Report

Egar Garcia

2/11/2019

Overview

Methodology

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:

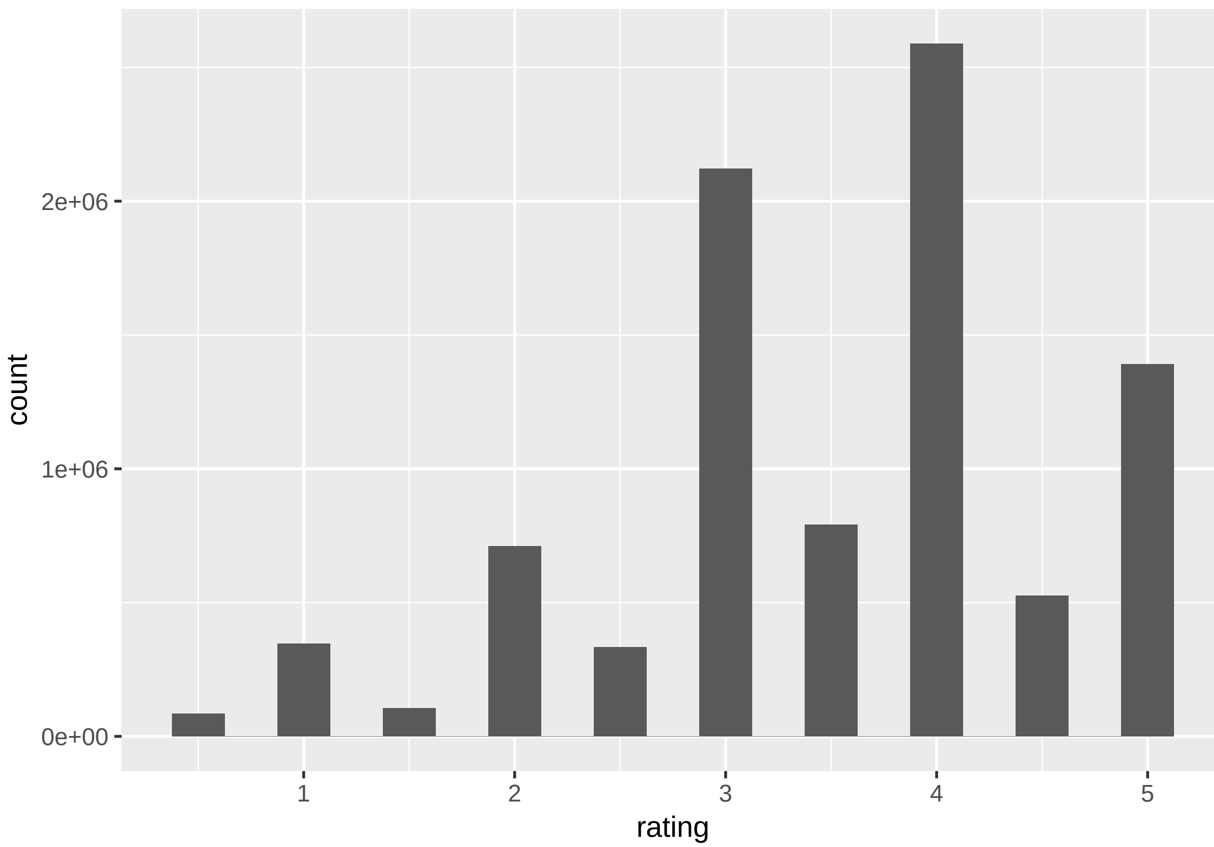
```r
sprintf("Train-RMSE: %f, Train-Acc: %f, Val-RMSE: %f, Val-Acc: %f",
        RMSE(training_pred, edx$rating),
        mean(training_pred == edx$rating),
        RMSE(validation_pred, validation$rating),
        mean(validation_pred == validation$rating))
```

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

Analysing the data

Let's take an initial look at how the ratings are distributed, by plotting an histogram to account the amount of the different ratings given by the customers:

