# Capstone Movielens Report

Azamat Kurbanaev

2026-02-25

## Contents

# 1 Overview and Executive Summary

The goal of the project is to build a Recommendation System using a 10M version of the MovieLens dataset. Following the Netflix Grand Prize Contest requirements, we will evaluate the *Root Mean Square Error* (hereafter *RMSE* for short) score, which, as shown in Section 24.1 *Case study: recommendation systems / Loss function* of the *Course Textbook (New Edition)*, is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i,j}^{N}(y_{i,j} - \hat{y}_{i,j})^2}$$

with $N$ being the number of user/movie combinations for which we make predictions and the sum occurring over all these combinations[1].

Our goal is to achieve a value of less than 0.86490 (compare with the *Netflix Grand Prize* requirement: of at least 0.8563[2]).

## 1.1 Datasets Overview

To start with we have to generate two datasets derived from the *MovieLens* dataset mentioned above:

- **edx:** we use it to develop and train our algorithms;
- **final_holdout_test:** according to the course requirements, we use it exclusively to evaluate the ***RMSE*** of our final algorithm.

For this purpose, a dedicated package, edx.capstone.movielens.data, has been developed by the author of *this Report*. The source code of the package is available on GitHub (the main `R` script that prepares the necessary datasets is located in the file init-movielens-data.R)[3].

We install the development version of this package from the *GitHub repository* using the following code snippet:

```
## Load Source Datasets from Specially Designed Package ------------------------
if(!require(edx.capstone.movielens.data)) {
  start <- put_start_date()
  pak::pak("AzKurban-edX-DS/edx.capstone.movielens.data")
  put_end_date(start)
}

put_log("Dataset loaded from `edx.capstone.movielens.data` package: edx")
```

In the code snippet above, we use the user-defined helper function put_log described in section Logging Functions of Appendix A.

Next, we attach the correspondent library to the global environment using the following line of code:

```
library(edx.capstone.movielens.data)
```

The line of code above is used in the init_source_datasets user-defined helper function described in section Initializing Input Datasets of Appendix A.

Finally, we have the datasets listed above:

```
summary(edx)
```

```
##      userId         movieId          rating        timestamp           title              genres
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:9000055     Length:900005
##  1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08   Class :character   Class :chara
##  Median :35738   Median : 1834   Median :4.000   Median :1.035e+09   Mode  :character   Mode  :chara
##  Mean   :35870   Mean   : 4122   Mean   :3.512   Mean   :1.033e+09
##  3rd Qu.:53607   3rd Qu.: 3626   3rd Qu.:4.000   3rd Qu.:1.127e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
```

```
summary(final_holdout_test)
```

```
##      userId         movieId          rating        timestamp           title              genres
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:999999      Length:999999
##  1st Qu.:18096   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.467e+08   Class :character   Class :chara
##  Median :35768   Median : 1827   Median :4.000   Median :1.035e+09   Mode  :character   Mode  :chara
##  Mean   :35870   Mean   : 4108   Mean   :3.512   Mean   :1.033e+09
##  3rd Qu.:53621   3rd Qu.: 3624   3rd Qu.:4.000   3rd Qu.:1.127e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
```

### 1.1.1 `edx` Dataset

Let's look into the details of the `edx` dataset:

```
str(edx)
```

```
## 'data.frame':    9000055 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : int  122 185 292 316 329 355 356 362 364 370 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983525 838983421 838983392 838983392 838984474 838983653 838984885 83
##  $ title    : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|A
```

Note that we have 9000055 rows and six columns in there:

```
dim_edx <- dim(edx)
print(dim_edx)
```

```
## [1] 9000055       6
```

First, let's note that we have 10677 different movies:

```
n_movies <- n_distinct(edx$movieId)
print(n_movies)
```

```
## [1] 10677
```

and 69878 different users in the dataset:

```
n_users <- n_distinct(edx$userId)
print(n_users)
```

```
## [1] 69878
```

Now, note the expressions below which confirm the fact explained in Section 33.7.1 *Movielens data* of the *Course Textbook (First Edition)* that not every user rated every movie[4]:

```
max_possible_ratings <- n_movies*n_users
sprintf("Maximum possible ratings: %s", max_possible_ratings)
```

```
## [1] "Maximum possible ratings: 746087406"
```

```
sprintf("Rows in `edx` dataset: %s", dim_edx[1])
```

```
## [1] "Rows in ‘edx‘ dataset: 9000055"
```

```
sprintf("Not every movie was rated: %s", max_possible_ratings > dim_edx[1])
```

```
## [1] "Not every movie was rated: TRUE"
```

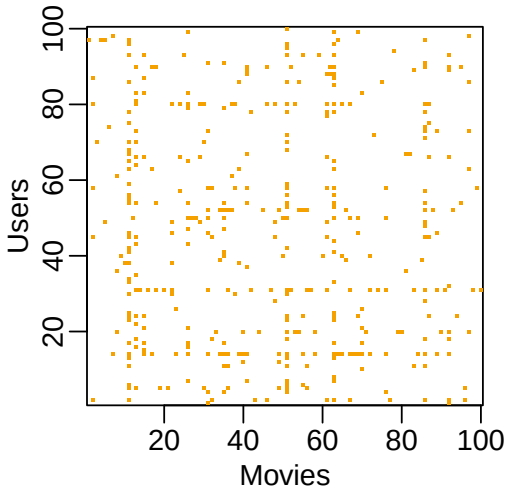
As also explained in Section 33.7.1 of the *Textbook*, we can think of these data as a very large matrix, with users on the rows and movies on the columns, with many empty cells. Therefore, we can think of a recommendation system as filling in the `NA`s in the dataset for the movies that some or all the users do not rate. A sample from the `edx` data below illustrates this idea[4]:

```
keep <- edx |>
  dplyr::count(movieId) |>
  top_n(4, n) |>
  pull(movieId)

tab <- edx |>
  filter(movieId %in% keep) |>
  filter(userId %in% c(13:20)) |>
  select(userId, title, rating) |>
  mutate(title = str_remove(title, ", The"),
         title = str_remove(title, ":.*")) |>
  pivot_wider(names_from = "title", values_from = "rating")

print(tab)
```

```
## # A tibble: 5 x 5
##   userId `Pulp Fiction (1994)` `Jurassic Park (1993)` `Silence of the Lambs (1991)` `Forrest Gump (19
##    <int>                 <dbl>                  <dbl>                         <dbl>                 
## 1     13                     4                     NA                            NA                 
## 2     16                    NA                      3                            NA                 
## 3     17                    NA                     NA                             5                 
## 4     18                     5                      3                             5                 
## 5     19                    NA                      1                            NA                 
```

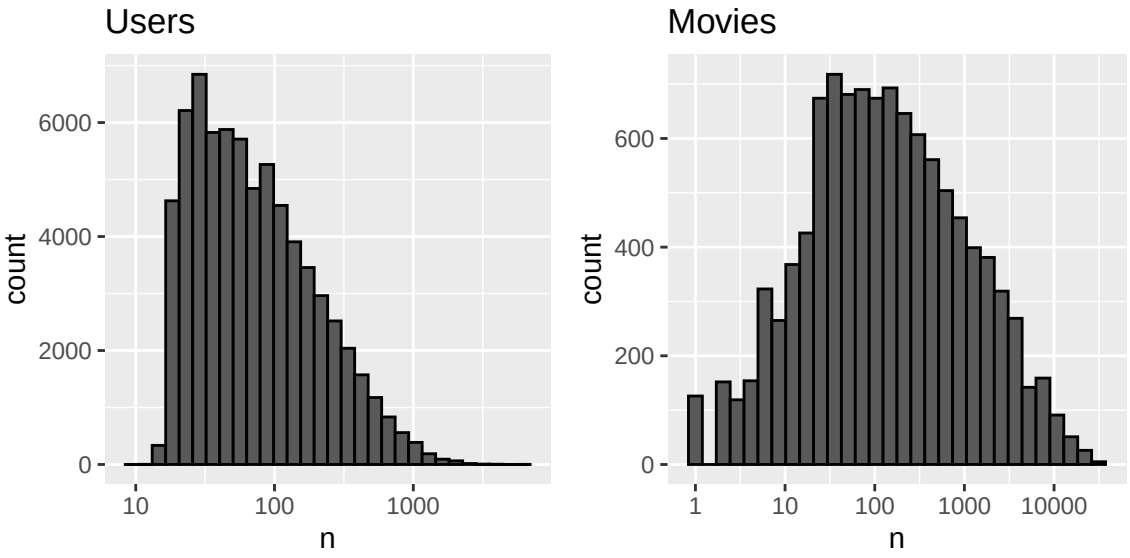
The following plot of the matrix for a random sample of 100 movies and 100 users with yellow indicating a user/movie combination for which we have a rating shows how *sparse* the matrix is[4]:

```
users <- sample(unique(edx$userId), 100)

rafalib::mypar()
edx|>
  filter(userId %in% users) |>
  select(userId, movieId, rating) |>
  mutate(rating = 1) |>
  pivot_wider(names_from = movieId, values_from = rating) |>
  (\(mat) mat[, sample(ncol(mat), 100)])() |>
  as.matrix() |>
  t() |>
  image(1:100, 1:100, z = _ , xlab = "Movies", ylab = "Users")
```




Further observations highlighted there that, as we can see from the distributions the author presented, some movies get rated more than others, and some users are more active than others in rating movies[4]:

```
p1 <- edx |>
  count(movieId) |>
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  ggtitle("Movies")

p2 <- edx |>
  count(userId) |>
  ggplot(aes(n)) +
  geom_histogram(bins = 30, color = "black") +
  scale_x_log10() +
  ggtitle("Users")

gridExtra::grid.arrange(p2, p1, ncol = 2)
```



Finally, the following code snippet demonstrates that no movies have a rating of 0. Movies are rated from 0.5 to 5.0 in 0.5 increments:

```
#library(dplyr)
s <- edx |> group_by(rating) |>
  summarise(n = n())
print(s)
```

```
## # A tibble: 10 x 2
##    rating       n
##     <dbl>   <int>
##  1    0.5   85374
##  2    1    345679
##  3    1.5  106426
##  4    2    711422
##  5    2.5  333010
##  6    3   2121240
##  7    3.5  791624
##  8    4   2588430
##  9    4.5  526736
## 10    5   1390114
```

# 2 Methods and Analysis

All the source code of the `R` scripts utilized in this project is available on the project's GitHub repository[5].

## 2.1 Preparing train and set datasets

We will split the `edx` dataset into a training set, which we will use to build and train our models, and a test set in which we will compute the accuracy of our predictions, the way described in Section 24.1 *Case study: recommendation systems* of the *Course Textbook (New Edition)*[6].We will also use the *Cross-Validation* method as described in Section 29.6 *Cross validation* of the *Course Textbook (New Edition)*[7].

To prepare datasets for processing, we will use the following functions (described in detail in section Initializing Input Datasets of Appendix A), specifically designed for these operations:

- make_source_datasets;
- init_source_datasets.

As a result of executing these functions, we will obtain the following datasets (detailed in Appendix B: Models Training Datasets), which we will use throughout *this Project*:

- `edx.mx` Matrix Object: The `edx` dataset converted to a matrix;
- `edx.sgr` Object: A dataset derived from the `edx` by splitting each row corresponding to a movie belonging to multiple genres;
- `movie_map` Object: A map establishing a correspondence between a movie ID and its title.;
- `date_days_map` Object: A map between the rating date and the number of days since the earliest entry in the edx dataset;
- `edx_CV` Object: A list of objects created from samples made from the `edx` dataset to use them for the *K-Fold Cross-Validation* in the way explained in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* we use in *this Project* is determined by the length of the list composing the `edx_CV` Object. In our case, the length of the `edx_CV` is 5, meaning we use *5-Fold Cross-Validation*.

## 2.2 Overall Mean Rating (OMR) Model

Let's begin our analysis by evaluating the simplest model described in Section 33.7.4: *A first model* of the *Course Textbook (First Edition)*[9], and then gradually refine it through further research.

### 2.2.1 Mathematical Description of the OMR Model

The *Overal Mean Rating* model that assumes the same rating for all movies and users with all the differences explained by random variation would look as follows:

$$Y_{i,j} = \mu + \varepsilon_{i,j}$$

with $\varepsilon_{i,j}$ independent errors sampled from the same distribution centered at 0 and $\mu$ is the average rating for all movies.

### 2.2.2 OMR Model Building

The complete source code of the *Overall Mean Rating* computation described in this section is available in the Overall Mean Rating (Naive) Model section of the capstone-movielens.main.R script on *GitHub*.

We know that the estimate that minimizes the RMSE is the least squares estimate of $\mu$ and, in this case, is the average of all ratings. We calculate this using the following line of code:

```
mu <- mean(edx$rating)
mu
```

```
## [1] 3.512465
```

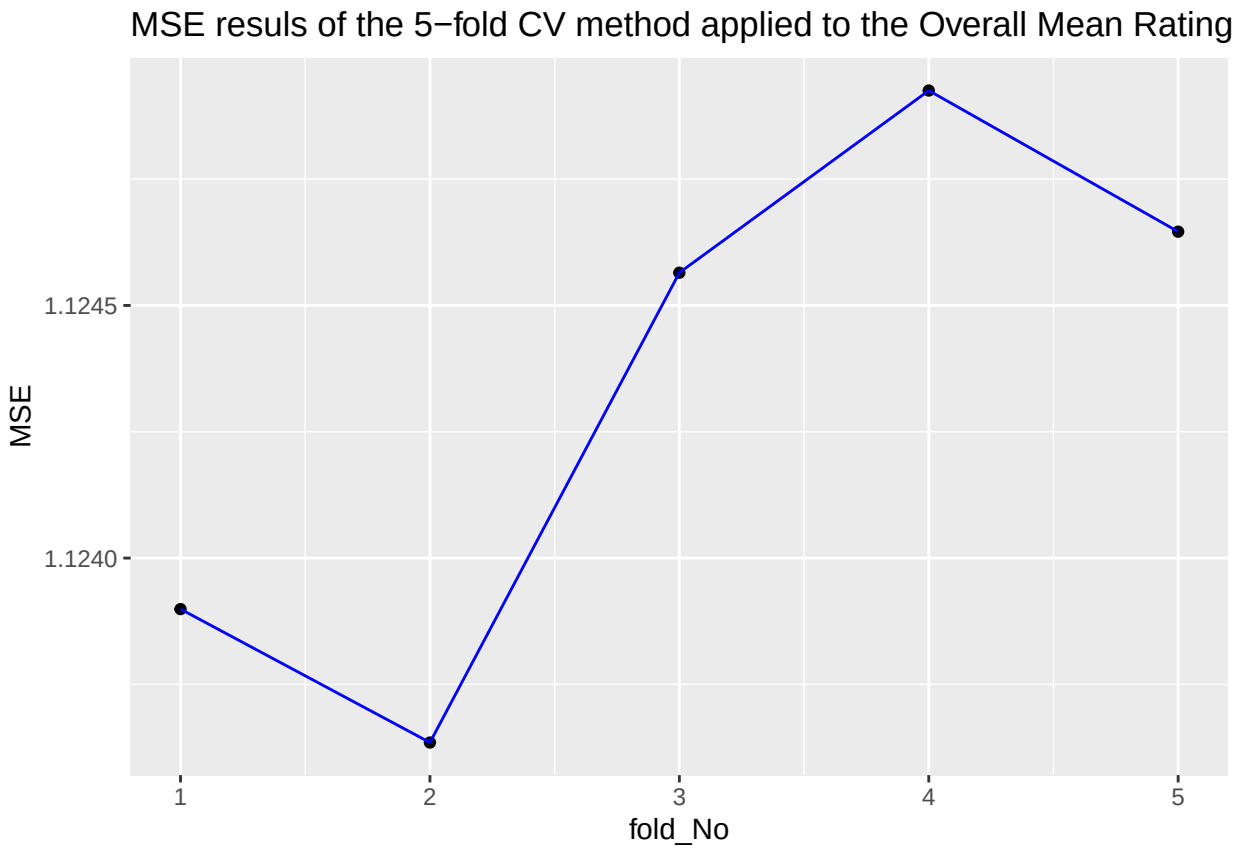
If we predict all unknown ratings with $\hat{\mu}$, we can use the following line of code to compute *Mean Squared Errors* (hereafter *MSE* for short) with the use of *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$):

```
mu.MSEs <- naive_model_MSEs(mu)
```

In the code snippet above, we use the user-defined function naive_model_MSEs described in section OMR Model: Helper Functions of Appendix A.

The following code snippet provides a visual representation of the results we have just obtained:

```
data.frame(fold_No = 1:5, MSE = mu.MSEs) |>
  data.plot(title = "MSE resuls of the 5-fold CV method applied to the Overall Mean Rating Model",
            xname = "fold_No",
            yname = "MSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

We can now calculate the *Root Mean Squared Error (RMSE)* for the current model as a *square root* of the average of the *MSE* values we obtained above, using the following line of code:

```
mu.RMSE <- sqrt(mean(mu.MSEs))
mu.RMSE
```

```
## [1] 1.060346
```

### 2.2.3 OMR Value Is the Best for the Current Model

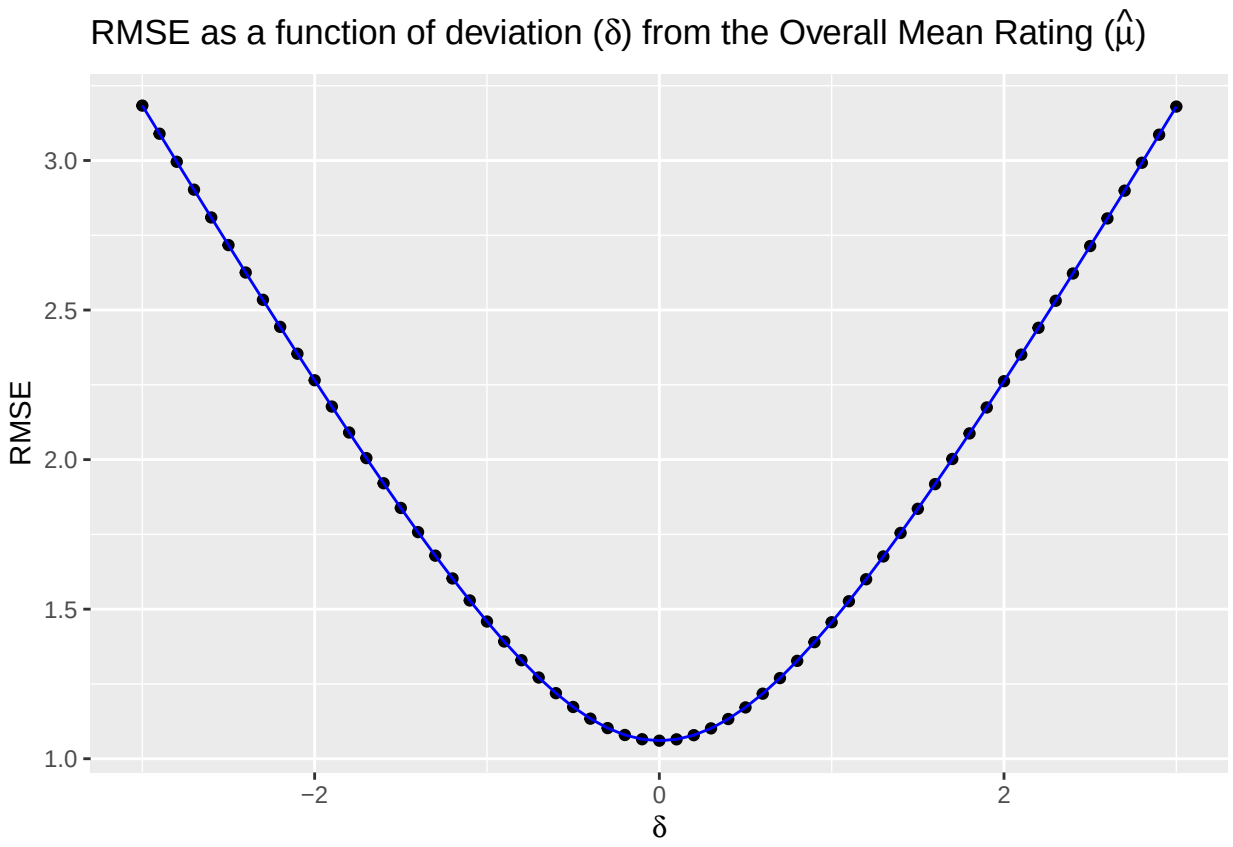
If we plug in any other number, we will get a higher *RMSE*. Let's prove this with a small experiment using the following code snippet:

```
deviation <- seq(0, 6, 0.1) - 3

deviation.RMSE <- sapply(deviation, function(delta){
  naive_model_RMSE(mu + delta)
})
```

In the code snippet above, we use the function naive_model_RMSE described in section OMR Model: Helper Functions of Appendix A to calculate *RMSEs* for the sequence of the deviation values.

Let's quickly analyze the resulting `deviation.RMSE` vector we have just got. The following data visualization code will help us with this:

```
data.frame(delta = deviation,
           delta.RMSE = deviation.RMSE) |>
data.plot(title = TeX(r'[RMSE as a function of deviation ($\delta$) from the Overall Mean Rating ($\hat
              xname = "delta",
              yname = "delta.RMSE",
              xlabel = TeX(r'[$\delta$]'),
              ylabel = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

The piece of code below prints a quick recap of the results:

```
which_min_deviation <- deviation[which.min(deviation.RMSE)]
min_rmse = min(deviation.RMSE)

print_log1("Minimum RMSE is achieved when the deviation from the mean is: %1",
         which_min_deviation)
```

```
## Minimum RMSE is achieved when the deviation from the mean is: 0
```

```
print_log1("Is the previously computed RMSE the best for the current model: %1",
         mu.RMSE == min_rmse)
```

```
## Is the previously computed RMSE the best for the current model: TRUE
```

To print the log, we use the function print_log1 described in section Logging Functions of Appendix A.

Finally, we add the *RMSE* value calculated for the *OMR Model* to our *Result Table* and print the table using the following piece of code:

```
RMSEs.ResultTibble.OMR <- RMSEs.ResultTibble |>
  RMSEs.AddRow("OMR Model",
               mu.RMSE,
               comment = "Overall Mean Rating (OMR) Model")
```

```
RMSE_kable(RMSEs.ResultTibble.OMR)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.864900 | |
| OMR Model | 1.060346 | Overall Mean Rating (OMR) Model |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

To win the grand prize of $1,000,000, a participating team had to get an RMSE of at least 0.8563[2]. So we can definitely do better![9]

## 2.3 User Effect (UE) Model

### 2.3.1 User Effect Analysis

To improve our model let's now take into consideration user effects as explained in Section 24.1 *Case study: recommendation systems / User effects* of the *Course Textbook (New Edition)*[10].

Let's start our analysis by calculating the average rating for each user using the following code snippet:

```
  put_log("Computing Average Ratings per User (User Mean Ratings)...")
  user.mean_ratings <- rowMeans(edx.mx, na.rm = TRUE)
  user_ratings.n <- rowSums(!is.na(edx.mx))

  edx.user_mean_ratings <-
    data.frame(userId = names(user.mean_ratings),
               mean_rating = user.mean_ratings,
               n = user_ratings.n)

  put_log("User Mean Ratings have been computed.")
```

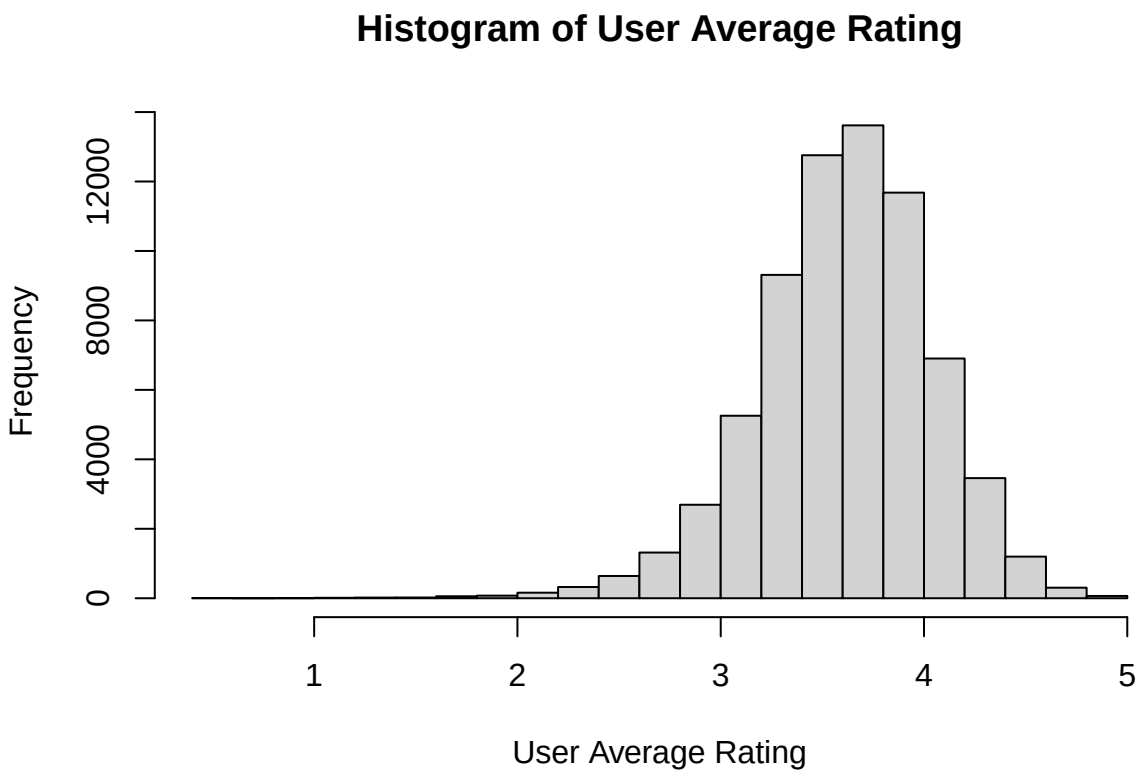
```
str(edx.user_mean_ratings)
```

```
## 'data.frame':    69878 obs. of  3 variables:
##  $ userId     : chr  "1" "2" "3" "4" ...
##  $ mean_rating: num  5 3.29 3.94 4.06 3.92 ...
##  $ n          : num  19 17 31 35 74 39 96 727 21 112 ...
```

In the code snippet above, we use the `edx.mx` Matrix Object (described in detail in Appendix B: Models Training Datasets), which, in fact, is the `edx` dataset converted to a matrix as described earlier in section Preparing train and set datasets.
We also use the user-defined helper function put_log described in section Logging Functions of Appendix A.

The code snippet below visualizes the frequency of the users' average ratings. As the author of the *Course Textbook* shows, such a visualization allows us to see that there is substantial variability in the average ratings across users[10]:

```
hist(edx.user_mean_ratings$mean_rating,
     xlab = "User Average Rating",
     main = "Histogram of User Average Rating",
     nclass = 30)
```


Histogram of User Average Rating
Frequency
0
4000
8000
12000
1
2
3
4
5
User Average Rating

### 2.3.2 Mathematical Description of the UE Model

Following the further explanation provided in Section 24.1 of the *Course Textbook* (mentioned above in the previous Section), to account for this variability, we use a linear model with a *treatment effect* $\alpha_i$ for each user. The sum $\mu + \alpha_i$ can be interpreted as the typical rating user $i$ gives to movies. So we write the model as follows:

$$Y_{i,j} = \mu + \alpha_i + \varepsilon_{i,j}$$

Statistics textbooks refer to the $\alpha$s as treatment effects. In the Netflix challenge papers, they refer to them as *bias*[10, 11].

### 2.3.3 UE Model Building

The complete source code of the *User Effect* computation described in this section is available in the User Effect (UE) Model section of the capstone-movielens.main.R script on *GitHub*.

As it is stated in Section 33.7.6: *User effects* of the *Course Textbook (First Edition)*, it can be shown that the least squares estimate $\hat{\alpha}_i$ is just the average of $y_{i,j} - \hat{\mu}$ for each user $i$. So we can compute them this way[12]:

```
a <- rowMeans(y - mu, na.rm = TRUE)
```

These considerations allow us to compute a *User Mean Ratings* using the following code snippet:

```
put_log("Computing Average Ratings per User (User Mean Ratings)...")
user.mean_ratings <- rowMeans(edx.mx, na.rm = TRUE)
user_ratings.n <- rowSums(!is.na(edx.mx))

edx.user_mean_ratings <-
  data.frame(userId = names(user.mean_ratings),
             mean_rating = user.mean_ratings,
             n = user_ratings.n)

put_log("User Mean Ratings have been computed.")
```

```
str(edx.user_mean_ratings)
```

```
## 'data.frame':    69878 obs. of  3 variables:
##  $ userId     : chr  "1" "2" "3" "4" ...
##  $ mean_rating: num  5 3.29 3.94 4.06 3.92 ...
##  $ n          : num  19 17 31 35 74 39 96 727 21 112 ...
```

And then we compute a *User Effect* using the following piece of code:

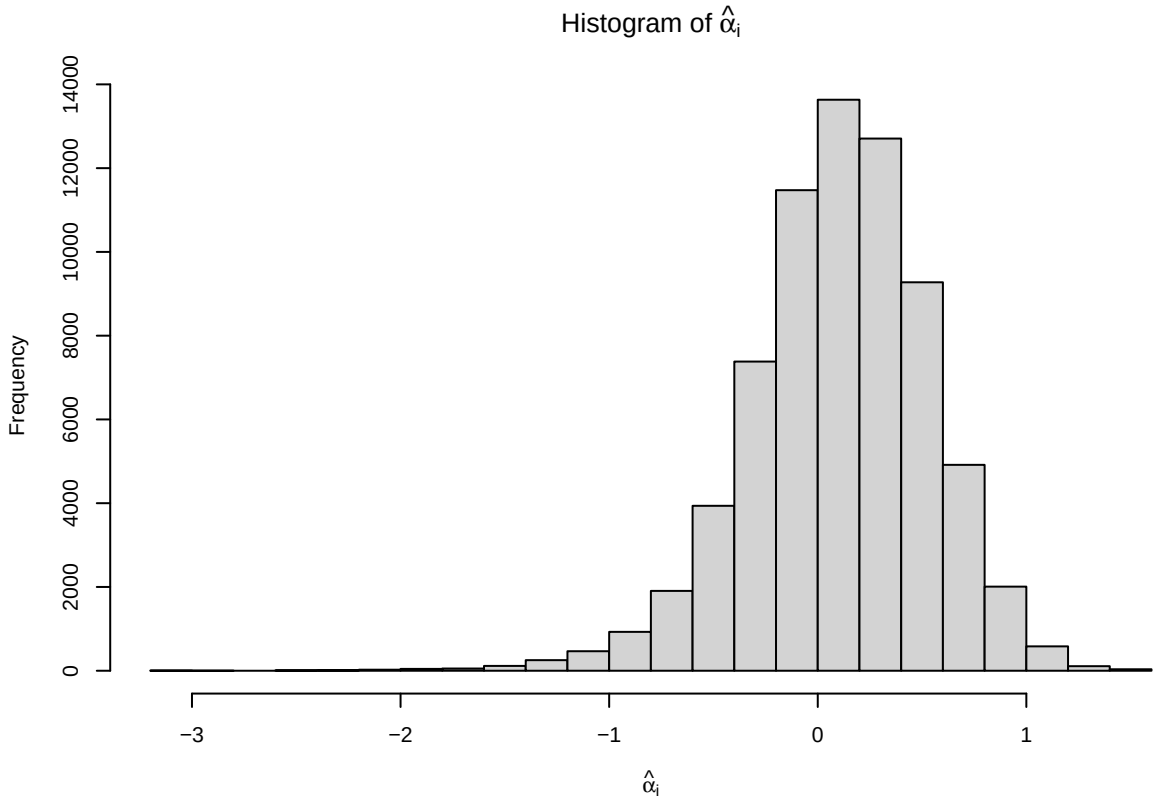
```
edx.user_effect <- edx.user_mean_ratings |>
  mutate(userId = as.integer(userId),
         a = mean_rating - mu)
```

```
str(edx.user_effect)
```

```
## 'data.frame':    69878 obs. of  4 variables:
##  $ userId     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ mean_rating: num  5 3.29 3.94 4.06 3.92 ...
##  $ n          : num  19 17 31 35 74 39 96 727 21 112 ...
##  $ a          : num  1.488 -0.218 0.423 0.545 0.406 ...
```

The following code snippet plots a histogram that provides a visual representation of the variability of the *User Effect* across users in the *data frame object* we have just obtained:

```
par(cex = 0.7)
hist(edx.user_effect$a, 30, xlab = TeX(r'[$\hat{alpha}_{i}$]'),
     main = TeX(r'[Histogram of $\hat{alpha}_{i}$]'))
```




Now, we are ready to compute the *Mean Squared Errors (MSEs)* for the samples used in *K-Fold Cross-Validation* (additionally using the clamp helper function described in section Utility Functions of Appendix A).

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

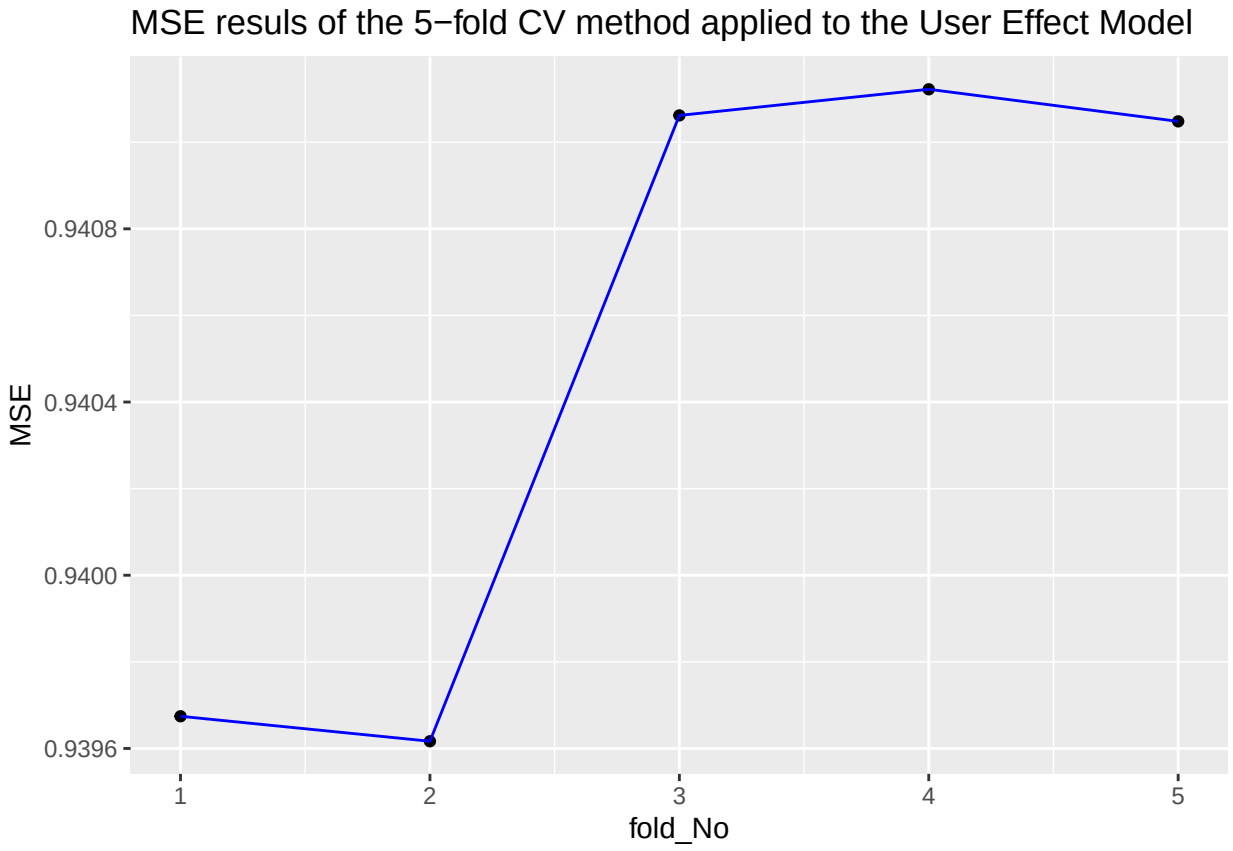
The code snippet that performs this operation is shown below:

```
put_log("Computing the RMSE taking into account user effects...")
edx.user_effect.MSEs <- sapply(edx_CV, function(cv_fold_dat){
  cv_fold_dat$validation_set |>
    left_join(edx.user_effect, by = "userId") |>
    mutate(resid = rating - clamp(mu + a)) |>
    pull(resid) |> mse()
})
```

In the code snippet above, we also use the user-defined function mse described in section (Root) Mean Squared Error Calculation of Appendix A.

The following code snippet visualizes the results obtained above:

```
data.frame(fold_No = 1:5, MSE = edx.user_effect.MSEs) |>
  data.plot(title = "MSE resuls of the 5-fold CV method applied to the User Effect Model",
            xname = "fold_No",
            yname = "MSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

We can now calculate the *RMSE* for the current model as a *square root* of the average of the *MSE* values we obtained above, using the following line of code:

```
edx.user_effect.RMSE <- sqrt(mean(edx.user_effect.MSEs))
```

Finally, we add the *RMSE* value calculated for the *UE Model* to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UE <- RMSEs.ResultTibble.OMR |>
  RMSEs.AddRow("UE Model",
               edx.user_effect.RMSE,
               comment = "User Effect (UE) Model")
```

```
RMSE_kable(RMSEs.ResultTibble.UE)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

## 2.4 User+Movie Effect (UME) Model

The complete source code of the *User+Movie Effect* (hereafter *UM Effect* or *UME* for short) computation described in this section is available in the User+Movie Effect (UME) Model section of the capstone-movielens.main.R script on *GitHub*.

### 2.4.1 Movie Effect Analysis

#### 2.4.1.1 Movies' Popularity

In Section 24.1 *Case study: recommendation systems / Movie effects* of the *Course Textbook (New Edition)*, the author draws our attention to the fact that some movies are generally rated higher than others[13].

To prove this fact, we can find out the movies with the highest number of ratings using the following code snippet:

```
edx.ordered_movie_ratings <- edx |> group_by(movieId, title) |>
  summarize(number_of_ratings = n()) |>
  arrange(desc(number_of_ratings))
```

```
print(head(edx.ordered_movie_ratings))
```

```
## # A tibble: 6 x 3
## # Groups:   movieId [6]
##   movieId title                                number_of_ratings
##     <int> <chr>                                            <int>
## 1     296 Pulp Fiction (1994)                              31362
## 2     356 Forrest Gump (1994)                              31079
## 3     593 Silence of the Lambs, The (1991)                 30382
## 4     480 Jurassic Park (1993)                             29360
## 5     318 Shawshank Redemption, The (1994)                 28015
## 6     110 Braveheart (1995)                                26212
```

Now, we can figure out the most given ratings in order from most to least. The code snippet below performs this operation:

```
edx.rating_groups <- edx |>  group_by(rating) |>
     summarise(count = n()) |>
     arrange(desc(count))
```

```
print(edx.rating_groups)
```

```
## # A tibble: 10 x 2
##    rating   count
##     <dbl>   <int>
##  1    4   2588430
##  2    3   2121240
##  3    5   1390114
##  4    3.5  791624
##  5    2    711422
##  6    4.5  526736
##  7    1    345679
##  8    2.5  333010
##  9    1.5  106426
## 10    0.5   85374
```

#### 2.4.1.2 Rating Distribution

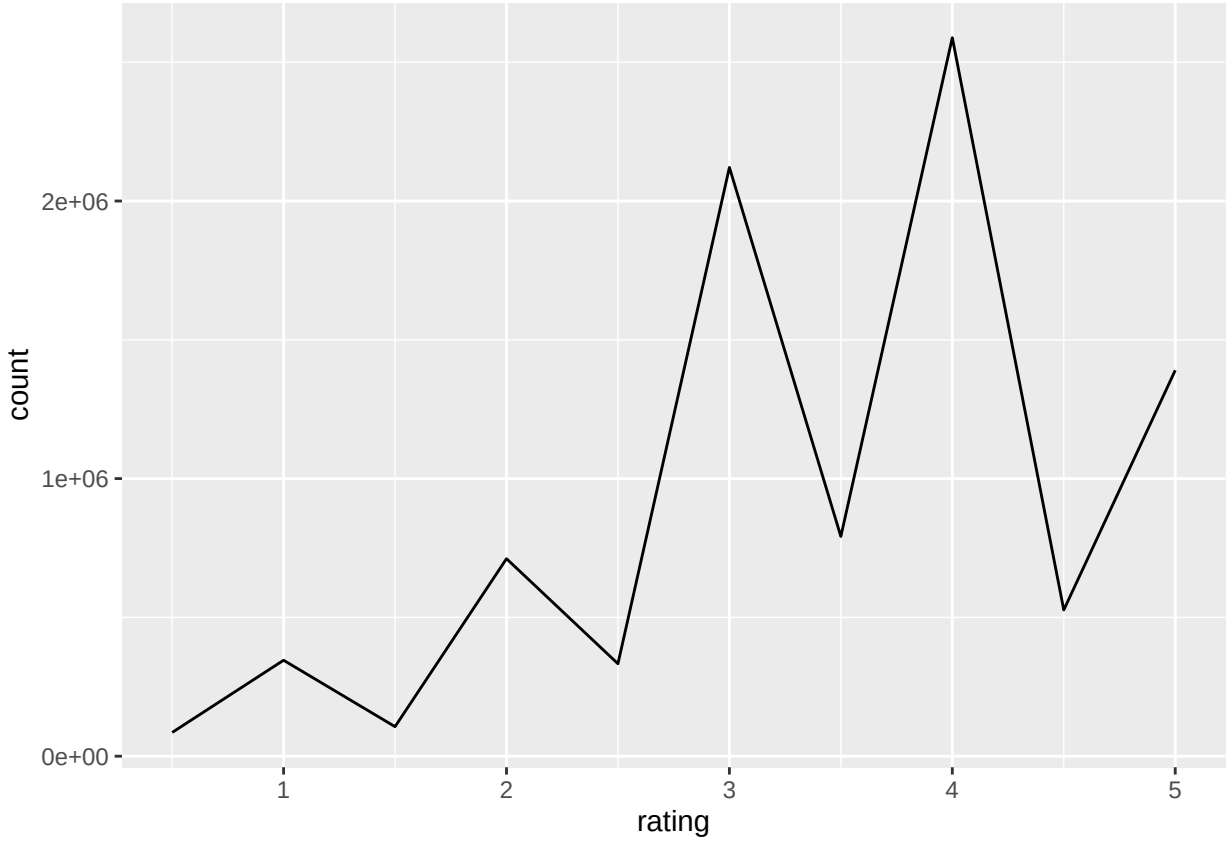
The following line of code allows us to summarize that in general, half-star ratings are less common than whole-star ratings (e.g., there are fewer ratings of 3.5 than there are ratings of 3 or 4, etc.):

```
print(edx |> group_by(rating) |> summarize(count = n()))
```

```
## # A tibble: 10 x 2
##    rating   count
##     <dbl>   <int>
##  1    0.5   85374
##  2    1    345679
##  3    1.5  106426
##  4    2    711422
##  5    2.5  333010
##  6    3   2121240
##  7    3.5  791624
##  8    4   2588430
##  9    4.5  526736
## 10    5   1390114
```

We can give even more visibility for the *rating distribution* by plotting the data for the *rating groups* (the `edx.rating_groups` object) we obtained above using the following code snippet:

```
edx.rating_groups |>
  ggplot(aes(x = rating, y = count)) +
  geom_line()
```



### 2.4.2 Mathematical Description of the UME Model

As explained in the Section 24.1, Subsection *Movie effects* of the *Textbook* (mentioned earlier in section Movies' Popularity), for the *Movie effects* we can use a linear model with a *treatment effect* $\beta_j$ for each movie, which can be interpreted as the movie effect, or the difference between the average rating for movie $j$ and the overall average $\mu$[13]:

$$Y_{i,j} = \mu + \alpha_i + \beta_j + \varepsilon_{i,j}$$

The author then gives us an idea of how to use an approximation by first computing the least square estimate $\hat{\mu}$ and $\hat{\alpha}_i$, and then estimating $\hat{\beta}_j$ as the average of the residuals $y_{i,j} - \hat{\mu} - \hat{\alpha}_i$.

Therefore, we will use this method in the *next Section*.

### 2.4.3 UME Model Building

The complete source code of the *User+Movie (UM) Effect* computation described in this section is available in the UME Model Building section of the capstone-movielens.main.R script on *GitHub*.

Below is the line of code that trains our model with the use of *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$):

```
  cv.UM_effect <- train_user_movie_effect.cv()
```

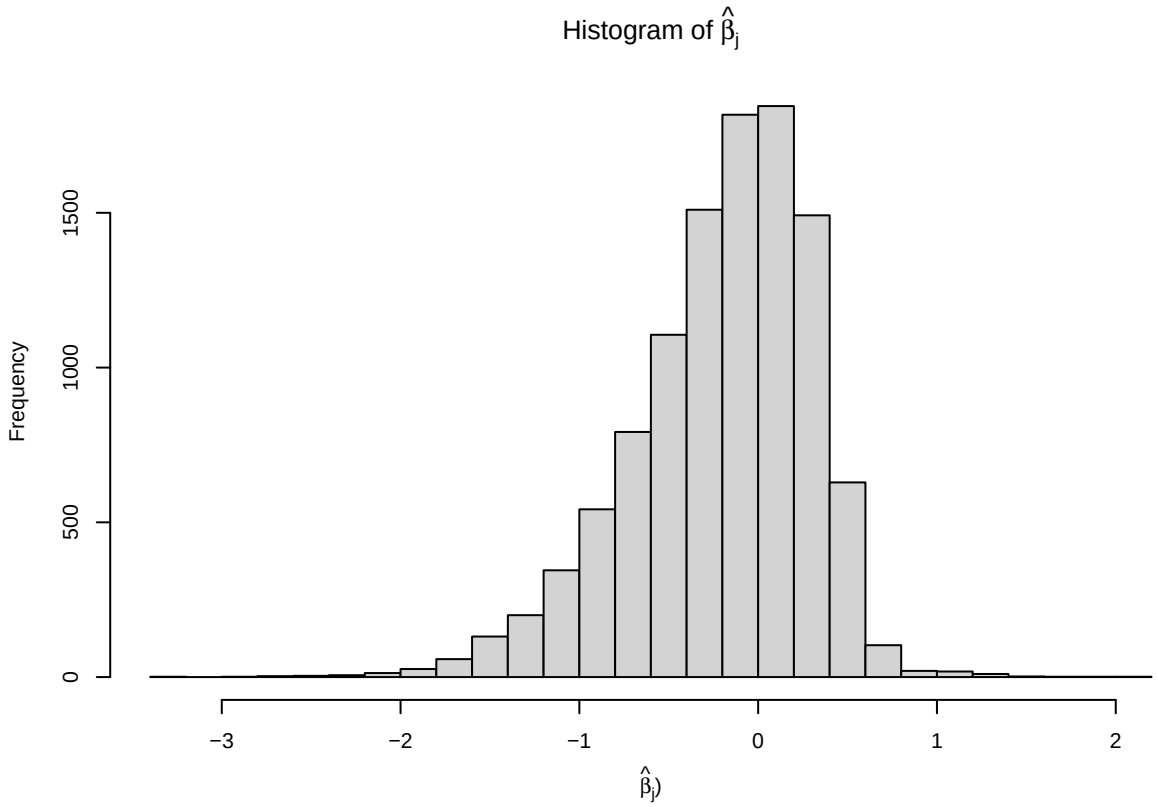
In the code snippet above we use the train_user_movie_effect.cv helper function described in section UME Model: Utility Functions of Appendix A.

```
str(cv.UM_effect)
```

```
## tibble [10,677 x 3] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ b      : num [1:10677] 0.335 -0.306 -0.365 -0.598 -0.444 ...
##  $ n      : num [1:10677] 18907 8593 5574 1253 5065 ...
```

The following code snippet plots a histogram that provides a visual representation of the variability of the *UM Effect* across movies in the *data frame object* we have just obtained:

```
par(cex = 0.7)
hist(cv.UM_effect$b, 30, xlab = TeX(r'[$\hat{beta}_{j}$)]'),
     main = TeX(r'[Histogram of $\hat{beta}_{j}$]'))
```




We can now construct predictors and calculate the *RMSE* score.

As we already outlined above, to accomplish such tasks, we use *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets).

In *this Project*, we use $K = 5$.

The following line of code validates our model by computing the *RMSE* score for the predictors constructed based on the *UM Effect*:

```
cv.UM_effect.RMSE <- calc_user_movie_effect_RMSE.cv(cv.UM_effect)
```

```
cv.UM_effect.RMSE
```

```
## [1] 0.8732081
```

In the code snippet above, we use the calc_user_movie_effect_RMSE.cv function described in section UME Model: Utility Functions of Appendix A.

Finally, we add the *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UME <- RMSEs.ResultTibble.UE |>
  RMSEs.AddRow("UME Model",
               cv.UM_effect.RMSE,
               comment = "User+Movie Effect (UME) Model")
```

```
RMSE_kable(RMSEs.ResultTibble.UME)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

### 2.4.4 UME Model Regularization

The complete version of the source code provided in this section can be found in the UME Model Regularization section of the capstone-movielens.main.R script.

We begin this section with the concept's mathematical description presented below in the subsection UME Model Regularization: *Mathematical Description*.

Next, we will implement the *UME Model Regularization* in the following three steps:

1. **Pre-configuration:** (Described in subsection UME Model Regularization: *Pre-configuration*) Preliminary determination of the optimal range of *regularization parameter* $\lambda$ values for the *K-Fold Cross-Validation* samples, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$);
2. **Fine-tuning:** (Described in subsection UME Model Regularization: *Fine-tuning*) Determining the best value of $\lambda$ with the highest possible accuracy that minimizes the *RMSE* score for the model.
3. **Retraining:** (Described in subsection UME Model Regularization: *Retraining on the* `edx` *with the best* $\lambda$) Retraining the model on the entire `edx` dataset with the best value of $\lambda$ determined in the previous step.

#### 2.4.4.1 UME Model Regularization: *Mathematical Description*

Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)* explains why and how we should use *Penalized least squares* to improve our predictions. The author also explains that the general idea of penalized regression is to control the total variability of the movie effects[14]:

$$\sum_{j=1}^{n} \beta_j^2$$

Specifically, instead of minimizing the least squares equation, we minimize an equation that adds a penalty:

$$\sum_{i,j} \left(y_{u,i} - \mu - \alpha_i - \beta_j\right)^2 + \lambda \sum_{j} \beta_j^2 \tag{1}$$

The first term is just the sum of squares and the second is a penalty that gets larger when many $\beta_i$s are large. Using calculus, we can actually show that the values of $\beta_i$ that minimize this equation are:

$$\hat{\beta}_j(\lambda) = \frac{1}{\lambda + n_j} \sum_{i=1}^{n_j} \left(Y_{i,j} - \mu - \alpha_i\right) \tag{2}$$

where $n_j$ is the number of ratings made for movie $j$.

This approach will have our desired effect: when our sample size $n_j$ is very large, we obtain a stable estimate and the penalty $\lambda$ is effectively ignored since $n_j + \lambda \approx n_j$. Yet when the $n_j$ is small, then the estimate $\hat{\beta}_i(\lambda)$ is shrunken towards 0. The larger the $\lambda$, the more we shrink[15].

#### 2.4.4.2 UME Model Regularization: *Pre-configuration*

The complete version of the source code provided in this section can be found in the UME Model Regularization: Pre-configuration section of the capstone-movielens.main.R script.

We need to do some pre-configuration to determine a suitable range of $\lambda$ for subsequent fine-tuning of our current model. For this purpose, we will use the user-defined helper function tune.model_param, passing the regularize.test_lambda.UM_effect.cv (another custom helper function) as the value of its argument fn_tune.test.param_value.

The functions tune.model_param and regularize.test_lambda.UM_effect.cv are described in the sections Model Tuning Utils and UME Model: Regularization, respectively, of Appendix A.

The following code snippet performs this operation:

```
lambdas <- seq(0, 1, 0.1)
cv.UME.preset.result <-
  tune.model_param(lambdas, regularize.test_lambda.UM_effect.cv)

put_log1("Preliminary regularization set-up of `lambda`s range for the UME Model has been completed
for the %1-Fold Cross-Validation samples.",
CVFolds_N)
```

```
str(cv.UME.preset.result)
```

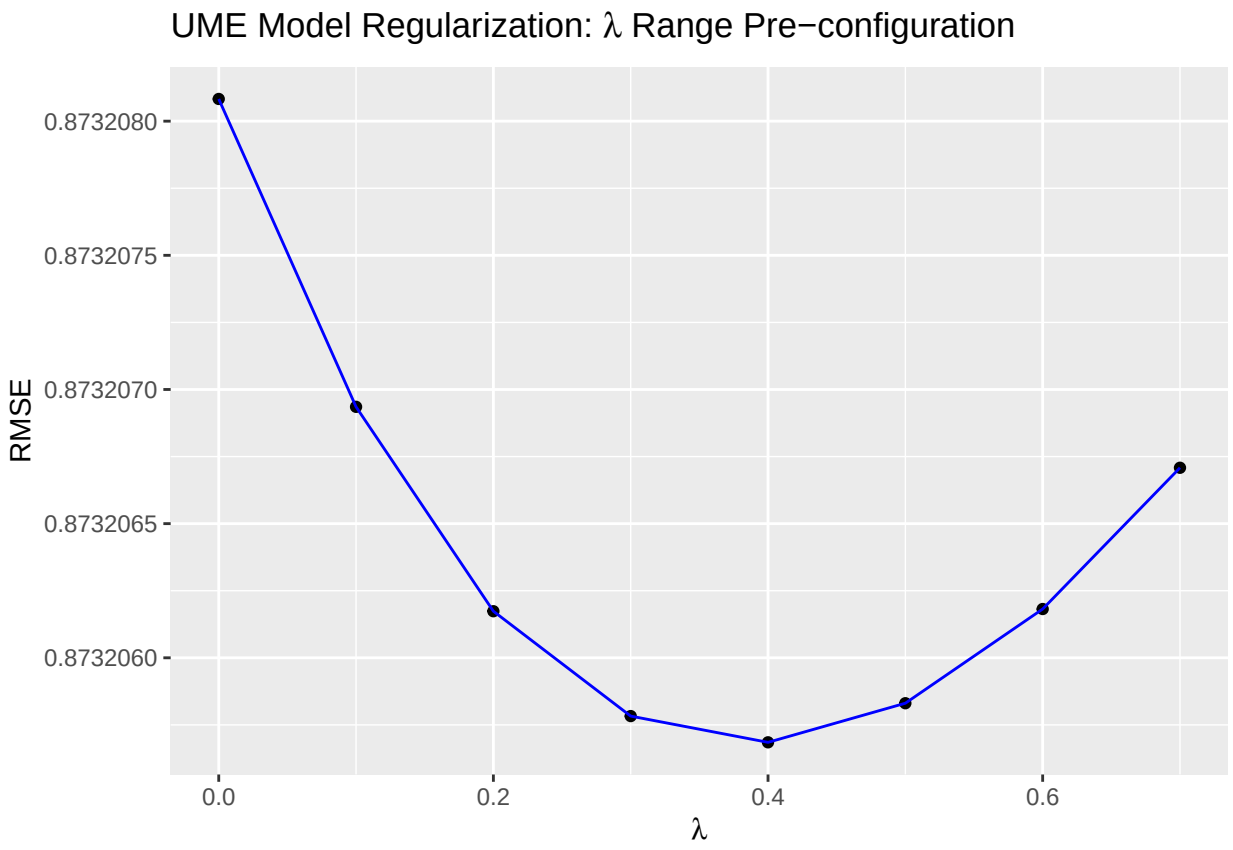
```
## List of 2
##  $ tuned.result:'data.frame':    8 obs. of  2 variables:
##   ..$ RMSE           : num [1:8] 0.873 0.873 0.873 0.873 0.873 ...
##   ..$ parameter.value: num [1:8] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
##  $ best_result : Named num [1:2] 0.4 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

```
cv.UME.preset.result$best_result
```

```
## param.best_value        best_RMSE
##        0.4000000        0.8732057
```

Now, let's visualize the results of the $\lambda$ range pre-configuration using the following code snippet:

```
cv.UME.preset.result$tuned.result |>
  data.plot(title = TeX(r'[UME Model Regularization: $\lambda$ Range Pre-configuration]'),
              xname = "parameter.value",
              yname = "RMSE",
              xlabel = TeX(r'[$\lambda$]'),
              ylabel = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

#### 2.4.4.3 UME Model Regularization: *Fine-tuning*

The complete version of the source code provided in this section can be found in the UME Model Regularization: Fine-tuning section of the capstone-movielens.main.R script on *GitHub*.

