MovieLens Project

Giovanni Azua Garcia giovanni.azua@outlook.com

11.12.2019

Overview

The solution herein corresponds to the MovieLens dataset and project. The MovieLens dataset contains movie ratings. Each sample in the dataset corresponds to a movie rating by an user and it's uniquely identified by the numeric `movieId` and `userId` attributes i.e. there are no duplicate ratings for the same user and movie. The rating is a numeric value between 0.5 and 5 by increments of 0.5 i.e. `seq(0.5, 5, by=0.5)`, the lowest rating is 0.5 and the highest 5. Additionally, each sample contains the movie title, movie genres and the rating timestamp in Unix epoch format. The MovieLens dataset is very sparse meaning most movie ratings are missing. That's it, most users only rate a small number of movies and fewer movies have thousands of ratings. In data "long format" this means that there are many missing rows or in "wide format" that there are many `NA`s. Our goal is to fill those `NA`s or put differently, be able to predict as accurately as possible the ratings an user would give to a movie based on existing ratings data e.g. by learning from her past preferences. The problem at hand is that of recommending movies to users based on existing ratings data. We would recommend a movie to an user when we predict that the user would rate a movie, for example, higher than 4. Two main approaches are provided in this solution and they're identified as 'BASIC' and 'ADVANCED'.

The first approach BASIC, extends the regularized approach presented in the Data Science course book by Prof. Rafael A. Irizarry where the global average, movie and user biases or effects are modeled. In this BASIC solution, additional support is provided to model the genres b_g and temporal effects b_d and b_w:

$$\hat{r}_{u,i} = \mu + \underbrace{b_i}_{\text{movie effect}} + \underbrace{b_u}_{\text{user effect}} + \underbrace{b_g}_{\text{genres effect}} + \underbrace{b_d + f_{\text{smooth}}(b_w)}_{\text{temporal effects}} + \epsilon_{u,i}$$

The temporal effects are partly covered by b_d that models the change in ratings depending on the day of the week (Mon-Sun) or day of the month (1-31) and it accounts for possible users' mood fluctuations e.g. users may overall provide better ratings on Sundays than Fridays or worse ratings towards the last days of the month e.g. 27-31. Another temporal effect covered that proved very significative is b_w the changes in rating patterns depending on how long the rating was given since the movie was first released. The effect b_w groups the ratings per number of week blocks since the movie was released and then fits a smoothing function $f_{\text{smooth}}(b_w)$. The minimum timestamp per movie is used as a proxy for the release date of that movie i.e. this solution assumes that the earliest movie rating sample (according to the timestamp attribute) corresponds to its release date. The loss function minimized for the BASIC approach is the following:

$$\text{BASIC}_{\text{loss}} = \frac{1}{N}\sum_{u,i}(r_{u,i} - (\mu + b_i + b_u + b_g + b_d + f_{\text{smooth}}(b_w)))^2 + \lambda\left(\sum_i b_i^2 + \sum_u b_u^2 + \sum_g b_g^2 + \sum_d b_d^2\right)$$

The ADVANCED second approach, extends the BASIC with low-rank matrix factorization using SGD (Stochastic Gradient Descent) to account for user-movie interactions. The ADVANCED model excludes the b_d from the BASIC approach because it provided only marginal improvement to the RMSE:

$$\hat{r}_{u,i} = \mu + \underbrace{b_i}_{\text{movie effect}} + \underbrace{b_u}_{\text{user effect}} + \underbrace{b_g}_{\text{genres effect}} + \underbrace{f_{\text{smooth}}(b_w)}_{\text{temporal effects}} + P_u^T Q_i + \epsilon_{u,i}$$

Where $P_{(N \text{ users}, \ K \text{ latent})}$ is a matrix containing N rows that correspond to the unique users and K described in the literature as latent dimensions or principal components. $Q_{(N \text{ movies}, \ K \text{ latent})}$ is a matrix containing M rows that correspond to the unique movies and K. Note that the two matrices P and Q are modeled in different ways depending on the algorithm. In this solution and for performance reasons the two matrices are in the transposed version of what was described before (the K latent dimensions are stored rowwise) to ensure faster computations due to the algorithm data access patterns to match the R column-major matrix representation. This will be explained in more detail later.