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



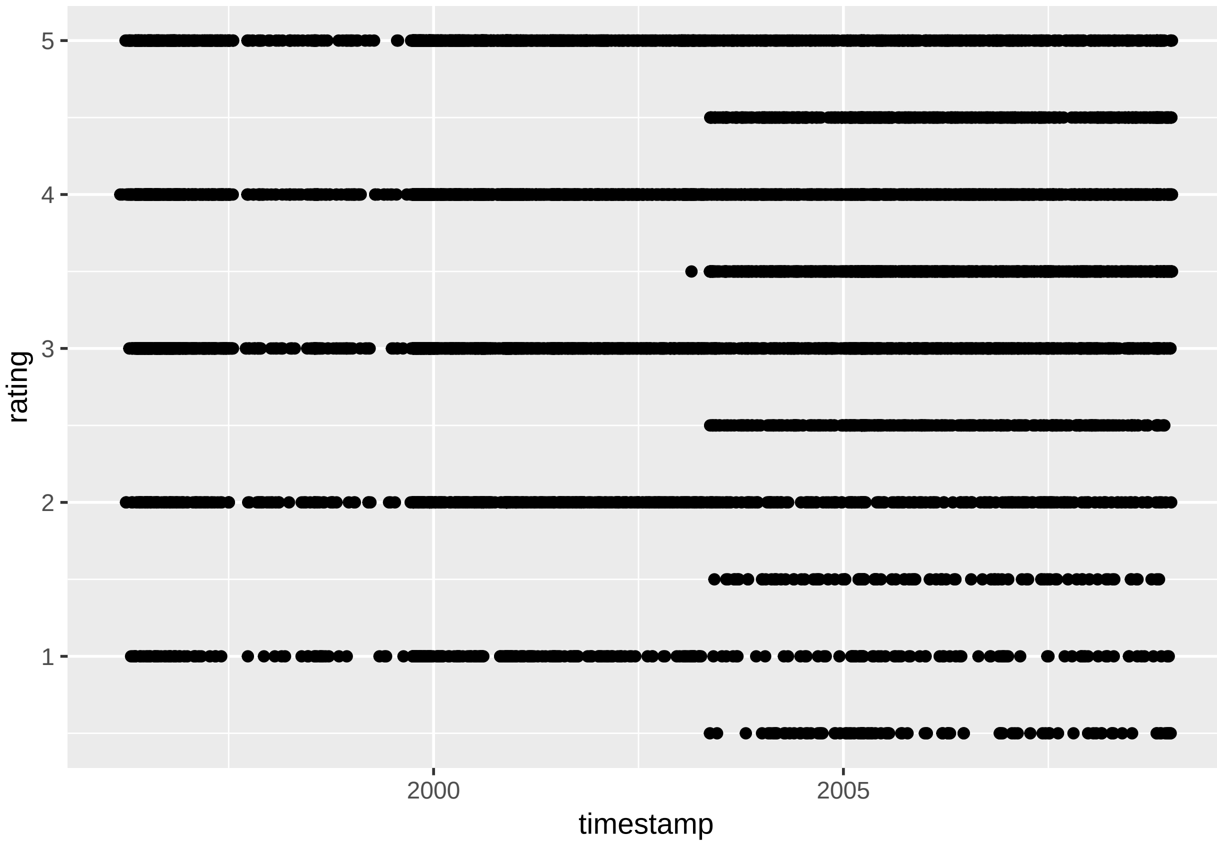
It can be observed that ratings can be interpreted as the number of starts from 1 to 5 that users give to a movie, however it can be seen that ratings ending with a half are used, at first impression looks like half start ratings are not very popular among users.

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.

```r
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 allowed after certain point in time, and before that time just full-stars were allowed.

Identifying the point in time to partition 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)
```

