 'm' is rated as 'r'
#'
#' @param s The dataset used to perform the prediction of
#' @return A vector containing the prediction
model$predict <- function(s) {
  # Adding p(r|u) and p(r|m) for each rating in each row of the set
  pred_dataset <- s %>%
    left_join(model$movie_info, by = 'movieId') %>%
    left_join(model$user_info, by = 'userId')

  # For missing estimates probabilities the same probability for
  # each rating is assumed
  pred_dataset[is.na(pred_dataset)] <- 1.0 / length(model$ratings)

  # Calculating the maximum of the products p(r|u) * p(r|m)
  # in each row of the dataset
  max_prod <- NULL
  selected_rating <- NULL
  for (i in 1:length(model$ratings)) {
    prod <-
      pred_dataset[[model$rating_movie_cols[i]]] *
      pred_dataset[[model$rating_user_cols[i]]]

    if (i <= 1) {
      selected_rating <- rep(model$ratings[i], nrow(s))
      max_prod <- prod
```

```
      } else {
        selected_rating <- ifelse(prod >= max_prod, model$ratings[i], selected_rating)
        max_prod <- ifelse(prod >= max_prod, prod, max_prod)
      }
    }

    selected_rating
  }

  model
}
```

The performance of this model is displayed in the following tables.

METHOD	SET_MODEL	TRAIN_RMSE	TRAIN_ACC	VAL_RMSE	VAL_ACC
RF Naive-Bayes	whole	0.9972727	0.3791883	1.00334042412932	0.3694334
RF Naive-Bayes	partitioned	0.9916377	0.3880895	0.998154169610961	0.3750204

METHOD	SET_MODEL	TRAIN_TIME	PRED_TRAIN_TIME	PRED_VAL_TIME
RF Naive-Bayes	whole	28.60017	27.56273	4.781722
RF Naive-Bayes	partitioned	21.66756	53.04081	4.975320

It can be observed that the accuracy was increased in comparison to the previously tested models, however the RMSE presents a poorer performance than the one based on the movie and user effets, non of the RMSEs is not enought for the objective of this project.

4.4 RF-Rec Model

This is another model based on the frequency of the ratings aiming to improve the accuracy of the predictions. The model is described in this paper: https://www.researchgate.net/publication/224262836_RF-Rec_Fast_and_Accurate_Computation_of_Recommendations_Based_on_Rating_Frequencies. The behind's idea is to model the tendency of users to provide extreme ratings rater than staying consistent in their ratings (as considered in the customer and user effects model).

Lets define the following functions:

$$f_{user}(u,r) = freq_{user}(u,r) + 1 + 1_{user}(u,r)$$

and:

$$f_{movie}(m,r) = freq_{movie}(m,r) + 1 + 1_{movie}(m,r)$$

where:

- $freq_{user}(u,r)$ is the frequency of the rating r given for the user u
- $freq_{movie}(m,r)$ is the frequency of the rating r given to the movie m
- $1_{user}(u,r)$ is 1 if r corresponds to the given average (rounded to the closest existing rating) of the user u, or 0 otherwise
- $1_{movie}(m,r)$ is 1 if r corresponds to the given average (rounded to the closest existing rating) of the movie m, or 0 otherwise

The prediction of the rating that an user gives to a movie is the value described by the following formula:

$$\hat{r}_{u,m} = \arg\max_{r \in R} \left\{ f_{user}(u,r) \cdot f_{movie}(m,v) \right\}$$

where:

- $\hat{r}_{u,m}$ is the predicted rating given by the user u to the movie m
- R is the set of all posible ratings

The following constructor function creates an object to repesent this model.