The loss function minimized for the ADVANCED approach is the following. Note that there are two separate lambdas, the ADVANCED lambda penalizes large coefficients in P and Q:

$$\text{ADVANCED}_{\text{loss}} = \underset{P,Q}{\text{argmin}} \sum_{u,i} \left(r_{u,i} - (\underbrace{\hat{r}_{u,i}}_{\text{prediction using BASIC}} + P_u^T Q_i) \right)^2 + \lambda_{\text{ADVANCED}} \left(\sum_u \parallel P_u \parallel^2 + \sum_i \parallel Q_i \parallel^2 \right)$$

For both cases BASIC and ADVANCED, the models were implemented in R fully integrated with the `caret` package. That is, they were implemented as a `caret` model so that they integrate with the `caret` machine learning infrastructure for: calibration (e.g. cross validation), training and prediction.

Methods and Analysis

In this section the two methods BASIC and ADVANCED are presented. For each approach, supporting examples and plots are shown, then the `caret` model implementation source listing is provided which is later executed as part of the results section where the calibration (cross validation), predictions and RMSEs are computed.

We start by running the provided common code which generates the edx (training) and validation sets. A `portable.set.seed(seed)` function is provided which will execute the correct `seet.seed(seed)` variation depending on the R version:

```
#########################################################################################
## Capstone Project MovieLens.
##
## Author: Giovanni Azua Garcia <giovanni.azua@outlook.com>
#########################################################################################

# clean the environment
rm(list = ls())
# trigger garbage collection and free some memory if possible
gc(TRUE, TRUE, TRUE)
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 485643 26.0    1066754   57   485643 26.0
## Vcells 929054  7.1    8388608   64   929054  7.1
```

```
#########################################################################################
## Install and load required library dependencies
#########################################################################################

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

```
## Loading required package: tidyverse
```

```
## -- Attaching packages ---------------------------------------------------------------------------
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

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

```
## Loading required package: caret

## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift
```

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

```
## Loading required package: data.table

##
## Attaching package: 'data.table'

## The following objects are masked from 'package:dplyr':
##
##     between, first, last

## The following object is masked from 'package:purrr':
##
##     transpose
```

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

```
## Loading required package: tictoc
```

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

```
## Loading required package: lubridate

##
## Attaching package: 'lubridate'

## The following objects are masked from 'package:data.table':
##
##     hour, isoweek, mday, minute, month, quarter, second, wday, week,
##     yday, year

## The following object is masked from 'package:base':
##
##     date
```

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

```
## Loading required package: doMC

## Loading required package: foreach
```

```
##
## Attaching package: 'foreach'

## The following objects are masked from 'package:purrr':
##
##     accumulate, when

## Loading required package: iterators

## Loading required package: parallel
```

```r
library(tidyverse)
library(caret)
library(data.table)
library(tictoc)
library(lubridate)
library(stringr)
library(doMC)

##########################################################################################
## Define important reusable functions e.g. the RMSE function
##########################################################################################

# Loss function: the root mean squares estimate
RMSE <- function(x, y) {
  sqrt(mean((x - y)^2))
}

# portable (across R versions) set.seed function implementation
portable.set.seed <- function(seed) {
  if (R.version$minor < "6") {
    set.seed(seed)
  } else {
    set.seed(seed, sample.kind="Rounding")
  }
}

##########################################################################################
## Create edx set, validation set
##########################################################################################

# Note: this process could take a couple of minutes

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

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

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

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

```
                                               title = as.character(title),
                                               genres = as.character(genres))

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

# Validation set will be 10% of MovieLens data

portable.set.seed(1)
```

```
## Warning in set.seed(seed, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

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

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

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

# Add rows removed from validation set back into edx set

removed <- anti_join(temp, validation)
```

```
## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")
```

```
edx <- rbind(edx, removed)

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

The dataset names are standarized to match the Data Science course book and thus, make it easier to follow. Please note that I have mixed up the naming conventions and the logic behind is the following. The same naming convention is used as in the book for the dataset and variable names covered there i.e. `train_set` the underscored lower case naming convention, whereas any new function name or model name would be in camel case e.g. `cFBasic`:

```
#########################################################################################
## Standarize the dataset names
#########################################################################################

# training set is edx
train_set <- edx

# validation set
validation_set <- validation
```