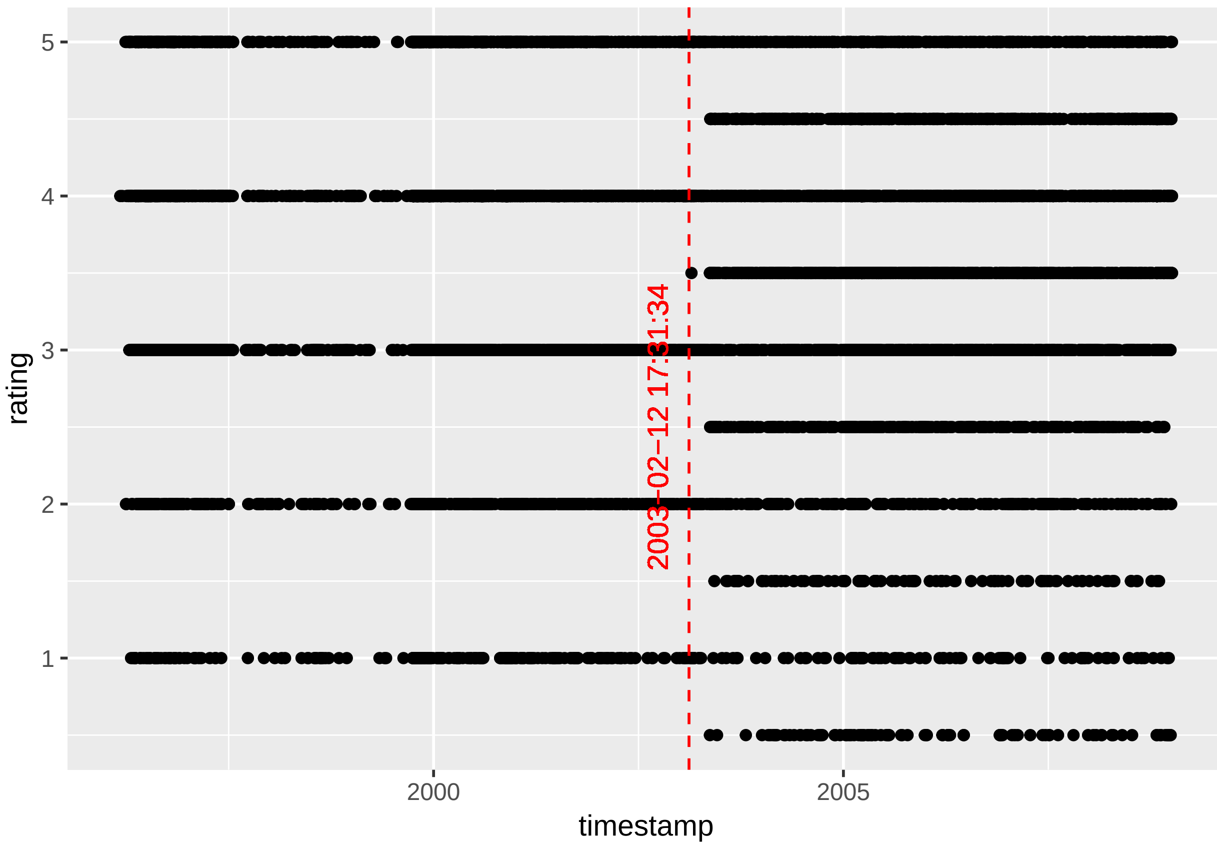
Converting `half_stars_startpoint` to a more redable representation using the following code, it can be seen that the point in time when half-star ratings started to appear was on **2003-02-12 17:31:34**.:

```
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')
```