```
#' This object-constructor function is used to generate a model
#' based in the RF-Rec schema.
#'
#' @param dataset The dataset used to fit the model
#' @return The model
RFRecModel <- function(dataset) {
  model <- list()

  # Average of all the observed ratings in the dataset
  model$mu <- mean(dataset$rating)

  # Getting the set of all the existing ratings
  model$ratings <- sort(unique(dataset$rating))

  # Names of the columns to be used to store the frequencies of the ratings
  # that an user gives, there would be as many columns as existing ratings
  model$rating_movie_cols <- paste('rating_movie', model$ratings, sep = '_')
  # Names of the columns to be used to store the frequencies of the ratings
  # that a movie is given, there would be as many columns as existing ratings
  model$rating_user_cols <- paste('rating_user', model$ratings, sep = '_')

  # Information of the movies, including the frequency of the received ratings
  # for each rating
  model$movie_info <- dataset %>%
    group_by(movieId, rating) %>%
    summarise(freq = n()) %>%
    spread(rating, freq, sep = '_movie_', fill = 0) %>%
    group_by(movieId) %>%
    summarise_at(model$rating_movie_cols, funs(sum(.))) %>%
    left_join(dataset %>% group_by(movieId) %>% summarise(movie_avg = mean(rating)),
              by = 'movieId')

  # Information of the user, including the frequency of the given ratings
  # for each rating
  model$user_info <- dataset %>%
    group_by(userId, rating) %>%
    summarise(freq = n()) %>%
    spread(rating, freq, sep = '_user_', fill = 0) %>%
    group_by(userId) %>%
    summarise_at(model$rating_user_cols, funs(sum(.))) %>%
    left_join(dataset %>% group_by(userId) %>% summarise(user_avg = mean(rating)),
              by = 'userId')

  #' The prediction function, it retrieves as prediction the rating 'r'
  #' that gives the maximun of the products:
  #'    (freq_user(u, r) + 1 + 1_user(u,r)) * (freq_movie(m, r) + 1 + 1_movie(m,r))
  #' where:
  #'  - 'freq_user(u, r)' is the frequency of the rating 'r' given for the user 'u'
```

```
  #'  - 'freq_movie(m, r)' is the frequency of the rating 'r' given to the movie 'm'
  #'  - '1_user(u,r)' is 1 if 'r' corresponds to the given average
  #'     (rounded to the closest existing rating) of the user 'u', or 0 otherwise
  #'  - '1_movie(m,r)' is 1 if 'r' corresponds to the given average
  #'     (rounded to the closest existing rating) of the movie 'm', or 0 otherwise
  #'
  #' @param s The dataset used to perform the prediction of
  #' @return A vector containing the prediction
  model$predict <- function(s) {
    pred_dataset <- s %>%
      left_join(model$movie_info, by = 'movieId') %>%
      left_join(model$user_info, by = 'userId')

    # In case of missing user or movie averages, using the global average
    pred_dataset$movie_avg[is.na(pred_dataset$movie_avg)] <- model$mu
    pred_dataset$user_avg[is.na(pred_dataset$user_avg)] <- model$mu
    # In case of missing frequencies using 0
    pred_dataset[is.na(pred_dataset)] <- 0

    # Getting the maximum 'r' which maximizes the product
    # (freq_user(u, r) + 1 + 1_user(u,r)) * (freq_movie(m, r) + 1 + 1_movie(m,r))
    # per row in the dataset
    max_prod <- NULL
    selected_rating <- NULL
    for (i in 1:length(model$ratings)) {
      # Calculating the product
      # (freq_user(u, r) + 1 + 1_user(u,r)) * (freq_movie(m, r) + 1 + 1_movie(m,r))
      prod <- (pred_dataset[[model$rating_movie_cols[i]]] + 1 +
               ifelse(pred2stars(s$timestamp, pred_dataset$movie_avg) == model$ratings[i],
                      1, 0)) *
              (pred_dataset[[model$rating_user_cols[i]]] + 1 +
               ifelse(pred2stars(s$timestamp, pred_dataset$user_avg) == model$ratings[i],
                      1, 0))

      if (i <= 1) {
        selected_rating <- rep(model$ratings[i], nrow(s))
        max_prod <- prod
      } else {
        selected_rating <- ifelse(prod > max_prod, model$ratings[i], selected_rating)
        max_prod <- ifelse(prod > max_prod, prod, max_prod)
      }
    }