As discussed in the overview section, both approaches presented require the minimum timestamp per movie `min_ts` to be present in each rating sample. The computation of `min_ts` requires joining the edx and validation sets as the minimum timestamp could be in either of the two and is important that it's the minimum across both sets. In practice, this "release date" value could be fetch from somewhere else as it's known for each movie but for simplicity the `min_ts` is used as a proxy.

```
#########################################################################################
## Computing and adding the minimum timestamp `min_ts` feature needed later. The minimum
```

```r
## timestamp is computed per movie and it's a proxy for the release date of a movie. Here
## I assume that the release date of a movie corresponds to the first available rating
## entry for that movie. This is needed in order to create a new feature, the number of
## weeks since the movie was launched.
##########################################################################################

## VALIDATION SET ACCESS ALERT! accessing the validation set to add a feature globally.
tic("adding the minimum timestamp feature to the full dataset")
ts_mins <- train_set %>%
  bind_rows(validation_set) %>%
  group_by(movieId) %>%
  summarise(min_ts = min(timestamp))

# add ts_min attribute to the train_set
train_set <- train_set %>%
  left_join(ts_mins, by="movieId")

# add ts_min attribute to the validation_set
validation_set <- validation_set %>%
  left_join(ts_mins, by="movieId")
toc()
```

```
## adding the minimum timestamp feature to the full dataset: 2.723 sec elapsed
```

We create an experimental set subset of the training edx set as follows:

```r
##########################################################################################
## Create experimental set as subset of the training set containing 1m samples
##########################################################################################

# set the seed again
portable.set.seed(1)
```

```
## Warning in set.seed(seed, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```r
# create subset of the training edx set
experimental_set <- train_set %>%
  sample_n(1000000)
```

BASIC method

To make a case and demonstrate the BASIC approach, the following code builds the first part of the method as provided by the course book and then the genres and temporal effects are modeled. Note that the following code is executed on a one million random samples subset of the training edx set and that the computed RMSEs depict the same decreasing trend as more effects are accounted for in the BASIC model, exactly as it's shown in the course book:

```r
# global average
mu <- mean(experimental_set$rating)
# compute predictions
rmse_results <- tibble(method = "Just the average",
                       RMSE = RMSE(experimental_set$rating, mu))

# lambda is the optimal one found using CV later
```

```r
lambda <- 5
# compute regularized movie effects
movie_avgs <- experimental_set %>%
  group_by(movieId) %>%
  summarize(b_i = sum(rating - mu)/(n() + lambda))
# compute predictions
predicted_ratings <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  mutate(pred=mu + b_i) %>%
  pull(pred)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularized Movie Effects",
                                 RMSE = RMSE(predicted_ratings, experimental_set$rating)))

# compute regularized user effects
user_avgs <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = sum(rating - mu - b_i)/(n() + lambda))
# compute predictions
predicted_ratings <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  pull(pred)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Regularized Movie + User Effects",
                                 RMSE = RMSE(predicted_ratings, experimental_set$rating)))

# show the progress so far, to see how the RMSE keeps decreasing as we account
# for more effect types
as.data.frame(rmse_results)
```

```
##                             method      RMSE
## 1                 Just the average 1.0612718
## 2        Regularized Movie Effects 0.9414304
## 3 Regularized Movie + User Effects 0.8414092
```

As continuation, now the genres effects are modeled. The same principle as in movie and user is employed to model the genre effects.