We are now ready to perform the *fine-tuning step* of our model *regularization* process to determine the best value for the $\lambda$ parameter.

The following piece of code prepares the interval of values for the $\lambda$ parameter, over which the operation has to be done:

```
endpoints <-
  get_fine_tune.param.endpoints(cv.UME.preset.result$tuned.result)

UM_effect.loop_starter <- c(endpoints["start"],
                            endpoints["end"],
                            8)
```

```
## *** Values of the endpoints and the divisor for the interval of ‘lambda‘ values ***
```

```
UM_effect.loop_starter
```

```
## start   end
##   0.3   0.5   8.0
```

The helper function get_fine_tune.param.endpoints used in the code snippet above is described in the sections Model Tuning Utils of Appendix A.

And the next code snippet below accomplishes the task of *fine-tuning* the model:

```
UME.rglr.fine_tune.cache.base_name <- "UME.rglr.fine-tune"

UME.rglr.fine_tune.results <-
  model.tune.param_range(UM_effect.loop_starter,
                         UME.rglr.fine_tune.cache.path,
                         UME.rglr.fine_tune.cache.base_name,
                         regularize.test_lambda.UM_effect.cv)

UME.rglr.fine_tune.RMSE.best <- UME.rglr.fine_tune.results$best_result["best_RMSE"]
```

The custom functions model.tune.param_range and regularize.test_lambda.UM_effect.cv used in the code snippet above are described in sections Model Tuning Utils and UME Model: Regularization, respectively, of Appendix A.

Below are the results of fine-tuning the *UME Model*:

```
## *** Path to the cache directory for intermediate fine-tuning results ***
```

```
UME.rglr.fine_tune.cache.path
```

```
## [1] "data/regularization/1.UM-effect/fine-tune"
```

```
## *** Fine-tuning results object data structure ***
```

```
str(UME.rglr.fine_tune.results)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 0.387 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: Named num [1:3] 3.87e-01 3.87e-01 9.54e-07
##   ..- attr(*, "names")= chr [1:3] "" "" ""
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 0.387 0.387 0.387 0.387 0.387 ...
##   ..$ RMSE           : num [1:9] 0.873 0.873 0.873 0.873 0.873 ...
```

```
## *** Fine-tuning: best results ***
```

```
UME.rglr.fine_tune.results$best_result
```

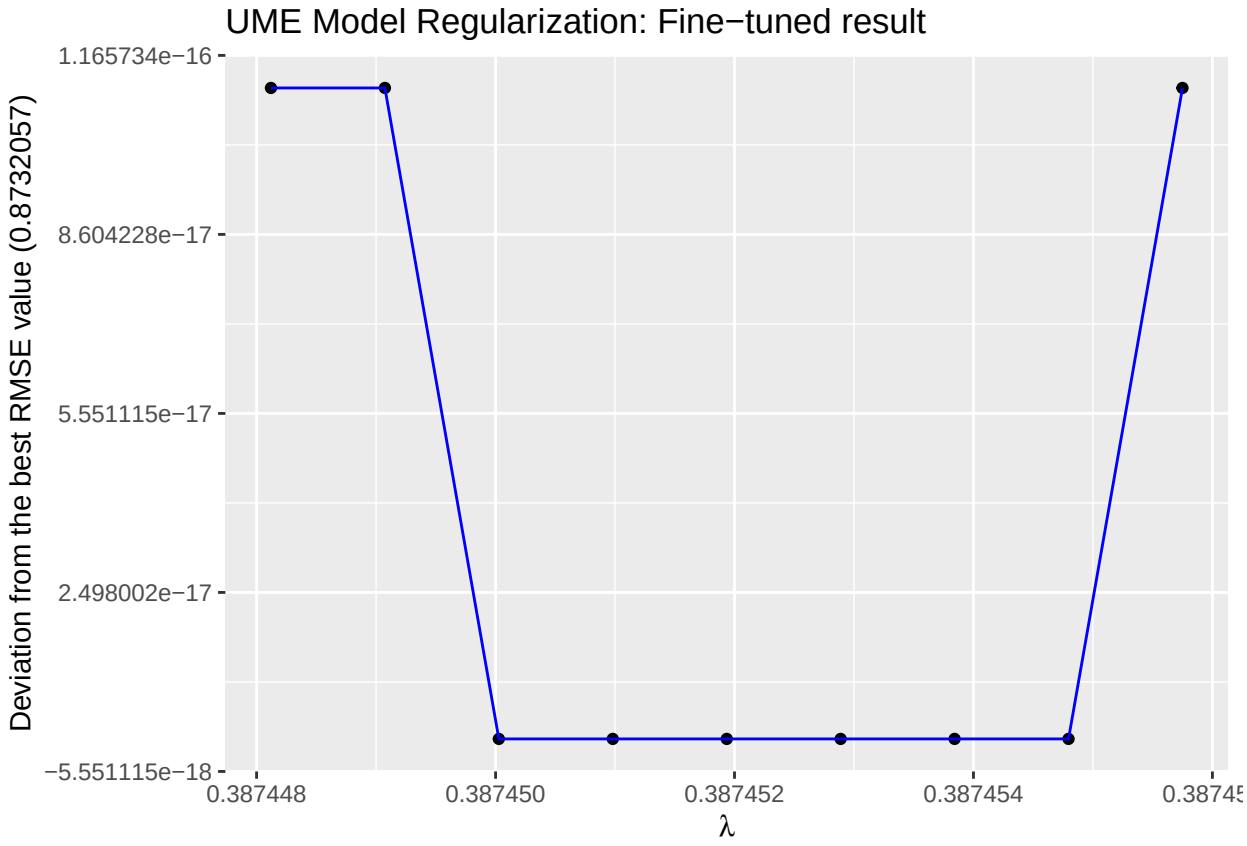
```
## param.best_value        best_RMSE
##        0.3874500        0.8732057
```

```
UME.rglr.fine_tune.RMSE.best
```

```
## best_RMSE
## 0.8732057
```

The following code snippet provides a visual representation of the *fine-tuning results* we have just computed:

```
UME.rglr.fine_tune.results$tuned.result |>
  data.plot(title = "UME Model Regularization: Fine-tuned result",
              xname = "parameter.value",
              yname = "RMSE",
              xlabel = TeX(r'[$\lambda$]'),
              ylabel = str_glue("Deviation from the best RMSE value (",
                                as.character(round(UME.rglr.fine_tune.RMSE.best, digits = 7)),
                                ")"),
              normalize = TRUE)
```




Note that in the code snippet above, we use the custom data visualization function data.plot (described in section Data Visualization Functions of Appendix A) with the argument normalize set to `TRUE`, which means that deviations from the minimum $y$ value are used to plot, rather than the $y$ values themselves.

#### 2.4.4.4 UME Model Regularization: *Retraining on the edx with the best* $\lambda$

The complete version of the source code provided in this section are available in the UME Model Regularization: Re-training with the best `lambda` section of the capstone-movielens.main.R script on *GitHub*.

Now, we can refine our *User+Movie Effect Model* by retraining on the entire `edx` dataset with the best value of the $\lambda$ parameter we just figured out (let's call it *Regularized User+Movie Effect Model* or *Regularized UME Model* for short), for the definitive *RMSE* calculation and use in subsequent models.

The following code snippet performs this operation:

```
best_result <- UME.rglr.fine_tune.results$best_result
UME.rglr.best_lambda <- best_result["param.best_value"]

put_log1("Re-training Regularized User+Movie Effect Model for the best `lambda`: %1...",
         UME.rglr.best_lambda)

rglr.UM_effect <- train_user_movie_effect(edx, UME.rglr.best_lambda)
```

In the code snippet above we use the train_user_movie_effect function described in section UME Model: Utility Functions of Appendix A.

```
## *** The Best UM Effect Fine-tuning Results ***
```

```
UME.rglr.fine_tune.results$best_result
```

```
## param.best_value        best_RMSE
##        0.3874500        0.8732057

## *** Regularized UM Effect Structure ***
```

```
str(rglr.UM_effect)
```

```
## tibble [10,677 x 3] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ b      : Named num [1:10677] 0.331 -0.305 -0.364 -0.599 -0.443 ...
##   ..- attr(*, "names")= chr [1:10677] "param.best_value" "param.best_value" "param.best_value" "para
##  $ n      : int [1:10677] 23790 10779 7028 1577 6400 12346 7259 821 2278 15187 ...
```

```
print_log1("Regularized UME Model has been re-trained for the best `lambda`: %1.",
         UME.rglr.best_lambda)
```

```
## Regularized UME Model has been re-trained for the best ‘lambda‘: 0.38745002746582.
```

Now, we are ready to construct predictors and calculate the *RMSE* score for the ultimately *Regularized UME Model* using the following line of code:

```
UME.rglr.retrain.RMSE <- calc_user_movie_effect_RMSE.cv(rglr.UM_effect)
```

```
print_log1("The best RMSE for the UME Model after being regularized: %1",
         UME.rglr.retrain.RMSE)
```

```
## The best RMSE for the UME Model after being regularized: 0.872972999076497
```

In the code snippet above, we use the calc_user_movie_effect_RMSE.cv function described in section UME Model: Utility Functions of Appendix A.

Finally, we add the definitive *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.rglr.UME <- RMSEs.ResultTibble.UME |>
  RMSEs.AddRow("Regularized User+Movie Effect Model",
               UME.rglr.retrain.RMSE,
               comment = "Computed for `lambda` = %1" |>
                 msg.glue(UME.rglr.best_lambda))
```

```
RMSE_kable(RMSEs.ResultTibble.rglr.UME)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for ‘lambda‘ = 0.38745002746582 |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

## 2.5 User+Movie+Genre Effect (UMGE) Model

The complete source code of the *User+Movie+Genre Effect* computation described in this section is available in the User+Movie+Genre Effect (UMGE) Model section of the capstone-movielens.main.R script on *GitHub*.

As we can see from the `edx` dataset structure, the `Movielens` dataset also has a genres column. This column includes every genre that applies to the movie (some movies fall under several genres):

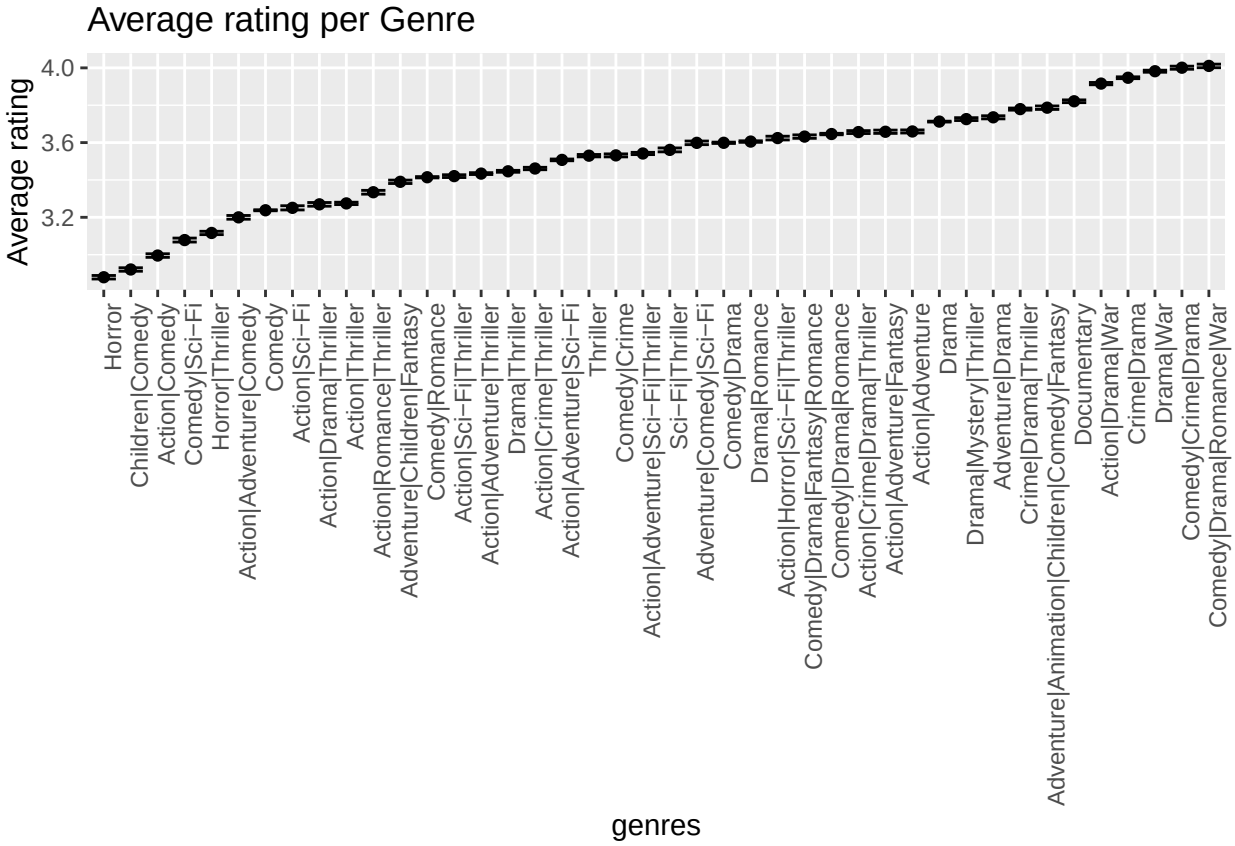
```
str(edx)
```

```
## 'data.frame':    9000055 obs. of  6 variables:
##  $ userId   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : int  122 185 292 316 329 355 356 362 364 370 ...
##  $ rating   : num  5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int  838985046 838983525 838983421 838983392 838983392 838984474 838983653 838984885 83
##  $ title    : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres   : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|Ac
```

### 2.5.1 Movie Genres Effect Analysis

The plot below shows strong evidence of a genre effect (for illustrative purposes, the plot shows only categories with more than 40, 000 ratings).

```
# Preparing data for plotting:
genre_ratins_grp <- edx |>
  mutate(genre_categories = as.factor(genres)) |>
  group_by(genre_categories) |>
  summarize(n = n(), rating_avg = mean(rating), se = sd(rating)/sqrt(n())) |>
  filter(n > 40000) |>
  mutate(genres = reorder(genre_categories, rating_avg)) |>
  select(genres, rating_avg, se, n)

# Creating plot:
genre_ratins_grp |>
  ggplot(aes(x = genres, y = rating_avg, ymin = rating_avg - 2*se, ymax = rating_avg + 2*se)) +
  geom_point() +
  geom_errorbar() +
  ggtitle("Average rating per Genre") +
  ylab("Average rating") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```




Below are worst and best ratings categories:

```
## [1] "The worst ratings are for the genre category: Horror"
```

```
## [1] "The best ratings are for the genre category: Comedy|Drama|Romance|War"
```

### 2.5.2 Mathematical Description of the UMGE Model

If we define a *genre treatment effect* $g_{i,j}$ for user's $i$ rating of movie $j$, we can use the following models to account for the `genre` effect:

$$Y_{i,j} = \mu + \alpha_i + \beta_j + g_{i,j} + \varepsilon_{i,j} \tag{3}$$

where $g_{i,j}$ is an *aggregation function* which is explained in detail in *Section 22.3: "Review of Aggregation Functions" of "Recommender Systems Handbook"* (*Chapter 22: "Aggregation of Preferences in Recommender Systems"*, p. 712) book[16].

In the formula above $g_{i,j}$ denotes a *genre effect* for user's $i$ rating of movie $j$, so that:

$$g_{i,j} = \sum_{k=1}^{K} x_{i,j}^{k} \gamma_k$$

with $x_{i,j}^{k} = 1$ if $g_{i,j}$ includes genre $k$, and $x_{i,j}^{k} = 0$ otherwise.

Therefore, for our current model, we can compute a predicted value

$$\hat{g}_{i,j} = g_{i,j} + \varepsilon_{i,j}$$

as a residual:

$$\hat{g}_{i,j} = Y_{i,j} - (\mu + \alpha_i + \beta_j) \tag{4}$$

### 2.5.3 UMGE Model Building

The complete source code of builing and training the current model is available in the UMGE Model Building section of the capstone-movielens.main.R script on *GitHub*.

Below is the line of code that trains our model with the use of *K-Fold Cross Validation* method, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$):

```
cv.UMG_effect <- train_user_movie_genre_effect.cv()
```

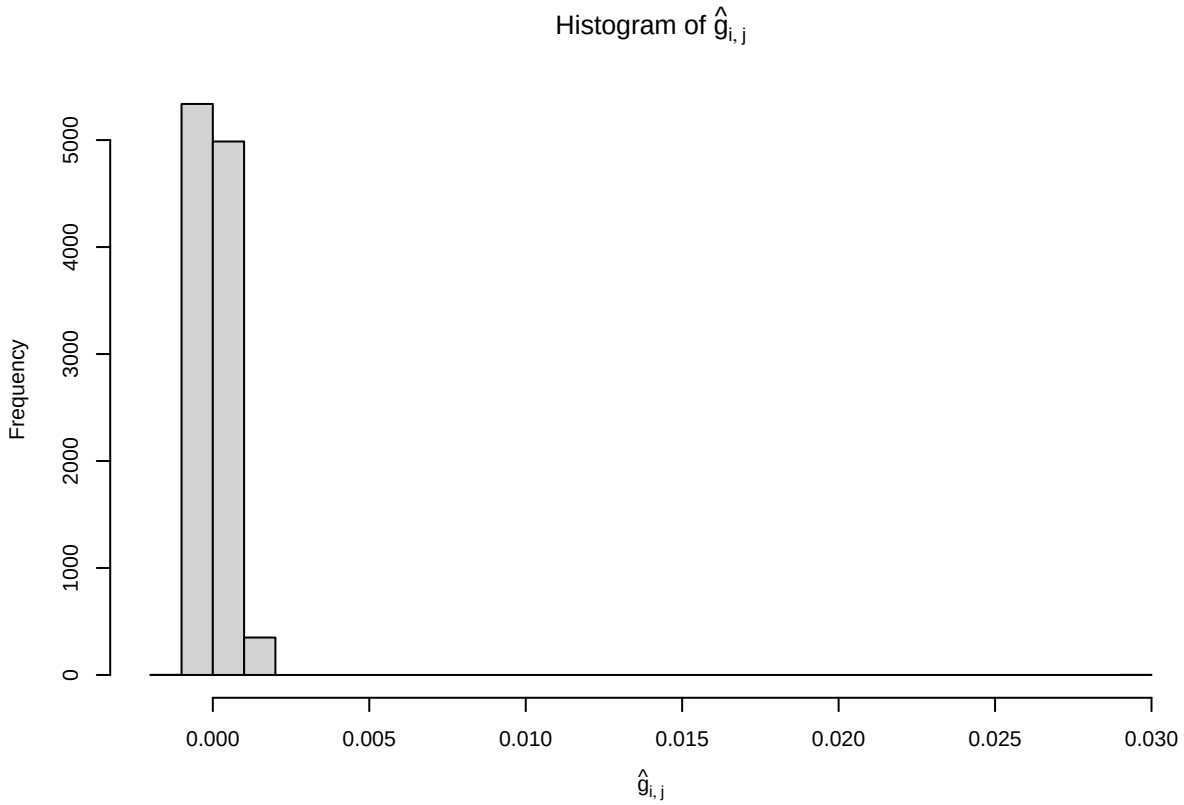
In the code snippet above we use the train_user_movie_genre_effect.cv function described in section UMGE Model: Utility Functions of Appendix A.

```
str(cv.UMG_effect)
```

```
## tibble [10,677 x 2] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ g      : num [1:10677] -9.28e-06 -7.23e-05 -1.07e-04 -6.46e-05 4.56e-05 ...
```

The following code snippet plots a histogram that provides a visual representation of the variability of the *UMG Effect* across movies in the *data frame object* we have just obtained:

```
par(cex = 0.7)
hist(cv.UMG_effect$g, 30, xlab = TeX(r'[$\hat{g}_{i,j}$]'),
     main = TeX(r'[Histogram of $\hat{g}_{i,j}$]'))
```




We can now construct predictors and calculate the *RMSE* score for the *UMGE Model* using the following line of code:

```
cv.UMG_effect.RMSE <- calc_user_movie_genre_effect_RMSE.cv(cv.UMG_effect)
```

```
cv.UMG_effect.RMSE
```

```
## [1] 0.872973
```

In the code snippet above, we use the calc_user_movie_genre_effect_RMSE.cv function described in section UMGE Model: Utility Functions of Appendix A.

Finally, we add the *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGE <- RMSEs.ResultTibble.rglr.UME |>
  RMSEs.AddRow("UMGE Model",
               cv.UMG_effect.RMSE,
               comment = "User+Movie+Genre Effect (UMGE) Model")
```

```
RMSE_kable(RMSEs.ResultTibble.UMGE)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

Unfortunately, for some reason, we do not see any improvement here yet.

### 2.5.4 UMGE Model Regularization

The complete version of the source code provided in this section can be found in the UMGE Model Regularization section of the capstone-movielens.main.R script.

We begin this section with the concept's mathematical description presented below in the subsection UMGE Model Regularization: *Mathematical Description*.

Next, we will implement the *UMGE Model Regularization* in the following three steps:

1. **Pre-configuration:** (Described in subsection UMGE Model Regularization: *Pre-configuration*) Preliminary determination of the optimal range of *regularization parameter* $\lambda$ values for the *K-Fold Cross-Validation* samples, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$);
2. **Fine-tuning:** (Described in the subsection UMGE Model Regularization: *Fine-tuning*) Determining the best value of $\lambda$ with the highest possible accuracy that minimizes the *RMSE* score for the model.
3. **Retraining:** (Described in subsection UMGE Model Regularization: *Retraining on the `edx` with the best* $\lambda$) Retraining the model on the entire `edx` dataset with the best value of $\lambda$ determined in the previous step.

#### 2.5.4.1 UMGE Model Regularization: *Mathematical Description*

We have already explained the idea of the *Linear Model Regularization* earlier in section UME Model Regularization. Let's extend the concept outlined there to our current model.

In this case, the formula (1) for adding a penalty takes the form:

$$\sum_{i,j} \left(y_{u,i} - \mu - \alpha_i - \beta_j - g_{i,j}\right)^2 + \lambda \sum_{i,j} g_{i,j}^2 \tag{5}$$

And the formula (2) for calculating the values of the *treatment effect* that minimizes the equation will take the form:

$$\hat{g}_{i,j}(\lambda) = \frac{1}{\lambda + n_g} \sum_{r=1}^{n_g} \left(Y_{i,j} - \mu - \alpha_i - \beta_j\right) \tag{6}$$

where $n_g$ is the number of ratings made for genre $g$.

#### 2.5.4.2 UMGE Model Regularization: *Pre-configuration*

The complete version of the source code provided in this section can be found in the UMGE Model Regularization: Pre-configuration section of the capstone-movielens.main.R script.

As we explained earlier in the UME Model Regularization: *Pre-configuration*, we need to do some pre-configuration to determine a suitable range of $\lambda$ for subsequent fine-tuning of our current model.

As before, we are going to use the helper function tune.model_param, but this time, passing another function as its argument fn_tune.test.param_value: the function regularize.test_lambda.UMG_effect.cv specifically design for the *UMGE Model*.

The functions tune.model_param and regularize.test_lambda.UMG_effect.cv are described in the sections Model Tuning Utils and UMGE Model: Regularization, respectively, of Appendix A.

The following code snippet performs this operation:

```
lambdas <- seq(0, 0.2, 0.01)
cv.UMGE.preset.result <-
  tune.model_param(lambdas, regularize.test_lambda.UMG_effect.cv)

put_log1("Preliminary regularization set-up of `lambda`s range for the UMGE Model has been completed
for the %1-Fold Cross Validation samples.",
CVFolds_N)
```

```
str(cv.UMGE.preset.result)
```

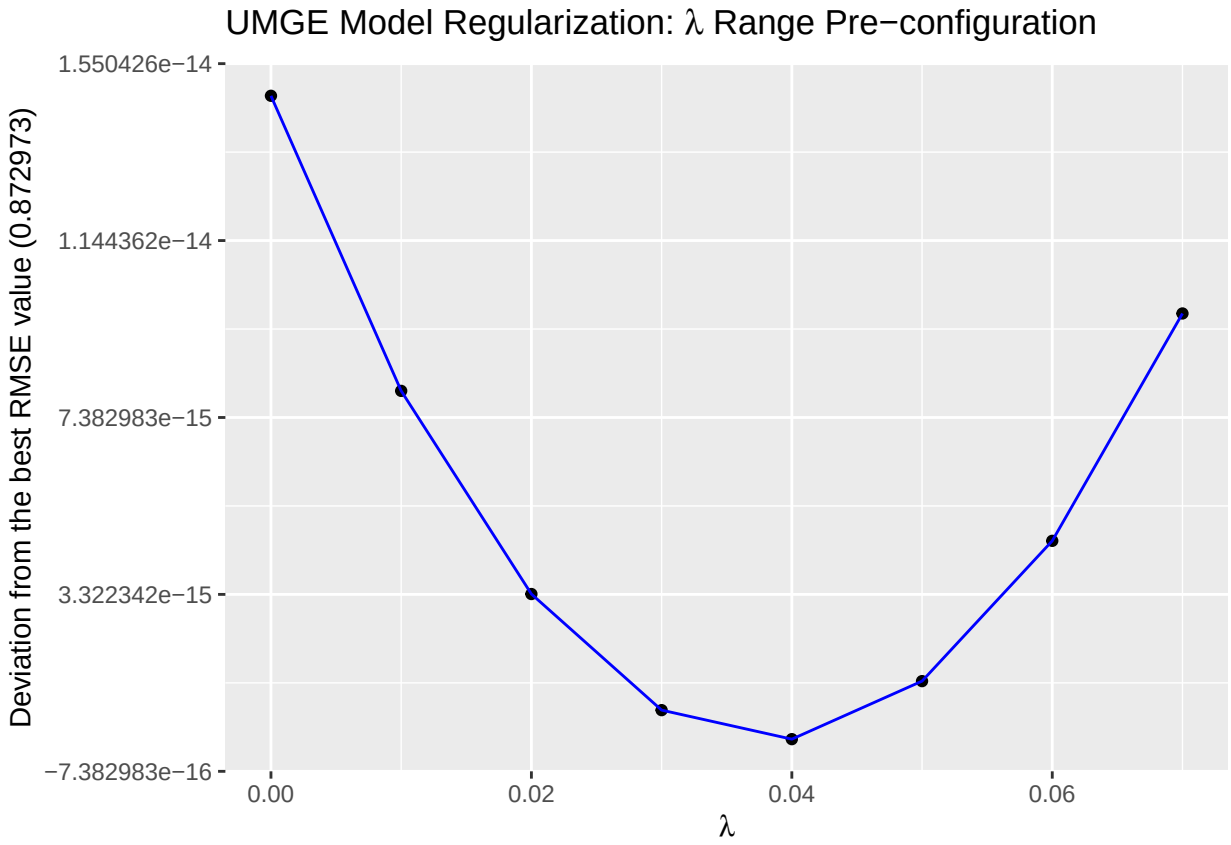
```
## List of 2
##  $ tuned.result:'data.frame':    8 obs. of  2 variables:
##   ..$ RMSE          : num [1:8] 0.873 0.873 0.873 0.873 0.873 ...
##   ..$ parameter.value: num [1:8] 0 0.01 0.02 0.03 0.04 0.05 0.06 0.07
##  $ best_result : Named num [1:2] 0.04 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

```
cv.UMGE.preset.result$best_result
```

```
## param.best_value        best_RMSE
##         0.040000         0.872973
```

Now, let's visualize the results of the $\lambda$ range pre-configuration using the following code snippet:

```
cv.UMGE.preset.result$tuned.result |>
  data.plot(title = TeX(r'[UMGE Model Regularization: $\lambda$ Range Pre-configuration]'),
              xname = "parameter.value",
              yname = "RMSE",
              xlabel = TeX(r'[$\lambda$]'),
              ylabel = str_glue("Deviation from the best RMSE value (",
                                as.character(round(cv.UMGE.preset.result$best_result["best_RMSE"],
                                                   digits = 7)),
                                ")"),
              normalize = TRUE)
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

#### 2.5.4.3 UMGE Model Regularization: *Fine-tuning*

The complete version of the source code provided in this section can be found in the UMGE Model Regularization: Fine-tuning section of the capstone-movielens.main.R script on *GitHub*.

We are now ready to perform the fine-tuning step of our model *regularization* process to determine the best value for the $\lambda$ parameter.

The following piece of code prepares the interval of values for the $\lambda$ parameter, over which the operation has to be done:

```
endpoints <-
  get_fine_tune.param.endpoints(cv.UMGE.preset.result$tuned.result)

UMG_effect.loop_starter <- c(endpoints["start"],
                             endpoints["end"],
                             8)
```

```
## *** Values of the endpoints and the divisor for the interval of ‘lambda‘ values ***
```

```
UMG_effect.loop_starter
```

```
## start   end
##  0.03  0.05  8.00
```

The helper function get_fine_tune.param.endpoints used in code snippet above is described in the sections Model Tuning Utils of Appendix A.

And the next code snippet below accomplishes the task of *fine-tuning* the model:

```
UMGE.rglr.fine_tune.cache.base_name <- "UMGE.rglr.fine-tuning"

UMGE.rglr.fine_tune.results <-
  model.tune.param_range(UMG_effect.loop_starter,
                         UMGE.rglr.fine_tune.cache.path,
                         UMGE.rglr.fine_tune.cache.base_name,
                         regularize.test_lambda.UMG_effect.cv)

UMGE.rglr.fine_tune.RMSE.best <- UMGE.rglr.fine_tune.results$best_result["best_RMSE"]
```

The custom functions model.tune.param_range and regularize.test_lambda.UMG_effect.cv used in the code snippet above are described in the sections Model Tuning Utils and UMGE Model: Regularization, respectively, of Appendix A.

Below are the results of fine-tuning the *UMGE Model*:

```
## *** Path to the cache directory for intermediate fine-tuning results ***
```

```
UMGE.rglr.fine_tune.cache.path
```

```
## [1] "data/regularization/2.UMG-effect/fine-tune"
```

```
## *** Fine-tuning results object data structure ***
```

```
str(UMGE.rglr.fine_tune.results)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 0.0359 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: Named num [1:3] 0.035 0.0425 0.000937
##   ..- attr(*, "names")= chr [1:3] "" "" ""
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 0.035 0.0359 0.0369 0.0378 0.0388 ...
##   ..$ RMSE           : num [1:9] 0.873 0.873 0.873 0.873 0.873 ...
```

```
## *** Fine-tuning: best results ***
```

```
UMGE.rglr.fine_tune.results$best_result
```

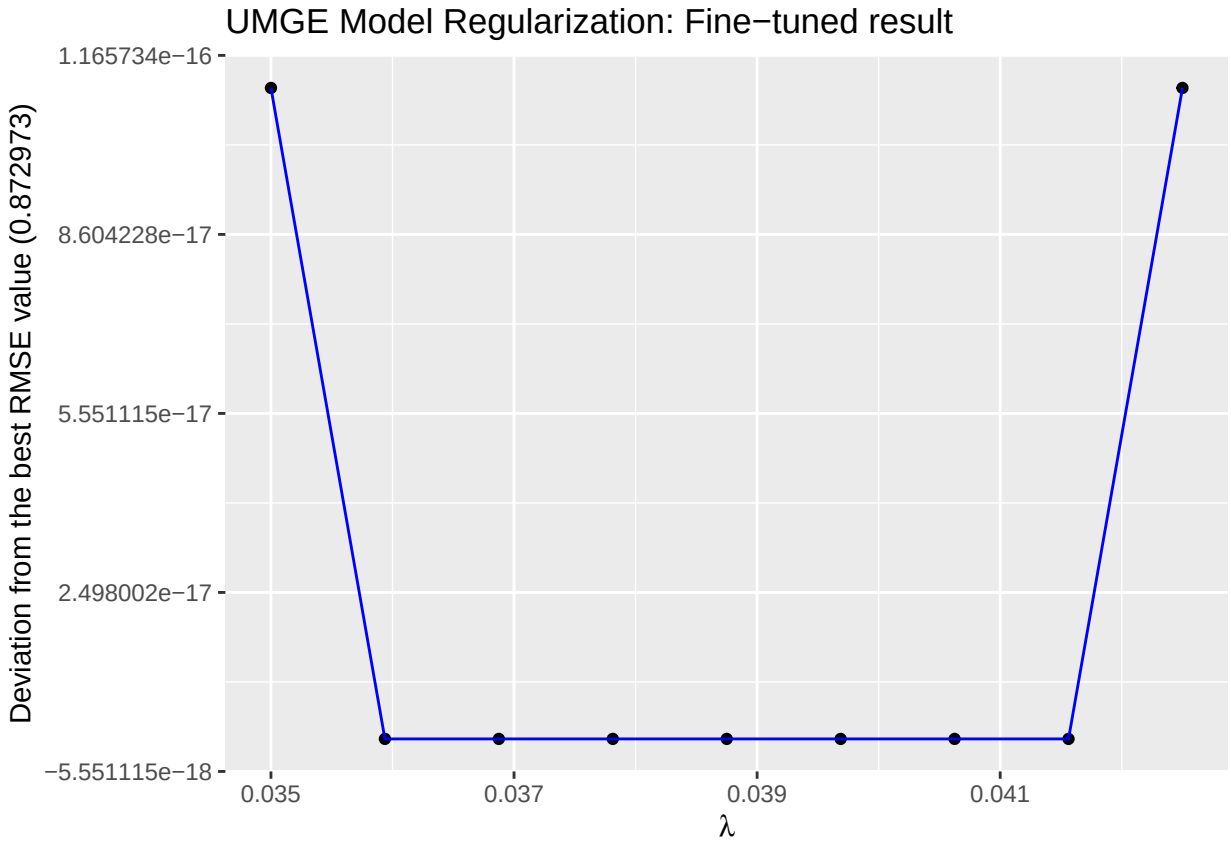
```
## param.best_value        best_RMSE
##        0.0359375        0.8729730
```

```
UMGE.rglr.fine_tune.RMSE.best
```

```
## best_RMSE
##  0.872973
```

The following code snippet provides a visual representation of the *fine-tuning results* we have just computed:

```
UMGE.rglr.fine_tune.results$tuned.result |>
  data.plot(title = "UMGE Model Regularization: Fine-tuned result",
              xname = "parameter.value",
              yname = "RMSE",
              xlabel = TeX(r'[$\lambda$]'),
              ylabel = str_glue("Deviation from the best RMSE value (",
                                as.character(round(UMGE.rglr.fine_tune.RMSE.best, digits = 7)),
                                ")"),
              normalize = TRUE)
```




Note that in the code snippet above, we use the custom data visualization function data.plot (described in section Data Visualization Functions of Appendix A) with the argument normalize set to `TRUE`, which means that deviations from the minimum $y$ value are used to plot, rather than the $y$ values themselves.

#### 2.5.4.4 UMGE Model Regularization: *Retraining on the `edx` with the best* $\lambda$

The complete version of the source code provided in this section are available in the UMGE Model Regularization: Re-training with the best `lambda` section of the capstone-movielens.main.R script on *GitHub*.

Now, we can refine our *User+Movie+Genre Effect (UMGE) Model* by retraining on the entire `edx` dataset with the best value of the $\lambda$ parameter we just figured out (let's call it *Regularized User+Movie+Genre Effect Model* or *Regularized UMGE Model* for short), for the definitive *RMSE* calculation and use in subsequent models.

The following code snippet performs this operation:

```
best_result <- UMGE.rglr.fine_tune.results$best_result
UMGE.rglr.best_lambda <- best_result["param.best_value"]

put_log1("Re-training Regularized User+Movie+Genre Effect Model for the best `lambda`: %1...",
         UMGE.rglr.best_lambda)

rglr.UMG_effect <- edx.sgr |> train_user_movie_genre_effect(UMGE.rglr.best_lambda)
```

In the code snippet above we use the train_user_movie_genre_effect function described in section UMGE Model: Utility Functions of Appendix A.

```
## *** The Best UMG Effect Fine-tuning Results ***
```

```
UMGE.rglr.fine_tune.results$best_result
```

```
## param.best_value        best_RMSE
##        0.0359375        0.8729730
```

```
## *** Regularized UMG Effect Structure ***
```

```
str(rglr.UMG_effect)
```

```
## tibble [10,677 x 2] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ g      : num [1:10677] -1.35e-05 -5.11e-05 -1.08e-04 -2.77e-05 -2.46e-04 ...
```

```
print_log1("Regularized UMGE Model has been re-trained for the best `lambda`: %1.",
         UMGE.rglr.best_lambda)
```

```
## Regularized UMGE Model has been re-trained for the best ‘lambda‘: 0.0359375.
```

Now, we are ready to construct predictors and calculate the *RMSE* score for the ultimately *Regularized UMGE Model* using the following line of code:

```
  rglr.UMG_effect.RMSE <- calc_user_movie_genre_effect_RMSE.cv(rglr.UMG_effect)
```

```
print_log1("The best RMSE for the UMGE Model after being regularized: %1",
         rglr.UMG_effect.RMSE)
```

```
## The best RMSE for the UMGE Model after being regularized: 0.872972778604307
```

In the code snippet above, we use the calc_user_movie_genre_effect_RMSE.cv function described in section UMGE Model: Utility Functions of Appendix A.

Finally, we add the definitive *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.rglr.UMGE <- RMSEs.ResultTibble.UMGE |>
  RMSEs.AddRow("Regularized User+Movie+Genre Effect Model",
               rglr.UMG_effect.RMSE,
               comment = "Computed for `lambda` = %1" |>
                 msg.glue(UMGE.rglr.best_lambda))
```

```
RMSE_kable(RMSEs.ResultTibble.rglr.UMGE)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for ‘lambda‘ = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for ‘lambda‘ = 0.0359375 |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

As we can see, the current model still doesn't show any significant improvement after *Regularization*, even though the data analysis we made earlier in the Movie Genres Effect Analysis section showed strong evidence of a genre effect.

Apparently, we need a better model to account for a genre effect more efficiently.

## 2.6 User+Movie+Genre+Year Effect (UMGYE) Model

### 2.6.1 Year Effect Analysis

The *Year Effect* visualization and analysis code in *this Section* are cited from the article Movie Recommendation System using R - BEST mentioned earlier in this report[17].

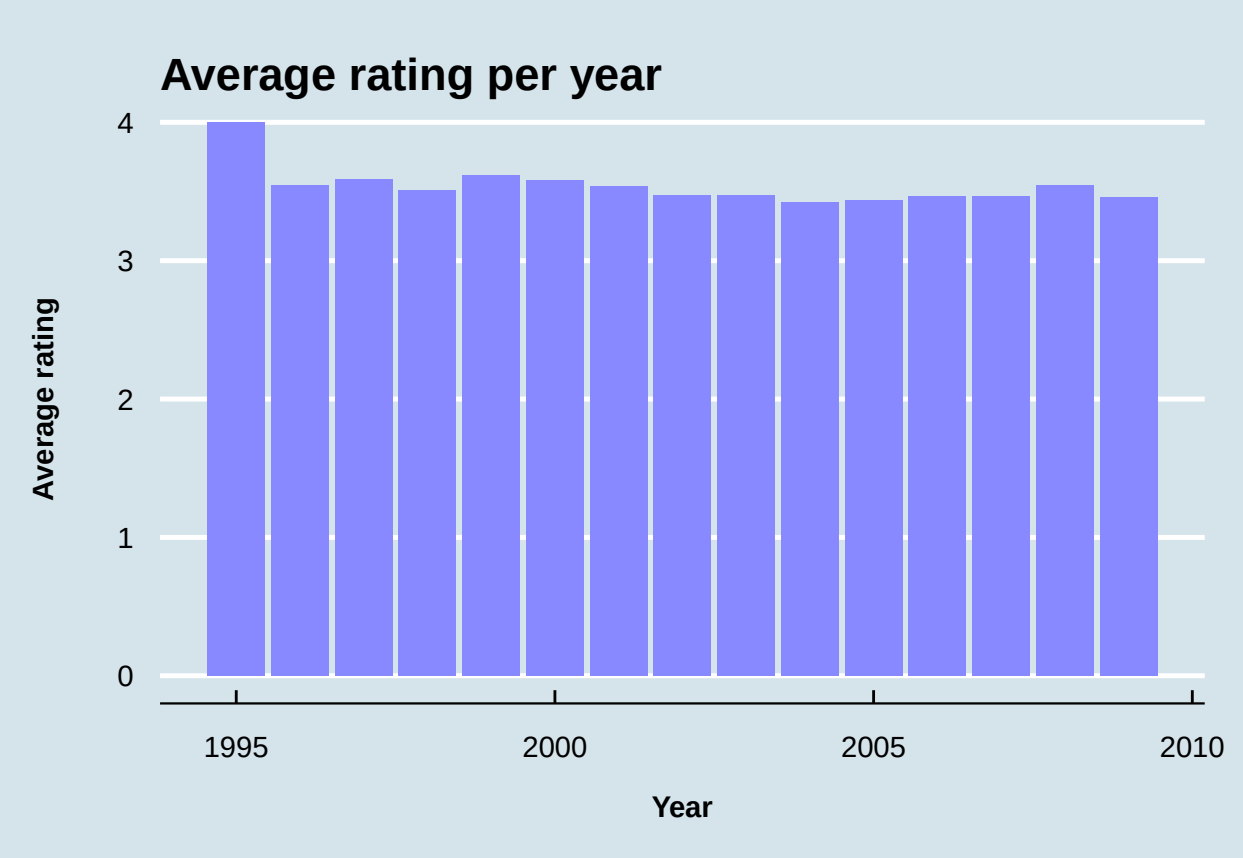
#### 2.6.1.1 Yearly rating count[17]

```
print(edx |>
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) |>
  group_by(year) |>
  summarize(count = n())
)
```

```
## # A tibble: 15 x 2
##     year   count
##    <dbl>   <int>
##  1  1995       2
##  2  1996  942772
##  3  1997  414101
##  4  1998  181634
##  5  1999  709893
##  6  2000 1144349
##  7  2001  683355
##  8  2002  524959
##  9  2003  619938
## 10  2004  691429
## 11  2005 1059277
## 12  2006  689315
## 13  2007  629168
## 14  2008  696740
## 15  2009   13123
```

#### 2.6.1.2 Average rating per year plot[17]

```
edx |>
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01"))) |>
  group_by(year) |>
  summarize(rating_avg = mean(rating)) |>
  ggplot(aes(x = year, y = rating_avg)) +
  geom_bar(stat = "identity", fill = "#8888ff") +
  ggtitle("Average rating per year") +
  xlab("Year") +
  ylab("Average rating") +
  scale_y_continuous(labels = comma) +
  theme_economist() +
  theme(axis.title.x = element_text(vjust = -5, face = "bold"),
        axis.title.y = element_text(vjust = 10, face = "bold"),
        plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))
```



### 2.6.2 Mathematical Description of the UMGYE Model

If we define $\gamma(v_{i,j})$ as the *year effect* for the year (here denotes by $v_{i,j}$) of rating a movie $j$ by a user $i$, the formula (3) describing the *UMGE Model*, for the current model, takes the form:

$$Y_{i,j} = \mu + \alpha_i + \beta_j + g_{i,j} + \gamma(v_{i,j}) + \varepsilon_{i,j} \tag{7}$$

Therefore, the formula (4) for calculation the prediction of a *genre effect* as a residual, for a *year effect*

$$\hat{\gamma}(v_{i,j}) = \gamma(v_{i,j}) + \varepsilon_{i,j}$$

takes the following form:

$$\hat{\gamma}(v_{i,j}) = Y_{i,j} - (\mu + \alpha_i + \beta_j + g_{i,j}) \tag{8}$$

### 2.6.3 UMGYE Model Building

The complete source code of building and training the current model is available in the UMGYE Model Building section of the capstone-movielens.main.R script on *GitHub*.

Below is the line of code that trains our model with the use of *K-Fold Cross Validation* method, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$):

```
  cv.UMGY_effect <- train_UMGY_effect.cv()
```

In the code snippet above we use the train_UMGY_effect.cv user-defined function described in section UMGE Model: Utility Functions of Appendix A.

```
str(cv.UMGY_effect)
```

```
## tibble [15 x 2] (S3: tbl_df/tbl/data.frame)
##  $ year: num [1:15] 1995 1996 1997 1998 1999 ...
##  $ ye  : num [1:15] 0.6694 0.1152 0.0118 0.0262 0.0335 ...
```

We can now construct predictors and calculate the *RMSE* score for the *UMGYE Model* using the following line of code:

```
cv.UMGY_effect.RMSE <- calc_UMGY_effect_RMSE.cv(cv.UMGY_effect)
```

```
cv.UMGY_effect.RMSE
```

```
## [1] 0.8723973
```

In the code snippets above, we use the calc_UMGY_effect_RMSE.cv user-defined function described in section UMGYE Model: Utility Functions of Appendix A.

Finally, we add the *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGYE <- RMSEs.ResultTibble.rglr.UMGE |>
  RMSEs.AddRow("UMGYE Model",
               cv.UMGY_effect.RMSE,
               comment = "User+Movie+Genre+Year Effect (UMGYE) Model")
```

```
RMSE_kable(RMSEs.ResultTibble.UMGYE)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for 'lambda' = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |

In the code snippets above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

Now, we have a bit better result than one for the previous (*Regularized UMGE*) model.

### 2.6.4 UMGYE Model Regularization

The complete version of the source code provided in this section can be found in the UMGYE Model Regularization section of the capstone-movielens.main.R script.

We begin this section with the concept's mathematical description presented below in the subsection UMGYE Model Regularization: *Mathematical Description*.

Next, we will implement the *UMGYE Model Regularization* in the following three steps:

1. **Pre-configuration:** (Described in subsection UMGYE Model Regularization: *Pre-configuration*) Preliminary determination of the optimal range of *regularization parameter* $\lambda$ values for the *K-Fold Cross-Validation* samples, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$);
2. **Fine-tuning:** (Described in the subsection UMGYE Model Regularization: *Fine-tuning*) Determining the best value of $\lambda$ with the highest possible accuracy that minimizes the *RMSE* score for the model.
3. **Retraining:** (Described in subsection UMGYE Model Regularization: *Retraining on the `edx` with the best* $\lambda$) Retraining the model on the entire `edx` dataset with the best value of $\lambda$ determined in the previous step.

#### 2.6.4.1 UMGYE Model Regularization: *Mathematical Description*

We have already explained the idea of the *Linear Model Regularization* earlier in section UME Model Regularization. We have also seen how the formula (1) for adding a penalty to the *UME Model* is transformed into the formula (5) for the *UMGE Model*. For the current model, this formula takes the form:

$$\sum_{i,j} \left(y_{u,i} - \mu - \alpha_i - \beta_j - g_{i,j} - \gamma(v_{i,j})\right)^2 + \lambda \sum_{i,j} \gamma(v_{i,j})^2 \tag{9}$$

And the formula (6) for calculating the values of the *treatment effect* that minimizes the equation, for the current model, takes the form:

$$\hat{\gamma}(v_{i,j}, \lambda) = \frac{1}{\lambda + n_v} \sum_{r=1}^{n_v} \left(Y_{i,j} - \mu - \alpha_i - \beta_j - g_{i,j}\right) \tag{10}$$

where $n_v$ is the number of ratings made in year $v$.

#### 2.6.4.2 UMGYE Model Regularization: *Pre-configuration*

The complete version of the source code provided in this section can be found in the UMGYE Model Regularization: Pre-configuration section of the capstone-movielens.main.R script on *GitHub*.

As for the previous models, we will use the function tune.model_param passing in the fn_tune.test.param_value argument the model-specific helper function (this time designed for the *UMGYE Model*): regularize.test_lambda.UMGY_effect.cv.

The functions tune.model_param and regularize.test_lambda.UMGY_effect.cv are described in the sections Model Tuning Utils and UMGYE Model: Regularization, respectively, of Appendix A.

The following code snippet performs this operation:

```
  lambdas <- seq(0, 512, 32)
  cv.UMGYE.preset.result <-
    tune.model_param(lambdas,
                     regularize.test_lambda.UMGY_effect.cv,
                     steps.beyond_min = 16)
```

```
str(cv.UMGYE.preset.result)
```

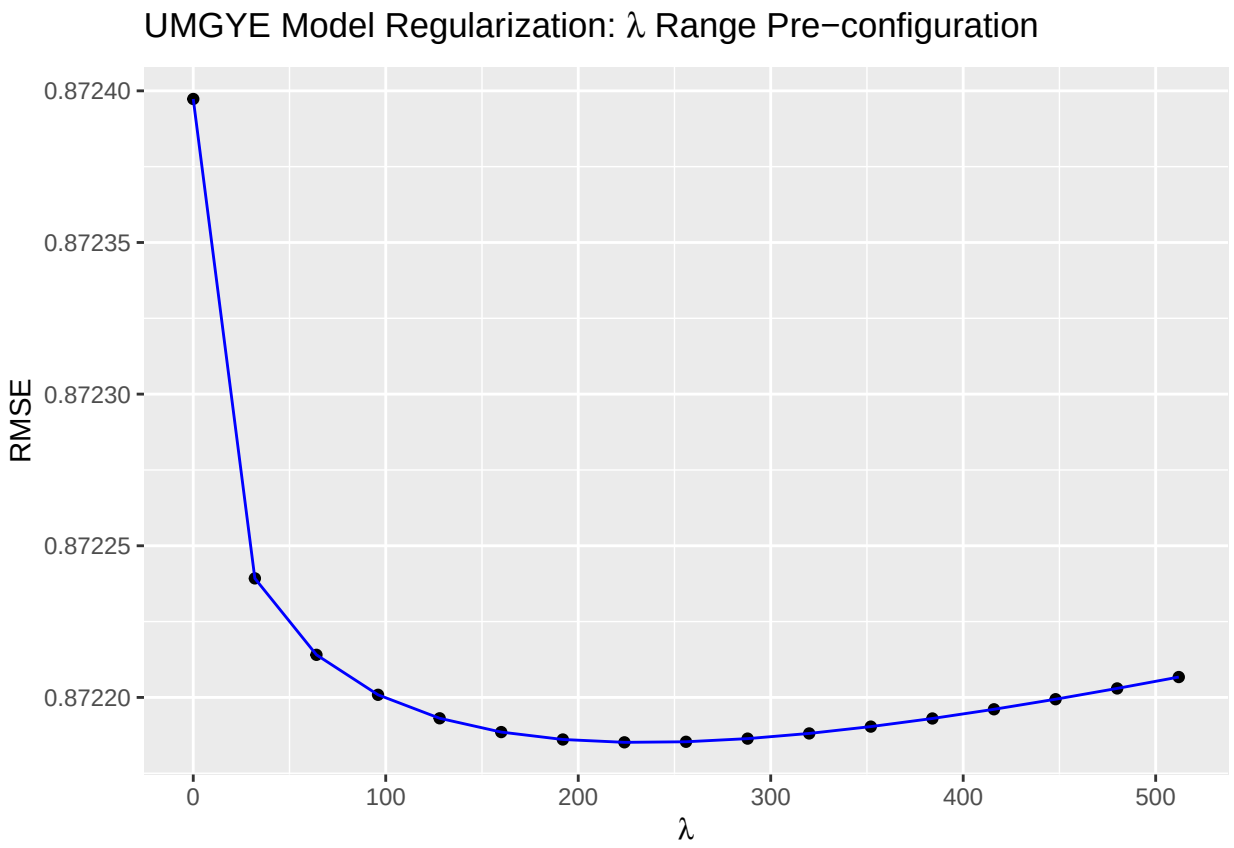
```
## List of 2
##  $ tuned.result:'data.frame':    17 obs. of  2 variables:
##   ..$ RMSE           : num [1:17] 0.872 0.872 0.872 0.872 0.872 ...
##   ..$ parameter.value: num [1:17] 0 32 64 96 128 160 192 224 256 288 ...
##  $ best_result : Named num [1:2] 224 0.872
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

```
cv.UMGYE.preset.result$best_result
```

```
## param.best_value        best_RMSE
##      224.0000000        0.8721852
```

Now, let's visualize the results of the $\lambda$ range pre-configuration using the following code snippet:

```
cv.UMGYE.preset.result$tuned.result |>
  data.plot(title = TeX(r'[UMGYE Model Regularization: $\lambda$ Range Pre-configuration]'),
            xname = "parameter.value",
            yname = "RMSE",
            xlabel = TeX(r'[$\lambda$]'),
            ylabel = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

#### 2.6.4.3 UMGYE Model Regularization: *Fine-tuning*

The complete version of the source code provided in this section can be found in the UMGYE Model Regularization: Fine-tuning section of the capstone-movielens.main.R script on *GitHub*.

We are now ready to perform the fine-tuning step of our model *regularization* process to determine the best value for the $\lambda$ parameter.

The following piece of code prepares the interval of values for the $\lambda$ parameter, over which the operation has to be done:

```
endpoints <-
  get_fine_tune.param.endpoints(cv.UMGYE.preset.result$tuned.result)

UMGYE.loop_starter <- c(endpoints["start"],
                        endpoints["end"],
                        8)
```

```
## *** Values of the endpoints and the divisor for the interval of ‘lambda‘ values ***
```

```
UMGYE.loop_starter
```

```
## start   end
##   192   256     8
```

The helper function get_fine_tune.param.endpoints used in code snippet above is described in the sections Model Tuning Utils of Appendix A.

And the next code snippet below accomplishes the task of *fine-tuning* the model:

```
cache.base_name <- "UMGYE.rglr.fine-tuning"

UMGYE.rglr.fine_tune.results <-
  model.tune.param_range(UMGYE.loop_starter,
                         UMGYE.rglr.fine_tune.cache.path,
                         cache.base_name,
                         regularize.test_lambda.UMGY_effect.cv)

UMGYE.rglr.fine_tune.RMSE.best <- UMGYE.rglr.fine_tune.results$best_result["best_RMSE"]
```

The custom functions model.tune.param_range and regularize.test_lambda.UMGY_effect.cv used in the code snippet above are described in the sections Model Tuning Utils and UMGYE Model: Regularization, respectively, of Appendix A.

Below are the results of fine-tuning the *UMGYE Model*:

```
## *** Path to the cache directory for intermediate fine-tuning results ***
```

```
UMGYE.rglr.fine_tune.cache.path
```

```
## [1] "data/regularization/3.UMGY-effect/fine-tune"
```

```
## *** Fine-tuning results object data structure ***
```

```
str(UMGYE.rglr.fine_tune.results)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 233.777 0.872
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: Named num [1:3] 2.34e+02 2.34e+02 7.63e-05
##   ..- attr(*, "names")= chr [1:3] "" "" ""
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 234 234 234 234 234 ...
##   ..$ RMSE           : num [1:9] 0.872 0.872 0.872 0.872 0.872 ...
```

```
## *** Fine-tuning: best results ***
```

```
UMGYE.rglr.fine_tune.results$best_result
```

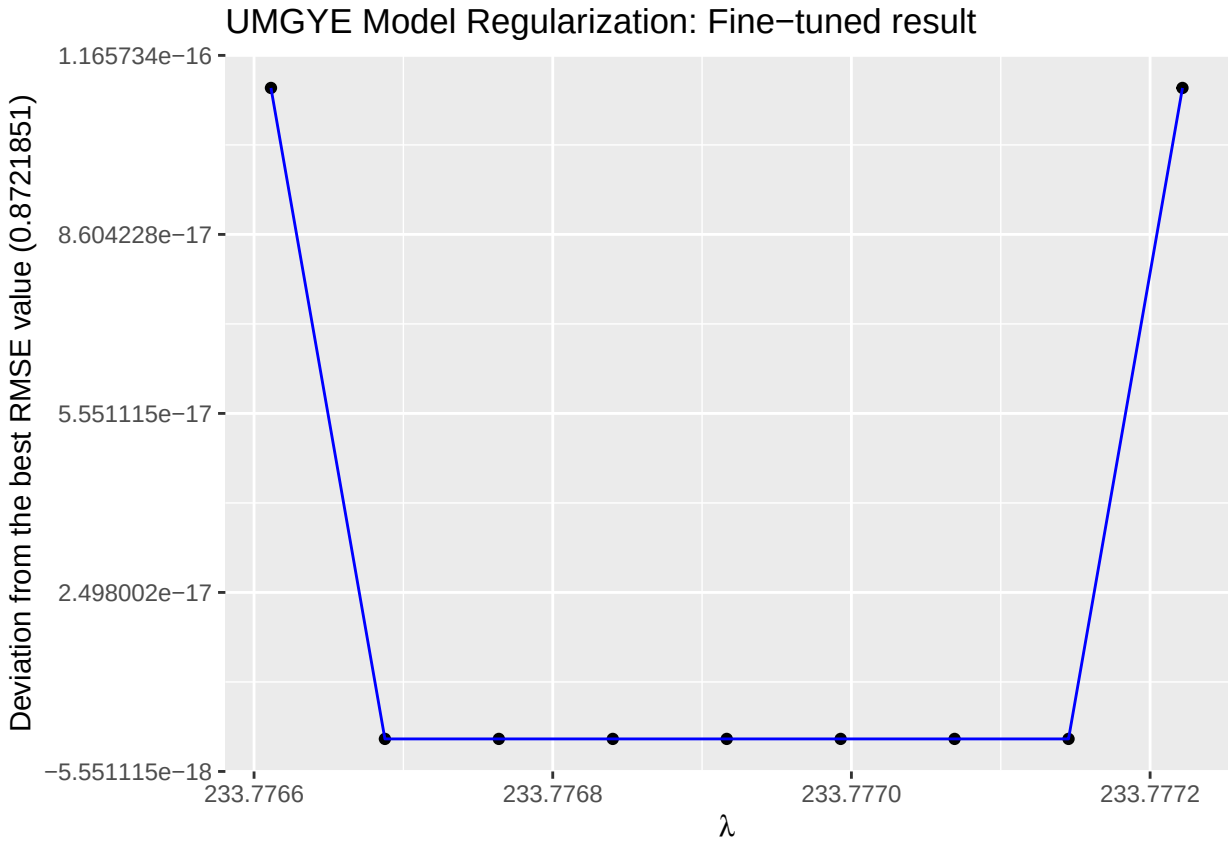
```
## param.best_value        best_RMSE
##      233.7766876        0.8721851
```

```
UMGYE.rglr.fine_tune.RMSE.best
```

```
## best_RMSE
## 0.8721851
```

The following code snippet provides a visual representation of the *fine-tuning results* we have just computed:

```
UMGYE.rglr.fine_tune.results$tuned.result |>
  data.plot(title = "UMGYE Model Regularization: Fine-tuned result",
              xname = "parameter.value",
              yname = "RMSE",
              xlabel = TeX(r'[$\lambda$]'),
              ylabel = str_glue("Deviation from the best RMSE value (",
                                as.character(round(UMGYE.rglr.fine_tune.RMSE.best, digits = 7)),
                                ")"),
              normalize = TRUE)
```




Note that in the code snippet above, we use the custom data visualization function data.plot (described in section Data Visualization Functions of Appendix A) with the argument normalize set to `TRUE`, which means that deviations from the minimum $y$ value are used to plot, rather than the $y$ values themselves.

#### 2.6.4.4 UMGYE Model Regularization: *Retraining on the `edx` with the best* $\lambda$

The complete version of the source code provided in this section are available in the UMGYE Model Regularization: Re-training with the best `lambda` section of the capstone-movielens.main.R script on *GitHub*.

Now, we can refine our *User+Movie+Genre+Year Effect (UMGYE) Model* by retraining on the entire `edx` dataset with the best value of the $\lambda$ parameter we just figured out (let's call it *Regularized User+Movie+Genre+Year Effect Model* or *Regularized UMGYE Model* for short), for the definitive *RMSE* calculation and use in subsequent models.

The following code snippet performs this operation:

```
best_result <- UMGYE.rglr.fine_tune.results$best_result
UMGYE.rglr.best_lambda <- best_result["param.best_value"]

put_log1("Re-training Regularized User+Movie+Genre+Year Effect Model for the best `lambda`: %1...",
         UMGYE.rglr.best_lambda)

rglr.UMGY_effect <- edx |> train_UMGY_effect(UMGYE.rglr.best_lambda)
```

In the code snippet above we use the train_UMGY_effect function described in section UMGYE Model: Utility Functions of Appendix A.