    selected_rating
  }

  model
}
```

The performance of this model is displayed in the following table.

METHOD	SET_MODEL	TRAIN_RMSE	TRAIN_ACC	VAL_RMSE	VAL_ACC
RF-Rec	whole	0.9944953	0.3784137	1.00039079902893	0.3691434

METHOD	SET_MODEL	TRAIN_RMSE	TRAIN_ACC	VAL_RMSE	VAL_ACC
RF-Rec	partitioned	0.9890047	0.3872291	0.99526980792245	0.3747734

METHOD	SET_MODEL	TRAIN_TIME	PRED_TRAIN_TIME	PRED_VAL_TIME
RF-Rec	whole	6.572248	1.868097	13.16650
RF-Rec	partitioned	6.926100	2.939885	20.77207

It can be observed that both the RMSEs and accuracies are very similar (although slightly worse) to the ones obtained in the Naive-Bayes approach. With higher accuracies that the model based on the movies and users effects, but the RMSEs not enough to reach the goal of this projects.

4.5 Model based on matrix factorization of residuals

Matrix factorization methods consinst in factorizing a matrix of ratings $R_{U\times M}$ – being U the number of users and M the number of movies – as the product of two matrices of lower dimension $P_{U\times K}$ and $Q_{M\times K}$ – being K a given number of factors –.

$$R = PQ^t$$

Matrix factorization methods consist in finding an approximation to such P and Q matrices given a number of factors K. From this perspective these methods also find some factors to classify both the user and the movie. In this way P would indicate the amount associated of each one of the factors per user, and Q similarly indicates the amount of each one of the factors per movie.

Following this idea, the prediction of the rating that a customer u gives to a movie m can be expressed as:

$$\hat{r}_{u,m} = \sum_{k=1}^{K} (P)_{u,k} \cdot (Q)_{m,k}$$

In practice the training observations are used to construct R, but in genaral they can not be completly fill the matrix R, so missing values can be filled with an average value. Some methods like gradient descend can be used to create an approximation of the factorization, fortunatelly the package `recosystem` can create an approximation for us: https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html.

For this project a matrix factorization is applied to the residuals of the model based on the movie and user effect. The residuals would be defined by the following equation, they are assumed to be averaged at zero:

$$\varepsilon_{u,m} = r_{u,m} - \mu - b_m - b_u$$

where:

- $r_{u,m}$ is the rating given by the user u to the movie m
- μ is the average of the observed ratings
- b_m is the observed effect for a particular movie m (movie bias)
- b_u is the observed effect for a particular user u (user bias)
- $\varepsilon_{u,m}$ is residual of the prediction of the rating for the user u to the movie m

Then a matrix factorization is applied to the residuals, the missed residuals from the observations are just filled with zeros. In this way the matrix of residuals $E = (\varepsilon_{u,m})$ involvimg all the users and movies, would be approximated as:

$$E = PQ^t$$

where:

- P is the matrix of factors per user, with size $U \times K$
- Q the matrix of factors per movie, with size $M \times K$
- K is the given number of factors

Following this idea a residual can be expressed as:

$$\varepsilon_{u,m} \approx \sum_{k=1}^{N} (P)_{u,k} \cdot (Q)_{m,k}$$

Then, the prediction of the rating that an user u gives to a movie m would be defined by the formula:

$$\hat{r}_{u,m} = \mu + b_m + b_u + \sum_{k=1}^{N} (P)_{u,k} \cdot (Q)_{m,k}$$

or interpretated as well as:

$$\hat{r}_{u,m} = \mu + b_m + b_u + \sum_{k=1}^{N} p_{u,k} \cdot q_{m,k}$$

where:

- $p_{u,k}$ is the amount of the factor k that the user u has
- $q_{m,k}$ is the amount of the factor k that the movie m has

The following constructor function creates an object to repesent this model, in this case a number of factors K of 30 is used.

```
if(!require(recosystem))
  install.packages("recosystem", repos = "http://cran.us.r-project.org")
library(recosystem)

#' This object-constructor function is used to generate a model
#' of the form:
#'   r_u,m ~ mu + b_m + b_u + sum {k = 1..K} (p_u,k * q_m_k)
#' where:
#'  - 'r_u,m' is the rating given by an user 'u' to a movie 'm'
#'  - 'mu' is the average of all the observed ratings
#'  - 'b_m' is the movie effect (movie bias) of a movie 'm'
#'  - 'b_u' is the user effect (user bias) of an user 'u'
#'  - 'p_u,k' is the amount of the factor 'k' that the user 'u' has
#'  - 'q_m,k' is the amount of the factor 'k' that the movie 'm' has
#'  - 'K' is the number of (latent) factors
#'
#' @param dataset The dataset used to fit the model
#' @return The model
ResidualsMatrixFactorizationModel <- function(dataset) {
  model <- list()