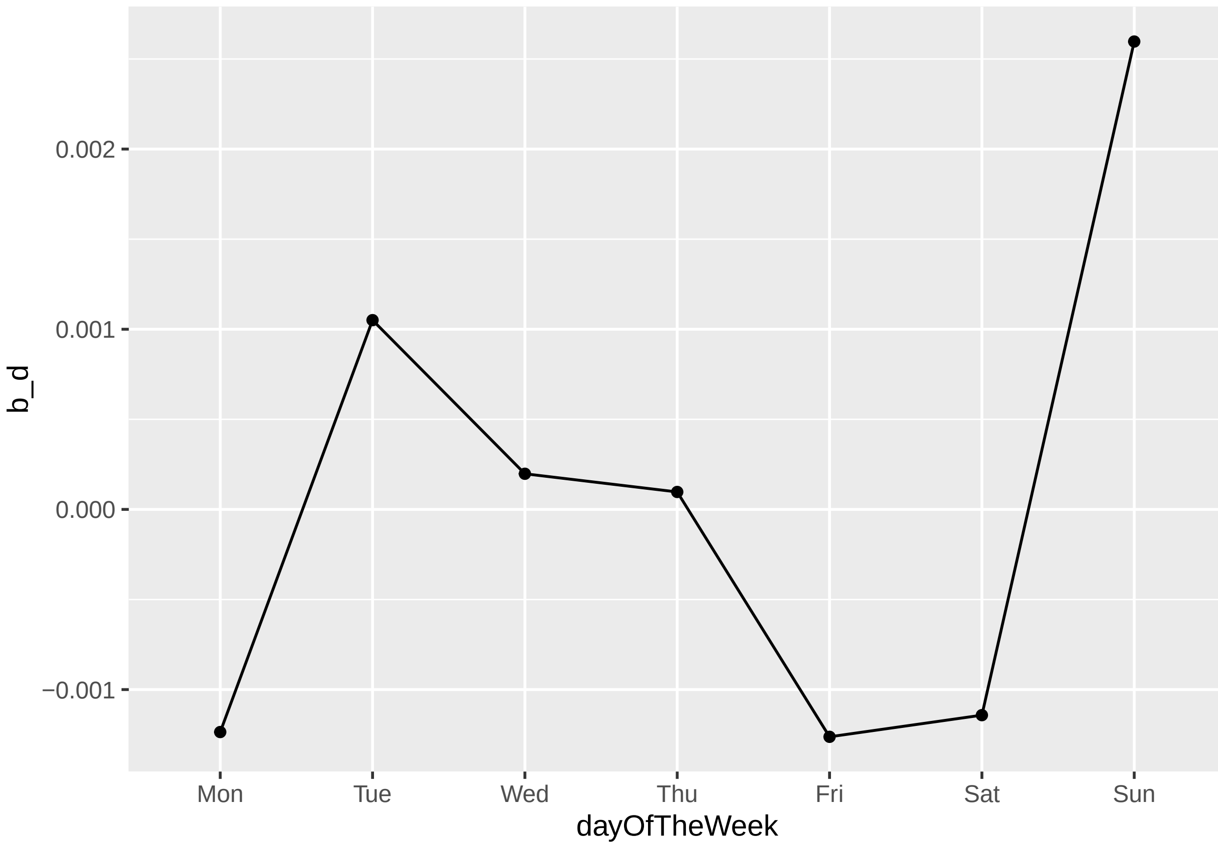
```r
# compute regularized genre effects
genre_avgs <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  group_by(genres) %>%
  summarize(b_g = sum(rating - (mu + b_i + b_u))/(n() + lambda))
# compute predictions
predicted_ratings <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  mutate(pred = mu + b_i + b_u + b_g) %>%
  pull(pred)
rmse_results <- bind_rows(rmse_results,
```

```
                           tibble(method="Regularized Movie + User + Genre Effects",
                                  RMSE = RMSE(predicted_ratings, experimental_set$rating)))

# show the progress so far, to see how the RMSE keeps decreasing as we account
# for more effect types
as.data.frame(rmse_results)
```