```
## *** The Best UMGY Effect Fine-tuning Results ***
```

```
UMGYE.rglr.fine_tune.results$best_result
```

```
## param.best_value        best_RMSE
##      233.7766876        0.8721851
```

```
## *** Regularized UMGY Effect Structure ***
```

```
str(rglr.UMGY_effect)
```

```
## tibble [15 x 2] (S3: tbl_df/tbl/data.frame)
##  $ year: num [1:15] 1995 1996 1997 1998 1999 ...
##  $ ye  : num [1:15] 0.0018 0.06807 0.01206 0.01017 0.00636 ...
```

```
print_log1("Regularized UMGYE Model has been re-trained for the best `lambda`: %1.",
         UMGYE.rglr.best_lambda)
```

```
## Regularized UMGYE Model has been re-trained for the best ‘lambda‘: 233.77668762207.
```

Now, we are ready to construct predictors and calculate the *RMSE* score for the ultimately *Regularized UMGYE Model* using the following line of code:

```
  rglr.UMGY_effect.RMSE <- calc_UMGY_effect_RMSE.cv(rglr.UMGY_effect)
```

```
print_log1("The best RMSE for the UMGYE Model after being regularized: %1",
         rglr.UMGY_effect.RMSE)
```

```
## The best RMSE for the UMGYE Model after being regularized: 0.872185666181505
```

In the code snippet above, we use the calc_UMGY_effect_RMSE.cv function described in section UMGYE Model: Utility Functions of Appendix A.

Finally, we add the definitive *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.rglr.UMGYE <- RMSEs.ResultTibble.UMGYE |>
  RMSEs.AddRow("Regularized UMGYE Model",
               rglr.UMGY_effect.RMSE,
               comment = "Computed for `lambda` = %1" |>
                 msg.glue(UMGYE.rglr.best_lambda))
```

```
RMSE_kable(RMSEs.ResultTibble.rglr.UMGYE)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for ‘lambda‘ = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for ‘lambda‘ = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for ‘lambda‘ = 233.77668762207 |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

## 2.7 User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model

### 2.7.1 Mathematical Description of the UMGYDE Model

If we define $d_{i,j}$ as the *number of days since the earliest record* in the `edx` dataset for movie $j$ rated by user $i$, then the formula (7) describing the *UMGYE Model*, for the current model, takes the form:

$$Y_{i,j} = \mu + \alpha_i + \beta_j + g_{i,j} + \gamma(v_{i,j}) + s(d_{i,j}) + \varepsilon_{i,j} \tag{11}$$

with $s$ a *smoothed day* function of $d_{i,j}$

Therefore, the formula (8) for calculation the prediction of a *year effect* as a residual, for a *smoothed day effect*

$$\hat{s}(d_{i,j}) = s(d_{i,j}) + \varepsilon_{i,j}$$

takes the following form:

$$\hat{s}(d_{i,j}) = Y_{i,j} - (\mu + \alpha_i + \beta_j + g_{i,j} + \gamma(v_{i,j})) \tag{12}$$

### 2.7.2 UMGYDE Model Building with `loess` Default Parameters

The complete source code of building and training the current model is available in the UMGYDEM Training with default parameters section of the capstone-movielens.main.R script on *GitHub*.

We are going to use the custom helper function (train_UMGY_SmoothedDay_effect.cv (described in section UMGYDE Model: Utility Functions of Appendix A) specifically designed to train the *UMGYDE Model*, which under the hood implicitly calls the stats::loess `R` function, passing its `degree` and `span` parameters values taken from the caller's degree and span arguments, respectively.

We will start by calling the train_UMGY_SmoothedDay_effect.cv with default argument values, which, in addition to not use *regularization technique* yet (which will be used in further training later for the current model), means that the loess function will be internally called with `degree = 1` and `span = 0.75` argument values.

Below is the line of code that trains our model with the default `degree` and `span` parameter values with the use of *K-Fold Cross Validation*, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$):

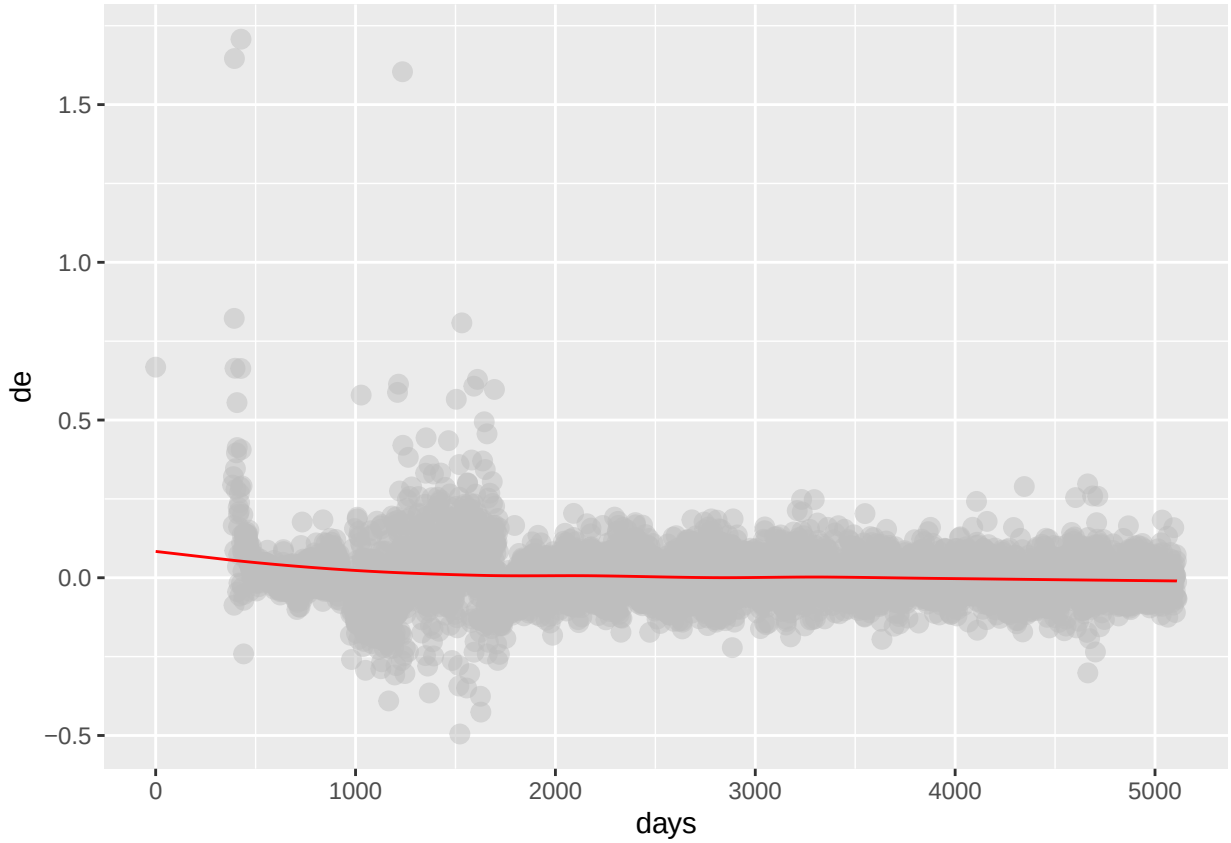
```
  cv.UMGYDE.default_params <- train_UMGY_SmoothedDay_effect.cv()
```

```
str(cv.UMGYDE.default_params)
```

```
## tibble [4,640 x 4] (S3: tbl_df/tbl/data.frame)
##  $ days       : int [1:4640] 0 385 388 389 392 393 394 396 397 399 ...
##  $ de         : num [1:4640] 0.6676 0.2939 0.1661 0.3206 -0.0871 ...
##  $ year       : num [1:4640] 1995 1996 1996 1996 1996 ...
##  $ de_smoothed: num [1:4640] 0.0834 0.0553 0.0552 0.0551 0.0549 ...
```

The following code snippet provides a visual representation of the *Smoothed Day Effect* data we have just computed:

```
cv.UMGYDE.default_params |>
  ggplot(aes(x = days)) +
  geom_point(aes(y = de), size = 3, alpha = .5, color = "grey") +
  geom_line(aes(y = de_smoothed), color = "red")
```




Now, let's construct predictors and calculate the *RMSE* score for the *current model* using the following line of code:

```
cv.UMGYDE.default_params.RMSE <- cv.UMGYDE.default_params |>
  calc_UMGY_SmoothedDay_effect.RMSE.cv()
```

```
cv.UMGYDE.default_params.RMSE
```

```
## [1] 0.872241
```

In the code snippets above, we use the calc_UMGY_SmoothedDay_effect.RMSE.cv custom helper function described in section UMGYDE Model: Utility Functions of Appendix A.

Finally, we add the *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGYDE <- RMSEs.ResultTibble.rglr.UMGYE |>
  RMSEs.AddRow("UMGYDE (Default) Model",
               cv.UMGYDE.default_params.RMSE,
               comment = "User+Movie+Genre+Year+Day Effect (UMGYDE) Model
computed using `stats::loess` function with `degree=1` & `span=0.75` parameter values.")
```

```
RMSE_kable(RMSEs.ResultTibble.UMGYDE)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for 'lambda' = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for 'lambda' = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using 'stats::loess' function with 'degree=1' & 'span=0.75' parameter values. |

In the code snippet above we also use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

Unfortunately, with the default parameters, we obtained an even worse result than before. Let's tune the current model by adjusting the `degree` and `span` parameters and see what we get.

### 2.7.3 UMGYDE Model Tuning by `degree` and `span` Parameters

The complete source code of the solution described in this section is available in the UMGYDE Model Tuning by `span` & `degree` parameters section of the capstone-movielens.main.R script on *GitHub*.

Since the `degree` parameter can only take three discrete values, 0, 1, and 2, we will perform the tuning in three stages - one for each value.

In addition, we will divide each stage into the following two steps for tuning the model by the `span` parameter:

1. **Pre-configuration:** Preliminary determination of the optimal range of the `span` parameter values for the *K-Fold Cross-Validation* samples, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$);
2. **Fine-tuning:** Determining the best value of the `span` parameter with the highest possible accuracy that minimizes the *RMSE* score for the model.

Finally, we will retrain the model on the entire `edx` dataset with the best `degree` and `span` values determined by the tuning result.

#### 2.7.3.1 UMGYDE Model Tuning: Stage 1 (`degree = 0`)

##### 2.7.3.1.1 Pre-configuration: Optimal `span` Range Determination (`degree = 0`)

The complete source code of the solution described in this section is available in the UMGYDE Model Tuning: Pre-configuration (`degree = 0`) section of the capstone-movielens.main.R script on *GitHub*.

The following code snippet determines the optimal range of `span` for the further model tuning when `degree = 0`:

```
spans <- seq(0.0005, 1, 0.001)
lss.UMGYDE.preset.degree0.result <-
  tune.model_param(spans, train_UMGY_SmoothedDay_effect.RMSE.cv.degree0)
```

In the code snippet above, we use the following functions described in Appendix A:

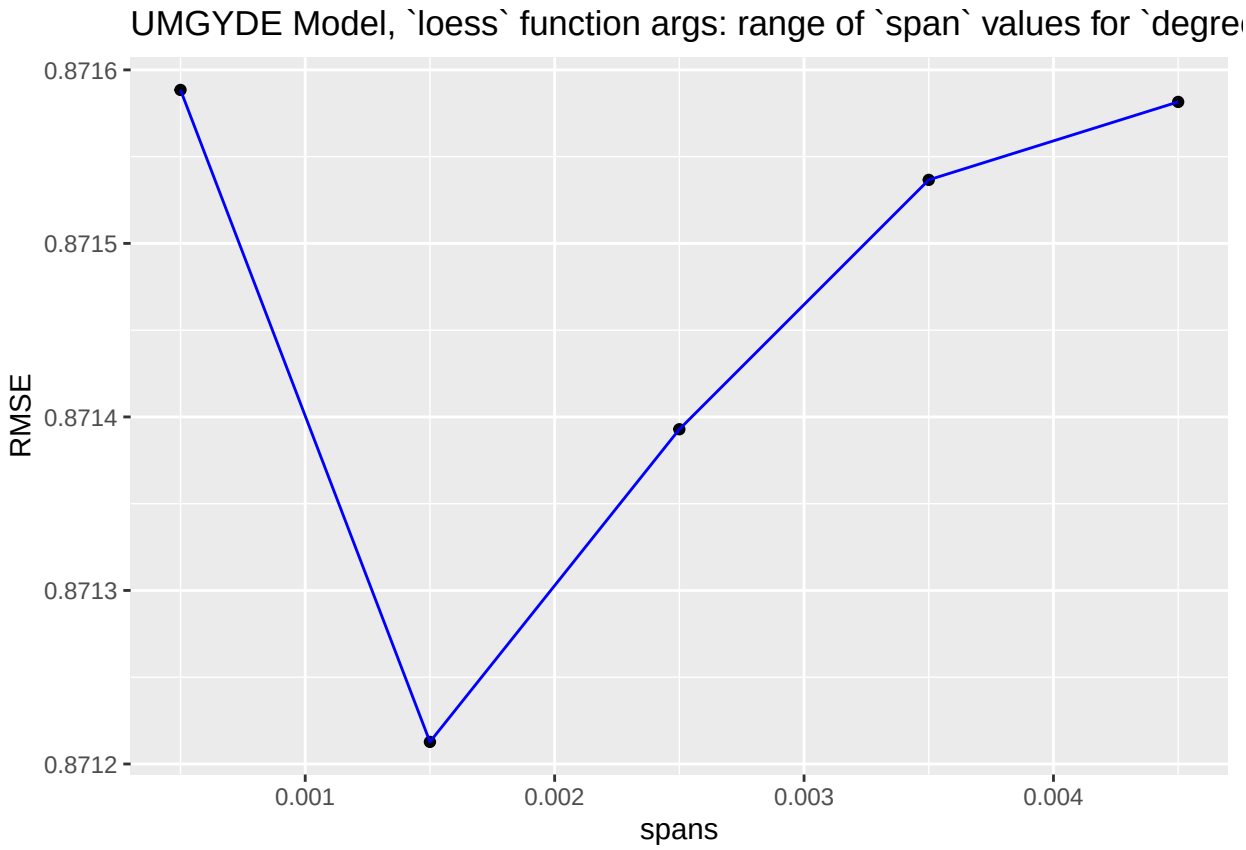
- tune.model_param (described in section Model Tuning Utils) to select the best value of the *tuning parameter* (`span` in our case) from the given sequence of the *parameter values* passed in the param_values argument.
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree0 (described in section UMGYDE Model: Tuning `loess` Params) used as an auxiliary function passed (in the fn_tune.test.param_value argument) to the tune.model_param to test the *UMGYDE Model* with the loess `degree` parameter set to 0.

```
str(lss.UMGYDE.preset.degree0.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE          : num [1:5] 0.872 0.871 0.871 0.872 0.872
##   ..$ parameter.value: num [1:5] 0.0005 0.0015 0.0025 0.0035 0.0045
##  $ best_result : Named num [1:2] 0.0015 0.8712
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

The code snippet below provides a visual representation of the dependence of the *RMSE* value on the loess `span` parameter value over the selected interval, with the parameter `degree = 0`, as we have figured out in this step of tuning:

```
plt.title = "UMGYDE Model, `loess` function args: range of `span` values for `degree = 0`"
lss.UMGYDE.preset.degree0.result$tuned.result |>
  data.plot(plt.title,
            xname = "parameter.value",
            yname = "RMSE",
            xlabel = "spans",
            ylabel = "RMSE")
```




```
rm(plt.title)
```

In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

#### 2.7.3.1.2 UMGYDE Model Fine-tuning (`degree = 0`)

The complete source code of the solution described in this section is available in the UMGYDE Model Fine-tuning (`degree = 0`) section of the capstone-movielens.main.R script on *GitHub*.

The following code snippet performs the *fine-tuning* of the *UMGYDE Model* by precise selection of the loess `span` parameter with the `degree` parameter set to 0:

```
lss.fine_tune.loop_starter <-
  c(lss.UMGYDE.preset.degree0.result$tuned.result$parameter.value[1],
    lss.UMGYDE.preset.degree0.result$tuned.result$parameter.value[3],
    8)

cache_file.base_name <- "UMGYDE.degree0.tuning-span"

lss.UMGYDE.fine_tune.degree0.result <-
  model.tune.param_range(lss.fine_tune.loop_starter,
                         UMGYDE.fine_tune.degree0.data.path,
                         cache_file.base_name,
                         train_UMGY_SmoothedDay_effect.RMSE.cv.degree0)
```

In the code snippet above, we use the following functions described in Appendix A: Support Functions:

- model.tune.param_range (described in section Model Tuning Utils) to enhance the accuracy of the *tuning parameter best value* using the *best value neighborhood* information passed in the loop_starter argument.
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree0 (already mentioned in the previous section) used as an auxiliary function passed (in the fn_tune.test.param_value argument) to the model.tune.param_range to test the *UMGYDE Model* with the loess `degree` parameter set to 0.

```
lss.fine_tune.loop_starter
```

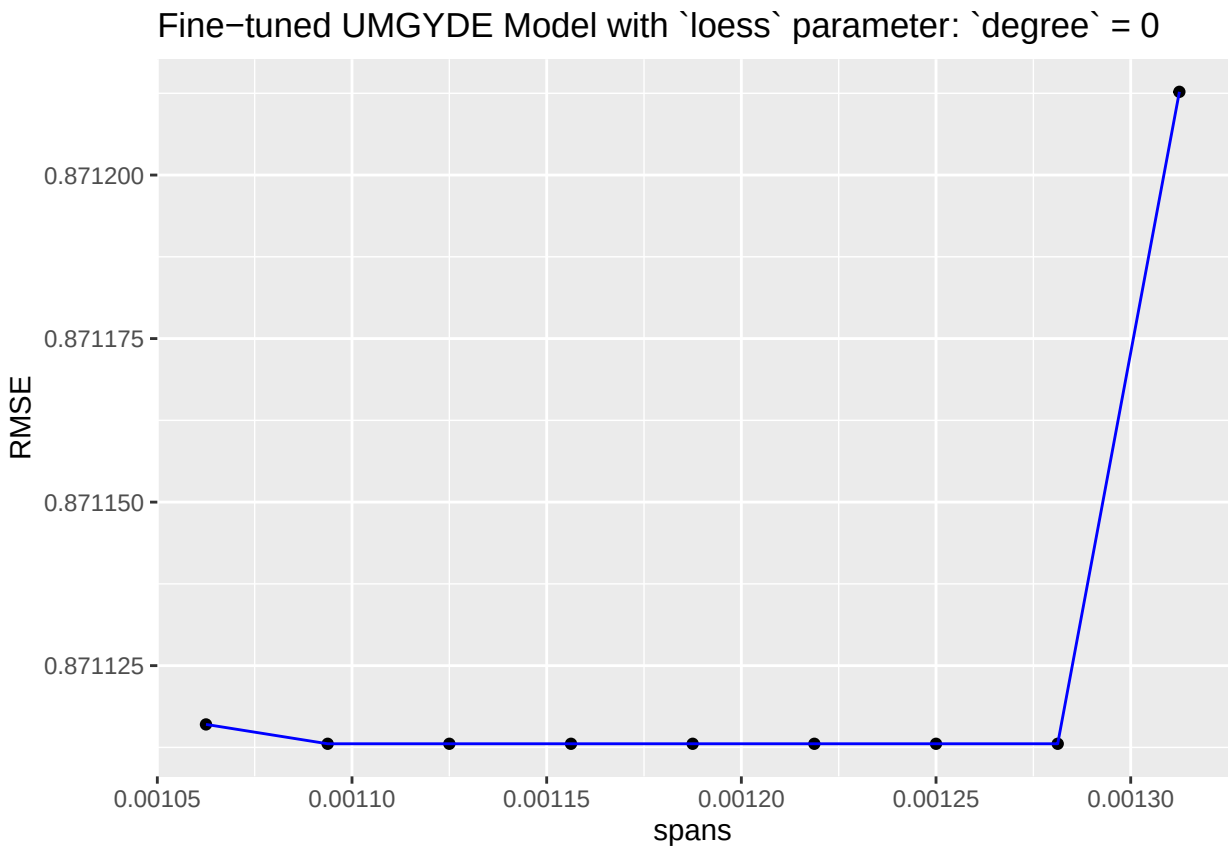
```
## [1] 0.0005 0.0025 8.0000
```

```
str(lss.UMGYDE.fine_tune.degree0.result)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 0.00109 0.87111
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: num [1:3] 1.06e-03 1.31e-03 3.13e-05
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 0.00106 0.00109 0.00113 0.00116 0.00119 ...
##   ..$ RMSE           : num [1:9] 0.871 0.871 0.871 0.871 0.871 ...
```

The code snippet below provides a visual representation of the dependence of the *RMSE* value on the loess `span` parameter value over the neighborhood of its *best value* (when the parameter `degree = 0`), which we have determined in this (*fine-tuning*) step:

```
plt.title = "Fine-tuned UMGYDE Model with `loess` parameter: `degree` = 0"

lss.UMGYDE.fine_tune.degree0.result$tuned.result |>
  data.plot(plt.title,
            xname = "parameter.value",
            yname = "RMSE",
            xlabel = "spans",
            ylabel = "RMSE")
```




```
rm(plt.title)
```

In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

Now, we can add the best *RMSE* figured out for the *UMGYDE Fine-tuned Model* with the loess `degree = 0` (let's call it *Tuned UMGYDE.d0 Model*) to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGYDE0 <- RMSEs.ResultTibble.UMGYDE |>
  RMSEs.AddRow("Tuned UMGYDE.d0 Model",
               lss.UMGYDE.fine_tune.degree0.result.best_RMSE,
               comment = "UMGYDE Model computed using function call: `loess(degree = 0, span = %1)`" |>
                 msg.glue(lss.UMGYDE.fine_tune.degree0.result.best_span))
```

```
RMSE_kable(RMSEs.ResultTibble.UMGYDE0)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for 'lambda' = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for 'lambda' = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using 'stats::loess' function with 'degree=1' & 'span=0.75' parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | UMGYDE Model computed using function call: 'loess(degree = 0, span = 0.00109375)' |

In the code snippet above we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

#### 2.7.3.2 UMGYDE Model Tuning: Stage 2 (`degree = 1`)

##### 2.7.3.2.1 Pre-configuration: Optimal `span` Range Determination (`degree = 1`)

The complete source code of the solution described in this section is available in the UMGYDE Model Tuning: Pre-configuration (`degree = 1`) section of the capstone-movielens.main.R script on *GitHub*.

Below is the code snippet that determines the optimal range of `span` for the further model tuning when `degree = 1`:

```
spans <- seq(0.0005, 1, 0.001)
lss.UMGYDE.preset.degree1.result <-
  tune.model_param(spans, train_UMGY_SmoothedDay_effect.RMSE.cv.degree1)
```

In the code snippet above, we use the following functions described in Appendix A: Support Functions:

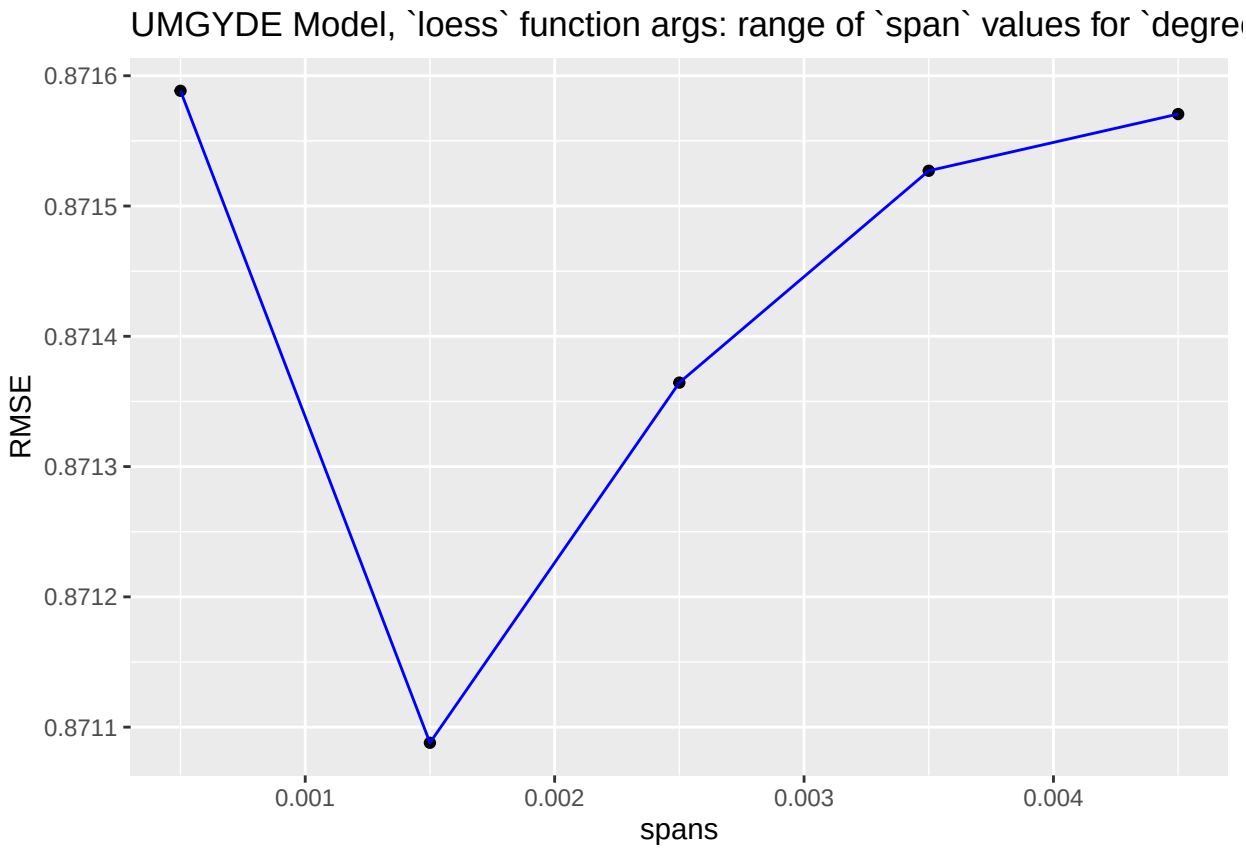
- tune.model_param (described in section Model Tuning Utils) to select the best value of the *tuning parameter* (`span` in our case) from the given sequence of the *parameter values* passed in the param_values argument.
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 (described in section UMGYDE Model: Tuning `loess` Params) used as an auxiliary function passed (in the fn_tune.test.param_value argument) to the tune.model_param to test the *UMGYDE Model* with the loess `degree` parameter set to 1.

```
str(lss.UMGYDE.preset.degree1.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE          : num [1:5] 0.872 0.871 0.871 0.872 0.872
##   ..$ parameter.value: num [1:5] 0.0005 0.0015 0.0025 0.0035 0.0045
##  $ best_result : Named num [1:2] 0.0015 0.8711
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

The code snippet below provides a visual representation of the dependence of the *RMSE* value on the loess `span` parameter value over the selected interval, with the parameter `degree = 1`, as we have figured out in this step of tuning:

```
lss.UMGYDE.preset.degree1.result$tuned.result |>
  data.plot(title = "UMGYDE Model, `loess` function args: range of `span` values for `degree = 1`",
            xname = "parameter.value",
            yname = "RMSE",
            xlabel = "spans",
            ylabel = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

#### 2.7.3.2.2 UMGYDE Model Fine-tuning (`degree = 1`)

The complete source code of the solution described in this section is available in the UMGYDE Model Fine-tuning (`degree = 1`) section of the capstone-movielens.main.R script on *GitHub*.

The following code snippet performs the *fine-tuning* of the *UMGYDE Model* by precise selection of the loess `span` parameter with the `degree` parameter set to 1:

```
lss.fine_tune.loop_starter <-
  c(lss.UMGYDE.preset.degree1.result$tuned.result$parameter.value[1],
    lss.UMGYDE.preset.degree1.result$tuned.result$parameter.value[3],
    8)

cache_file.base_name <- "UMGYDE.degree1.tuning-span"

lss.UMGYDE.fine_tune.degree1.result <-
  model.tune.param_range(lss.fine_tune.loop_starter,
                         UMGYDE.fine_tune.degree1.data.path,
                         cache_file.base_name,
                         train_UMGY_SmoothedDay_effect.RMSE.cv.degree1)
```

In the code snippet above, we use the following functions described in Appendix A:

- model.tune.param_range (described in section Model Tuning Utils) to enhance the accuracy of the *tuning parameter best value* using the *best value neighborhood* information passed in the loop_starter argument.
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 (already mentioned in the previous section) used as an auxiliary function passed (in the fn_tune.test.param_value argument) to the model.tune.param_range to test the *UMGYDE Model* with the loess `degree` parameter set to 1.

```
lss.fine_tune.loop_starter
```

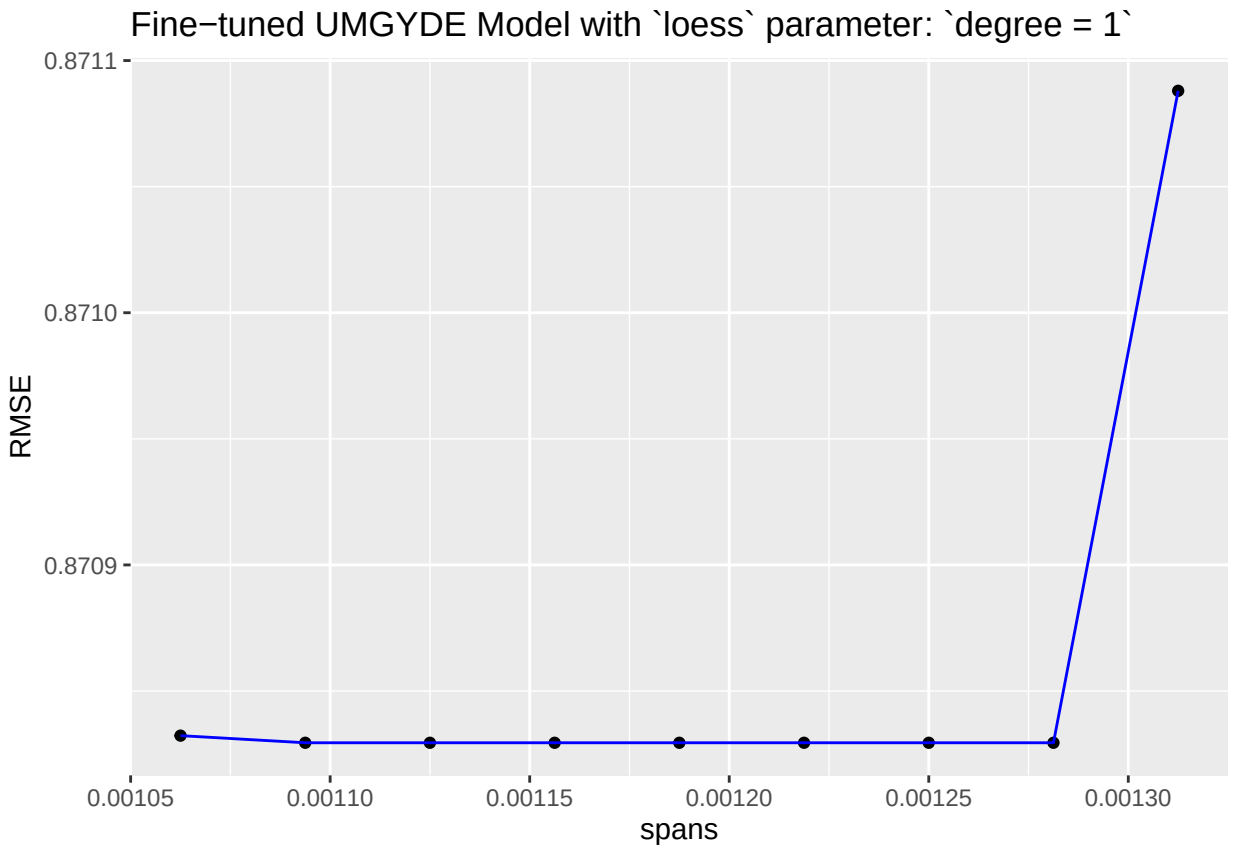
```
## [1] 0.0005 0.0025 8.0000
```

```
str(lss.UMGYDE.fine_tune.degree1.result)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 0.00109 0.87083
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: num [1:3] 1.06e-03 1.31e-03 3.13e-05
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 0.00106 0.00109 0.00113 0.00116 0.00119 ...
##   ..$ RMSE           : num [1:9] 0.871 0.871 0.871 0.871 0.871 ...
```

The code snippet below provides a visual representation of the dependence of the *RMSE* value on the loess `span` parameter value over the neighborhood of its *best value* (when the parameter `degree = 1`), which we have determined in this (*fine-tuning*) step:

```
lss.UMGYDE.fine_tune.degree1.result$tuned.result |>
  data.plot(title = "Fine-tuned UMGYDE Model with `loess` parameter: `degree = 1`",
                                xname = "parameter.value",
                                yname = "RMSE",
                                xlabel = "spans",
                                ylabel = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

Now, we can add the best *RMSE* figured out for the *UMGYDE Fine-tuned Model* with the loess `degree = 1` (let's call it *Tuned UMGYDE.d1 Model*) to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGYDE1 <- RMSEs.ResultTibble.UMGYDE0 |>
  RMSEs.AddRow("Tuned UMGYDE.d1 Model",
               lss.UMGYDE.fine_tune.degree1.result.best_RMSE,
               comment = "UMGYDE Model computed using function call: `loess(degree = 1, span = %1)`" |>
                 msg.glue(lss.UMGYDE.fine_tune.degree1.result.best_span))
```

```
RMSE_kable(RMSEs.ResultTibble.UMGYDE1)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for 'lambda' = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for 'lambda' = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using 'stats::loess' function with 'degree=1' & 'span=0.75' parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | UMGYDE Model computed using function call: 'loess(degree = 0, span = 0.00109375)' |
| Tuned UMGYDE.d1 Model | 0.8708295 | UMGYDE Model computed using function call: 'loess(degree = 1, span = 0.00109375)' |

In the code snippet above we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

#### 2.7.3.3 UMGYDE Model Tuning: Stage 3 (`degree = 2`)

##### 2.7.3.3.1 Pre-configuration: Optimal `span` Range Determination (`degree = 2`)

The complete source code of the solution described in this section is available in the UMGYDE Model Tuning: Pre-configuration (`degree = 2`) section of the capstone-movielens.main.R script on *GitHub*.

Below is the code snippet that determines the optimal range of `span` for the further model tuning when `degree = 2`:

```
spans <- seq(0.0005, 1, 0.001)
lss.UMGYDE.preset.degree2.result <-
  tune.model_param(spans, train_UMGY_SmoothedDay_effect.RMSE.cv.degree2)
```

In the code snippet above, we use the following functions described in Appendix A:

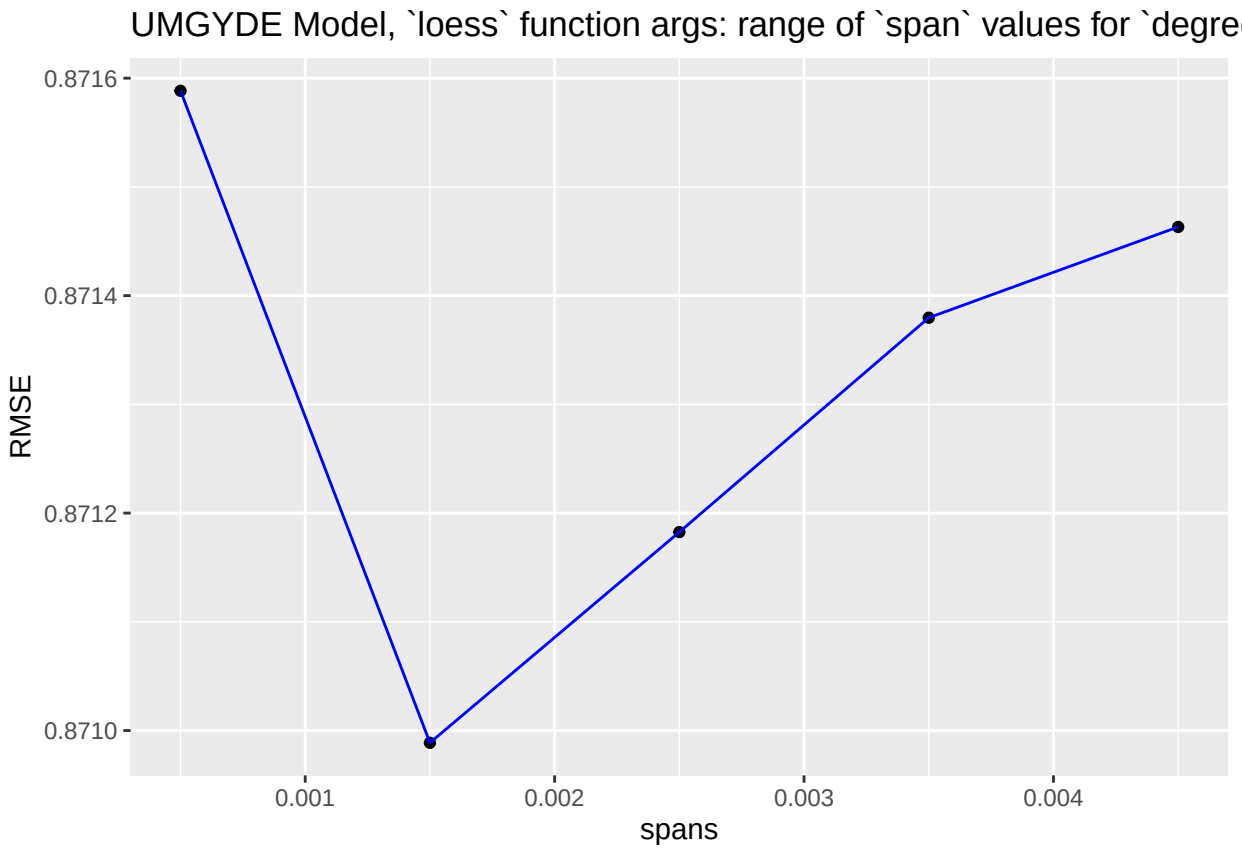
- tune.model_param (described in section Model Tuning Utils) to select the best value of the *tuning parameter* (`span` in our case) from the given sequence of the *parameter values* passed in the param_values argument.
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree2 (described in section UMGYDE Model: Tuning `loess` Params) used as an auxiliary function passed (in the fn_tune.test.param_value argument) to the tune.model_param to test the *UMGYDE Model* with the loess `degree` parameter set to 2.

```
str(lss.UMGYDE.preset.degree2.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE          : num [1:5] 0.872 0.871 0.871 0.871 0.871
##   ..$ parameter.value: num [1:5] 0.0005 0.0015 0.0025 0.0035 0.0045
##  $ best_result : Named num [1:2] 0.0015 0.871
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

The code snippet below provides a visual representation of the dependence of the *RMSE* value on the loess `span` parameter value over the selected interval, with the parameter `degree = 2`, as we have figured out in this step of tuning:

```
lss.UMGYDE.preset.degree2.result$tuned.result |>
  data.plot(title = "UMGYDE Model, `loess` function args: range of `span` values for `degree = 2`",
            xname = "parameter.value",
            yname = "RMSE",
            xlabel = "spans",
            ylabel = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

#### 2.7.3.3.2 UMGYDE Model Fine-tuning (`degree = 2`)

The complete source code of the solution described in this section is available in the UMGYDE Model Fine-tuning (`degree = 2`) section of the capstone-movielens.main.R script on *GitHub*.

The following code snippet performs the *fine-tuning* of the *UMGYDE Model* by precise selection of the loess `span` parameter with the `degree` parameter set to 2:

```
lss.fine_tune.loop_starter <-
  c(lss.UMGYDE.preset.degree2.result$tuned.result$parameter.value[1],
    lss.UMGYDE.preset.degree2.result$tuned.result$parameter.value[3],
    8)

cache_file.base_name <- "UMGYDE.degree2.tuning-span"

lss.UMGYDE.fine_tune.degree2.result <-
  model.tune.param_range(lss.fine_tune.loop_starter,
                         UMGYDE.fine_tune.degree2.data.path,
                         cache_file.base_name,
                         train_UMGY_SmoothedDay_effect.RMSE.cv.degree2)
```

In the code snippet above, we use the following functions described in Appendix A: Support Functions:

- model.tune.param_range (described in section Model Tuning Utils) to enhance the accuracy of the *tuning parameter best value* using the *best value neighborhood* information passed in the loop_starter argument.
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree2 (already mentioned in the previous section) used as an auxiliary function passed (in the fn_tune.test.param_value argument) to the model.tune.param_range to test the *UMGYDE Model* with the loess `degree` parameter set to 2.

```
lss.fine_tune.loop_starter
```

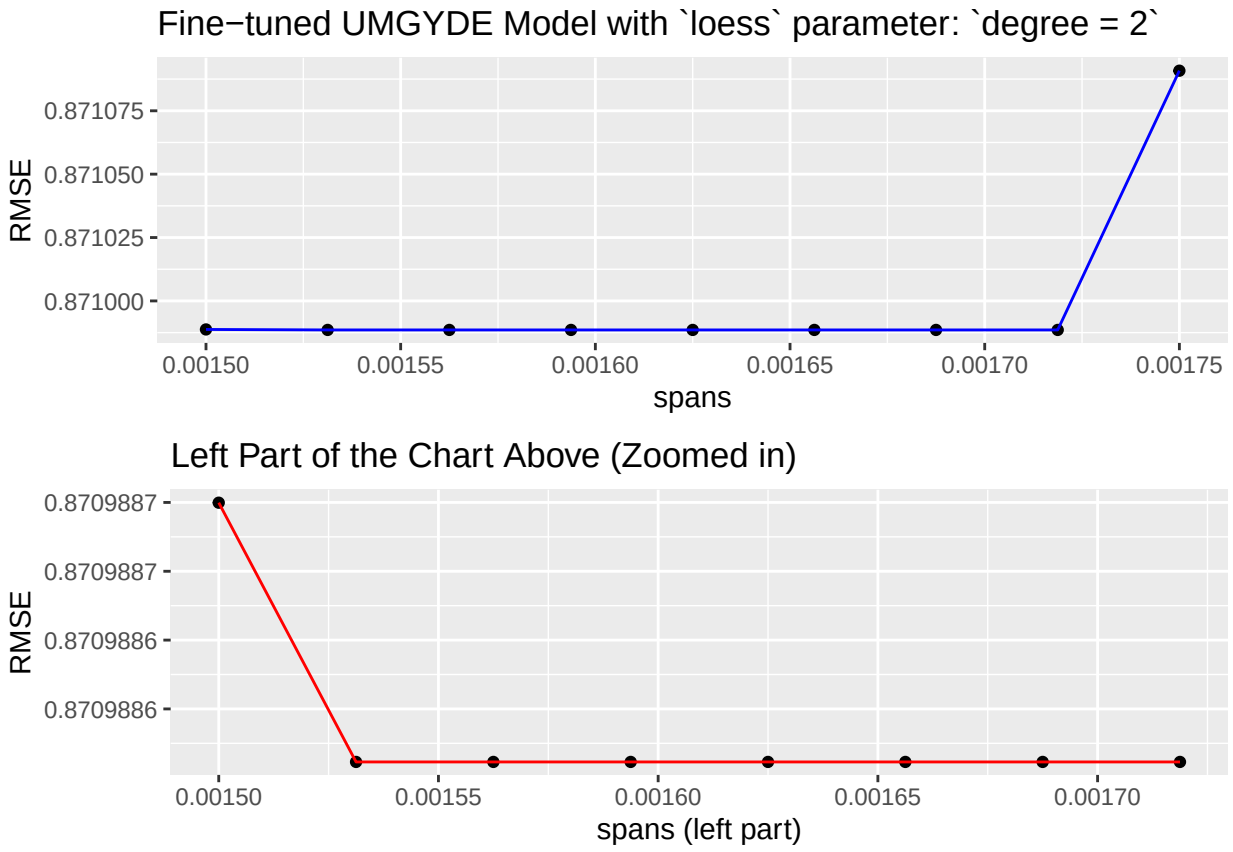
```
## [1] 0.0005 0.0025 8.0000
```

```
str(lss.UMGYDE.fine_tune.degree2.result)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 0.00153 0.87099
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: num [1:3] 1.50e-03 1.75e-03 3.13e-05
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 0.0015 0.00153 0.00156 0.00159 0.00163 ...
##   ..$ RMSE           : num [1:9] 0.871 0.871 0.871 0.871 0.871 ...
```

The code snippet below provides a visual representation of the dependence of the *RMSE* value on the loess `span` parameter value over the neighborhood of its *best value* (when the parameter `degree = 2`), which we have determined in this (*fine-tuning*) step. Since the left endpoint value of the *RMSE* range has much smaller deviation from the minimum than the value of the right endpoint, for visual clarity, we plot the left part of the graph once more on an *enlarged scale* (below the main graph):

```
lss.UMGYDE.fine_tune.degree2.result$tuned.result |>
  data.plot.left_detailed(title = "Fine-tuned UMGYDE Model with `loess` parameter: `degree = 2`",
                          title.left = "Left Part of the Chart Above (Zoomed in)",
                          left.n = 8,
                          xname = "parameter.value",
                          yname = "RMSE",
                          xlabel1 = "spans",
                          ylabel1 = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot.left_detailed described in section Data Visualization Functions of Appendix A.

Now, we can add the best *RMSE* value figured out for the *UMGYDE Fine-tuned Model* with the loess `degree = 2` (let's call it *Tuned UMGYDE.d2 Model*) to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGYDE2 <- RMSEs.ResultTibble.UMGYDE1 |>
  RMSEs.AddRow("Tuned UMGYDE.d2 Model",
               lss.UMGYDE.fine_tune.degree2.result.best_RMSE,
               comment = "UMGYDE Model computed using function call: `loess(degree = 2, span = %1)`" |>
                 msg.glue(lss.UMGYDE.fine_tune.degree2.result.best_span))
```

```
RMSE_kable(RMSEs.ResultTibble.UMGYDE2)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for 'lambda' = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for 'lambda' = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using 'stats::loess' function with 'degree=1' & 'span=0.75' parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | UMGYDE Model computed using function call: 'loess(degree = 0, span = 0.00109375)' |
| Tuned UMGYDE.d1 Model | 0.8708295 | UMGYDE Model computed using function call: 'loess(degree = 1, span = 0.00109375)' |
| Tuned UMGYDE.d2 Model | 0.8709886 | UMGYDE Model computed using function call: 'loess(degree = 2, span = 0.00153125)' |

In the code snippet above we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

#### 2.7.3.4 UMGYDE Model Tuning: Re-training on the edx with the Best Params



The complete version of the source code provided in this section are available in the UMGYDE Tuned Model: Retraining with the best params section of the capstone-movielens.main.R script on *GitHub*.

Now, we can refine the current model by retraining on the entire `edx` dataset with the best values of the loess parameters `degree` and `span` we just figured out (let's call it *Tuned UMGYDE Best Model*), for the definitive *RMSE* calculation and use in subsequent models:

The following code snippet performs this operation:

```
# The Best Parameters and RMSE Value
lss.best_results <- data.frame(degree = degree,
                          span = c(lss.UMGYDE.fine_tune.degree0.result.best_span,
                                   lss.UMGYDE.fine_tune.degree1.result.best_span,
                                   lss.UMGYDE.fine_tune.degree2.result.best_span),

                          RMSE = c(lss.UMGYDE.fine_tune.degree0.result.best_RMSE,
                                   lss.UMGYDE.fine_tune.degree1.result.best_RMSE,
                                   lss.UMGYDE.fine_tune.degree2.result.best_RMSE))

lss.best_RMSE.idx <- which.min(lss.best_results$RMSE)


lss.UMGYDE.best_params <-
  c(degree =  lss.best_results[lss.best_RMSE.idx, "degree"],  # 1
    span = lss.best_results[lss.best_RMSE.idx, "span"], # 0.00087,
    RMSE = lss.best_results[lss.best_RMSE.idx, "RMSE"]) # 0.8568619

lss.best_degree <- lss.UMGYDE.best_params["degree"]
lss.best_span <- lss.UMGYDE.best_params["span"]
lss.best_RMSE <- lss.UMGYDE.best_params["RMSE"]

put_log2("Re-training model using `loess` function with the best parameters:
span = %1, degree = %2", lss.best_span, lss.best_degree)

lss.UMGYD_effect <- edx |>
  train_UMGY_SmoothedDay_effect(lss.best_degree, lss.best_span)
```



In the code snippet above we use the train_UMGY_SmoothedDay_effect custom helper function described in section UMGYDE Model: Utility Functions of Appendix A.

```
lss.UMGYDE.best_params
```

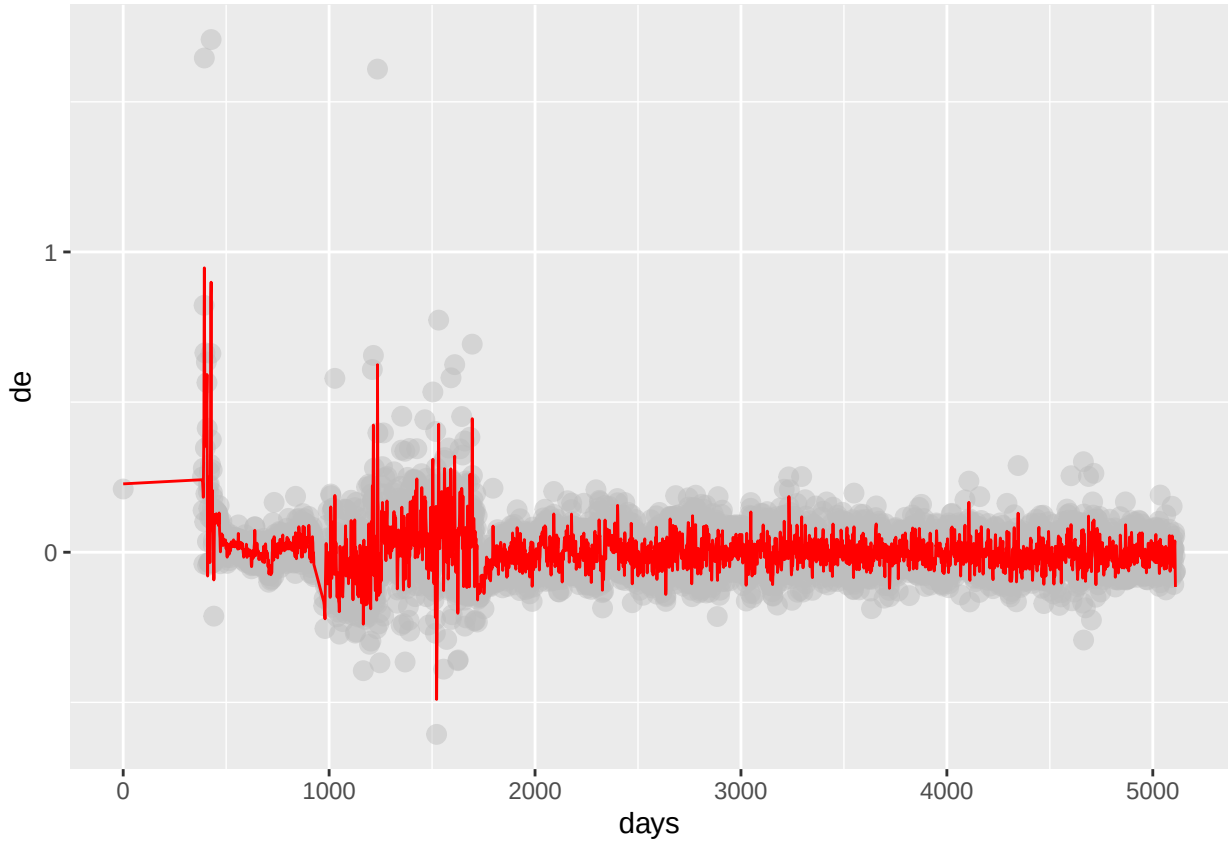
```
##     degree       span       RMSE
## 1.00000000 0.00109375 0.87082947
```

```
str(lss.UMGYD_effect)
```

```
## tibble [4,640 x 4] (S3: tbl_df/tbl/data.frame)
##  $ days       : int [1:4640] 0 385 388 389 392 393 394 396 397 399 ...
##  $ de         : num [1:4640] 0.2106 0.2506 0.1395 0.2797 -0.0392 ...
##  $ year       : num [1:4640] 1995 1996 1996 1996 1996 ...
##  $ de_smoothed: num [1:4640] 0.228 0.242 0.205 0.183 0.44 ...
```

The following code snippet provides a visual representation of the final *Smoothed Day Effect* data we have just obtained:

```
lss.UMGYD_effect |>
  ggplot(aes(x = days)) +
  geom_point(aes(y = de), size = 3, alpha = .5, color = "grey") +
  geom_line(aes(y = de_smoothed), color = "red")
```




It should be noted, however, that the graph above is not as smooth as when training the model with default [loess] parameters (`degree` and `span`), although we have now obtained a better (*RMSE*) result.