  # The average of all the ratings in the dataset
  model$mu <- mean(dataset$rating)

  # Getting the movie bias for each movie
  model$movie_info <- dataset %>%
    group_by(movieId) %>%
    summarise(movie_bias = mean(rating - model$mu))
```

```r
# Getting the user bias for each user
model$user_info <- dataset %>%
  left_join(model$movie_info, by = 'movieId') %>%
  group_by(userId) %>%
  summarise(user_bias = mean(rating - movie_bias - model$mu))

# Gettint the training set containing the residuals
training_set <- dataset %>%
  left_join(model$movie_info, by = 'movieId') %>%
  left_join(model$user_info, by = 'userId') %>%
  mutate(residual = rating - (model$mu + movie_bias + user_bias))

# Training set to perform the matroix factorization of the residuals
train_data <- data_memory(user_index = training_set$userId,
                          item_index = training_set$movieId,
                          rating = training_set$residual,
                          index1 = T)
# Training a recomender, this is the one that does the matrix factorization
# in this case for 30 factors
model$recommender <- Reco()
model$recommender$train(train_data,
                        opts = c(dim = 30, costp_l2 = 0.1, costq_l2 = 0.1,
                                 lrate = 0.1, niter = 100, nthread = 6,
                                 verbose = F))

#' The prediction function, it retrieves as prediction:
#'   mu + b_m + b_u + sum {k = 1..K} (p_u,k * q_m_k)
#' where:
#'  - 'mu' is the average of all the observed ratings during training
#'  - 'b_m' is the movie effect (movie bias) observed during training for a movie 'm'
#'  -  b_u' is the user effect (user bias) observed during training for an user 'u'
#'  - 'p_u,k' is the quantity of the factor 'k' that the user 'u' has,
#'            calculated by matrix factorization
#'  - 'q_m,k' is the quantity of the factor 'k' that the movie 'm' has,
#'            calculated by matrix factorization
#'  - 'K' is the number of (latent) factors, in this case 30
#'
#' @param s The dataset used to perform the prediction of
#' @return A vector containing the prediction
model$predict <- function(s) {
  # Dataset used to do the prediction, based on the movie and user
  pred_data <- data_memory(user_index = s$userId, item_index = s$movieId, index1 = T)

  # Predicting the residuals
  pred_residuals <- model$recommender$predict(pred_data, out_memory())

  # Prediction based in movie and user effects, plus the prediction of the residual
  s %>%
    left_join(model$movie_info, by = 'movieId') %>%
    left_join(model$user_info, by = 'userId') %>%
    mutate(pred = pred_residuals + model$mu +
                  ifelse(!is.na(movie_bias), movie_bias, 0) +
                  ifelse(!is.na(user_bias), user_bias, 0)) %>%
```