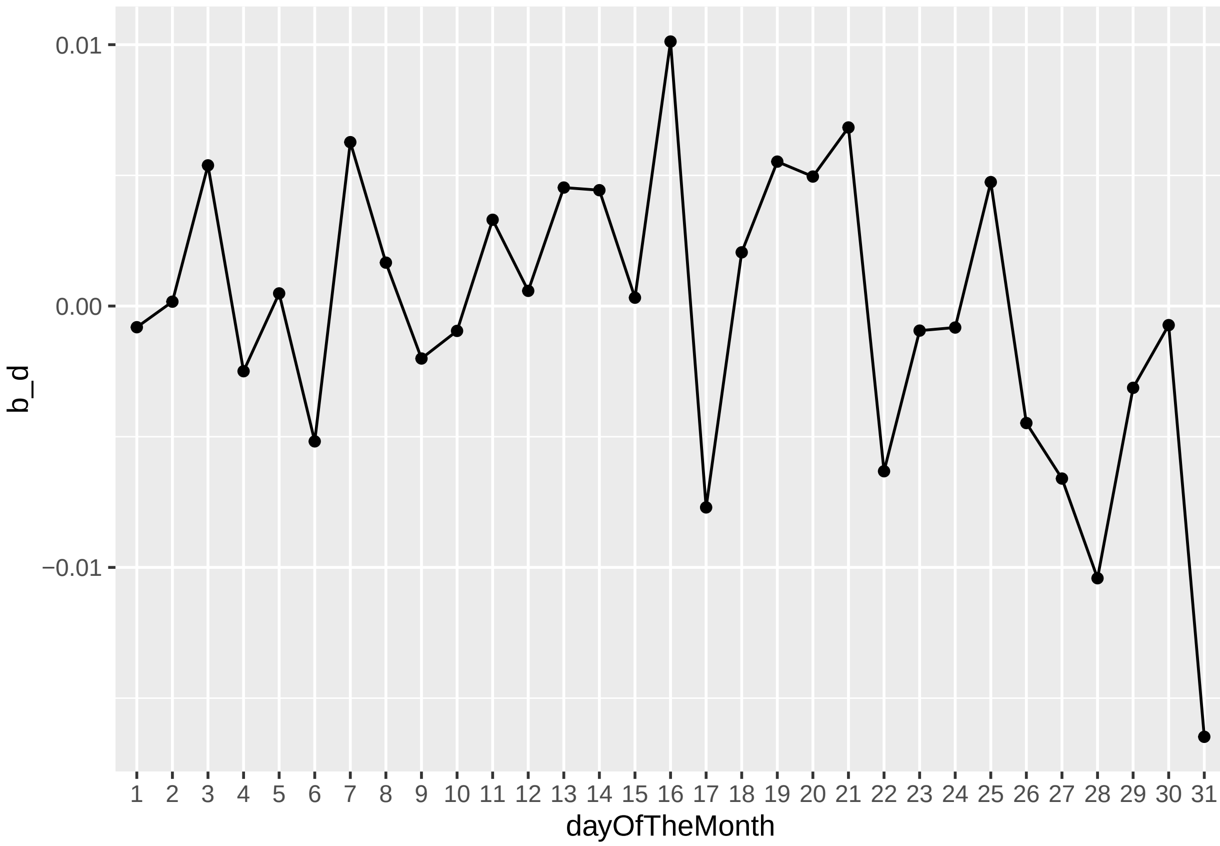
```
##                                      method      RMSE
## 1                          Just the average 1.0612718
## 2                 Regularized Movie Effects 0.9414304
## 3          Regularized Movie + User Effects 0.8414092
## 4 Regularized Movie + User + Genre Effects 0.8410014
```

Now the temporal effects day of the week or day of the month are modeled. We can see for this experimental sample training subset that the day of the month apparently provides better RMSE though selecting which day (of the week or of the month) method to use will be part of the calibration (cross validation) process. Note that the code `as.factor(wday(as_datetime(timestamp))` helps extract the day of the week feature from each timestamp which is then recoded into `Mon-Sun` for clarity but not needed in the actual model implementation. The code `as.factor(day(round_date(as_datetime(timestamp), unit = "day")))` helps extract the day of the month. The two generated plots reveal how the mean ratings per group depict dramatic rating changes e.g. Sunday ratings are much higher in average than Friday or Saturday.

```
# add day of week feature to the experimental set
experimental_set <- experimental_set %>%
  mutate(dayOfTheWeek = recode_factor(as.factor(wday(as_datetime(timestamp))),
                                      `1`='Mon', `2`='Tue', `3`='Wed', `4`='Thu',
                                      `5`='Fri', `6`='Sat', `7`='Sun'))
# compute regularized day of the week effects
day_avgs <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  group_by(dayOfTheWeek) %>%
  summarise(b_d = sum(rating - (mu + b_i + b_u + b_g))/(n() + lambda))
# plot the day of the week effects
day_avgs %>%
  ggplot(aes(dayOfTheWeek, b_d, group = 1)) + geom_point() + geom_line()
```




```
# compute predictions
predicted_ratings <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  left_join(day_avgs, by='dayOfTheWeek') %>%
  mutate(pred = mu + b_i + b_u + b_g + b_d) %>%
  pull(pred)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Reg. Movie + User + Genre + Day of Week Effects",
                                 RMSE = RMSE(predicted_ratings, experimental_set$rating)))

# add day of month feature to the experimental set
experimental_set <- experimental_set %>%
  mutate(dayOfTheMonth = as.factor(day(round_date(as_datetime(timestamp), unit = "day"))))
# compute regularized day of the month effects
day_avgs <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  group_by(dayOfTheMonth) %>%
  summarise(b_d = sum(rating - (mu + b_i + b_u + b_g))/(n() + lambda))
# plot the day of the month effects
day_avgs %>%
  ggplot(aes(dayOfTheMonth, b_d, group = 1)) + geom_point() + geom_line()
```



