

Capstone Movielens Report

Azamat Kurbanayev

2025-06-07

Contents

1 Introduction / Overview / Executive Summary	3
1.1 Datasets Overview	3
1.1.1 edx Dataset	5
2 Methods / Analysis	9
2.1 Defining Logging and Time Measuring Helper Functions	9
2.2 Preparing train and test datasets	11
2.2.1 The make_source_datasets function	11
2.2.2 Common Helper Functions	20
2.3 Overall Mean Rating (OMR) Model	21
2.3.1 Mathematical Description of the OMR Model	21
2.3.2 OMR Model Building	21
2.3.3 OMR Value Is the Best for the Current Model	23
2.4 User Effect (UE) Model	25
2.4.1 User Effect Analysis	25
2.4.2 Mathematical Description of the UE Model	29
2.4.3 UE Model Building	29
2.5 User+Movie Effect (UME) Model	33
2.5.1 Movie Effect Analysis	33
2.5.2 Mathematical Description of the UME Model	39
2.5.3 UME Model: Support Functions	39
2.5.4 UME Model Building	42
2.5.5 UME Model Regularization	44
2.6 User+Movie+Genre Effect (UMGE) Model	52
2.6.1 Movie Genres Effect Analysis	52
2.6.2 Mathematical Description of the UMGE Model	54
2.6.3 UMGE Model: Support Functions	55

2.6.4	UMGE Model Building	59
2.6.5	UMGE Model Regularization	61
2.7	User+Movie+Genre+Year Effect (UMGYE) Model	69
2.7.1	Year Effect Analysis	69
2.7.2	Mathematical Description of the UMGYE Model	71
2.7.3	UMGYE Model: Support Functions	72
2.7.4	UMGYE Model Building	79
2.7.5	UMGYE Model Regularization	81
2.8	User+Movie+Genre+Year+Day Effect (UMGYDE) Model	89
2.8.1	UMGYDE Model: Helper Functions	90
2.8.2	UMGYDE Model Building With Default Parameters	104
2.8.3	UMGYDE Model Tuning by <code>degree</code> and <code>span</code> Parameters	107
2.8.4	UMGYDE Model Regularization	126
2.9	UMGYDE Model: Matrix Factorization (MF)	138
2.9.1	MF: Matematical Description	138
2.9.2	MF: Helper Functions	139
2.9.3	MF: Performing Operation	141
3	Conclusion	145
4	Appendix	147
4.1	Data Helper Functions	147
4.1.1	<code>union_cv_results</code> Function	147
4.1.2	<code>data.plot</code> Function	148
4.1.3	<code>data.plot.left.n</code> Function	149
4.1.4	<code>data.plot.left_detailed</code> Function	151
4.2	Common Helper Functions	155
4.2.1	Common Helper Functions: Result RMSEs Table	155
4.2.2	Common Helper Functions: Regularization	160

1 Introduction / Overview / 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 (RMSE)* score, which, as shown in [Section 23.2 Loss function](#) of the *Course Textbook*, 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* one 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 the following package has been developed by the author of this report: `edx.capstone.movieLens.data`. The source code of the package is available [on GitHub](#)[\[3\]](#).

Let's install the development version of this package from the GitHub repository and attach the correspondent library to the global environment:

```
if(!require(edx.capstone.movieLens.data)) pak::pak("AzKurban-edX-DS/edx.capstone.movieLens.data")

library(edx.capstone.movieLens.data)
edx <- edx.capstone.movieLens.data::edx
final_holdout_test <- edx.capstone.movieLens.data::final_holdout_test
```

Now, 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:900000
##  1st Qu.:18124   1st Qu.:  648   1st Qu.:3.000   1st Qu.:9.468e+08   Class  :character   Class  :charact
##  Median :35738   Median : 1834   Median :4.000   Median :1.035e+09   Mode   :character   Mode   :charact
##  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
##  1st Qu.:18096  1st Qu.:  648  1st Qu.:3.000   1st Qu.:9.467e+08   Class  :character
##  Median :35768  Median : 1827  Median :4.000   Median :1.035e+09   Mode   :character
##  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 838984885 838984885 ...  
## $ title     : chr "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...  
## $ genres    : chr "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|Drama|Sci-Fi|Thriller" "Action|Drama|Sci-Fi|Thriller" ...
```

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 23.1.1 Movielens data](#) of the *Course Textbook*[4] that not every user rated every movie:

```
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 that section, 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 NAs 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[5]:

```
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")

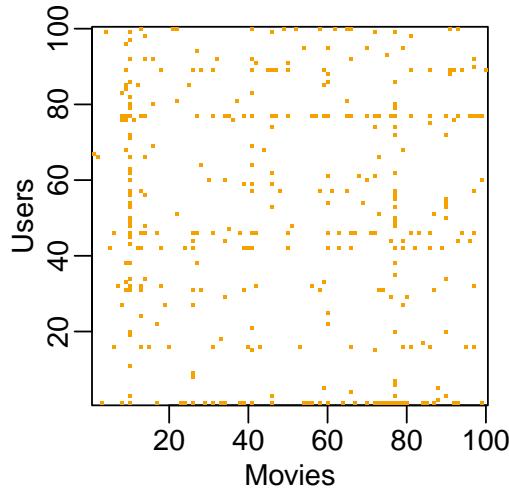
print(tab)

## # A tibble: 5 x 5
##   userId `Pulp Fiction (1994)` `Jurassic Park (1993)` `Silence of the Lambs (1991)` `Forrest Gump (1994)`
##   <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:

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

rafalib::mpar()
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")
```

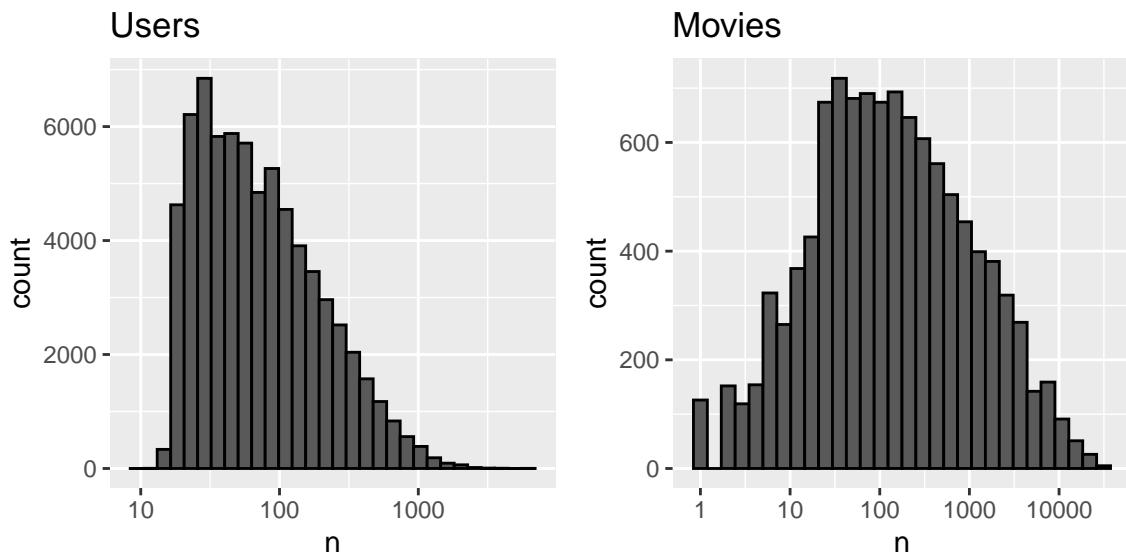


Further observations highlighted there that, as we can see from the distributions the author presented, some movies get rated more than others, and some users are more active than others in rating movies:

```
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, we can see 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 / Analysis



All the source code of the R-scripts is available on the project's [GitHub repository](#)[6].

2.1 Defining Logging and Time Measuring Helper Functions

First, let's define some helper functions for logging and time-measuring features that we will use in our R scripts. Some of them are listed below:

```
# Logging Helper functions -----
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)
}

print_start_date <- function(){
  print(date())
  Sys.time()
}

put_start_date <- function(){
  put(date())
  Sys.time()
}

print_end_date <- function(start){
  print(date())
  print(Sys.time() - start)
}

put_end_date <- function(start){
  put(date())
  put(Sys.time() - start)
}

msg.set_arg <- function(msg_template, arg, arg.name = "%1") {
  msg_template |>
    str_replace_all(arg.name, as.character(arg))
}

msg.glue <- function(msg_template, arg, arg.name = "%1"){
  msg_template |>
    msg.set_arg(arg, arg.name) |>
    str_glue()
}

print_log <- function(msg){
  print(str_glue(msg))
}
```

```

put_log <- function(msg){
  put(str_glue(msg))
}

get_log1 <- function(msg_template, arg1) {
  str_glue(str_replace_all(msg_template, "%1", as.character(arg1)))
}
print_log1 <- function(msg_template, arg1){
  print(get_log1(msg_template, arg1))
}
put_log1 <- function(msg_template, arg1){
  put(get_log1(msg_template, arg1))
}

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()
}
print_log2 <- function(msg_template, arg1, arg2){
  print(get_log1(msg_template, arg1, arg2))
}
put_log2 <- function(msg_template, arg1, arg2){
  put(get_log1(msg_template, arg1, arg2))
}

# ...

```



The full source code of these functions is available in the [Logging Helper functions](#) section of the [capstone-movielens.main.R](#) script on *GitHub*.

2.2 Preparing train and test 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 23.1.1 MovieLens data](#) of the *Course Textbook* mentioned above[5]. We will also use the *5-Fold Cross Validation* method as described in [Section 29.6 Cross validation](#) of the *Course Textbook*. To prepare datasets for processing, we will use the following functions, specifically designed for these operations:

- `make_source_datasets`
- `init_source_datasets`



The full source code of the function listed above is available in the [Initialize input datasets](#) section of the `data.helper.functions.R` script on [GitHub](#).

2.2.1 The `make_source_datasets` function

Let's take a closer look at the objects we will receive as a result of executing this function.

```
make_source_datasets <- function(){  
  # ...  
  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)  
}
```

2.2.1.1 `edx.mx` Matrix Object

We will use the array representation described in [Section 17.5 of the Textbook](#), for the training data: we denote ranking for movie j by user i as $y_{i,j}$. To create this matrix, we use `tidyverse::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 dimensions:")
```

```

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" ...

```

2.2.1.2 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)

```

```
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 (1995" ...
## $ 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:23371423
## 1st Qu.:	18140	1st Qu.: 616	1st Qu.: 3.000	1st Qu.: 9.472e+08	Class :character	Class :character
## Median :	35784	Median : 1748	Median : 4.000	Median : 1.042e+09	Mode :character	Mode :character
## 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		

Note that we use the `splitGenreRows` function to split rows of the original dataset:

```

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 function mentioned above is also available in the [Initialize input datasets](#) section of the `data.helper.functions.R` script on *GitHub*.

2.2.1.3 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")

  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 can't use them as IDs[5].

2.2.1.4 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")

  str(date_days_map)

## 'data.frame': 6519590 obs. of 5 variables:
## $ timestamp: int 838985046 838983525 838983421 838983392 838984474 838983653 838984885 838983707 838983708 ...
```

```

## $ date_time: POSIXct, format: "1996-08-02 11:24:06" "1996-08-02 10:58:45" "1996-08-02 10:57:01" "1996-08-02 10:57:01" ...
## $ 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  1995  Min.   :
## 1st Qu.:9.783e+08 1st Qu.:2001-01-01 05:05:01.75 1st Qu.:2001 2001  1st Qu.:2001
## Median :1.091e+09 Median :2004-08-03 01:08:18.50 Median :2004 2004  Median :2004
## Mean   :1.066e+09 Mean   :2003-10-10 23:15:02.07 Mean   :2003 2003  Mean   :311
## 3rd Qu.:1.152e+09 3rd Qu.:2006-07-04 20:41:57.50 3rd Qu.:2006 2006  3rd Qu.:2006
## Max.   :1.231e+09 Max.   :2009-01-05 05:02:16.00 Max.   :2009 2009  Max.   :511

```

2.2.1.5 edx_CV Object

Here we have a list of sample objects we need to perform the *5-Fold Cross Validation* as explained in Section 29.6.1 K-fold cross validation of the *Course Textbook*:

```

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://rafaelab.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://rafaelab.dfci.harvard.edu/dsbook-part-2/linear-models/treatment-effect-models.html#sec-a)
  #> for the training data.
}

```

```

#> To create this matrix, we use `tidyverse::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 dimensions:")
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")

```

```
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 ...
##     ...$ 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 ...
##     ...$ timestamp   : int [1:7172311] 838985046 838983525 838983421 838983392 838983653 838984885 838984885 ...
##     ...$ title       : chr [1:7172311] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Star Trek: Generations (1994)" ...
##     ...$ genres      : chr [1:7172311] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" ...
##   ..$ train_mx      : num [1:69878, 1:10677] 5 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 ...
##     ...$ 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 ...
##     ...$ timestamp   : int [1:18669190] 838985046 838985046 838983525 838983525 838983421 838983421 838983421 ...
##     ...$ title       : chr [1:18669190] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, The (1995)" ...
##     ...$ 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 868245920 ...
##     ...$ title       : chr [1:1827744] "Stargate (1994)" "Flintstones, The (1994)" "Speed (1994)" "Aladdin (1992)" ...
##     ...$ genres      : chr [1:1827744] "Action|Adventure|Sci-Fi" "Children|Comedy|Fantasy" "Action|Romantic" ...

```

```

## $ :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 ...
## ...$ 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 838984474 838984474 838984474 838984474 ...
## ...$ 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|Th ...
## ..$ 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 ...
## ...$ 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 838983525 838983421 838983421 838983421 838983421 ...
## ...$ 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 868245608 868245608 868245608 ...
## ...$ title        : chr [1:1827749] "Jungle Book, The (1994)" "Robin Hood: Men in Tights (1993)" "Sl ...
## ...$ genres       : chr [1:1827749] "Adventure|Children|Romance" "Comedy" "Comedy|Drama|Romance" "An ...
## $ :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 ...
## ...$ 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 838984474 838984474 838984474 838984474 ...
## ...$ 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|Th ...
## ..$ train_mx     : num [1:69878, 1:10677] 5 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 ...
## ...$ 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 838983525 838983421 838983421 838983421 838983421 ...
## ...$ 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 3 3 3 4.5 5 ...
## ...$ timestamp    : int [1:1827748] 838983653 838983707 838984068 838984941 868245608 868246191 868246191 868246191 868246191 ...
## ...$ 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|M ...
## $ :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 838983653 838983653 838983653 838983653 ...
## ...$ title     : chr [1:7172311] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
## ...$ genres    : chr [1:7172311] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" ...
## ...$ 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 838983653 838983653 838983653 838983653 ...
## ...$ title     : chr [1:18669192] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, The (1995)" ...
## ...$ 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 868245920 868245920 868245920 868245920 ...
## ... $. title    : chr [1:1827744] "Speed (1994)" "Robin Hood: Men in Tights (1993)" "Aladdin (1992)" "The Lion King (1994)" ...
## ... $. 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 838983653 838983653 838983653 838983653 ...
## ... $. title    : chr [1:7172301] "Boomerang (1992)" "Net, The (1995)" "Outbreak (1995)" "Stargate (1994)" ...
## ... $. genres   : chr [1:7172301] "Comedy|Romance" "Action|Crime|Thriller" "Action|Drama|Sci-Fi|Thriller" ...
## ... $. train_mx  : num [1:69878, 1:10677] 5 NA NA NA NA NA NA NA NA ...
## ...- attr(*, "dimnames")=List of 2
## ... .$. : 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 838983653 838983653 838983653 838983653 ...
## ...$ title     : chr [1:18669194] "Boomerang (1992)" "Boomerang (1992)" "Net, The (1995)" "Net, The (1995)" ...
## ...$ 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 868245920 868245920 868245920 868245920 ...
## ... $. title    : chr [1:1827754] "Star Trek: Generations (1994)" "Jungle Book, The (1994)" "Speed (1994)" ...
## ... $. genres   : chr [1:1827754] "Action|Adventure|Drama|Sci-Fi" "Adventure|Children|Romance" "Action|Romance" ...

```



This code snippet is a part of the `make_source_datasets` function code described above.

Note that we used the `sample_train_validation_sets` function call to split the original dataset (`edx` in this case):

```
split_sets <- edx |>
  sample_train_validation_sets(fold_i*1000)
```

which returns a pair of train/validation sets:

```
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`: Dataset created: train_set")
  put(summary(train_set))

  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)

  put_log("Function: `sample_train_validation_sets`: Dataset created: validation_set")
  put(summary(validation_set))

  # CV train & test sets Consistency Test
  validation.left_join.Nas <- train_set |>
    mutate(tst.col = rating) |>
    select(userId, movieId, tst.col) |>
    data.consistency.test(validation_set)

  put_log("Function: `sample_train_validation_sets`:
Below are the data consistency verification results")
  put(validation.left_join.Nas)

  # Return result datasets -----
```

```
    list(train_set = train_set,
         validation_set = validation_set)
}
```



The `sample_train_validation_sets` function is defined in the same script as the `make_source_datasets`⁴ one, from where it is called.

2.2.2 Common Helper Functions

For our further analysis, we are going to use the following *common helper functions*:

2.2.2.1 clamp function

As explained in [Section 24.4 User effects](#) of the *Course Textbook* we know ratings can't be below 0.5 or above 5. For this reason, we will use the `clamp` function described in that section:

```
clamp <- function(x, min = 0.5, max = 5) pmax(pmin(x, max), min)
```

2.2.2.2 Functions to calculate (Root) Mean Squared Error

We will need the following functions to calculate (R)MSEs:

```
mse <- function(r) mean(r^2)

mse_cv <- function(r_list) {
  mses <- sapply(r_list, mse(r))
  mean(mses)
}

rmse <- function(r) sqrt(mse(r))
# rmse_cv <- function(r_list) sqrt(mse_cv(r_list))

rmse2 <- function(true_ratings, predicted_ratings) {
  rmse(true_ratings - predicted_ratings)
}
```



All the *common helper functions*, including those described above, are defined in the [common-helper.functions.R](#) script on *GitHub*.

2.3 Overall Mean Rating (OMR) Model

Let's begin our analysis by evaluating the simplest model described in Section 23.3 *The First Model of the Course Textbook*, and then gradually refine it through further research.

2.3.1 Mathematical Description of the OMR Model

It is about a 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 μ the *true* rating for all movies.

2.3.2 OMR Model Building

We know that the estimate that minimizes the RMSE is the least squares estimate of μ and, in this case, is the average of all ratings:

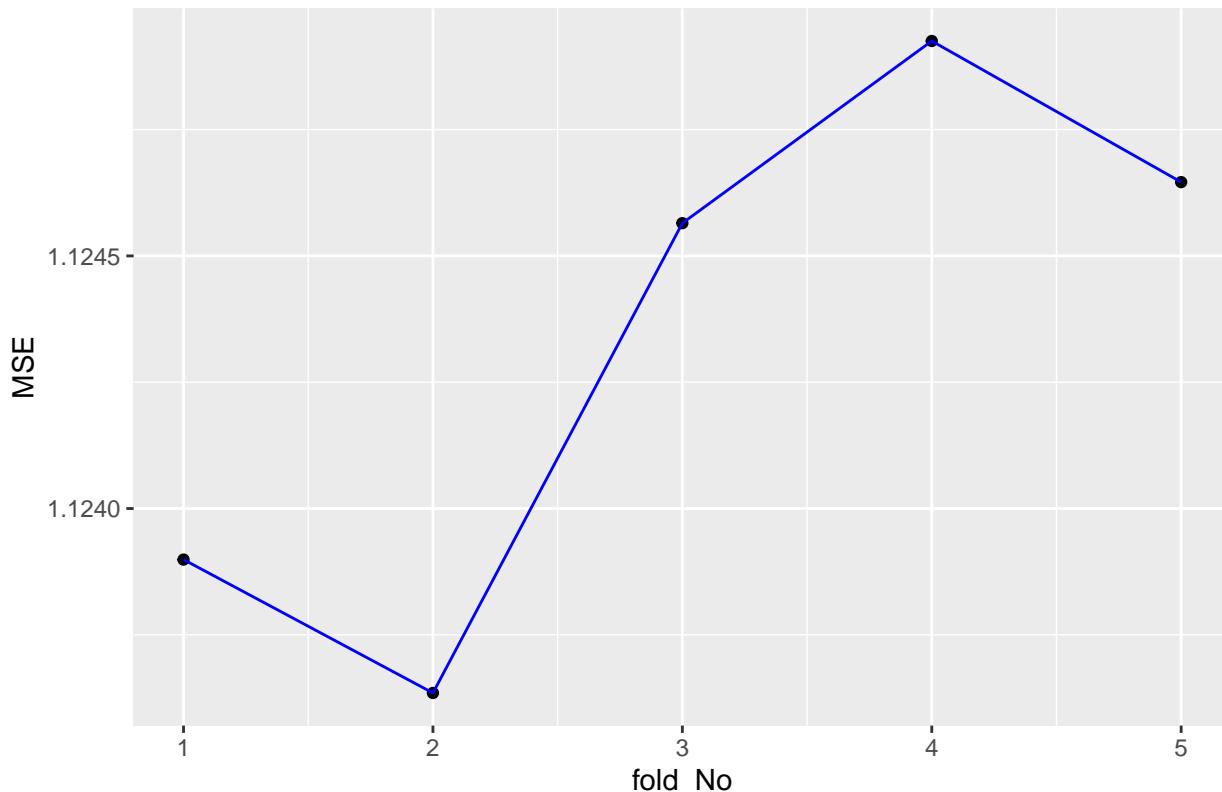
```
mu <- mean(edx$rating)
print(mu)
```

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

If we predict all unknown ratings with $\hat{\mu}$, we obtain the following RMSE:

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

MSE results of the 5-fold CV method applied to the Overall Mean Rating



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

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



For the *Mean Squared Errors* data visualization we use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

Here we also used `naive_model_MSEs` function defined in the `common-helper.functions.R` script (already mentioned above) to compute *Mean Squared Errors* using *5-Fold Cross Validation* method:

```
naive_model_MSEs <- function(val) {
  sapply(edx_CV, function(cv_item){
    mse(cv_item$validation_set$rating - val)
  })
}
```

One more function, defined in the `same script`, that we will need for further analysis of the current model, is the `naive_model_RMSE` one:

```
naive_model_RMSE <- function(val){
  sqrt(mean(naive_model_MSEs(val)))
}
```

2.3.3 OMR Value Is the Best for the Current Model

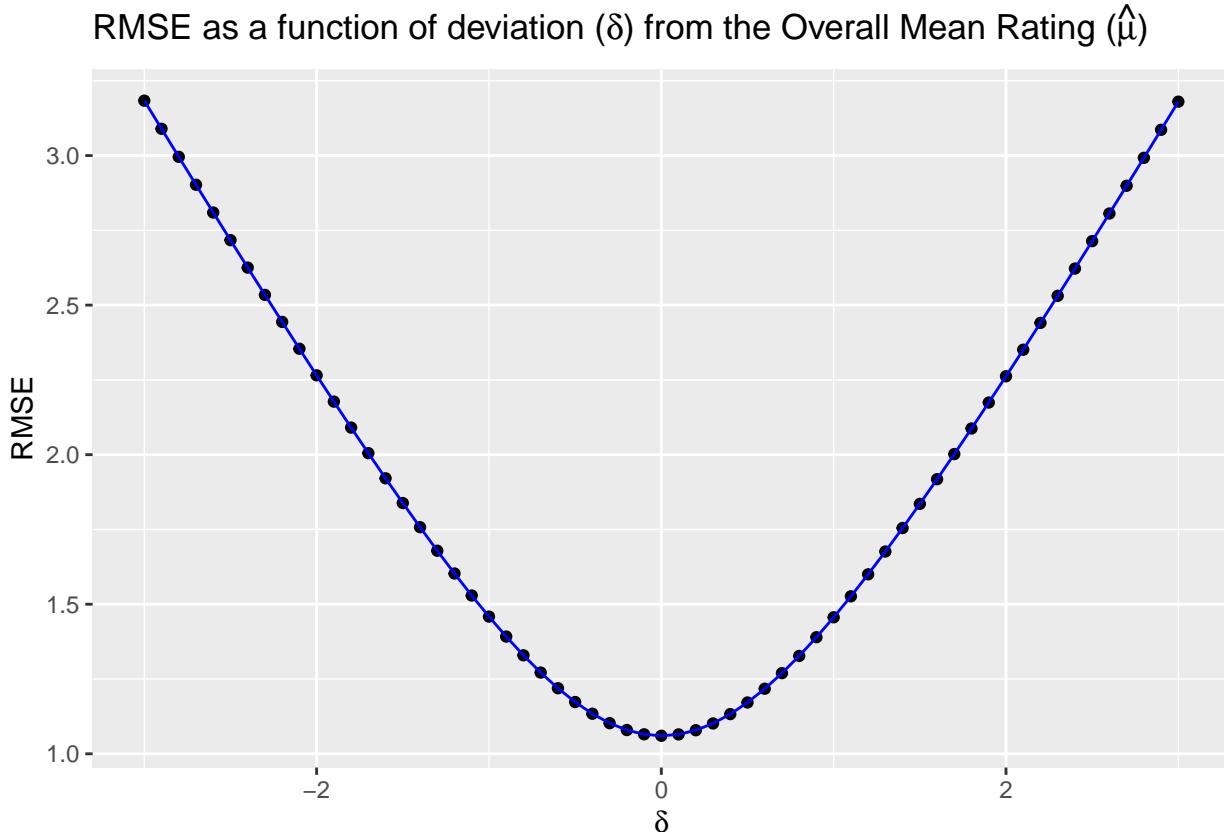
If we plug in any other number, we will get a higher RMSE. Let's prove that by the following small investigation:

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

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

Let's make a quick investigation of the `deviation.RMSE` result we have just got:

```
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 ($\hat{\mu}$)]'),
            xname = "delta",
            yname = "delta.RMSE",
            xlabel = TeX(r'[$\delta$]'),
            ylabel = "RMSE")
```



```
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

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

RMSE_kable(RMSEs.ResultTibble.OMR)

```

Method	RMSE	Comment
Project Objective	0.864900	
Overall Mean Rating Model	1.060346	

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![7]

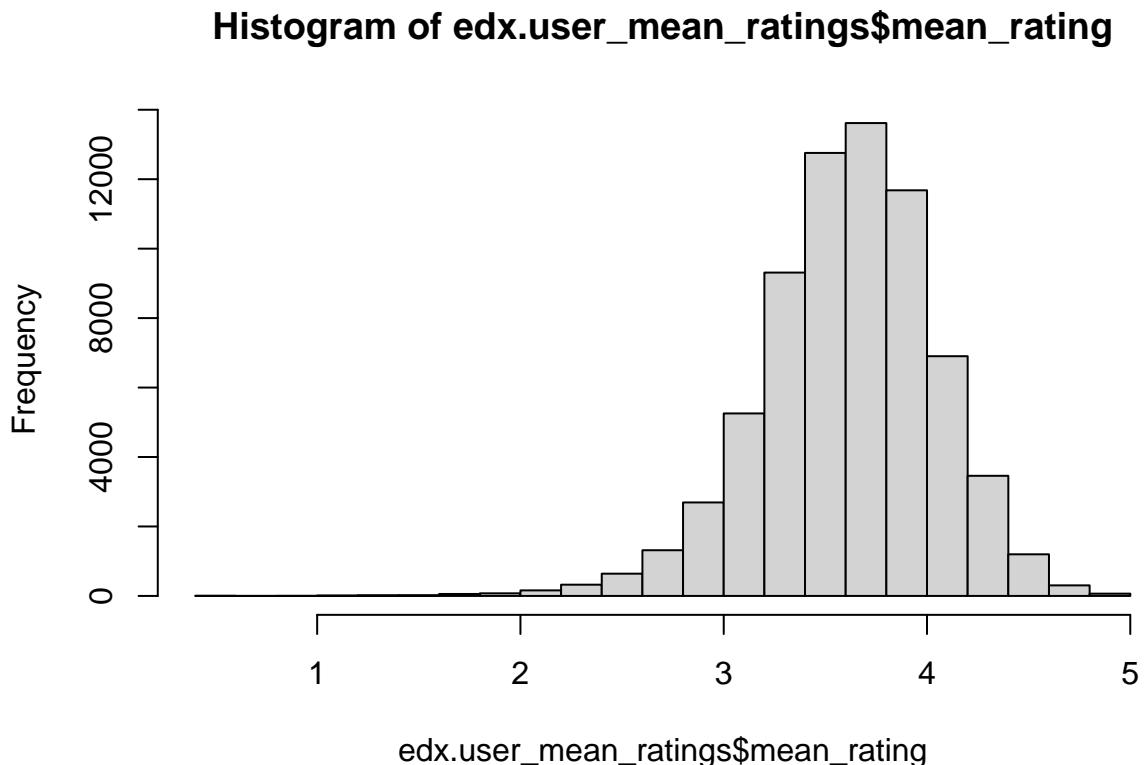
2.4 User Effect (UE) Model

2.4.1 User Effect Analysis

To improve our model let's now take into consideration user effects as explained in [Section 23.4 User effects](#) of the *Course Textbook*[8].