Now, we are ready to construct predictors and calculate the *RMSE* score for the ultimately *Tuned UMGYDE Best Model* using the following line of code:

```
lss.UMGYD_effect.RMSE <- calc_UMGY_SmoothedDay_effect.RMSE.cv(lss.UMGYD_effect)
```

```
lss.UMGYD_effect.RMSE
```

```
## [1] 0.870785
```

In the code snippets above, we use the calc_UMGY_SmoothedDay_effect.RMSE.cv custom helper function described in section UMGYDE Model: Utility Functions of Appendix A.

Finally, we add the *RMSE* value obtained above for the fully tuned model (let's call it *Tuned UMGYDE Best Model*) to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGYDE.tuned <- RMSEs.ResultTibble.UMGYDE2 |>
  RMSEs.AddRow("Tuned UMGYDE Best Model",
               lss.UMGYD_effect.RMSE,
               comment = "UMGYDE Model computed using `loess` function call with the best degree & span
```

```
RMSE_kable(RMSEs.ResultTibble.UMGYDE.tuned)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for 'lambda' = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for 'lambda' = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using 'stats::loess' function with 'degree=1' & 'span=0.75' parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | UMGYDE Model computed using function call: 'loess(degree = 0, span = 0.00109375)' |
| Tuned UMGYDE.d1 Model | 0.8708295 | UMGYDE Model computed using function call: 'loess(degree = 1, span = 0.00109375)' |
| Tuned UMGYDE.d2 Model | 0.8709886 | UMGYDE Model computed using function call: 'loess(degree = 2, span = 0.00153125)' |
| Tuned UMGYDE Best Model | 0.8707850 | UMGYDE Model computed using 'loess' function call with the best degree & span values. |

In the code snippet above we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

Now, as the final stage of the tuning process, let's *regularize* our model (as we did with the previous models).

### 2.7.4 UMGYDE Model Regularization

The complete version of the source code provided in this section can be found in the UMGYDE Model Regularization section of the capstone-movielens.main.R script.

We begin this section with the concept's mathematical description presented below in the subsection UMGYDE Model Regularization: *Mathematical Description*.

Next, we will implement the *UMGYDE Model Regularization* in the following three steps:

1. **Pre-configuration:** (Described in subsection UMGYDE Model Regularization: *Pre-configuration*) Preliminary determination of the optimal range of *regularization parameter* $\lambda$ values for the *K-Fold Cross-Validation* samples, where the $K$ is the length of the `edx_CV` Object (described in detail in Appendix B: Models Training Datasets) to which the *K-Fold Cross-Validation* is applied (in *this Project*, we use $K = 5$);
2. **Fine-tuning:** (Described in the subsection UMGYDE Model Regularization: *Fine-tuning*) Determining the best value of $\lambda$ with the highest possible accuracy that minimizes the *RMSE* score for the model.
3. **Retraining:** (Described in subsection UMGYDE Model Regularization: *Retraining on the `edx` with the best* $\lambda$) Retraining the model on the entire `edx` dataset with the best value of $\lambda$ determined in the previous step.

#### 2.7.4.1 UMGYDE Model Regularization: *Mathematical Description*

We have already explained the idea of the *Linear Model Regularization* earlier in section UME Model Regularization. We have also seen how the formula (1) for adding a penalty to the *UME Model* is transformed into the formula (5) for the *UMGE Model* and then into the formula (9) for the *UMGYE Model*. For the current model, this formula takes the form:

$$\sum_{i,j} \left(y_{u,i} - \mu - \alpha_i - \beta_j - g_{i,j} - \gamma(v_{i,j}) - s(d_{i,j})\right)^2 + \lambda \sum_{i,j} s(d_{i,j})^2 \tag{13}$$

And the formula (10) for calculating the values of the *treatment effect* that minimizes the equation will take the form:

$$\hat{s}(d_{i,j}, \lambda) = \frac{1}{\lambda + n_d} \sum_{r=1}^{n_d} \left(Y_{i,j} - \mu - \alpha_i - \beta_j - g_{i,j} - \gamma(v_{i,j})\right) \tag{14}$$

where $n_d$ is the number of ratings made on the day $d$.

#### 2.7.4.2 UMGYDE Model Regularization: *Pre-configuration*

The complete version of the source code shown in this section is available in the UMGYDE Model Regularization: Pre-configuration section of the capstone-movielens.main.R script on *GitHub*.

As for previous models, we will use the function tune.model_param passing in the fn_tune.test.param_value argument the model-specific helper function (this time designed for the *UMGYDE Model*): regularize.test_lambda.UMGYD_effect.cv.

The functions tune.model_param and regularize.test_lambda.UMGYD_effect.cv are described in the sections Model Tuning Utils and UMGYE Model: Regularization, respectively, of Appendix A.

The following piece of code performs this operation:

```
  lambdas <- seq(0, 256, 16)
  cv.UMGYDE.preset.result <-
    tune.model_param(lambdas, regularize.test_lambda.UMGYD_effect.cv)
```

```
str(cv.UMGYDE.preset.result)
```

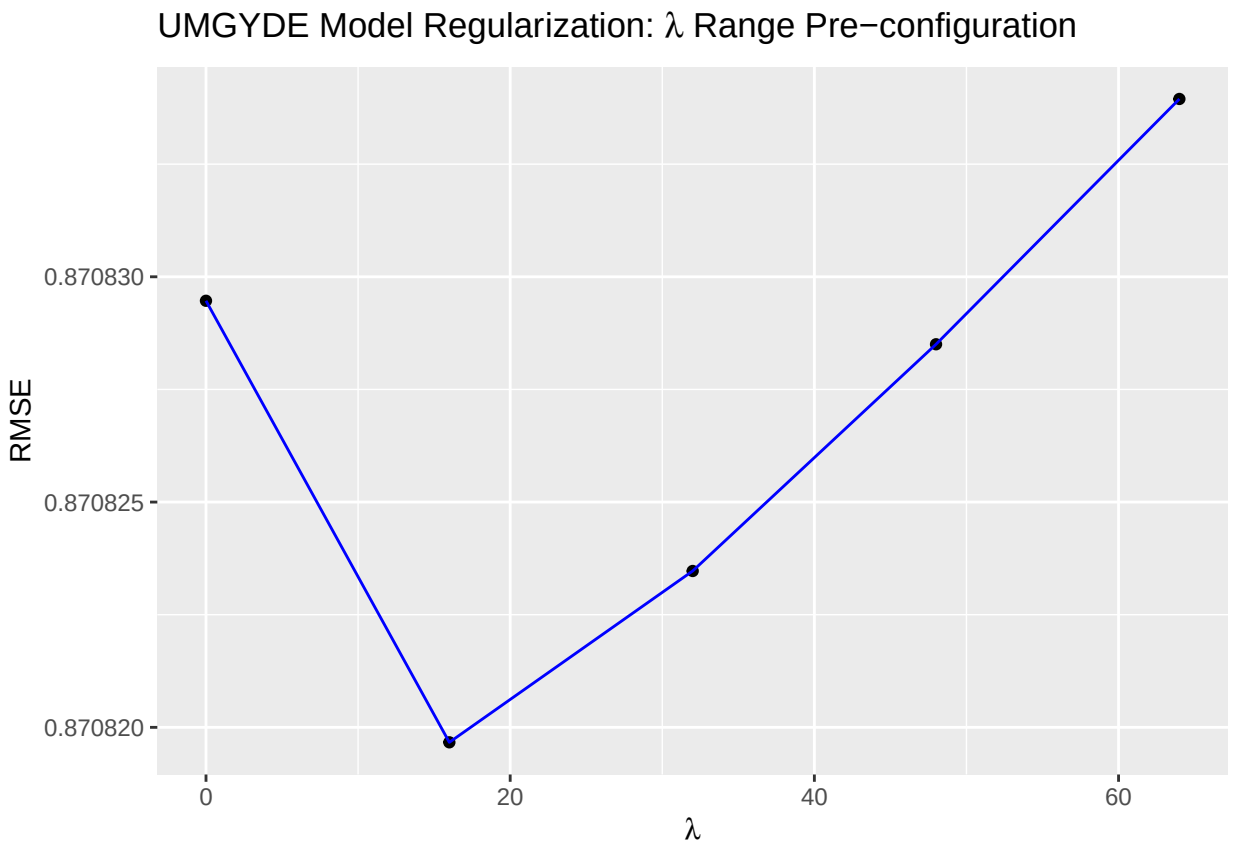
```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE           : num [1:5] 0.871 0.871 0.871 0.871 0.871
##   ..$ parameter.value: num [1:5] 0 16 32 48 64
##  $ best_result : Named num [1:2] 16 0.871
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

```
cv.UMGYDE.preset.result$best_result
```

```
## param.best_value        best_RMSE
##       16.0000000        0.8708197
```

Now, let's visualize the results of the $\lambda$ range pre-configuration using the following code snippet:

```
cv.UMGYDE.preset.result$tuned.result |>
  data.plot(title = TeX(r'[UMGYDE Model Regularization: $\lambda$ Range Pre-configuration]'),
            xname = "parameter.value",
            yname = "RMSE",
            xlabel = TeX(r'[$\lambda$]'),
            ylabel = "RMSE")
```




In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of Appendix A.

#### 2.7.4.3 UMGYDE Model Regularization: *Fine-tuning*

The complete version of the source code shown in this section can be found in the UMGYDE Model Regularization: Fine-tuning section of the capstone-movielens.main.R script on *GitHub*.

We are now ready to perform the fine-tuning step of our model *regularization* process to determine the best value for the $\lambda$ parameter.

The following piece of code prepares the interval of values for the $\lambda$ parameter, over which the operation has to be done:

```
endpoints <-
  get_fine_tune.param.endpoints(cv.UMGYDE.preset.result$tuned.result)

UMGYDE.loop_starter <- c(endpoints["start"],
                         endpoints["end"],
                         8)
```

The helper function get_fine_tune.param.endpoints used in code snippet above is described in the sections Model Tuning Utils of Appendix A.

```
## *** Values of the endpoints and the divisor for the interval of `lambda` values ***
```

```
UMGYDE.loop_starter
```

```
## start   end
##     0    32     8
```

And the next code snippet below accomplishes the task of *fine-tuning* the model:

```
cache.base_name <- "UMGYDE.rglr.fine-tuning"

UMGYDE.rglr.fine_tune.results <-
  model.tune.param_range(UMGYDE.loop_starter,
                         UMGYDE.rglr.fine_tune.cache.path,
                         cache.base_name,
                         regularize.test_lambda.UMGYD_effect.cv)

UMGYDE.rglr.fine_tune.RMSE.best <- UMGYDE.rglr.fine_tune.results$best_result["best_RMSE"]
```

The custom functions model.tune.param_range and regularize.test_lambda.UMGYD_effect.cv used in the code snippet above are described in the sections Model Tuning Utils and UMGYDE Model: Regularization, respectively, of Appendix A.

Below are the results of fine-tuning the *UMGYDE Model*:

```
## *** Path to the cache directory for intermediate fine-tuning results ***
```

```
UMGYDE.rglr.fine_tune.cache.path
```

```
## [1] "data/regularization/4.UMGYD-effect/fine-tune"
```

```
## *** Fine-tuning results object data structure ***
```

```
str(UMGYDE.rglr.fine_tune.results)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 10.714 0.871
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: Named num [1:3] 1.07e+01 1.07e+01 1.53e-05
##   ..- attr(*, "names")= chr [1:3] "" "" ""
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 10.7 10.7 10.7 10.7 10.7 ...
##   ..$ RMSE           : num [1:9] 0.871 0.871 0.871 0.871 0.871 ...
```

```
## *** Fine-tuning: best results ***
```

```
UMGYDE.rglr.fine_tune.results$best_result
```

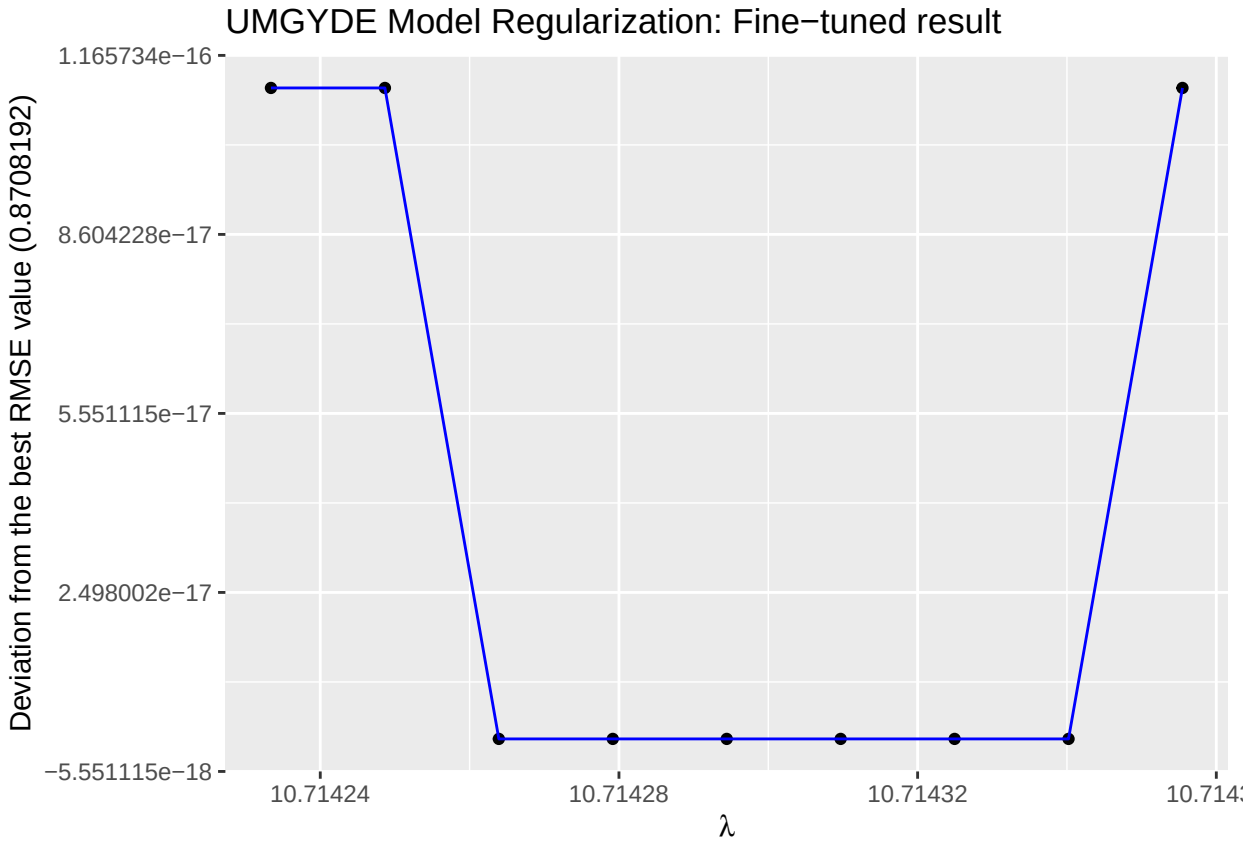
```
## param.best_value        best_RMSE
##       10.7142639        0.8708192
```

```
UMGYDE.rglr.fine_tune.RMSE.best
```

```
## best_RMSE
## 0.8708192
```

The following code snippet provides a visual representation of the *fine-tuning results* we have just computed:

```
UMGYDE.rglr.fine_tune.results$tuned.result |>
  data.plot(title = "UMGYDE Model Regularization: Fine-tuned result",
              xname = "parameter.value",
              yname = "RMSE",
              xlabel = TeX(r'[$\lambda$]'),
              ylabel = str_glue("Deviation from the best RMSE value (",
                                as.character(round(UMGYDE.rglr.fine_tune.RMSE.best, digits = 7)),
                                ")"),
              normalize = TRUE)
```




Note that in the code snippet above, we use the custom data visualization function data.plot (described in section Data Visualization Functions of Appendix A) with the argument normalize set to `TRUE`, which means that deviations from the minimum $y$ value are used to plot, rather than the $y$ values themselves.

#### 2.7.4.4 UMGYDE Model Regularization: *Retraining on the* `edx` *with the best* $\lambda$

The complete version of the source code shown in this section is available in the UMGYDE Model Regularization: Retraining with the best params section of the capstone-movielens.main.R script on *GitHub*.

Now, we can refine the *UMGYDE Model* by retraining on the entire `edx` dataset with the best value of the $\lambda$ parameter we just figured out (let's call it *Regularized UMGYDE Model*), for the definitive *RMSE* calculation and use in subsequent models:

The following code snippet performs this operation:

```
best_result <- UMGYDE.rglr.fine_tune.results$best_result
UMGYDE.rglr.best_lambda <- best_result["param.best_value"]

put_log1("Re-training Regularized User+Movie+Genre+Year+(Smoothed)Day Effect Model for the best `lambda
         UMGYDE.rglr.best_lambda)

rglr.UMGYD_effect <- edx |>
  regularize.train_UMGYD_effect(UMGYDE.rglr.best_lambda)
```

In the code snippet above we use the regularize.train_UMGYD_effect function described in section UMGYDE Model: Regularization of Appendix A.

```
## *** The Best UMGYD Effect Fine-tuning Results ***
```

```
UMGYDE.rglr.fine_tune.results$best_result
```

```
## param.best_value        best_RMSE
##       10.7142639        0.8708192

## *** Regularized UMGYD Effect Structure ***
```

```
  str(rglr.UMGYD_effect)
```

```
## tibble [4,640 x 4] (S3: tbl_df/tbl/data.frame)
##  $ days       : int [1:4640] 0 385 388 389 392 393 394 396 397 399 ...
##  $ de         : Named num [1:4640] 0.0331 0.1537 0.1174 0.239 -0.0277 ...
##   ..- attr(*, "names")= chr [1:4640] "param.best_value" "param.best_value" "param.best_value" "param
##  $ year       : num [1:4640] 1995 1996 1996 1996 1996 ...
##  $ de_smoothed: num [1:4640] 0.025 0.1492 0.1638 0.1565 0.0627 ...
```

```
print_log1("Regularized UMGYDE Model has been re-trained for the best `lambda`: %1.",
         UMGYDE.rglr.best_lambda)
```

```
## Regularized UMGYDE Model has been re-trained for the best ‘lambda‘: 10.7142639160156.
```

Now, we are ready to construct predictors and calculate the *RMSE* score for the ultimately *Regularized UMGYE Model* using the following line of code:

```
rglr.UMGYD_effect.RMSE <- calc_UMGY_SmoothedDay_effect.RMSE.cv(rglr.UMGYD_effect)
```

In the code snippets above, we use the calc_UMGY_SmoothedDay_effect.RMSE.cv user-defined described in section UMGYDE Model: Utility Functions of Appendix A.

Finally, we add the definitive *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
RMSEs.ResultTibble.UMGYDE.rglr.tuned <- RMSEs.ResultTibble.UMGYDE.tuned |>
  RMSEs.AddRow("Regularized UMGYDE Model",
               rglr.UMGYD_effect.RMSE,
               comment = "The best tuned and regularized UMGYDE Model.")
```

```
RMSE_kable(RMSEs.ResultTibble.UMGYDE.rglr.tuned)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for ‘lambda‘ = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for ‘lambda‘ = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for ‘lambda‘ = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using ‘stats::loess‘ function with ‘degree=1‘ & ‘span=0.75‘ parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | UMGYDE Model computed using function call: ‘loess(degree = 0, span = 0.00109375)‘ |
| Tuned UMGYDE.d1 Model | 0.8708295 | UMGYDE Model computed using function call: ‘loess(degree = 1, span = 0.00109375)‘ |
| Tuned UMGYDE.d2 Model | 0.8709886 | UMGYDE Model computed using function call: ‘loess(degree = 2, span = 0.00153125)‘ |
| Tuned UMGYDE Best Model | 0.8707850 | UMGYDE Model computed using ‘loess‘ function call with the best degree & span values. |
| Regularized UMGYDE Model | 0.8707750 | The best tuned and regularized UMGYDE Model. |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

Now, we have a better result than the ones for the previous models, but still insufficient to meet the Project Objective. Despite this, let’s conduct our very first *final holdout test* to see what we have actually achieved so far.

#### 2.7.4.5 UMGYDE Model: Final Holdout Test (Preliminary Assessment)

The complete version of the source code shown in this section is available in the UMGYDE Model: Final Holdout Test (Preliminary Assessment) section of the capstone-movielens.main.R script on *GitHub*.

Let's see what we have achieved so far, using the `final_holdout_test` dataset for the first time.

First, we compute predictions for our current model using the following code snippet:

```
final.UMGYDE.predicted <- final_holdout_test |>
  UMGY_SmoothedDay_effect.predict(rglr.UMGYD_effect)
```

In the code snippet above we use the UMGY_SmoothedDay_effect.predict function described in section UMGYDE Model: Utility Functions of Appendix A.

```
str(final.UMGYDE.predicted)
```

```
## 'data.frame':    999999 obs. of  5 variables:
##  $ userId   : int  1 1 1 2 2 2 3 3 4 4 ...
##  $ movieId  : int  231 480 586 151 858 1544 590 4995 34 432 ...
##  $ timestamp: int  838983392 838983653 838984068 868246450 868245645 868245920 1136075494 1133571200
##  $ rating   : num  5 5 5 3 2 3 3.5 4.5 5 3 ...
##  $ predicted: Named num  4.49 5 4.63 3.3 4.07 ...
##   ..- attr(*, "names")= chr [1:999999] "param.best_value" "param.best_value" "param.best_value" "para
```

Then we compute the *RMSE* score using the following code snippet:

```
final.UMGYDE.predicted.RMSE <- rmse2(final_holdout_test$rating,
                                     final.UMGYDE.predicted$predicted)
```

In the code snippet above, we use the rmse2 user-defined function described in section (Root) Mean Squared Error Calculation of Appendix A.

Finally, we add the *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
final.RMSEs.ResultTibble.UMGYDE.rglr.tuned <- RMSEs.ResultTibble.UMGYDE.rglr.tuned |>
  RMSEs.AddRow("Best UMGYDE Model (Final Test)",
               final.UMGYDE.predicted.RMSE,
               comment = "Final Holdout Test of the best tuned and regularized UMGYDE Model.")
```

```
RMSE_kable(final.RMSEs.ResultTibble.UMGYDE.rglr.tuned)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for 'lambda' = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for 'lambda' = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for 'lambda' = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using 'stats::loess' function with 'degree=1' & 'span=0.75' parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | UMGYDE Model computed using function call: 'loess(degree = 0, span = 0.00109375)' |
| Tuned UMGYDE.d1 Model | 0.8708295 | UMGYDE Model computed using function call: 'loess(degree = 1, span = 0.00109375)' |
| Tuned UMGYDE.d2 Model | 0.8709886 | UMGYDE Model computed using function call: 'loess(degree = 2, span = 0.00153125)' |
| Tuned UMGYDE Best Model | 0.8707850 | UMGYDE Model computed using 'loess' function call with the best degree & span values. |
| Regularized UMGYDE Model | 0.8707750 | The best tuned and regularized UMGYDE Model. |
| Best UMGYDE Model (Final Test) | 0.8804225 | Final Holdout Test of the best tuned and regularized UMGYDE Model. |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

As expected, we have not yet achieved a result that meets our Project Objective. Let's see what else we can do to achieve our ultimate goal.

As explained in Section 33.11 `Matrix Factorization` of the *Course Textbook (First Edition)*, so far our models ignore *"an important source of variation related to the fact that groups of movies have similar rating patterns and groups of users have similar rating patterns as well..."* [18]

The author shows that in this case, the *Matrix Factorization* method can significantly improve our results. In the next section, we will apply this method to our latest model and see what results we achieve.

## 2.8 UMGYDE Model: Matrix Factorization (MF)

The complete source code shown in this section is available in the Matrix Factorization (MF) section of the capstone-movielens.main.R script on *GitHub*.

For our next solution, we will use the `recosystem` package to perform the *Parallel Matrix Factorization*.

### 2.8.1 MF: Matematical Description

As outlined in this article[19], the idea of the *Matrix Factorization* method is to approximate the whole rating matrix $R_{m\times n}$ by the product of two matrices of lower dimensions, $P_{n\times k}$ and $Q_{n\times k}$, such that

$$\mathbb{R} \approx \mathbb{P}\mathbb{Q}^{\mathbb{T}} \tag{15}$$

In relation to our model, the expression (15) will take the form:

$$\mathbf{R} \sim \hat{\mathbf{R}} + \mathbf{P}\mathbf{Q}^{\mathbf{T}} + \varepsilon \tag{16}$$

where:

- $\mathbf{R}$ is the $U_{\mathrm{m}} \times M_n$ rating matrix with $U_m$ users and $M_n$ movies;
- $\hat{\mathbf{R}}$ represents the predictions from our best model: *Regularized UMGYDE Model*;
- $\mathbf{P}$ and $\mathbf{Q}$ are $P_{m\times k}$ and $Q_{n\times k}$ matrices, respectively, where $k$ is the number of *latent features* to be found.

If we denote the $u$-th row of $\mathbf{P}$ as $\mathbf{p}_u$ and the $v$-th row of $\mathbf{Q}$ as $\mathbf{q}_v$, then the unknown rating $\mathbf{r}_{u,v}$ given by user $u$ on movie item $v$ for our model can be estimated as $\hat{\mathbf{r}}_{u,v} + \mathbf{p}_u\mathbf{q}_v^T$, where the $\hat{\mathbf{r}}_{u,v}$ is the prediction given by our last *Regularized UMGYDE Model*.

A typical solution for $\mathbf{P}$ and $\mathbf{Q}$ is given by the following optimization problem [20, 21]:

$$\min_{P,Q} \sum_{(u,v)\in R} [f(\mathbf{p}_u, \mathbf{q}_v; \mathbf{r}_{u,v}) + \mu_P \parallel \mathbf{p}_u \parallel_1 + \mu_Q \parallel \mathbf{q}_v \parallel_1 + \frac{\lambda_P}{2} \parallel \mathbf{p}_u \parallel_2^2 + \frac{\lambda_Q}{2} \parallel \mathbf{q}_v \parallel_2^2]$$

where $(u, v)$ are locations of observed entries in $\mathbf{R}, \mathbf{r}_{u,v}$ is the observed ratings, $f$ is the loss function, and $\mu_P$, $\mu_Q$, $\lambda_P$, $\lambda_Q$ are penalty parameters to avoid overfitting.[19]

> *The process of solving the matrices $P$ and $Q$ is referred to as model training, and the selection of penalty parameters is called parameter tuning. In `recosystem`, we provide convenient functions for these two tasks, and additionally have functions for model exporting (outputting $P$ and $Q$ matrices) and prediction.*[19]

### 2.8.2 MF: Model Building

The complete source code shown in this section is available in the Perform the Matrix Factorization & Final Test section of the capstone-movielens.main.R script on *GitHub*.

We assume that the *Matrix Factorization* method will be the last one to provide sufficient results, so we will use the entire `edx` dataset as the *Training Set* and `final_holdout_test` as the *Test Set* for our final model, skipping the intermediate *training/validation split operations* that were performed for the previous models.

#### 2.8.2.1 MF: Getting Residuals From the UMGYDE Model Prediction Values

First, we will obtain residuals from the *UMGYDE Model* prediction values using the following line of code:

```
mf.edx.residual <- mf.residual.dataframe(edx)
```

In the code snippets above, we use the mf.residual.dataframe user-defined helper function described in section UMGYDE Model: Matrix Factorization of Appendix A.

```
str(mf.edx.residual)
```

```
## 'data.frame':    9000055 obs. of  3 variables:
##  $ userId : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId: int  122 185 292 316 329 355 356 362 364 370 ...
##  $ rsdl   : Named num  0.4868 0.3223 0.0386 0.1137 0.1269 ...
##   ..- attr(*, "names")= chr [1:9000055] "param.best_value" "param.best_value" "param.best_value" "pa
```

#### 2.8.2.2 MF: Transforming Input Data to Be Compatible With the `recosystem` Package

Next, we convert the residuals data and `final_holdout_test` dataset to the input data objects compatible with the recosystem package using the following code snippet:

```
set.seed(5430)
mf.edx.residual.reco <- with(mf.edx.residual,
                             data_memory(user_index = userId,
                                         item_index = movieId,
                                         rating = rsdl))

final_holdout_test.reco <- with(final_holdout_test,
                        data_memory(user_index = userId,
                                    item_index = movieId,
                                    rating = rating))
```

```
## *** 'Reco' input data objects structure ***
```

```
str(mf.edx.residual.reco)
```

```
## Formal class 'DataSource' [package "recosystem"] with 3 slots
##   ..@ source:List of 3
##   .. ..$ : int [1:9000055] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ : int [1:9000055] 122 185 292 316 329 355 356 362 364 370 ...
##   .. ..$ : num [1:9000055] 0.4868 0.3223 0.0386 0.1137 0.1269 ...
##   ..@ index1: logi FALSE
##   ..@ type  : chr "memory"
```

```
str(final_holdout_test.reco)
```

```
## Formal class 'DataSource' [package "recosystem"] with 3 slots
##   ..@ source:List of 3
##   .. ..$ : int [1:999999] 1 1 1 2 2 2 3 3 4 4 ...
##   .. ..$ : int [1:999999] 231 480 586 151 858 1544 590 4995 34 432 ...
##   .. ..$ : num [1:999999] 5 5 5 3 2 3 3.5 4.5 5 3 ...
##   ..@ index1: logi FALSE
##   ..@ type  : chr "memory"
```

#### 2.8.2.3 MF: Creating and Tuning the `Reco` Object

Now we can create a `reco` object of the `Reco` class and tune it using the `reco$tune` method, passing the residual data prepared in the previous step in the `train_data` argument using the following code snippet:

```
reco <- Reco()

reco.tuned <- reco$tune(mf.edx.residual.reco, opts = list(dim = c(10, 20, 30),
                                                          lrate    = c(0.1, 0.2),
                                                          nthread  = 4,
                                                          niter    = 10,
                                                          verbose  = TRUE))
```

```
## *** ‘Reco‘ object structure ***
```

```
str(reco)
```

```
## Reference class ’RecoSys’ [package "recosystem"] with 2 fields
##  $ model     :Reference class ’RecoModel’ [package "recosystem"] with 5 fields
##   ..$ path    : chr ""
##   ..$ nuser   : int 71568
##   ..$ nitem   : int 65134
##   ..$ nfac    : int 30
##   ..$ matrices:List of 3
##   .. ..$ P:Formal class ’float32’ [package "float"] with 1 slot
## ..@ Data: int [1:30, 1:71568] NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ...
##   .. ..$ Q:Formal class ’float32’ [package "float"] with 1 slot
## ..@ Data: int [1:30, 1:65134] NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ...
##   .. ..$ b:Formal class ’float32’ [package "float"] with 1 slot
## ..@ Data: int [1:1] 0.00114
##   ..and 16 methods, of which 2 are  possibly relevant:
##   ..  initialize, show#envRefClass
##  $ train_pars:List of 12
##   ..$ loss    : int 0
##   ..$ dim     : int 30
##   ..$ costp_l1: num 0
##   ..$ costp_l2: num 0.01
##   ..$ costq_l1: num 0
##   ..$ costq_l2: num 0.1
##   ..$ lrate   : num 0.1
##   ..$ niter   : num 20
##   ..$ nthread : num 4
##   ..$ nbin    : int 20
##   ..$ nmf     : logi FALSE
##   ..$ verbose : logi TRUE
##  and 19 methods, of which 5 are  possibly relevant:
##    output, predict, show#envRefClass, train, tune
```

```
## *** Tuned ‘Reco‘ object structure ***
```

```
str(reco.tuned)
```

```
## List of 2
##  $ min:List of 7
##   ..$ dim     : int 30
##   ..$ costp_l1: num 0
##   ..$ costp_l2: num 0.01
##   ..$ costq_l1: num 0
##   ..$ costq_l2: num 0.1
##   ..$ lrate   : num 0.1
##   ..$ loss_fun: num 0.795
##  $ res:’data.frame’: 96 obs. of  7 variables:
##   ..$ dim     : int [1:96] 10 20 30 10 20 30 10 20 30 10 ...
##   ..$ costp_l1: num [1:96] 0 0 0 0.1 0.1 0.1 0 0 0 0.1 ...
##   ..$ costp_l2: num [1:96] 0.01 0.01 0.01 0.01 0.01 0.01 0.1 0.1 0.1 0.1 ...
##   ..$ costq_l1: num [1:96] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ costq_l2: num [1:96] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 ...
##   ..$ lrate   : num [1:96] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 ...
##   ..$ loss_fun: num [1:96] 0.808 0.813 0.822 0.851 0.87 ...
##   ..- attr(*, "out.attrs")=List of 2
##   .. ..$ dim     : Named int [1:6] 3 2 2 2 2 2
##   .. .. ..- attr(*, "names")= chr [1:6] "dim" "costp_l1" "costp_l2" "costq_l1" ...
##   .. ..$ dimnames:List of 6
##   .. .. ..$ dim     : chr [1:3] "dim=10" "dim=20" "dim=30"
##   .. .. ..$ costp_l1: chr [1:2] "costp_l1=0.0" "costp_l1=0.1"
##   .. .. ..$ costp_l2: chr [1:2] "costp_l2=0.01" "costp_l2=0.10"
##   .. .. ..$ costq_l1: chr [1:2] "costq_l1=0.0" "costq_l1=0.1"
##   .. .. ..$ costq_l2: chr [1:2] "costq_l2=0.01" "costq_l2=0.10"
##   .. .. ..$ lrate   : chr [1:2] "lrate=0.1" "lrate=0.2"
```

#### 2.8.2.4 MF: Final Training

Finally, we train the model for the last time using the `reco$train` method, passing the same residual data we used for the tuning, in the `train_data` argument, using the following code snippet:

```
reco$train(mf.edx.residual.reco, opts = c(reco.tuned$min,
                                          niter = 20,
                                          nthread = 4))
```

### 2.8.3 MF: Final Holdout Test

From now on, we will work with the `final_holdout_test` dataset for the final testing of our model, including obtaining final predictions and computing the final *RMSE* score.

Now, let's compute the final predictions, combining the predicted residuals from the `reco` objects and predicted values we lastly get from the *UMGYDE Model* as described in section UMGYDE Model: Final Holdout Test (Preliminary Assessment), using the following code snippet:

```
mf.reco.residual <- reco$predict(final_holdout_test.reco, out_memory())

mf.predicted_ratings <-
  clamp(final.UMGYDE.predicted$predicted + mf.reco.residual)
```

In the code snippet above, we use the clamp user-defined helper function described in section Utility Functions of Appendix A.

```
str(mf.reco.residual)
```

```
##  num [1:999999] 0.146 -0.271 0.121 0.376 0.586 ...
```

```
str(mf.predicted_ratings)
```

```
##  Named num [1:999999] 4.63 4.73 4.75 3.68 4.66 ...
##  - attr(*, "names")= chr [1:999999] "param.best_value" "param.best_value" "param.best_value" "param.
```

And ultimately, we are ready to compute the final *RMSE* score using the following code snippet:

```
final_holdout_test.RMSE <- rmse2(final_holdout_test$rating,
                                 mf.predicted_ratings)
```

In the code snippet above, we use the rmse2 user-defined helper function described in section (Root) Mean Squared Error Calculation of Appendix A.

Finally, we add the *RMSE* value obtained above to our *Result Table* and print the table using the following code snippet:

```
final.MF.RMSEs.ResultTibble <- final.RMSEs.ResultTibble.UMGYDE.rglr.tuned |>
  RMSEs.AddRow("MF (Final Test)",
               final_holdout_test.RMSE,
               comment = "Matrix Factorization of the Best Model Residuals, Final Holdout Test")
```

```
RMSE_kable(final.MF.RMSEs.ResultTibble)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for ‘lambda‘ = 0.38745002746582 |
| UMGE Model | 0.8729730 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | Computed for ‘lambda‘ = 0.0359375 |
| UMGYE Model | 0.8723973 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | Computed for ‘lambda‘ = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using ‘stats::loess‘ function with ‘degree=1‘ & ‘span=0.75‘ parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | UMGYDE Model computed using function call: ‘loess(degree = 0, span = 0.00109375)‘ |
| Tuned UMGYDE.d1 Model | 0.8708295 | UMGYDE Model computed using function call: ‘loess(degree = 1, span = 0.00109375)‘ |
| Tuned UMGYDE.d2 Model | 0.8709886 | UMGYDE Model computed using function call: ‘loess(degree = 2, span = 0.00153125)‘ |
| Tuned UMGYDE Best Model | 0.8707850 | UMGYDE Model computed using ‘loess‘ function call with the best degree & span values. |
| Regularized UMGYDE Model | 0.8707750 | The best tuned and regularized UMGYDE Model. |
| Best UMGYDE Model (Final Test) | 0.8804225 | Final Holdout Test of the best tuned and regularized UMGYDE Model. |
| MF (Final Test) | 0.7875754 | Matrix Factorization of the Best Model Residuals, Final Holdout Test |

In the code snippet above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

Eventually, we have reached our ultimate goal, as the *RMSE* we obtained has achieved the required Project Objective.

# 3 Results

The *Result Summary Table* below, constructed using the following code snippet, shows in detail the progress of our *Recommender System* improvement:

```
total.RMSEs.ResultTibble <- RMSEs.AddDiffColumn(final.MF.RMSEs.ResultTibble)
```

```
RMSE.Total_kable(total.RMSEs.ResultTibble)
```

| Method | RMSE | Diff | Comment |
|---|---|---|---|
| Project Objective | 0.8649000 | 0.0000000 | |
| OMR Model | 1.0603462 | -0.1954462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | 0.0905500 | User Effect (UE) Model |
| UME Model | 0.8732081 | 0.0965881 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | 0.0002351 | Computed for ‘lambda‘ = 0.38745002746582 |
| UMGE Model | 0.8729730 | 0.0000000 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | 0.0000003 | Computed for ‘lambda‘ = 0.0359375 |
| UMGYE Model | 0.8723973 | 0.0005755 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | 0.0002117 | Computed for ‘lambda‘ = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | -0.0000553 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using ‘stats::loess‘ function with ‘degree=1‘ & ‘span=0.75‘ parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | 0.0011279 | UMGYDE Model computed using function call: ‘loess(degree = 0, span = 0.00109375)‘ |
| Tuned UMGYDE.d1 Model | 0.8708295 | 0.0002836 | UMGYDE Model computed using function call: ‘loess(degree = 1, span = 0.00109375)‘ |
| Tuned UMGYDE.d2 Model | 0.8709886 | -0.0001591 | UMGYDE Model computed using function call: ‘loess(degree = 2, span = 0.00153125)‘ |
| Tuned UMGYDE Best Model | 0.8707850 | 0.0002036 | UMGYDE Model computed using ‘loess‘ function ca with the best degree & span values. |
| Regularized UMGYDE Model | 0.8707750 | 0.0000100 | The best tuned and regularized UMGYDE Model. |
| Best UMGYDE Model (Final Test) | 0.8804225 | -0.0096476 | Final Holdout Test of the best tuned and regularized UMGYDE Model. |
| MF (Final Test) | 0.7875754 | 0.0928471 | Matrix Factorization of the Best Model Residuals, F Holdout Test |

In the code snippet above, we use the RMSEs.AddDiffColumn and RMSE.Total_kable functions described in section Result RMSEs Tibble Functions of Appendix A.

# 4 Conclusion

In this project, a *Machine Learning Algorithm* was successfully developed for a *Recommender System* (hereafter referred to as *RS MLA*) to predict movie ratings using the 10M version of the MovieLens dataset.

Since it was needed to convert the original dataset into a *Data Source*, suitable for processing by the *RS MLA*, a package edx.capstone.movielens.data, specifically designed for this purpose, was developed and used, providing the *RS MLA* with the necessary datasets.

Following the Netflix Grand Prize Contest requirements, the *Root Mean Squared Error (RMSE)* metric was used to evaluate the accuracy of our models.

By incorporating essential factors such as overall average rating, movie, user, and some other observable effects, as well as efficient techniques such as *Regularization* and *Matrix Factorization* methods, the accuracy was significantly improved compared to the baseline model.

The baseline model was the *Overall Mean Rating* model described in section Overall Mean Rating (OMR) Model.

The following effects were taken into account:

- ***User:*** based on which the *User Effect (UE) Model* described in section User Effect (UE) Model was built (significantly improved the *RMSE* compared to the baseline model);
- ***Movie:*** based on which (and the *User Effect*) the *Regularized User+Movie Effect (UME) Model* described in section User+Movie Effect (UME) Model was eventually built (even better improved the *RMSE* compared to the previous (*UE*) model);
- ***Genre:*** based on which (and the previous effects) the *Regularized User+Movie+Genre Effect (UMGE) Model* described in section User+Movie+Genre Effect (UMGE) Model was built (even after *regularization* it showed very little improvement in *RMSE* compared to the previous (*UME*) model; before regularization, for some reason, there was no improvement at all);
- ***Year:*** based on which (and the previous effects) the *Regularized User+Movie+Genre+Year Effect (UMGYE) Model* described in section User+Movie+Genre+Year Effect (UMGYE) Model was built (provided a much better improvement in *RMSE* compared to the previous model but still two orders of magnitude less than the *UE* and *UME* models showed);
- ***Smoothed Day*** based on which (and the previous effects) the *Regularized (and Tuned) User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* described in section User+Movie+Genre+Year+Sn Effect (UMGYDE) Model was built (showed almost twice better improvement than the previous (*UMGYE*) model but still not good enough to meet the *Project Objective*);.
- ***Smoothed Day (Matrix Factorization):*** (described in section UMGYDE Model: Matrix Factorization (MF)) *Matrix Factorization* method performed on the *Regularized UMGYDE Model* provided substantial improvement comparable to that shown by the *UE* and *UME* models, and ultimately allowed us to reach the *Project Objective*.

The *Result Summary Table* provided above in section Results shows in detail the progress of our *Recommender System* improvement.

Although the *Project Objective* was achieved, further research could explore the use of other methods, such as advanced algorithms described in *Part V Advanced Algorithms* of the *Recommender Systems Handbook* (written by *Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor*)[16], or *Deep Learning Models* as described in article Advanced Deep Learning Models for Improving Movie Rating Predictions: A Benchmarking Study[22].

# 5 Appendix A: Support Functions

## 5.1 Common Helper Functions

### 5.1.1 Logging Functions

The full source code of the functions described in this section is available in the logging-functions.R script on *GitHub*.

#### 5.1.1.1 open_logfile Function

A function to initialize the log file.

##### 5.1.1.1.1 Usage

```
open_logfile(".data-analysis")
```

##### 5.1.1.1.2 Arguments

- **file_name:** A name used as a base to generate a unique log file name;

##### 5.1.1.1.3 Details

This function is a wrapper for the logr::log_open function. It also calls internally the following functions to generate a unique file name based on the file_name argument and *system time*:

- Sys.time;
- stringr::str_replace;
- stringr::str_replace_all

For more information on the *inner functions* listed above, see the R Documentation.

##### 5.1.1.1.4 Value

The path of the log.

#### 5.1.1.1.5 Source Code

The source code of the open_logfile function is shown below:

```
library(logr)
library(stringr)

open_logfile <- function(file_name){
  log_file_name <- as.character(Sys.time()) |>
    str_replace_all(':', '_') |>
    str_replace(' ', 'T') |>
    str_c(file_name)

  log_open(file_name = log_file_name)
}
```

The source code of the open_logfile function is defined in the logging-functions.R script on *GitHub*.

### 5.1.1.2 print_start_date Function

Prints the system date and time at the time the function is called and returns the system's idea of the current date and time.

#### 5.1.1.2.1 Usage

```
start_date <- print_start_date()
```

```
## [1] "Wed Feb 25 20:34:14 2026"
```

```
start_date
```

```
## [1] "2026-02-25 20:34:14 MSK"
```

#### 5.1.1.2.2 Details

This function internally calls base::print, base::date, and base::Sys.time `R` functions.

It first prints the value returned by the `date` function, then returns the object returned by the `Sys.time` function.

#### 5.1.1.2.3 Value

An object of class `POSIXct` (see DateTimeClasses).

#### 5.1.1.2.4 Source Code

The source code of the print_start_date function is shown below:

```
print_start_date <- function(){
  print(date())
  Sys.time()
}
```

The source code of the print_start_date function is defined in the logging-functions.R script on *GitHub*.

### 5.1.1.3 put_start_date Function

Outputs the system date and time to the currently opened log (using the open_logfile function described above) at the time the function is called and returns the system's idea of the current date and time.

#### 5.1.1.3.1 Usage

```
  start <- put_start_date()
```

#### 5.1.1.3.2 Details

This function internally calls logr::put, base::date, and base::Sys.time `R` functions.

It first outputs the value returned by the `date` function to the currently opened log, then returns the object returned by the `Sys.time` function.

#### 5.1.1.3.3 Value

An object of class `POSIXct` (see DateTimeClasses).

#### 5.1.1.3.4 Source Code

The source code of the put_start_date function is shown below:

```
put_start_date <- function(){
  put(date())
  Sys.time()
}
```

The source code of the put_start_date function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.4 print_end_date Function

Prints the system date and time at the time the function is called, and then the time elapsed since the date and time specified by the start argument.

##### 5.1.1.4.1 Usage

```
start_date <- print_start_date()
```

```
## [1] "Wed Feb 25 20:34:14 2026"
```

```
Sys.sleep(3)
print_end_date(start_date)
```

```
## [1] "Wed Feb 25 20:34:18 2026"
## Time difference of 3.076213 secs
```

##### 5.1.1.4.2 Arguments

- **start:** An object of class `POSIXct` representing the value of the moment from which time is measured.

##### 5.1.1.4.3 Details

This function internally calls base::print, base::date, and base::Sys.time `R` functions.

It first prints the value returned by the `date` function, then builds an object of class `difftime` representing the time elapsed since the date and time specified by the start argument, prints the object, and, finally, returns the object to the client code.

##### 5.1.1.4.4 Value

An object of class `difftime` representing the time elapsed since the date and time specified by the start argument.

##### 5.1.1.4.5 Source Code

The source code of the print_end_date function is shown below:

```
print_end_date <- function(start){
  print(date())
  print(Sys.time() - start)
}
```

The source code of the print_end_date function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.5 put_end_date Function

Outputs the system date and time to the currently open log (using the open_logfile function described above).

The first value in the output is the system date and time at the time the function is called, and the second value is the time elapsed since the date and time specified by the start argument.

##### 5.1.1.5.1 Usage

```
put_end_date(start)
```

##### 5.1.1.5.2 Arguments

- **start:** An object of class `POSIXct` representing the value of the moment from which time is measured.

##### 5.1.1.5.3 Details

This function works similarly to the print_end_date function described above, but unlike the latter, it outputs the information to the currently open log file.

##### 5.1.1.5.4 Value

An object of class `difftime` representing the time elapsed since the date and time specified by the start argument.

##### 5.1.1.5.5 Source Code

The source code of the put_end_date function is shown below:

```
put_end_date <- function(start){
  put(date())
  put(Sys.time() - start)
}
```

The source code of the put_end_date function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.6 print_log Function

Prints a multiline string specified by the msg argument.

##### 5.1.1.6.1 Usage

```
print_log("Hello World!")
```

```
## Hello World!
```

##### 5.1.1.6.2 Arguments

- **msg:** A string to print (can be multiline).

##### 5.1.1.6.3 Details

This function internally calls stringr::str_glue and base::print R functions to print the (possibly multiline) string specified by the msg argument.

##### 5.1.1.6.4 Value

A glue object, as created by glue::as_glue().

##### 5.1.1.6.5 Source Code

The source code of the print_log function is shown below:

```
print_log <- function(msg){
  print(str_glue(msg))
}
```

The source code of the print_log function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.7 put_log Function

Outputs a multiline string specified by the msg argument to the currently opened log (using the open_logfile function described above).

##### 5.1.1.7.1 Usage

```
put_log("Computing Average Ratings per User (User Mean Ratings)...")
```

```
## Computing Average Ratings per User (User Mean Ratings)...
```

##### 5.1.1.7.2 Arguments

- **msg:** A string to print (can be multiline).

##### 5.1.1.7.3 Details

This function internally calls stringr::str_glue and logr::put `R` functions to output the (possibly multiline) string specified by the msg argument.

The function works similarly to the print_log function described above, but unlike the latter, it outputs the information to the currently open log file.

##### 5.1.1.7.4 Value

A glue object, as created by glue::as_glue().

##### 5.1.1.7.5 Source Code

The source code of the put_log function is shown below:

```
put_log <- function(msg){
  put(str_glue(msg))
}
```

The source code of the put_log function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.8 get_log1 Function

Creates a character vector from the template passed in the msg_template argument, using the value specified by the arg1 argument to replace all the occurrences of the pattern *"%1"* found in the template.

##### 5.1.1.8.1 Usage

```
put(get_log1(msg_template, arg1))
```

In the code snippet above, the function `get_log1` is called internally from the put_log1 function described below in *this Section*.

##### 5.1.1.8.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of the pattern *"%1"* to be replaced by the value specified by the arg1 argument.

- **arg1:** Value to replace all occurrences of the pattern *"%1"* in the message template passed in the msg_template argument.

##### 5.1.1.8.3 Details

This function internally calls stringr::str_glue and stringr::str_replace_all `R` functions to form a character vector from the template passed in the msg_template argument, using the value specified by the arg1 argument to replace all the occurrences of the pattern *"%1"* found in the template.

This is an auxiliary function intended for use in other custom logging functions, such as the put_log1 (as shown in the Usage section of *this Description*) and print_log1 described below in *this Section*.

##### 5.1.1.8.4 Value

A glue object, returned by the internally called stringr::str_glue function, created from the character vector returned by the (also internally called) function stringr::str_replace_all.

#### 5.1.1.8.5 Source Code

The source code of the get_log1 function is shown below:

```
get_log1 <- function(msg_template, arg1) {
  str_glue(str_replace_all(msg_template, "%1", as.character(arg1)))
}
```

The source code of the get_log1 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.9 print_log1 Function

Prints a log created from the template passed in the msg_template argument, using the value specified by the arg1 argument to replace all the occurrences of the pattern *"%1"* found in the template.

##### 5.1.1.9.1 Usage

```
which_min_deviation <- deviation[which.min(deviation.RMSE)]
min_rmse = min(deviation.RMSE)

print_log1("Minimum RMSE is achieved when the deviation from the mean is: %1",
         which_min_deviation)
```

```
## Minimum RMSE is achieved when the deviation from the mean is: 0
```

```
print_log1("Is the previously computed RMSE the best for the current model: %1",
         mu.RMSE == min_rmse)
```

```
## Is the previously computed RMSE the best for the current model: TRUE
```

##### 5.1.1.9.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of the pattern *"%1"* to be replaced by the value specified by the arg1 argument.

- **arg1:** Value to replace all occurrences of the pattern *"%1"* in the message template passed in the msg_template argument.

##### 5.1.1.9.3 Details

This function internally calls the base::print `R` function to print the object returned by the get_log1 function (also called internally) described above.

##### 5.1.1.9.4 Value

A glue object, returned by the internally called function get_log1 described above.

#### 5.1.1.9.5 Source Code

The source code of the print_log1 function is shown below:

```
print_log1 <- function(msg_template, arg1){
  print(get_log1(msg_template, arg1))
}
```

The source code of the print_log1 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.10 put_log1 Function

Outputs a log to the currently open log file (using the open_logfile function described above), created from the template passed in the msg_template argument, using the value specified by the arg1 argument to replace all the occurrences of the pattern *"%1"* found in the template.

##### 5.1.1.10.1 Usage

```
which_min_deviation <- deviation[which.min(deviation.RMSE)]
min_rmse = min(deviation.RMSE)
```

```
put_log1("Minimum RMSE is achieved when the deviation from the mean is: %1",
         which_min_deviation)

put_log1("Is the previously computed RMSE the best for the current model: %1",
         mu.RMSE == min_rmse)
```

```
## Minimum RMSE is achieved when the deviation from the mean is: 0

## Is the previously computed RMSE the best for the current model: TRUE
```

##### 5.1.1.10.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of the pattern *"%1"* to be replaced by the value specified by the arg1 argument.

- **arg1:** Value to replace all occurrences of the pattern *"%1"* in the message template passed in the msg_template argument.

##### 5.1.1.10.3 Details

This function internally calls the logr::put `R` function to output the log object returned by the (also called internally) get_log1 function described above.

The function works similarly to the print_log1 function described above, but unlike the latter, it outputs the information to the currently open log file.

##### 5.1.1.10.4 Value

A glue object, returned by the internally called function get_log1 described above.

#### 5.1.1.10.5 Source Code

The source code of the put_log1 function is shown below:

```
put_log1 <- function(msg_template, arg1){
  put(get_log1(msg_template, arg1))
}
```

The source code of the put_log1 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.11 get_log2 Function

Creates a character vector from the template passed in the msg_template argument, using the values specified by the arg1 and arg2 arguments to replace all the occurrences of the placeholders *"%1"* and *"%2"*, respectively, found in the template.

##### 5.1.1.11.1 Usage

```
put(get_log2(msg_template, arg1, arg2))
```

In the code snippet above, the function `get_log2` is called internally from the put_log2 function described below in *this Section*.

##### 5.1.1.11.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"* and *"%2"*, to be replaced by the values specified by the arg1 and arg2 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

##### 5.1.1.11.3 Details

This function internally calls stringr::str_glue and stringr::str_replace_all `R` functions to form a character vector from the template passed in the msg_template argument, using the values specified by the arg1 and arg2 arguments to replace all the occurrences of the placeholders *"%1"* and *"%2"*, respectively, found in the template.

This is an auxiliary function intended for use in other custom logging functions, such as the put_log2 (as shown in the Usage section of *this Description*) and print_log2 described below in *this Section*.

##### 5.1.1.11.4 Value

A glue object, returned by the internally called stringr::str_glue function, created from the character vector returned by the (also internally called) function stringr::str_replace_all.

#### 5.1.1.11.5 Source Code

The source code of the get_log2 function is shown below:

```
get_log2 <- function(msg_template, arg1, arg2) {
  msg_template |>
    str_replace_all("%1", as.character(arg1)) |>
    str_replace_all("%2", as.character(arg2)) |>
    str_glue()
}
```

The source code of the get_log2 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.12 print_log2 Function

Prints a log created from the template passed in the msg_template argument, using the values specified by the arg1 and arg2 arguments to replace all the occurrences of the placeholders *"%1"* and *"%2"*, respectively, found in the template.