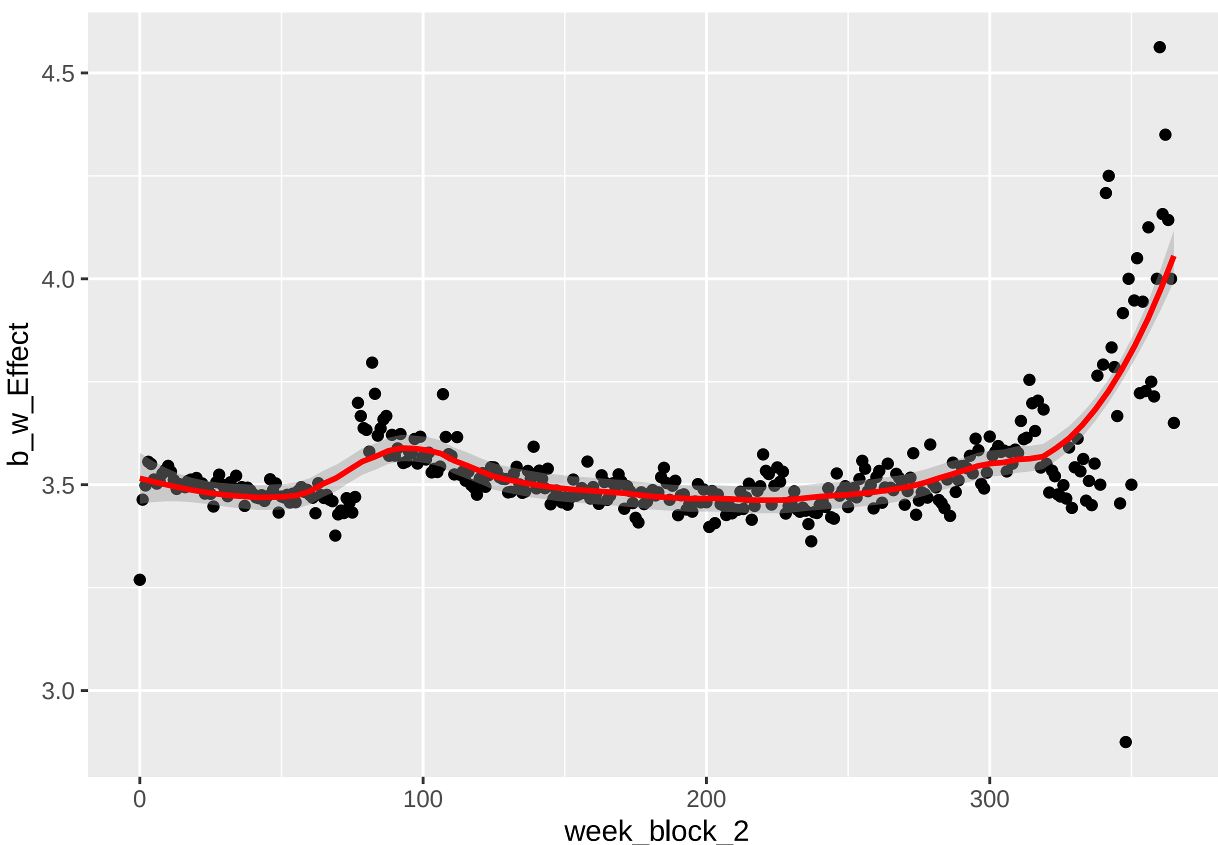

```
# compute predictions
predicted_ratings <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  left_join(day_avgs, by='dayOfTheMonth') %>%
  mutate(pred = mu + b_i + b_u + b_g + b_d) %>%
  pull(pred)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Reg. Movie + User + Genre + Day of Month Effects",
                                 RMSE = RMSE(predicted_ratings, experimental_set$rating)))

# show the progress so far, to see how the RMSE keeps decreasing as we account
# for more effect types
as.data.frame(rmse_results)
```

```
##                                            method      RMSE
## 1                                Just the average 1.0612718
## 2                       Regularized Movie Effects 0.9414304
## 3                Regularized Movie + User Effects 0.8414092
## 4        Regularized Movie + User + Genre Effects 0.8410014
## 5  Reg. Movie + User + Genre + Day of Week Effects 0.8410004
## 6 Reg. Movie + User + Genre + Day of Month Effects 0.8409852
```