If we visualize the average rating for each user the way the [the author](#) shows, we can see that there is substantial variability in the average ratings across users:

```
hist(edx.user_mean_ratings$mean_rating, nclass = 30)
```



2.4.1.1 Ratings per user

eponymous homonymous



The code in this section is cited from [the eponymous section](#) of the article *Movie Recommendation System using R - BEST*[9].

2.4.1.1.1 User rating count (activity measure)

```
print(edx |>
  group_by(userId) |>
  summarize(count = n()) |>
  slice_head(n = 10)
)

## # A tibble: 10 x 2
##   userId count
##     <int> <int>
## 1     1     19
## 2     2     17
## 3     3     31
## 4     4     35
## 5     5     74
## 6     6     39
## 7     7     96
## 8     8    727
## 9     9     21
## 10    10    112
```

2.4.1.1.2 User rating summary

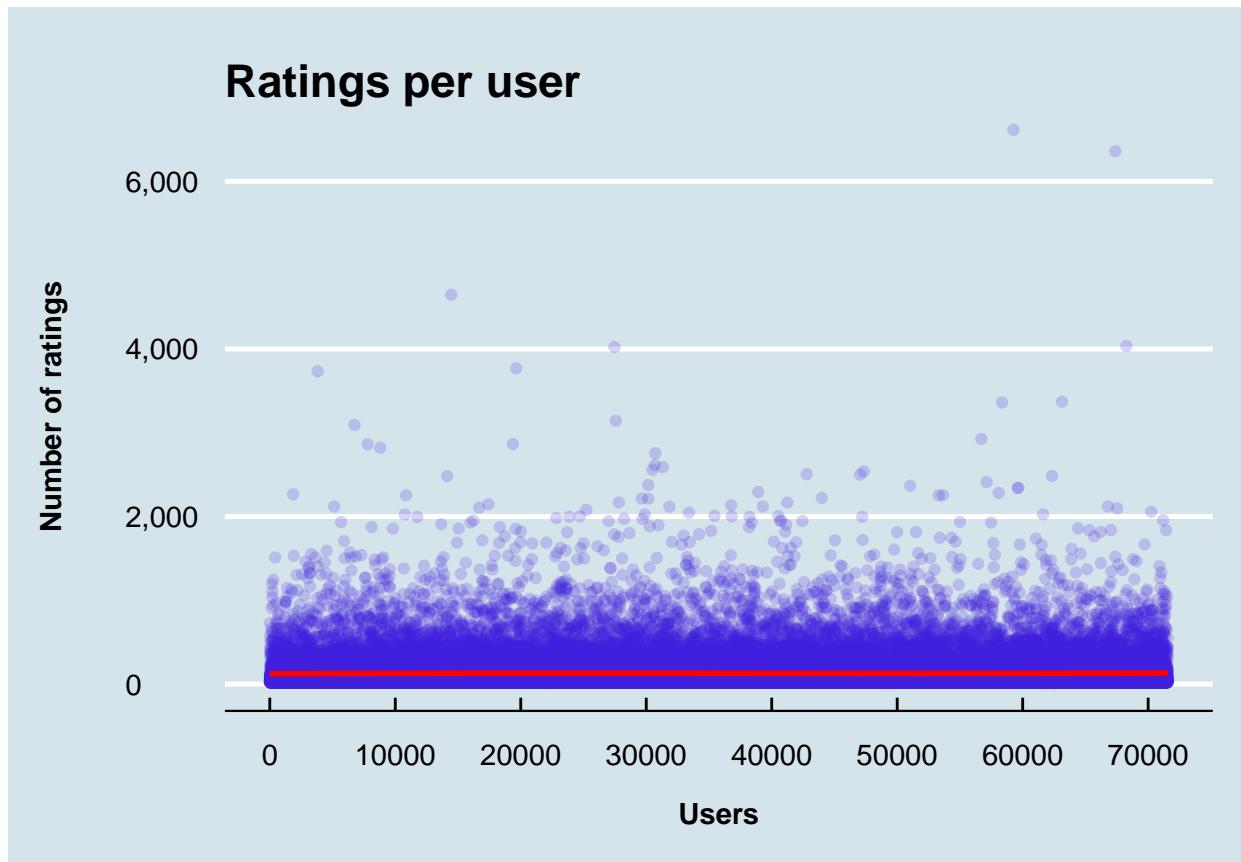
```
summary(edx |> group_by(userId) |> summarize(count = n()) |> select(count))

##      count
##  Min.   : 10.0
##  1st Qu.: 32.0
##  Median : 62.0
##  Mean   : 128.8
##  3rd Qu.: 141.0
##  Max.   :6616.0
```

2.4.1.1.3 Ratings per user plot

```
edx |>
  group_by(userId) |>
  summarize(count = n()) |>
  ggplot(aes(x = userId, y = count)) +
  geom_point(alpha = 0.2, color = "#4020dd") +
  geom_smooth(color = "red") +
  ggtitle("Ratings per user") +
  xlab("Users") +
  ylab("Number of ratings") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  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"))
```

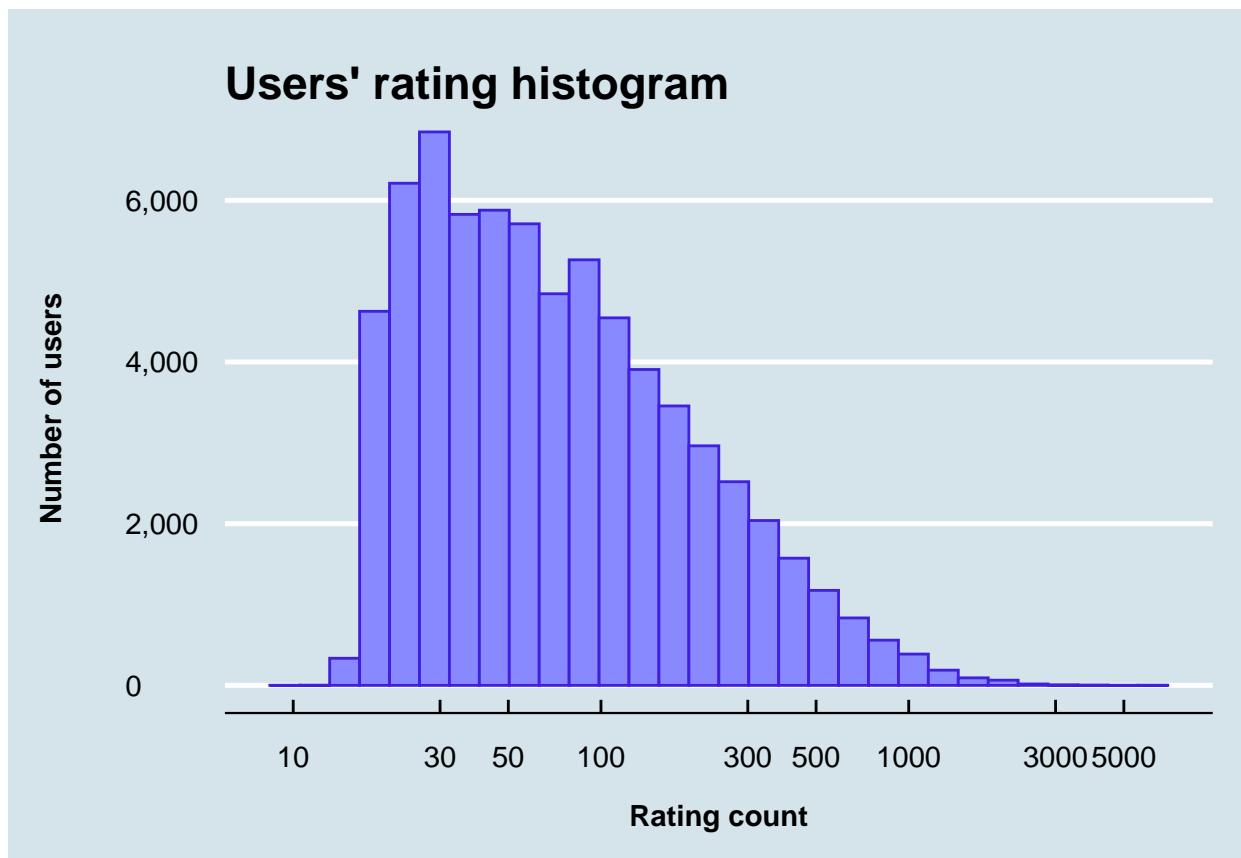
```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



2.4.1.1.4 Users' rating histogram

```
edx |>
  group_by(userId) |>
  summarize(count = n()) |>
  ggplot(aes(x = count)) +
  geom_histogram(fill = "#8888ff", color = "#4020dd") +
  ggtitle("Users' rating histogram") +
  xlab("Rating count") +
  ylab("Number of users") +
  scale_y_continuous(labels = comma) +
  scale_x_log10(n.breaks = 10) +
  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"))
```

‘stat_bin()’ using ‘bins = 30’. Pick better value with ‘binwidth’.



2.4.2 Mathematical Description of the UE Model

Following the author's further explanation, to account for this variability, we will use a linear model with a *treatment effect* α_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 α s as treatment effects. In the Netflix challenge papers, they refer to them as *bias*[8, 10].

2.4.3 UE Model Building

As it is stated here[8], 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:

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

These considerations allows us to compute a *User Mean Ratings* the following way:

```
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)
```

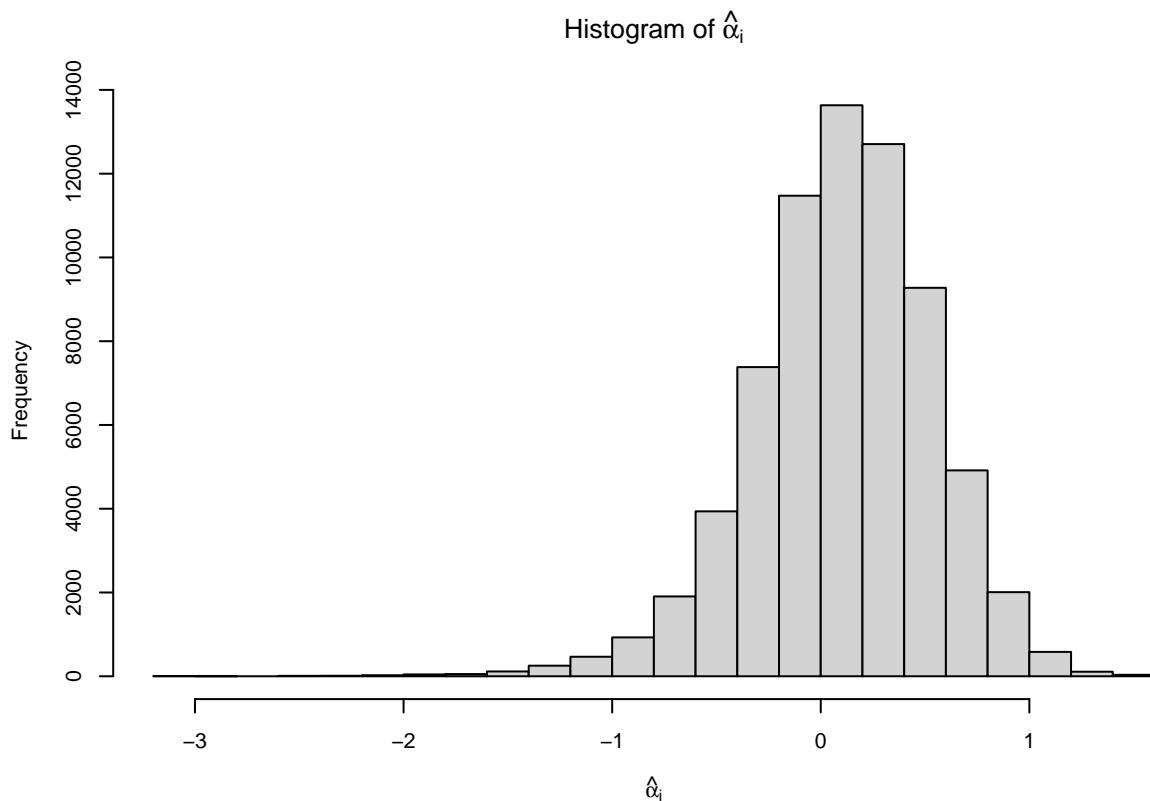
```
## 'data.frame':    69878 obs. of  3 variables:
## $ userId      : chr  "1" "2" "3" "4" ...
## $ mean_rating: num  5 3.29 3.94 4.06 3.92 ...
## $ n           : num  19 17 31 35 74 39 96 727 21 112 ...
```

And then we compute a *User Effect* this way:

```
put_log("Computing User Effect per users ...")
edx.user_effect <- edx.user_mean_ratings |>
  mutate(userId = as.integer(userId),
         a = mean_rating - mu)

put_log("A User Effect Model has been builded")
```

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



```
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 full source code of the *User Effect* computation is available in the [Model building: User Effect](#) section of the [capstone-movielens.main.R](#) script on *GitHub*.

Now, we are ready to compute the *Mean Squared Errors* from the *5-Fold Cross Validation* (additionally using the `clamp` helper function we described above):

```
put_log("Computing the RMSE taking into account user effects...")
start <- put_start_date()
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)) |>
```

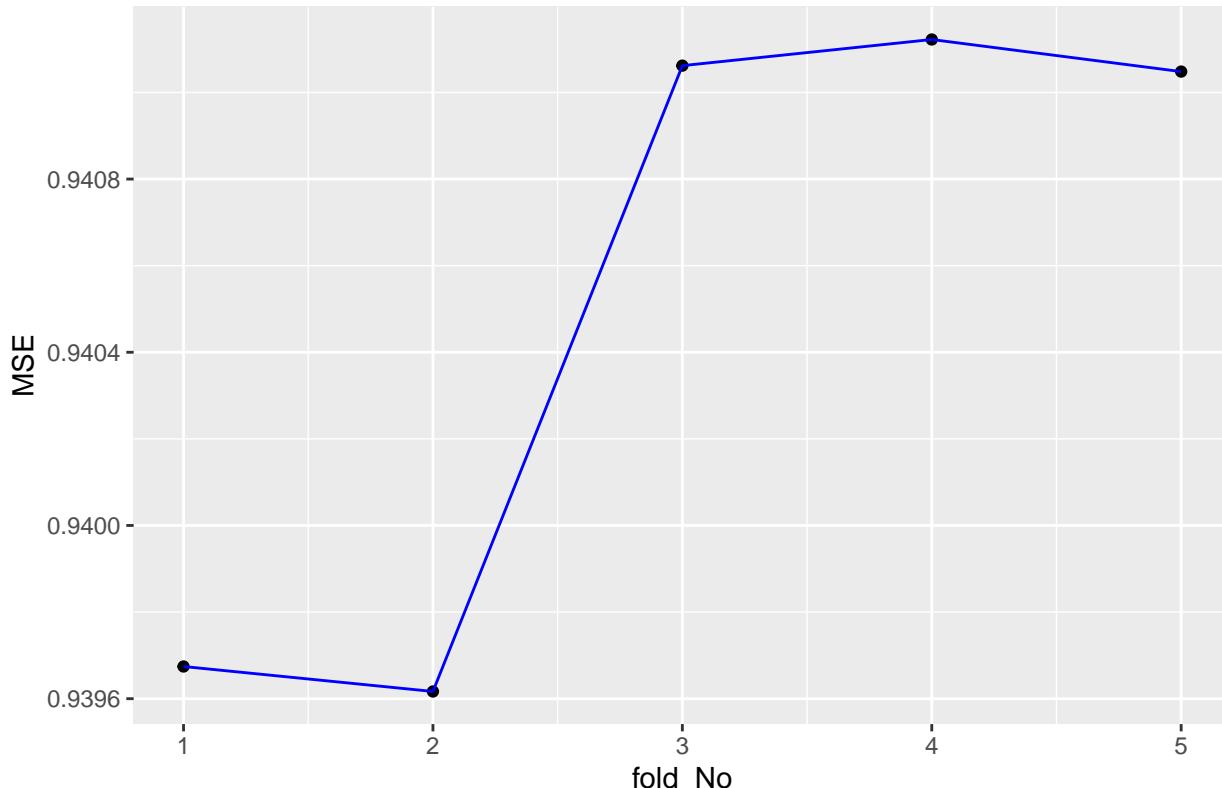
```

    pull(resid) |> mse()
})
put_end_date(start)

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

```

MSE results of the 5-fold CV method applied to the User Effect Model



For the *Mean Squared Errors* data visualization we use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

Finally, we calculate the RMSE as the *square root* of the average of the *Mean Squared Errors* we obtained through the *5-Fold Cross Validation* above:

```

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

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

RMSE_kable(RMSEs.ResultTibble.UE)

```

Method	RMSE	Comment
Project Objective	0.8649000	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	



The full source code of the *User Effect Model RMSE* computation is available in the [Compute RMSE for User Effect Model](#) section of the [capstone-movielens.main.R](#) script on *GitHub*.

2.5 User+Movie Effect (UME) Model

2.5.1 Movie Effect Analysis

2.5.1.1 Movies' Popularity

In 23.5 Movie effects section of the *Course Textbook* the author draws our attention to the fact that some movies are generally rated higher than others.

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

```
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:

```
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     4.5 791624
## 5     2 711422
## 6     4.5 526736
## 7     1 345679
## 8     2.5 333010
## 9     1.5 106426
## 10    0.5  85374
```

2.5.1.2 Rating Distribution

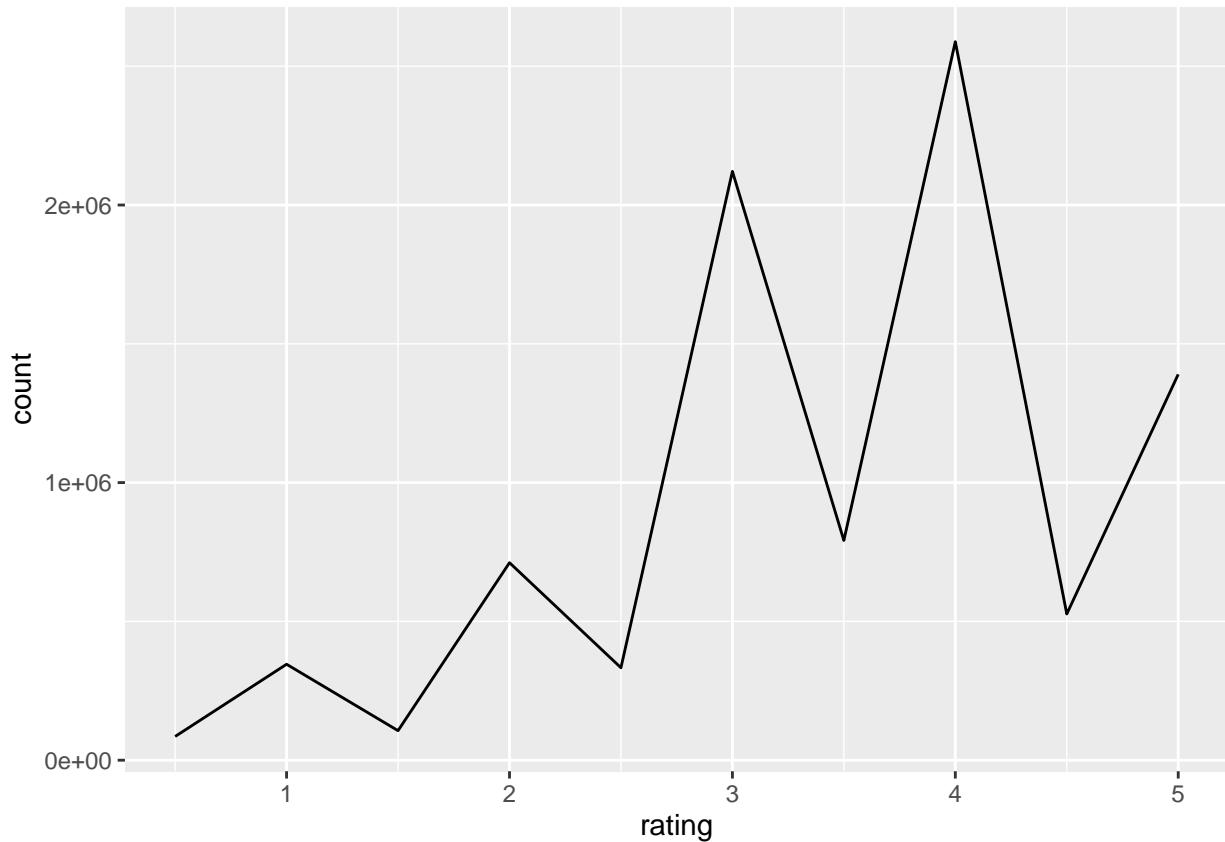
The following code allows us to summarize that in general, half-star ratings are less common than whole-star ratings (e.g., there are fewer ratings of 3.5 than there are ratings of 3 or 4, etc.):

```
print(edx.rating_groups |> arrange(rating))

## # 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
```

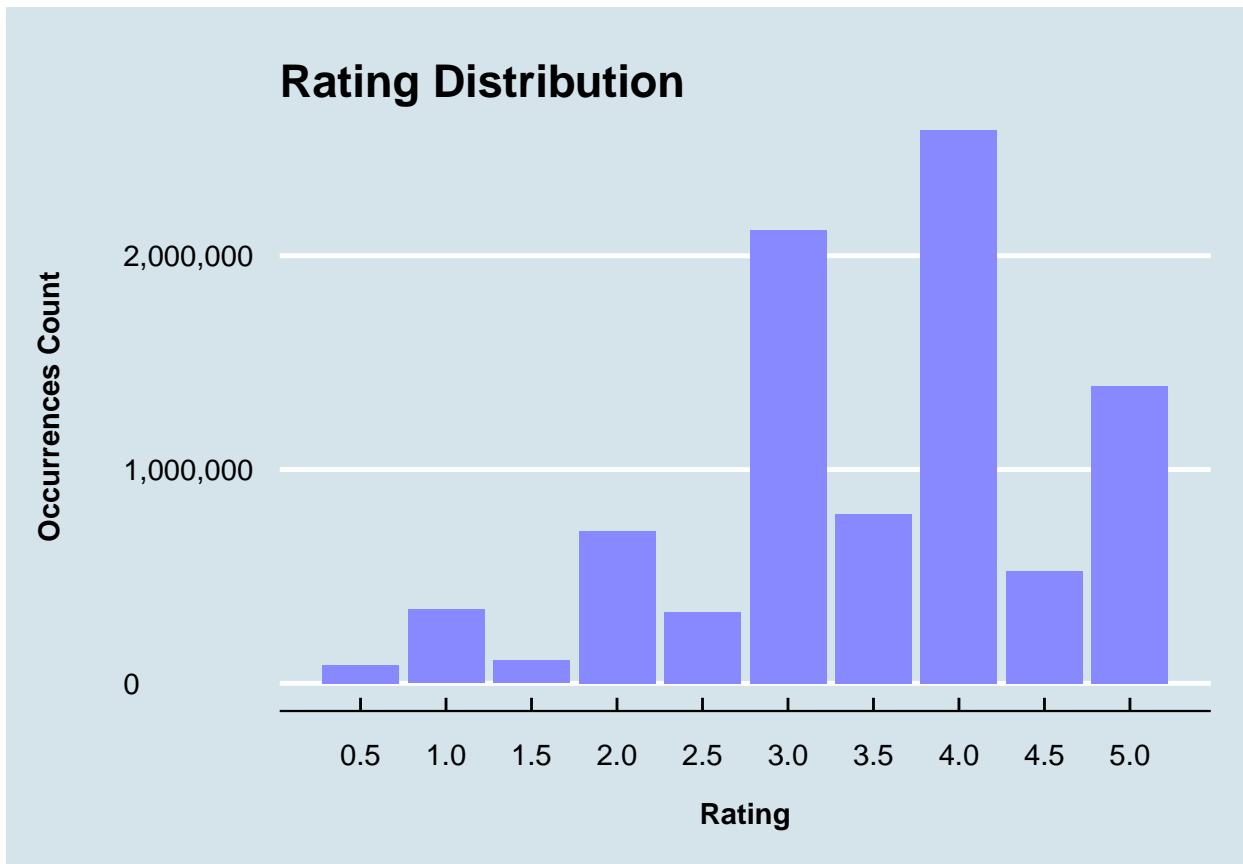
We can visually see that from the following plot:

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



The more sophisticated way of visualizing the rating distribution shown below is cited from the [Rating distribution plot](#) section of the article *Movie Recommendation System using R - BEST[9]* already referenced above in the [Ratings per user](#) section of this report.