##### 5.1.1.12.1 Usage

```
print_log2("%1-Fold Cross Validation ultimate RMSE: %2", CVFolds_N, mu.RMSE)
```

```
## 5-Fold Cross Validation ultimate RMSE: 1.06034618796622
```

##### 5.1.1.12.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"* and *"%2"*, to be replaced by the values specified by the arg1 and arg2 arguments, respectively.
- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.
- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

##### 5.1.1.12.3 Details

This function internally calls the base::print `R` function to print the object returned by the get_log2 function (also called internally) described above.

##### 5.1.1.12.4 Value

A glue object, returned by the internally called function get_log2 described above.

##### 5.1.1.12.5 Source Code

The source code of the print_log2 function is shown below:

```
print_log2 <- function(msg_template, arg1, arg2){
  print(get_log2(msg_template, arg1, arg2))
}
```

The source code of the print_log2 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.13 put_log2 Function

Outputs a log to the currently open log file (using the open_logfile function described above), created from the template passed in the msg_template argument, using the values specified by the arg1 and arg2 arguments to replace all the occurrences of the placeholders *"%1"* and *"%2"*, respectively, found in the template.

##### 5.1.1.13.1 Usage

```
put_log2("%1-Fold Cross Validation ultimate RMSE: %2", CVFolds_N, mu.RMSE)
```

```
## 5-Fold Cross Validation ultimate RMSE: 1.06034618796622
```

##### 5.1.1.13.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"* and *"%2"*, to be replaced by the values specified by the arg1 and arg2 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

##### 5.1.1.13.3 Details

This function internally calls the logr::put `R` function to output the log object returned by the (also called internally) get_log2 function described above.

The function works similarly to the print_log2 function described above, but unlike the latter, it outputs the information to the currently open log file.

##### 5.1.1.13.4 Value

A glue object, returned by the internally called function get_log2 described above.

#### 5.1.1.13.5 Source Code

The source code of the put_log2 function is shown below:

```
put_log2 <- function(msg_template, arg1, arg2){
  put(get_log2(msg_template, arg1, arg2))
}
```

The source code of the put_log2 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.14 get_log3 Function

Creates a character vector from the template passed in the msg_template argument, using the values specified by the arg1, arg2, and arg3 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, and *"%3"*, respectively, found in the template.

##### 5.1.1.14.1 Usage

```
put(get_log3(msg_template, arg1, arg2, arg3))
```

In the code snippet above, the function `get_log3` is called internally from the put_log3 function described below in *this Section*.

##### 5.1.1.14.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"*, *"%2"*, and *"%3"* to be replaced by the values specified by the arg1, arg2, and arg3 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

- **arg3:** Value to replace all occurrences of the placeholder *"%3"* in the message template passed in the msg_template argument.

##### 5.1.1.14.3 Details

This function internally calls stringr::str_glue and stringr::str_replace_all `R` functions to form a character vector from the template passed in the msg_template argument, using the values specified by the arg1, arg2, and arg3 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, and *"%3"*, respectively, found in the template.

This is an auxiliary function intended for use in other custom logging functions, such as the put_log3 (as shown in the Usage section of *this Description*) and print_log3 described below in *this Section*.

#### 5.1.1.14.4 Value

A glue object, returned by the internally called stringr::str_glue function, created from the character vector returned by the (also internally called) function stringr::str_replace_all.

#### 5.1.1.14.5 Source Code

The source code of the get_log3 function is shown below:

```
get_log3 <- function(msg_template, arg1, arg2, arg3) {
  msg_template |>
    str_replace_all("%1", as.character(arg1)) |>
    str_replace_all("%2", as.character(arg2)) |>
    str_replace_all("%3", as.character(arg3)) |>
    str_glue()
}
```

The source code of the get_log3 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.15 print_log3 Function

Prints a log created from the template passed in the msg_template argument, using the values specified by the arg1, arg2, and arg3 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, and *"%3"*, respectively, found in the template.

##### 5.1.1.15.1 Usage

```
print_log3("RMSE value has been computed using `loess` function
with the best parameters for the %1-Fold Cross Validation samples:
degree = %2;
span = %3.",
        CVFolds_N,
        lss.best_degree,
        lss.best_span)
```

```
## RMSE value has been computed using ‘loess‘ function
## with the best parameters for the 5-Fold Cross Validation samples:
## degree = 1;
## span = 0.00109375.
```

##### 5.1.1.15.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"*, *"%2"*, and *"%3"* to be replaced by the values specified by the arg1, arg2, and arg3 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

- **arg3:** Value to replace all occurrences of the placeholder *"%3"* in the message template passed in the msg_template argument.

##### 5.1.1.15.3 Details

This function internally calls the base::print `R` function to print the object returned by the get_log3 function (also called internally) described above.

##### 5.1.1.15.4 Value

A glue object, returned by the internally called function get_log3 described above.

#### 5.1.1.15.5 Source Code

The source code of the print_log3 function is shown below:

```
print_log3 <- function(msg_template, arg1, arg2, arg3){
  print(get_log3(msg_template, arg1, arg2, arg3))
}
```

The source code of the print_log3 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.16 put_log3 Function

Outputs a log to the currently open log file (using the open_logfile function described above), created from the template passed in the msg_template argument, using the values specified by the arg1, arg2, and arg3 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, and *"%3"*, respectively, found in the template.

##### 5.1.1.16.1 Usage

```
put_log3("RMSE value has been computed using `loess` function
with the best parameters for the %1-Fold Cross Validation samples:
degree = %2;
span = %3.",
        CVFolds_N,
        lss.best_degree,
        lss.best_span)
```

```
## RMSE value has been computed using ‘loess‘ function
## with the best parameters for the 5-Fold Cross Validation samples:
## degree = 1;
## span = 0.00109375.
```

##### 5.1.1.16.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"*, *"%2"*, and *"%3"* to be replaced by the values specified by the arg1, arg2, and arg3 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

- **arg3:** Value to replace all occurrences of the placeholder *"%3"* in the message template passed in the msg_template argument.

##### 5.1.1.16.3 Details

This function internally calls the logr::put `R` function to output the log object returned by the (also called internally) get_log3 function described above.

The function works similarly to the print_log3 function described above, but unlike the latter, it outputs the information to the currently open log file.

#### 5.1.1.16.4 Value

A glue object, returned by the internally called function get_log3 described above.

#### 5.1.1.16.5 Source Code

The source code of the put_log3 function is shown below:

```
put_log3 <- function(msg_template, arg1, arg2, arg3){
  put(get_log3(msg_template, arg1, arg2, arg3))
}
```

The source code of the put_log3 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.17 get_log4 Function

Creates a character vector from the template passed in the msg_template argument, using the values specified by the arg1, arg2, arg3, and arg4 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, *"%3"*, and *"%4"*, respectively, found in the template.

##### 5.1.1.17.1 Usage

```
put(get_log4(msg_template, arg1, arg2, arg3, arg4))
```

In the code snippet above, the function `get_log4` is called internally from the put_log4 function described below in *this Section*.

##### 5.1.1.17.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"*, *"%2"*, *"%3"*, and *"%4"* to be replaced by the values specified by the arg1, arg2, arg3, and arg4 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

- **arg3:** Value to replace all occurrences of the placeholder *"%3"* in the message template passed in the msg_template argument.

- **arg4:** Value to replace all occurrences of the placeholder *"%4"* in the message template passed in the msg_template argument.

##### 5.1.1.17.3 Details

This function internally calls stringr::str_glue and stringr::str_replace_all `R` functions to form a character vector from the template passed in the msg_template argument, using the values specified by the arg1, arg2, arg3, and arg4 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, *"%3"*, and *"%4"*, respectively, found in the template.

This is an auxiliary function intended for use in other custom logging functions, such as the put_log4 (as shown in the Usage section of *this Description*) and print_log4 described below in *this Section*.

#### 5.1.1.17.4 Value

A glue object, returned by the internally called stringr::str_glue function, created from the character vector returned by the (also internally called) function stringr::str_replace_all.

#### 5.1.1.17.5 Source Code

The source code of the get_log4 function is shown below:

```
get_log4 <- function(msg_template, arg1, arg2, arg3, arg4) {
  msg_template |>
    str_replace_all("%1", as.character(arg1)) |>
    str_replace_all("%2", as.character(arg2)) |>
    str_replace_all("%3", as.character(arg3)) |>
    str_replace_all("%4", as.character(arg4)) |>
    str_glue()
}
```

The source code of the get_log4 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.18 print_log4 Function

Prints a log created from the template passed in the msg_template argument, using the values specified by the arg1, arg2, arg3, and arg4 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, *"%3"*, and *"%4"*, respectively, found in the template.

##### 5.1.1.18.1 Usage[23]

```
arg1 <- "Data Collection"
arg2 <- "Data Cleaning and Transformation"
arg3 <- "Statistical Analysis"
arg4 <- "Data Visualization"

msg_template <- "The 4 Important Aspects of Data Science are as follows:
  1. %1;
  2. %2;
  3. %3;
  4. %4."

print_log4(msg_template, arg1, arg2, arg3, arg4)
```

```
## The 4 Important Aspects of Data Science are as follows:
## 1. Data Collection;
## 2. Data Cleaning and Transformation;
## 3. Statistical Analysis;
## 4. Data Visualization.
```

##### 5.1.1.18.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"*, *"%2"*, *"%3"*, and *"%4"* to be replaced by the values specified by the arg1, arg2, arg3, and arg4 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

- **arg3:** Value to replace all occurrences of the placeholder *"%3"* in the message template passed in the msg_template argument.

- **arg4:** Value to replace all occurrences of the placeholder *"%4"* in the message template passed in the msg_template argument.

#### 5.1.1.18.3 Details

This function internally calls the base::print `R` function to print the object returned by the get_log4 function (also called internally) described above.

#### 5.1.1.18.4 Value

A glue object, returned by the internally called function get_log4 described above.

#### 5.1.1.18.5 Source Code

The source code of the print_log4 function is shown below:

```
print_log4 <- function(msg_template, arg1, arg2, arg3, arg4){
  print(get_log4(msg_template, arg1, arg2, arg3, arg4))
}
```

The source code of the print_log4 function is defined in the logging-functions.R script on *GitHub*.

#### 5.1.1.19 put_log4 Function

Outputs a log to the currently open log file (using the open_logfile function described above), created from the template passed in the msg_template argument, using the values specified by the arg1, arg2, arg3, and arg4 arguments to replace all the occurrences of the placeholders *"%1"*, *"%2"*, *"%3"*, and *"%4"*, respectively, found in the template.

##### 5.1.1.19.1 Usage[23]

```
arg1 <- "Data Collection"
arg2 <- "Data Cleaning and Transformation"
arg3 <- "Statistical Analysis"
arg4 <- "Data Visualization"

msg_template <- "The 4 Important Aspects of Data Science are as follows:
  1. %1;
  2. %2;
  3. %3;
  4. %4."
```

```
put_log4(msg_template, arg1, arg2, arg3, arg4)
```

```
## The 4 Important Aspects of Data Science are as follows:
## 1. Data Collection;
## 2. Data Cleaning and Transformation;
## 3. Statistical Analysis;
## 4. Data Visualization.
```

##### 5.1.1.19.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of each of the placeholders *"%1"*, *"%2"*, *"%3"*, and *"%4"* to be replaced by the values specified by the arg1, arg2, arg3, and arg4 arguments, respectively.

- **arg1:** Value to replace all occurrences of the placeholder *"%1"* in the message template passed in the msg_template argument.

- **arg2:** Value to replace all occurrences of the placeholder *"%2"* in the message template passed in the msg_template argument.

- **arg3:** Value to replace all occurrences of the placeholder *"%3"* in the message template passed in the msg_template argument.

- **arg4:** Value to replace all occurrences of the placeholder *"%4"* in the message template passed in the msg_template argument.

#### 5.1.1.19.3 Details

This function internally calls the logr::put `R` function to output the log object returned by the (also called internally) get_log4 function described above.

The function works similarly to the print_log4 function described above, but unlike the latter, it outputs the information to the currently open log file.

#### 5.1.1.19.4 Value

A glue object, returned by the internally called function get_log4 described above.

#### 5.1.1.19.5 Source Code

The source code of the put_log4 function is shown below:

```
put_log4 <- function(msg_template, arg1, arg2, arg3, arg4){
  put(get_log4(msg_template, arg1, arg2, arg3, arg4))
}
```

The source code of the put_log4 function is defined in the logging-functions.R script on *GitHub*.

### 5.1.2 Utility Functions

The complete source code of the functions described in this section are available in the Utility Functions section of the common-helper.functions.R script on *GitHub*.

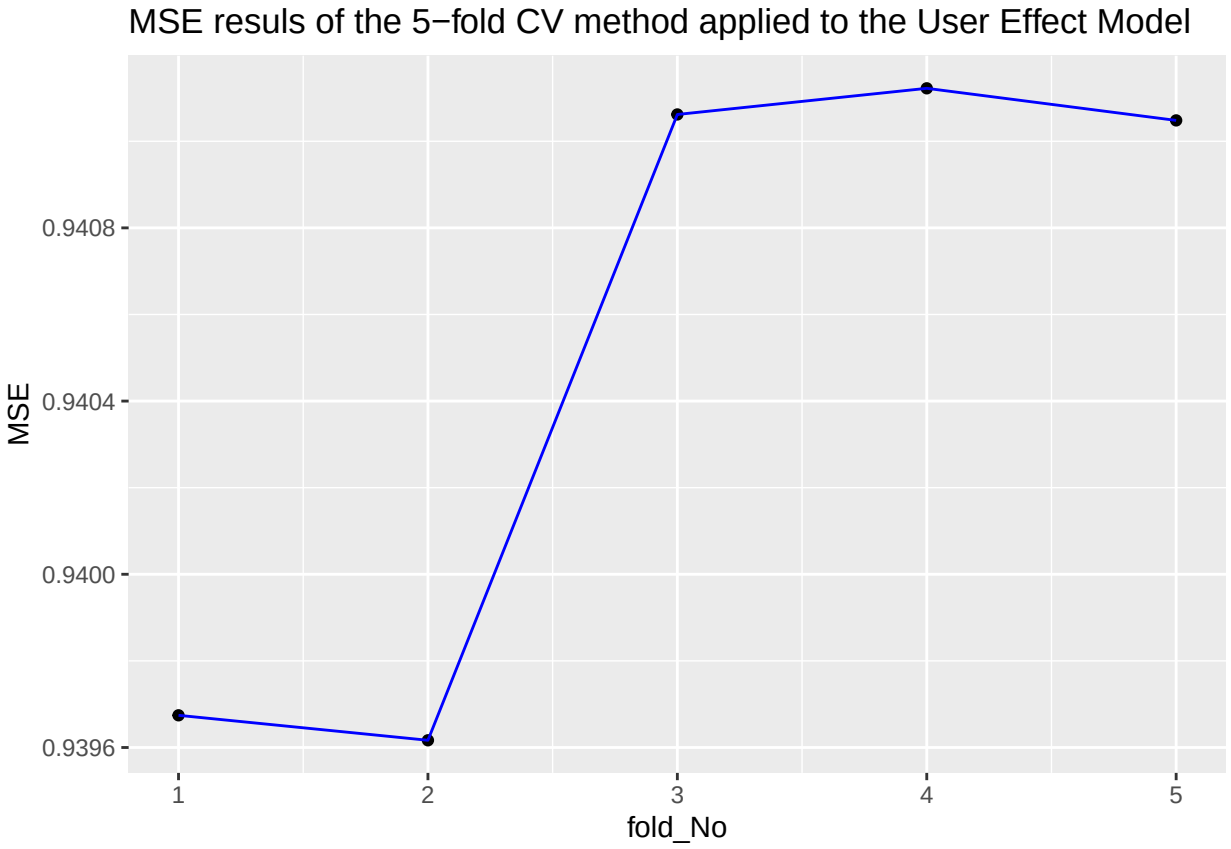
#### 5.1.2.1 clamp Function

Clamps the input value passed in r argument between the values passed in min and max arguments.

##### 5.1.2.1.1 Usage

```
edx.user_effect.MSEs <- sapply(edx_CV, function(cv_fold_dat){
  cv_fold_dat$validation_set |>
    left_join(edx.user_effect, by = "userId") |>
    mutate(resid = rating - clamp(mu + a)) |>
    pull(resid) |> mse()
})
```

```
data.frame(fold_No = 1:5, MSE = edx.user_effect.MSEs) |>
  data.plot(title = "MSE resuls of the 5-fold CV method applied to the User Effect Model",
            xname = "fold_No",
            yname = "MSE")
```



In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of *this Appendix*.

#### 5.1.2.1.2 Arguments

- **r:** Value to clamp;
- **min:** (Optional, 0.5 by default) Minimum allowed value;
- **max:** (Optional, 5 by default) Maximum allowed value;

:::

#### 5.1.2.1.3 Details

As explained in Section 24.1 *Case study: recommendation systems / User effects* of the *Course Textbook* we know ratings cannot be below 0.5 or above 5. For this reason, we need a function described in that section[6]:

This function is a wrapper for the base::pmin and base::pmax `R` functions, which are called internally to constrain the input value passed in the r argument to be within the values passed in the min and max arguments, respectively.

in *this Project* this function is used to clamp the computed movie rating values between 0.5 and 5.

#### 5.1.2.1.4 Value

The value clamped between min and max argument values.

#### 5.1.2.1.5 Source Code

The source code of the clamp function is shown below:

```
clamp <- function(x, min = 0.5, max = 5) pmax(pmin(x, max), min)
```

The source code of the clamp function is also available in the Utility Functions section of the common-helper.functions.R script on *GitHub*.

#### 5.1.2.2 msg.set_arg Function

Creates a character vector from the template passed in the msg_template argument, using the value specified by the arg argument to replace all the occurrences of the placeholder passed in the arg.name argument found in the template.

##### 5.1.2.2.1 Usage

```
msg_template |>
  msg.set_arg(arg, arg.name) |>
  str_glue()
```

In the above code snippet, the `msg.set_arg` function is called internally from the msg.glue function described below in *this Section*.

##### 5.1.2.2.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of a pattern specified by the arg.name argument.

- **arg.name:** (Optional, *"%1"* by default) Substring (pattern or *placeholder*) to replace all occurrences found in the message template passed in the msg_template argument with the value specified in the arg argument.

- **arg:** Value to replace all occurrences of the pattern specified by the arg.name argument in the message template passed in the msg_template argument.

##### 5.1.2.2.3 Details

This function is a wrapper for the stringr::str_replace_all `R` function, which is called internally to replace all occurrences of the pattern (placeholder) specified by the arg_name argument (*"%1"* by default), in the message template passed in the msg_template argument with the value specified in the arg argument.

This is an auxiliary function intended to be called internally from the msg.glue function (described below in *this Section*) as shown above in the Usage section of *this Description*.

##### 5.1.2.2.4 Value

The character vector returned by the internally called stringr::str_replace_all().

#### 5.1.2.2.5 Source Code

The source code of the msg.set_arg function is shown below:

```
msg.set_arg <- function(msg_template, arg, arg.name = "%1") {
  msg_template |>
    str_replace_all(arg.name, as.character(arg))
}
```

The source code of the msg.set_arg function is also available in the Utility Functions section of the common-helper.functions.R script on *GitHub*.

#### 5.1.2.3 msg.glue Function

Creates a glue::glue object from the template passed in the msg_template argument, using the value specified by the arg argument to replace all the occurrences of the placeholder passed in the arg.name argument found in the template.

##### 5.1.2.3.1 Usage

```
#### Add a row to the RMSE Result Table for the Regularized User+Movie Effect Model --------
RMSEs.ResultTibble.rglr.UME <- RMSEs.ResultTibble.UME |>
  RMSEs.AddRow("Regularized UME Model",
               UME.rglr.retrain.RMSE,
               comment = "Computed for `lambda` = %1" |>
                 msg.glue(UME.rglr.best_lambda))
```

```
RMSE_kable(RMSEs.ResultTibble.rglr.UME)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.8649000 | |
| OMR Model | 1.0603462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | User Effect (UE) Model |
| UME Model | 0.8732081 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | Computed for ‘lambda‘ = 0.38745002746582 |

In the *code snippet* above, we use the RMSEs.AddRow and RMSE_kable functions described in section Result RMSEs Tibble Functions of *this Appendix*.

##### 5.1.2.3.2 Arguments

- **msg_template:** Any object coercible to a character vector containing at least one message template element, i.e., a string (possibly multiline) containing at least one occurrence of a pattern specified by the arg.name argument.

- **arg.name:** (Optional, *“%1”* by default) Substring (pattern or *placeholder*) to replace all occurrences found in the message template passed in the msg_template argument with the value specified in the arg argument.

- **arg:** Value to replace all occurrences of the pattern specified by the arg.name argument in the message template passed in the msg_template argument.

##### 5.1.2.3.3 Details

This function internally calls the stringr::str_glue `R` function to format the character vector returned by the (also called internally) msg.set_arg function described above in *this Section*.

#### 5.1.2.3.4 Value

A glue::glue object created from the character vector returned by the internally called msg.set_arg().

#### 5.1.2.3.5 Source Code

The source code of the msg.glue function is shown below:

```
msg.glue <- function(msg_template, arg, arg.name = "%1"){
  msg_template |>
    msg.set_arg(arg, arg.name) |>
    str_glue()
}
```

The source code of the msg.glue function is also available in the Utility Functions section of the common-helper.functions.R script on *GitHub*.

### 5.1.3 (Root) Mean Squared Error Calculation

The complete source code of the functions described in this section are available in the (R)MSE-related functions section of the common-helper.functions.R script on *GitHub*.

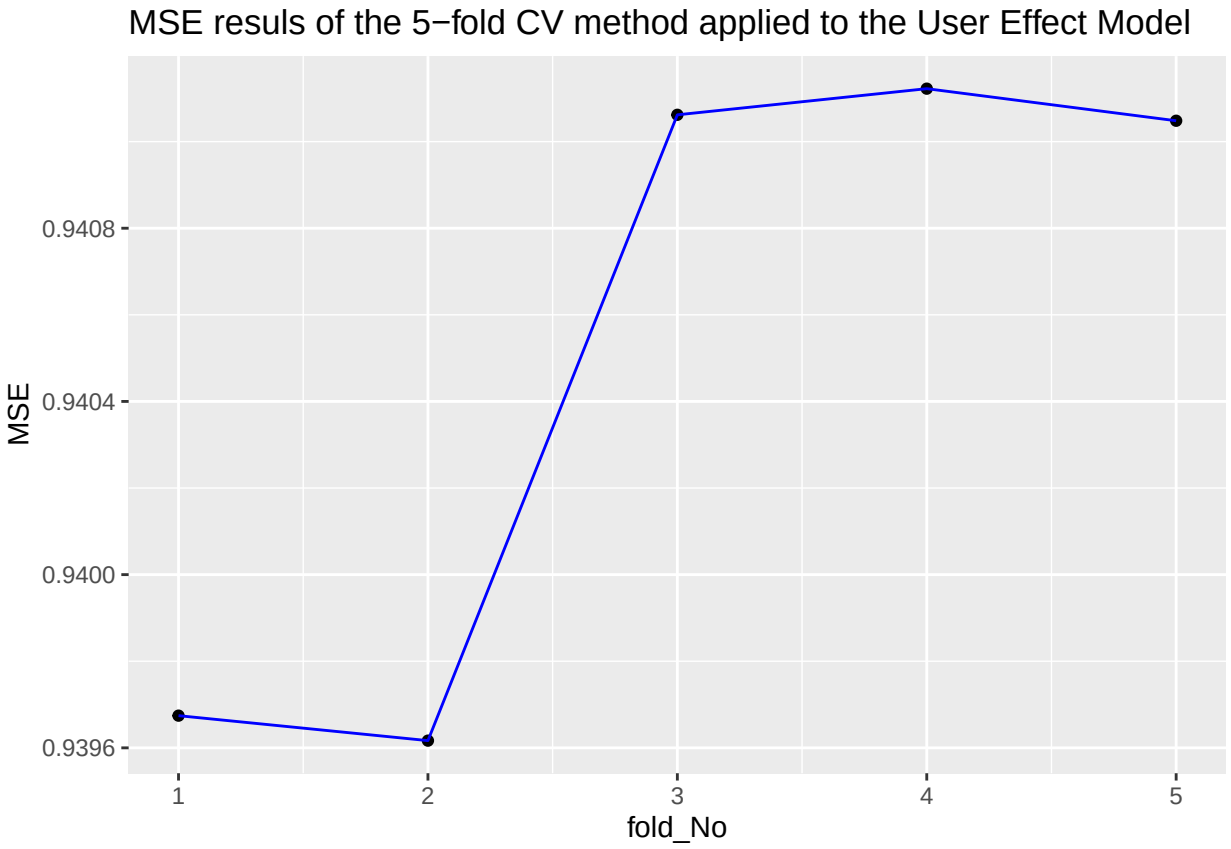
#### 5.1.3.1 mse Function

Calculates *Mean Squared Error* of the set of *residuals* (passed in the r argument) as described in Section 24.1 *Case study: recommendation systems / Loss function* of the *Course Textbook (New Edition)*[1].

##### 5.1.3.1.1 Usage

```
edx.user_effect.MSEs <- sapply(edx_CV, function(cv_fold_dat){
  cv_fold_dat$validation_set |>
    left_join(edx.user_effect, by = "userId") |>
    mutate(resid = rating - clamp(mu + a)) |>
    pull(resid) |> mse()
})
```

```
data.frame(fold_No = 1:5, MSE = edx.user_effect.MSEs) |>
  data.plot(title = "MSE resuls of the 5-fold CV method applied to the User Effect Model",
            xname = "fold_No",
            yname = "MSE")
```



In the code snippet above, we use the custom data visualization function data.plot described in section Data Visualization Functions of *this Appendix*.

#### 5.1.3.1.2 Arguments

- **r:** Numeric vector representing a set of *residuals* (for example of computing a *residuals* vector, see section Usage of *this Description*.

#### 5.1.3.1.3 Source Code

The source code of the mse function is shown below:

```
mse <- function(r) mean(r^2)
```

The source code of the mse function is also available in the (R)MSE-related functions section of the common-helper.functions.R script on *GitHub*.

#### 5.1.3.2 mse_cv Function

Calculates the average value of the *Mean Squared Errors* calculated with the use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

##### 5.1.3.2.1 Usage

```
mse_cv(r_list)
```

##### 5.1.3.2.2 Arguments

- **r_list:** A list of numeric vectors, each representing a set of residuals as described in Section 24.1 *Case Study: Recommender Systems / Loss Function* of the *Course Textbook (New Edition)*[1], computed in the corresponding iteration of *K-Fold Cross-Validation* (the $K$ is 5 for *this Project*).

##### 5.1.3.2.3 Source Code

The source code of the mse_cv function is shown below:

```
mse_cv <- function(r_list) {
  mses <- sapply(r_list, mse(r))
  mean(mses)
}
```

The source code of the mse_cv function is also available in the (R)MSE-related functions section of the common-helper.functions.R script on *GitHub*.

#### 5.1.3.3 rmse Function

Calculates *Root Mean Squared Error* as described in Section 24.1 *Case study: recommendation systems / Loss function* of the *Course Textbook (New Edition)*[1].

##### 5.1.3.3.1 Usage

```
rmse(r)
```

##### 5.1.3.3.2 Arguments

- **r:** Numeric vector representing a set of residuals as described in Section 24.1 *Case study: recommendation systems / Loss function* of the *Course Textbook (New Edition)*[1].

##### 5.1.3.3.3 Source Code

The source code of the rmse function is shown below:

```
rmse <- function(r) sqrt(mse(r))
```

The source code of the rmse function is also available in the (R)MSE-related functions section of the common-helper.functions.R script on *GitHub*.

#### 5.1.3.4 rmse2 Function

Calculates *Root Mean Squared Error* as described in Section 24.1 *Case study: recommendation systems / Loss function* of the *Course Textbook (New Edition)*[1].

##### 5.1.3.4.1 Usage

```
rmse2(true_ratings, predicted_ratings)
```

##### 5.1.3.4.2 Arguments

- **true_ratings:** Numeric vector representing a set of *actual rating* values stored in the validation dataset.

- **predicted_ratings:** A numeric vector representing a set of *predicted rating* values computed from the data stored in the test dataset.

##### 5.1.3.4.3 Source Code

The source code of the rmse2 function is shown below:

```
rmse2 <- function(true_ratings, predicted_ratings) {
  rmse(true_ratings - predicted_ratings)
}
```

The source code of the rmse2 function is also available in the (R)MSE-related functions section of the common-helper.functions.R script on *GitHub*.

### 5.1.4 Result RMSEs Tibble Functions

The complete source code of the functions described in this section are available in the RMSEs Result Tibble section of the common-helper.functions.R script on *GitHub*.

#### 5.1.4.1 CreateRMSEs_ResultTibble Function

Creates a *RMSEs Result Tibble* of type *data frame*.

##### 5.1.4.1.1 Usage

```
# Create the table and add a first row for the Project Objective
RMSEs.ResultTibble <- CreateRMSEs_ResultTibble()
```

##### 5.1.4.1.2 Details

This function is a wrapper for the tibble::tibble function, which is called internally to construct a data frame object representing the *RMSEs Result Tibble* to store the *RMSE* values for every model used in the project to predict movie ratings.

The result data frame, which is returned to the client code, has the following columns:

- *Method* (chr);
- *RMSE* (num);
- *Comment* (chr)

##### 5.1.4.1.3 Value

A tibble returned by the internally called tibble::tibble()

##### 5.1.4.1.4 Source Code

The source code of the CreateRMSEs_ResultTibble function is shown below:

```
CreateRMSEs_ResultTibble <- function(){
  tibble(Method = c("Project Objective"),
         RMSE = project_objective,
         Comment = " ")
}
```

The source code of the CreateRMSEs_ResultTibble function is also available in the RMSEs Result Tibble section of the common-helper.functions.R script on *GitHub*.

#### 5.1.4.2 RMSEs.AddRow Function

Adds a row to the original *RMSEs Result Tibble* passed in the RMSEs argument.

##### 5.1.4.2.1 Usage

```
### Add a row to the RMSE Result Tibble ----------------------------------------
RMSEs.ResultTibble.OMR <- RMSEs.ResultTibble |>
  RMSEs.AddRow("OMR Model",
               mu.RMSE,
               comment = "Overall Mean Rating (OMR) Model")
```

```
RMSE_kable(RMSEs.ResultTibble.OMR)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.864900 | |
| OMR Model | 1.060346 | Overall Mean Rating (OMR) Model |

In the code snippet above, we use the RMSE_kable functions described below in *this Section*.

##### 5.1.4.2.2 Arguments

- ***RMSEs:*** Original *RMSEs Result Tibble* (of type *data frame*) to modify;
- ***method:*** The *Method or Model* being evaluated;
- ***value:*** The *Root Mean Squared Error (RMSE)* value;
- ***comment:*** (Optional, *Empty string* by default) Detailed information about the record or any other comment (if needed);
- ***before:*** (Optional, *Integer*, `NULL` by default) One-based row index where to add the new rows, if NULL (default): after last row;

##### 5.1.4.2.3 Details

This function is a wrapper for the tibble::add_row function, which is called internally to modify a data frame object representing the original *RMSEs Result Tibble*, which stores the *RMSE* values for the models used in the project to predict movie ratings.

For more information about the *RMSEs Result Tibble*, see the Details section of the CreateRMSEs_ResultTibble function description above.

#### 5.1.4.2.4 Value

The modified tibble returned by the internally called tibble::add_row().

#### 5.1.4.2.5 Source Code

The source code of the RMSEs.AddRow function is shown below:

```
RMSEs.AddRow <- function(RMSEs,
                         method,
                         value,
                         comment = "",
                         before = NULL){
  RMSEs |>
    add_row(Method = method,
            RMSE = value,
            Comment = comment,
            .before = before)
}
```

The source code of the RMSEs.AddRow function is also available in the RMSEs Result Tibble section of the common-helper.functions.R script on *GitHub*.

#### 5.1.4.3 RMSEs.AddDiffColumn Function

Adds a calculated column named *Diff* to the *RMSEs Result Tibble* (passed in the RMSEs argument) that contains, for each row, the difference between the value of the *RMSE* column for the previous row and the value of the same column for the current row. This is true for each row except the first, for which the value is always zero.

##### 5.1.4.3.1 Usage

```
total.RMSEs.ResultTibble <- RMSEs.AddDiffColumn(final.MF.RMSEs.ResultTibble)
```

```
RMSE.Total_kable(total.RMSEs.ResultTibble)
```

| Method | RMSE | Diff | Comment |
|---|---|---|---|
| Project Objective | 0.8649000 | 0.0000000 | |
| OMR Model | 1.0603462 | -0.1954462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | 0.0905500 | User Effect (UE) Model |
| UME Model | 0.8732081 | 0.0965881 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | 0.0002351 | Computed for ‘lambda‘ = 0.38745002746582 |
| UMGE Model | 0.8729730 | 0.0000000 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | 0.0000003 | Computed for ‘lambda‘ = 0.0359375 |
| UMGYE Model | 0.8723973 | 0.0005755 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | 0.0002117 | Computed for ‘lambda‘ = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | -0.0000553 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using ‘stats::loess‘ function with ‘degree=1‘ & ‘span=0.75‘ parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | 0.0011279 | UMGYDE Model computed using function call: ‘loess(degree = 0, span = 0.00109375)‘ |
| Tuned UMGYDE.d1 Model | 0.8708295 | 0.0002836 | UMGYDE Model computed using function call: ‘loess(degree = 1, span = 0.00109375)‘ |
| Tuned UMGYDE.d2 Model | 0.8709886 | -0.0001591 | UMGYDE Model computed using function call: ‘loess(degree = 2, span = 0.00153125)‘ |
| Tuned UMGYDE Best Model | 0.8707850 | 0.0002036 | UMGYDE Model computed using ‘loess‘ function ca with the best degree & span values. |
| Regularized UMGYDE Model | 0.8707750 | 0.0000100 | The best tuned and regularized UMGYDE Model. |
| Best UMGYDE Model (Final Test) | 0.8804225 | -0.0096476 | Final Holdout Test of the best tuned and regularized UMGYDE Model. |
| MF (Final Test) | 0.7875754 | 0.0928471 | Matrix Factorization of the Best Model Residuals, F Holdout Test |

In the code snippet above, we use the RMSE.Total_kable functions described below in *this Section*.

##### 5.1.4.3.2 Arguments

- ***RMSEs:*** *RMSEs Result Tibble* to modify;

#### 5.1.4.3.3 Details

This function is a wrapper for the tibble::add_column function, which is called internally to modify a data frame object representing the original *RMSEs Result Tibble*. It also internally uses the RMSEs.AddRow function described above to perform the necessary intermediate calculations.

For more information about the *RMSEs Result Tibble*, see the Details section of the CreateRMSEs_ResultTibble function description above.

#### 5.1.4.3.4 Value

The modified tibble returned by the internally called tibble::add_column().

#### 5.1.4.3.5 Source Code

The source code of the RMSEs.AddDiffColumn function is shown below:

```
RMSEs.AddDiffColumn <- function(RMSEs){
  RMSEs.Diff <- RMSEs |>
    RMSEs.AddRow(NULL, project_objective, before = 1) #/>

  RMSEs.Diff <- RMSEs.Diff[-nrow(RMSEs.Diff),]

  RMSEs |>
  add_column(Diff = RMSEs.Diff$RMSE - RMSEs$RMSE,
             .before = "Comment")

}
```

The source code of the RMSEs.AddDiffColumn function is also available in the RMSEs Result Tibble section of the common-helper.functions.R script on *GitHub*.

### 5.1.4.4 RSME.tibble.col_width Function

Converts the *RMSEs Result Tibble* column width value passed in argument x to a character string compatible for use as the `width` argument value to pass to the kableExtra::column_spec function.

#### 5.1.4.4.1 Usage

```
  RMSEs |>
    kable(align='lcl', booktabs = T, padding = 5) |>
    row_spec(0, bold = T) |>
    column_spec(column = 1, width = RSME.tibble.col_width(col1width)) |>
    column_spec(column = 2, width = RSME.tibble.col_width(col2width)) |>
    column_spec(column = 3, width = RSME.tibble.col_width(col3width))
```

In the code snippet above, the function `RSME.tibble.col_width` is called internally from the RMSE_kable function described below in *this Section*.

#### 5.1.4.4.2 Arguments

- ***x:*** (Numeric) Column width value to be formatted;

#### 5.1.4.4.3 Details

This is a utility function, a wrapper for the msg.glue function described in section Utility Functions of *this Appendix*, which is called internally to format the value passed in argument x for subsequent passing to the function kableExtra::column_spec as a value of the `width` argument.

As a utility, this function is intended for use in other user-defined functions, such as the RMSE_kable (as shown above in the Usage section of *this Description*) and RMSE.Total_kable described below in *this Section*.

#### 5.1.4.4.4 Value

A character string representing the formatted column width value.

#### 5.1.4.4.5 Source Code

The source code of the RSME.tibble.col_width function is shown below:

```
RSME.tibble.col_width <- function(x){
  "%1em" |> msg.glue(x)
}
```

The source code of the RSME.tibble.col_width function is also available in the RMSEs Result Tibble section of the common-helper.functions.R script on *GitHub*.

#### 5.1.4.5 RMSE_kable Function

Prints the *RMSEs Result Tibble*.

It is assumed that the tibble has the following three columns:

1. *Method*;
2. *RMSE*;
3. *Comment*;

##### 5.1.4.5.1 Usage

```
RMSE_kable(RMSEs.ResultTibble.OMR)
```

| Method | RMSE | Comment |
|---|---|---|
| Project Objective | 0.864900 | |
| OMR Model | 1.060346 | Overall Mean Rating (OMR) Model |

##### 5.1.4.5.2 Arguments

- ***RMSEs:*** *RMSEs Result Tibble* to print;
- ***col1width:*** Width of the first column (Optional, 15 by default);
- ***col2width:*** Width of the second column (Optional, 5 by default);
- ***col3width:*** Width of the third column (Optional, 30 by default).

##### 5.1.4.5.3 Details

This function is a wrapper for the knitr::kable function, which is called internally to create a table (based on the *RMSEs Result Tibble* data), formatted for output in the project report.

The kableExtra::column_spec function, along with the RSME.tibble.col_width function described above, are also used internally to specify the width of the table columns.

For more information about the *RMSEs Result Tibble*, see the Details section of the CreateRMSEs_ResultTibble function description above.

##### 5.1.4.5.4 Value

The table created by the internally called knitr::kable().

#### 5.1.4.5.5 Source Code

The source code of the RMSE_kable function is shown below:

```
RMSE_kable <- function(RMSEs,
                       col1width = 15,
                       col2width = 5,
                       col3width = 30){
  RMSEs |>
    kable(align='lcl', booktabs = T, padding = 5) |>
    row_spec(0, bold = T) |>
    column_spec(column = 1, width = RSME.tibble.col_width(col1width)) |>
    column_spec(column = 2, width = RSME.tibble.col_width(col2width)) |>
    column_spec(column = 3, width = RSME.tibble.col_width(col3width))
}
```

The source code of the RMSE_kable function is also available in the RMSEs Result Tibble section of the common-helper.functions.R script on *GitHub*.

#### 5.1.4.6 RMSE.Total_kable Function

Prints the *Final RMSEs Result Tibble*.

It is assumed that the tibble has the following four columns:

1. *Method*;
2. *RMSE*;
3. *Diff*;
4. *Comment*;

##### 5.1.4.6.1 Usage

```
RMSE.Total_kable(total.RMSEs.ResultTibble)
```

| Method | RMSE | Diff | Comment |
|---|---|---|---|
| Project Objective | 0.8649000 | 0.0000000 | |
| OMR Model | 1.0603462 | -0.1954462 | Overall Mean Rating (OMR) Model |
| UE Model | 0.9697962 | 0.0905500 | User Effect (UE) Model |
| UME Model | 0.8732081 | 0.0965881 | User+Movie Effect (UME) Model |
| Regularized UME Model | 0.8729730 | 0.0002351 | Computed for ‘lambda‘ = 0.38745002746582 |
| UMGE Model | 0.8729730 | 0.0000000 | User+Movie+Genre Effect (UMGE) Model |
| Regularized UMGE Model | 0.8729728 | 0.0000003 | Computed for ‘lambda‘ = 0.0359375 |
| UMGYE Model | 0.8723973 | 0.0005755 | User+Movie+Genre+Year Effect (UMGYE) Model |
| Regularized UMGYE Model | 0.8721857 | 0.0002117 | Computed for ‘lambda‘ = 233.77668762207 |
| UMGYDE (Default) Model | 0.8722410 | -0.0000553 | User+Movie+Genre+Year+Day Effect (UMGYDE) Model computed using ‘stats::loess‘ function with ‘degree=1‘ & ‘span=0.75‘ parameter values. |
| Tuned UMGYDE.d0 Model | 0.8711131 | 0.0011279 | UMGYDE Model computed using function call: ‘loess(degree = 0, span = 0.00109375)‘ |
| Tuned UMGYDE.d1 Model | 0.8708295 | 0.0002836 | UMGYDE Model computed using function call: ‘loess(degree = 1, span = 0.00109375)‘ |
| Tuned UMGYDE.d2 Model | 0.8709886 | -0.0001591 | UMGYDE Model computed using function call: ‘loess(degree = 2, span = 0.00153125)‘ |
| Tuned UMGYDE Best Model | 0.8707850 | 0.0002036 | UMGYDE Model computed using ‘loess‘ function ca with the best degree & span values. |
| Regularized UMGYDE Model | 0.8707750 | 0.0000100 | The best tuned and regularized UMGYDE Model. |
| Best UMGYDE Model (Final Test) | 0.8804225 | -0.0096476 | Final Holdout Test of the best tuned and regularized UMGYDE Model. |
| MF (Final Test) | 0.7875754 | 0.0928471 | Matrix Factorization of the Best Model Residuals, F Holdout Test |

#### 5.1.4.6.2 Arguments

- ***RMSEs:*** The *Final RMSEs Result Tibble* to print;
- ***col1width:*** (Optional, 15 by default) Width of the first column;
- ***col2width:*** (Optional, 7 by default) Width of the second column;
- ***col3width:*** (Optional, 5 by default) Width of the third column;
- ***col4width:*** (Optional, 25 by default) Width of the fourth column;

#### 5.1.4.6.3 Details

This function is a wrapper for the RMSE_kable function described above, which is called internally to print the table, created from the final *RMSEs Result Tibble* data.

It is assumed that the *Final RMSEs Result Tibble* is previously constructed from the original *RMSEs Result Tibble* using the RMSEs.AddDiffColumn function described above.

The kableExtra::column_spec function, along with the RSME.tibble.col_width function described above, are also used internally to set the width of the last column of the table.

For more information about the *RMSEs Result Tibble*, see the Details section of the CreateRMSEs_ResultTibble function description above.

#### 5.1.4.6.4 Value

The table returned by the internally called RMSE_kable().

#### 5.1.4.6.5 Source Code

The source code of the RMSE.Total_kable function is shown below:

```
RMSE.Total_kable <- function(RMSEs,
                        col1width = 15,
                        col2width = 7,
                        col3width = 5,
                        col4width = 25){
  RMSEs |>
    RMSE.Total_kable(col1width,
                col2width,
                col3width) |>
    column_spec(column = 4,
                 width = RSME.tibble.col_width(col4width))
}
```

The source code of the RMSE.Total_kable function is also available in the RMSEs Result Tibble section of the common-helper.functions.R script on *GitHub*.

### 5.1.5 Model Tuning Utils

The complete source code of the functions described in this section are available in the Regularization section of the common-helper.functions.R script on *GitHub*.

#### 5.1.5.1 mean_reg Function

Calculates the arithmetic mean taking into account the *regularization parameter* $\lambda$ passed in the lambda argument. We will call it the *penalized mean*.

##### 5.1.5.1.1 Usage

```
UM.effect <- train_set |>
  left_join(edx.user_effect, by = "userId") |>
  mutate(resid = rating - (mu + a)) |>
  group_by(movieId) |>
  summarise(b = mean_reg(resid, lambda), n = n())
```

In the sample code snippet above, the `mean_reg` function is called internally from the train_user_movie_effect function described below in section UME Model: Utility Functions of *this Appendix*.

##### 5.1.5.1.2 Arguments

- **vals:** Values (usually a numeric vector) to calculate the *penalized mean*;
- **lambda:** (Optional, *Numeric*, 0 by default) *Regularization parameter* $\lambda$.
- **na.rm:** (Optional, *Boolean*, `TRUE` by default) Logical value indicating whether `NA` values should be stripped before the computation proceeds.

#### 5.1.5.1.3 Details

This function is used in the *regularization technique* (described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14]) to calculate the *penalized mean* of the values passed in the vals argument as follows:

$$\mu(\lambda) = \frac{1}{\lambda + N} \sum_{i=1}^{N} x_i$$

This is a helper function intended for use in other helper functions (such as, for instance, train_user_movie_effect as shown above in the Usage section of *this Description*) that compute *penalized estimates* during the *regularization* process of the models being trained[14]. All these functions are described below in section Models Training: Support Functions of *this Appendix*.

When the lambda parameter is `0` (meaning the $\lambda = 0$), the function calculates the simple arithmetic mean:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$

If the lambda parameter is `0` (the default), the function is equivalent to the standard R function base::mean, calling with the parameter `trim = 0` (the default for the `base::mean` function).

#### 5.1.5.1.4 Value

(Numeric) *Penalized mean* of the values passed in the vals argument.

#### 5.1.5.1.5 Source Code

The source code of the mean_reg function is shown below:

```
mean_reg <- function(vals, lambda = 0, na.rm = TRUE){
  if (is.na(lambda)) {
    stop("Function: mean_reg
`lambda` is `NA`")
  }

  names(lambda) <- NULL
  sums <- sum(vals, na.rm = na.rm)
  N <- ifelse(na.rm, sum(!is.na(vals)), length(vals))
  sums/(N + lambda)
}
```

The source code of the mean_reg is also available in the Regularization section of the common-helper.functions.R script on *GitHub*.

#### 5.1.5.2 get_fine_tune.param.endpoints.idx Function

Implements an algorithm that finds in the *RMSE* column of the data frame passed in the preset.result argument the indices of the pair of the nearest neighboring elements with values greater than the minimum to the left and right of the element (or group of elements) with the minimum value.

The algorithm assumes that the set of values in the *RMSE* column is a monotonically non-increasing function of the index until the minimum is reached and monotonically non-decreasing thereafter. That is, it is assumed that the function has only one minimum on the given set.

##### 5.1.5.2.1 Usage

The following code snippet creates the object `cv.UME.preset.result` as a result of the tune.model_param function call, which we can subsequently use to extract a value to pass in the preset.result argument to the function *being described here*:

```
lambdas <- seq(0, 1, 0.1)
cv.UME.preset.result <-
  tune.model_param(lambdas, regularize.test_lambda.UM_effect.cv)
```

```
str(cv.UME.preset.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    8 obs. of  2 variables:
##   ..$ RMSE           : num [1:8] 0.873 0.873 0.873 0.873 0.873 ...
##   ..$ parameter.value: num [1:8] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
##  $ best_result : Named num [1:2] 0.4 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

The *custom helper function* tune.model_param used in the *code snippet above* is described below in *this Section*.

The following code snippet provides an example of passing the data contained in the object created in the *code snippet above* to the preset.result argument of the function *being described*:

```
  preset.result <- cv.UME.preset.result$tuned.result
  print(preset.result)
```

```
##        RMSE parameter.value
## 1 0.8732081             0.0
## 2 0.8732069             0.1
## 3 0.8732062             0.2
## 4 0.8732058             0.3
## 5 0.8732057             0.4
## 6 0.8732058             0.5
## 7 0.8732062             0.6
## 8 0.8732067             0.7
```

```
# Internal call from the user-defined function `get_fine_tune.param.endpoints`
preset.result.idx <- get_fine_tune.param.endpoints.idx(preset.result)
preset.result.idx
```

```
## start   end  best
##     4     6     5
```

In the above code snippet, the `get_fine_tune.param.endpoints.idx` function is called internally from the get_fine_tune.param.endpoints function described below in *this Section*.

#### 5.1.5.2.2 Arguments

- **preset.result:** Data frame that has an *RMSE* column containing the `RMSE` values corresponding to the *regularization parameter* $\lambda$ values for the model being regularized, as described in section Details of the tune.model_param function description below.

#### 5.1.5.2.3 Details

This is an auxiliary function intended to be called internally from the other user-defined functions, such as the get_fine_tune.param.endpoints (as shown above in the Usage section of *this Description*) and model.tune.param_range (both described below in *this Section*).

The element stored in the `$tuned.result` variable of the *list* object returned by the tune.model_param function (described below *in this Section*), is a *data frame* suitable for use as the value of the preset.result argument.

#### 5.1.5.2.4 Value

Vector containing the index of the first element corresponding to the minimum `RMSE` value in the *RMSE* column of the data frame passed in the preset.result argument, along with the indices of the pair of the nearest neighboring elements with values greater than the minimum to the left and right of the element (or group of elements) with the minimum value:

```
c(start, # interval start index
  end,   # interval end index
  best)  # index of the best `RMSE`
```

The Usage section of *this Description* above provides an example of the value returned by the function *being described*.

#### 5.1.5.2.5 Source Code

The source code of the get_fine_tune.param.endpoints.idx function is shown below:

```
get_fine_tune.param.endpoints.idx <- function(preset.result) {
  best_RMSE <- min(preset.result$RMSE)
  best_RMSE.idx <- which.min(preset.result$RMSE)
  # best_lambda <- preset.result$parameter.value[best_RMSE.idx]

  preset.result.N <- length(preset.result$RMSE)
  i <- best_RMSE.idx
  j <- i

  while (i > 1) {
    i <- i - 1

    if (preset.result$RMSE[i] > best_RMSE) {
      break
    }
  }

  while (j < preset.result.N) {
    j <- j + 1

    if (preset.result$RMSE[j] > best_RMSE) {
      break
    }
  }

  c(start = i,
    end = j,
    best = best_RMSE.idx)
}
```

The source code of the get_fine_tune.param.endpoints.idx function is also available in the Regularization section of the common-helper.functions.R script on *GitHub*.

#### 5.1.5.3 get_fine_tune.param.endpoints Function

Implements an algorithm that finds in the *RMSE* column of the data frame passed in the preset.result argument the pair of the nearest neighboring elements with values greater than the minimum to the left and right of the element (or group of elements) with the minimum value.

The algorithm assumes that the set of values in the *RMSE* column is a monotonically non-increasing function of the index until the minimum is reached and monotonically non-decreasing thereafter. That is, it is assumed that the function has only one minimum on the given set.

##### 5.1.5.3.1 Usage

The following code snippet creates the object `cv.UME.preset.result` as a result of the tune.model_param function call, which we can subsequently use to extract a value to pass in the preset.result argument to the function *being described here*:

```
lambdas <- seq(0, 1, 0.1)
cv.UME.preset.result <-
  tune.model_param(lambdas, regularize.test_lambda.UM_effect.cv)
```

```
str(cv.UME.preset.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    8 obs. of  2 variables:
##   ..$ RMSE           : num [1:8] 0.873 0.873 0.873 0.873 0.873 ...
##   ..$ parameter.value: num [1:8] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
##  $ best_result : Named num [1:2] 0.4 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

```
# Object to be passed to the `preset.result` argument of the `get_fine_tune.param.endpoints` function:
print(cv.UME.preset.result$tuned.result)
```

```
##        RMSE parameter.value
## 1 0.8732081             0.0
## 2 0.8732069             0.1
## 3 0.8732062             0.2
## 4 0.8732058             0.3
## 5 0.8732057             0.4
## 6 0.8732058             0.5
## 7 0.8732062             0.6
## 8 0.8732067             0.7
```

The *custom helper function* tune.model_param used in the *code snippet above* is described below in *this Section*.

In the following code snippet, the data stored in the `$tuned.result` variable of the *list* object returned by the tune.model_param function in the *code snippet above*, is passed in the preset.result argument to the function *being described*:

```
endpoints <-
  get_fine_tune.param.endpoints(cv.UME.preset.result$tuned.result)

endpoints
```

```
## start   end  best
##   0.3   0.5   0.4
```

#### 5.1.5.3.2 Arguments

- **preset.result:** Data frame that has an *RMSE* column containing the `RMSE` values corresponding to the *regularization parameter* $\lambda$ values for the model being regularized, as described in section Details of the tune.model_param function description below.

#### 5.1.5.3.3 Details

This function internally calls get_fine_tune.param.endpoints.idx function described above in *this Section* to figure out the indices of the elements, the values of which have to be determined.

The element stored in the `$tuned.result` variable of the *list* object returned by the tune.model_param function (described below *in this Section*), is a *data frame* suitable for use as the value of the preset.result argument.

#### 5.1.5.3.4 Value

Vector containing the *regularization parameter* $\lambda$ value corresponding to the first element with the minimum `RMSE` value in the *RMSE* column of the data frame passed in the preset.result argument, along with the pair of the $\lambda$ values corresponding to the nearest neighboring elements with values greater than the minimum `RMSE` to the left and right of the element (or group of elements) with the minimum `RMSE` value:

The Usage section of *this Description* above provides an example of the value returned by the function *being described*.

#### 5.1.5.3.5 Source Code

The source code of the get_fine_tune.param.endpoints function is shown below:

```
get_fine_tune.param.endpoints <- function(preset.result) {

  preset.result.idx <- get_fine_tune.param.endpoints(preset.result)

  i <- preset.result.idx["start"]
  j <- preset.result.idx["end"]
  best.idx <- preset.result.idx["best"]

  c(start = preset.result$parameter.value[i],
    end = preset.result$parameter.value[j],
    best = preset.result$parameter.value[best.idx])
}
```

The source code of the get_fine_tune.param.endpoints function is also available in the Regularization section of the common-helper.functions.R script on *GitHub*.

#### 5.1.5.4 get_best_param.result Function

Selects the best values from a pair of matched vectors: a vector of sorted *tuning parameter* values and a vector of corresponding *Root Mean Squared Errors (RMSE)* values passed in the param_values and RMSEs arguments, respectively.

##### 5.1.5.4.1 Usage

In the following code snippet, the `get_best_param.result` function is called internally from the model.tune.param_range function described below in *this Section*:

```
    param_values.best_result <-
      get_best_param.result(tuned.result$parameter.value,
                            tuned.result$RMSE)

    put_log2("Function `model.tune.param_range`:
Currently reached best RMSE for `parameter value = %1`: %2",
             param_values.best_result["param.best_value"],
             param_values.best_result["best_RMSE"])
```

In the code snippet above, we use the custom function put_log2 described in section Logging Functions of *this Appendix*.

##### 5.1.5.4.2 Arguments

- **param_values:** *Numeric vector* containing a sorted list of the *tuning parameter* values to search for the best one during the *Model Regularization* process;

- **RMSEs:** *Numeric vector* containing a list of *RMSE* values computed during the *Model Regularization* process.

##### 5.1.5.4.3 Details

This is a *auxiliary function* used to select the *best values* during the *Model Parameter Tuning* process (including the *Model Regularization* procedure that utilizes the *penalized approach* to regularize predictions as explained in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14]).

With this approach, the best value of the *tuning parameter* is considered to be the value corresponding to the minimum value of the *RMSE* for the model being tested.

In *This Project*, this function is used internally in the model.tune.param_range custom helper function (as shown above in the Usage section of *this Description*) described below in *this Section*.

#### 5.1.5.4.4 Value

*Numeric vector* containing a pair of values: the *best tuning parameter* value and corresponding *minimal RMSE*.

#### 5.1.5.4.5 Source Code

The source code of the get_best_param.result function is shown below:

```
get_best_param.result <- function(param_values, RMSEs){
  best_pvalue_idx <- which.min(RMSEs)
  c(param.best_value = param_values[best_pvalue_idx],
    best_RMSE = RMSEs[best_pvalue_idx])
}
```

The source code of the get_best_param.result function is also available in the Regularization section of the common-helper.functions.R script on *GitHub*.

#### 5.1.5.5 tune.model_param Function

Implements an algorithm for selecting the best value of the *tuning parameter* from the sorted *parameter vector* passed in the param_values argument.