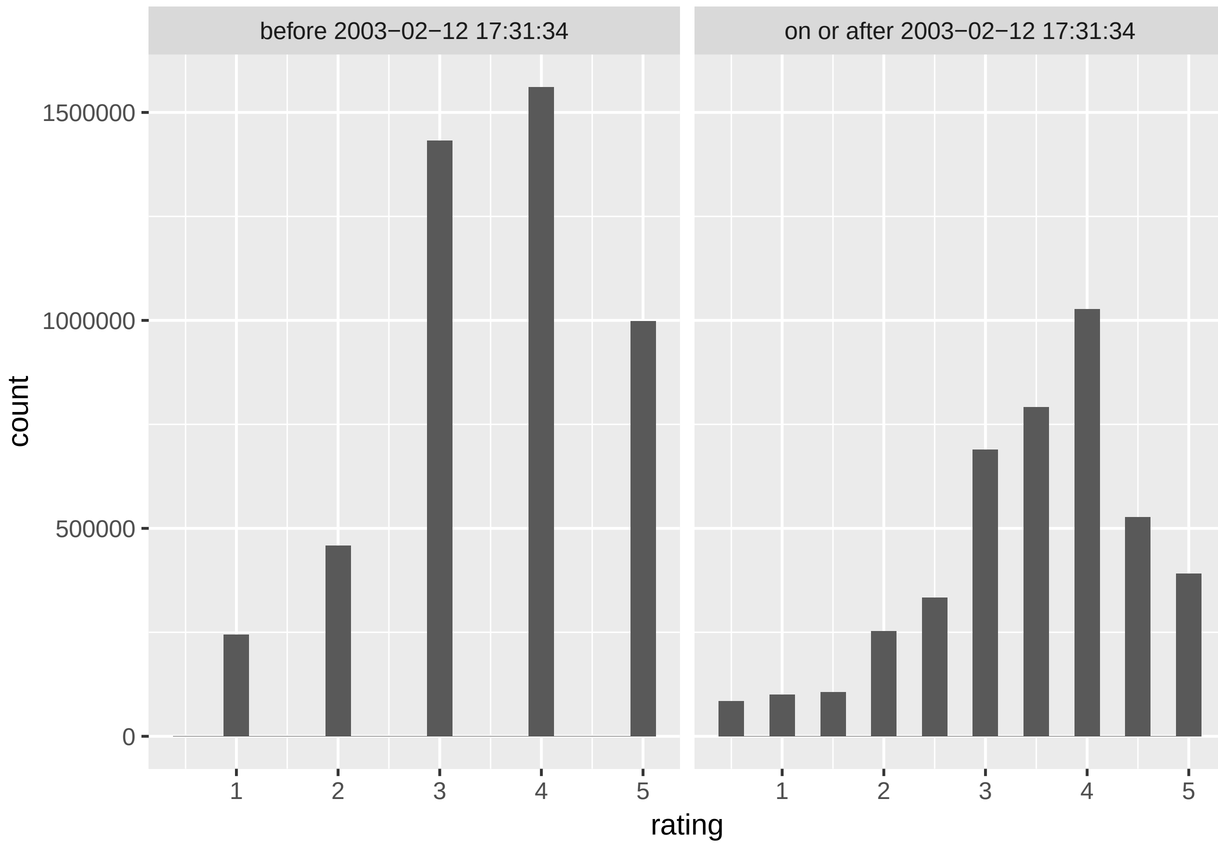


A clear partition of the dataset can be observed, the first one for ratings before 2003-02-12 17:31:34 where only full-stars were allowed, and a second one for ratings after that point in time where half-stars were allowed.

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.

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:

```
#' 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,
```

```r
#'    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:

```r
#' 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:

```r
#' 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 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(training_set, validation_set, model_generator) {
  dataset_names <- c('Training', 'Validation')
  datasets <- c()
  modes <- c()
  rmses <- c()
  accuracies <- c()
  counter <- 0

  for (is_partitioned in c(FALSE, TRUE)) {
    # Chosing the mode PARTITIONED or WHOLE
    if (is_partitioned) {
      model <- PartitionedModel(training_set, model_generator)
    } else {
      model <- model_generator(training_set)
    }

    for (dataset_name in dataset_names) {
      # Chosing the dataset to evaluate
      if (dataset_name == 'Training') {
        ds <- training_set
      } else {
        ds <- validation_set
      }

      counter <- counter + 1

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

      datasets[counter] <- dataset_name
      modes[counter] <- ifelse(is_partitioned, 'PARTITIONED', 'WHOLE')

      # Calculating the RMSE
      rmses[counter] <- RMSE(pred, ds$rating)
      # Calculating the accuracy
      accuracies[counter] <- mean(pred2stars(ds$timestamp, pred) == ds$rating)
    }
  }

  data.frame('Dataset' = datasets,
```

```
              'Mode' = modes,
              'RMSE' = rmses,
              'Accuracy' = accuracies)
}
```

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(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 command calculates the prediction performance and includes the results as a table in this report:

```
knitr::kable(get_performance_metrics(edx, validation, RModeModel),
             caption = "Results for the model based on the rating mode")
```

Table 1: Results for the model based on the rating mode

Dataset	Mode	RMSE	Accuracy
Training	WHOLE	1.167044	0.2876016
Validation	WHOLE	1.168016	0.2874203
Training	PARTITIONED	1.167044	0.2876016
Validation	PARTITIONED	1.168016	0.2874203

Methods

Model based on the average of ratings

Let's start experimenting with different models by first taking 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:

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

Where:

- $Y_{u,m}$ 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:

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

The following constructor function creates objects that represent this model:

```
#' 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.
```

```r
#' @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 table. It can be observed that the performance in terms of RMSE was improved in comparison to the mode model, but the accuracy went worse, either way the performance is still bellow the one desired in this project.

Table 2: Results for the average based model

Dataset	Mode	RMSE	Accuracy
Training	WHOLE	1.060331	0.2614450
Validation	WHOLE	1.061202	0.2619273
Training	PARTITIONED	1.059362	0.2614450
Validation	PARTITIONED	1.060221	0.2619273

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:

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

However running the previous 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:

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

Where:

- $Y_{u,m}$ 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 $Y_{u,m} - \mu$ 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 $Y_{u,m} - \mu - b_m$ 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:

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

The following constructor function creates an object for representing this model:

```r
#' This object-constructor function is used to generate a model
#' of the form:
#'   Y_u,m = mu + b_m + b_u + E_u,m
#'
#' Where 'Y_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',
#' and 'E_u,m' is the error in the prediction.
#'
#' @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 per movie
  model$movie_info <- dataset %>%
    group_by(movieId) %>%
    summarise(movie_bias = mean(rating - model$mu))

  # Getting the user bias per 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:
  #'   Y_u,m = 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',
  #' and '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 +
```

```r
                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 table. It can be observed that the performance in terms of RMSE and accuracy was improved in comparison to the previous models. In fact, the values obtained in the RMSEs applied to the validation set are enought to reach the objective of this project.

Table 3: Results for the model based on movie and user effects

Dataset	Mode	RMSE	Accuracy
Training	WHOLE	0.8567039	0.3590048
Validation	WHOLE	0.8653488	0.3559134
Training	PARTITIONED	0.8524909	0.3615478
Validation	PARTITIONED	0.8619846	0.3581904

Naive Bayes Model

RF-Rec Model

https://www.researchgate.net/publication/224262836_RF-Rec_Fast_and_Accurate_Computation_of_Recommendations_Based_on_Rating_Frequencies

Matrix Factorization

https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html