```
edx.rating_groups |>
  ggplot(aes(x = rating, y = count)) +
  geom_bar(stat = "identity", fill = "#8888ff") +
  ggtitle("Rating Distribution") +
  xlab("Rating") +
  ylab("Occurrences Count") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  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"))
```



This graph is another confirmation of what we found out above: rounded ratings occur more often than half-starred ones. The upward trend previously discussed is now perfectly clear, although it seems to top right between the 3 and 4-star ratings lowering the occurrences count afterward. That might be due to users being more hesitant to rate with the highest mark for whichever reasons they might hold[9].

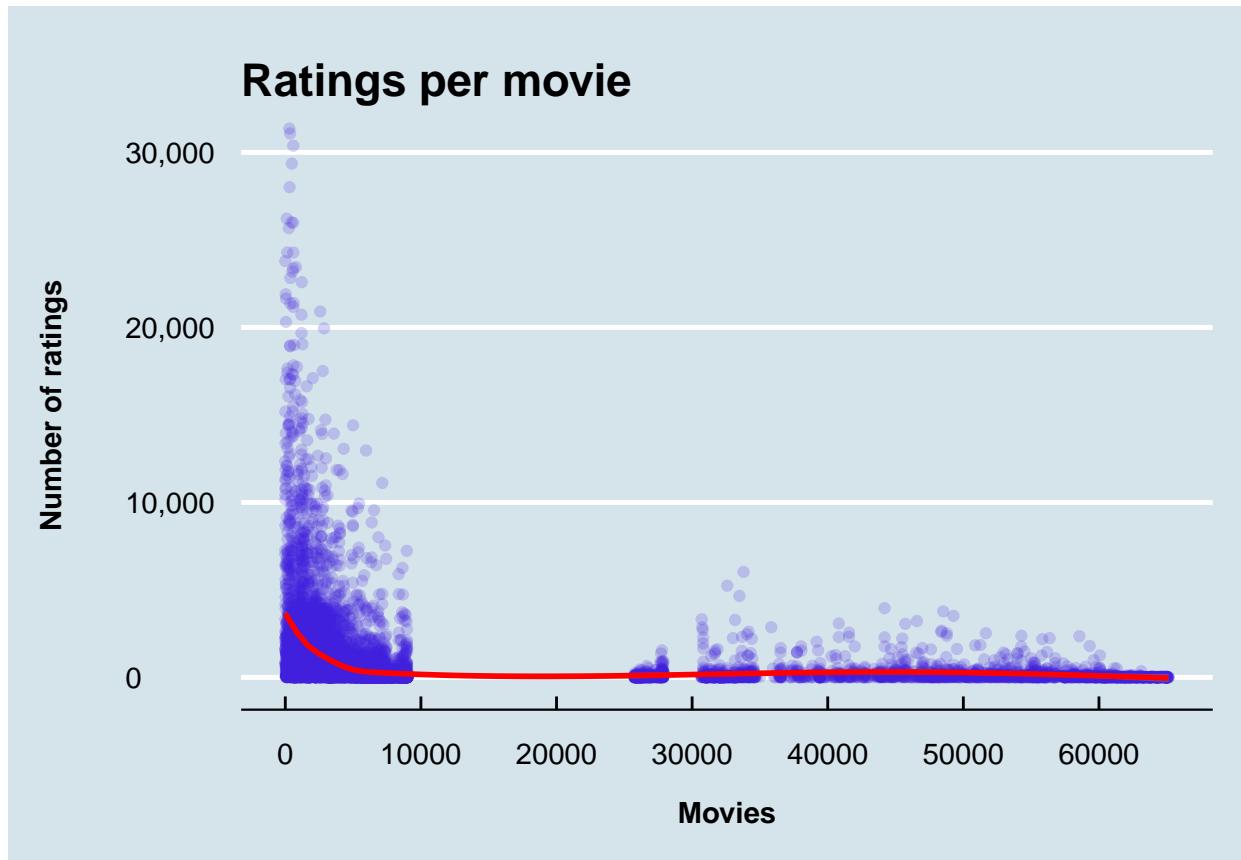
The two more plots below, from the same article, give us an additional visual representation of the movies' popularity.

2.5.1.2.1 Ratings per movie plot

```
edx.movie_groups <- edx |>
  group_by(movieId) |>
  summarize(count = n())

edx.movie_groups |>
  ggplot(aes(x = movieId, y = count)) +
  geom_point(alpha = 0.2, color = "#4020dd") +
  geom_smooth(color = "red") +
  ggtitle("Ratings per movie") +
  xlab("Movies") +
  ylab("Number of ratings") +
  scale_y_continuous(labels = comma) +
  scale_x_continuous(n.breaks = 10) +
  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"))

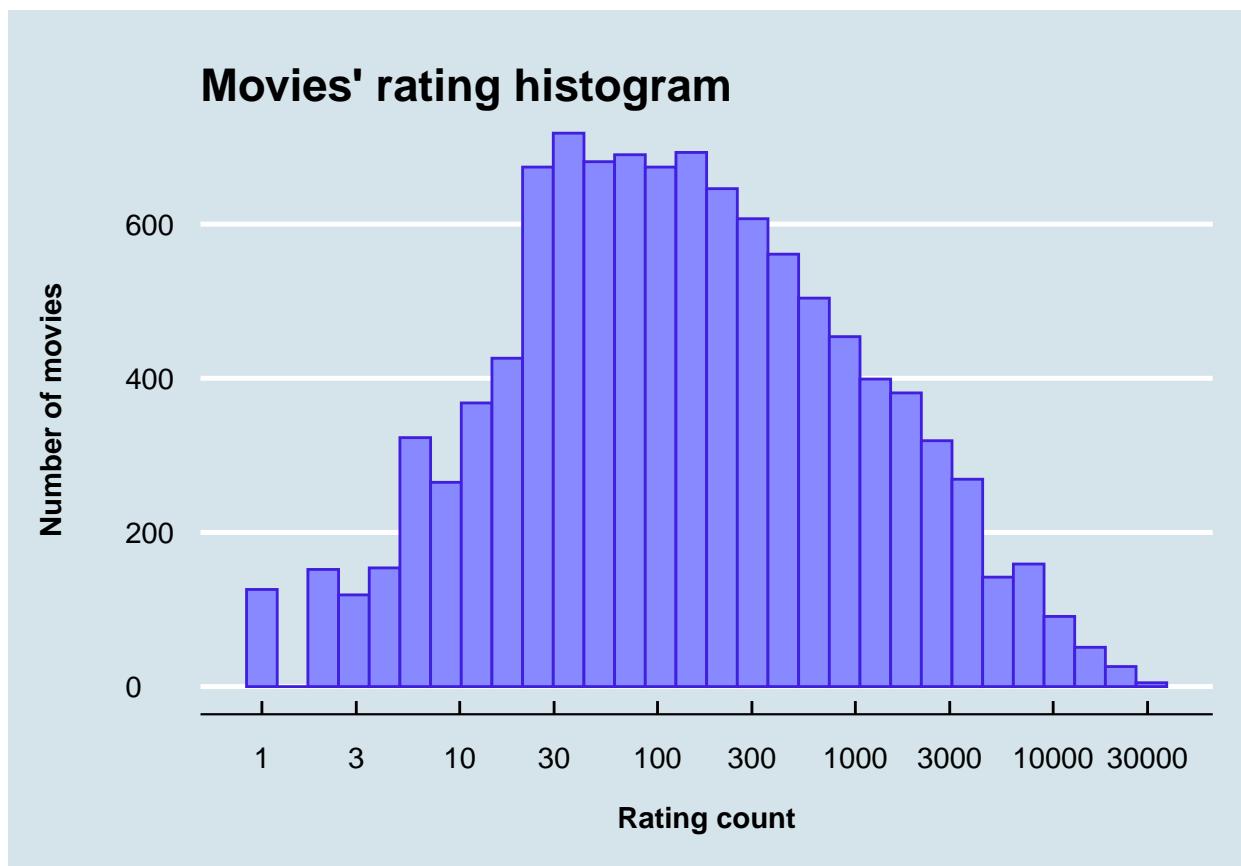
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



2.5.1.2.2 Movies' rating histogram

```
edx.movie_groups |>
  ggplot(aes(x = count)) +
  geom_histogram(fill = "#8888ff", color = "#4020dd") +
  ggtitle("Movies' rating histogram") +
  xlab("Rating count") +
  ylab("Number of movies") +
  scale_y_continuous(labels = comma) +
  scale_x_log10(n.breaks = 10) +
  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"))
```

'stat_bin()' using 'bins = 30'. Pick better value with 'binwidth'.



This histogram better conveys the information provided by the summary() function, where the quantiles' values state that half the movies are rated between 30 and 565 times[9].

2.5.2 Mathematical Description of the UME Model

The author of the *Course Textbook* mentioned above also explains that in this case one can use a linear model with a *treatment effect* β_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 μ :

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

The author then shows 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 $y_{i,j} - \hat{\mu} - \hat{\alpha}_i$:

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

Inspired by this idea, a few support functions were developed by the author of this report, which we will use for our further analysis.

2.5.3 UME Model: Support Functions



The complete source code of the functions described in this section is available in the [UME Model Support Functions](#) section of the [UM-effect.functions.R](#) script on *GitHub*.

2.5.3.1 train_user_movie_effect Function

We use this function to build and train our model using the `train_set` dataset:

```
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 function described above accepts the `lambda` parameter, which we will need later for the *Model Regularization* method. We also use the `mean_reg` function call, which we will need later for the *Regularization* techniques (for details, see the `mean_reg` function description in the Section [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report). We will explain that later in the [User+Movie Effect Model Regularization] section. For now, we omit the `lambda` parameter, accepting its default value `lambda = 0`. In this case, the `mean_reg` function is equivalent to the standard R function `base::mean`.

2.5.3.2 `train_user_movie_effect.cv` Function

We use the `train_user_movie_effect.cv` function to build and train our model using the *5-Fold Cross Validation* method. Below, we provide the most important part of the code of that function:

```
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  
}
```



Here we use the function call `union_cv_results`, which is defined in the script [common-helper.functions.R](#), to aggregate the *5-Fold Cross Validation* method results (for details, see the `union_cv_results` function description in the [Data Helper Functions](#) section of the [Appendix](#) to this report).

2.5.3.3 `calc_user_movie_effect_MSE` Function

The source code of the function `calc_user_movie_effect_MSE` defined in the `UME-effect.functions.R` script to calculate the *Mean Squared Error (MSE)* of the *UME Model* for the given *Test Set* is provided 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
}

```

2.5.3.4 calc_user_movie_effect_MSE.cv Function

The source code of the function `calc_user_movie_effect_MSE.cv` defined in the `UM-effect.functions.R` script to calculate the *5-Fold Cross Validation MSE* result of the *UME Model* is provided 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)
}

```

2.5.3.5 calc_user_movie_effect_RMSE Function

The source code of the function `calc_user_movie_effect_RMSE` defined in the `UM-effect.functions.R` script to calculate the Root Mean Squared Error (RMSE) of the UME Model for the given Test Set is provided below:

```

calc_user_movie_effect_RMSE <- function(test_set, um_effect){
  mse <- test_set |> calc_user_movie_effect_MSE(um_effect)
  sqrt(mse)
}

```

2.5.3.6 calc_user_movie_effect_RMSE.cv Function

The source code of the function `calc_user_movie_effect_RMSE.cv` defined in the `UM-effect.functions.R` script to calculate the *5-Fold Cross Validation RMSE* result of the *UME Model* is provided 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", CVFolds_N, um_effect_RMSE)
  um_effect_RMSE
}

```

2.5.4 UME Model Building



The complete source code of builing and training the current model is available in the [Model building: User+Movie Effect](#) section of the [capstone-movielens.main.R](#) script on [GitHub](#).

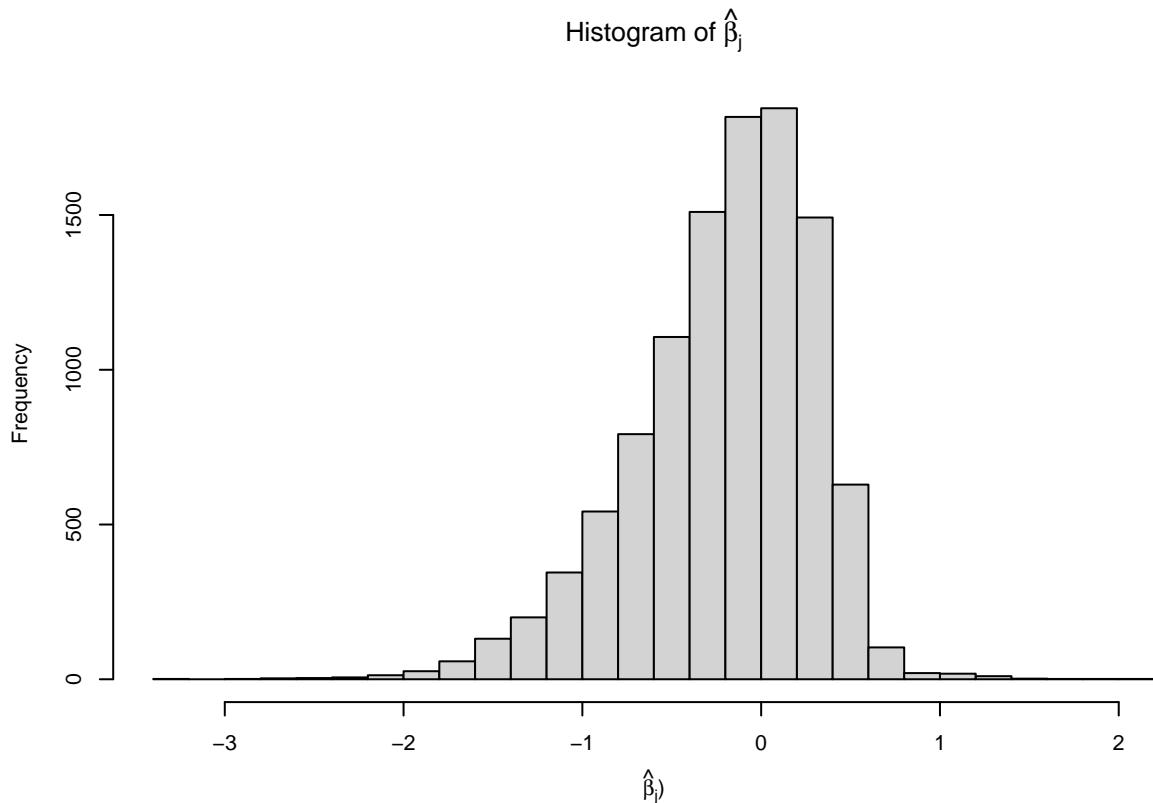
Below, we provide the most significant part of the code for training our model using the [5-Fold Cross Validation](#) method:

```
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 ...

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 see how much the RMSE improves[11]:

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

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

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

Method	RMSE	Comment
Project Objective	0.8649000	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	

2.5.5 UME Model Regularization

Section 23.6 *Penalized least squares* of the *Course Textbook* 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:

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

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

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

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

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

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 λ 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 λ , the more we shrink[12].

We will implement the *Regularization* method on our models (starting from the current model) in following three steps:

1. **Pre-configuration:** Preliminary determination of the optimal range of λ values for the 5-Fold Cross Validation samples;
2. **Fine-tuning:** figuring out the value of λ that minimizes the model's RMSE.
3. **Retraining:** retraining the model with the best value of the parameter λ obtained in the previous step.

2.5.5.1 UME Model Regularization: Support Functions



The `regularize.test_lambda.UM_effect.cv` function described below are defined in the Regularization section of the `UM-effect.functions.R` script.

2.5.5.1.1 `regularize.test_lambda.UM_effect.cv` Function

This function calculates *RMSE* of the *UME Model* using *5-Fold Cross Validation* method for the given λ parameter value:

```
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)  
}
```



Note that we reuse the function `train_user_movie_effect.cv` calling it from the `regularize.test_lambda.UM_effect.cv`, but now with the λ parameter different from the default (' $\lambda = 0$ ') value.

2.5.5.2 UME Model Regularization: Pre-configuration

Let's perform the preconfiguration to determine the appropriate range of λ for subsequent fine-tuning of our current model:

We are going to use the `tune.model_param` function described in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UM_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that 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"
```

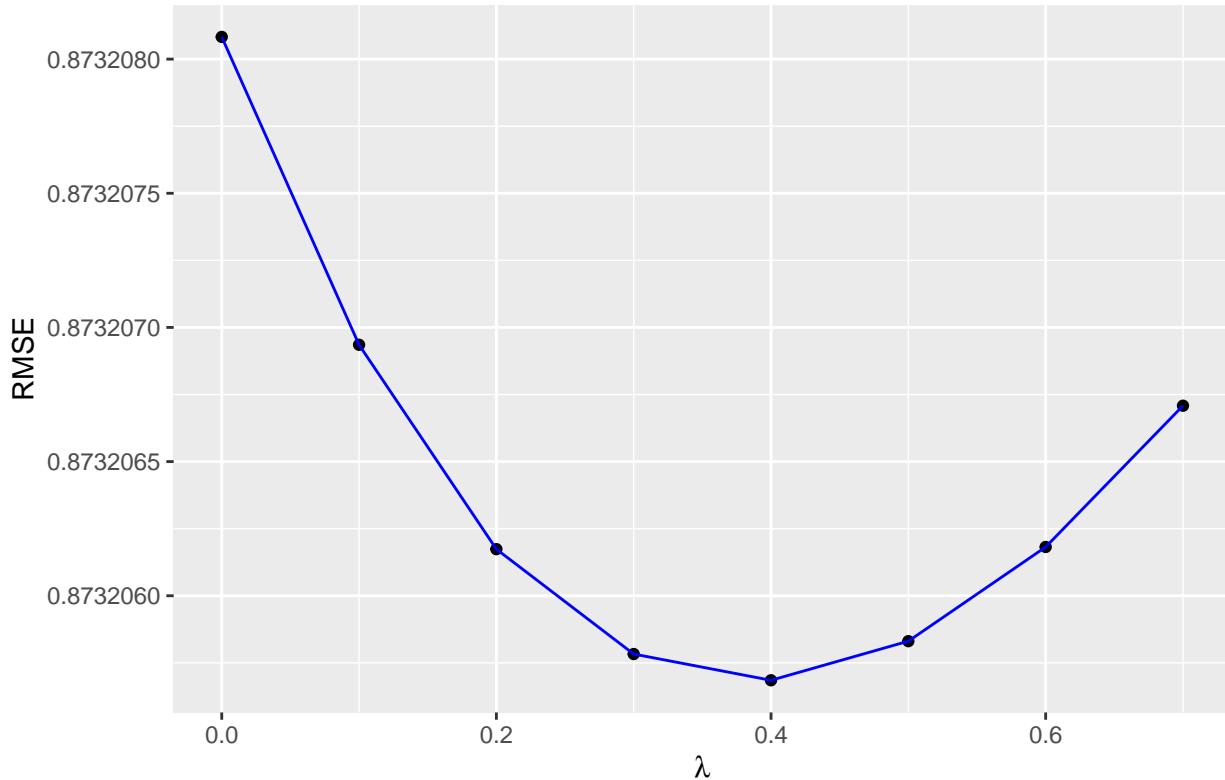


The complete version of the source code provided in this section can be found in the [UME Model Regularization: Pre-configuration](#) section of the [capstone-movielens.main.R](#) script.

Now, let's visualize the results of the λ range preconfiguration:

```
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")
```

UME Model Regularization: λ Range Pre-configuration



We use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

2.5.5.3 UME Model Regularization: Fine-tuning

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

Here we are going to use the `model.tune.param_range` function described in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UM_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that performs this operation:

```
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  
#> [1] 0.3 0.5 8.0  
  
UME.rgldr.fine_tune.cache.base_name <- "UME.rgldr.fine-tune"  
  
UME.rgldr.fine_tune.results <-  
  model.tune.param_range(UM_effect.loop_starter,  
                         UME.rgldr.fine_tune.cache.path,  
                         UME.rgldr.fine_tune.cache.base_name,  
                         regularize.test_lambda.UM_effect.cv)  
  #endpoint.min_diff = 1e-07/4  
  
UME.rgldr.fine_tune.RMSE.best <- UME.rgldr.fine_tune.results$best_result["best_RMSE"]  
  
## *** Fine-tuning results object data structure ***  
  
## 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 ***  
  
## param.best_value      best_RMSE  
##          0.3874500    0.8732057
```



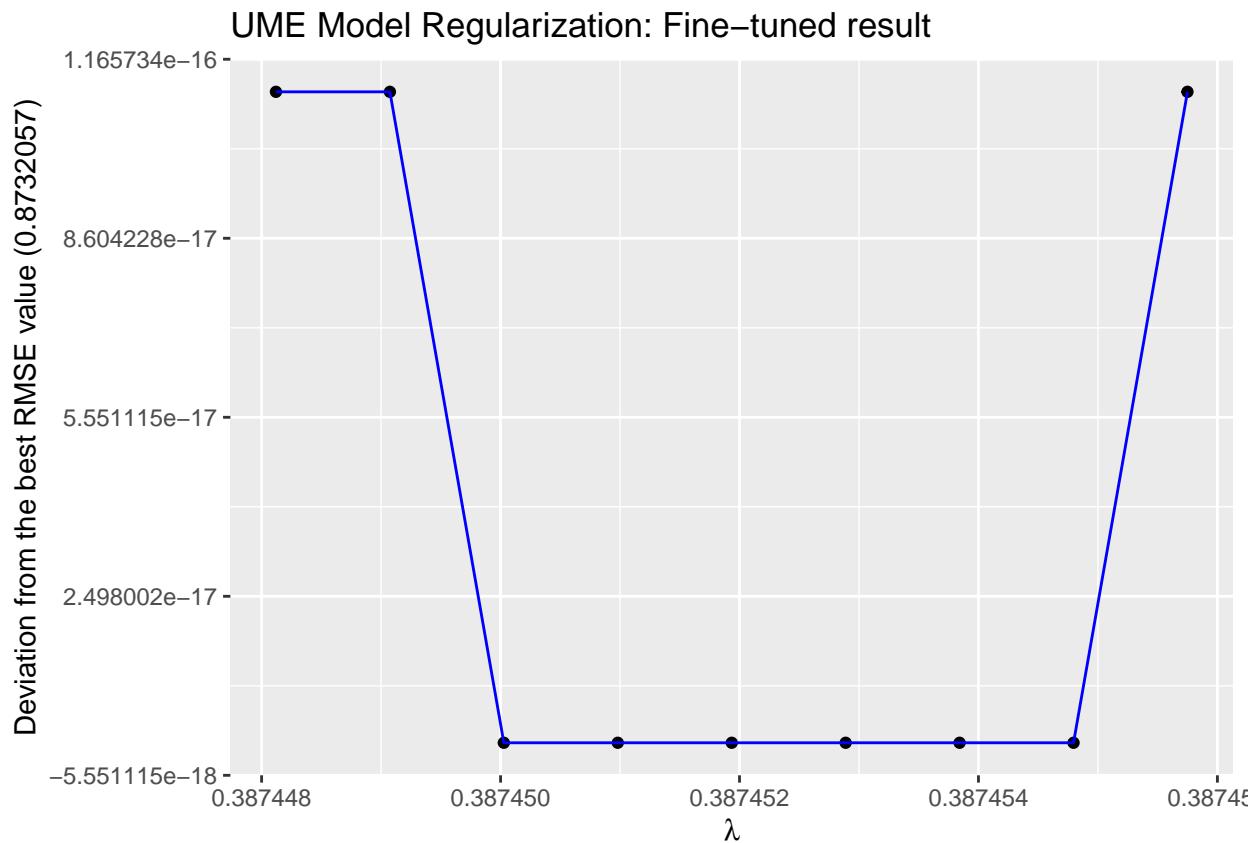
The complete version of the source code provided in this section can be also found in the [Fine-tuning Step of the Regularization Method for the User+Movie Model](#) section of the `capstone-movielens.main.R` script on [GitHub](#).

Let's visualize the fine-tuning results:

```

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)

```



2.5.5.4 UME Model Regularization: Retraining Model with the best λ

Now, we can calculate the *Regularized User+Movie Effect* by retraining our model on the entire `edx` dataset with the best value of the λ parameter we just calculated, for the definitive *Root Mean Squared Error* calculation and use in subsequent models.

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

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

## *** The Best Fine-tuning Results **

## param.best_value      best_RMSE
##          0.3874500    0.8732057

## *** Regularized User+Movie Effect Structure **

## 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" "param...
##   $ n      : int [1:10677] 23790 10779 7028 1577 6400 12346 7259 821 2278 15187 ...

## Regularized User+Movie Effect Model has been re-trained for the best 'lambda': 0.38745002746582.
```



The complete version of the source code provided in this section are available in the [Re-training Regularized User+Movie Effect Model for the best \$\lambda\$](#) section of the `capstone-movielens.main.R` script on *GitHub*.

We calculate the *Root Mean Squared Error* for the ultimately computed *User+Movie Effect* using `calc_user_movie_effect_RMSE.csv` function described above as follows:

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

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

```
## The best RMSE after being regularized: 0.872972999076497
```

Finally, we add the definitive result for the current model to our *Result Table*:

```
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.rgldr.UME)
```

Method	RMSE	Comment
Project Objective	0.8649000	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
Regularized UME Model	0.8729730	Computed for 'lambda' = 0.38745002746582

2.6 User+Movie+Genre Effect (UMGE) Model

As mentioned in [Section 23.7: Exercises](#) of the *Chapter “23 Regularization” of the Course Textbook* the `Movielens` dataset also has a genres column. This column includes every genre that applies to the movie (some movies fall under several genres)[[13](#)].

2.6.1 Movie Genres Effect Anasysis

The plot below shows strong evidence of a genre effect (for illustrative purposes, the plot shows only categories with more than 20, 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)

dim(genre_ratins_grp)

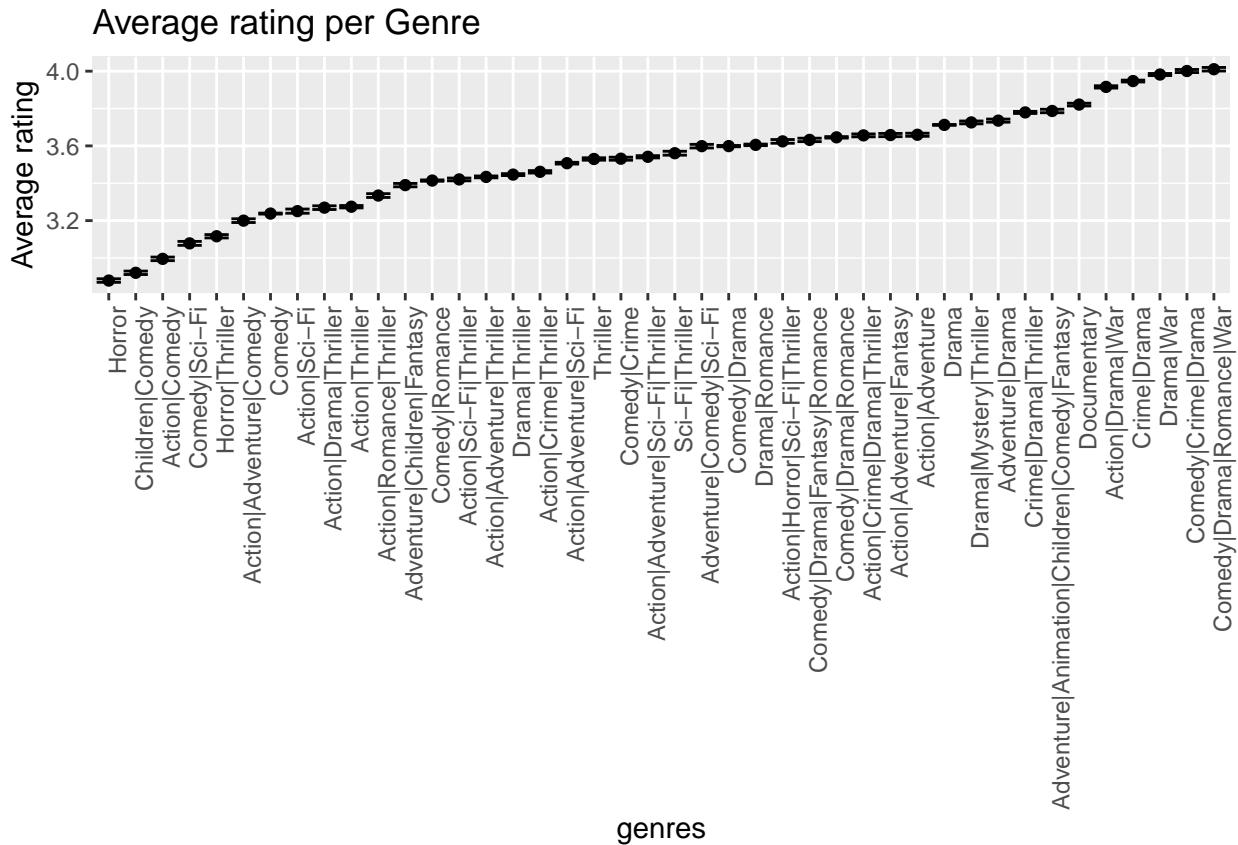
## [1] 42  4

genre_ratins_grp_sorted <- genre_ratins_grp |> sort_by.data.frame(~ rating_avg)
print(genre_ratins_grp_sorted)

## # A tibble: 42 x 4
##   genres           rating_avg      se      n
##   <fct>          <dbl>    <dbl>  <int>
## 1 Horror           2.88  0.00472  68738
## 2 Children|Comedy 2.92  0.00460  63483
## 3 Action|Comedy   3.00  0.00480  51289
## 4 Comedy|Sci-Fi   3.08  0.00526  44599
## 5 Horror|Thriller 3.12  0.00447  75000
## 6 Action|Adventure|Comedy 3.20  0.00498  45118
## 7 Comedy           3.24  0.00133  700889
## 8 Action|Sci-Fi   3.25  0.00561  49733
## 9 Action|Drama|Thriller 3.27  0.00492  45246
## 10 Action|Thriller 3.27  0.00319  96535
## # i 32 more rows

# 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"
```

Another way of visualizing a genre effect is shown in the section [Average rating for each genre](#) of the article [Movie Recommendation System using R - BEST](#) mentioned above[9]:

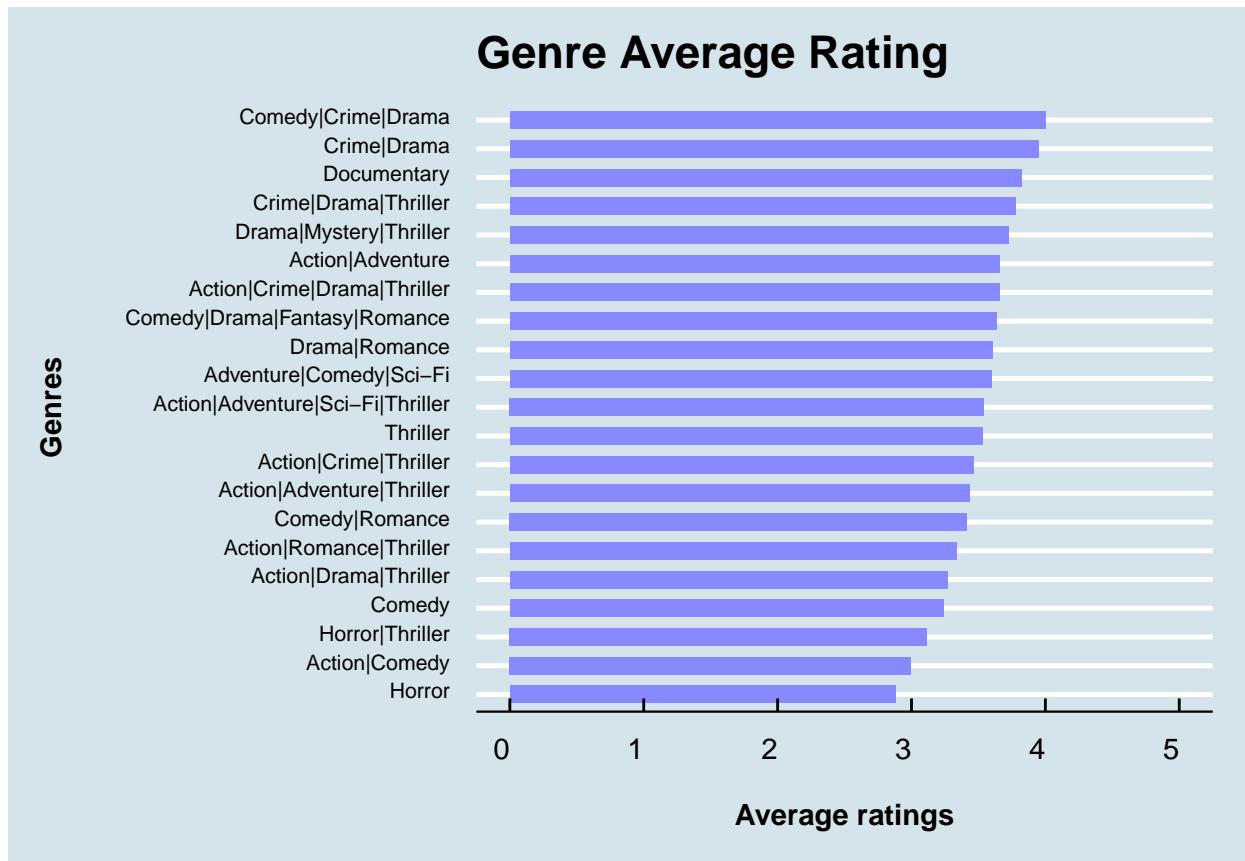
```
# For better visibility, we reduce the data for plotting
# while keeping the worst and best rating rows:
plot_ind <- odd(1:nrow(genre_ratins_grp))
plot_dat <- genre_ratins_grp_sorted[plot_ind,]

plot_dat |>
  ggplot(aes(x = rating_avg, y = genres)) +
  ggtitle("Genre Average Rating") +
  geom_bar(stat = "identity", width = 0.6, fill = "#8888ff") +
  xlab("Average ratings") +
  ylab("Genres") +
  scale_x_continuous(labels = comma, limits = c(0.0, 5.0)) +
  theme_economist()
```

```

theme(plot.title = element_text(vjust = 3.5),
      axis.title.x = element_text(vjust = -5, face = "bold"),
      axis.title.y = element_text(vjust = 10, face = "bold"),
      axis.text.x = element_text(vjust = 1, hjust = 1, angle = 0),
      axis.text.y = element_text(vjust = 0.25, hjust = 1, size = 8),
      plot.margin = margin(0.7, 0.5, 1, 1.2, "cm"))

```



2.6.2 Mathematical Description of the UMGE Model

To account for a *genre effect* we will use the model suggested in the [Section 23.7: Exercises](#) of the *Chapter 23 “Regularization” of the Course Textbook*[13]:

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:

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

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[14].