##### 5.1.5.5.1 Usage

```
  lambdas <- seq(0, 1, 0.1)
  cv.UME.preset.result <-
    tune.model_param(lambdas, regularize.test_lambda.UM_effect.cv)
```

```
print(cv.UME.preset.result)
```

```
## $tuned.result
##        RMSE parameter.value
## 1 0.8732081             0.0
## 2 0.8732069             0.1
## 3 0.8732062             0.2
## 4 0.8732058             0.3
## 5 0.8732057             0.4
## 6 0.8732058             0.5
## 7 0.8732062             0.6
## 8 0.8732067             0.7
##
## $best_result
## param.best_value        best_RMSE
##        0.4000000        0.8732057
```

In the *code snippet* above, the `tune.model_param` function accepts a pointer to the auxiliary function regularize.test_lambda.UM_effect.cv (described below in section UME Model: Regularization of *this Appendix*), as a value of an argument fn_tune.test.param_value.

##### 5.1.5.5.2 Arguments

- **param_values:** Numeric vector containing a sorted list of parameter values to search for the best one during the model regularization process;
- **fn_tune.test.param_value:** Model-specific helper function that validates the model by computing the model's *RMSE* value for a given parameter value;
- **break.if_min:** (Optional, `TRUE` by default) Boolean value that determines whether the algorithm should terminate after completing the number of steps specified by the steps.beyond_min argument from the moment the minimum `RMSE` value is found;
- **steps.beyond_min:** (Optional, 2 by default; only takes effect if break.if_min is `TRUE`) Specifies the number of steps after finding the minimum *RMSE* value after which the algorithm should terminate.

#### 5.1.5.5.3 Details

With the approach used in this algorithm, the best value of the *tuning parameter* is considered to be the value corresponding to the minimum value of the *Root Mean Square Error (RMSE)* for the model being tested.

To find the best value, the algorithm iterates over the values from the list passed in the param_values argument, computing the *RMSE* of the model for each of them, using the helper function specified in the fn_tune.test.param_value argument.

The algorithm continues iterating until the end of the parameter values list if break.if_min is `FALSE`.

Otherwise, the algorithm terminates after the number of cycles specified by the argument steps.beyond_min has completed since the minimum *RMSE* value was reached.

The algorithm assumes that the set of the *RMSE* values is a monotonically non-increasing function of the *tuning parameter* until the minimum is reached and monotonically non-decreasing thereafter. That is, it is assumed that the function can have only one minimum on the given set.

#### 5.1.5.5.4 Value

This function returns a list of two objects:

```
list(tuned.result = data.frame(RMSE = RMSEs_tmp,
                               parameter.value = param_vals_tmp),
     best_result = param_values.best_result)
```

The objects comprising the return list are as follows:

- `best_result`: Vector containing a *pair of values*: the found *best value of the tuning parameter* for which the *RMSE* value is minimal, along with the minimal *RMSE* value itself;
- `tuned.result`: Data frame containing two variables:
  - *parameter.value*: A vector containing all processed values of the *tuning parameter*;
  - *RMSE*: A vector containing the *RMSE* values corresponding to the processed values of the *tuning parameter*.

#### 5.1.5.5.5 Source Code

Below is the simplified version of the source code of the tune.model_param function:

```
tune.model_param <- function(param_values,
                             fn_tune.test.param_value,
                             break.if_min = TRUE,
                             steps.beyond_min = 2){
  n <- length(param_values)
  param_vals_tmp <- numeric()
  RMSEs_tmp <- numeric()
  RMSE_min <- Inf
  i_max.beyond_RMSE_min <- Inf
  prm_val.best <- NA

  # ...

  for (i in 1:n) {
    put_log1("Function: `tune.model_param`: Iteration %1", i)
    prm_val <- param_values[i]
    param_vals_tmp[i] <- prm_val

    RMSE_tmp <- fn_tune.test.param_value(prm_val)
    RMSEs_tmp[i] <- RMSE_tmp

    plot(param_vals_tmp[RMSEs_tmp], RMSEs_tmp[RMSEs_tmp])

    if(RMSE_tmp > RMSE_min){
      warning("Function: `tune.model_param`: `RSME` reached its minimum: ",
              RMSE_min, "for parameter value: ", prm_val)

      if (i > i_max.beyond_RMSE_min) {
        warning("Function: `tune.model_param`:
Operation is breaked (after `RSME` reached its minimum) on the following step: ", i)
        break
      }
      next
    }

    RMSE_min <- RMSE_tmp
    prm_val.best <- prm_val

    if (break.if_min) {
      i_max.beyond_RMSE_min <- i + steps.beyond_min
    }
  }

  param_values.best_result <- c(param.best_value = prm_val.best,
                                best_RMSE = RMSE_min)

  list(tuned.result = data.frame(RMSE = RMSEs_tmp,
                                 parameter.value = param_vals_tmp),
       best_result = param_values.best_result)
}
```

The complete source code of the tune.model_param function is available in the Regularization section of the common-helper.functions.R script on *GitHub*.

#### 5.1.5.6 model.tune.param_range Function

Implements an algorithm for fine-tuning the model under test by figuring out the most precise value possible of the *tuning parameter* over a given interval determined by the *start* and *end* values of the interval passed in the loop_starter argument.

##### 5.1.5.6.1 Usage

```
endpoints <-
  get_fine_tune.param.endpoints(cv.UME.preset.result$tuned.result)

UM_effect.loop_starter <- c(endpoints["start"],
                            endpoints["end"],
                            8)
```

```
UM_effect.loop_starter
```

```
## start   end
##   0.3   0.5   8.0
```

In the code snippet above, we use the user-defined auxiliary function get_fine_tune.param.endpoints described above in *this Section*.

```
UME.rglr.fine_tune.cache.base_name <- "UME.rglr.fine-tune"

UME.rglr.fine_tune.results <-
  model.tune.param_range(UM_effect.loop_starter,
                         UME.rglr.fine_tune.cache.path,
                         UME.rglr.fine_tune.cache.base_name,
                         regularize.test_lambda.UM_effect.cv)

UME.rglr.fine_tune.RMSE.best <- UME.rglr.fine_tune.results$best_result["best_RMSE"]
```

```
str(UME.rglr.fine_tune.results)
```

```
## List of 3
##  $ best_result           : Named num [1:2] 0.387 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
##  $ param_values.endpoints: Named num [1:3] 3.87e-01 3.87e-01 9.54e-07
##   ..- attr(*, "names")= chr [1:3] "" "" ""
##  $ tuned.result          :'data.frame':  9 obs. of  2 variables:
##   ..$ parameter.value: num [1:9] 0.387 0.387 0.387 0.387 0.387 ...
##   ..$ RMSE           : num [1:9] 0.873 0.873 0.873 0.873 0.873 ...
```

```
print(UME.rglr.fine_tune.results)
```

```
## $best_result
## param.best_value        best_RMSE
##        0.3874500        0.8732057
##
## $param_values.endpoints
##
## 3.874491e-01 3.874557e-01 9.536743e-07
##
## $tuned.result
##   parameter.value      RMSE
## 1       0.3874481 0.8732057
## 2       0.3874491 0.8732057
## 3       0.3874500 0.8732057
## 4       0.3874510 0.8732057
## 5       0.3874519 0.8732057
## 6       0.3874529 0.8732057
## 7       0.3874538 0.8732057
## 8       0.3874548 0.8732057
## 9       0.3874557 0.8732057
```

#### 5.1.5.6.2 Arguments

***loop_starter***

A numeric vector of the form `c(start, end, dvs)`, where `start` and `end` are the endpoints of the interval over which the value of the *tuning parameter* that minimizes *RMSE* has to be found.

The `dvs` is a divisor used to calculate the *step size* for partitioning the interval into a sequence of parameter values, among which the value that minimizes *RMSE* has to be found.

So the *step size* of the sequence is calculated as follows:

$$step = \frac{end - start}{dvs}$$

The sequence obtained as a result of the transformation is equivalent to the one generated by the function base::seq as follows:

```
seq(start, end, step)
```

In the function body, the *step size* is stored in the local variable `seq_increment`, and the values of *start*, *end*, and *dvs* are stored in the local variables `seq_start`, `seq_end`, and `interval_divisor`, respectively.

In fact, the seq function is called internally to generate the sequence during the algorithm's execution by the following line of code:

```
    test_param_vals <- seq(seq_start, seq_end, seq_increment)
```

***tune__dir__path***

To improve performance, the algorithm caches intermediate results in the file system. The value passed in *this argument* specifies the path to the directory where the files are cached.

***cache__file__base__name***

The algorithm generates unique names for cache files based on this and the loop_starter arguments' values, as well as a few other intermediate values calculated at runtime.

***fn__tune.test.param__value***

Name of a model-specific helper function to pass to the same-named argument of the tune.model_param function (see description above) that is called internally during the execution of the algorithm.

***max.identical.min__RMSE.count***

(Optional, 4 by default) If more than one identical minimum *RMSE* value is calculated during execution, the number of identical minimums is limited by the value of *this argument*. When it is reached, the algorithm considers the task execution to be complete.

***endpoint.min__diff***

(Optional, 0 by default) Defines the sensitivity threshold for determining the neighborhood boundaries of the minimum `RMSE` value (see the Details section below for more details).

***break.if__min***

(Optional, `TRUE` by default) Boolean value to pass to the same-named argument of the tune.model_param function (see description above) that is called internally during the execution of the algorithm.

***steps.beyond__min***

(Optional, 2 by default) Numeric value to pass to the same-named argument of the tune.model_param function (see description above) that is called internally during the execution of the algorithm.

#### 5.1.5.6.3 Details

With the approach used in this algorithm, the best value of the *tuning parameter* (we call this the *best value of the tuning parameter*) is considered to be the value corresponding to the minimum value of the *Root Mean Square Error (RMSE)* for the model being tested.

During the execution, the algorithm organizes a cycle consisting of the following steps:

1. Calculates the step size for partitioning the interval into a sequence of values and save it to the local variable `seq_increment` as described in section loop_starter argument description above using the following line of code:

```
    seq_increment <- (seq_end - seq_start)/interval_divisor
```

2. As shown in the following code snippet, checks if the *step size* of the generated sequence is less than the minimum allowed value (in the current implementation, we use `1e-13`):

```
    if (seq_increment < 0.0000000000001) {
      warning("Function `model.tune.param_range`:
parameter value increment is too small.")

      put_log2("Function `model.tune.param_range`:
Final best RMSE for `parameter value = %1`: %2",
               param_values.best_result["param.best_value"],
               param_values.best_result["best_RMSE"])

      put(param_values.best_result)
      break
    }
```

If the check condition is met, the loop is terminated, and the execution proceeds to the Finalizing Execution section of the code described in Finalizing Execution subsection below:

Otherwise, proceeds to the next step:

In the code snippet above, we use the custom function put_log2 described in section Logging Functions of *this Appendix*.

3. (The *step size* is no less than the minimum allowed value: `seq_increment >= 1e-13`) Creates a vector containing a sequence of parameter values (as described in section loop_starter argument description above) using the following line of code:

```
    test_param_vals <- seq(seq_start, seq_end, seq_increment)
```

4. Generates a unique name for the current iteration's cache file based on the values passed in the cache_file_base_name and loop_starter arguments, along with the values stored in the local variables `seq_start`, `seq_end`, and `interval_divisor`, concatenates the name with the path passed in the tune_dir_path argument, and stores the value in the local variable `file_path_tmp`.

The following code snippet performs this step of execution:

```
    file_name_tmp <- cache_file_base_name |>
      str_c("_") |>
      str_c(as.character(loop_starter[1])) |>
      str_c("_") |>
      str_c(as.character(loop_starter[3])) |>
      str_c("_") |>
      str_c(as.character(interval_divisor)) |>
      str_c(".") |>
      str_c(as.character(seq_start)) |>
      str_c("-") |>
      str_c(as.character(seq_end)) |>
      str_c(".RData")

    file_path_tmp <- file.path(tune_dir_path, file_name_tmp)

    put_log1("Function `model.tune.param_range`:
File path generated: %1", file_path_tmp)
```

In the code snippet above, we use the custom logging function put_log1 described above in section Logging Functions of *this Appendix*.

5. Checks whether a file with the full name stored in the `file_path_tmp` variable exists, and if so, loads all the necessary information for the current iteration from it and goes to step 8. Otherwise, proceeds to the next step.

The following code snippet performs this step of execution:

```
    if (file.exists(file_path_tmp)) {
      put_log1("Function `model.tune.param_range`:
Loading tuning data from file: %1...", file_path_tmp)

      load(file_path_tmp)
      put_log1("Function `model.tune.param_range`:
Tuning data has been loaded from file: %1", file_path_tmp)

      tuned.result <- tuned_result$tuned.result
    } else {
      # ...
    }
```

6. Calls the helper function tune.model_param and stores the returned result in the local variable `tuned_result`, passing the vector created earlier in step 3 to the argument param_values along with the values of the remaining arguments passed in the following arguments of the same name to *this function*:

- fn_tune.test.param_value;
- break.if_min;
- steps.beyond_min.

The following code snippet performs this step of execution:

```
      tuned_result <- tune.model_param(test_param_vals,
                                       fn_tune.test.param_value,
                                       break.if_min,
                                       steps.beyond_min)
```

7. Extracts the data frame element stored in the variable `tuned_result$tuned.result` from the *list object* saved in the local variable `tuned_result` in the previous step and stores it in the local variable `tuned.result`. Then stores the variable `tuned_result` (along with the necessary auxiliary information) in the cache file with the full name created earlier in step 4.

The following code snippet performs this step of execution:

```
      tuned.result <- tuned_result$tuned.result

      save(tuned_result,
           param.best_value,
           seq_increment,
           interval_divisor,
           file = file_path_tmp)

      put_log1("Function `model.tune.param_range`:
File saved: %1", file_path_tmp)
```

8. Determines endpoints for a new interval of the *tuning parameter* values in the neighborhood of the current minimum *RMSE* value (figured out so far) with use of the helper function get_fine_tune.param.endpoints.idx (described above in *this Section*), to improve the accuracy of the *RMSE* calculation in the next iteration.

The following code snippet performs this step of execution:

```
    bound.idx <- get_fine_tune.param.endpoints.idx(tuned.result)
    start.idx <- bound.idx["start"]
    end.idx <- bound.idx["end"]

    prm_val.leftmost.tmp <- tuned.result$parameter.value[start.idx]
    RMSE.leftmost.tmp <- tuned.result$RMSE[start.idx]

    prm_val.rightmost.tmp <- tuned.result$parameter.value[end.idx]
    RMSE.rightmost.tmp <- tuned.result$RMSE[end.idx]

    min_RMSE <- tuned.result$RMSE[best_RMSE.idx]
    min_RMSE.prm_val <- tuned.result$parameter.value[best_RMSE.idx]

    seq_start <- prm_val.leftmost.tmp
    seq_end <- prm_val.rightmost.tmp
```

9. Checks whether the local variable `best_RMSE` is already initialized, and if not, initializes it to the minimum *RMSE* value for the current iteration, figured out in the previous step, along with local variables intended to store the endpoint values.

The following code snippet performs this step of execution:

```
    if (is.na(best_RMSE)) {
      prm_val.leftmost <- prm_val.leftmost.tmp
      RMSE.leftmost <- RMSE.leftmost.tmp

      prm_val.rightmost <- prm_val.rightmost.tmp
      RMSE.rightmost <- RMSE.rightmost.tmp

      param.best_value <- min_RMSE.prm_val
      best_RMSE <- min_RMSE
    }
```

10. Checks the deviations from the minimum *RMSE* value (stored in `min_RMSE` local variable) for the values, corresponding to the endpoints of the new interval of the *tuning parameter* values (figured out earlier in step 8), and updates the corresponding local variables related to the endpoints, if the deviation reached the minimum threshold specified by the endpoint.min_diff argument.

The following code snippet performs this step of execution:

```
    if (RMSE.leftmost.tmp - min_RMSE >= endpoint.min_diff) {
      prm_val.leftmost <- prm_val.leftmost.tmp
      RMSE.leftmost <- RMSE.leftmost.tmp
    }

    if (RMSE.rightmost.tmp - min_RMSE >= endpoint.min_diff) {
      prm_val.rightmost <- prm_val.rightmost.tmp
      RMSE.rightmost <- RMSE.rightmost.tmp
    }
```

11. Makes sure that the index of the start endpoint of the parameter interval figured out in the step8 is less than the index of the end endpoint and, if not, the loop is terminated with a corresponding *warning message* and execution proceeds to the Finalizing Execution section of the code described below in section Finalizing Execution subsection of *this Description*.

Otherwise, proceeds to the next step.

The following code snippet performs this step of execution:

```
    if (end.idx - start.idx <= 0) {
      warning("`tuned.result$parameter.value` sequential start index are the same or greater than end or
      put_log1("Function `model.tune.param_range`:
Current minimal RMSE: %1", rmse_min)

      put_log2("Function `model.tune.param_range`:
Reached minimal RMSE for the test parameter value = %1: %2",
               param_values.best_result["param.best_value"],
               param_values.best_result["best_RMSE"])

      put(param_values.best_result)
      break
    }
```

In the code snippet above, we use the custom logging functions put_log1 and put_log2 described above in section Logging Functions of *this Appendix*.

12. Makes sure that the minimum *RMSE* value achieved so far (and stored in the local variable `best_RMSE`) is no less than the minimum *RMSE* value calculated for the current iteration and stored in the local variable `min_RMSE` (that is `best_RMSE >= min_RMSE`) and, if not, ***terminates the execution*** with an error message (this case is considered a **fatal error!**)

Otherwise, the further execution depends on the following condition:

- If `best_RMSE == min_RMSE`, proceeds to the next step with a corresponding *warning message*;
- Otherwise, proceeds to step 14.

The following code snippet performs this step of execution:

```
    if (best_RMSE == min_RMSE) {

            warning("Currently computed minimal RMSE equals the previously reached best one: ",
              best_RMSE, "
Currently computed minial value is: ", min_RMSE)

      put_log2("Function `model.tune.param_range`:
Current minimal RMSE for `parameter value = %1`: %2",
               tuned.result$parameter.value[which.min(tuned.result$RMSE)],
               min_RMSE)

      put_log2("Function `model.tune.param_range`:
So far reached best RMSE for `parameter value = %1`: %2",
               param_values.best_result["param.best_value"],
               param_values.best_result["best_RMSE"])

      put(param_values.best_result)


      # the next step of the current iteration
      # ...

    } else if (best_RMSE < min_RMSE) {
      # Fatal Error! (stops the execution)

      warning("Current minimal RMSE is greater than previously computed best value: ",
              best_RMSE, "
Currently computed minial value is: ", min_RMSE)
      stop("Current minimal RMSE is greater than previously computed best value: ",
           best_RMSE, "
Currently computed minial value is: ", min_RMSE)
    }

    # step 14 of the current iteration
```

13. (`best_RMSE == min_RMSE`) Checks if the number of equal values of the *best RMSE* in the *tuned result* for the current iteration (stored in the local variable `tuned.result$RMSE`) reached the value specified by the value passed in the argument max.identical.min_RMSE.count, and, if so, saves the best values of the *tuning parameter* and corresponding *RMSE* in the local variable `param_values.best_result`, terminates the loop with a corresponding warning message, and proceeds to the Finalizing Execution section of the code described below in section Finalizing Execution subsection of *this Description*.

Otherwise, proceeds to the next step.

The following code snippet performs this step of execution:

```
    if (sum(tuned.result$RMSE[tuned.result$RMSE == min_RMSE]) >= max.identical.min_RMSE.count) {
      warning("Minimal `RMSE`identical values count reached it maximum allowed value: ",
              max.identical.min_RMSE.count)

      put(tuned.result$RMSE)

      param_values.best_result <-
        get_best_param.result(tuned.result$parameter.value,
                              tuned.result$RMSE)

      put_log2("Function `model.tune.param_range`:
    Reached the best RMSE for `parameter value = %1`: %2",
               param_values.best_result["param.best_value"],
               param_values.best_result["best_RMSE"])
      break
    }

    # proceed to the next step of the current iteration
    # ...
```

To retrieve the best results for the current iteration from the data stored in the local variable `tuned.result`, the helper function get_best_param.result described above is used.

14. (`best_RMSE >= min_RMSE`) Updates the following local variables:

- `best_RMSE` with the value stored in the `min_RMSE`(meaning the best *RMSE* has been possibly improved in the current iteration);
- `param.best_value` with the corresponding *best parameter value* (stored in the `min_RMSE.prm_val`)
- `param_values.best_result` with the best results retrieved from the `tuned.result`.

The following code snippet performs this step of execution:

```
    best_RMSE <- min_RMSE
    param.best_value <- min_RMSE.prm_val

    param_values.best_result <-
      get_best_param.result(tuned.result$parameter.value,
                            tuned.result$RMSE)

    put_log2("Function `model.tune.param_range`:
Currently reached best RMSE for `parameter value = %1`: %2",
             param_values.best_result["param.best_value"],
             param_values.best_result["best_RMSE"])

    put(param_values.best_result)
  }
  # End repeat loop
```

In the code snippet above, we use the user-defined auxiliary function get_best_param.result described above in *this Section*.

15. Proceeds to the next iteration, starting from step 1

***Finalizing Execution***

Finally, the algorithm initializes the objects to be included in the return list with the results obtained during execution and returns the *list object* to the client code.

The following code snippet performs this step of execution:

```
  # Finalizing execution:
  n <- length(tuned.result$parameter.value)

  parameter.value <- tuned.result$parameter.value
  result.RMSE <- tuned.result$RMSE

  if (result.RMSE[1] == best_RMSE) {
    parameter.value[1] <- prm_val.leftmost
    result.RMSE[1] <- RMSE.leftmost
  }

  if (result.RMSE[n] == best_RMSE) {
    parameter.value[n+1] <- prm_val.rightmost
    result.RMSE[n+1] <- RMSE.rightmost
  }

  list(best_result = param_values.best_result,
       param_values.endpoints = c(prm_val.leftmost, prm_val.rightmost, seq_increment),
       tuned.result = data.frame(parameter.value = parameter.value,
                                 RMSE = result.RMSE))
```

#### 5.1.5.6.4 Value

From the source code of the function *being described here*, we can see that it returns a list of three objects:

```
list(best_result = param_values.best_result,
     param_values.endpoints = c(prm_val.leftmost, prm_val.rightmost, seq_increment),
     tuned.result = data.frame(parameter.value = parameter.value,
                               RMSE = result.RMSE))
```

The objects comprising the return list are as follows:

- `best_result`: A vector containing a *pair of values*: the found *best value of the tuning parameter* for which the *RMSE* value is minimal, along with the minimal *RMSE* value itself;
- `param_values.endpoints`: Information related to the final interval of the *tuning parameter* values;
- `tuned.result`: A data frame containing two variables:
  - *parameter.value*: A vector containing all processed values of the *tuning parameter*;
  - *RMSE*: A vector containing the *RMSE* values corresponding to the processed *tuning parameter* values.

#### 5.1.5.6.5 Source Code

The complete source code of the function model.tune.param_range is available in the Regularization section of the common-helper.functions.R script on *GitHub*.

The essential parts of the source code are also provided and explained in the Details section of *this Description*.

## 5.2 Data Helper Functions

### 5.2.1 Initializing Input Datasets

The complete source code of the functions described in this section is available in the the Initializing input datasets section of the data.helper.functions.R script on *GitHub*.

#### 5.2.1.1 make_source_datasets Function

Creates the major input datasets (*Movielens Datasets*) used throughout the project.

##### 5.2.1.1.1 Usage

```
movielens_datasets <- make_source_datasets()
```

The above line of code creates the *Movielens Datasets* using the function `make_source_datasets`, which is called internally from the custom helper function init_source_datasets described below in *this Section*.

##### 5.2.1.1.2 Details

To produce the *Movielens Datasets* (listed in the Value section of *this Appendix* below), the function uses the `edx` as the original dataset, and, during the execution, it calls internally the following *user-defined data helper functions* described below in section Data Processing Functions of *this Appendix*:

- splitGenreRows to create the `edx.sgr` Object;
- sample_train_validation_sets to create *Train* and *Validation* sets of the *inner datasets* of the `edx_CV` Object.

Additionally, it also calls the following *custom logging functions* described above in section Logging Functions of *this Appendix*:

- put_start_date;
- put_end_date;
- put_log;
- put_log1.

The function being described is a utility and is used internally in the init_source_datasets function as shown above in the Usage section of *this Description*.

#### 5.2.1.1.3 Value

The list of the following datasets (described in detail in Appendix B: Models Training Datasets) are returned to the client code:

- edx.mx Matrix Object;
- edx.sgr Object;
- movie_map Object;
- date_days_map Object;
- edx_CV Object.

#### 5.2.1.1.4 Source Code

The source code of the make_source_datasets function is shown below:

```
make_source_datasets <- function(){
  put_log("Function: `make_source_datasets`: Creating source datasets...")

  put_log("Function: `make_source_datasets`: Creating Rating Matrix from `edx` dataset...")
  edx.mx <- edx |>
    mutate(userId = factor(userId),
           movieId = factor(movieId)) |>
    select(movieId, userId, rating) |>
    pivot_wider(names_from = movieId, values_from = rating) |>
    column_to_rownames("userId") |>
    as.matrix()

  put_log("Function: `make_source_datasets`:
Matrix created: `edx.mx` of the following dimentions:")
  put(dim(edx.mx))

  #> To be able to map movie IDs to titles we create the following lookup table:
  movie_map <- edx |> select(movieId, title, genres) |>
    distinct(movieId, .keep_all = TRUE)

  put_log("Function: `make_source_datasets`: Dataset created: movie_map")
  put(summary(movie_map))

  put_log("Function: `make_source_datasets`: Creating Date-Days Map dataset...")
  date_days_map <- edx |>
    mutate(date_time = as_datetime(timestamp)) |>
    mutate(date = as_date(date_time)) |>
    mutate(year = year(date_time)) |>
    mutate(days = as.integer(date - min(date))) |>
    select(timestamp, date_time, date, year, days) |>
    distinct(timestamp, .keep_all = TRUE)

  put_log("Function: `make_source_datasets`: Dataset created: date_days_map")
  put(summary(date_days_map))
```

```
  put_log("Function: `make_source_datasets`:
To account for the Movie Genre Effect, we need a dataset with split rows
for movies belonging to multiple genres.")
  edx.sgr <- splitGenreRows(edx)



  #> We will use K-fold cross validation as explained in
  #> Section 29.6.1: "K-fold validation" of the Cource Textbook:
  #> https://rafalab.dfci.harvard.edu/dsbook-part-2/ml/resampling-methods.html#k-fold-cross-validation
  #> We are going to compute the following version of the MSE introducing in that section:

  # $$
  #   \mbox{MSE}(\lambda) \approx\frac{1}{B} \sum_{b = 1}^B \frac{1}{N}\sum_{i = 1}^N \left(\hat{y}_i^b
  # $$

  start <- put_start_date()
  edx_CV <- lapply(kfold_index,  function(fold_i){

    put_log1("Method `make_source_datasets`:
Creating K-Fold Cross-Validation Datasets, Fold %1", fold_i)

    #> We split the initial datasets into training sets, which we will use to build
    #> and train our models, and validation sets in which we will compute the accuracy
    #> of our predictions, the way described in the `Section 23.1.1 Movielens data`
    #> (https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movielens-data)
    #> of the Course Textbook.

    split_sets <- edx |>
      sample_train_validation_sets(fold_i*1000)

    train_set <- split_sets$train_set
    validation_set <- split_sets$validation_set

    put_log("Function: `make_source_datasets`:
Sampling 20% from the split-row version of the `edx` dataset...")
    split_sets.gs <- edx.sgr |>
      sample_train_validation_sets(fold_i*2000)

    train.sgr <- split_sets.gs$train_set
    validation.sgr <- split_sets.gs$validation_set

    # put_log("Function: `make_source_datasets`: Dataset created: validation.sgr")
    # put(summary(validation.sgr))

    #> We will use the array representation described in `Section 17.5 of the Textbook`
    #> (https://rafalab.dfci.harvard.edu/dsbook-part-2/linear-models/treatment-effect-models.html#sec-a
    #> for the training data.
    #> To create this matrix, we use `tidyr::pivot_wider` function:

    # train_set <- mutate(train_set, userId = factor(userId), movieId = factor(movieId))
    # train.sgr <- mutate(train.sgr, userId = factor(userId), movieId = factor(movieId))
```

```
    put_log("Function: `make_source_datasets`: Creating Rating Matrix from Train Set...")
    train_mx <- train_set |>
      mutate(userId = factor(userId),
             movieId = factor(movieId)) |>
      select(movieId, userId, rating) |>
      pivot_wider(names_from = movieId, values_from = rating) |>
      column_to_rownames("userId") |>
      as.matrix()

    put_log("Function: `make_source_datasets`:
Matrix created: `train_mx` of the following dimentions:")
    put(dim(train_mx))

    list(train_set = train_set,
         train_mx = train_mx,
         train.sgr = train.sgr,
         validation_set = validation_set)
  })
  put_end_date(start)
  put_log("Function: `make_source_datasets`:
Set of K-Fold Cross-Validation datasets created: edx_CV")

  tuning_sets <- edx |>
    sample_train_validation_sets(5423)

  list(edx_CV = edx_CV,
       edx.mx = edx.mx,
       edx.sgr = edx.sgr,
       tuning_sets = tuning_sets,
       movie_map = movie_map,
       date_days_map = date_days_map)
}
```

The source code of the make_source_datasets function is also available in the Initializing input datasets section of the data.helper.functions.R script on *GitHub*.

#### 5.2.1.2 init_source_datasets Function

Creates or loads from the *Local Cache* file the major input datasets used throughout the project.

##### 5.2.1.2.1 Usage

```
movielens_datasets <- init_source_datasets()
```

##### 5.2.1.2.2 Details

The function checks whether the *Movielens Datasets* are cached on the hard disk, calling the base::file.exists `R` function, and, if so, loads them from the *Local Cache* file calling the load_movielens_data_from_file user-defined auxiliary function described below in section Data Processing Functions of *this Appendix*.

Otherwise, it calls the make_source_datasets function described above in *this Section*, which creates all the necessary datasets using the `edx` as the original dataset, saves the result datasets in a dedicated cache file using the base::save 'R' function, and returns the datasets to the client code.

##### 5.2.1.2.3 Value

The list of the following datasets (described in detail in Appendix B: Models Training Datasets) is returned to the client code:

- `edx.mx` Matrix Object;
- `edx.sgr` Object;
- `movie_map` Object;
- `date_days_map` Object;
- `edx_CV` Object.

#### 5.2.1.2.4 Source Code

The simplified version of the source code of the init_source_datasets function is shown below:

```
init_source_datasets <- function(){
  if(file.exists(movielens_datasets_file_path)){
    movielens_datasets <- load_movielens_data_from_file(movielens_datasets_file_path)
  } else if(file.exists(movielens_datasets_zip)) {
    unzip(movielens_datasets_zip, movielens_datasets_file_path)

    if(!file.exists(movielens_datasets_file_path)) {
      put_log("Method `init_source_datasets`:
File does not exists: {movielens_datasets_file}.")
      stop("Failed to unzip MovieLens data zip-archive.")
    }
    movielens_datasets <- load_movielens_data_from_file(movielens_datasets_file_path)
  } else {
    library(edx.capstone.movielens.data)
    movielens_datasets <- make_source_datasets()
    put("Method `init_source_datasets`:
All required datasets have been created.")

    save(movielens_datasets, file =  movielens_datasets_file_path)

    if(!file.exists(movielens_datasets_file_path)) {
      warning("MovieLens data was not saved to file.")
    } else {
      put_log("Method `init_source_datasets`:
Datasets have been saved to file: {movielens_datasets_file_path}.")

      zip(movielens_datasets_zip, movielens_datasets_file_path)

      if(!file.exists(movielens_datasets_zip)){
        warning("Failed to zip MovieLens data file.")
      } else {
        put_log("Method `init_source_datasets`:
Zip-archive created: {movielens_datasets_zip}.")
      }
    }
  }
  movielens_datasets
}
```

The complete source code of the init_source_datasets function is available in the Initializing input datasets section of the data.helper.functions.R script on *GitHub*.

### 5.2.2 Data Processing Functions

The complete source code of the functions described in this section is available in the the Data processing functions section of the data.helper.functions.R script on *GitHub*.

#### 5.2.2.1 load_movielens_data_from_file Function

Reloads *Movielens* datasets from the cache file (previously created by the user-defined helper function init_source_datasets described above in section Initializing Input Datasets to *this Appendix*) using the path passed in the file_path argument.

##### 5.2.2.1.1 Usage

```
movielens_datasets <- load_movielens_data_from_file(movielens_datasets_file_path)
```

The above line of code loads the *Movielens Datasets* using the function `load_movielens_data_from_file`, which is called internally from the function init_source_datasets mentioned above in the *current Description*.

##### 5.2.2.1.2 Arguments

- **file_path:** Full path to the *cache file*.

##### 5.2.2.1.3 Details

This function is a wrapper for the `R` function base::load. It also internally calls the auxiliary `R` function base::file.exists to check for the existence of the *cache file* at the path specified by the file_path argument, as well as the following custom logging functions described in section Logging Functions of *this Appendix*:

- put_start_date;
- put_end_date.
- put_log1;

This is an auxiliary function that is used internally in the init_source_datasets function as shown above in the Usage section of *this Description*.

#### 5.2.2.1.4 Value

The list of the following datasets (described in detail in Appendix B: Models Training Datasets) is returned to the client code:

- `edx.mx` Matrix Object;
- `edx.sgr` Object;
- `movie_map` Object;
- `date_days_map` Object;
- `edx_CV` Object.

#### 5.2.2.1.5 Source Code

The source code of the load_movielens_data_from_file function is shown below:

```
load_movielens_data_from_file <- function(file_path){
  put_log1("Loading MovieLens datasets from file: %1...",
           file_path)
  start <- put_start_date()
  load(file_path)
  put_end_date(start)
  put_log1("MoviLens datasets have been loaded from file: %s.",
           file_path)
  movielens_datasets
}
```

The source code of the load_movielens_data_from_file function is also available in the Data processing functions section of the data.helper.functions.R script on *GitHub*.

#### 5.2.2.2 splitByUser Function

Divides the set of *row indices* of the dataset passed in the data argument into the groups defined by the `.$userId` variable in that dataset.

##### 5.2.2.2.1 Usage

```
    sapply(splitByUser(data),
           function(i) sample(i, ceiling(length(i)*.2))) |>
    unlist() |>
    sort()
```

In the code snippet above, the function `splitByUser` is called internally from the sample_train_validation_sets function described below in *this Section*.

##### 5.2.2.2.2 Arguments

- **data:** Dataset to *split by user*;

##### 5.2.2.2.3 Details

This function is a wrapper for the base::split `R` function, which is called internally to divide the *set of row indices* of the *dataset* passed in the data argument into the groups defined by the variable `.$userId`.

This is an auxiliary function intended for use in the sample_train_validation_sets function as shown in the Usage section above of the *current Description*.

##### 5.2.2.2.4 Value

List of vectors containing the values (corresponding *row indices* of the dataset passed in the data) for the `.$userId` groups. The components of the list are named by the values of the `.$userId` (after converting to a factor). For more details, please refer to the documentation of the base::split function, which, in fact, forms the return value.

##### 5.2.2.2.5 Source Code

The source code of the splitByUser function is shown below:

```
splitByUser <- function(data){
  split(1:nrow(data), data$userId)
}
```

The source code of the splitByUser function is also available in the Data processing functions section of the data.helper.functions.R script on *GitHub*.

#### 5.2.2.3 sample_train_validation_sets Function

Implements an algorithm to split the input dataset passed in the data argument into *Train* and *Validation* sets in the way described in Section 29.6 *Cross validation* of the *Course Textbook (New Edition)*[7].

##### 5.2.2.3.1 Usage

```
split_sets <- edx |>
  sample_train_validation_sets(fold_i*1000)
```

In the code snippet above, the function `sample_train_validation_sets` is called internally from the make_source_datasets function described above in section Initializing Input Datasets of *this Appendix*.

##### 5.2.2.3.2 Arguments

- **data:** Input dataset to split;
- **seed:** Seed value;

##### 5.2.2.3.3 Details

This is an auxiliary function intended for use in the make_source_datasets function as shown in the Usage section above of the *current Description*.

To split the dataset passed in the data argument, the algorithm first takes a sample of 20% of the size of the set of its *row indices* for each user and sorts them by their corresponding `.$userId` value.

Next, the algorithm creates a *Train Set* from the rows with indices that *IS NOT* in the *set of samples* taken in the previous step.

Finally, it creates a *Validation Set* from the remaining data, including *ONLY THOSE ROWS* that have `.$userId` and `.$movieId` contained in the *Train Set*.

During its execution, the algorithm internally calls the user-defined helper function splitByUser described above in *this Section*, as well as the custom logging function put_log described above in section Logging Functions of *this Appendix*.

#### 5.2.2.3.4 Value

List of size 2, consisting of the *Training* and *Testing* datasets created during the algorithm execution.

#### 5.2.2.3.5 Source Code

Below is a simplified version of the source code of the sample_train_validation_sets function:

```
sample_train_validation_sets <- function(data, seed){
  put_log("Function: `sample_train_validation_sets`: Sampling 20% of the `data` data...")
  set.seed(seed)
  validation_ind <-
    sapply(splitByUser(data),
           function(i) sample(i, ceiling(length(i)*.2))) |>
    unlist() |>
    sort()

  put_log("Function: `sample_train_validation_sets`:
Extracting 80% of the original `data` not used for the Validation Set,
excluding data for users who provided no more than a specified number of ratings: {min_nratings}.")
  train_set <- data[-validation_ind,]

  put_log("Function: `sample_train_validation_sets`:
To make sure we don't include movies in the Training Set that should not be there,
we exclude entries using the semi_join function from the Validation Set.")
  tmp.data <- data[validation_ind,]

  validation_set <- tmp.data |>
    semi_join(train_set, by = "movieId") |>
    semi_join(train_set, by = "userId") |>
    as.data.frame()

  # Add rows excluded from `validation_set` into `train_set`
  tmp.excluded <- anti_join(tmp.data, validation_set)
  train_set <- rbind(train_set, tmp.excluded)

  # Return result datassets
  list(train_set = train_set,
       validation_set = validation_set)
}
```

The complete source code of the sample_train_validation_sets function is available in the Data processing functions section of the data.helper.functions.R script on *GitHub*.

#### 5.2.2.4 splitGenreRows Function

Splits rows of the original dataset passed in the data argument for movies that belong to multiple genres.

##### 5.2.2.4.1 Usage

```
edx.sgr <- splitGenreRows(edx)
```

The above line of code splits rows of the `edx` dataset using the function `splitGenreRows`, which is called internally from the function make_source_datasets described above in section Initializing Input Datasets of *this Appendix*.

##### 5.2.2.4.2 Arguments

- **data:** A dataset containing rows corresponding to movies that belong to multiple genres..

##### 5.2.2.4.3 Details

The function is a wrapper for the tidyr::separate_rows `R` function, which is called internally to operate.

This is an auxiliary function intended for use in the make_source_datasets function as shown in the Usage section above of *this Description*.

##### 5.2.2.4.4 Value

Dataset object with split rows (those belong to multiple genres) described in detail below in the `edx.sgr` Object of the Appendix B: Models Training Datasets.

##### 5.2.2.4.5 Source Code

The source code of the splitGenreRows function is shown below:

```
splitGenreRows <- function(data){
  put("Splitting dataset rows related to multiple genres...")
  start <- put_start_date()
  gs_splitted <- data |>
    separate_rows(genres, sep = "\\|")
  put("Dataset rows related to multiple genres have been splitted to have single genre per row.")
  put_end_date(start)
  gs_splitted
}
```

The source code of the splitGenreRows function is also available in the Data processing functions section of the data.helper.functions.R script on *GitHub*.

### 5.2.3 Models Training Functions

#### 5.2.3.1 union_cv_results Function

Aggregates the input data frames passed in the data_list argument into a single data frame.

##### 5.2.3.1.1 Usage

```
user_movie_effects_united <- union_cv_results(user_movie_effects_ls)
```

In the above code snippet, the `union_cv_results` function is called internally from the train_user_movie_effect.cv function described in section UME Model: Utility Functions of *this Appendix*.

##### 5.2.3.1.2 Arguments

- **data_list:** List of data frames representing the results of the *K-Fold Cross-Validation* execution (as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8]).

##### 5.2.3.1.3 Details

This function internally uses the base::union `R` function to combine the data from the list of data frames passed in the data_list argument into a single data frame for further processing.

The function is used to aggregate the results of the *K-Fold Cross-Validation*, where the $K$ value is determined by the length of the `edx_CV` Object (described in Appendix B: Models Training Datasets). In *this Project*, we use $K = 5$).

This is a utility function intended to be called internally from the other user-defined function, such as, for instance, the train_user_movie_effect.cv, as shown above in the Usage section of *this Description*.

##### 5.2.3.1.4 Value

A data frame that is a union of the input list of data frames.

#### 5.2.3.1.5 Source Code

The source code of the union_cv_results function is shown below:

```
union_cv_results <- function(data_list) {
  out_dat <- data_list[[1]]

  for (i in 2:CVFolds_N){
    out_dat <- union(out_dat,
                     data_list[[i]])
  }

  out_dat
}
```

The source code of the union_cv_results is also available in the Model training section of the data.helper.functions.R script on *GitHub*.

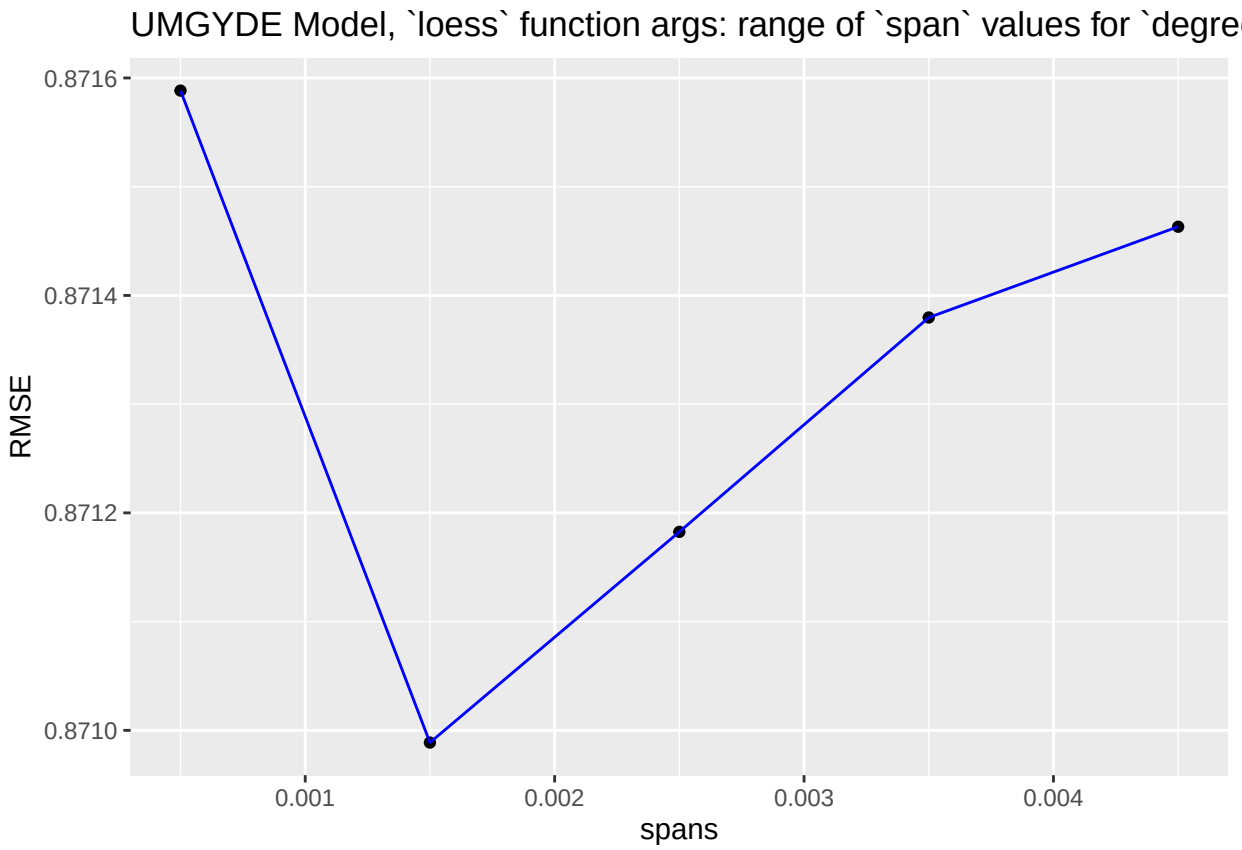
### 5.2.4 Data Visualization Functions

#### 5.2.4.1 data.plot Function

Plots a graph based on the dataset passed in the data parameter, using both points and a line.

##### 5.2.4.1.1 Usage

```
lss.UMGYDE.preset.degree2.result$tuned.result |>
  data.plot(title = "UMGYDE Model, `loess` function args: range of `span` values for `degree = 2`",
            xname = "parameter.value",
            yname = "RMSE",
            xlabel = "spans",
            ylabel = "RMSE")
```



#### 5.2.4.1.2 Arguments

- **data:** Dataset to use for the plot;
- **title:** Title of the plot;
- **xname:** The name of the dataset column used as the source of the $x$ variable for the plot;
- **yname:** The name of the dataset column used as the source of the $y$ variable for the plot;
- **xlabel:** (Optional, *Character string*, `NULL` by default) The $x$ axis label. If `NULL`, the value of the `xname` parameter is used for the label;
- **ylabel:** (Optional, *Character string*, `NULL` by default) The $y$ axis label. If `NULL`, the value of the `yname` parameter is used for the label.;
- **line_col:** (Optional, *Character string*, ***blue*** by default) Line color.
- **normalize:** (Optional, *Boolean*, `FALSE` by default) If `TRUE`, deviations from the minimum $y$ value are used to plot, rather than the $y$ values themselves. Otherwise, the $y$ values are used.

#### 5.2.4.1.3 Details

The function is a wrapper for the ggplot2::ggplot function and is a simple and convenient tool for visualizing the data of *this Project*.

#### 5.2.4.1.4 Source Code

The soure code of the data.plot function is shown below:

```
data.plot <- function(data,
                      title,
                      xname,
                      yname,
                      xlabel = NULL,
                      ylabel = NULL,
                      line_col = "blue",
                      normalize = FALSE) {
  y <- data[, yname]

  if (normalize) {
    y <- y - min(y)
  }

  if (is.null(xlabel)) {
    xlabel = xname
  }
  if (is.null(ylabel)) {
    ylabel = yname
  }

  aes_mapping <- aes(x = data[, xname], y = y)

  data |>
    ggplot(mapping = aes_mapping) +
    ggtitle(title) +
    xlab(xlabel) +
    ylab(ylabel) +
    geom_point() +
    geom_line(color=line_col)
}
```

The source code of the data.plot is also available in the Data Visualization section of the data.helper.functions.R script on *GitHub*.

#### 5.2.4.2 data.plot.left.n Function

Plots a graph based on the first $n$ rows (the $n$ is specified by the left.n parameter) of the dataset passed in the data parameter, using both points and a line.

The dataset passed in the data parameter is assumed to be sorted by the column used for the $x$ axis (specified by the column name passed in the xname parameter).

##### 5.2.4.2.1 Usage

```
data.plot.left_detailed <- function(data,
                                    left.n = 0,
                                    title = NULL,
                                    title.left = NULL,
                                    xname,
                                    yname,
                                    xlabel1 = NULL,
                                    xlabel2 = NULL,
                                    ylabel1 = NULL,
                                    ylabel2 = NULL,
                                    line_col1 = "blue",
                                    line_col2 = "red",
                                    normalize = FALSE) {
# ...

  p2 <- data |>
    data.plot.left.n(left.n = left.n,
                     title = title.left,
                     xname = xname,
                     yname = yname,
                     xlabel = xlabel2,
                     ylabel = ylabel2,
                     line_col = line_col2,
                     normalize = normalize)
  grid.arrange(p1, p2)
}
```

In the code snippet above, the function `data.plot.left.n` is called internally from the data.plot.left_detailed function described below in this section.

#### 5.2.4.2.2 Arguments

- **data:** Dataset to use for the plot.
- **left.n:** (Optional, *Integer*, 0 by default)If a positive numeric value, specifies the number of the first rows of the dataset to plot. Otherwise, the entire dataset data is set to plot.
- **title:** Title of the plot.
- **xname:** The name of the dataset column used as the source of the $x$ variable for the plot.
- **yname:** The name of the dataset column used as the source of the $y$ variable for the plot.
- *xlabel*:** The $x$ axis label. If `NULL`, the value of the `xname` parameter is used for the label.
- **ylabel:** The $y$ axis label. If `NULL`, the value of the `yname` parameter is used for the label.
- **line_col:** (Optional, *Character string*, ***red*** by default) Line color.
- **normalize:** (Optional, *Boolean*, `FALSE` by default) If `TRUE`, deviations from the minimum $y$ value are used to plot, rather than the $y$ values themselves. Otherwise, the $y$ values are used.

#### 5.2.4.2.3 Details

The function is a wrapper for the data.plot function described above and is a simple and convenient tool for more detailed visualization of the left part of the graph, initially built based on the entire data set passed in the data parameter.

This is an auxiliary function that is used internally in the data.plot.left_detailed function (described below in this section) as shown above in the Usage section of *this Description*.

#### 5.2.4.2.4 Source Code

The source code of the data.plot.left.n function is shown below:

```
data.plot.left.n <- function(data,
                             left.n = 0,
                             title,
                             xname,
                             yname,
                             xlabel,
                             ylabel,
                             line_col = "red",
                             normalize = FALSE) {
  x_col <- data[, xname]
  y_col <- data[, yname]


  data.left <- data

  if (left.n > 0) {
    data.left <- data |>
      head(left.n)
  }

  data.left |>
    data.plot(title = title,
              xname = xname,
              yname = yname,
              xlabel = xlabel,
              ylabel = ylabel,
              line_col = line_col,
              normalize = normalize)
}
```

The source code of the data.plot.left.n is also available in the Data Visualization section of the data.helper.functions.R script on *GitHub*.

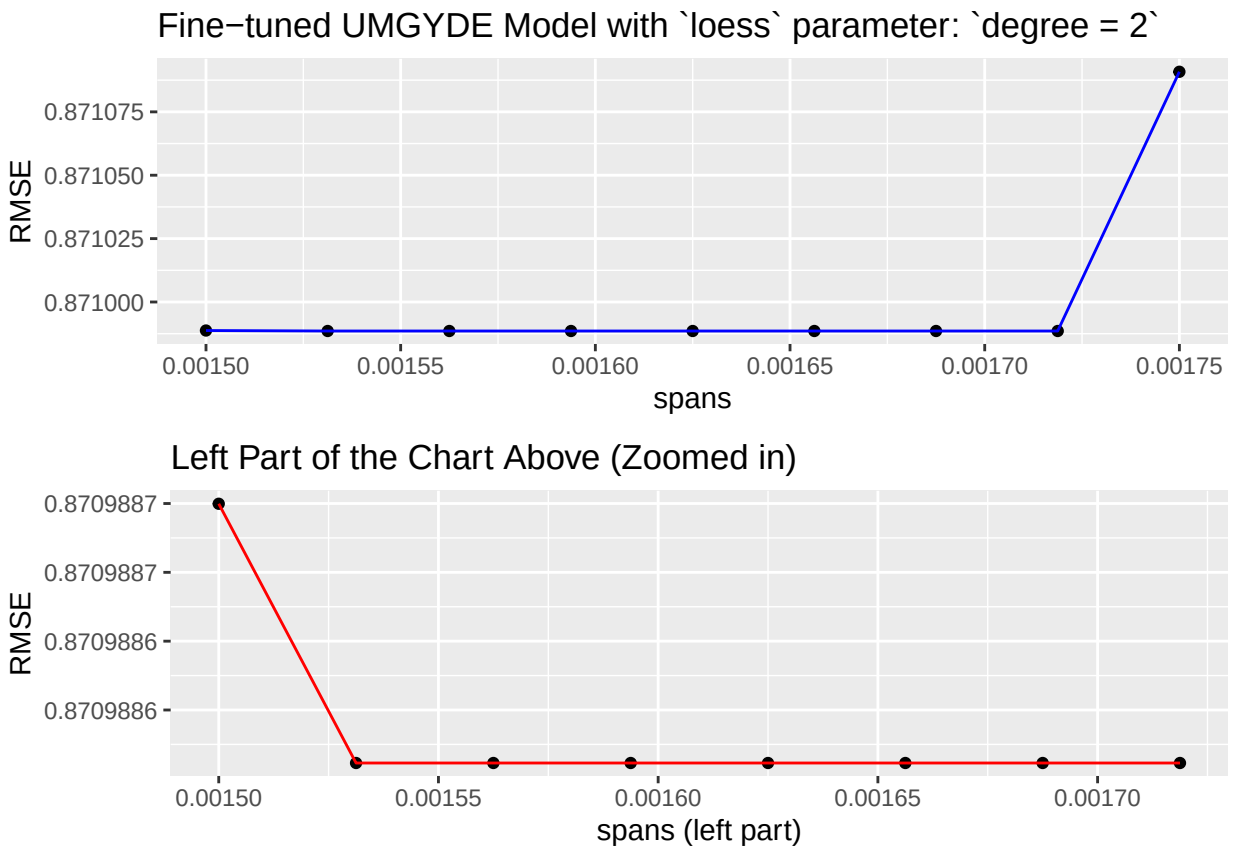
#### 5.2.4.3 data.plot.left_detailed Function

Plots two graphs, placing them vertically one above the other: the first one at the top is the *main graph* plotted by the internally called auxiliary function data.plot (described above in this section) based on the data passed in the data argument, and the second one is the left part of the *main graph*, based on the first $n$ rows of the data used for the *main graph*, and plotted on an *enlarged scale* (the $n$ value is passed in the left.n argument).

The dataset passed in the data parameter is assumed to be sorted by the column used for the $x$ axis (specified by the column name passed in the xname parameter).

##### 5.2.4.3.1 Usage

```
lss.UMGYDE.fine_tune.degree2.result$tuned.result |>
  data.plot.left_detailed(title = "Fine-tuned UMGYDE Model with `loess` parameter: `degree = 2`",
                          title.left = "Left Part of the Chart Above (Zoomed in)",
                          left.n = 8,
                          xname = "parameter.value",
                          yname = "RMSE",
                          xlabel1 = "spans",
                          ylabel1 = "RMSE")
```



#### 5.2.4.3.2 Arguments

- **data:** Dataset to use for the plot.
- **left.n:** If a positive numeric value, specifies the number of the first rows of the dataset to plot. Otherwise, the entire dataset data is set to plot.
- **title:** Title of the plot.
- **xname:** The name of the dataset column used as the source of the $x$ variable for the plot.
- **yname:** The name of the dataset column used as the source of the $y$ variable for the plot.
- **xlabel1:** (Optional, *Character string*, `NULL` by default) The $x$ axis label for the first plot. If `NULL`, the value of the `xname` parameter is used for the label.
- **xlabel2:** (Optional, *Character string*, `NULL` by default) The $x$ axis label for the second plot. If `NULL`, the value of the `xname` parameter is used for the label.
- **ylabel1:** (Optional, *Character string*, `NULL` by default) The $y$ axis label for the first plot. If `NULL`, the value of the `yname` parameter is used for the label.
- **ylabel2:** (Optional, *Character string*, `NULL` by default) The $y$ axis label for the second plot. If `NULL`, the value of the `yname` parameter is used for the label.
- **line_col1** (Optional, *Character string*, ***blue*** by default) The line color (main graqh).
- **line_col2** (Optional, *Character string*, ***red*** by default) The line color (second graph: the left part of the main graph, which is shown on an enlarged scale).
- **normalize:** (Optional, *Boolean*, `FALSE` by default) If `TRUE`, deviations from the minimum $y$ value are used to plot, rather than the $y$ values themselves. Otherwise, the $y$ values are used.

#### 5.2.4.3.3 Details

Internally, this function calls the data.plot and data.plot.left.n functions (the both described above in this section) to build two graphs, which it then combines into a grid using gridExtra::grid.arrange `R` function.