```
      .$pred
  }

  model
}
```

The performance of this model is displayed in the following table.

METHOD	SET_MODEL	TRAIN_RMSE	TRAIN_ACC	VAL_RMSE	VAL_ACC
Res Matrix Factorization	whole	0.7956945	0.3869611	0.822312055133129	0.3773004
Res Matrix Factorization	partitioned	0.8249818	0.3733984	0.841700924342994	0.3677034

METHOD	SET_MODEL	TRAIN_TIME	PRED_TRAIN_TIME	PRED_VAL_TIME
Res Matrix Factorization	whole	1.593571	8.997145	2.217617
Res Matrix Factorization	partitioned	1.626103	31.828144	3.629137

This is the model that presents the best results for both the RMSE and accuarcy of all the methods tested before, if fitting using the whole training set. The obtained RMSEs are good to reach the goal for this project.

5 Results

5.1 RMSE

Let's take a look at the results of the previously tested models. First by analysing the result sorted by RMSE applied to the validation set, which is the metric considered for the success of this project. The RMSEs are sorted in ascending order, i.e. from the best to the worst.

	METHOD	SET_MODEL	VAL_RMSE
11	Res Matrix Factorization	whole	0.822312055133129
12	Res Matrix Factorization	partitioned	0.841700924342994
6	Movie and User Effect	partitioned	0.861984562840588
5	Movie and User Effect	whole	0.865348824577316
10	RF-Rec	partitioned	0.99526980792245
8	RF Naive-Bayes	partitioned	0.998154169610961
9	RF-Rec	whole	1.00039079902893
7	RF Naive-Bayes	whole	1.00334042412932
4	R Average	partitioned	1.06022101747849
3	R Average	whole	1.06120181029262
1	R Mode	whole	1.16801599486538
2	R Mode	partitioned	1.16801599486538

The model that visible performed the best in regards to RMSE is the one base on matrix factorization of the residuals applied to the whole training set `edx`.

It can be observed that the 4 best results corresponds to the two methods that can accomplish the goal for this project, which is an RMSE of 0.87750 or less, these are the ones based on the movie and user effects and matrix factorization of residuals.

In general the partitioned models produce slightly better results in terms of RMSE than their versions applied to the whole set, the exception is the model based on the matrix factorizaions of residuals, which actually seems to fit better in the whole set than in partitions.

5.2 Accuracy

Now let's look at the results in terms of Accuracy, the following are the results sorted by accuracy in descendent order, i.e. from the best to the worst.

	METHOD	SET_MODEL	VAL_ACC
11	Res Matrix Factorization	whole	0.3773004
8	RF Naive-Bayes	partitioned	0.3750204
10	RF-Rec	partitioned	0.3747734
7	RF Naive-Bayes	whole	0.3694334
9	RF-Rec	whole	0.3691434
12	Res Matrix Factorization	partitioned	0.3677034
6	Movie and User Effect	partitioned	0.3581904
5	Movie and User Effect	whole	0.3559134
1	R Mode	whole	0.2874203
2	R Mode	partitioned	0.2874203
3	R Average	whole	0.2619273
4	R Average	partitioned	0.2619273

Again, the model that performed the best now in regards to accuracy is the one base on matrix factorization of the residuals applied to the whole training set `edx`, however the methods based on the frequency of ratings are pretty close.

In general the methods that performed the best in regards to accuracy are the ones based on ratings' frequency, but the differency is small among the first 8 on the table. Anyway, accuracy is not part of the goal of this project, so these results are just for informative purposes.

5.3 Processing time

Finally lets take a look to the following two tables that show the time (in seconds) that takes for the models to perform the training (using the training set `edx`) and then the prediction (applied to the validation set `validation`), sorted in ascending order, i.e. from quickest to slowest.

	METHOD	SET_MODEL	TRAIN_TIME
3	R Average	whole	0.0237150
4	R Average	partitioned	0.5551729
1	R Mode	whole	0.6628990
2	R Mode	partitioned	1.0479901
11	Res Matrix Factorization	whole	1.5935715
12	Res Matrix Factorization	partitioned	1.6261033
5	Movie and User Effect	whole	6.0666859
9	RF-Rec	whole	6.5722480
10	RF-Rec	partitioned	6.9261000
6	Movie and User Effect	partitioned	7.9679801
8	RF Naive-Bayes	partitioned	21.6675551
7	RF Naive-Bayes	whole	28.6001720

	METHOD	SET_MODEL	PRED_VAL_TIME
3	R Average	whole	0.0000150
1	R Mode	whole	0.0000300
2	R Mode	partitioned	0.1864691
4	R Average	partitioned	0.1952541
5	Movie and User Effect	whole	0.4283781
6	Movie and User Effect	partitioned	1.2714341
11	Res Matrix Factorization	whole	2.2176170
12	Res Matrix Factorization	partitioned	3.6291368
7	RF Naive-Bayes	whole	4.7817218
8	RF Naive-Bayes	partitioned	4.9753201
9	RF-Rec	whole	13.1665020
10	RF-Rec	partitioned	20.7720711

The results vary a lot between models, but in general the methods based in ratings' frequency seem to perform the worst, possibly because they need to perform operations to extend the datasets. The methods based on the mode and average are pretty fast because the amount of opperations is minimal, however they are not considered for the purposed of this project.

From these results in terms of processing time we can observe that the methods based on movie/user effects and matrix factorization of residuals are in acceptable times for their use as a solution.

6 Conclusion