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:

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

2.6.3 UMGE Model: Support Functions



Some of the functions described in this section accept the `lambda` parameter, which we will need later for the *Model Regularization* method. We use the `mean_reg` function call as well, which we will also need for the *Regularization* techniques to apply to our models (for details, see the `mean_reg` function description in the Section [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report). We have already explained that in the [UME Model Regularization](#) section. For now, we omit the `lambda` parameter, accepting its default value `lambda = 0`. Let's recall that in this case, the `mean_reg` function is equivalent to the standard R function `base::mean`.

2.6.3.1 `train_user_movie_genre_effect` Function

We use this function to build and train our model using a *Train Set* (passing it to the `train.sgr` parameter), which can be any sample of the `edx.sgr` dataset (or the entire dataset itself) described above in the [edx.sgr Object](#) section. Here, I only remind that it is the dataset created from the `edx` one by splitting its rows to ensure each belongs to a single *genre*. Here is the source code of the function:

```
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))
}

```

2.6.3.2 `train_user_movie_genre_effect.cv` Function

We use this function, defined in the `UMG-effect.functions.R` script, to train the current model using the *5-Fold Cross Validation* method. Below, we provide a slightly simplified version of the source code of that function:

```

train_user_movie_genre_effect.cv <- function(lambda = 0){
  # ...

  put_log1("Function `train_user_movie_genre_effect.cv`:
  Computing User+Movie+Genre Effects list for %1-Fold Cross Validation samples...",
  CVFolds_N)

  start <- put_start_date()
  user_movie_genre_effects_ls <- lapply(kfold_index, function(fold_i){
    cv_fold_dat <- edx_CV[[fold_i]]

    put_log2("Processing User+Movie+Genre Effects for %1-Fold Cross Validation samples (Fold %2)...",
            CVFolds_N,
            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
  })
  put_end_date(start)
  put_log1("Function `train_user_movie_genre_effect.cv`:
  User+Movie+Genre Effects list has been computed for %1-Fold Cross Validation samples.",
  CVFolds_N)

  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
}

```



Here we use the function call `union_cv_results` to aggregate the *5-Fold Cross Validation* method results (for details, see the `union_cv_results` function description in the [Data Helper Functions](#) section of the [Appendix](#) to this report).

2.6.3.3 calc_user_movie_genre_effect_MSE Function

The source code of the function `calc_user_movie_genre_effect_MSE` defined in the `UMG-effect.functions.R` script to calculate the *Mean Squared Error (MSE)* of the *UMGE Model* for the given *Test Set* is provided 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()  
}
```

2.6.3.4 calc_user_movie_genre_effect_MSE.cv Function

The source code of the function `calc_user_movie_genre_effect_MSE.cv` defined in the `UMG-effect.functions.R` script to calculate the *5-Fold Cross Validation MSE* result of the *UMGE Model* is provided below:

```
calc_user_movie_genre_effect_MSE.cv <- function(umg_effect){  
  put_log("Computing RMSEs.ResultTibble on Validation Sets...")  
  start <- put_start_date()  
  user_movie_genre_effects_MSEs <- sapply(edx_CV, function(cv_dat){  
    cv_dat$validation_set |> calc_user_movie_genre_effect_MSE(umg_effect)  
  })  
  put_end_date(start)  
  
  plot(user_movie_genre_effects_MSEs)  
  put_log1("MSE values have been plotted for the %1-Fold Cross Validation samples.",  
          CVFolds_N)  
  
  mean(user_movie_genre_effects_MSEs)  
}
```

2.6.3.5 calc_user_movie_genre_effect_RMSE Function

The source code of the function `calc_user_movie_genre_effect_RMSE` defined in the `UMG-effect.functions.R` script to calculate the *Root Mean Squared Error (RMSE)* of the *UMGE Model* for the given *Test Set* is provided below:

```
calc_user_movie_genre_effect_RMSE <- function(test_set, umg_effect){  
  umg_mse <- test_set |> calc_user_movie_genre_effect_MSE(umg_effect)  
  sqrt(umg_mse)  
}
```

2.6.3.6 calc_user_movie_genre_effect_RMSE.cv Function

The source code of the function `calc_user_movie_genre_effect_RMSE.cv` defined in the `UMG-effect.functions.R` script to calculate the *5-Fold Cross Validation RMSE* result of the *UMGE Model* is provided below:

```
calc_user_movie_genre_effect_RMSE.cv <- function(umg_effect){  
  umg_effect_RMSE <- sqrt(calc_user_movie_genre_effect_MSE.cv(umg_effect))  
  put_log2("%1-Fold Cross Validation ultimate RMSE: %2",  
          CVFolds_N,  
          umg_effect_RMSE)  
  
  umg_effect_RMSE  
}
```

2.6.4 UMGE Model Building



The complete source code of builing and training the current model is available in the [Model building: User+Movie+Genre Effect](#) section of the [capstone-movielens.main.R](#) script on *GitHub*.

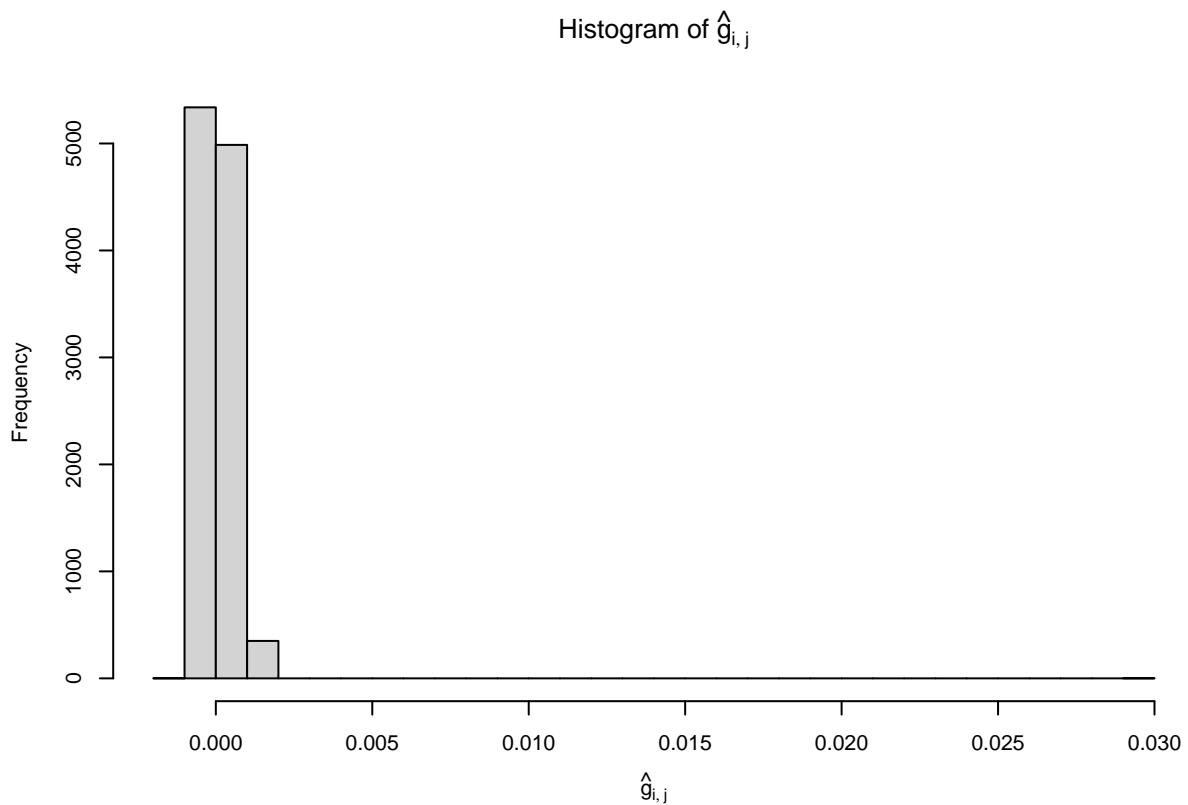
Below, we provide the most significant part of the code for training our model using the **5-Fold Cross Validation** method:

```
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 ...

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* of the current model using the `calc_user_movie_genre_effect_RMSE` function described above:

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

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

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

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



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

2.6.5 UMGE Model Regularization

We have already explained the idea of the *Linear Model Regularization* in the [UME Model Regularization](#) section above. Let's extend the concept outlined there to our current model.

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

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

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

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

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

As stated in the [UME Model Regularization](#) section, we implement the *Regularization* method for our models in the following three steps:

1. **Pre-configuration:** Preliminary determination of the optimal range of λ values for the [5-Fold Cross Validation](#) samples;
2. **Fine-tuning:** figuring out the value of λ that minimizes the model's RMSE.
3. **Retraining:** retraining the model with the best value of the parameter λ obtained in the previous step.

2.6.5.1 UMGE Model Regularization: Support Functions



The `regularize.test_lambda.UMG_effect.cv` function described below are defined in the `Regularization` section of the `UM-effect.functions.R` script.

2.6.5.1.1 `regularize.test_lambda.UMG_effect.cv` Function

This function calculates *RMSE* of the *UMGE Model* using *5-Fold Cross Validation* method for the given λ parameter value:

```
regularize.test_lambda.UMG_effect.cv <- function(lambda){  
  if (is.na(lambda)) {  
    stop("Function: regularize.test_lambda.user_movie_genre_effect.cv  
`lambda` is `NA`")  
  }  
  
  umg_effect <- train_user_movie_genre_effect.cv(lambda)  
  calc_user_movie_genre_effect_RMSE.cv(umg_effect)  
}
```



Note that we reuse the function `train_user_movie_genre_effect.cv` calling it from the `regularize.test_lambda.UMG_effect.cv`, but now with the λ parameter different from the default (' $\lambda = 0$ ') value.

2.6.5.2 UMGE Model Regularization: Pre-configuration

Let's perform the preconfiguration to determine the appropriate range of λ for subsequent fine-tuning of our current model:

We are going to use the `tune.model_param` function described in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UMG_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that performs this operation:

```
put_log1("Preliminary setting-up of `lambda`'s range for %1-Fold Cross Validation samples...",  
        CVFolds_N)  
  
start <- put_start_date()  
lambdas <- seq(0, 0.2, 0.01)  
cv.UMGE.preset.result <-  
  tune.model_param(lambdas, regularize.test_lambda.UMG_effect.cv)  
put_end_date(start)  
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
```

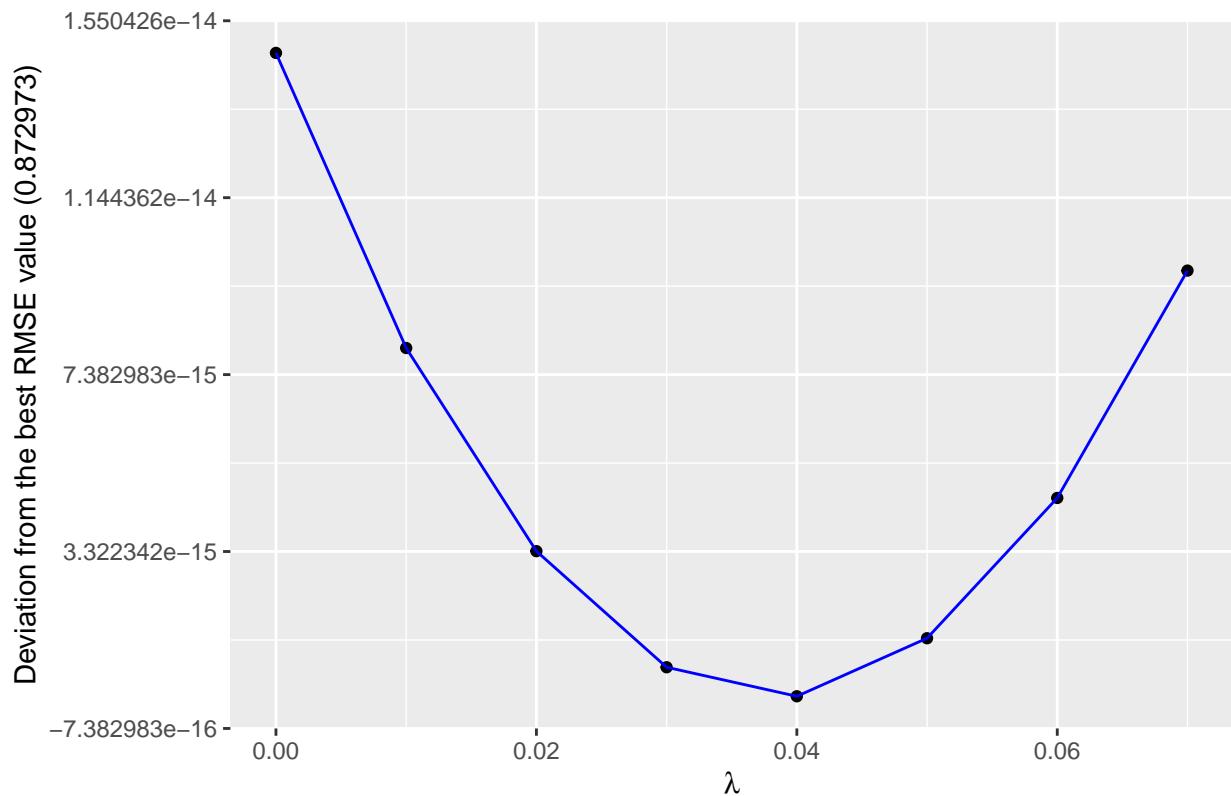


The complete version of the source code provided in this section can be found in the [UMGE Model Regularization: Pre-configuration](#) section of the [capstone-movielens.main.R](#) script.

Now, let's visualize the results of the λ range preconfiguration:

```
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)
```

UMGE Model Regularization: λ Range Pre-configuration



We use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

2.6.5.3 UMGE Model Regularization: Fine-tuning

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

Here we are going to use the `model.tune.param_range` function described in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UMG_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that performs this operation:

```
endpoints <-  
  get_fine_tune.param.endpoints(cv.UMGE.preset.result$tuned.result)  
  
UMG_effect.loop_starter <- c(endpoints["start"],  
  endpoints["end"],  
  8)  
UMG_effect.loop_starter  
#> [1] 0.0  0.1  8.0  
  
UMGE.rgldr.fine_tune.cache.base_name <- "UMGE.rgldr.fine-tuning"  
  
UMGE.rgldr.fine_tune.results <-  
  model.tune.param_range(UMG_effect.loop_starter,  
    UMGE.rgldr.fine_tune.cache.path,  
    UMGE.rgldr.fine_tune.cache.base_name,  
    regularize.test_lambda.UMG_effect.cv)  
  
UMGE.rgldr.fine_tune.RMSE.best <- UMGE.rgldr.fine_tune.results$best_result["best_RMSE"]  
# best_RMSE  
# 0.872973  
  
str(UMGE.rgldr.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 ...  
  
UMGE.rgldr.fine_tune.results$best_result  
  
## param.best_value      best_RMSE  
##          0.0359375     0.8729730
```



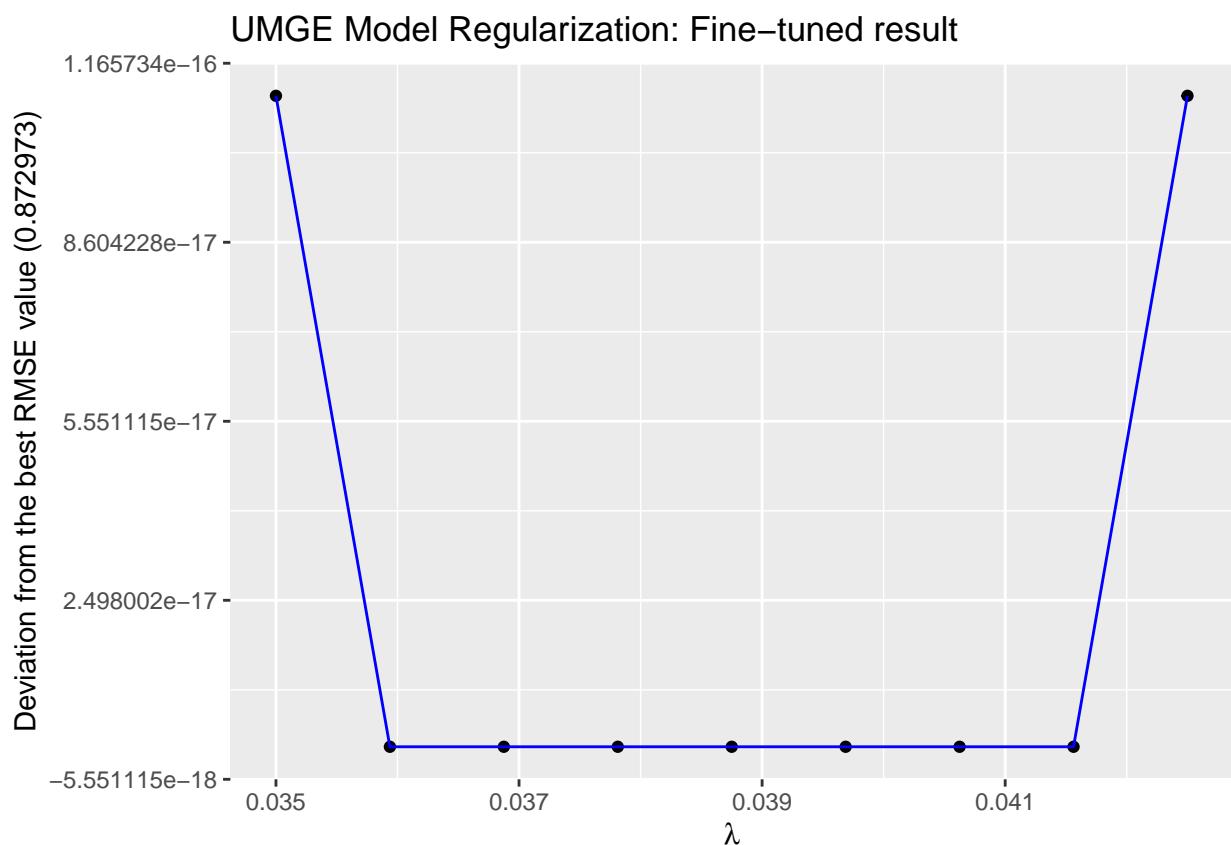
The complete version of the source code provided in this section can be also found in the [Fine-tuning Step of the Regularization Method for the User+Movie Model](#) section of the `capstone-movielens.main.R` script on *GitHub*.

Let's visualize the fine-tuning results:

```

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)

```



2.6.5.4 UMGE Model Regularization: Retraining Model with the best λ

Now, we can calculate the *Regularized UMG Effect* by retraining our model on the entire `edx` dataset with the best value of the λ parameter we just calculated, for the definitive *Root Mean Squared Error* calculation and use in subsequent models.

```
best_result <- UMGE.rglr.fine_tune.results$best_result
# param.best_value      best_RMSE
#       0.03554688      0.87297303

UMGE.rglr.best_lambda <- best_result["param.best_value"]
UMGE.rglr.best_RMSE <- best_result["best_RMSE"]

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)

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 ...
```



The complete version of the source code provided in this section are available in the [Re-training Regularized UMG Effect Model for the best \$\lambda\$](#) section of the `capstone-movielens.main.R` script on *GitHub*.

We calculate the *Root Mean Squared Error* for the ultimately computed *UMG Effect* using `calc_UMG_effect_RMSE.cv` function described above as follows:

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

## Regularized User+Movie+Genre Effect RMSE has been computed for the best 'lambda = 0.0359375': 0.8729
```

Finally, we add the definitive result for the current model to our *Result Table*:

```
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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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



As we can see, the current model still does not show much improvement after *regularization*, even though the data analysis we made in the [Movie Genres Effect] section showed strong evidence of a genre effect. It looks like we need a better model to account for a genre effect more efficiently.

Or, maybe, we have implemented the current model not quite correctly (?)

2.7 User+Movie+Genre+Year Effect (UMGYE) Model

2.7.1 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[9].

2.7.1.1 Yearly rating count[9]

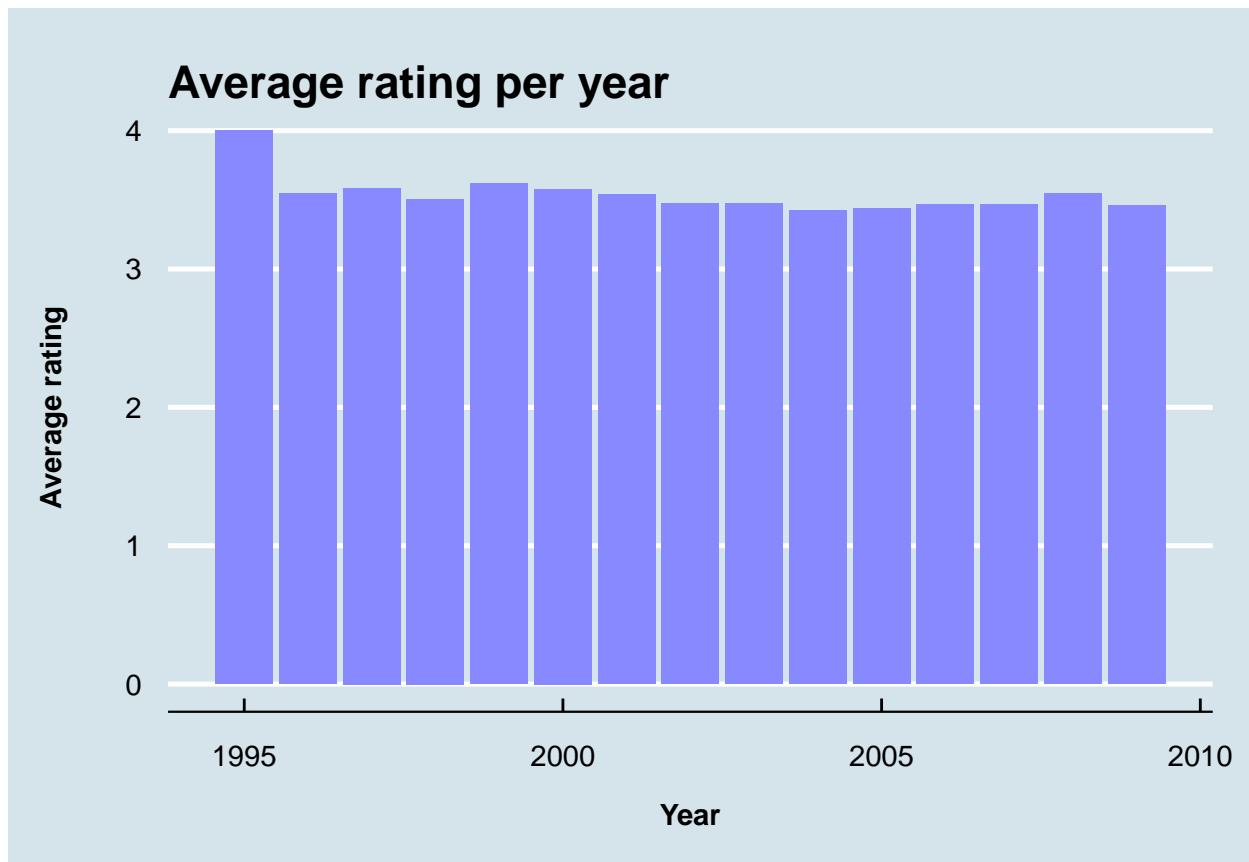
```
print(edx |>
  mutate(year = year(as_datetime(timestamp, origin = "1970-01-01")))) |>
  group_by(year) |>
  summarize(count = n())
)

## # 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
```

2.7.1.2 Average rating per year plot[9]

```
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"))
```



2.7.2 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:

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

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:

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

2.7.3 UMGYE Model: Support Functions



Some of the functions described in this section accept the `lambda` parameter, which we will need later for the *Model Regularization* method. We use the `mean_reg` function call as well, which we will also need for the *Regularization* techniques to apply to our models (for details, see the `mean_reg` function description in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report). We have already explained that in the [UME Model Regularization](#) section. For now, we omit the `lambda` parameter, accepting its default value `lambda = 0`. Let's recall that in this case, the `mean_reg` function is equivalent to the standard R function `base::mean`.

2.7.3.1 `calc_date_general_effect` Function

This is a helper function that calculates the *date general effect* of the user ratings and is intended to be used in other functions and scripts (such as `train_UMGY_effect`) described below.

The following is a slightly simplified version of the function's source code:

```
calc_date_general_effect <- function(train_set, lambda = 0){
  if (is.na(lambda)) {
    stop("Function: calc_date_general_effect
`lambda` is `NA`")
  }

  if(lambda == 0) put_log("Function `calc_date_general_effect`:
Computing Date Global Effect for given Train Set data...")
  else put_log1("Function `calc_date_general_effect`:
Computing Date Global Effect for lambda: %1...",
                lambda)
  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))
  dg_effect
}
```



The complete version of the source code of the `calc_date_general_effect` function is available in the [UMGYE Model Support Functions](#) section of the `UMGY-effect.functions.R` script on [GitHub](#).

2.7.3.1.1 Return A data structure containing information about *Date General Effect*.

2.7.3.2 `calc_date_general_effect.cv` Function

This is a helper function that calculates the *date general effect* of the user ratings using the *5-Fold Cross Validation* method and is intended to be used in other functions and scripts (such as `train_UMGY_effect.cv`) described below.