#### 5.2.4.3.4 Source Code

The soure code of the data.plot.left_detailed function is shown below:

```
data.plot.left_detailed <- function(data,
                                    left.n = 0,
                                    title = NULL,
                                    title.left = NULL,
                                    xname,
                                    yname,
                                    xlabel1 = NULL,
                                    xlabel2 = NULL,
                                    ylabel1 = NULL,
                                    ylabel2 = NULL,
                                    line_col1 = "blue",
                                    line_col2 = "red",
                                    normalize = FALSE) {
  if(is.null(xlabel1)) {
    xlabel1 <- xname
  }
  if(is.null(xlabel2)) {
    xlabel2 <- str_glue(xlabel1, " (left part)")
  }
  if(is.null(ylabel1)) {
    ylabel1 <- yname
  }
  if(is.null(ylabel2)) {
    ylabel2 <- ylabel1
  }

  p1 <- data |>
    data.plot(title = title,
              xname = xname,
              yname = yname,
              xlabel = xlabel1,
              ylabel = ylabel1,
              line_col = line_col1)

  p2 <- data |>
    data.plot.left.n(left.n = left.n,
                     title = title.left,
                     xname = xname,
                     yname = yname,
                     xlabel = xlabel2,
                     ylabel = ylabel2,
                     line_col = line_col2,
                     normalize = normalize)
  grid.arrange(p1, p2)
}
```

The source code of the data.plot.left_detailed is also available in the Data Visualization section of the data.helper.functions.R script on *GitHub*.

## 5.3 Models Training: Support Functions

### 5.3.1 OMR Model: Helper Functions

The complete source code of the functions described in this section is available in the OMR-model.functions.R script on *GitHub*.

#### 5.3.1.1 naive_model_MSEs Function

Validates the *Overall Mean Rating (OMR) Model* with use of *K-Fold Cross-Validation* method as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

##### 5.3.1.1.1 Usage

```
mu <- mean(edx$rating)
mu.MSEs <- naive_model_MSEs(mu)
```

##### 5.3.1.1.2 Arguments

- **mu:** The *Overall Mean Rating (OMR)* value.

##### 5.3.1.1.3 Details

The function iterates through the `edx_CV` Object items internally calling the user-defined helper function mse (described above in section (Root) Mean Squared Error Calculation of *this Appendix*) for each iteration.

##### 5.3.1.1.4 Value

Numeric vector of size $K$ containing the Mean Squared Error (MSE) values obtained from applying the *K-Fold Cross-Validation* to the *Overall Average Rating (OMR) Model*, where the $K$ value is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

#### 5.3.1.1.5 Source Code

The source code of the naive_model_MSEs function is shown below:

```
naive_model_MSEs <- function(mu) {
  sapply(edx_CV, function(cv_item){
    mse(cv_item$validation_set$rating - mu)
  })
}
```

The source code of the naive_model_MSEs function is defined in the OMR-model.functions.R script on *GitHub*.

#### 5.3.1.2 naive_model_RMSE Function

Validates the *Overall Mean Rating (OMR) Model* with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

##### 5.3.1.2.1 Usage

```
deviation <- seq(0, 6, 0.1) - 3

deviation.RMSE <- sapply(deviation, function(delta){
  naive_model_RMSE(mu + delta)
})
```

##### 5.3.1.2.2 Arguments

- **mu:** The overall mean rating value.

##### 5.3.1.2.3 Details

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

This function internally calls the naive_model_MSEs function described above and then calculates the square root of the mean of the obtained results.

##### 5.3.1.2.4 Value

The Root Mean Squared Errors (RMSE) value obtained from applying the *K-Fold Cross-Validation* method to the *Overall Average Rating (OMR) Model*, where the $K$ value is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.1.2.5 Source Code

The source code of the naive_model_RMSE function is shown below:

```
naive_model_RMSE <- function(mu){
  sqrt(mean(naive_model_MSEs(mu)))
}
```

The source code of the naive_model_RMSE function is defined in the OMR-model.functions.R script on *GitHub*.

### 5.3.2 UME Model: Utility Functions

The complete source code of the functions described in this section is available in the Utility Functions section of the UM-effect.functions.R script on *GitHub*.

#### 5.3.2.1 train_user_movie_effect Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie Effect (UME) Model* using the training dataset passed in the train_set argument.

##### 5.3.2.1.1 Usage

```
rglr.UM_effect <- train_user_movie_effect(edx, UME.rglr.best_lambda)
```

```
str(rglr.UM_effect)
```

```
## tibble [10,677 x 3] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ b      : Named num [1:10677] 0.331 -0.305 -0.364 -0.599 -0.443 ...
##   ..- attr(*, "names")= chr [1:10677] "param.best_value" "param.best_value" "param.best_value" "para
##  $ n      : int [1:10677] 23790 10779 7028 1577 6400 12346 7259 821 2278 15187 ...
```

##### 5.3.2.1.2 Arguments

- **train_set:** Training dataset;
- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.2.1.3 Details

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

*Regularization* is committed through the *penalized regression* by computing the *penalized mean* performed by the internally called mean_reg function described above in section Model Tuning Utils of *this Appendix*.

##### 5.3.2.1.4 Value

Data frame object representing the trained (and also *regularized* if lambda arg value other than default) *UM Effect Model*.

#### 5.3.2.1.5 Source Code

The source code of the train_user_movie_effect function is shown below:

```
train_user_movie_effect <- function(train_set, lambda = 0){
  if (is.na(lambda)) {
    stop("Function: train_user_movie_effect
`lambda` is `NA`")
  }

  UM.effect <- train_set |>
    left_join(edx.user_effect, by = "userId") |>
    mutate(resid = rating - (mu + a)) |>
    group_by(movieId) |>
    summarise(b = mean_reg(resid, lambda), n = n())

  stopifnot(!is.na(mean(UM.effect$b)))
  UM.effect
}
```

The source code of the train_user_movie_effect function is also available in the Utility Functions section of the UM-effect.functions.R script on *GitHub*.

#### 5.3.2.2 train_user_movie_effect.cv Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie Effect (UME) Model* with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

##### 5.3.2.2.1 Usage

```
  cv.UM_effect <- train_user_movie_effect.cv()
```

```
str(cv.UM_effect)
```

```
## tibble [10,677 x 3] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ b      : num [1:10677] 0.335 -0.306 -0.365 -0.598 -0.444 ...
##  $ n      : num [1:10677] 18907 8593 5574 1253 5065 ...
```

##### 5.3.2.2.2 Arguments

- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.2.2.3 Details

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

Internally, this function calls the helper function train_user_movie_effect described above for each iteration of the *K-Fold Cross-Validation* applied to the `edx_CV` Object described below in Appendix B: Models Training Datasets. It also calls internally the union_cv_results function described above in section Models Training Functions of this *Appendix* to aggregate the cross-validation results.

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.2.2.4 Value

Data frame object representing the *UM Effect Model* trained (and also *regularized* if the lambda argument value is other than the default) with use of *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

#### 5.3.2.2.5 Source Code

The simplified version of the source code of the train_user_movie_effect.cv function is shown below:

```
train_user_movie_effect.cv <- function(lambda = 0){
# ...
  start <- put_start_date()
  user_movie_effects_ls <- lapply(edx_CV, function(cv_fold_dat){
    cv_fold_dat$train_set |> train_user_movie_effect(lambda)
  })
  put_end_date(start)
  put_log("Function: train_user_movie_effect.cv:
User+Movie Effect list have been computed")

  user_movie_effects_united <- union_cv_results(user_movie_effects_ls)

  user_movie_effect <- user_movie_effects_united |>
    group_by(movieId) |>
    summarise(b = mean(b), n = mean(n))
  # ...
  user_movie_effect
}
```

The complete source code of the train_user_movie_effect.cv function is available in the Utility Functions section of the UM-effect.functions.R script on *GitHub*.

#### 5.3.2.3 calc_user_movie_effect_MSE Function

Validates the *UM Effect (UME) Model* built and trained by the train_user_movie_effect or train_user_movie_effect.cv function call (both described above in *this Section*).

##### 5.3.2.3.1 Usage

```
user_movie_effects_MSEs <- sapply(edx_CV, function(cv_fold_dat){
  cv_fold_dat$validation_set |> calc_user_movie_effect_MSE(um_effect)
})
```

In the code snippet above, the function `calc_user_movie_effect_MSE` is called internally from the calc_user_movie_effect_MSE.cv function (described below in *this Section*) for each iteration of the *K-Fold Cross-Validation*, to calculate the *MSE* for the corresponding validation dataset of the `edx_CV` Object (described in Appendix B: Models Training Datasets).

##### 5.3.2.3.2 Arguments

- **test_set:** Validation dataset;
- **um_effect:** Data frame object representing the *UM Effect Model*.

##### 5.3.2.3.3 Details

The function calculates *Mean Squared Error (MSE)* for the *UME Model* passed in the um_effect argument using the validation dataset passed in the test_set arg.

This is an auxiliary function intended to be called internally from the calc_user_movie_effect_MSE.cv (described below in *this Section*) as shown in the Usage section above of *this Description*.

##### 5.3.2.3.4 Value

*MSE* value calculated for the *UME Model*.

#### 5.3.2.3.5 Source Code

The source code of the calc_user_movie_effect_MSE function is shown below:

```
calc_user_movie_effect_MSE <- function(test_set, um_effect){
  mse.result <- test_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(um_effect, by = "movieId") |>
    mutate(resid = rating - clamp(mu + a + b)) |>
    pull(resid) |> mse()

  stopifnot(!is.na(mse.result))
  mse.result
}
```

The source code of the calc_user_movie_effect_MSE function is also available in the Utility Functions section of the UM-effect.functions.R script on *GitHub*.

#### 5.3.2.4 calc_user_movie_effect_MSE.cv Function

Validates the *UM Effect (UME) Model* built and trained by the train_user_movie_effect or train_user_movie_effect.cv function call (both described above in *this Section*) with use of *K-Fold Cross-Validation* (as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8]).

##### 5.3.2.4.1 Usage

```
user_movie_effects_MSE <- calc_user_movie_effect_MSE.cv(um_effect)
```

In the code snippet above, the function *being described here* is called internally from the calc_user_movie_effect_RMSE.cv function described below in *this Section*.

##### 5.3.2.4.2 Arguments

- **um_effect:** Data frame object representing the *UM Effect Model*.

##### 5.3.2.4.3 Details

This function calculates *Mean Squared Error (MSE)* for the *UME Model* passed in the um_effect argument using the validation datasets contained in the `edx_CV` Object described below in Appendix B: Models Training Datasets.

It calls internally the calc_user_movie_effect_MSE function for each iteration of the *K-Fold Cross-Validation*, calculating the *MSE* for the corresponding validation dataset of the `edx_CV` Object, and then calculates the mean of the obtained results.

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

This is an auxiliary function intended to be called internally from the calc_user_movie_effect_RMSE.cv as shown in the Usage section above of *this Description*.

##### 5.3.2.4.4 Value

The *MSE* value calculated for the *UME Model* using *K-Fold Cross-Validation* (see the Details section of *this Description* above for more details).

#### 5.3.2.4.5 Source Code

The source code of the calc_user_movie_effect_MSE.cv function is shown below:

```
calc_user_movie_effect_MSE.cv <- function(um_effect){
  put_log("Function: user_movie_effects_MSE.cv:
Computing the RMSE taking into account User+Movie Effects...")
  start <- put_start_date()
  user_movie_effects_MSEs <- sapply(edx_CV, function(cv_fold_dat){
    cv_fold_dat$validation_set |> calc_user_movie_effect_MSE(um_effect)
  })
  put_end_date(start)

  put_log1("Function: user_movie_effects_MSE.cv:
MSE values have been plotted for the %1-Fold Cross-Validation samples.",
           CVFolds_N)

  mean(user_movie_effects_MSEs)
}
```

The source code of the calc_user_movie_effect_MSE.cv function is also available in the Utility Functions section of the UM-effect.functions.R script on *GitHub*.

#### 5.3.2.5 calc_user_movie_effect_RMSE.cv Function

Validates the *UM Effect (UME) Model* (built and trained by the train_user_movie_effect or train_user_movie_effect.cv function call) with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

##### 5.3.2.5.1 Usage

```
cv.UM_effect.RMSE <- calc_user_movie_effect_RMSE.cv(cv.UM_effect)
```

```
cv.UM_effect.RMSE
```

```
## [1] 0.8732081
```

##### 5.3.2.5.2 Arguments

- **um_effect:** Data frame object representing the *UM Effect Model*.

##### 5.3.2.5.3 Details

The function calculates *Root Squared Error (RMSE)* for the *UME Model* passed in the um_effect argument using the validation datasets contained in the `edx_CV` Object described in Appendix B: Models Training Datasets.

This function internally calls the calc_user_movie_effect_MSE.cv function described above, to calculate the *MSE* for the *UME Model* with use of *K-Fold Cross-Validation*, and then calculates the square root of the result.
The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.2.5.4 Value

The *RMSE* value calculated for the *UME Model* using *K-Fold Cross-Validation* (see the Details section of *this Description* above for more details).

#### 5.3.2.5.5 Source Code

The source code of the calc_user_movie_effect_RMSE.cv function is shown below:

```
calc_user_movie_effect_RMSE.cv <- function(um_effect){
  user_movie_effects_MSE <- calc_user_movie_effect_MSE.cv(um_effect)
  um_effect_RMSE <- sqrt(user_movie_effects_MSE)
  put_log2("Function: user_movie_effects_RMSE.cv:
%1-Fold Cross-Validation ultimate RMSE: %2", length(edx_CV), um_effect_RMSE)
  um_effect_RMSE
}
```

The source code of the calc_user_movie_effect_RMSE.cv function is also available in the Utility Functions section of the UM-effect.functions.R script on *GitHub*.

### 5.3.3 UME Model: Regularization

#### 5.3.3.1 regularize.test_lambda.UM_effect.cv Function

Validates the *regularization parameter* $\lambda$ passed in the lambda argument for the *User+Movie Effect (UME) Model* in the *regularization process* using *penalized approach* with use of *K-Fold Cross-Validation*.

##### 5.3.3.1.1 Usage

```
  lambdas <- seq(0, 1, 0.1)
  cv.UME.preset.result <-
    tune.model_param(lambdas, regularize.test_lambda.UM_effect.cv)
```

```
str(cv.UME.preset.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    8 obs. of  2 variables:
##   ..$ RMSE           : num [1:8] 0.873 0.873 0.873 0.873 0.873 ...
##   ..$ parameter.value: num [1:8] 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
##  $ best_result : Named num [1:2] 0.4 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

In the code snippet above the function *being described here* is passed in the fn_tune.test.param_value argument to the tune.model_param function described above in section Model Tuning Utils of *this Appendix*.
Such a use of this function is also noted in the Details section of *this Description* below.

##### 5.3.3.1.2 Arguments

- **lambda:** *Regularization parameter* $\lambda$ used to validate the model in the *regularization process*.

##### 5.3.3.1.3 Details

The function is used in the *regularization process* of the *UME Model* using the *penalized approach* to regularize predictions for the given *regularization parameter* $\lambda$ (passed in the lambda argument) as explained in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14].

Internally it uses the train_user_movie_effect.cv and calc_user_movie_effect_RMSE.cv functions described above to validate the *UME Model* for the given $\lambda$ with use of the *K-Fold Cross-Validation* method, as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8], where the value of $K$ is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

This function is an auxiliary function specifically designed to be passed in the `fn_tune.test.param_value` argument to the tune.model_param (as shown above in the Usage section of *this Description*) and the model.tune.param_range functions described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.3.1.4 Value

The *Root Mean Squared Errors (RMSE)* value computed for the UME Model applying the *regularization parameter* $\lambda$ passed in the lambda argument.

#### 5.3.3.1.5 Source Code

The source code of the regularize.test_lambda.UM_effect.cv function is shown below:

```
regularize.test_lambda.UM_effect.cv <- function(lambda){
  if (is.na(lambda)) {
    stop("Function: regularize.test_lambda.UM_effect.cv
`lambda` is `NA`")
  }
  um_effect <- train_user_movie_effect.cv(lambda)
  calc_user_movie_effect_RMSE.cv(um_effect)
}
```

The source code of the regularize.test_lambda.UM_effect.cv function is also available in the Regularization section of the UM-effect.functions.R script on *GitHub*.

### 5.3.4 UMGE Model: Utility Functions

The complete source code of the functions described in this section is available in the Utility Functions section of the UMG-effect.functions.R script on *GitHub*.

#### 5.3.4.1 train_user_movie_genre_effect Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie+Genre Effect (UMGE) Model* using the training sample of the `Movielens` dataset (passed in the train.sgr argument), specially prepared to train the *UMGE Model*.

##### 5.3.4.1.1 Usage

```
rglr.UMG_effect <- edx.sgr |> train_user_movie_genre_effect(UMGE.rglr.best_lambda)
```

```
str(rglr.UMG_effect)
```

```
## tibble [10,677 x 2] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ g      : num [1:10677] -1.35e-05 -5.11e-05 -1.08e-04 -2.77e-05 -2.46e-04 ...
```

##### 5.3.4.1.2 Arguments

- **train.sgr:** Training sample of the `Movielens` dataset with split rows belonging to multiple genres;
- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.4.1.3 Details

The dataset passed in the train.sgr argument is assumed to be the result returned by the splitGenreRows helper function described above in section Data Processing Functions of *this Appendix*.

For example, the `edx.sgr` Object is an appropriate dataset as the value of the train.sgr.

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

*Regularization* is committed through the *penalized regression* by computing the *penalized mean* performed by the internally called mean_reg function described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.4.1.4 Value

Data frame object representing the trained (and also *regularized* if lambda arg value other than default) *UMG Effect Model*.

#### 5.3.4.1.5 Source Code

The source code of the train_user_movie_genre_effect function is shown below:

```
train_user_movie_genre_effect <- function(train.sgr, lambda = 0){
  if (is.na(lambda)) {
    stop("Function: train_user_movie_genre_effect
`lambda` is `NA`")
  }

  genre_bias <- train.sgr |>
      left_join(edx.user_effect, by = "userId") |>
      left_join(rglr.UM_effect, by = "movieId") |>
      mutate(resid = rating - (mu + a + b)) |>
      group_by(genres) |>
      summarise(g = mean_reg(resid, lambda), n = n())

    train.sgr |>
      left_join(genre_bias, by = "genres") |>
      left_join(rglr.UM_effect, by = "movieId") |>
      group_by(movieId) |>
      summarise(g = mean(g))
}
```

The source code of the train_user_movie_genre_effect function is also available in the Utility Functions section of the UMG-effect.functions.R script on *GitHub*.

#### 5.3.4.2 train_user_movie_genre_effect.cv Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie+Genre Effect (UMGE) Model* with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

##### 5.3.4.2.1 Usage

In the following code snippet, the function *being described here* is called with the default value of the lambda argument (therefore, the *Regularization technique* is not used in this call):

```
# `lambda = 0` by default (the regularization technique is not applied)
cv.UMG_effect <- train_user_movie_genre_effect.cv()
```

```
 str(cv.UMG_effect)
```

```
## tibble [10,677 x 2] (S3: tbl_df/tbl/data.frame)
##  $ movieId: int [1:10677] 1 2 3 4 5 6 7 8 9 10 ...
##  $ g      : num [1:10677] -9.28e-06 -7.23e-05 -1.07e-04 -6.46e-05 4.56e-05 ...
```

In the following code snippet, *this function* is called internally from the other custom helper function regularize.test_lambda.UMG_effect.cv described below in section UMGE Model: Regularization of *this Appendix*. In this case, when the function is passed a non-default value in the lambda argument, the *Regularization technique* is applied:

```
umg_effect <- train_user_movie_genre_effect.cv(lambda)
```

##### 5.3.4.2.2 Arguments

- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.4.2.3 Details

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

To use *regularization*, in *this Project*, the function is called internally from the other *custom helper function* regularize.test_lambda.UMG_effect.cv (described below in section UMGE Model: Regularization of *this Appendix*) as shown above in the Usage section of *this Description*.

Internally, this function calls the helper function train_user_movie_genre_effect (described above in *this Section*) for each iteration of the *K-Fold Cross-Validation* applied to the `edx_CV` Object described below

in Appendix B: Models Training Datasets. It also calls internally the union_cv_results function described above in section Models Training Functions of this *Appendix* to aggregate the cross-validation results.

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

#### 5.3.4.2.4 Value

Data frame object representing the *UMG Effect Model* trained (and also *regularized* if the lambda argument value is other than the default) with use of *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

#### 5.3.4.2.5 Source Code

The simplified version of the source code of the train_user_movie_genre_effect.cv function is shown below:

```
train_user_movie_genre_effect.cv <- function(lambda = 0){
  user_movie_genre_effects_ls <- lapply(kfold_index, function(fold_i){
    cv_fold_dat <- edx_CV[[fold_i]]
    umg_effect <- cv_fold_dat$train.sgr |> train_user_movie_genre_effect(lambda)

    put_log2("User+Movie+Genre Effects have been computed for the Fold %1
of the %2-Fold Cross Validation samples.",
             fold_i,
             CVFolds_N)

    umg_effect
  })

  user_movie_genre_effects_united <- union_cv_results(user_movie_genre_effects_ls)

  user_movie_genre_effect <- user_movie_genre_effects_united |>
    group_by(movieId) |>
    summarise(g = mean(g))

  if(lambda == 0) put_log("Function `train_user_movie_genre_effect.cv`:
Training completed: User+Movie+Genre Effects model.")
  else put_log1("Function `train_user_movie_genre_effect.cv`:
Training completed: User+Movie+Genre Effects model for lambda: %1...",
                lambda)

  user_movie_genre_effect
}
```

The complete source code of the train_user_movie_genre_effect.cv function is available in the Utility Functions section of the UMG-effect.functions.R script on *GitHub*.

#### 5.3.4.3 calc_user_movie_genre_effect_MSE Function

Validates the *UMG Effect (UMGE) Model* built and trained by the train_user_movie_genre_effect or train_user_movie_genre_effect.cv function call (both described above in *this Section*).

##### 5.3.4.3.1 Usage

```
user_movie_genre_effects_MSEs <- sapply(edx_CV, function(cv_dat){
  cv_dat$validation_set |> calc_user_movie_genre_effect_MSE(umg_effect)
})
```

In the code snippet above, the function *being described here* is called internally from the calc_user_movie_genre_effect_MSE.cv function (described below in *this Section*) for each iteration of the *K-Fold Cross-Validation*, to calculate the *MSE* for the corresponding validation dataset of the `edx_CV` Object (described in Appendix B: Models Training Datasets).

##### 5.3.4.3.2 Arguments

- **test_set:** Validation dataset;
- **umg_effect:** Data frame object representing the *UMG Effect Model*.

##### 5.3.4.3.3 Details

The function calculates *Mean Squared Error (MSE)* for the *UMGE Model* passed in the umg_effect argument using the validation dataset passed in the test_set arg.

This is an auxiliary function intended to be called internally from the calc_user_movie_genre_effect_MSE.cv (described below in *this Section*) as shown in the Usage section above of *this Description*.

##### 5.3.4.3.4 Value

*MSE* value calculated for the *UMGE Model*.

#### 5.3.4.3.5 Source Code

The source code of the calc_user_movie_genre_effect_MSE function is shown below:

```
calc_user_movie_genre_effect_MSE <- function(test_set, umg_effect){
  test_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(rglr.UM_effect, by = "movieId") |>
    left_join(umg_effect, by = "movieId") |>
    mutate(resid = rating - clamp(mu + a + b + g)) |>
    pull(resid) |> mse()
}
```

The source code of the calc_user_movie_genre_effect_MSE function is also available in the Utility Functions section of the UMG-effect.functions.R script on *GitHub*.

#### 5.3.4.4 calc_user_movie_genre_effect_MSE.cv Function

Validates the *UMG Effect (UMGE) Model* built and trained by the train_user_movie_genre_effect or train_user_movie_genre_effect.cv function call (both described above in *this Section*) with use of *K-Fold Cross-Validation* (as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8]).

##### 5.3.4.4.1 Usage

```
umg_effect_RMSE <- sqrt(calc_user_movie_genre_effect_MSE.cv(umg_effect))
```

In the code snippet above, the function *being described here* is called internally from the calc_user_movie_genre_effect_RMSE.cv function described below in *this Section*.

##### 5.3.4.4.2 Arguments

- **umg_effect:** Data frame object representing the *UMG Effect Model*.

##### 5.3.4.4.3 Details

This function calculates *MSE* for the *UMGE Model* passed in the umg_effect argument using the validation datasets contained in the `edx_CV` Object described below in Appendix B: Models Training Datasets.

It calls internally the calc_user_movie_genre_effect_MSE function for each iteration of the *K-Fold Cross-Validation*, calculating the *MSE* for the corresponding validation dataset of the `edx_CV` Object, and then calculates the mean of the obtained results.

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

This is an auxiliary function intended to be called internally from the calc_user_movie_genre_effect_RMSE.cv as shown in the Usage section above of *this Description*.

##### 5.3.4.4.4 Value

The *MSE* value calculated for the *UMGE Model* using *K-Fold Cross-Validation* (see the Details section of *this Description* above for more details).

#### 5.3.4.4.5 Source Code

The simplified version of the source code of the calc_user_movie_genre_effect_MSE.cv function is shown below:

```
calc_user_movie_genre_effect_MSE.cv <- function(umg_effect){
  user_movie_genre_effects_MSEs <- sapply(edx_CV, function(cv_dat){
    cv_dat$validation_set |> calc_user_movie_genre_effect_MSE(umg_effect)
  })

  mean(user_movie_genre_effects_MSEs)
}
```

The completed source code of the calc_user_movie_genre_effect_MSE.cv function is available in the Utility Functions section of the UMG-effect.functions.R script on *GitHub*.

#### 5.3.4.5 calc_user_movie_genre_effect_RMSE.cv Function

Validates the *UMG Effect (UMGE) Model* (built and trained by the train_user_movie_genre_effect or train_user_movie_genre_effect.cv function call) with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

##### 5.3.4.5.1 Usage

```
#### Compute UMGE Model RMSE ---------------------------------------------------
cv.UMG_effect.RMSE <- calc_user_movie_genre_effect_RMSE.cv(cv.UMG_effect)
```

```
cv.UMG_effect.RMSE
```

```
## [1] 0.872973
```

##### 5.3.4.5.2 Arguments

- **umg_effect:** Data frame object representing the *UMG Effect Model*.

##### 5.3.4.5.3 Details

The function calculates *Root Squared Error (RMSE)* for the *UMGE Model* passed in the umg_effect argument using the validation datasets contained in the `edx_CV` Object described below in Appendix B: Models Training Datasets.
During its execution, it internally calls the calc_user_movie_genre_effect_MSE.cv function (described above in *this Section*) to calculate the *MSE* for the *UMGE Model* (with use of *K-Fold Cross-Validation*), and then calculates the square root of the result.

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.4.5.4 Value

The *RMSE* value calculated for the *UMGE Model* using *K-Fold Cross-Validation* (see the Details section of *this Description* above for more details).

#### 5.3.4.5.5 Source Code

The simplified version of the source code of the calc_user_movie_genre_effect_RMSE.cv function is shown below:

```
calc_user_movie_genre_effect_RMSE.cv <- function(umg_effect){
  sqrt(calc_user_movie_genre_effect_MSE.cv(umg_effect))
}
```

The complete source code of the calc_user_movie_genre_effect_RMSE.cv function is available in the Utility Functions section of the UMG-effect.functions.R script on *GitHub*.

### 5.3.5 UMGE Model: Regularization

#### 5.3.5.1 regularize.test_lambda.UMG_effect.cv Function

Validates the *regularization parameter* $\lambda$ passed in the lambda argument for the *User+Movie+Genre Effect (UMGE) Model* in the *regularization process* using *penalized approach* with use of *K-Fold Cross-Validation*.

##### 5.3.5.1.1 Usage

```
lambdas <- seq(0, 0.2, 0.01)
cv.UMGE.preset.result <-
  tune.model_param(lambdas, regularize.test_lambda.UMG_effect.cv)
```

```
str(cv.UMGE.preset.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    8 obs. of  2 variables:
##   ..$ RMSE           : num [1:8] 0.873 0.873 0.873 0.873 0.873 ...
##   ..$ parameter.value: num [1:8] 0 0.01 0.02 0.03 0.04 0.05 0.06 0.07
##  $ best_result : Named num [1:2] 0.04 0.873
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

In the code snippet above the function *being described here* is passed in the fn_tune.test.param_value argument to the tune.model_param function described above in section Model Tuning Utils of *this Appendix*.
Such a use of this function is also noted in the Details section of *this Description* below.

##### 5.3.5.1.2 Arguments

- **lambda:** *Regularization parameter* $\lambda$ used to validate the model in the *regularization process*.

##### 5.3.5.1.3 Details

The function is used in the *regularization process* of the *UMGE Model* using the *penalized approach* to regularize predictions for the given *regularization parameter* $\lambda$ (passed in the lambda argument) as explained in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14].

Internally it uses the train_user_movie_genre_effect.cv and calc_user_movie_genre_effect_RMSE.cv functions (described above in UMGE Model: Utility Functions section) to validate the *UMGE Model* for the given $\lambda$ with use of the *K-Fold Cross-Validation* method, as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8], where the value of $K$ is determined by

the length of the edx_CV Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

This function is an auxiliary function specifically designed to be passed in the `fn_tune.test.param_value` argument to the tune.model_param (as shown above in the Usage section of *this Description*) and the model.tune.param_range functions described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.5.1.4 Value

The *Root Mean Squared Errors (RMSE)* value computed for the UMGE Model applying the *regularization parameter* $\lambda$ passed in the lambda argument.

#### 5.3.5.1.5 Source Code

The simplified version of the source code of the regularize.test_lambda.UMG_effect.cv function is shown below:

```
regularize.test_lambda.UMG_effect.cv <- function(lambda){
  umg_effect <- train_user_movie_genre_effect.cv(lambda)
  calc_user_movie_genre_effect_RMSE.cv(umg_effect)
}
```

The complete source code of the regularize.test_lambda.UMG_effect.cv function is available in the Regularization section of the UMG-effect.functions.R script on *GitHub*.

### 5.3.6 UMGYE Model: Utility Functions

The complete source code of the functions described in this section is available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.1 calc_date_general_effect Function

This is an auxiliary function that calculates the *Date General Effect* (optionally *regularizes* if the lambda argument value is non-default) of the user ratings, using a *training dataset* passed in the train_set argument, and is intended for use in other custom helper functions (such as train_UMGY_effect described below in *this Section*), as shown below in section Usage of *this Description*.

##### 5.3.6.1.1 Usage

```
train_set |>
  calc_date_general_effect(lambda) |>
  calc_UMGY_effect()
```

In the code snippet above, the function *being described here* is called, alongside the calc_UMGY_effect, within the train_UMGY_effect (both described below in *this Section*).

##### 5.3.6.1.2 Arguments

- **train_set:** Training dataset;
- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.6.1.3 Details

The *Regularization technique* (described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14]) is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has non-default value (to be more exact, greater than zero).

*Regularization* is performed through *penalized regression* by computing the *penalized mean* using the internally called mean_reg function (described above in section Model Tuning Utils of *this Appendix*).

#### 5.3.6.1.4 Value

*Data frame* object, representing (optionally, *Regularized*) *Day General Effect*.

#### 5.3.6.1.5 Source Code

Below is a slightly simplified version of the source code of the calc_date_general_effect function:

```
calc_date_general_effect <- function(train_set, lambda = 0){
  dg_effect <- train_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(rglr.UM_effect, by = "movieId") |>
    left_join(rglr.UMG_effect, by = "movieId") |>
    left_join(date_days_map, by = "timestamp") |>
    mutate(resid = rating - (mu + a + b + g)) |>
  group_by(days) |>
    summarise(de = mean_reg(resid, lambda),
              year = mean(year))

  if(lambda == 0) put_log("Function `calc_date_general_effect`:
Date Global Effect has been computed.")
  else put_log1("Function `calc_date_general_effect`:
Date Global Effect has been computed for lambda: %1...",
                lambda)
  dg_effect
}
```

The complete source code of the calc_date_general_effect function is available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.2 calc_date_general_effect.cv Function

This is an auxiliary function that calculates the *Date General Effect* (optionally *regularizes* if the lambda argument value is non-default) of the user ratings, using the *K-Fold Cross Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8], and is intended to be called within another custom helper function, the train_UMGY_effect.cv) (described below in *this Section*) as shown below in the Usage section of *this Description*.

##### 5.3.6.2.1 Usage

```
calc_date_general_effect.cv(lambda) |> calc_UMGY_effect()
```

In the above code snippet, the function *being described here* is called, alongside the calc_UMGY_effect, within the train_UMGY_effect.cv (both described below in *this Section*).

##### 5.3.6.2.2 Arguments

- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.6.2.3 Details

The *Regularization technique*, described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14], is applied to the model being trained only when the *regularization parameter* $\lambda$, passed in the lambda argument, has a non-default value (to be more exact, is a positive number).

Internally, this function calls the custom helper function calc_date_general_effect (described above in *this Section*) for each iteration of the *K-Fold Cross-Validation* applied to the `edx_CV` Object, described below in Appendix B: Models Training Datasets. It also internally calls the user-defined auxiliary function union_cv_results, described above in section Models Training Functions of *this Appendix*, to aggregate the cross-validation results.

The $K$ value for the *K-Fold Cross-Validation* method is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.6.2.4 Value

*Data frame* object, representing (optionally, *Regularized*) *Date General Effect*.

#### 5.3.6.2.5 Source Code

Below is a simplified version of the source code of the calc_date_general_effect.cv function:

```
calc_date_general_effect.cv <- function(lambda = 0){

  date_general_effect_ls <- lapply(edx_CV,  function(cv_fold_dat){
    cv_fold_dat$train_set |> calc_date_general_effect(lambda)
  })
  put_log1("Function `calc_date_general_effect.cv`:
Date Global Effect list has been computed for %1-Fold Cross Validation samples.",
           CVFolds_N)

  date_general_effect_united <- union_cv_results(date_general_effect_ls)

  date_general_effect <- date_general_effect_united |>
    group_by(days) |>
    summarise(de = mean(de, na.rm = TRUE), year = mean(year, na.rm = TRUE))

  if(lambda == 0) put_log("Function `calc_date_general_effect.cv`:
Training completed: Date Global Effects model.")
  else put_log1("Function `calc_date_general_effect.cv`:
Training completed: Date Global Effects model for lambda: %1...",
                lambda)

  date_general_effect
}
```

The complete source code of the calc_date_general_effect.cv function is available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.3 calc_UMGY_effect Function

Builds the *User+Movie+Genre+Year Effect (UMGYE) Model* based on the *Date General Effect* data structure returned by the calc_date_general_effect or calc_date_general_effect.cv function (both described above in *this Section*) and passed in the date_general_effect argument as shown in the example provided below in the Usage section of *this Description*.

##### 5.3.6.3.1 Usage

```
train_set |>
  calc_date_general_effect(lambda) |>
  calc_UMGY_effect()
```

In the code snippet above, the function *being described here* is called, alongside the calc_date_general_effect (described above in *this Section*), within the train_UMGY_effect (described below in *this Section*).

##### 5.3.6.3.2 Arguments

- **date_general_effect:** *Data frame* object representing the *Date General Effect*;

##### 5.3.6.3.3 Details

This is an auxiliary function intended to be called internally from the following functions (both described below in *this Section*):

- train_UMGY_effect (an example of calling the function *being described* from *this function* is shown above in the Usage section of *this Description*);
- train_UMGY_effect.cv.

##### 5.3.6.3.4 Value

Data frame object representing the *UMGYE Model*.

#### 5.3.6.3.5 Source Code

The source code of the calc_UMGY_effect function is shown below:

```
calc_UMGY_effect <- function(date_general_effect){
  date_general_effect |>
    group_by(year) |>
    summarise(ye = mean(de, na.rm = TRUE))
}
```

The source code of the calc_UMGY_effect function is also available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.4 train_UMGY_effect Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie+Genre+Year Effect* (hereafter *UMGYE* for short) *Model* using the training sample of the `Movielens` dataset passed in the train_set argument.

##### 5.3.6.4.1 Usage

```
  rglr.UMGY_effect <- edx |> train_UMGY_effect(UMGYE.rglr.best_lambda)
```

```
str(rglr.UMGY_effect)
```

```
## tibble [15 x 2] (S3: tbl_df/tbl/data.frame)
##  $ year: num [1:15] 1995 1996 1997 1998 1999 ...
##  $ ye  : num [1:15] 0.0018 0.06807 0.01206 0.01017 0.00636 ...
```

##### 5.3.6.4.2 Arguments

- **train_set:** Training sample of the `Movielens` dataset;
- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.6.4.3 Details

This function calls internally the following *user-defined auxiliary functions* (both described above in *this Section*):

- calc_date_general_effect;
- calc_UMGY_effect.

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

##### 5.3.6.4.4 Value

Data frame object representing the trained (and optionally *regularized*) *UMGY Effect Model*.

#### 5.3.6.4.5 Source Code

The source code of the train_UMGY_effect function is shown below:

```
train_UMGY_effect <- function(train_set, lambda = 0){
  if (is.na(lambda)) {
    stop("Function: train_UMGY_effect
`lambda` is `NA`")
  }

  train_set |>
    calc_date_general_effect(lambda) |>
    calc_UMGY_effect()
}
```

The source code of the train_UMGY_effect function is also available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.5 train_UMGY_effect.cv Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie+Genre+Year Effect (UMGYE) Model* with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.6.5.1 Usage

```
train_UMGY_effect.cv(lambda) |>
  calc_UMGY_effect_RMSE.cv()
```

In the code snippet above, the function *being described here* is called alongside the calc_UMGY_effect_RMSE.cv (described below in *this Section*) within the custom helper function regularize.test_lambda.UMGY_effect.cv (described below in section UMGYE Model: Regularization of *this Appendix*).

##### 5.3.6.5.2 Arguments

- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.6.5.3 Details

This function calls internally the following *user-defined auxiliary functions* (both described above in *this Section*):

- calc_date_general_effect.cv;
- calc_UMGY_effect.

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

To use *regularization*, in *this Project*, this function is called within the regularize.test_lambda.UMGY_effect.cv custom helper function (described below in section UMGYE Model: Regularization of *this Appendix*) as shown above in the Usage section of *this Description*.

##### 5.3.6.5.4 Value

Data frame object representing the trained (and optionally *regularized*) *UMGY Effect Model*.

#### 5.3.6.5.5 Source Code

The source code of the train_UMGY_effect.cv function is shown below:

```
train_UMGY_effect.cv <- function(lambda = 0){
  if (is.na(lambda)) {
    stop("Function: train_UMGY_effect.cv
`lambda` is `NA`")
  }

  calc_date_general_effect.cv(lambda) |> calc_UMGY_effect()
}
```

The source code of the train_UMGY_effect.cv function is also available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.6 calc_UMGY_effect_MSE Function

Validates the *UMGY Effect (UMGYE) Model* built and trained by the train_UMGY_effect or train_UMGY_effect.cv function call (both described above in *this Section*).

##### 5.3.6.6.1 Usage

In the following code snippet, the function *being described here* is called internally from the calc_UMGY_effect_MSE.cv function (described below in *this Section*) for each iteration of the *K-Fold Cross-Validation*, to calculate the *MSE* for the corresponding validation dataset of the `edx_CV` Object (described in Appendix B: Models Training Datasets).

```
UMGY_effect_MSEs <- sapply(edx_CV, function(cv_fold_dat){
  cv_fold_dat$validation_set |> calc_UMGY_effect_MSE(UMGY_effect)
})
```

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.6.6.2 Arguments

- **test_set:** *Validation* dataset;
- **UMGY_effect:** *Data frame* object representing the *UMGY Effect Model*.

##### 5.3.6.6.3 Details

The function calculates *Mean Squared Error (MSE)* for the *UMGYE Model* passed in the UMGY_effect argument using the validation dataset passed in the test_set arg.

This is an auxiliary function intended to be called within the calc_UMGY_effect_MSE.cv (described below in *this Section*) as shown in the Usage section above of *this Description*.

##### 5.3.6.6.4 Value

*MSE* value calculated for the *UMGYE Model*.

#### 5.3.6.6.5 Source Code

The source code of the calc_UMGY_effect_MSE function is shown below:

```
calc_UMGY_effect_MSE <- function(test_set, UMGY_effect){
  test_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(rglr.UM_effect, by = "movieId") |>
    left_join(rglr.UMG_effect, by = "movieId") |>
    left_join(date_days_map, by = "timestamp") |>
    left_join(UMGY_effect, by='year') |>
    mutate(resid = rating - clamp(mu + a + b + g + ye)) |>
    pull(resid) |> mse()
}
```

The source code of the calc_UMGY_effect_MSE function is also available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.7 calc_UMGY_effect_MSE.cv Function

Validates the *UMGY Effect (UMGYE) Model* built and trained by the train_UMGY_effect or train_UMGY_effect.cv function call (both described above in *this Section*) with use of *K-Fold Cross-Validation* (as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8]).

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.6.7.1 Usage

```
UMGY_effect_RMSE <- sqrt(calc_UMGY_effect_MSE.cv(UMGY_effect))
```

In the code snippet above, the function *being described here* is called internally from the calc_UMGY_effect_RMSE.cv function described below in *this Section*.

##### 5.3.6.7.2 Arguments

- **UMGY_effect:** *Data frame* object representing the *UMGY Effect Model*.

##### 5.3.6.7.3 Details

The function calculates the *MSE* metric for the *UMGYE Model* passed in the UMGY_effect argument using the validation datasets contained in the `edx_CV` Object described below in Appendix B: Models Training Datasets.

It internally calls the calc_UMGY_effect_MSE function for each iteration of the *K-Fold Cross-Validation*, calculating the *MSE* for the corresponding validation dataset of the `edx_CV` Object, and then calculates the mean of the obtained results.

This is an auxiliary function used internally in the calc_UMGY_effect_RMSE.cv function (described below in *this Section*) as shown in the Usage section above of *This Description*.

##### 5.3.6.7.4 Value

The *MSE* value calculated for the *UMGYE Model* using *K-Fold Cross-Validation* (see the Details section of *this Description* above for more details).

#### 5.3.6.7.5 Source Code

The simplified version of the source code of the calc_UMGY_effect_MSE.cv function is shown below:

```
calc_UMGY_effect_MSE.cv <- function(UMGY_effect){
  start <- put_start_date()
  UMGY_effect_MSEs <- sapply(edx_CV, function(cv_fold_dat){
    cv_fold_dat$validation_set |> calc_UMGY_effect_MSE(UMGY_effect)
  })
  put_end_date(start)
  put_log1("Function: calc_UMGY_effect_MSE.cv
Date (Year) Effect MSE values have been computed for the %1-Fold Cross Validation samples.",
           CVFolds_N)

  mean(UMGY_effect_MSEs)
}
```

The completed source code of the calc_UMGY_effect_MSE.cv function is available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

#### 5.3.6.8 calc_UMGY_effect_RMSE.cv Function

Validates the *UMGY Effect (UMGYE) Model* (built and trained by the train_UMGY_effect or train_UMGY_effect.cv function call) with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the edx_CV Object (in *this Project*, we use $K = 5$).

##### 5.3.6.8.1 Usage

```
#### Compute User+Movie+Genre+Year Effect Model RMSE ---------------------------
cv.UMGY_effect.RMSE <- calc_UMGY_effect_RMSE.cv(cv.UMGY_effect)
```

```
cv.UMGY_effect.RMSE
```

```
## [1] 0.8723973
```

##### 5.3.6.8.2 Arguments

- **UMGY_effect:** *Data frame* object representing the *UMGY Effect Model*.

##### 5.3.6.8.3 Details

The function calculates *Root Squared Error (RMSE)* for the *UMGYE Model* object passed in the UMGY_effect argument using the validation datasets contained in the edx_CV Object described below in Appendix B: Models Training Datasets.
During its execution, it internally calls the calc_UMGY_effect_MSE.cv function (described above in *this Section*) to calculate the *MSE* for the *UMGE Model* (with use of *K-Fold Cross-Validation*), and then calculates the square root of the result.

##### 5.3.6.8.4 Value

The *RMSE* value calculated for the *UMGYE Model* using *K-Fold Cross-Validation*.

#### 5.3.6.8.5 Source Code

The source code of the calc_UMGY_effect_RMSE.cv function is shown below:

```
calc_UMGY_effect_RMSE.cv <- function(UMGY_effect){
  UMGY_effect_RMSE <- sqrt(calc_UMGY_effect_MSE.cv(UMGY_effect))
  put_log2("%1-Fold Cross Validation ultimate RMSE: %2",
           CVFolds_N,
           UMGY_effect_RMSE)

  UMGY_effect_RMSE
}
```

The source code of the calc_UMGY_effect_RMSE.cv function is also available in the Utility Functions section of the UMGY-effect.functions.R script on *GitHub*.

### 5.3.7 UMGYE Model: Regularization

#### 5.3.7.1 regularize.test_lambda.UMGY_effect.cv Function

Validates the *regularization parameter* $\lambda$ passed in the lambda argument for the *User+Movie+Genre+Year Effect (UMGYE) Model* in the *regularization process* using *penalized approach* with use of *K-Fold Cross-Validation*.

##### 5.3.7.1.1 Usage

```
  lambdas <- seq(0, 512, 32)
  cv.UMGYE.preset.result <-
    tune.model_param(lambdas,
                     regularize.test_lambda.UMGY_effect.cv,
                     steps.beyond_min = 16)
```

```
str(cv.UMGYE.preset.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    17 obs. of  2 variables:
##   ..$ RMSE           : num [1:17] 0.872 0.872 0.872 0.872 0.872 ...
##   ..$ parameter.value: num [1:17] 0 32 64 96 128 160 192 224 256 288 ...
##  $ best_result : Named num [1:2] 224 0.872
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

In the code snippet above the function *being described here* is passed in the fn_tune.test.param_value argument to the tune.model_param function described above in section Model Tuning Utils of *this Appendix*.
Such a use of this function is also noted in the Details section of *this Description* below.

##### 5.3.7.1.2 Arguments

- **lambda:** *Regularization parameter* $\lambda$ used to validate the model in the *regularization process*.

#### 5.3.7.1.3 Details

The function is used in the *regularization process* of the *UMGYE Model* using the *penalized approach* to regularize predictions for the given *regularization parameter* $\lambda$ (passed in the lambda argument) as explained in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14].

Internally it uses the train_UMGY_effect.cv and calc_UMGY_effect_RMSE.cv functions (described above in UMGYE Model: Utility Functions section) to validate the *UMGYE Model* for the given $\lambda$ with use of the *K-Fold Cross-Validation* method, as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8], where the value of $K$ is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

This function is an auxiliary function specifically designed to be passed in the `fn_tune.test.param_value` argument to the tune.model_param (as shown above in the Usage section of *this Description*) and the model.tune.param_range functions described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.7.1.4 Value

The *Root Mean Squared Errors (RMSE)* value computed for the UMGYE Model applying the *regularization parameter* $\lambda$ passed in the lambda argument.

#### 5.3.7.1.5 Source Code

The source code of the regularize.test_lambda.UMGY_effect.cv function is shown below:

```
regularize.test_lambda.UMGY_effect.cv <- function(lambda){
  if (is.na(lambda)) {
    stop("Function: regularize.test_lambda.UMGY_effect.cv
`lambda` is `NA`")
  }

  train_UMGY_effect.cv(lambda) |>
    calc_UMGY_effect_RMSE.cv()
}
```

The source code of the regularize.test_lambda.UMGY_effect.cv function is also available in the Regularization section of the UMGY-effect.functions.R script on *GitHub*.

### 5.3.8 UMGYDE Model: Utility Functions

The complete source code of the functions described in this section is available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.1 calc_day_general_effect Function

Computes the *Day General Effect* (hereafter *DG Effect* or *DGE* for short), optionally *regularized* if the lambda argument value is non-default, of user ratings on a *given day since the earliest record* in the `edx` dataset, using a *training dataset* passed in the train_set argument.

##### 5.3.8.1.1 Usage

```
train_set |>
  calc_day_general_effect(lambda) |>
  calc_UMGY_SmoothedDay_effect(degree, span)
```

In the code snippet above, the function *being described here* is called, alongside the calc_UMGY_SmoothedDay_effect, within the train_UMGY_SmoothedDay_effect (both described below in this section).

##### 5.3.8.1.2 Arguments

- **train_set:** Training dataset containing data to compute the *DG Effect*;
- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.8.1.3 Details

This is an auxiliary function intended to be called within the following custom helper functions: the train_UMGY_SmoothedDay_effect (as shown above in the Usage section of *this Description*) and the calc_day_general_effect.cv (both described below in *this Section*).

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

*Regularization* is performed through *penalized regression* by computing the *penalized mean* using the internally called mean_reg function (described above in section Model Tuning Utils of *this Appendix*).

#### 5.3.8.1.4 Value

*Data frame* object, representing (optionally, *Regularized*) *Day General Effect*.

#### 5.3.8.1.5 Source Code

Below is a slightly simplified version of the source code of the calc_day_general_effect function:

```
calc_day_general_effect <- function(train_set, lambda = 0){
  gday_effect <- train_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(rglr.UM_effect, by = "movieId") |>
    left_join(rglr.UMG_effect, by = "movieId") |>
    left_join(date_days_map, by = "timestamp") |>
    left_join(rglr.UMGY_effect, by='year') |>
    mutate(resid = rating - (mu + a + b + g + ye)) |>
    group_by(days) |>
    summarise(de = mean_reg(resid, lambda),
              year = mean(year))

  if(lambda == 0) put_log("Function `calc_day_general_effect`:
Day General Effect has been computed.")
  else put_log1("Function `calc_day_general_effect`:
Day General Effect has been computed for lambda: %1...",
                lambda)
  gday_effect
}
```

The complete source code of the calc_day_general_effect function is available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.2 calc_day_general_effect.cv Function

This is an auxiliary function that calculates the *Day General Effect* (optionally *regularizes* if the lambda argument value is non-default) of the user ratings, using the *K-Fold Cross Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8], and is intended to be called within another custom helper function, the train_UMGY_SmoothedDay_effect.cv) (described below in *this Section*) as shown below in the Usage section of *this Description*.

##### 5.3.8.2.1 Usage

```
calc_day_general_effect.cv(lambda) |>
  calc_UMGY_SmoothedDay_effect(degree, span)
```

In the above code snippet, the function *being described here* is called, alongside the calc_UMGY_SmoothedDay_effect, within the train_UMGY_SmoothedDay_effect.cv (both described below in *this Section*).

##### 5.3.8.2.2 Arguments

- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.8.2.3 Details

The *Regularization technique*, described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14], is applied to the model being trained only when the *regularization parameter* $\lambda$, passed in the lambda argument, has a non-default value (to be more exact, is a positive number).

Internally, this function calls the custom helper functions calc_day_general_effect function (described above in *this Section*) for each iteration of the *K-Fold Cross-Validation* applied to the `edx_CV` Object described below in Appendix B: Models Training Datasets. It also internally calls the user-defined auxiliary function union_cv_results, described above in section Models Training Functions of *this Appendix*, to aggregate the cross-validation results.

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.8.2.4 Value

*Data frame* object, representing (optionally, *Regularized*) *Day General Effect*.

#### 5.3.8.2.5 Source Code

Below is a simplified version of the source code of the calc_day_general_effect.cv function:

```
calc_day_general_effect.cv <- function(lambda = 0){

  put_log1("Function `calc_day_general_effect.cv`:
Computing Day General Effect list for %1-Fold Cross Validation samples...",
           CVFolds_N)

  gday_effect_ls <- lapply(edx_CV,  function(cv_fold_dat){
    cv_fold_dat$train_set |> calc_day_general_effect(lambda)
  })

  gday_effect_united <- union_cv_results(gday_effect_ls)

  gday_effect_united |>
    group_by(days) |>
    summarise(de = mean(de), year = mean(year))

}
```

The complete source code of the calc_day_general_effect.cv function is available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.3 calc_UMGY_SmoothedDay_effect Function

Builds the *User+Movie+Genre+Year+SmoothedDay Effect* (hereafter *UMGYDE* for short) *Model* based on the *Day General Effect* (*DG Effect* or *DGE* for short) data structure returned by the calc_day_general_effect or calc_day_general_effect.cv function (both described above) and passed in the date_general_effect argument.

##### 5.3.8.3.1 Usage

```
train_set |>
  calc_day_general_effect(lambda) |>
  calc_UMGY_SmoothedDay_effect(degree, span)
```

In the code snippet above, the function *being described here* is called, alongside the calc_day_general_effect (described above in *this Section*), within the train_UMGY_SmoothedDay_effect (described below in *this Section*).

##### 5.3.8.3.2 Arguments

- **day_general_effect:** Data frame object representing the *DG Effect* (to pass in the `data` argument to the internally called stats::loess function);
- **degree:** (Optional, *Integer*, `NA` by default) The degree of the polynomials to pass in the same-named argument to the internally called stats::loess (with the default value replaced by `degree = 1`);
- **span:** (Optional, *Numeric*, `NA` by default) The parameter $\alpha$ (which controls the degree of smoothing) to pass in the same-named argument to the internally called stats::loess (with the default value replaced by `span = 0.75`).

##### 5.3.8.3.3 Details

This auxiliary function is a wrapper for the stats::loess `R` function, intended to be used internally in the following custom helper functions (both described below in *this Section*):

- train_UMGY_SmoothedDay_effect;
- train_UMGY_SmoothedDay_effect.cv.

An example of using it in the train_UMGY_SmoothedDay_effect function is shown above in the Usage section of *this Description*:

##### 5.3.8.3.4 Value

Data frame object representing the *UMGYDE Model*.

#### 5.3.8.3.5 Source Code

Below is a simplified version of the source code of the calc_UMGY_SmoothedDay_effect function:

```
calc_UMGY_SmoothedDay_effect <- function(day_gen_effect,
                                         degree = NA,
                                         span = NA){

  if(is.na(degree)) degree = 1
  if(is.na(span)) span = 0.75

  fit <- loess(de ~ days, span = span, degree = degree, data = day_gen_effect)
  smth_day_effect <- day_gen_effect |> mutate(de_smoothed = fit$fitted)

  put_log2("Function `calc_UMGY_SmoothedDay_effect`:
Model has been trained using `loess` function with the following parameters:
degree: %1;
span: %2.",
           degree,
           span)

  smth_day_effect
}
```

The complete source code of the calc_UMGY_SmoothedDay_effect function is available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.4 train_UMGY_SmoothedDay_effect Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie+Genre+Year+SmoothedDay Effect* (hereafter *UMGYDE* for short) *Model* using the training sample of the `Movielens` dataset passed in the train_set argument.

##### 5.3.8.4.1 Usage

```
lss.UMGYD_effect <- edx |>
  train_UMGY_SmoothedDay_effect(lss.best_degree, lss.best_span)
```

##### 5.3.8.4.2 Arguments

- **train_set:** Training sample of the `Movielens` dataset;
- **degree:** (Optional, *Integer*, `NA` by default) The degree of the polynomials to pass in the same-named argument to the internally called calc_UMGY_SmoothedDay_effect described above;
- **span:** (Optional, *Numeric*, `NA` by default) The parameter $\alpha$ (which controls the degree of smoothing) to pass in the same-named argument to the internally called calc_UMGY_SmoothedDay_effect described above.
- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.8.4.3 Details

Internally, this function calls the helper functions calc_day_general_effect and calc_UMGY_SmoothedDay_effect (both described above in this section), passing the values of the arguments degree and span to the latter function in the same-named arguments.

> The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

##### 5.3.8.4.4 Value

Data frame object representing the trained (and optionally *regularized*) *UMGYDE Model*.

##### 5.3.8.4.5 Source Code

The source code of the train_UMGY_SmoothedDay_effect function is shown below:

```
train_UMGY_SmoothedDay_effect <- function(train_set,
                                          degree = NA,
                                          span = NA,
                                          lambda = 0){
  if (is.na(lambda)) {
    stop("Function: train_UMGY_SmoothedDay_effect
`lambda` is `NA`")
  }

  train_set |>
    calc_day_general_effect(lambda) |>
    calc_UMGY_SmoothedDay_effect(degree, span)
}
```

The source code of the train_UMGY_SmoothedDay_effect function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.5 train_UMGY_SmoothedDay_effect.cv Function

Trains (and optionally *regularizes* if the lambda argument value is non-default) the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

##### 5.3.8.5.1 Usage