2.7.3.2.1 Source Code



The complete version of the source code of the `calc_date_general_effect.cv` function is available in the [UMGYE Model Support Functions](#) section of the [UMGY-effect.functions.R](#) script on [GitHub](#).

The following is a simplified version of the function's source code:

```
calc_date_general_effect.cv <- function(lambda = 0){
  if (is.na(lambda)) {
    stop("Function: calc_date_general_effect.cv
`lambda` is `NA`")
  }

  date_general_effect_ls <- lapply(edx_CV,  function(cv_fold_dat){
    cv_fold_dat$train_set |> calc_date_general_effect(lambda)
  })
  str(date_general_effect_ls)

  date_general_effect_united <- union_cv_results(date_general_effect_ls)

  date_general_effect_united |>
    group_by(days) |>
    summarise(de = mean(de, na.rm = TRUE), year = mean(year, na.rm = TRUE))
}
```



Here we use the function call `union_cv_results` to aggregate the *5-Fold Cross Validation* results (for details, see the `union_cv_results` function description in the [Data Helper Functions](#) section of the [Appendix](#) to this report).

2.7.3.2.2 Return A data structure containing information about *Date General Effect* as result of *5-Fold Cross Validation*.

2.7.3.3 `calc_UMGY_effect` Function

This is a helper function that calculates the *UMGY effect* of the user ratings and is intended to be used in other functions and scripts (such as `train_UMGY_effect`) described below.

2.7.3.3.1 Parameters

The function accepts the only parameter `date_general_effect`, which is a data structure returned by any of the following functions described above:

- `calc_date_general_effect`;
- `calc_date_general_effect.cv`.

2.7.3.3.2 Source Code

The following is the source code of the function:

```
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 [UMGYE Model Support Functions](#) section of the [UMGY-effect.functions.R](#) script on *GitHub*.

2.7.3.3.3 Return A data structure containing information about *User+Movie+Genre+Year (UMGY) Effect*.

2.7.3.4 `train_UMGY_effect` Function

This function is used to train the current model using the *Train Set* dataset passed in the `train_set` parameter. The following is the source code of the function:

```
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 [UMGYE Model Support Functions](#) section of the [UMGY-effect.functions.R](#) script on *GitHub*.

2.7.3.4.1 Return A data structure containing information about *User+Movie+Genre+Year (UMGY) Effect*.

2.7.3.5 `train_UMGY_effect.cv` Function

This function is used to train the current model using the *5-Fold Cross Validation*. The following is the source code of the function:

```
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 [UMGYE Model Support Functions](#) section of the `UMGY-effect.functions.R` script on *GitHub*.

2.7.3.5.1 Return A data structure containing information about *User+Movie+Genre+Year (UMGY) Effect*.

2.7.3.6 `calc_UMGY_effect_MSE` Function

The function calculates the *Mean Squared Error (MSE)* of the *UMGYE Model* for the given *Test Set* passed in the `test_set` parameter.

2.7.3.6.1 Parameters

- **test_set:** A *Test Set* dataset used for *UMGYE Model* validation;
- **UMGY_effect:** A data structure returned by any of the following functions described above:
 - `calc_UMGY_effect`;
 - `train_UMGY_effect`;
 - `train_UMGY_effect.cv`.

2.7.3.6.2 Source Code

The source code of the function is provided 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 function `calc_UMGY_effect_MSE` is also available in the [UMGYE Model Support Functions](#) script.

2.7.3.7 `calc_UMGY_effect_MSE.cv` Function

The function calculates the *5-Fold Cross Validation MSE* result for the *UMGY Effect* data passed in the `UMGY_effect` parameter.

2.7.3.7.1 Parameters

The function accepts the only parameter: `UMGY_effect`, which is a data structure returned by the one of the following functions described above:

- `calc_UMGY_effect`;
- `train_UMGY_effect`;
- `train_UMGY_effect.cv`.

2.7.3.7.2 Source Code

The source code of the function is provided 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 source code of the function `calc_UMGY_effect_MSE.csv` is also available in the [UMGYE Model Support Functions](#) script.

2.7.3.8 `calc_UMGY_effect_RMSE` Function

The function calculates the *Mean Squared Error (MSE)* of the *UMGYE Model* for the given *Test Set* passed in the `test_set` parameter.

2.7.3.8.1 Parameters

- `test_set`: A *Test Set* dataset used for *UMGYE Model* validation;
- `UMGY_effect`: A data structure returned by any of the following functions described above:
 - `calc_UMGY_effect`;
 - `train_UMGY_effect`;
 - `train_UMGY_effect.csv`.

2.7.3.8.2 Source Code

The source code of the function is provided below:

```
calc_UMGY_effect_RMSE <- function(test_set, UMGY_effect){  
  mse <- test_set |> calc_UMGY_effect_MSE(UMGY_effect)  
  sqrt(mse)  
}
```



The source code of the function `calc_UMGY_effect_RMSE` is also available in the [UMGYE Model Support Functions](#) script.

2.7.3.9 `calc_UMGY_effect_RMSE.csv` Function

The function calculates the *5-Fold Cross Validation MSE* result for the *UMGY Effect* data passed in the `UMGY_effect` parameter.

2.7.3.9.1 Parameters

The function accepts the only parameter: `UMGY_effect`, which is a data structure returned by the one of the following functions described above:

- `calc_UMGY_effect`;
- `train_UMGY_effect`;
- `train_UMGY_effect.csv`.

2.7.3.9.2 Source Code

The source code of the function is provided 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 function `calc_UMGY_effect_RMSE.cv` is also available in the [UMGYE Model Support Functions](#) script.

2.7.4 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, we provide the most significant line of the code for training our model using the **5-Fold Cross Validation** method:

```
cv.UMGY_effect <- train_UMGY_effect.cv()  
  
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 ...  
  
summary(cv.UMGY_effect)  
  
##      year          ye  
##  Min.   :1995   Min.   :-0.06477  
##  1st Qu.:1998   1st Qu.:-0.02190  
##  Median :2002   Median : 0.01182  
##  Mean   :2002   Mean   : 0.04811  
##  3rd Qu.:2006   3rd Qu.: 0.02937  
##  Max.   :2009   Max.   : 0.66940
```



We use the `train_UMGY_effect.cv` function described above to compute the *UMGY Effect*.

We can now construct predictors and calculate the *RMSE* of the current model using the `calc_UMGY_effect_RMSE.cv` function described above:

```
cv.UMGY_effect.RMSE <- calc_UMGY_effect_RMSE.cv(cv.UMGY_effect)  
  
RMSEs.ResultTibble.UMGYE <- RMSEs.ResultTibble.rgldr.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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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



Now, we have a bit better result than one for the previous model.

2.7.5 UMGYE Model Regularization

We have already explained the idea of *Linear Model Regularization* in the **UME Model Regularization** section above. 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:

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

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

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

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

As with the previous models, we implement the *Regularization* method for the current model in the following three steps:

1. **Pre-configuration:** Preliminary determination of the optimal range of λ values for the **5-Fold Cross Validation** samples;
2. **Fine-tuning:** figuring out the value of λ that minimizes the model's RMSE.
3. **Retraining:** retraining the model with the best value of the parameter λ obtained in the previous step.

2.7.5.1 UMGYE Model Regularization: Support Functions



The `regularize.test_lambda.UMGY_effect.cv` function described below are defined in the `Regularization` section of the `UM-effect.functions.R` script.

2.7.5.1.1 `regularize.test_lambda.UMGY_effect.cv` Function

This function calculates *RMSE* of the *UMGYE Model* using *5-Fold Cross Validation* for the given λ parameter value:

```
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()  
}
```



Note that we reuse the function `train_UMGY_effect.cv` calling it from the `regularize.test_lambda.UMGY_effect.cv`, but now with the λ parameter different from the default (' $\lambda = 0$ ') value.

2.7.5.2 UMGYE Model Regularization: Pre-configuration

Let's perform the preconfiguration to determine the appropriate range of λ for subsequent fine-tuning of our current model:

We are going to use the `tune.model_param` function described in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UMGY_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that 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
##          224.000000     0.8721852
```

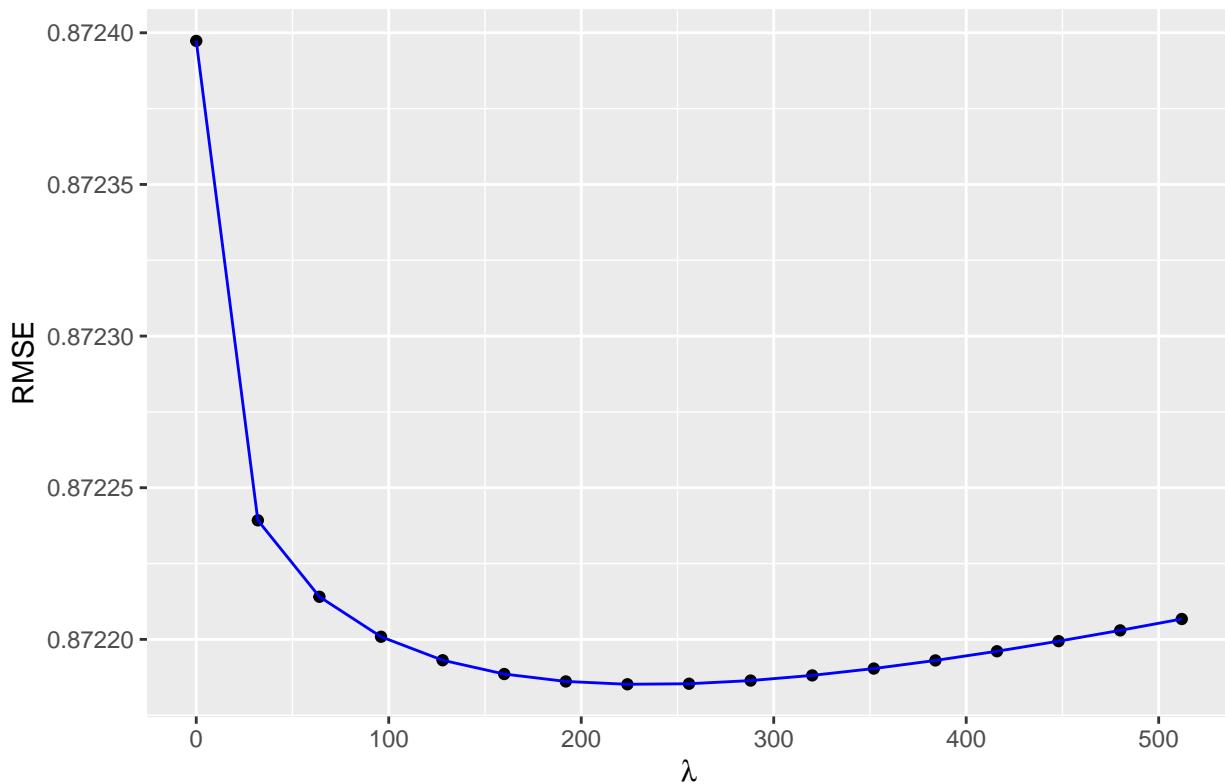


The complete version of the source code provided in this section can be found in the [UMGYE Model Regularization: Pre-configuration](#) section of the `capstone-movielens.main.R` script on [GitHub](#).

Now, let's visualize the results of the λ range preconfiguration:

```
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")
```

UMGYE Model Regularization: λ Range Pre-configuration



We use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

2.7.5.3 UMGYE Model Regularization: Fine-tuning



The complete version of the source code provided in this section can be found in the [Fine-tuning Step of the Regularization Method for the User+Movie Model](#) 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 λ parameter.

Here we are going to use the `model.tune.param_range` function described in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UMGY_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that performs this operation:

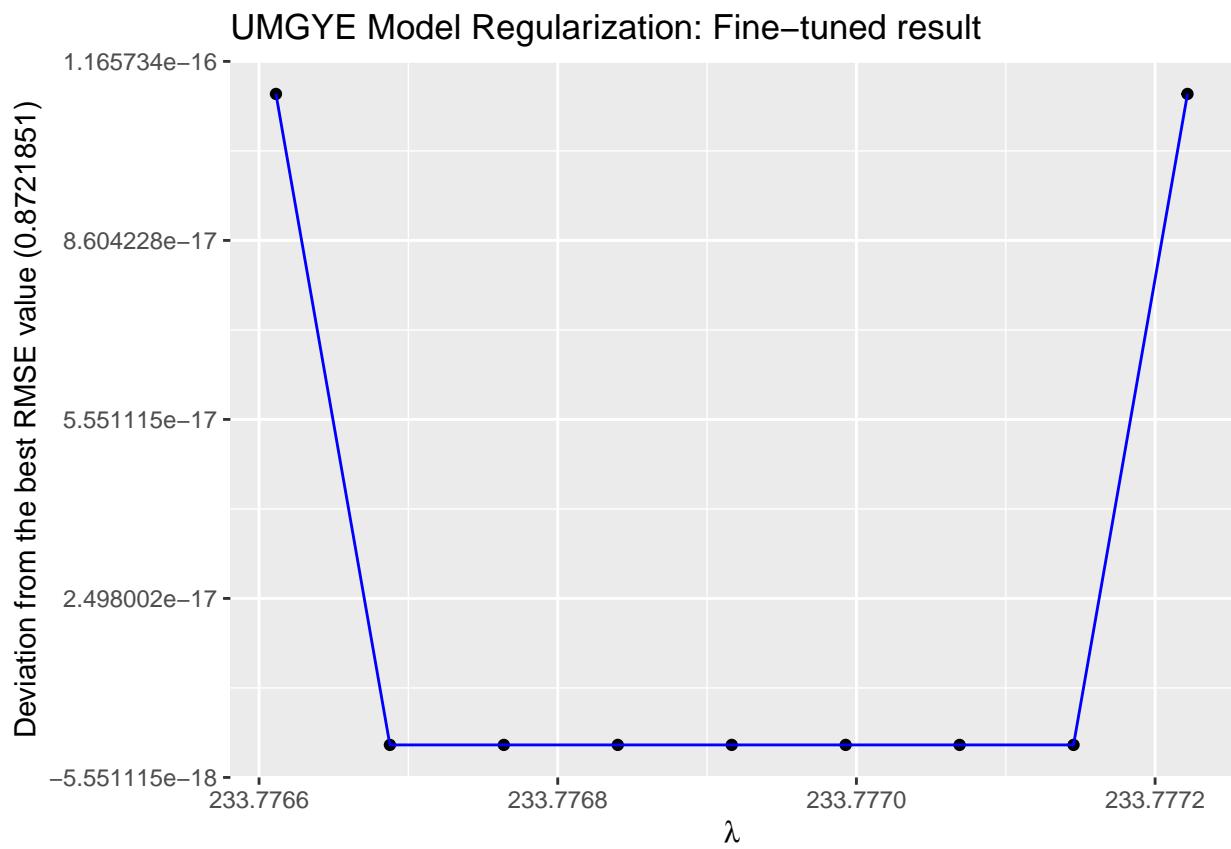
```
endpoints <-  
  get_fine_tune.param.endpoints(cv.UMGYE.preset.result$tuned.result)  
  
UMGYE.loop_starter <- c(endpoints["start"],  
                         endpoints["end"],  
                         8)  
UMGYE.loop_starter  
#> [1]  
  
cache.base_name <- "UMGYE.rglr.fine-tuning"  
  
UMGYE.rglr.fine_tune.results <-  
  model.tune.param_range(UMGYE.loop_starter,  
                         UMGYE.rglr.fine_tune.cache.path,  
                         cache.base_name,  
                         regularize.test_lambda.UMGY_effect.cv)  
  
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 ...  
  
UMGYE.rglr.fine_tune.results$best_result  
  
## param.best_value      best_RMSE  
##      233.7766876      0.8721851
```



Under the hood, this code block internally calls the helper function `get_fine_tune.param.endpoints` described in the [Common Helper Functions: Regularization](#) section of the [Appendix](#) to this report to figure out the boundaries of the range of values from the nearest neighborhood of the value corresponding to the best RMSE in the data passed in as a parameter.

Let's visualize the fine-tuning results:

```
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)
```



We use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

2.7.5.4 UMGYE Model Regularization: Retraining Model with the best λ



The complete version of the source code provided in this section are available in the [Re-train Regularized UMGYE Model for the best lambda](#) section of the `capstone-movielens.main.R` script on [GitHub](#).

Now, we can calculate the *Regularized UMGYE Effect* by retraining our model on the entire `edx` dataset with the best value of the λ parameter we just calculated, for the definitive *Root Mean Squared Error* calculation and use in subsequent models.

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



Here we use the helper function `train_UMGY_effect` described above in the [UMGYE Model Regularization: Support Functions](#) section.

We calculate the *Root Mean Squared Error* for the ultimately computed *UMGYE Effect* using the helper function `calc_UMGY_effect_RMSE.cv` described above in the [UMGYE Model Regularization: Support Functions](#) section as follows:

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

Finally, we add the definitive result for the current model to our *Result Table*:

```
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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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

2.8 User+Movie+Genre+Year+Day Effect (UMGYDE) Model

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 (11) describing the *UMGYDE Model*, for the current model, takes the form:

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

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:

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

2.8.1 UMGYDE Model: Helper Functions



Some of the functions described in this section accept the `lambda` parameter, which we will need later for the *Model Regularization* method. We use the `mean_reg` function call as well, which we will also need for the *Regularization* techniques to apply to our models (for details, see the `mean_reg` function description in the *Regularization: Common Helper Functions* section of the [Appendix](#) to this report). We have already explained that in the [UME Model Regularization](#) section. For now, we omit the `lambda` parameter, implicitly using its default value `lambda = 0`. Let's recall that in this case, the `mean_reg` function is equivalent to the standard R function `base::mean`.

2.8.1.1 `calc_day_general_effect` Function

This is a helper function that calculates the *Day General Effect* of user ratings on a *given day since the earliest record* in the `edx` dataset and is intended to be used in other functions and scripts (such as `train_UMGY_SmoothedDay_effect`) described below.

2.8.1.1.1 Parameters

`train_set`

The *Train Set* dataset.

`lambda`

As explained [above](#) in this section, this parameter is used for the *Regularization* techniques. Otherwise, it is omitted, meaning its default (zero) value is used.

2.8.1.1.2 Source Code

The following is a simplified version of the function's source code:

```
calc_day_general_effect <- function(train_set, lambda = 0){  
  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))  
}
```



The complete version of the source code of the `calc_day_general_effect` function is available in the [UMGYDE Model Support Functions](#) section of the [UMGYD-effect.functions.R](#) script on *GitHub*.

2.8.1.1.3 Return

A data frame object representing the *Day General Effect*.

2.8.1.2 `calc_day_general_effect.cv` Function

This is a helper function that calculates the *Day General Effect* of user ratings on a *given day since the earliest record* in the `edx` dataset using the *5-Fold Cross Validation* and is intended to be used in other functions and scripts (such as `train_UMGY_SmoothedDay_effect.cv`) described below.

2.8.1.2.1 Parameters

lambda

As explained [above](#) in this section, this parameter is used for the *Regularization* techniques. Otherwise, it is omitted, meaning its default (zero) value is used.

2.8.1.2.2 Source Code



The complete version of the source code of the `calc_day_general_effect.cv` function is available in the [UMGYDE Model Support Functions](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

The following is a simplified version of the function's source code:

```
calc_day_general_effect.cv <- function(lambda = 0){  
  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))  
}
```



Here we use the function call `union_cv_results` to aggregate the *5-Fold Cross Validation* results (for details, see the `union_cv_results` function description in the [Data Helper Functions](#) section of the [Appendix](#) to this report).

2.8.1.2.3 Return

A data frame object representing the *Day General Effect*.

2.8.1.3 loess_de Function

This is a helper function that calculates the *Smoothed Day Effect* of user ratings and is intended to be used in other functions and scripts (such as `calc_UMGY_SmoothedDay_effect`) described below.

2.8.1.3.1 Parameters

de_bias.dat

A data frame object representing the *Day General Effect*. It can be returned by the following functions described above:

- `calc_day_general_effect`;
- `calc_day_general_effect.cv`.

degree

A parameter to pass to the internally called `stats::loess` function (see the [Details](#) section below), meaning the degree of the polynomials to be used.

span

A parameter to pass to the internally called `stats::loess` function (see the [Details](#) section below) to control the degree of smoothing.

2.8.1.3.2 Details

The function is a wrapper for the `stats::loess` and calls it under the hood.

2.8.1.3.3 Source Code

The following is the source code of the function:

```
loess_de <- function(de_bias.dat, degree = NA, span = NA){  
  if(is.na(degree)) degree = 2  
  if(is.na(span)) span = 0.75  
  loess(de ~ days, span = span, degree = degree, data = de_bias.dat)  
}
```



The source code of the `loess_de` function is also available in the [UMGYDE Model Support Functions](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

2.8.1.3.4 Return

An object of class `loess` returned by the internally called function `stats::loess` (see the [Details](#) section above).

2.8.1.4 `calc_UMGY_SmoothedDay_effect` Function

This is a helper function that calculates the *Smoothed Day Effect* of the user ratings and is intended to be used in other functions and scripts (such as `train_UMGY_SmoothedDay_effect`) described below.

2.8.1.4.1 Parameters

day_gen_effect

A data frame object representing the *Day General Effect*. It can be returned by the following functions described above:

- `calc_day_general_effect`;
- `calc_day_general_effect.cv`.

degree

A parameter to pass to the internally called `loess_de` function (see the [Details](#) section below), meaning the degree of the polynomials to be used.

span

A parameter to pass to the internally called `loess_de` function (see the [Details](#) section below) to control the degree of smoothing.

2.8.1.4.2 Details

Under the hood, the function internally calls the `loess_de` described above and constructs the *User+Movie+Genre+Year+(Smoothed)Day (UMGYD) Effect* data object based on the object of class `loess` it returns along with the data passed by the `day_gen_effect` parameter.

2.8.1.4.3 Source Code

The following is the source code of the function:

```
calc_UMGY_SmoothedDay_effect <- function(day_gen_effect,
                                             degree = NA,
                                             span = NA){
  put_log2("Function `calc_UMGY_SmoothedDay_effect`:
Training model using `loess` function with the following parameters:
degree: %1;
span: %2
...",
         degree,
         span)
  # browser()
  fit <- day_gen_effect |> loess_de(degree, span)
  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 source code of the `calc_UMGY_SmoothedDay_effect` function is also available in the [UMGYDE Model Support Functions](#) section of the [UMGYD-effect.functions.R](#) script on [GitHub](#).

2.8.1.4.4 Return

A data frame object representing the *User+Movie+Genre+Year+(Smoothed)Day (UMGYD) Effect*.

2.8.1.5 `train_UMGY_SmoothedDay_effect` Function

This is a helper function that calculates the *Smoothed Day Effect* of the user ratings and is intended to be used in other functions and scripts (such as `train_UMGY_SmoothedDay_effect`) described below.

2.8.1.5.1 Parameters

train_set

The *Train Set* dataset.

degree

A parameter to pass to the internally called `calc_UMGY_SmoothedDay_effect` function (see the Details section below), meaning the degree of the polynomials to be used.

span

A parameter to pass to the internally called `calc_UMGY_SmoothedDay_effect` function (see the Details section below) to control the degree of smoothing.

lambda

As explained above in this section, this parameter is used for the *Regularization* techniques. Otherwise, it is omitted, meaning its default (zero) value is used.

2.8.1.5.2 Details

Under the hood, the function internally calls the `calc_day_general_effect` and then `calc_UMGY_SmoothedDay_effect` described above, and returns the *User+Movie+Genre+Year+(Smoothed)Day (UMGYD) Effect* data object that the latter inner function returns.

2.8.1.5.3 Source Code

The following is the source code of the function:

```
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) |>
    train_UMGY_SmoothedDay_effect(degree, span)
}
```



The source code of the `train_UMGY_SmoothedDay_effect` function is also available in the [UMGYDE Model Support Functions](#) section of the [UMGYD-effect.functions.R](#) script on [GitHub](#).

2.8.1.5.4 Return

A data frame object representing the *User+Movie+Genre+Year+(Smoothed)Day (UMGYD) Effect*.

2.8.1.6 `train_UMGY_SmoothedDay_effect.cv` Function

This is a helper function that calculates the *Smoothed Day Effect* of the user ratings and is intended to be used in other functions and scripts (such as `train_UMGY_SmoothedDay_effect.cv`) described below.

2.8.1.6.1 Parameters

degree

A parameter to pass to the internally called `calc_UMGY_SmoothedDay_effect` function (see the Details section below), meaning the degree of the polynomials to be used.

span

A parameter to pass to the internally called `calc_UMGY_SmoothedDay_effect` function (see the Details section below) to control the degree of smoothing.

lambda

As explained above in this section, this parameter is used for the *Regularization* techniques. Otherwise, it is omitted, meaning its default (zero) value is used.

2.8.1.6.2 Details

Under the hood, the function internally calls the `calc_day_general_effect.cv` and then `calc_UMGY_SmoothedDay_effect` described above, and returns the *User+Movie+Genre+Year+(Smoothed)Day (UMGYD) Effect* data object that the latter inner function returns.

2.8.1.6.3 Source Code

The following is the source code of the function:

```
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.cv(lambda) |>
    calc_UMGY_SmoothedDay_effect(degree, span)
}
```



The source code of the `train_UMGY_SmoothedDay_effect.cv` function is also available in the [UMGYDE Model Support Functions](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

2.8.1.6.4 Return

A data frame object representing the *User+Movie+Genre+Year+(Smoothed)Day (UMGYD) Effect*.

2.8.1.7 UMGY_SmoothedDay_effect.predict Function

This is a helper function that predicts user rating values on the *Test Set* passed by the `test_set` parameter, taking into account the *User+Movie+Genre+Year+(Smoothed)Day (UMGYD) Effect* passed by the `day_smoothed_effect` parameter.

2.8.1.7.1 Parameters

`test_set`

A *Test Set* dataset for the *UMGYDE Model* validation.

`day_smoothed_effect`

A data object representing the *UMGYD Effect*, which can be returned by any of the following functions described above:

- `calc_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect.cv`.

2.8.1.7.2 Source Code

Below is the source code of the function:

```
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 [UMGYDE Model Support Functions](#) section of the `UMGYD-effect.functions.R` script on *GitHub*.

2.8.1.7.3 Return

A data frame object containing predicted values.

2.8.1.8 `calc_UMGY_SmoothedDay_effect.MSE` Function

The function calculates the *Mean Squared Error (MSE)* of the *UMGYDE Model* for the given *Test Set* passed by the `test_set` parameter.

2.8.1.8.1 Parameters

`test_set`

A *Test Set* dataset for the *UMGYDE Model* validation.

`day_smoothed_effect`

A data object representing the *UMGYD Effect*, which can be returned by any of the following functions described above:

- `calc_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect.cv`.

2.8.1.8.2 Source Code

The source code of the function is provided 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 function `calc_UMGY_SmoothedDay_effect.MSE` is also available in the [UMGYDE Model Support Functions](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

2.8.1.8.3 Return

The *Mean Squared Error (MSE)* value of the *UMGYDE Model* computed on the *Test Set* passed by the `test_set` parameter.

2.8.1.9 `calc_UMGY_SmoothedDay_effect.MSE.cv` Function

The function calculates the *5-Fold Cross Validation MSE* result for the *UMGYD Effect* data passed by the `day_smoothed_effect` parameter.

2.8.1.9.1 Parameters

day_smoothed_effect

A data object representing the *UMGYD Effect*, which can be returned by any of the following functions described above:

- `calc_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect.cv`.

2.8.1.9.2 Source Code

Below is a simplified version of the function's source code:

```
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)  
  })  
  
  mean(smth_day_effect_MSEs)  
}
```



The complete version of the source code of the `calc_UMGY_SmoothedDay_effect.MSE.cv` function is available in the [UMGYDE Model Support Functions](#) section of the [UMGYD-effect.functions.R](#) script on *Github*.

2.8.1.9.3 Return

The *Mean Squared Error (MSE)* value of the *UMGYDE Model* computed as the mean of the *5-Fold Cross Validation* results.

2.8.1.10 `calc_UMGY_SmoothedDay_effect.RMSE` Function

The function calculates the *Root Mean Squared Error (RMSE)* of the *UMGYDE Model* for the given *Test Set* passed by the `test_set` parameter.

2.8.1.10.1 Parameters

test_set

A *Test Set* dataset for the *UMGYDE Model* validation.

day_smoothed_effect

A data object representing the *UMGYD Effect*, which can be returned by any of the following functions described above:

- `calc_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect.cv`.

2.8.1.10.2 Source Code

The source code of the function is provided below:

```
calc_UMGY_SmoothedDay_effect.RMSE <- function(test_set, day_smoothed_effect){
  day_smth_effect_MS <- test_set |>
    calc_UMGY_SmoothedDay_effect.MSE(day_smoothed_effect)

  sqrt(day_smth_effect_MS)
}
```



The source code of the function `calc_UMGY_SmoothedDay_effect.RMSE` is also available in the [UMGYDE Model Support Functions](#) section of the [UMGYD-effect.functions.R](#) script.

2.8.1.10.3 Return

The *Root Mean Squared Error (RMSE)* value of the *UMGYDE Model* computed on the *Test Set* passed by the `test_set` parameter.

2.8.1.11 `calc_UMGY_SmoothedDay_effect.RMSE.cv` Function

The function calculates the *5-Fold Cross Validation RMSE* result for the *UMGYD Effect* data passed by the `day_smoothed_effect` parameter.

2.8.1.11.1 Parameters

day_smoothed_effect

A data object representing the *UMGYD Effect*, which can be returned by any of the following functions described above:

- `calc_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect`;
- `train_UMGY_SmoothedDay_effect.cv`.

2.8.1.11.2 Source Code

The source code of the function is provided 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 function `calc_UMGY_SmoothedDay_effect.RMSE.cv` is also available in the [UMGYDE Model Support Functions](#) section of the [UMGYD-effect.functions.R](#) script.

2.8.1.11.3 Return

The *Root Mean Squared Error (RMSE)* of the *UMGYD Model* computed as the square root of the *5-Fold Cross Validation MSE* result.

2.8.2 UMGYDE Model Building With 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 our helper function `train_UMGY_SmoothedDay_effect.cv` described in the [UMGYDE Model: Helper Functions](#) section, which, as we explained, internally calls the R function `stats::loess`.

We will start by calling the `train_UMGY_SmoothedDay_effect.cv` with default parameters, which means that the `loess` function will be implicitly called with `degree = 1` and `span = 0.75` parameters (the default values for our custom `loess_de` function, which, in turn, calls the `loess` under the hood).

Below, we provide the most significant line of the code for training our model using the [5-Fold Cross Validation](#) method:

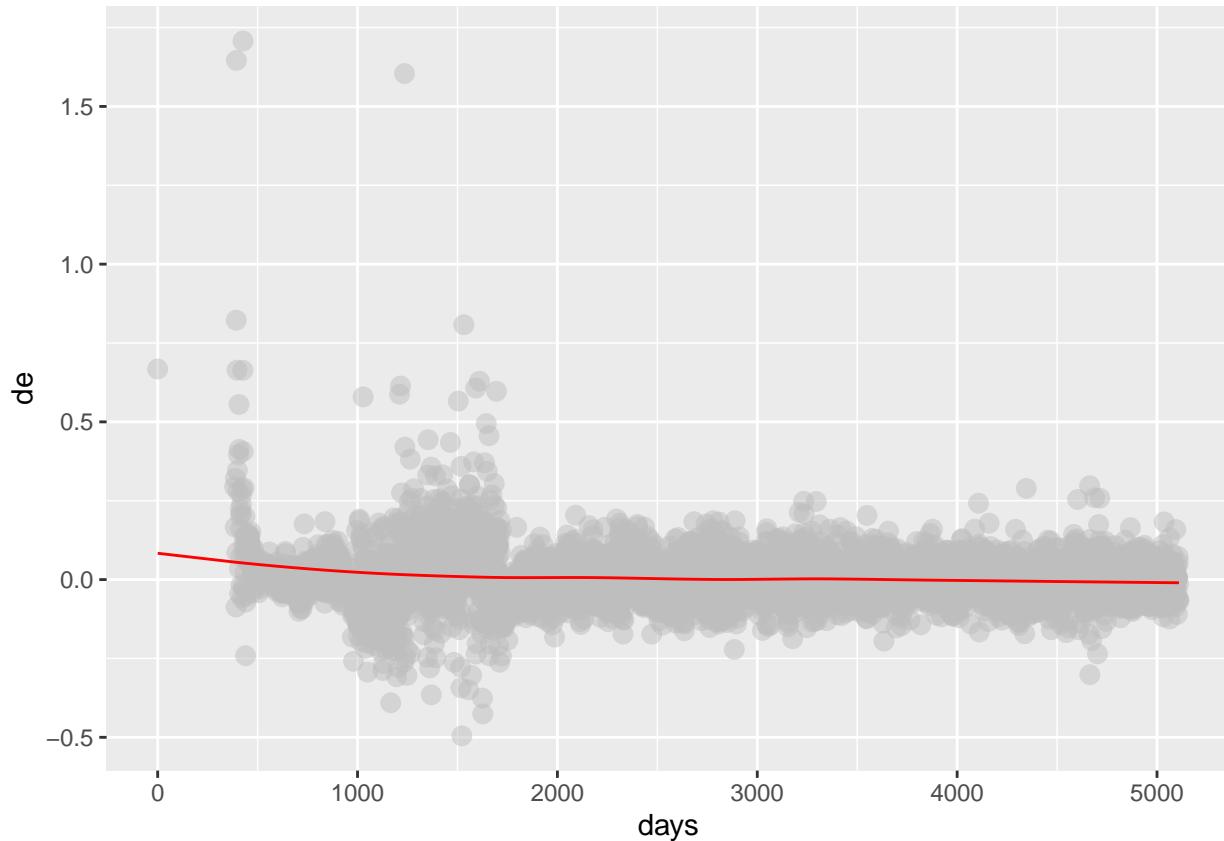
```
cv.UMGYDE.default_params <- train_UMGY_SmoothedDay_effect.cv()

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 ...
```

Below is the visual representation of the UMGYD 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")
```



Let's calculate the *RMSE* of the current model using the `calc_UMGY_SmoothedDay_effect.RMSE.cv` function described above:

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

RMSEs.ResultTibble.UMGYDE <- RMSEs.ResultTibble.rgblr.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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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.

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.



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

2.8.3 UMGYDE Model Tuning by `degree` and `span` Parameters

2.8.3.1 UMGYDE Model Tuning: Support Functions

2.8.3.1.1 `train_UMGY_SmoothedDay_effect.RMSE.cv` Function

The function is designed to tune the `stats::loess` function's parameters (`degree` and `span`), which is ultimately called internally under the hood.

Parameters

- **`degree`:** A parameter to pass to the internally called `train_UMGY_SmoothedDay_effect.cv` function (see the [Details](#) section below) and, ultimately under the hood, to the `stats::loess` to specify the degree of the polynomials to be used.
- **`span`:** A parameter to pass to the internally called `train_UMGY_SmoothedDay_effect.cv` function (see the [Details](#) section below) and, ultimately under the hood, to the `stats::loess` to control the degree of smoothing.

Details

Under the hood, the function internally calls the `train_UMGY_SmoothedDay_effect.cv` and then `calc_UMGY_SmoothedDay_effect.RMSE.cv` described above to compute the RMSE of the model to be tuned.

Source Code

Below is a simplified version of the function's source code:

```
train_UMGY_SmoothedDay_effect.RMSE.cv <- function(degree = NA, span = NA) {  
  train_UMGY_SmoothedDay_effect.cv(degree, span) |>  
  calc_UMGY_SmoothedDay_effect.RMSE.cv()  
}
```



The complete version of the source code of the `train_UMGY_SmoothedDay_effect.RMSE.cv` function is available in the [Tuning loess Parameters](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

Return

The *Root Mean Squared Error (RMSE)* of the *UMGYD Model*.

2.8.3.1.2 `train_UMGY_SmoothedDay_effect.RMSE.cv.degree0` Function

The overloaded version of the `train_UMGY_SmoothedDay_effect.RMSE.cv` function described above with the fixed `degree` parameter value of 0.

Parameters

- `span`: A parameter to pass to the internally called `train_UMGY_SmoothedDay_effect.RMSE.cv` function (see the [Details](#) section below) and, ultimately under the hood, to the `stats::loess` to control the degree of smoothing.

Details

Under the hood, the function internally calls the `train_UMGY_SmoothedDay_effect.RMSE.cv` described above with the fixed `degree` parameter value of 0 and the `span` parameter value taken from the [parameter of the same name](#) of this function to compute the `RMSE` of the model to be tuned.

Source Code

Below is the function's source code:

```
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 [Tuning loess Parameters](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

Return

Root Mean Square Error (RMSE) of the *UMGYD* model trained using the `stats::loess` function called with the given `span` parameter and the `degree` parameter value equal to 0.

2.8.3.1.3 `train_UMGY_SmoothedDay_effect.RMSE.cv.degree1` Function

The overloaded version of the `train_UMGY_SmoothedDay_effect.RMSE.cv` function described above with the fixed `degree` parameter value of 1.

Parameters

- `span`: A parameter to pass to the internally called `train_UMGY_SmoothedDay_effect.RMSE.cv` function (see the [Details](#) section below) and, ultimately under the hood, to the `stats::loess` to control the degree of smoothing.

Details

Under the hood, the function internally calls the `train_UMGY_SmoothedDay_effect.RMSE.cv` described above with the fixed `degree` parameter value of 1 and the `span` parameter value taken from the [parameter of the same name](#) of this function to compute the `RMSE` of the model to be tuned.

Source Code

Below is the function's source code:

```
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 [Tuning loess Parameters](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

Return

Root Mean Square Error (RMSE) of the `UMGYD` model trained using the `stats::loess` function called with the given `span` parameter and the `degree` parameter value equal to 1.

2.8.3.1.4 `train_UMGY_SmoothedDay_effect.RMSE.cv.degree2` Function

The overloaded version of the `train_UMGY_SmoothedDay_effect.RMSE.cv` function described above with the fixed `degree` parameter value of 2.

Parameters

- `span`: A parameter to pass to the internally called `train_UMGY_SmoothedDay_effect.RMSE.cv` function (see the [Details](#) section below) and, ultimately under the hood, to the `stats::loess` to control the degree of smoothing.

Details

Under the hood, the function internally calls the `train_UMGY_SmoothedDay_effect.RMSE.cv` described

above with the fixed `degree` parameter value of 2 and the `span` parameter value taken from the [parameter of the same name](#) of this function to compute the RMSE of the model to be tuned.

Source Code

Below is the function's source code:

```
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 [Tuning loess Parameters](#) section of the `UMGYD-effect.functions.R` script on [GitHub](#).

Return

Root Mean Square Error (RMSE) of the *UMGYD* model trained using the `stats::loess` function called with the given `span` parameter and the `degree` parameter value of 2.

2.8.3.2 UMGYDE Model Tuning: Step 1 (degree = 0)

2.8.3.2.1 Pre-configuration: span Range Determination



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*.

Below the most significant part of the code performing the current model tuning, the *Pre-configuration Step* is provided:

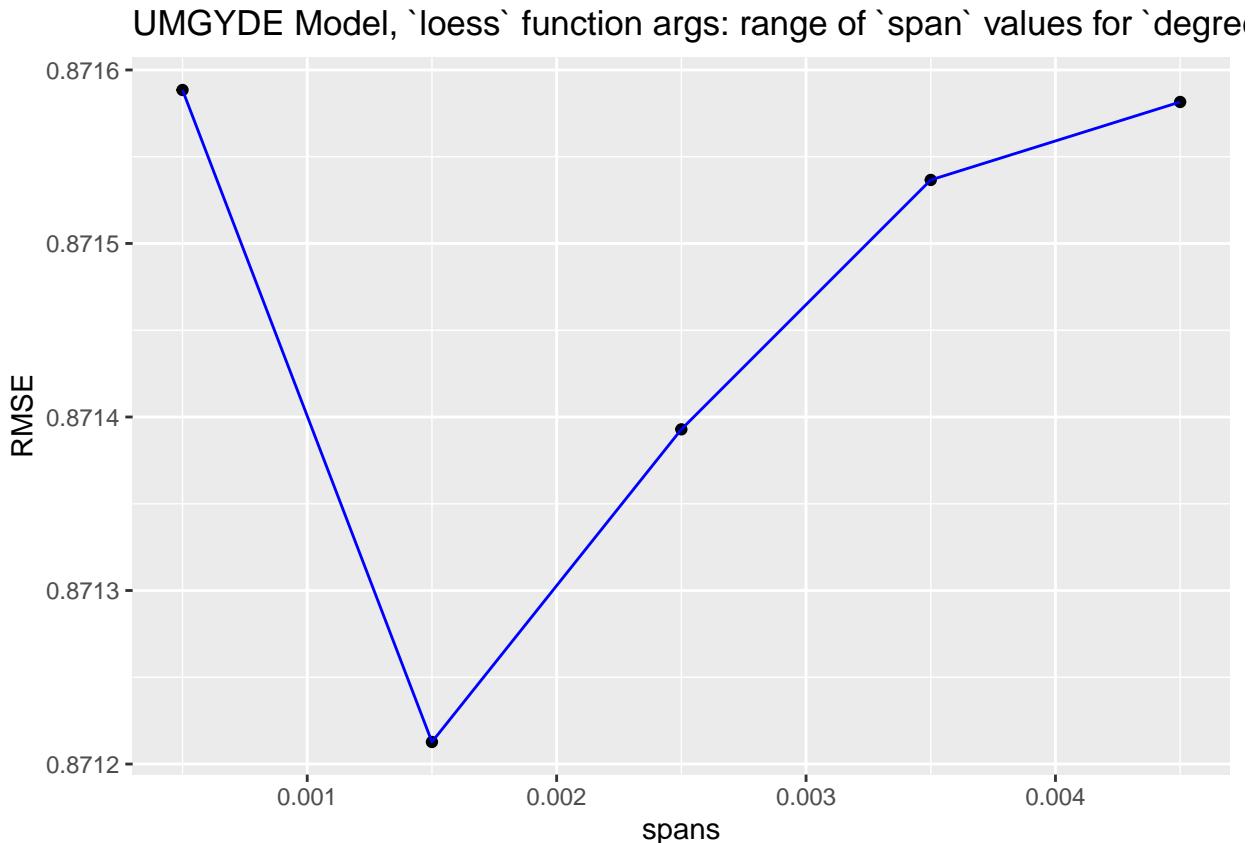
```
spans <- seq(0.0005, 1, 0.001)
lss.UMGYDE.preset.degree0.result <-
  tune.model_param(spans, train_UMGY_SmoothedDay_effect.RMSE.cv.degree0)

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"
```

Below is the visual representation of the `span` parameter values range 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")
```



2.8.3.2.2 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*.

Below the most significant part of the code performing the current model *Fine-tuning* is provided:

```
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)
```

```
lss.fine_tune.loop_starter
```

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

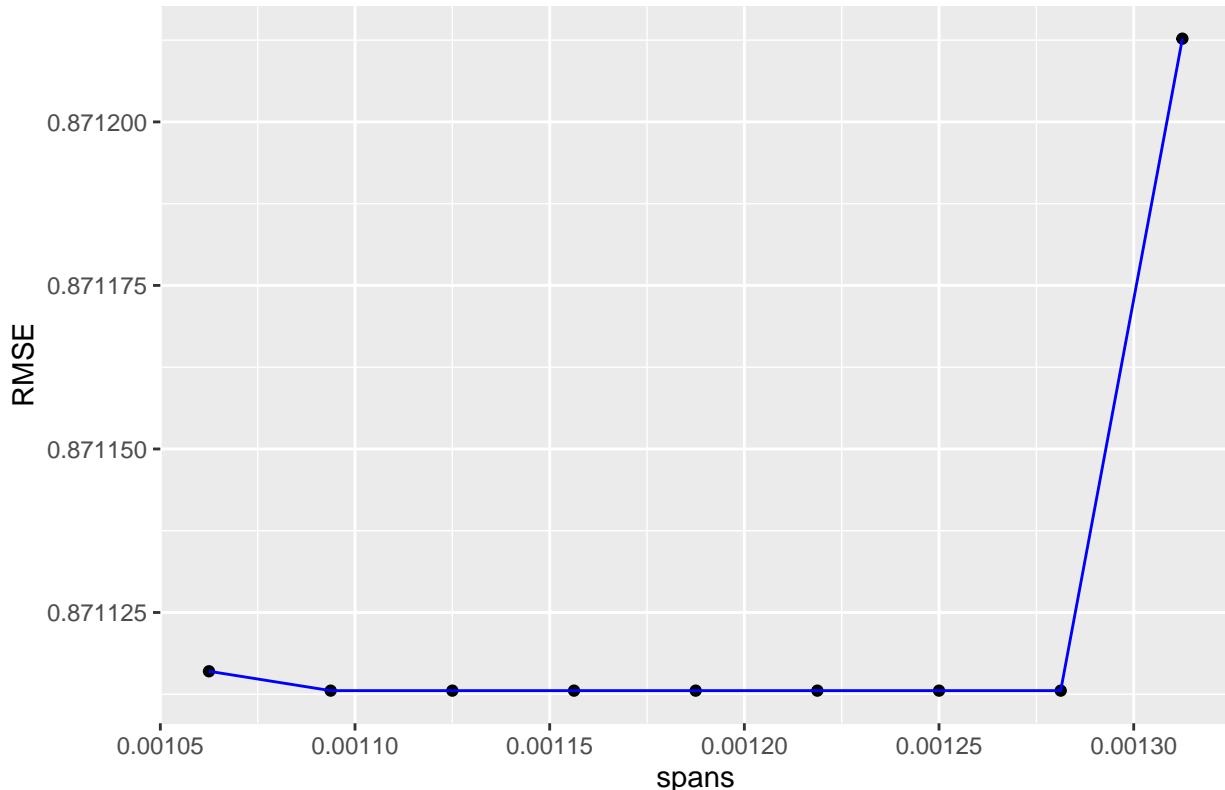
```
str(lss.UMGYDE.fine_tune.degree0.result)
```

```
## 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 ...
```

Below is the visual representation of the current model *Fine-tuning* results for the parameter value: `degree = 0`:

```
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")
```

Fine–tuned UMGYDE Model with `loess` parameter: `degree` = 0



```
rm=plt.title)
```

Now, we can add the *UMGYDE Model Tuning* result for `degree = 0` to our *Result Table*:

```
RMSEs.ResultTibble.UMGYDEO <- 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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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 'loess' function with default 'degree' & 'span' parameters.
Tuned UMGYDE.d0 Model	0.8711131	UMGYDE Model computed using function call: 'loess(degree = 0, span = 0.00109375)'

2.8.3.3 UMGYDE Model Tuning: Step 2 (`degree = 1`)

2.8.3.3.1 Pre-configuration: span Range Determination



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 the most significant part of the code performing the current model tuning, the *Pre-configuration Step* is provided:

```
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"

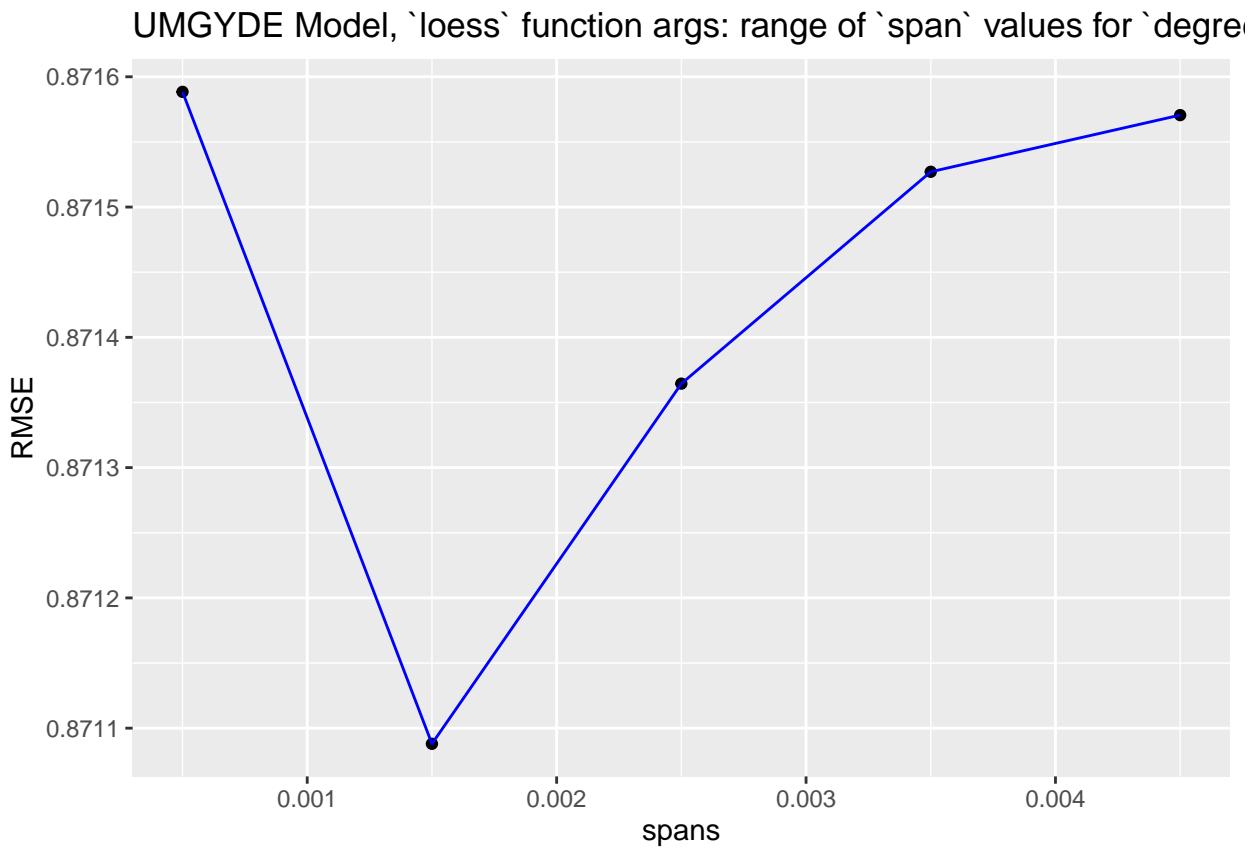
```

Below is the visual representation of the `span` parameter values range 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`",
            xlabel = "parameter.value",
            ylabel = "RMSE",
            xname = "spans",
            yname = "RMSE")

```



2.8.3.3.2 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](#).

Below the most significant part of the code performing the current model *Fine-tuning* is provided:

```

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)

```

```

lss.fine_tune.loop_starter
```

```

## [1] 0.0005 0.0025 8.0000
```

```

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

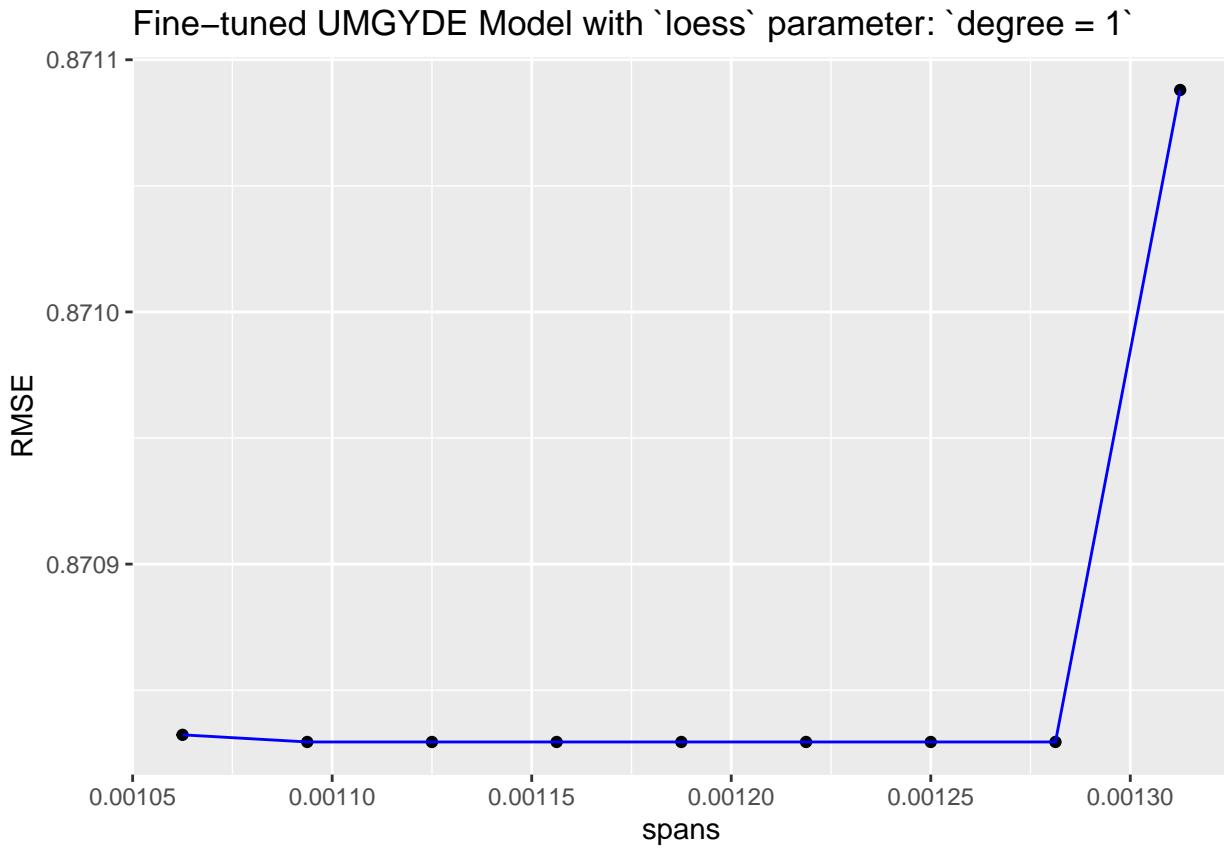
```

## 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 ...
```

Below is the visual representation of the current model *Fine-tuning* results for the parameter value: degree = 1:

```

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")
```



Now, we can add the *UMGYDE Model Tuning* result for `degree = 1` to our *Result Table*:

```
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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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 ‘loess’ function with default ‘degree’ & ‘span’ parameters.
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)’

2.8.3.4 UMGYDE Model Tuning: Step 3 (`degree = 2`)

2.8.3.4.1 Pre-configuration: `span` Range Determination



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 the most significant part of the code performing the current model tuning, the *Pre-configuration Step* is provided:

```
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)

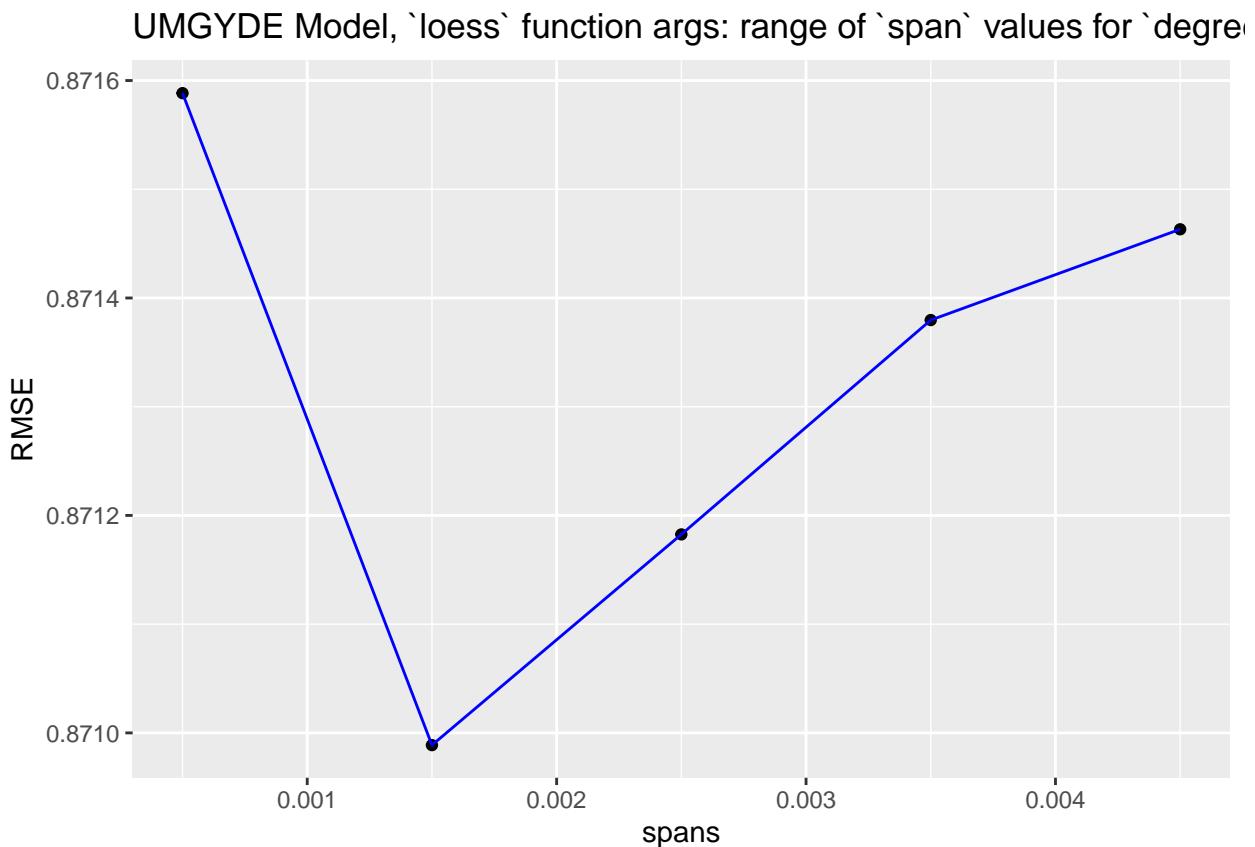
## List of 2
## $ tuned.result:'data.frame':   5 obs. of  2 variables:
##   ..$ RMSE          : num [1:5] 0.872 0.871 0.871 0.871 0.871
##   ..$ parameter.value: num [1:5] 0.0005 0.0015 0.0025 0.0035 0.0045
## $ best_result : Named num [1:2] 0.0015 0.871
##   ..- attr(*, "names")= chr [1:2] "param.best_value"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