```
train_UMGY_SmoothedDay_effect.cv(best_degree,
                                 best_span,
                                 lambda)
```

In the above code snippet, the function *being described here* is called internally from the regularize.train_UMGYD_effect.cv function described below in section UMGYDE Model: Regularization of *this Appendix*.

##### 5.3.8.5.2 Arguments

- **degree:** (Optional, *Integer*, `NA` by default) The degree of the polynomials to pass in the same-named argument to the internally called calc_UMGY_SmoothedDay_effect described above in *this Section*;

- **span:** (Optional, *Numeric*, `NA` by default) The parameter $\alpha$ (which controls the degree of smoothing) to pass in the same-named argument to the internally called calc_UMGY_SmoothedDay_effect described above in *this Section*;

- **lambda:** (Optional, *Numeric*, 0 by default) Regularization parameter $\lambda$.

##### 5.3.8.5.3 Details

Internally, this function calls the auxiliary functions calc_day_general_effect.cv and calc_UMGY_SmoothedDay_effect (both described above in this section), passing the values of the arguments degree and span to the latter function in the same-named arguments.

This function is also called within other helper functions, such as the train_UMGY_SmoothedDay_effect.RMSE.cv and regularize.train_UMGYD_effect.cv (described below in sections UMGYDE Model: Tuning `loess` Params and UMGYDE Model: Regularization of *this Appendix*, respectively). The usage in the latter function is shown in the Usage section of *this Description* above.

The *Regularization technique* described in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14] is applied to the model being trained only when the *regularization parameter* $\lambda$ passed in the lambda argument, has a value other than the default (to be more exact, greater than zero).

#### 5.3.8.5.4 Value

Data frame object representing the trained (and optionally *regularized*) *UMGYDE Model*.

#### 5.3.8.5.5 Source Code

The source code of the train_UMGY_SmoothedDay_effect.cv function is shown below:

```
train_UMGY_SmoothedDay_effect.cv <- function(degree = NA,
                                             span = NA,
                                             lambda = 0){
  if (is.na(lambda)) {
    stop("Function: train_UMGY_SmoothedDay_effect.cv
`lambda` is `NA`")
  }

  calc_day_general_effect.cv(lambda) |>
    calc_UMGY_SmoothedDay_effect(degree, span)
}
```

The source code of the train_UMGY_SmoothedDay_effect.cv function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.6 UMGY_SmoothedDay_effect.predict Function

Predicts *user rating* values on the *test dataset* passed in the test_set argument based on the data of the trained *UMGYD Effect (UMGYDE) Model* passed in the day_smoothed_effect argument.

##### 5.3.8.6.1 Usage

```
final.UMGYDE.predicted <- final_holdout_test |>
  UMGY_SmoothedDay_effect.predict(rglr.day_smoothed_effect)
```

##### 5.3.8.6.2 Arguments

- **test_set:** *Validation* dataset;
- **day_smoothed_effect:** *Data frame* object representing the *UMGYDE Model*.

##### 5.3.8.6.3 Details

This function implements an algorithm for predicting the *user rating values* in accordance with the *mathematical description* of the *UMGYDE Model*, as described in section Mathematical Description of the UMGYDE Model of *this Report*.

This function is also called within the custom helper function calc_UMGY_SmoothedDay_effect.MSE described below in *this Section*.

##### 5.3.8.6.4 Value

*Data frame* object containing *predicted user rating* values based on the *UMGYDE Model*

#### 5.3.8.6.5 Source Code

The source code of the UMGY_SmoothedDay_effect.predict function is shown below:

```
UMGY_SmoothedDay_effect.predict <- function(test_set, day_smoothed_effect) {
  test_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(rglr.UM_effect, by = "movieId") |>
    left_join(rglr.UMG_effect, by = "movieId") |>
    left_join(date_days_map, by = "timestamp") |>
    left_join(rglr.UMGY_effect, by='year') |>
    left_join(day_smoothed_effect, by='days') |>
    mutate(predicted = clamp(mu + a + b + g +
                               ifelse(is.na(ye), 0, ye) +
                               ifelse(is.na(de_smoothed),
                                      0,
                                      de_smoothed))) |>
    select(userId, movieId, timestamp, rating, predicted)
}
```

The source code of the UMGY_SmoothedDay_effect.predict function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.7 calc_UMGY_SmoothedDay_effect.MSE Function

Validates the *UMGYD Effect (UMGYDE) Model* (represented by the data object passed in the day_smoothed_effect argument) on the *Test dataset* passed in the test_set argument.

##### 5.3.8.7.1 Usage

In the code snippet below, the function *being described here* is called from the calc_UMGY_SmoothedDay_effect.MSE.cv function (described below in *this Section*) for each iteration of the *K-Fold Cross-Validation*, to calculate the *MSE* for the corresponding validation dataset of the `edx_CV` Object (described in Appendix B: Models Training Datasets):

```
smth_day_effect_MSEs <- sapply(edx_CV, function(cv_fold_dat){
  cv_fold_dat$validation_set |>
    calc_UMGY_SmoothedDay_effect.MSE(day_smoothed_effect)
})
```

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.8.7.2 Arguments

- **test_set:** *Validation* dataset;
- **day_smoothed_effect:** *Data frame* object representing the *UMGYDE Model*.

##### 5.3.8.7.3 Details

The function calculates the *Mean Squared Error (MSE)* metric for the *UMGYDE Model*.

In *this Project*, the following functions (described above in *this Section*) return the data object representing the *UMGYDE Model* (to be passed in the day_smoothed_effect argument):

- calc_UMGY_SmoothedDay_effect
- train_UMGY_SmoothedDay_effect
- train_UMGY_SmoothedDay_effect.cv.

Internally, this function calls the user-defined auxiliary function UMGY_SmoothedDay_effect.predict (described above in *this Section*), passing to it the day_smoothed_effect argument value in the *same-named* argument.

This is an auxiliary function used internally in the calc_UMGY_SmoothedDay_effect.MSE.cv function (described below in *this Section*) as shown in the Usage section of *This Description* above.

#### 5.3.8.7.4 Value

*MSE* value calculated for the *UMGYDE Model*.

#### 5.3.8.7.5 Source Code

The source code of the calc_UMGY_SmoothedDay_effect.MSE function is shown below:

```
calc_UMGY_SmoothedDay_effect.MSE <- function(test_set, day_smoothed_effect) {
  test_set |>
    UMGY_SmoothedDay_effect.predict(day_smoothed_effect) |>
    mutate(resid = rating - predicted) |>
    pull(resid) |> mse()
}
```

The source code of the calc_UMGY_SmoothedDay_effect.MSE function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.8 calc_UMGY_SmoothedDay_effect.MSE.cv Function

Validates the *UMGYD Effect (UMGYDE) Model* represented by the data object passed in the day_smoothed_effect argument with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.8.8.1 Usage

```
calc_UMGY_SmoothedDay_effect.RMSE.cv <- function(day_smoothed_effect){
  sqrt(calc_UMGY_SmoothedDay_effect.MSE.cv(day_smoothed_effect))
}
```

In the code snippet above, the function *being described here* is called internally from the calc_UMGY_SmoothedDay_effect.RMSE.cv function described below in *this Section*.

##### 5.3.8.8.2 Arguments

- **day_smoothed_effect:** *Data frame* object representing the *UMGYDE Model*.

##### 5.3.8.8.3 Details

The function calculates the *MSE* metric for the *UMGYDE Model* passed in the day_smoothed_effect argument using the validation datasets contained in the `edx_CV` Object described below in Appendix B: Models Training Datasets.

In *this Project*, the following functions (described above in *this Section*) return the data object representing the *UMGYDE Model* (to be passed in the day_smoothed_effect argument):

- calc_UMGY_SmoothedDay_effect
- train_UMGY_SmoothedDay_effect
- train_UMGY_SmoothedDay_effect.cv.

Internally, this function calls the calc_UMGY_SmoothedDay_effect.MSE function for each iteration of the *K-Fold Cross-Validation*, passing day_smoothed_effect value in the same-named argument to the called function and calculating the *MSE* for the corresponding validation dataset of the `edx_CV` Object. After that, it calculates the mean of the obtained results.

This is an auxiliary function used internally in the calc_UMGY_SmoothedDay_effect.RMSE.cv function (described below in this section) as shown in the Usage section of *This Description* above.

#### 5.3.8.8.4 Value

The *MSE* value calculated for the *UMGYDE Model* using *K-Fold Cross-Validation* (see the Details section of *this Description* above for more details).

#### 5.3.8.8.5 Source Code

The slightly simplified version of the source code of the calc_UMGY_SmoothedDay_effect.MSE.cv function is shown below:

```
calc_UMGY_SmoothedDay_effect.MSE.cv <- function(day_smoothed_effect){
  smth_day_effect_MSEs <- sapply(edx_CV, function(cv_fold_dat){
    cv_fold_dat$validation_set |>
      calc_UMGY_SmoothedDay_effect.MSE(day_smoothed_effect)
  })
  put_end_date(start)
  put_log1("Function `calc_UMGY_SmoothedDay_effect.MSE.cv`:
MSE value have been computed for the %1-Fold Cross Validation samples.",
           CVFolds_N)
  mean(smth_day_effect_MSEs)
}
```

The completed source code of the calc_UMGY_SmoothedDay_effect.MSE.cv function is available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.8.9 calc_UMGY_SmoothedDay_effect.RMSE.cv Function

Validates the *UMGYD Effect (UMGYDE) Model* represented by the data object passed in the day_smoothed_effect argument with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

##### 5.3.8.9.1 Usage

```
lss.UMGYD_effect.RMSE <- calc_UMGY_SmoothedDay_effect.RMSE.cv(lss.UMGYD_effect)
```

```
lss.UMGYD_effect.RMSE
```

```
## [1] 0.870785
```

##### 5.3.8.9.2 Arguments

- **day_smoothed_effect:** *Data frame* object representing the *UMGYDE Model*.

##### 5.3.8.9.3 Details

The function calculates *Root Squared Error (RMSE)* for the *UMGYDE Model* object using the validation datasets contained in the `edx_CV` Object described below in Appendix B: Models Training Datasets.

During its execution, it internally calls the calc_UMGY_SmoothedDay_effect.MSE.cv auxiliary function described above in *this Section* to compute the *MSE* for the *UMGYDE Model* and then calculates the *square root* of the obtained result.

In *this Project*, a *UMGYDE* data object (representing the *UMGYDE Model*) being passed in the day_smoothed_effect argument) can be created by any of the following functions (described above in *this Section*):

- calc_UMGY_SmoothedDay_effect
- train_UMGY_SmoothedDay_effect
- train_UMGY_SmoothedDay_effect.cv.

This function is also called internally as an auxiliary function from the train_UMGY_SmoothedDay_effect.RMSE.cv and regularize.test_lambda.UMGYD_effect.cv functions described below in sections UMGYDE Model: Tuning `loess` Params and UMGYDE Model: Regularization, respectively.

#### 5.3.8.9.4 Value

*RMSE* value calculated for the *UMGYDE model* using *K-fold cross-validation*.

#### 5.3.8.9.5 Source Code

The source code of the calc_UMGY_SmoothedDay_effect.RMSE.cv function is shown below:

```
calc_UMGY_SmoothedDay_effect.RMSE.cv <- function(day_smoothed_effect){
  sqrt(calc_UMGY_SmoothedDay_effect.MSE.cv(day_smoothed_effect))
}
```

The source code of the calc_UMGY_SmoothedDay_effect.RMSE.cv function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

### 5.3.9 UMGYDE Model: Tuning `loess` Params

The complete source code of the functions described in this section is available in the Tuning `loess` Parameters section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.9.1 train_UMGY_SmoothedDay_effect.RMSE.cv Function

Trains and validate (without *regularization*) the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* for the given stats::loess parameters passed in the corresponding arguments (degree and span), with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

##### 5.3.9.1.1 Usage

```
train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 <- function(span) {
  train_UMGY_SmoothedDay_effect.RMSE.cv(degree = 1, span)
}
```

In the above code snippet, the function *being described here* is called within the train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 function described below in *this Section*.

##### 5.3.9.1.2 Arguments

- **degree:** (Optional, *Integer*, `NA` by default) *The loess parameter*: the degree of the polynomials to pass in the same-named argument to the internally called train_UMGY_SmoothedDay_effect.cv described above;

- **span:** (Optional, *Numeric*, `NA` by default) *The loess parameter*: the parameter $\alpha$ (which controls the degree of smoothing) to pass in the same-named argument to the internally called train_UMGY_SmoothedDay_effect.cv described above.

#### 5.3.9.1.3 Details

This is a wrapper for the function train_UMGY_SmoothedDay_effect.cv (described above in section UMGYDE Model: Utility Functions of *this Appendix*), which it calls internally, passing the degree and span values as arguments of the same name.

Unlike the inner function train_UMGY_SmoothedDay_effect.cv, this one additionally validates the *UMGYDE Model* (created and trained by this inner function), for which it calls another custom helper function, calc_UMGY_SmoothedDay_effect.RMSE.cv (also described above in section UMGYDE Model: Utility Functions), computing *RMSE* score for the model.

This is an auxiliary function intended to be called internally from the following helper functions (described below in this section), specifically designed for tuning the `degree` and `span` loess parameters:

- train_UMGY_SmoothedDay_effect.RMSE.cv.degree0: a helper function designed for tuning the loess `span` parameter while keeping the `degree` parameter fixed at 0;
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree1: a helper function designed for tuning the loess `span` parameter while keeping the `degree` parameter fixed at 1;
- train_UMGY_SmoothedDay_effect.RMSE.cv.degree2: a helper function designed for tuning the loess `span` parameter while keeping the `degree` parameter fixed at 2;

#### 5.3.9.1.4 Value

The *RMSE* value calculated for the *UMGYDE Model* fitted for the given loess parameters (passed in the degree and span arguments).

#### 5.3.9.1.5 Source Code

The slightly simplified version of the source code of the train_UMGY_SmoothedDay_effect.RMSE.cv function is shown below:

```
train_UMGY_SmoothedDay_effect.RMSE.cv <- function(degree = NA, span = NA) {
  smth_de.RMSE <- train_UMGY_SmoothedDay_effect.cv(degree, span) |>
    calc_UMGY_SmoothedDay_effect.RMSE.cv()

  put_log2("Function `train_UMGY_SmoothedDay_effect.RMSE.cv`:
RMSE has been computed for the `loess` function parameters:
degree = %1, span = %2.",
           degree,
           span)

  smth_de.RMSE
}
```

The complete source code of the train_UMGY_SmoothedDay_effect.RMSE.cv function is available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.9.2 train_UMGY_SmoothedDay_effect.RMSE.cv.degree0 Function

Trains and validates (without *regularization*) the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* for the following values of the stats::loess parameters:

- a fixed `degree` value of 0;
- the `span` value passed in the span argument.

##### 5.3.9.2.1 Usage

```
spans <- seq(0.0005, 1, 0.001)
lss.UMGYDE.preset.degree0.result <-
  tune.model_param(spans, train_UMGY_SmoothedDay_effect.RMSE.cv.degree0)
```

```
str(lss.UMGYDE.preset.degree0.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE           : num [1:5] 0.872 0.871 0.871 0.872 0.872
##   ..$ parameter.value: num [1:5] 0.0005 0.0015 0.0025 0.0035 0.0045
##  $ best_result : Named num [1:2] 0.0015 0.8712
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

In the code snippet above the function *being described here* is passed in the fn_tune.test.param_value argument to the tune.model_param function described above in section Model Tuning Utils of *this Appendix*.

##### 5.3.9.2.2 Arguments

- **span:** (Optional, *Numeric*, `NA` by default) *The loess parameter*: the parameter $\alpha$ (which controls the degree of smoothing) to pass in the same-named argument to the internally called train_UMGY_SmoothedDay_effect.RMSE.cv function described above.

##### 5.3.9.2.3 Details

This function is a wrapper for the function train_UMGY_SmoothedDay_effect.RMSE.cv (described above in *this Section*) that it calls internally, passing it the following stats::loess parameter values:

- `degree`: 0 in the degree argument;
- `span`: The span argument value (of this function) in the same-named argument (span).

This is an auxiliary function intended to be passed in the `fn_tune.test.param_value` argument to the tune.model_param and model.tune.param_range functions described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.9.2.4 Value

The *RMSE* value calculated for the *UMGYDE Model* fitted for the given loess parameters (`degree = 0` and the `span` value passed in the span argument), and trained with use of *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

#### 5.3.9.2.5 Source Code

The source code of the train_UMGY_SmoothedDay_effect.RMSE.cv.degree0 function is shown below:

```
train_UMGY_SmoothedDay_effect.RMSE.cv.degree0 <- function(span) {
  train_UMGY_SmoothedDay_effect.RMSE.cv(degree = 0, span)
}
```

The source code of the train_UMGY_SmoothedDay_effect.RMSE.cv.degree0 function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.9.3 train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 Function

Trains and validates (without *regularization*) the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* for the following values of the stats::loess parameters:

- a fixed `degree` value of 1;
- the `span` value passed in the span argument.

##### 5.3.9.3.1 Usage

```
spans <- seq(0.0005, 1, 0.001)
lss.UMGYDE.preset.degree1.result <-
    tune.model_param(spans, train_UMGY_SmoothedDay_effect.RMSE.cv.degree1)
```

```
str(lss.UMGYDE.preset.degree1.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE           : num [1:5] 0.872 0.871 0.871 0.872 0.872
##   ..$ parameter.value: num [1:5] 0.0005 0.0015 0.0025 0.0035 0.0045
##  $ best_result : Named num [1:2] 0.0015 0.8711
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

In the code snippet above the function *being described here* is passed in the fn_tune.test.param_value argument to the tune.model_param function described above in section Model Tuning Utils of *this Appendix*.

##### 5.3.9.3.2 Arguments

- **span:** (Optional, *Numeric*, `NA` by default) *The loess parameter*: the parameter $\alpha$ (which controls the degree of smoothing) to pass in the same-named argument to the internally called train_UMGY_SmoothedDay_effect.RMSE.cv function described above.

##### 5.3.9.3.3 Details

This function is a wrapper for the function train_UMGY_SmoothedDay_effect.RMSE.cv (described above in *this Section*) that it calls internally, passing it the following stats::loess parameter values:

- `degree`: 1 in the degree argument;
- `span`: The span argument value (of this function) in the same-named argument (span).

This is an auxiliary function intended to be passed in the `fn_tune.test.param_value` argument to the tune.model_param and model.tune.param_range functions described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.9.3.4 Value

The *RMSE* value calculated for the *UMGYDE Model* fitted for the given loess parameters (`degree = 1` and the `span` value passed in the span argument), and trained with use of *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

#### 5.3.9.3.5 Source Code

The source code of the train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 function is shown below:

```
train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 <- function(span) {
  train_UMGY_SmoothedDay_effect.RMSE.cv(degree = 1, span)
}
```

The source code of the train_UMGY_SmoothedDay_effect.RMSE.cv.degree1 function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.9.4 train_UMGY_SmoothedDay_effect.RMSE.cv.degree2 Function

Trains and validates (without *regularization*) the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* for the following values of the stats::loess parameters:

- a fixed `degree` value of 2;
- the `span` value passed in the span argument.

##### 5.3.9.4.1 Usage

```
spans <- seq(0.0005, 1, 0.001)
lss.UMGYDE.preset.degree2.result <-
  tune.model_param(spans, train_UMGY_SmoothedDay_effect.RMSE.cv.degree2)
```

```
str(lss.UMGYDE.preset.degree2.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE           : num [1:5] 0.872 0.871 0.871 0.871 0.871
##   ..$ parameter.value: num [1:5] 0.0005 0.0015 0.0025 0.0035 0.0045
##  $ best_result : Named num [1:2] 0.0015 0.871
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

In the code snippet above the function *being described here* is passed in the fn_tune.test.param_value argument to the tune.model_param function described above in section Model Tuning Utils of *this Appendix*.

##### 5.3.9.4.2 Arguments

- **span:** (Optional, *Numeric*, `NA` by default) *The loess parameter*: the parameter $\alpha$ (which controls the degree of smoothing) to pass in the same-named argument to the internally called train_UMGY_SmoothedDay_effect.RMSE.cv function described above.

##### 5.3.9.4.3 Details

This function is a wrapper for the function train_UMGY_SmoothedDay_effect.RMSE.cv (described above in *this Section*) that it calls internally, passing it the following stats::loess parameter values:

- `degree`: 2 in the degree argument;
- `span`: The span argument value (of this function) in the same-named argument (span).

This is an auxiliary function intended to be passed in the `fn_tune.test.param_value` argument to the tune.model_param and model.tune.param_range functions described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.9.4.4 Value

The *RMSE* value calculated for the *UMGYDE Model* fitted for the given loess parameters (`degree = 2` and the `span` value passed in the span argument), and trained with use of *K-Fold Cross-Validation*, where the $K$ is the length of the `edx_CV` Object (in *this Project*, we use $K = 5$).

#### 5.3.9.4.5 Source Code

The source code of the train_UMGY_SmoothedDay_effect.RMSE.cv.degree2 function is shown below:

```
train_UMGY_SmoothedDay_effect.RMSE.cv.degree2 <- function(span) {
  train_UMGY_SmoothedDay_effect.RMSE.cv(degree = 2, span)
}
```

The source code of the train_UMGY_SmoothedDay_effect.RMSE.cv.degree2 function is also available in the Utility Functions section of the UMGYD-effect.functions.R script on *GitHub*.

### 5.3.10 UMGYDE Model: Regularization

The complete source code of the functions described in this section is available in the Regularization section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.10.1 regularize.train_UMGYD_effect Function

Trains and *regularizes* the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* for a given value of the parameter $\lambda$ passed in the lambda argument and the *best values* of the stats::loess parameters (`degree` and `span`, preliminary figured out and stored in the global variable `lss.UMGYDE.best_params`), using the training set of the `Movielens` dataset passed in the train_set argument.

##### 5.3.10.1.1 Usage

```
  rglr.UMGYD_effect <- edx |>
    regularize.train_UMGYD_effect(UMGYDE.rglr.best_lambda)
```

```
str(rglr.UMGYD_effect)
```

```
## tibble [4,640 x 4] (S3: tbl_df/tbl/data.frame)
##  $ days       : int [1:4640] 0 385 388 389 392 393 394 396 397 399 ...
##  $ de         : Named num [1:4640] 0.0331 0.1537 0.1174 0.239 -0.0277 ...
##   ..- attr(*, "names")= chr [1:4640] "param.best_value" "param.best_value" "param.best_value" "param
##  $ year       : num [1:4640] 1995 1996 1996 1996 1996 ...
##  $ de_smoothed: num [1:4640] 0.025 0.1492 0.1638 0.1565 0.0627 ...
```

##### 5.3.10.1.2 Arguments

- **train_set:** Training sample of the `Movielens` dataset;
- **lambda:** Regularization parameter $\lambda$.

#### 5.3.10.1.3 Details

This function is a wrapper for the function train_UMGY_SmoothedDay_effect (described above in section UMGYDE Model: Utility Functions of *this Appendix*) that it calls internally, passing it the best values of the stats::loess parameters `degree` and `span` in the corresponding arguments (degree and span, respectively), stored in the global variable `lss.UMGYDE.best_params`:

```
print(lss.UMGYDE.best_params)
```

```
##     degree       span       RMSE
## 1.00000000 0.00109375 0.87082947
```

The function *being described here* is used in the *regularization process* of the *UMGYDE Model* using the *penalized approach* to regularize predictions for the given *regularization parameter* $\lambda$ (passed in the lambda argument) as explained in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14].

#### 5.3.10.1.4 Value

*Data frame* object representing the *UMGYDE Model*, regularized by the given *regularization parameter* $\lambda$ and tuned for the loess *best parameters* `degree` and `span`.

#### 5.3.10.1.5 Source Code

The source code of the regularize.train_UMGYD_effect function is shown below:

```
regularize.train_UMGYD_effect <- function(train_set, lambda) {
  best_degree <- lss.UMGYDE.best_params["degree"]
  best_span <- lss.UMGYDE.best_params["span"]

  train_set |>
    train_UMGY_SmoothedDay_effect(best_degree,
                                  best_span,
                                  lambda)
}
```

The source code of the regularize.train_UMGYD_effect function is also available in the Regularization section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.10.2 regularize.train_UMGYD_effect.cv Function

Trains and *regularizes* the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* for a given value of the parameter $\lambda$ passed in the lambda argument and the *best values* of the stats::loess parameters (`degree` and `span`, preliminary figured out and stored in the global variable `lss.UMGYDE.best_params`), with use of *K-Fold Cross-Validation* as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8].

The `K` value for the *K-Fold Cross-Validation* is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use `K = 5`).

##### 5.3.10.2.1 Usage

```
regularize.train_UMGYD_effect.cv(lambda) |>
  calc_UMGY_SmoothedDay_effect.RMSE.cv()
```

In the above code snippet, the function *being described here* is called within the regularize.test_lambda.UMGYD_effect.cv function described below in *this Section*, alongside the calc_UMGY_SmoothedDay_effect.RMSE.cv described above in section UMGYDE Model: Utility Functions.

##### 5.3.10.2.2 Arguments

- **lambda:** Regularization parameter $\lambda$.

##### 5.3.10.2.3 Details

This function is a wrapper for the function train_UMGY_SmoothedDay_effect.cv (described above in section UMGYDE Model: Utility Functions of *this Appendix*) that it calls internally, passing it the best values of the stats::loess parameters `degree` and `span` in the corresponding arguments (degree and span, respectively), stored in the global variable `lss.UMGYDE.best_params`:

```
print(lss.UMGYDE.best_params)
```

```
##     degree       span       RMSE
## 1.00000000 0.00109375 0.87082947
```

The function *being described here* is used in the *regularization process* of the *UMGYDE Model* using the *penalized approach* to regularize predictions for the given *regularization parameter* $\lambda$ (passed in the lambda argument) as explained in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14].

This is an auxiliary function intended to be called internally from the regularize.test_lambda.UMGYD_effect.cv function (described below in *this Section*) as shown in the Usage section of *this Description* above.

#### 5.3.10.2.4 Value

*Data frame* object representing the *UMGYDE Model*, tuned for the loess *best parameters* `degree` and `span`, and trained using *K-Fold Cross-Validation*.

#### 5.3.10.2.5 Source Code

The source code of the regularize.train_UMGYD_effect.cv function is shown below:

```
regularize.train_UMGYD_effect.cv <- function(lambda) {
  best_degree <- lss.UMGYDE.best_params["degree"]
  best_span <- lss.UMGYDE.best_params["span"]

  train_UMGY_SmoothedDay_effect.cv(best_degree,
                                   best_span,
                                   lambda)
}
```

The source code of the regularize.train_UMGYD_effect.cv function is also available in the Regularization section of the UMGYD-effect.functions.R script on *GitHub*.

#### 5.3.10.3 regularize.test_lambda.UMGYD_effect.cv Function

Validates the *regularization parameter* $\lambda$ passed in the lambda argument for the *User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model* in the *regularization process* using *penalized approach* with use of *K-Fold Cross-Validation*.

##### 5.3.10.3.1 Usage

```
lambdas <- seq(0, 256, 16)
cv.UMGYDE.preset.result <-
  tune.model_param(lambdas, regularize.test_lambda.UMGYD_effect.cv)
```

```
str(cv.UMGYDE.preset.result)
```

```
## List of 2
##  $ tuned.result:'data.frame':    5 obs. of  2 variables:
##   ..$ RMSE           : num [1:5] 0.871 0.871 0.871 0.871 0.871
##   ..$ parameter.value: num [1:5] 0 16 32 48 64
##  $ best_result : Named num [1:2] 16 0.871
##   ..- attr(*, "names")= chr [1:2] "param.best_value" "best_RMSE"
```

In the code snippet above the function *being described here* is passed in the fn_tune.test.param_value argument to the tune.model_param function described above in section Model Tuning Utils of *this Appendix*.
Such a use of this function is also noted in the Details section of *this Description* below.

##### 5.3.10.3.2 Arguments

- **lambda:** *Regularization parameter* $\lambda$ used to validate the model in the *regularization process*.

##### 5.3.10.3.3 Details

The function is used in the *regularization process* of the *UMGYDE Model* using the *penalized approach* to regularize predictions for the given *regularization parameter* $\lambda$ (passed in the lambda argument) as explained in Section 24.2 *Penalized least squares* of the *Course Textbook (New Edition)*[14].

Internally it uses the regularize.train_UMGYD_effect.cv (described above in *this Section*) and calc_UMGY_SmoothedDay_effect.RMSE.cv functions (described in section UMGYDE Model: Utility Functions above) to validate the *UMGYDE Model* for the given $\lambda$ with use of the *K-Fold Cross-Validation* method, as described in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8], where the value of $K$ is determined by the length of the `edx_CV` Object described below in Appendix B: Models Training Datasets (in *this Project*, we use $K = 5$).

This function is an auxiliary function specifically designed to be passed in the `fn_tune.test.param_value` argument to the tune.model_param (as shown above in the Usage section of *this Description*) and the model.tune.param_range functions described above in section Model Tuning Utils of *this Appendix*.

#### 5.3.10.3.4 Value

The *Root Mean Squared Errors (RMSE)* value computed for the UMGYDE Model applying the *regularization parameter* $\lambda$ passed in the lambda argument.

#### 5.3.10.3.5 Source Code

The source code of the regularize.test_lambda.UMGYD_effect.cv function is shown below:

```
regularize.test_lambda.UMGYD_effect.cv <- function(lambda){
  if (is.na(lambda)) {
    stop("Function: regularize.test_lambda.UMGYD_effect.cv
`lambda` is `NA`")
  }

  regularize.train_UMGYD_effect.cv(lambda) |>
    calc_UMGY_SmoothedDay_effect.RMSE.cv()
}
```

The source code of the regularize.test_lambda.UMGYD_effect.cv function is also available in the Regularization section of the UMGYD-effect.functions.R script on *GitHub*.

### 5.3.11 UMGYDE Model: Matrix Factorization

The complete source code of the functions described in this section is available in the MF.functions.R script on *GitHub*.

#### 5.3.11.1 mf.residual.dataframe Function

Computes residuals, defined as the difference between the observed and predicted values of the fully trained *UMGYDE Model* as described in section User+Movie+Genre+Year+SmoothedDay Effect (UMGYDE) Model.

##### 5.3.11.1.1 Usage

```
mf.edx.residual <- mf.residual.dataframe(edx)
```

##### 5.3.11.1.2 Arguments

- **train_set:** Training dataset;

##### 5.3.11.1.3 Details

This function calculates the *residuals* on the *training dataset* passed as the train_set argument, taking into account all the effects of the previously trained models, and creates a new *data frame* that includes the results as a *calculated column* `rsdsl`, along with the `movieId` and `userId` columns selected from the *training dataset*.

##### 5.3.11.1.4 Value

*Data frame* containing the following three variables:

- ***rsdl***: (*Numeric*) Calculated column containing results of the function computation;
- ***userId***: (*Integer*) User Identifier;
- ***movieId*** (*Integer*) Movie Identifier.

#### 5.3.11.1.5 Source Code

The source code of the mf.residual.dataframe function is shown below:

```
mf.residual.dataframe <- function(train_set){
  train_set |>
    left_join(edx.user_effect, by = "userId") |>
    left_join(rglr.UM_effect, by = "movieId") |>
    left_join(rglr.UMG_effect, by = "movieId") |>
    left_join(date_days_map, by = "timestamp") |>
    left_join(rglr.UMGY_effect, by='year') |>
    left_join(rglr.UMGYD_effect, by='days') |>
    mutate(rsdl = rating - (mu + a + b + g + ye + de_smoothed)) |>
    select(userId, movieId, rsdl)
}
```

The source code of the mf.residual.dataframe function is also available in the MF.functions.R script on *GitHub*.

# 6 Appendix B: Models Training Datasets

## 6.1 `edx.mx` Matrix Object

We use the array representation described in Section 17.6 *Analysis of variance (ANOVA)* of the *Course Textbook (New Edition)*, for the training data[24]. For this purpose, we will convert the `edx` dataset into a matrix using the tidyr::pivot_wider function:

```
  put_log("Function: `make_source_datasets`: Creating Rating Matrix from `edx` dataset...")
  edx.mx <- edx |>
    mutate(userId = factor(userId),
           movieId = factor(movieId)) |>
    select(movieId, userId, rating) |>
    pivot_wider(names_from = movieId, values_from = rating) |>
    column_to_rownames("userId") |>
    as.matrix()

  put_log("Function: `make_source_datasets`:
Matrix created: `edx.mx` of the following dimentions:")
  put(dim(edx.mx))
```

To create the `edx.mx` object, the above code snippet is used in the make_source_datasets function described in section Initializing Input Datasets of Appendix A.

The *code snippet* above uses *custom logging function* put_log described in section Logging Functions of Appendix A.

```
  str(edx.mx)
```

```
##  num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   ..$ : chr [1:10677] "122" "185" "292" "316" ...
```

## 6.2 edx.sgr Object

To account for the Movie Genre Effect more accurately, we need a dataset with split rows for movies belonging to multiple genres:

```
  put_log("Function: `make_source_datasets`:
To account for the Movie Genre Effect, we need a dataset with split rows
for movies belonging to multiple genres.")
  edx.sgr <- splitGenreRows(edx)
```

To create the edx.sgr object, the above code snippet is used in the make_source_datasets function described in section Initializing Input Datasets of Appendix A.

The *code snippet* above uses the *user-defined helper function* splitGenreRows (described in section Data Processing Functions) to split the rows of the original dataset (edx in this case), as well as the *custom logging function* put_log described in section Logging Functions of Appendix A.

```
str(edx.sgr)
```

```
## tibble [23,371,423 x 6] (S3: tbl_df/tbl/data.frame)
##  $ userId   : int [1:23371423] 1 1 1 1 1 1 1 1 1 1 ...
##  $ movieId  : int [1:23371423] 122 122 185 185 185 292 292 292 292 316 ...
##  $ rating   : num [1:23371423] 5 5 5 5 5 5 5 5 5 5 ...
##  $ timestamp: int [1:23371423] 838985046 838985046 838983525 838983525 838983525 838983421 838983421
##  $ title    : chr [1:23371423] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, The (19
##  $ genres   : chr [1:23371423] "Comedy" "Romance" "Action" "Crime" ...
```

```
summary(edx.sgr)
```

```
##      userId         movieId          rating        timestamp           title              genres
##  Min.   :    1   Min.   :    1   Min.   :0.500   Min.   :7.897e+08   Length:23371423    Length:233714
##  1st Qu.:18140   1st Qu.:  616   1st Qu.:3.000   1st Qu.:9.472e+08   Class :character   Class :chara
##  Median :35784   Median : 1748   Median :4.000   Median :1.042e+09   Mode  :character   Mode  :chara
##  Mean   :35886   Mean   : 4277   Mean   :3.527   Mean   :1.035e+09
##  3rd Qu.:53638   3rd Qu.: 3635   3rd Qu.:4.000   3rd Qu.:1.131e+09
##  Max.   :71567   Max.   :65133   Max.   :5.000   Max.   :1.231e+09
```

## 6.3 movie_map Object

To be able to map movie IDs to titles, we create the following lookup table:

```
movie_map <- edx |> select(movieId, title, genres) |>
    distinct(movieId, .keep_all = TRUE)

  put_log("Function: `make_source_datasets`: Dataset created: movie_map")
```

To create the `movie_map` object, the above code snippet is used in the make_source_datasets function described in section Initializing Input Datasets of Appendix A.

The *code snippet* above uses *custom logging function* put_log described in section Logging Functions of Appendix A.

```
str(movie_map)
```

```
## 'data.frame':    10677 obs. of  3 variables:
##  $ movieId: int  122 185 292 316 329 355 356 362 364 370 ...
##  $ title  : chr  "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
##  $ genres : chr  "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|Adv
```

```
summary(movie_map)
```

```
##     movieId         title              genres
##  Min.   :    1   Length:10677       Length:10677
##  1st Qu.: 2754   Class :character   Class :character
##  Median : 5434   Mode  :character   Mode  :character
##  Mean   :13105
##  3rd Qu.: 8710
##  Max.   :65133
```

Note that titles cannot be considered unique, so we cannot use them as IDs[6].

## 6.4 `date_days_map` Object

We have a `timestamp` field in the `edx` dataset. To be able to map the date, year, and number of days since the earliest record in the `edx` dataset with the corresponding value in this field, we create the following lookup table:

```
  put_log("Function: `make_source_datasets`: Creating Date-Days Map dataset...")
  date_days_map <- edx |>
    mutate(date_time = as_datetime(timestamp)) |>
    mutate(date = as_date(date_time)) |>
    mutate(year = year(date_time)) |>
    mutate(days = as.integer(date - min(date))) |>
    select(timestamp, date_time, date, year, days) |>
    distinct(timestamp, .keep_all = TRUE)

  put_log("Function: `make_source_datasets`: Dataset created: date_days_map")
```

To create the `date_days_map` object, the above code snippet is used in the make_source_datasets function described in section Initializing Input Datasets of Appendix A.

The *code snippet* above uses *custom logging function* put_log described in section Logging Functions of Appendix A.

```
  str(date_days_map)
```

```
## 'data.frame':    6519590 obs. of  5 variables:
##  $ timestamp: int  838985046 838983525 838983421 838983392 838984474 838983653 838984885 838983707 83
##  $ date_time: POSIXct, format: "1996-08-02 11:24:06" "1996-08-02 10:58:45" "1996-08-02 10:57:01" "199
##  $ date     : Date, format: "1996-08-02" "1996-08-02" "1996-08-02" "1996-08-02" ...
##  $ year     : num  1996 1996 1996 1996 1996 ...
##  $ days     : int  571 571 571 571 571 571 571 571 571 571 ...
```

```
  summary(date_days_map)
```

```
##    timestamp           date_time                       date                year          days
##  Min.   :7.897e+08   Min.   :1995-01-09 11:46:49.00   Min.   :1995-01-09   Min.   :1995   Min.   :
##  1st Qu.:9.783e+08   1st Qu.:2001-01-01 05:05:01.75   1st Qu.:2001-01-01   1st Qu.:2001   1st Qu.:218
##  Median :1.091e+09   Median :2004-08-03 01:08:18.50   Median :2004-08-03   Median :2004   Median :349
##  Mean   :1.066e+09   Mean   :2003-10-10 23:15:02.07   Mean   :2003-10-10   Mean   :2003   Mean   :319
##  3rd Qu.:1.152e+09   3rd Qu.:2006-07-04 20:41:57.50   3rd Qu.:2006-07-04   3rd Qu.:2006   3rd Qu.:419
##  Max.   :1.231e+09   Max.   :2009-01-05 05:02:16.00   Max.   :2009-01-05   Max.   :2009   Max.   :51
```

## 6.5 edx_CV Object

edx_CV object is a list of sample objects we need to perform the *K-Fold Cross-Validation* as explained in Section 29.6 *Cross validation / K-fold cross validation* of the *Course Textbook (New Edition)*[8]:

```
  start <- put_start_date()
  edx_CV <- lapply(kfold_index,  function(fold_i){

    put_log1("Method `make_source_datasets`:
Creating K-Fold Cross-Validation Datasets, Fold %1", fold_i)

    #> We split the initial datasets into training sets, which we will use to build
    #> and train our models, and validation sets in which we will compute the accuracy
    #> of our predictions, the way described in the `Section 23.1.1 Movielens data`
    #> (https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movielens-data)
    #> of the Course Textbook.

    split_sets <- edx |>
      sample_train_validation_sets(fold_i*1000)

    train_set <- split_sets$train_set
    validation_set <- split_sets$validation_set

    put_log("Function: `make_source_datasets`:
Sampling 20% from the split-row version of the `edx` dataset...")
    split_sets.gs <- edx.sgr |>
      sample_train_validation_sets(fold_i*2000)

    train.sgr <- split_sets.gs$train_set
    validation.sgr <- split_sets.gs$validation_set

    # put_log("Function: `make_source_datasets`: Dataset created: validation.sgr")
    # put(summary(validation.sgr))

    #> We will use the array representation described in `Section 17.5 of the Textbook`
    #> (https://rafalab.dfci.harvard.edu/dsbook-part-2/linear-models/treatment-effect-models.html#sec-a
    #> for the training data.
    #> To create this matrix, we use `tidyr::pivot_wider` function:

    put_log("Function: `make_source_datasets`: Creating Rating Matrix from Train Set...")
    train_mx <- train_set |>
      mutate(userId = factor(userId),
             movieId = factor(movieId)) |>
      select(movieId, userId, rating) |>
      pivot_wider(names_from = movieId, values_from = rating) |>
      column_to_rownames("userId") |>
      as.matrix()

    put_log("Function: `make_source_datasets`:
Matrix created: `train_mx` of the following dimentions:")
    put(dim(train_mx))
```

```
    list(train_set = train_set,
         train_mx = train_mx,
         train.sgr = train.sgr,
         validation_set = validation_set)
  })
  put_end_date(start)
  put_log("Function: `make_source_datasets`:
Set of K-Fold Cross-Validation datasets created: edx_CV")
```

To create the `edx_CV` object, the above code snippet is used in the make_source_datasets function described in section Initializing Input Datasets of Appendix A.

The $K$ value for the *K-Fold Cross-Validation* is determined by the length of the list composing the `edx_CV` object. In *this Project*, we choose $K = 5$.

The *code snippet* above uses the *user-defined helper function* sample_train_validation_sets (described in section Data Processing Functions) to split the original dataset (`edx` in this case) into *Train* and *Validation* datasets, as well as the following *custom logging function* described in section Logging Functions of Appendix A:

- put_start_date;
- put_end_date;
- put_log;
- put_log1.

```
str(edx_CV)
```

```
## List of 5
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172311 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172311] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172311] 122 185 292 329 356 362 364 370 420 466 ...
##   .. ..$ rating   : num [1:7172311] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172311] 838985046 838983525 838983421 838983392 838983653 838984885 8389
##   .. ..$ title    : chr [1:7172311] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Star Trel
##   .. ..$ genres   : chr [1:7172311] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thr
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "329" ...
##   ..$ train.sgr     : tibble [18,669,190 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669190] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669190] 122 122 185 185 292 292 292 292 316 316 ...
##   .. ..$ rating   : num [1:18669190] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669190] 838985046 838985046 838983525 838983525 838983421 838983421 838
##   .. ..$ title    : chr [1:18669190] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, Th
##   .. ..$ genres   : chr [1:18669190] "Comedy" "Romance" "Action" "Crime" ...
##   ..$ validation_set:'data.frame':   1827744 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827744] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827744] 316 355 377 588 260 376 648 1049 110 1252 ...
##   .. ..$ rating   : num [1:1827744] 5 5 5 5 5 3 2 3 4.5 4 ...
##   .. ..$ timestamp: int [1:1827744] 838983392 838984474 838983834 838983339 868244562 868245920 8682
```

```
##   .. ..$ title    : chr [1:1827744] "Stargate (1994)" "Flintstones, The (1994)" "Speed (1994)" "Alado
##   .. ..$ genres   : chr [1:1827744] "Action|Adventure|Sci-Fi" "Children|Comedy|Fantasy" "Action|Roma
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172306 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172306] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172306] 122 185 292 316 329 355 356 364 370 377 ...
##   .. ..$ rating   : num [1:7172306] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172306] 838985046 838983525 838983421 838983392 838983392 838984474 83898
##   .. ..$ title    : chr [1:7172306] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172306] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thr
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,201 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669201] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669201] 122 122 185 185 185 292 292 316 316 329 ...
##   .. ..$ rating   : num [1:18669201] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669201] 838985046 838985046 838983525 838983525 838983525 838983421 8389
##   .. ..$ title    : chr [1:18669201] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, Th
##   .. ..$ genres   : chr [1:18669201] "Comedy" "Romance" "Action" "Crime" ...
##   ..$ validation_set:'data.frame':   1827749 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827749] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827749] 362 520 539 594 539 590 733 1210 1252 1408 ...
##   .. ..$ rating   : num [1:1827749] 5 5 5 5 3 5 3 4 4 3.5 ...
##   .. ..$ timestamp: int [1:1827749] 838984885 838984679 838984068 838984679 868246262 868245608 86824
##   .. ..$ title    : chr [1:1827749] "Jungle Book, The (1994)" "Robin Hood: Men in Tights (1993)" "Sle
##   .. ..$ genres   : chr [1:1827749] "Adventure|Children|Romance" "Comedy" "Comedy|Drama|Romance" "Ani
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172307 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172307] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172307] 122 185 292 316 329 355 362 370 377 420 ...
##   .. ..$ rating   : num [1:7172307] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172307] 838985046 838983525 838983421 838983392 838983392 838984474 83898
##   .. ..$ title    : chr [1:7172307] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172307] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thr
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,195 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669195] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669195] 122 122 185 185 185 292 292 292 316 329 ...
##   .. ..$ rating   : num [1:18669195] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669195] 838985046 838985046 838983525 838983525 838983525 838983421 8389
##   .. ..$ title    : chr [1:18669195] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, Th
##   .. ..$ genres   : chr [1:18669195] "Comedy" "Romance" "Action" "Crime" ...
##   ..$ validation_set:'data.frame':   1827748 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827748] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827748] 356 364 539 616 590 719 780 786 151 213 ...
##   .. ..$ rating   : num [1:1827748] 5 5 5 5 5 3 3 3 4.5 5 ...
##   .. ..$ timestamp: int [1:1827748] 838983653 838983707 838984068 838984941 868245608 868246191 86824
##   .. ..$ title    : chr [1:1827748] "Forrest Gump (1994)" "Lion King, The (1994)" "Sleepless in Seat
##   .. ..$ genres   : chr [1:1827748] "Comedy|Drama|Romance|War" "Adventure|Animation|Children|Drama|Mu
```

```
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172311 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172311] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172311] 122 185 292 316 329 355 356 362 364 370 ...
##   .. ..$ rating   : num [1:7172311] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172311] 838985046 838983525 838983421 838983392 838983392 838984474 8389
##   .. ..$ title    : chr [1:7172311] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172311] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thr
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,192 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669192] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669192] 122 122 185 185 292 292 316 316 329 329 ...
##   .. ..$ rating   : num [1:18669192] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669192] 838985046 838985046 838983525 838983525 838983421 838983421 838
##   .. ..$ title    : chr [1:18669192] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, Th
##   .. ..$ genres   : chr [1:18669192] "Comedy" "Romance" "Action" "Thriller" ...
##   ..$ validation_set:'data.frame':   1827744 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827744] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827744] 377 520 588 616 110 648 1049 1356 1148 1276 ...
##   .. ..$ rating   : num [1:1827744] 5 5 5 5 5 2 3 3 4 3.5 ...
##   .. ..$ timestamp: int [1:1827744] 838983834 838984679 838983339 838984941 868245777 868244699 8682
##   .. ..$ title    : chr [1:1827744] "Speed (1994)" "Robin Hood: Men in Tights (1993)" "Aladdin (1992
##   .. ..$ genres   : chr [1:1827744] "Action|Romance|Thriller" "Comedy" "Adventure|Animation|Children
##  $ :List of 4
##   ..$ train_set     :'data.frame':   7172301 obs. of  6 variables:
##   .. ..$ userId   : int [1:7172301] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:7172301] 122 185 292 316 355 356 364 370 420 466 ...
##   .. ..$ rating   : num [1:7172301] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:7172301] 838985046 838983525 838983421 838983392 838984474 838983653 8389
##   .. ..$ title    : chr [1:7172301] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate
##   .. ..$ genres   : chr [1:7172301] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thr
##   ..$ train_mx      : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA NA ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:69878] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:10677] "122" "185" "292" "316" ...
##   ..$ train.sgr     : tibble [18,669,194 x 6] (S3: tbl_df/tbl/data.frame)
##   .. ..$ userId   : int [1:18669194] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ movieId  : int [1:18669194] 122 122 185 185 292 292 316 329 329 355 ...
##   .. ..$ rating   : num [1:18669194] 5 5 5 5 5 5 5 5 5 5 ...
##   .. ..$ timestamp: int [1:18669194] 838985046 838985046 838983525 838983525 838983421 838983421 838
##   .. ..$ title    : chr [1:18669194] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, Th
##   .. ..$ genres   : chr [1:18669194] "Comedy" "Romance" "Crime" "Thriller" ...
##   ..$ validation_set:'data.frame':   1827754 obs. of  6 variables:
##   .. ..$ userId   : int [1:1827754] 1 1 1 1 2 2 2 2 3 3 ...
##   .. ..$ movieId  : int [1:1827754] 329 362 377 594 110 376 539 736 1252 1408 ...
##   .. ..$ rating   : num [1:1827754] 5 5 5 5 5 3 3 3 4 3.5 ...
##   .. ..$ timestamp: int [1:1827754] 838983392 838984885 838983834 838984679 868245777 868245920 8682
##   .. ..$ title    : chr [1:1827754] "Star Trek: Generations (1994)" "Jungle Book, The (1994)" "Speed
##   .. ..$ genres   : chr [1:1827754] "Action|Adventure|Drama|Sci-Fi" "Adventure|Children|Romance" "Act
```

# References


[1] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 24: Regularization, Section 24.1: Case study: recommendation systems / Loss function*. Oct. 27, 2025. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#sec-netflix-loss-function (visited on 10/27/2025) (cit. on pp. 7, 143, 145–147).

[2] Robert M. Bell Andreas Toscher Michael Jahrer. *The BigChaos Solution to the Netflix Grand Prize. commendo research & consulting*. Sept. 5, 2009. URL: https://www.asc.ohio-state.edu/statistics/statgen/joul_aut2009/BigChaos.pdf (visited on 02/18/2025) (cit. on pp. 7, 18).

[3] Azamat Kurbanaev. *edX Data Science: Capstone, MovieLens Datasets. Package: edx.capstone.movielens.data*. Version 0.0.0.9000. Feb. 5, 2025. URL: https://github.com/AzKurban-edX-DS/edx.capstone.movielens.data (visited on 02/05/2025) (cit. on p. 7).

[4] Rafael A. Irizarry. *Introduction to Data Science, First Edition, Chapter 33: Large datasets, Section 33.7.1: Movilens data. Data Analysis and Prediction Algorithms with R*. Oct. 24, 2019. URL: https://rafalab.dfci.harvard.edu/dsbook/large-datasets.html#movielens-data (visited on 10/24/2019) (cit. on pp. 9–11).

[5] Azamat Kurbanaev. *edX Data Science: Capstone-MovieLens Project. A movie recommendation system using the MovieLens dataset*. Version 1.0.0.0. May 5, 2025. URL: https://github.com/AzKurban-edX-DS/Capstone-MovieLens/tree/main (visited on 05/05/2025) (cit. on p. 13).

[6] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 24: Regularization, Section 24.1: Case study: recommendation systems. Statistics and Prediction Algorithms Through Case Studies*. Oct. 27, 2025. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#sec-recommendation-systems (visited on 10/27/2025) (cit. on pp. 13, 138, 288).

[7] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 29: Resampling and Model Assessment, Section 29.6: Cross validation*. Oct. 27, 2025. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/ml/resampling-methods.html#cross-validation (visited on 10/27/2025) (cit. on pp. 13, 192).

[8] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 29: Resampling and Model Assessment, Section 29.6: Cross validation / K-fold cross validation*. Oct. 27, 2025. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/ml/resampling-methods.html#k-fold-cross-validation (visited on 10/27/2025) (cit. on pp. 13, 145, 195, 206, 208, 211, 215, 217, 219, 223, 227, 229, 231, 235, 241, 245, 247, 250, 253, 259, 265, 267, 269, 280, 282, 290).

[9] Rafael A. Irizarry. *Introduction to Data Science, First Edition, Chapter 33: Large datasets, Section 33.7.4: A first model. Data Analysis and Prediction Algorithms with R*. Oct. 24, 2019. URL: https://rafalab.dfci.harvard.edu/dsbook/large-datasets.html#a-first-model (visited on 10/24/2019) (cit. on pp. 14, 18).

[10] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 24: Regularization, Section 24.1: Case study: recommendation systems / User effects*. Oct. 27, 2025. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#user-effects (visited on 10/27/2025) (cit. on pp. 19, 21).

[11] Robert Bell Yehuda Koren Yahoo Research and Chris Volinsky. *Matrix Factorization Techniques for Recommender Systems*. Aug. 1, 2009. URL: https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf (visited on 02/18/2025) (cit. on p. 21).

[12] Rafael A. Irizarry. *Introduction to Data Science, First Edition, Chapter 33: Large datasets, Section 33.7.6: User Effects. Data Analysis and Prediction Algorithms with R*. Oct. 24, 2019. URL: https://rafalab.dfci.harvard.edu/dsbook/large-datasets.html#user-effects (visited on 10/24/2019) (cit. on p. 21).

[13] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 24: Regularization, Section 24.1: Case study: recommendation systems / Movie effects*. Oct. 27, 2025. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#sec-movie-effects (visited on 10/27/2025) (cit. on pp. 25, 28).

[14] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 24: Regularization, Section 24.2: Penalized least squares*. Oct. 27, 2025. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#penalized-least-squares (visited on 10/27/2025) (cit. on pp. 31, 160, 167, 209, 211, 219, 221, 223, 231, 233, 235, 239, 241, 250, 251, 253, 257, 260, 279, 280, 282).

[15] Rafael A. Irizarry. *Introduction to Data Science, First Edition, Chapter 33: Large datasets, Section 33.9.2: Penalized least squares. Data Analysis and Prediction Algorithms with R*. Oct. 24, 2019. URL: https://rafalab.dfci.harvard.edu/dsbook/large-datasets.html#penalized-least-squares (visited on 10/24/2019) (cit. on p. 31).

[16] Francesco Ricci. *Recommender Systems Handbook*. Ed. by Paul B. Kantor Lior Rokach Bracha Shapira. Springer, New York, 2011. ISBN: ISBN 978-0-387-85819-7. DOI: 10.1007/978-0-387-85820-3. URL: https://github.com/vwang0/recommender_system/blob/master/Recommender%20Systems%20Handbook.pdf (cit. on pp. 40, 106).

[17] Amir Motefaker. *Movie Recommendation System using R - BEST*. Version 284. July 18, 2024. URL: https://www.kaggle.com/code/amirmotefaker/movie-recommendation-system-using-r-best/notebook (visited on 02/18/2025) (cit. on pp. 52, 53).

[18] Rafael A. Irizarry. *Introduction to Data Science, First Edition, Chapter 33: Large datasets, Section 33.11 Matrix factorization. Data Analysis and Prediction Algorithms with R*. Oct. 24, 2019. URL: https://rafalab.dfci.harvard.edu/dsbook/large-datasets.html#matrix-factorization (visited on 10/24/2019) (cit. on p. 97).

[19] Yixuan Qiu. *recosystem: Recommender System Using Parallel Matrix Factorization*. May 5, 2023. URL: https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html (visited on 05/05/2023) (cit. on p. 98).

[20] Wei-Sheng Ghin et al. *A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems. ACM TIST 2015a*. 2015. URL: https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/libmf_journal.pdf (cit. on p. 98).

[21] Wei-Sheng Chin et al. *A Learning-Rate Schedule for Stochastic Gradient Methods to Matrix Factorization. PAKDD 2015b*. 2015. URL: https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/mf_adaptive_pakdd.pdf (cit. on p. 98).

[22] Dr. Rahul Mehta Manisha Valera. *Advanced Deep Learning Models for Improving Movie Rating Predictions: A Benchmarking Study*. Dec. 1, 2024. URL: https://www.sciencedirect.com/science/article/pii/S2772485925000134#:~:text=In%20this%20work%2C%20the%20authors,capturing%20user%20preferences%20over%20time. (cit. on p. 106).

[23] The Pragmatic Editorial Team. *The 4 Important Aspects of Data Science*. URL: https://www.pragmaticinstitute.com/resources/articles/data/4-important-aspects-of-data-science/ (cit. on pp. 133, 135).

[24] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 17: Treatment effect models, Section 17.6: Analysis of variance (ANOVA)*. Oct. 27, 2024. URL: https://rafalab.dfci.harvard.edu/dsbook-part-2/linear-models/treatment-effect-models.html#sec-anova (visited on 10/27/2025) (cit. on p. 286).