Below is the visual representation of the `span` parameter values range 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")

```



2.8.3.4.2 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](#).

Below the most significant part of the code performing the current model *Fine-tuning* is provided:

```

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)

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 ...

```

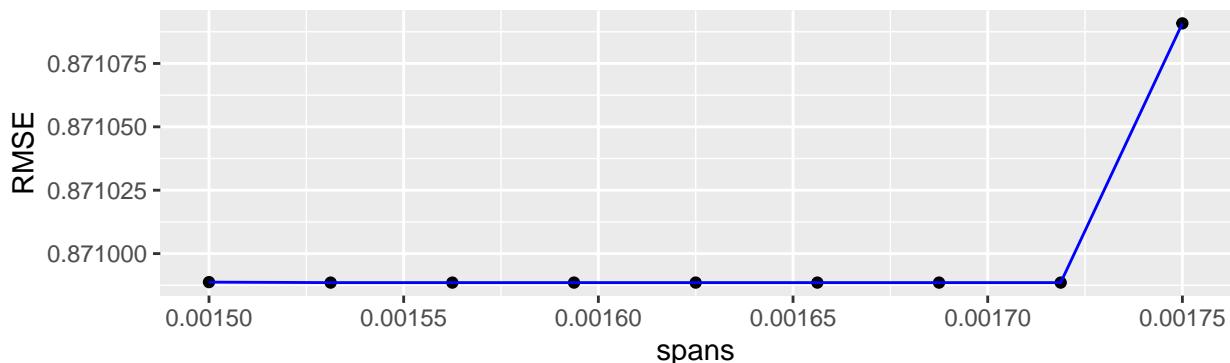
Below is the visual representation of the current model *Fine-tuning* results for the parameter value: `degree = 2`:

```

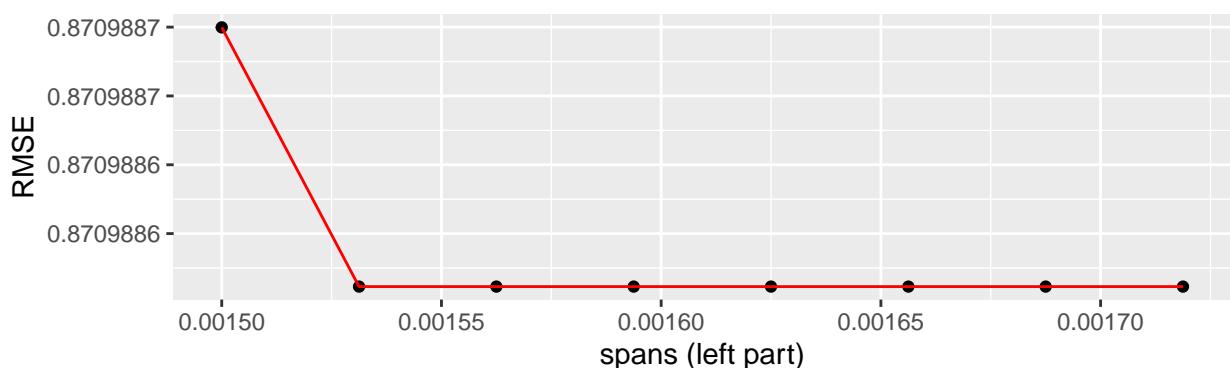
lss.UMGYDE.fine_tune.degree2.result$tuned.result |>
  data.plot.left_detailed(title = "Fine-tuned UMGYDE Model with `loess` parameter: `degree = 2`",
                         title.left = "Left Part of the Chart Above (Zoomed in)",
                         left.n = 8,
                         xname =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

```

Fine-tuned UMGYDE Model with `loess` parameter: `degree = 2`



Left Part of the Chart Above (Zoomed in)



The custom data visualization function `data.plot.left_detailed` we used above is described in the [Data Helper Functions](#) section of the [Appendix](#) to this report.

Now, we can add the *UMGYDE Model Tuning* result for `degree = 2` to our *Result Table*:

```
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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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 ‘loess’ function with default ‘degree’ & ‘span’ parameters.
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)’

2.8.3.5 UMGYDE Model Tuning: Re-training with the Best degree & span Parameters



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 calculate the *Regularized UMGYDE Effect* by retraining our model on the entire `edx` dataset with the best values of the `degree` and `span` parameters we figured out above, for the definitive *Root Mean Squared Error* calculation and use in subsequent processing.

The most significant part of the code that performs this operation is shown below:

```

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)

lss.UMGYDE.best_params

##      degree      span      RMSE
## 1 1.00000000 0.00109375 0.87082947

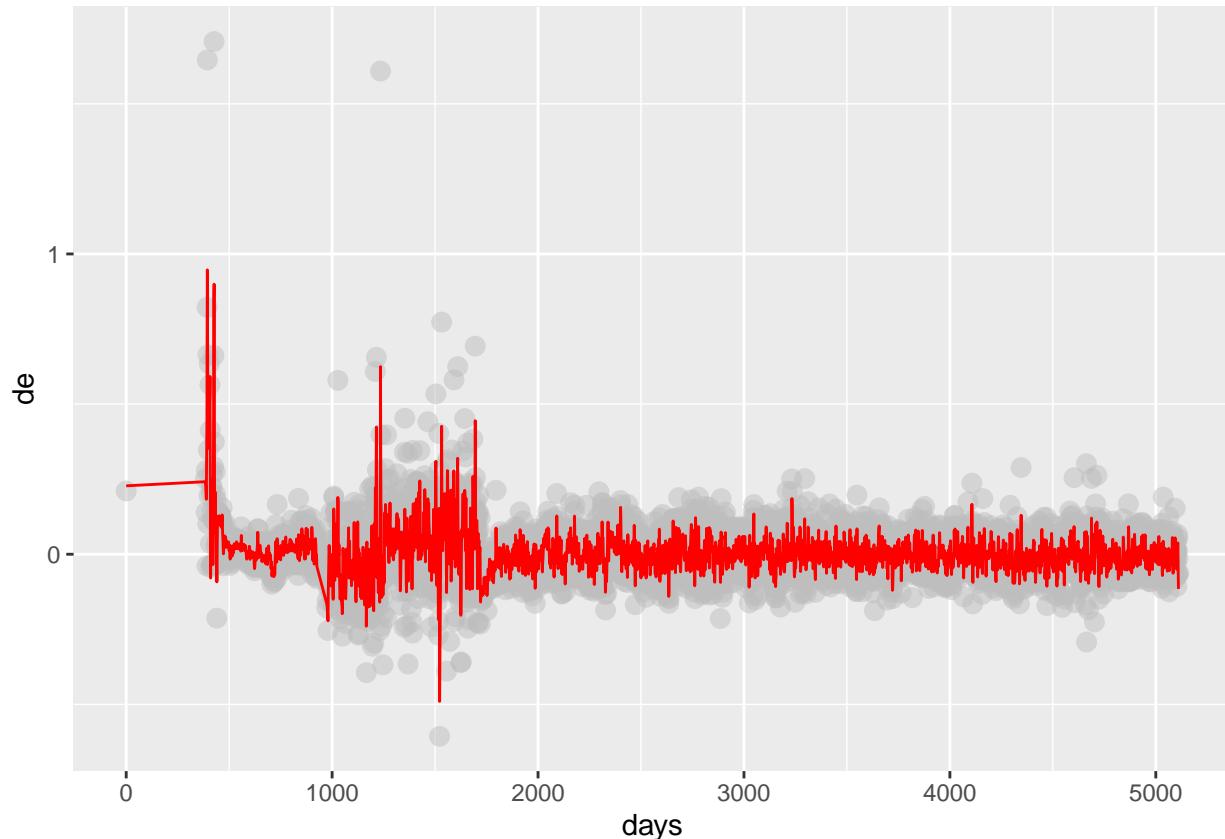
```



Here we use the helper function `train_UMGY_SmoothedDay_effect` described above in the [UMGYDE Model: Helper Functions](#) section.

Let's now visualize final Smoothed Day effect we have 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 before, although the data now provides better results.

We calculate the *Root Mean Squared Error* for the ultimately computed *UMGYDE Effect* using the helper function `calc_UMGY_SmoothedDay_effect.RMSE.cv` described above in the [UMGYDE Model: Helper Functions](#) section above as follows:

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

Finally, we add the definitive result for the current model to our *Result Table*:

```
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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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 'loess' function with default 'degree' & 'span' parameters.
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.

Finally, let's regularize our tuned model as we did with the previous ones.

2.8.4 UMGYDE Model Regularization

2.8.4.1 UMGYDE Model Regularization: Mathematical Description

We have already explained the idea of *Linear Model Regularization* in the [UME Model Regularization](#) section above. 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 (??) for the *UMGYDE Model*. For the current model, this formula takes the form:

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

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

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

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

As with the previous models, we implement the *Regularization* method for the current model in the following three steps:

1. **Pre-configuration:** Preliminary determination of the optimal range of λ values for the [5-Fold Cross Validation](#) samples;
2. **Fine-tuning:** figuring out the value of λ that minimizes the model's RMSE.
3. **Retraining:** retraining the model with the best value of the parameter λ obtained in the previous step.

2.8.4.2 UMGYDE Model Regularization: Helper Functions

2.8.4.2.1 `regularize.train_UMGYD_effect` Function

The function is an overloaded version of the `train_UMGY_SmoothedDay_effect` function described above, with the fixed values of the `degree` and `span` parameters set to the best values figured out during the tuning process described in the [UMGYDE Model Tuning] section above.

Parameters

- **`train_set`:** The *Train Set* dataset .
- **`lambda`:** In accordance with the description in the [UMGYDE Model Regularization: Mathematical Description](#) section above, this is the *Regularization parameter* λ used for the *UMGYDE Model Regularization* process.

Source Code

The following is the source code of the function:

```
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](#).

Return

A data frame object representing the *UMGYD Effect*.

2.8.4.2.2 `regularize.train_UMGYD_effect.cv` Function



The source code of the `regularize.train_UMGYD_effect.cv` function is defined in the [Regularization](#) section of the [UMGYD-effect.functions.R](#) script on [GitHub](#).

The function is an overloaded version of the `train_UMGY_SmoothedDay_effect.cv` function described above, with the fixed values of the degree and span parameters set to the best values figured out during the tuning process described in the [UMGYDE Model Tuning] section above.

Parameters

- ***lambda***: In accordance with the description in the [UMGYDE Model Regularization: Mathematical Description](#) section above, this is the *Regularization parameter* λ used for the *UMGYDE Model Regularization* process.

Source Code

The source code of the function is provided 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)  
}
```



Note that we reuse the function `train_UMGY_SmoothedDay_effect.cv` calling it internally from the `regularize.train_UMGYD_effect.cv`, but now with the λ parameter different from the default ('lambda = 0') value.

Return

A data frame object representing the *UMGYD Effect*.

2.8.4.2.3 `regularize.test_lambda.UMGYD_effect.cv` Function

The function calculates *RMSE* of the *UMGYDE Model* using *5-Fold Cross Validation* for the given λ parameter value.

- ***lambda:*** In accordance with the description in the [UMGYDE Model Regularization: Mathematical Description](#) section above, this is the *Regularization parameter* λ used for the *UMGYDE Model Regularization* process.

Details

Under the hood, the function internally calls the `regularize.train_UMGYD_effect.cv` and then `calc_UMGY_SmoothedDay_effect.RMSE.cv` described above, and returns the *RMSE* value that the latter inner function calculates for the given `lambda` parameter value passed to the former one.

Source Code

The source code of the function is provided below:

```
regularize.test_lambda.UMGYD_effect.cv <- function(lambda){  
  if (is.na(lambda)) {  
    stop("Function: regularize.test_lambda.UMGY_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*.

Return

Root Mean Squared Error (RMSE) of the *UMGYD Model* trained for the given lambda parameter value.

2.8.4.3 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](#).

Let's perform the preconfiguration to determine the appropriate range of λ for subsequent fine-tuning of our current model:

We are going to use the `tune.model_param` function described in the Regularization: Common Helper Functions section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UMGYD_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that 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"

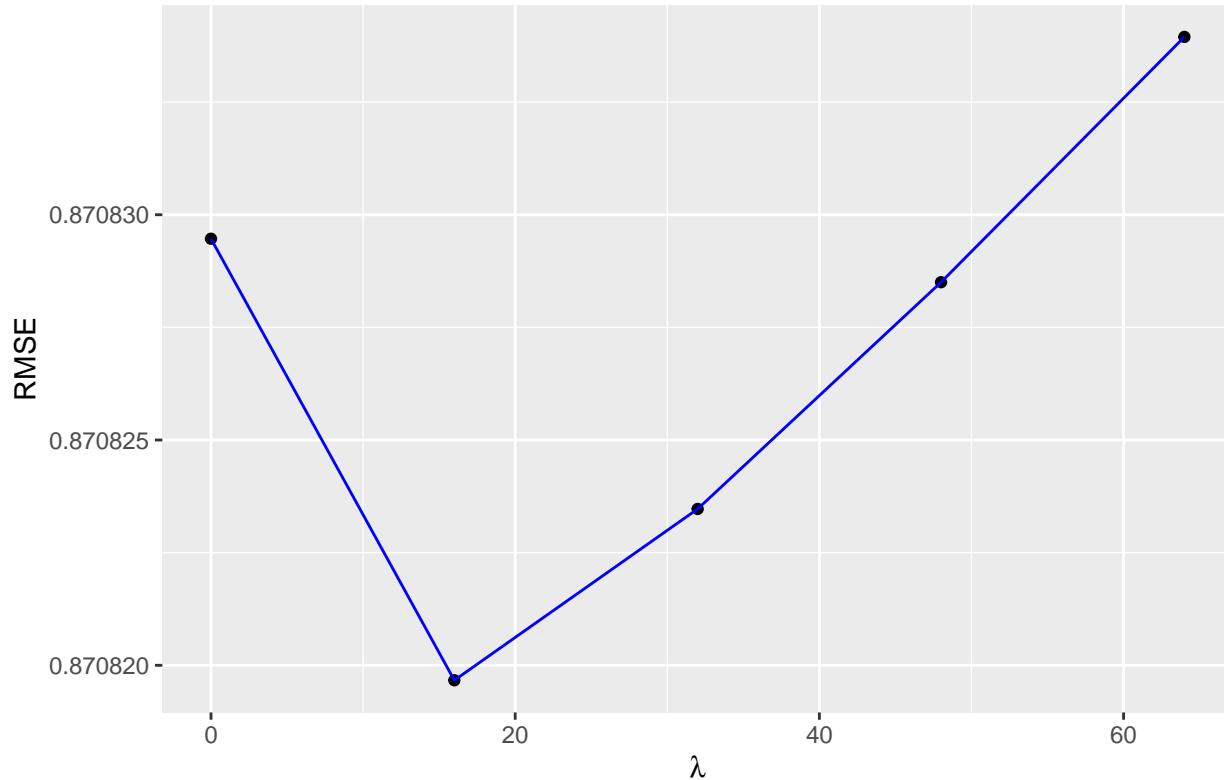
cv.UMGYDE.preset.result$best_result

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

Now, let's visualize the results of the λ range preconfiguration:

```
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")
```

UMGYDE Model Regularization: λ Range Pre-configuration



We use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

2.8.4.4 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 λ parameter.

Here we are going to use the `model.tune.param_range` function described in the Regularization: Common Helper Functions section of the [Appendix](#) to this report, passing the `regularize.test_lambda.UMGYD_effect.cv` function as the value of the `fn_tune.test.param_value` parameter.

Below we provide the most significant part of the code that performs this operation:

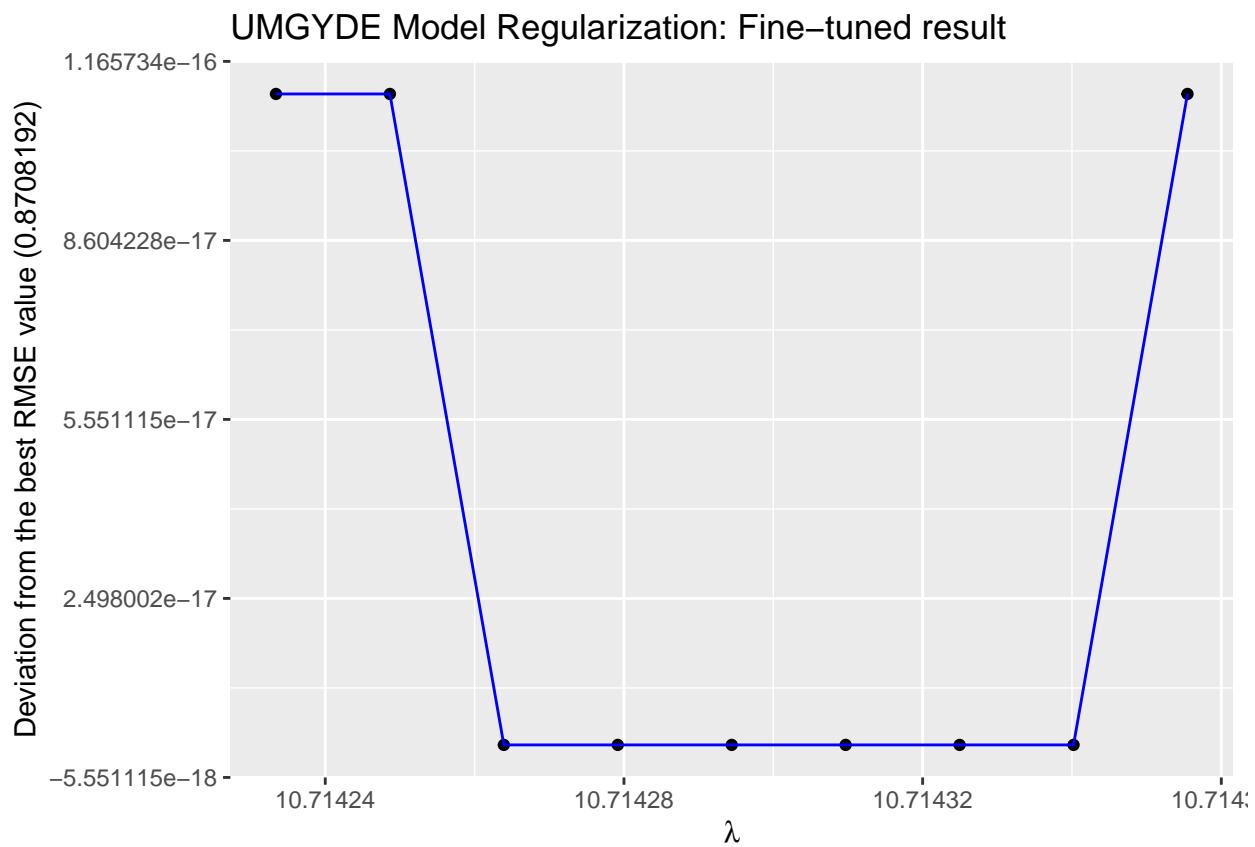
```
endpoints <-  
  get_fine_tune.param.endpoints(cv.UMGYDE.preset.result$tuned.result)  
  
UMGYDE.loop_starter <- c(endpoints["start"],  
                         endpoints["end"],  
                         8)  
  
cache.base_name <- "UMGYDE.rglr.fine-tuning"  
  
UMGYDE.rglr.fine_tune.results <-  
  model.tune.param_range(UMGYDE.loop_starter,  
                         UMGYDE.rglr.fine_tune.cache.path,  
                         cache.base_name,  
                         regularize.test_lambda.UMGYD_effect.cv)  
  
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 ...  
  
UMGYDE.rglr.fine_tune.results$best_result  
  
## param.best_value      best_RMSE  
##          10.7142639    0.8708192
```



Under the hood, this code block internally calls the helper function `get_fine_tune.param.endpoints` described in the Regularization: Common Helper Functions section of the [Appendix](#) to this report to figure out the boundaries of the range of values from the nearest neighborhood of the value corresponding to the best RMSE in the data passed in as a parameter.

Let's visualize the fine-tuning results:

```
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)
```



We use the custom data visualization function `data.plot` described in the [Data Helper Functions](#) section of the [Appendix](#) to this report, which is defined in the [Data Visualization](#) section of the `data.helper.functions.R` script on [GitHub](#).

2.8.4.5 UMGYDE Model Regularization: Retraining Model with the best λ



The complete version of the source code shown in this section is available in the [UMGYDE Regularized Model: Retraining with the best params](#) section of the `capstone-movielens.main.R` script on [GitHub](#).

Now, we can compute the *Regularized UMGYDE Effect* by re-training our model on the entire `edx` dataset with the best value of the λ parameter we have just figured out and use it in subsequent analysis.

The most significant part of the code for this operation is shown below:

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

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

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

str(rgldr.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 ...
```



Here we use the helper function `regularize.train_UMGYD_effect` described above in the [UMGYDE Model: Helper Functions](#) section.

We calculate the *Root Mean Squared Error* for the ultimately computed *Regularized UMGYDE Effect* using the helper function `calc_UMGY_SmoothedDay_effect.RMSE.cv` described above in the [UMGYDE Model: Helper Functions](#) section as follows:

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

As always, we add the definitive result for the current model to our *Result Table*:

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

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

Method	RMSE	Comment
Project Objective	0.8649000	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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 ‘loess’ function with default ‘degree’ & ‘span’ parameters.
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.



Now, we have a better result than the ones for the previous models, but still insufficient to meet the Project Objective. Let’s see what else we can do to achieve our ultimate goal.

2.8.4.6 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](#) 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. Firstly, we compute predictions for our current model:

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

```
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: num  4.49 5 4.63 3.3 4.07 ...
```



Here we use the helper function `UMGY_SmoothedDay_effect.predict` described above in the [UMGYDE Model: Helper Functions](#) section.

Then we compute the *RMSE* and add the result to our *Result Table*:

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

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.")
```



Here we use the helper function `rmse2` to calculate the *RMSE* that is defined in the [\(R\)MSE-related functions](#) section of the `common-helper.functions.R` script on [GitHub](#).

And finally, we print out the results:

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

Method	RMSE	Comment
Project Objective	0.8649000	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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 ‘loess’ function with default ‘degree’ & ‘span’ parameters.
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.



As expected, we have not yet achieved a result that meets our Project Objective.

As explained in the [Chapter 24 Matrix Factorization](#) of the *Course Textbook*, so far our models “*ignore an important source of information related to the fact that groups of movies have similar rating patterns and groups of users have similar rating patterns as well...*” [15]

Then the author shows that in this case, the *Matrix Factorization* method can considerably improve our results. In the next section, we will apply this method to our current model and see what we get.

2.9 UMGYDE Model: Matrix Factorization (MF)

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

2.9.1 MF: Mathematical Description

As outlined in [this article](#)[16], 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

$$\mathbf{R} \approx \mathbf{P}\mathbf{Q}^T \quad (15)$$

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

$$\mathbf{R} \sim \hat{\mathbf{R}} + \mathbf{P}\mathbf{Q}^T + \varepsilon \quad (16)$$

where:

- \mathbf{R} is the $U_m \times M_n$ 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 [17, 18]:

$$\min_{P,Q} \sum_{(u,v) \in R} [f(\mathbf{p}_u, \mathbf{q}_v; \mathbf{r}_{u,v}) + \mu_P \|\mathbf{p}_u\|_1 + \mu_Q \|\mathbf{q}_v\|_1 + \frac{\lambda_P}{2} \|\mathbf{p}_u\|_2^2 + \frac{\lambda_Q}{2} \|\mathbf{q}_v\|_2^2]$$

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.[16]

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.[16]

2.9.2 MF: Helper Functions

2.9.2.1 `mf.residual.dataframe` Function

2.9.2.1.1 Description

Computes residuals, defined as the difference between the observed and predicted values of the fully trained *UMGYDE Model* as described in the [User+Movie+Genre+Year+Day Effect \(UMGYDE\) Model](#) section.

2.9.2.1.2 Parameters

train_set

The dataset to be trained.

2.9.2.1.3 Source Code

The following is the source code of the function:

```
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*.

2.9.2.1.4 Return

A data frame object containing the residuals (see the [Description](#) above for the details).

2.9.2.2 UMGYDE_model.predict Function

This is a wrapper for the `UMGY_SmoothedDay_effect.predict` function described in the [UMGYDE Model: Helper Functions](#) section above with the fixed `day_smoothed_effect` parameter value obtained after the *UMGYDE Model* (described in the [User+Movie+Genre+Year+Day Effect \(UMGYDE\) Model](#) section) has been fully trained.

2.9.2.2.1 Parameters

test_set

The dataset for testing the *UMGYDE Model* after it has been fully trained.

2.9.2.2.2 Source Code

The following is the source code of the function:

```
UMGYDE_model.predict <- function(test_set) {  
  test_set |>  
    UMGY_SmoothedDay_effect.predict(rglr.UMGYD_effect)  
}
```



The source code of the `UMGYDE_model.predict` function is also available in the [MF.functions.R](#) script on *GitHub*.

2.9.2.2.3 Return

A data frame object containing predicted values.

2.9.3 MF: Performing Operation



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 is going to be the last one to provide sufficient results, so we will use the entire `edx` dataset as a *Training Set* and `final_holdout_test` as a *Test Set* for our final model, skipping the intermediate *train/test* operations on the split *train/validation* datasets.

2.9.3.1 MF: Getting Residuals From the UMGYDE Model Prediction Values

First, we get residuals from the *UMGYDE Model* prediction values using the `mf.residual.dataframe` function described above:

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

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

```
## 'data.frame': 9000055 obs. of 3 variables:  
## $ userId : int 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
```

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

Now, let's convert the residuals data and `final_holdout_test` dataset to the input data compatible with the `recosystem` package to process:

```
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))
```

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

```
## Formal class 'DataSource' [package "recosystem"] with 3 slots  
## ..@ source:List of 3  
## ...$. : int [1:9000055] 1 1 1 1 1 1 1 1 1 ...  
## ...$. : 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"

```

2.9.3.3 MF: Creating and Tuning the Reco Object

We then create a `reco` object of the `Reco` class and tune it using the `reco$tune` method, passing to it as a `train_data` parameter residual data prepared in the previous step:

```

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))

```

```
str(reco)
```

```

## Reference class 'RecoSys' [package "recosystem"] with 2 fields
## $ model      :Reference class 'RecoModel' [package "recosystem"] with 5 fields
##   ..$ path     : chr ""
##   ..$ nuser    : int 0
##   ..$ nitem    : int 0
##   ..$ nfac     : int 0
##   ..$ matrices: list()
##   ..and 16 methods, of which 2 are possibly relevant:
##   .. initialize, show#envRefClass
## $ train_pars: list()
## and 19 methods, of which 5 are possibly relevant:
##   output, predict, show#envRefClass, train, tune

```

```
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] 0.807 0.812 0.821 0.86 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"

```

2.9.3.4 MF: Final Training

Finally, we train the model for the last time using the `reco$train` method, passing to it as a `train_data` parameter the same residual data we used for the tuning:

```

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

```

2.9.3.5 MF: Final Holdout Test

From now on, we will work with the `final_holdout_test` dataset for the final testing of our model, obtaining final predictions, and computing the *Root Mean Squared Error*.

First, we get the final predictions combining the predicted residuals from the `reco` objects and predicted values we lastly get from the *UMGYDE Model* as described in the [UMGYDE Model: Final Holdout Test \(Preliminary Assessment\)](#) section:

```

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

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

str(mf.reco.residual)

##  num [1:999999] 0.107 -0.264 0.128 0.245 0.682 ...

str(mf.predicted_ratings)

##  num [1:999999] 4.59 4.74 4.76 3.55 4.76 ...

```

Next, we compute the *RMSE* using the `rmse2` custom helper function, already mentioned above in the [UMGYDE Model: Final Holdout Test \(Preliminary Assessment\)](#) section:

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



The function `rmse2` is defined in the [\(R\)MSE-related functions](#) section of the [common-helper.functions.R](#) script on *GitHub*.

And add the result to our *Result Table*:

```
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	
Overall Mean Rating Model	1.0603462	
User Effect Model	0.9697962	
User+Movie Effect Model	0.8732081	
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 'loess' function with default 'degree' & 'span' parameters.
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)	0.7875645	Matrix Factorization of the Best Model Residuals, Final Holdout Test



Finally, we have reached our ultimate goal, since the *Root Mean Squared Error* we obtained has achieved the required *Project Objective*.

3 Conclusion

In this project, a *Machine Learning Algorithm* was successfully developed for a *Recommender System* (hereinafter 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 the [Overall Mean Rating \(OMR\) Model](#) section.

The following effects were taken into account:

- **User:** based on which the *User Effect (UE) Model* described in the [User Effect \(UE\) Model](#) section 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 the [User+Movie Effect \(UME\) Model](#) section 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 the [User+Movie+Genre Effect \(UMGE\) Model](#) section 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 the [User+Movie+Genre+Year Effect \(UMGYE\) Model](#) section was built (provided a much better improvement in *RMSE* compared to the previous (*UMGE*) model but still two orders of magnitude less than the *UE* and *UME* models showed);
- **Day** based on which (and the previous effects) the *Regularized (and Tuned) User+Movie+Genre+Year+Day Effect (UMGYDE) Model* described in the [User+Movie+Genre+Year+Day Effect \(UMGYDE\) Model](#) section was built (showed almost twice better improvement than the previous (*UMGYE*) model but still not good enough to meet the *Project Objective*);
- **Day (Matrix Factorization):** (described in the [UMGYDE Model: Matrix Factorization \(MF\) section](#)) *Matrix Factorization* method performed on the *Regularized UMGYDE Model* provided substantial improvement comparable to that shown by the *UE* and *UME* models, and finally permitted to reach the *Project Objective*.

The *Result Summary Table* constructed below 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	
Overall Mean Rating Model	1.0603462	-0.1954462	
User Effect Model	0.9697962	0.0905500	
User+Movie Effect Model	0.8732081	0.0965881	
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 ‘loess’ function with default ‘degree’ & ‘span’ parameters.
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 call 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)	0.7875645	0.0928580	Matrix Factorization of the Best Model Residuals, F Holdout Test

4 Appendix

4.1 Data Helper Functions



The complete source code of the functions described in this section is available in the [data.helper.functions.R](#) script on *GitHub*.

4.1.1 `union_cv_results` Function

Aggregates the input data frames into a single data frame.

4.1.1.1 Usage

```
union_cv_results(data_list = data)
```

4.1.1.2 Parameters

- **data_list:** List of data frames representing the results of the *N-Fold Cross Validation* method execution.;

4.1.1.3 Details

The function is used to aggregate the *N-Fold Cross Validation* method result data into a single data frame for further processing.

4.1.1.4 Value

A data frame that is a union of the input data frames.

4.1.1.5 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*.

4.1.2 `data.plot` Function

Plots a graph based on the dataset passed in the `data` parameter, using both points and a line.

4.1.2.1 Usage

```
data.plot(
  data,
  title,
  xname,
  yname,
  xlabel = NULL,
  ylabel = NULL,
  line_col = "blue",
  normalize = FALSE)
```

4.1.2.2 Parameters

- **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 = NULL:**** The *x* axis label. If **NULL**, the value of the **xname** parameter is used for the label;
- **ylabel = NULL:** The *y* axis label. If **NULL**, the value of the **yname** parameter is used for the label.;
- **line_col = blue:** The line color;
- **normalize = FALSE:** If **TRUE**, the deviation of *y* from its mean is used to plot, rather than *y* values. Otherwise, the value of *y* is used.

4.1.2.3 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.

4.1.2.4 Source Code

The source code of the `[data.plot]`((<https://github.com/AzKurban-edX-DS/Capstone-MovieLens/blob/main/r/src/support-functions/data.helper.functions.R#L592>)) 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*.

4.1.3 `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).

4.1.3.1 Usage

```
data.plot.left.n <- function(data,
                                left.n = 0,
                                title,
                                xname,
                                yname,
                                xlabel,
                                ylabel,
                                line_col = "red",
                                normalize = FALSE)
```

4.1.3.2 Parameters

- **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.
- **xlabel = NULL:**** The *x* axis label. If `NULL`, the value of the `xname` parameter is used for the label.
- **ylabel = NULL:** The *y* axis label. If `NULL`, the value of the `yname` parameter is used for the label.
- **line_col = blue:** The line color.
- **normalize = FALSE:** If `TRUE`, the deviation of *y* from its mean is used to plot, rather than *y* values. Otherwise, the value of *y* is used.

4.1.3.3 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.

4.1.3.4 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](#).

4.1.4 `data.plot.left_detailed` Function

Combines two graphs plotted by the internally called helper functions in a grid: the first is a standard graph plotted by the `data.plot` function described above, the second is a graph based on the first n rows (the n is

specified by the `left.n` parameter) of the dataset plotted by the `data.plot.left.n` function also described above (The data for both graphs is passed in the `[data].(#func.data.plot.left.n.params.data)` parameter).



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).

4.1.4.1 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)
```

4.1.4.2 Parameters

- **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 = NULL:**** The *x* axis label for the first plot. If `NULL`, the value of the `xname` parameter is used for the label.
- **xlabel2 = NULL:**** The *x* axis label for the second plot. If `NULL`, the value of the `xname` parameter is used for the label.

- **ylabel1 = NULL:** The y axis label for the first plot. If `NULL`, the value of the `yname` parameter is used for the label.
- **ylabel2 = NULL:** The y axis label for the second plot. If `NULL`, the value of the `yname` parameter is used for the label.
- **line_col = blue:** The line color.
- **normalize = FALSE:** If `TRUE`, the deviation of y from its mean is used to plot, rather than y values. Otherwise, the value of y is used.

4.1.4.3 Details

Under the hood, the function calls internally the `data.plot` and `data.plot.left.n` functions (described above) to build two graphs and combines them into a grid.

4.1.4.4 Source Code

The source 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](#).

4.2 Common Helper Functions

4.2.1 Common Helper Functions: Result RMSEs Table



The complete source code of the functions described in this section are available in the [Model Tuning](#) section of the [common-helper.functions.R](#) script on *GitHub*.

4.2.1.1 **RMSEs.** Function



The complete source code of this function is available in the [RMSEs Result Tibble](#) section of the [common-helper.functions.R](#) script on *GitHub*.

4.2.1.1.1 Usage

4.2.1.1.2 Parameters

- *RMSEs:* Result (RMSEs) Table to modify;

• : ;

• : ;

• : ;

• : ;

• : ;

• : ;

• : ;

4.2.1.1.3 Details

4.2.1.1.4 Source Code

The soure code of the function is provided below:

4.2.1.1.5 Return The modified *Resut (RMSEs) Table*.

4.2.1.2 RMSEs.AddRow Function



The complete source code of this function is available in the [RMSEs Result Tibble](#) section of the [common-helper.functions.R](#) script on *GitHub*.

4.2.1.2.1 Usage

```
RMSEs . AddRow(RMSEs,  
                 method,  
                 value,  
                 comment = "",  
                 before = NULL)
```

4.2.1.2.2 Parameters

- **RMSEs:** *Resut (RMSEs) Table* to modify;
- **method:** The *Method or Model* being evaluated;
- **value:** The *Root Mean Squared Error (RMSE)* value;
- **comment:** Detailed information about the record or any other comment (if needed);
- **before:** One-based row index where to add the new rows, if NULL (default): after last row;

4.2.1.2.3 Source Code

The soure code of the function is provided below:

```

RMSEs.AddRow <- function(RMSEs,
                           method,
                           value,
                           comment = "",
                           before = NULL){
  RMSEs |>
    add_row(Method = method,
            RMSE = value,
            Comment = comment,
            .before = before)
}

```

4.2.1.2.4 Return The modified *Resut (RMSEs) Table*.

4.2.1.3 RMSEs.AddDiffColumn Function



The complete source code of this function is available in the [RMSEs Result Tibble](#) section of the [common-helper.functions.R](#) script on *GithHub*.

Adds a calculated column named *Diff* to the *Result(RMSEs)* table (passed in the **RMSEs** parameter) 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.

4.2.1.3.1 Usage

```

RMSEs.ModifiedTable <- RMSEs.AddDiffColumn(RMSEs)

```

4.2.1.3.2 Parameters

- **RMSEs:** *Resut (RMSEs) Table* to modify;

4.2.1.3.3 Source Code

The soure code of the function is provided below:

```

RMSEs.AddDiffColumn <- function(RMSEs){
  RMSEs.Diff <- RMSEs |>
    RMSEs.AddRow(NULL, project_objective, before = 1) #/|
    #RMSEs.AddRow(NULL, 0, before = 1)

  RMSEs.Diff <- RMSEs.Diff[-nrow(RMSEs.Diff),]
}

```

```

RMSEs |>
add_column(Diff = RMSEs.Diff$RMSE - RMSEs$RMSE,
            .before = "Comment")

}

```



Under the hood, the function calls internally the `RMSEs.AddRow` function described above.

4.2.1.3.4 Return The modified *Resut (RMSEs) Table*.

4.2.1.4 `RMSE_kable` Function



The complete source code of this function is available in the `RMSEs Result Tibble` section of the `common-helper.functions.R` script on *GitHub*.

Prints the *Resut (RMSEs) Table*.



It is assumed that the table has the following three columns:

1. *Method*;
2. *RMSE*;
3. *Comment*;

4.2.1.4.1 Usage

```

RMSE_kable <- function(RMSEs,
                        col1width = 15,
                        col2width = 5,
                        col3width = 30)

```

4.2.1.4.2 Parameters

- ***RMSEs***: *Resut (RMSEs) Table* to print;
- ***col1width***: Width of the first column;
- ***col2width***: Width of the second column;
- ***col3width***: Width of the third column;

4.2.1.4.3 Source Code

The source code of the function is provided 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))
}
```

4.2.1.5 RMSE.Total_kable Function



The complete source code of this function is available in the [RMSEs Result Tibble](#) section of the [common-helper.functions.R](#) script on *GitHub*.

Prints the *Resut (RMSEs) Table*.



It is assumed that the table has the following four columns:

1. *Method*;
2. *RMSE*;
3. *Diff*;
4. *Comment*;

4.2.1.5.1 Usage

```
RMSE.Total_kable <- function(RMSEs,
                               col1width = 15,
                               col2width = 7,
                               col3width = 5,
                               col4width = 25)
```

4.2.1.5.2 Parameters

- **RMSEs:** *Resut (RMSEs) Table* to print;
- **col1width:** Width of the first column;

- ***col2width***: Width of the second column;
- ***col3width***: Width of the third column;
- ***col4width***: Width of the fourth column;

4.2.1.5.3 Source Code

The source code of the function is provided below:

```
RMSE.Total_kable <- function(RMSEs,
                                col1width = 15,
                                col2width = 7,
                                col3width = 5,
                                col4width = 25){
  RMSEs |>
    RMSE_kable(col1width,
                col2width,
                col3width) |>
    column_spec(column = 4,
                width = RSME.tibble.col_width(col4width))
}
```



Under the hood, the function calls internally the `RMSE_kable` function described above.

4.2.2 Common Helper Functions: Regularization



The complete source code of the functions described in this section are available in the [Model Tuning](#) section of the `common-helper.functions.R` script on *GitHub*.

4.2.2.1 `mean_reg` Function

A general function for computing the arithmetic mean given a *regularization parameter* λ . We will call it the *penalized mean*.

4.2.2.1.1 Usage

```
mean_reg(vals, lambda = 0, na.rm = TRUE)
```

4.2.2.1.2 Parameters

- **vals:** values (usually a numeric vector) to calculate the *penalized mean*;
- **lambda:** a *regularization parameter* λ used in the *Regularization techniques*.
- **na.rm:** a logical evaluating to TRUE or FALSE indicating whether NA values should be stripped before the computation proceeds.

4.2.2.1.3 Details

We call this function internally from the functions that compute the *penalized estimates* as described in section [23.6 Penalized Least Squares](#) of the *Course Textbook*[12].



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).

4.2.2.1.4 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*.

4.2.2.1.5 Value

The function calculates the *penalized mean* of the `vals` parameter as follows:

$$\mu(\lambda) = \frac{1}{\lambda + N} \sum_{i=1}^N x_i$$

When the `lambda` parameter is 0 (meaning the $\lambda = 0$), the function calculates the simple arithmetic mean as follows:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i$$

4.2.2.2 `tune.model_param` Function

The function searches for the parameter value corresponding to the minimum value of the RMSE from the list of values specified by the `param_values` parameter.

4.2.2.2.1 Signature

```
tune.model_param <- function(param_values,
                                fn_tune.test.param_value,
                                break.if_min = TRUE,
                                steps.beyond_min = 2){

  # ...
  list(tuned.result = data.frame(RMSE = RMSEs_tmp,
                                 parameter.value = param_vals_tmp),
       best_result = param_values.best_result)
}
```

4.2.2.2.2 Parameters

- **`param_values`:** A list of values to search for the value corresponding to the minimum value of the RMSE ;
- **`fn_tune.test.param_value`:** A helper function that calculates the value of the RMSE for a given parameter value.;
- **`break.if_min = TRUE`:** A Boolean parameter that determines whether the function should terminate after completing the number of steps specified by the parameter `steps.beyond_min`, after the minimum value of the RMSE has been found;
- **`steps.beyond_min = 2`:** (takes effect only if `break.if_min` parameter is TRUE) Specifies the number of steps after finding the minimum value of the RMSE, upon completion of which the function should terminate.

4.2.2.2.3 Details

During execution, the function uses a helper function specified by the `fn_tune.test.param_value` parameter, which calculates the RMSE value for the given parameter from the list determined by the `param_values` parameter.



Note that the algorithm assumes that the dependence of the RMSE on the input parameter is a monotonically decreasing function until a minimum is reached and monotonically increasing thereafter. That is, it is assumed that the function has a single minimum on the given interval.

4.2.2.2.4 Value

The function returns a data structure containing the found value of the input parameter `param_values` for which the RMSE value is minimal, as well as the minimum RMSE value itself, along with a sequence of all calculated RMSE values:

```
list(tuned.result = data.frame(RMSE = RMSEs_tmp,
                               parameter.value = param_vals_tmp),
     best_result = param_values.best_result)
```

4.2.2.2.5 Source Code

Below is the most significant part 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`:
`RMSE` reached its minimum: ", RMSE_min, "
for parameter value: ", prm_val)
      put_log2("Function: `tune.model_param`:
Current `RMSE` value is %1 related to parameter value: %2",
               RMSE_tmp,
               prm_val)

      if (i > i_max.beyond_RMSE_min) {
        warning("Function: `tune.model_param`:
Operation is broken (after `RMSE` 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)
}

```

4.2.2.3 `model.tune.param_range` Function

The function fine-tunes the model by searching for the best possible value of the input parameter over a given interval for which the corresponding RMSE value is minimal.

4.2.2.3.1 Signature

```
model.tune.param_range <- function(loop_starter,
                                      tune_dir_path,
                                      cache_file_base_name,
                                      fn_tune.test.param_value,
                                      max.identical.min_RMSE.count = 4,
                                      endpoint.min_diff = 0,
                                      break.if_min = TRUE,
                                      steps.beyond_min = 2){
  # ...
  list(best_result = param_values.best_result,
       param_values.endpoints = c(prm_val.leftmost, prm_val.rightmost, seq_increments),
       tuned.result = data.frame(parameter.value = parameter.value,
                                 RMSE = result.RMSE))
}
```

4.2.2.3.2 Parameters

loop_starter

A numeric vector of the form `c(start, end, dvs)`, where `start` and `end` are the endpoints of the interval on which the parameter value that minimizes RMSE is sought. `dvs` is a divisor for splitting the interval to transform it into a sequence of values among which the value that minimizes RMSE is sought. For this purpose, the sequence step 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 `seq` as follows:

```
seq(start, end, step)
```

In fact, the `seq` function is called internally to generate the sequence during the execution of the `model.tune.param_range` function.

tune_dir_path

To improve performance, the algorithm caches intermediate results in the file system. This parameter 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` parameter, as well as some other intermediate values calculated during the execution.

fn_tune.test.param_value

This is a helper function name that is passed to the same-named parameter of the [tune.model_param](#) function that is called internally during the execution (see the description of the [tune.model_param](#) function [above](#)).

max.identical.min_RMSE.count = 4

If more than one identical minimum RMSE value is calculated during execution, the number of identical minimums is limited by the value of this parameter. When it is reached, the algorithm considers the task execution to be complete.

endpoint.min_diff = 0

Defines the sensitivity threshold for determining the neighborhood boundaries of the minimum RMSE value (for details, see the [Details](#) section [below](#)).

break.if_min = TRUE

This is a parameter that is required for the [tune.model_param](#) function that is called internally during execution (see the description of the [tune.model_param](#) function [above](#)).

steps.beyond_min = 2

This is a parameter that is required for the [tune.model_param](#) function that is called internally during execution (see the description of the [tune.model_param](#) function [above](#)).

4.2.2.3.3 Details

First, the function generates a sequence of the [input parameter values](#) based on the [loop_starter](#) parameter values, as described above (see the [loop_starter](#) parameter description for the details), which is used as one of the input parameters for the helper function [tune.model_param](#), which is repeatedly called during the execution, performing fine-tuning of the model.

The [tune.model_param](#) function returns a range of input parameter values associated with the set of corresponding *Root Mean Squared Errors* that is also guaranteed to include their minimum value, as described in the [Value](#) subsection of the [tune.model_param](#) function description.

Next, the function figures out the boundary indices of a range of values from the neighborhood of the minimum RMSE by calling internally another helper function [get_fine_tune.param.endpoints.idx](#) (see the description below) and, using one more helper function [get_best_param.result](#) (see the description below), a pair of values, corresponding to the best result: minimal RMSE and corresponding input parameter value (which is considered the best).

The found values of the boundary indices are then used as the endpoints of a new interval to regenerate the sequence based on it in the next iteration, just as it was done based on the values of the [loop_starter](#) parameter at the very beginning of the execution. The value of [step divisor](#), used to calculate the step of the generated sequence, remains unchanged during the entire execution (see the description of the [loop_starter](#) parameter above for details).

Thus, with each subsequent iteration, the minimum RMSE value is calculated more accurately, over an ever-decreasing interval, with the boundary values tending to the minimum RMSE value, and the sequence generated with an ever-decreasing step.

The calculation is completed when subsequent calculated values of the minimum RMSE stop improving and reach the most accurate possible value.

4.2.2.3.4 Value

A data structure containing the minimum RMSE value reached during the fine-tuning process, the corresponding input parameter value, and information about the final sequence, on which the best output values were found:

```
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))
```

4.2.2.3.5 Source Code

Below is the simplified version of the source code of the `model.tune.param_range` function:

```
model.tune.param_range <- function(loop_starter,
                                      tune_dir_path,
                                      cache_file_base_name,
                                      fn_tune.test.param_value,
                                      max.identical.min_RMSE.count = 4,
                                      is.cv = TRUE,
                                      endpoint.min_diff = 0, #1e-07,
                                      break.if_min = TRUE,
                                      steps.beyond_min = 2){

  seq_start <- loop_starter[1]
  seq_end <- loop_starter[2]
  interval_divisor <- loop_starter[3]

  if (interval_divisor < 4) {
    interval_divisor <- 4
  }

  prm_val.leftmost <- seq_start
  prm_val.rightmost <- seq_end

  RMSE.leftmost <- NA
  RMSE.rightmost <- NA

  best_RMSE <- NA
  param.best_value <- 0

  param_values.best_result <- c(param.best_value = param.best_value,
                                 best_RMSE = best_RMSE)
  # Start repeat loop
  repeat{
    seq_increment <- (seq_end - seq_start)/interval_divisor

    if (seq_increment < 0.00000000000001) {
      warning("Function `model.tune.param_range`:
parameter value increment is too small.")
      break
    }
  }
}
```

```

}

test_param_vals <- seq(seq_start, seq_end, seq_increment)

tuned_result <- tune.model_param(test_param_vals,
                                    fn_tune.test.param_value,
                                    break.if_min,
                                    steps.beyond_min)

tuned.result <- tuned_result$tuned.result
plot(tuned.result$parameter.value, tuned.result$RMSE)

bound.idx <- get_fine_tune.param.endpoints.idx(tuned.result)
start.idx <- bound.idx["start"]
end.idx <- bound.idx["end"]
best_RMSE.idx <- bound.idx["best"]

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

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
}

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
}

if (end.idx - start.idx <= 0) {
  warning(`tuned.result$parameter.value` sequential start index are the same or greater than end or
break

```

```

}

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)

  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)

    param_values.best_result <-
      get_best_param.result(tuned.result$parameter.value,
                            tuned.result$RMSE)
    break
  }

} else if (best_RMSE < min_RMSE) {
  stop("Current minimal RMSE is greater than previously computed best value: ",
       best_RMSE,
  Currently computed minial value is: ", min_RMSE)
}

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)
}

# End repeat loop

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))
}

```



The complete version of the source code of the `model.tune.param_range` is available in the [Model Tuning](#) section of the `common-helper.functions.R` script on [GitHub](#).

4.2.2.4 `get_fine_tune.param.endpoints` Function

The algorithm finds the values of the nearest neighboring elements of the element corresponding to the minimum RMSE value in the given sequence.

4.2.2.4.1 Signature

```
get_fine_tune.param.endpoints <- function(preset.result) {  
  # ...  
  
  c(start = preset.result$parameter.value[i],  
    end = preset.result$parameter.value[j],  
    best = preset.result$parameter.value[best.idx])  
}
```

4.2.2.4.2 Parameters

preset.result

A data structure compatible with the `tuned.result` item of the data structure returned by the `tune.model_param` function (see the `Value` subsection of the `tune.model_param` function description section for more details). Here is a sample of using this parameter in the `model.tune.param_range` function code:

```
tuned_result <- tune.model_param(test_param_vals,  
                                    fn_tune.test.param_value,  
                                    break.if_min,  
                                    steps.beyond_min)  
  
tuned.result <- tuned_result$tuned.result  
# tuned.result = data.frame(RMSE, parameter.value)  
# ...  
bound.idx <- get_fine_tune.param.endpoints(tuned.result)  
# bounnd.idx = c(start.index, end.index, min_RMSE.index)
```

4.2.2.4.3 Details

Under the hood, the function internally calls another helper function `get_fine_tune.param.endpoints.idx` described below to figure out the boundary indices of the range of values from the nearest neighborhood of the value corresponding to the best RMSE in the data passed in the `preset.result` parameter.

4.2.2.4.4 Value

A vector containing the element value corresponding to the minimum RMSE in a given sequence, along with the values of its nearest neighboring elements:

```
c(start = preset.result$parameter.value[i],  
  end = preset.result$parameter.value[j],  
  best = preset.result$parameter.value[best.idx])
```

4.2.2.4.5 Source Code

The source code of the `get_fine_tune.param.endpoints` function is provided below:

```
get_fine_tune.param.endpoints <- function(preset.result) {  
  
  preset.result.idx <- get_fine_tune.param.endpoints.idx(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])  
}
```

4.2.2.5 `get_fine_tune.param.endpoints.idx` Function

4.2.2.5.1 Signature

```
get_fine_tune.param.endpoints.idx <- function(preset.result) {  
  # ...  
  
  c(start = i,  
    end = j,  
    best = best_RMSE.idx)  
}
```

4.2.2.5.2 Parameters

preset.result

A data structure compatible with the `tuned.result` item of the data structure returned by the `tune.model_param` function (see the `Value` subsection of the `tune.model_param` function description section for more details). Here is a sample of using this parameter in the `model.tune.param_range` function code:

```
tuned_result <- tune.model_param(test_param_vals,  
                                    fn_tune.test.param_value,  
                                    break.if_min,  
                                    steps.beyond_min)  
  
tuned.result <- tuned_result$tuned.result  
# tuned.result = data.frame(RMSE, parameter.value)  
# ...  
bound.idx <- get_fine_tune.param.endpoints.idx(tuned.result)  
# bounnd.idx = c(start.index, end.index, min_RMSE.index)
```

4.2.2.5.3 Details

The algorithm finds the indices of the nearest neighboring elements of the element corresponding to the minimum RMSE value in the given sequence.

4.2.2.5.4 Value

A vector containing the index of the element corresponding to the minimum RMSE value in a given sequence, along with the indices of its nearest neighboring elements:

```
c(start, # interval start index  
  end,   # interval end index  
  best)  # index of the best `RMSE`
```

4.2.2.5.5 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)
}

```

4.2.2.6 `get_best_param.result` Function

4.2.2.6.1 Signature

```
get_best_param.result <- function(param_values, rmses){  
  # ...  
  
  c(param.best_value = param_values[best_pvalue_idx],  
    best_RMSE = rmses[best_pvalue_idx])
```

4.2.2.6.2 Parameters

A pair of matched vectors, the first of which contains the values of the input parameters for tuning, and the second, the corresponding values of the RMSE.

- **param_values:** A vector containing the values of the input parameters for tuning;;
- **rmses:** A vector containing values of the corresponding Root Mean Squared Errors..

4.2.2.6.3 Details

Extracts the best values from the input data and returns them as a vector containing a pair of the best *input parameter* value and the corresponding *minimum RMSE*.

4.2.2.6.4 Value

A vector containing a pair of the best *input parameter* value and the corresponding *minimum RMSE*:

```
c(param.best_value, best_RMSE)
```

4.2.2.6.5 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])
```

References

- [1] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.2: Loss function. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfcii.harvard.edu/dsbook-part-2/highdim/regularization.html#sec-netflix-loss-function> (visited on 02/18/2025) (cit. on p. 3).
- [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. 3, 24).
- [3] Azamat Kurbanayev. *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. 3).

- [4] Rafael A. Irizarry. *Introduction to Data Science, Part II. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/> (visited on 02/18/2025) (cit. on p. 5).
- [5] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.1.1: MovieLens data. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movielens-data> (visited on 02/18/2025) (cit. on pp. 6, 11, 13).
- [6] Azamat Kurbanayev. *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. 9).
- [7] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.3: A first model. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#a-first-model> (visited on 02/18/2025) (cit. on p. 24).
- [8] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.4: User effects. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#user-effects> (visited on 02/18/2025) (cit. on pp. 25, 29).
- [9] 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. 26, 36, 38, 53, 69).
- [10] 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](https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf) (visited on 02/18/2025) (cit. on p. 29).
- [11] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.5: Movie effects. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#movie-effects> (visited on 02/18/2025) (cit. on p. 42).
- [12] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.6: Penalized least squares. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#penalized-least-squares> (visited on 02/18/2025) (cit. on pp. 44, 161).
- [13] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 23: Regularization, Section 23.7: Exercises. Statistics and Prediction Algorithms Through Case Studies*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/regularization.html#exercises> (visited on 02/18/2025) (cit. on pp. 52, 54).
- [14] 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](https://doi.org/10.1007/978-0-387-85820-3). URL: https://github.com/vwang0/recommender_system/blob/master/Recommender%20Systems%20Handbook.pdf (cit. on p. 54).
- [15] Rafael A. Irizarry. *Introduction to Data Science, Part II, Chapter 24: Matrix Factorization. Factor analysis*. Dec. 27, 2024. URL: <https://rafalab.dfci.harvard.edu/dsbook-part-2/highdim/matrix-factorization.html> (visited on 02/18/2025) (cit. on p. 137).
- [16] 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. 138).
- [17] 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. 138).

- [18] 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. 138).