At this point it's possible to model the temporal effects coming from the "week" effect or elapsed time in block of weeks since the movie release date to the date of the rating. Note the code for computing the "week" is as follows `ceiling(lubridate::as.duration(lubridate::as_datetime(min_ts) %--% lubridate::as_datetime(timestamp)) / lubridate::dweeks(2))` it is computed as the duration i.e. elapsed time or difference between the timestamp `min_ts` and the rating `timestamp` divided by the week span or blocks `dweeks(weekSpan)` which is the number of seconds within the block of weeks. The following code and plot illustrate how strong the "week" effect is on the ratings. The plot is constructed without excluding the other effects and in order to emphasize this "week" effect. The plot reveals something really interesting, the older movies that are rated most recently tend to have much higher rating. A possible explanation is that users watching these older movies most recently and providing the rating may have been already positively biased by past ratings or recommendations from friends and were less surprised and happier with those movies. The model hyper-parameters used are: span in weeks or week blocks `weekSpan=2` and for loess `span=0.3` and `degree=2`. These hyper-parameters were found to be the best as part of the model calibration (cross validation) later:

```
# compute the week block effect for each rating since the movie release date and smooth it
experimental_set %>%
  mutate(week_block_2 = ceiling(as.duration(as_datetime(min_ts) %--%
                                            as_datetime(timestamp)) / dweeks(2))) %>%
  group_by(week_block_2) %>%
  summarise(b_w_Effect=mean(rating)) %>%
  ggplot(aes(week_block_2, b_w_Effect)) + geom_point() +
  geom_smooth(color="red", span=0.3, method.args=list(degree=2))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
# fit the elapsed time week model
week_fit <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  left_join(day_avgs, by='dayOfTheMonth') %>%
```

```r
  mutate(week = ceiling(as.duration(as_datetime(min_ts) %--%
                                    as_datetime(timestamp)) / dweeks(2))) %>%
  group_by(week) %>%
  summarise(rating_residual=mean(rating - (mu + b_i + b_u + b_g + b_d))) %>%
  loess(rating_residual~week, data=., span=0.3, degree=2)
# compute predictions
predicted_ratings <- experimental_set %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  left_join(day_avgs, by='dayOfTheMonth') %>%
  mutate(week = ceiling(as.duration(as_datetime(min_ts) %--%
                                    as_datetime(timestamp)) / dweeks(2))) %>%
  mutate(pred = mu + b_i + b_u + b_g + b_d + predict(week_fit, .)) %>%
  pull(pred)
rmse_results <- bind_rows(rmse_results,
                          tibble(method="Reg. Movie + User + Genre + Day of Month + Week Effects",
                                 RMSE = RMSE(predicted_ratings, experimental_set$rating)))

# show the progress so far, to see how the RMSE keeps decreasing as we account
# for more effect types
as.data.frame(rmse_results)
```

```
##                                                    method      RMSE
## 1                                        Just the average 1.0612718
## 2                              Regularized Movie Effects 0.9414304
## 3                       Regularized Movie + User Effects 0.8414092
## 4               Regularized Movie + User + Genre Effects 0.8410014
## 5          Reg. Movie + User + Genre + Day of Week Effects 0.8410004
## 6         Reg. Movie + User + Genre + Day of Month Effects 0.8409852
## 7 Reg. Movie + User + Genre + Day of Month + Week Effects 0.8404594
```

```r
# remove experimental variables and data that are no longer needed
rm(predicted_ratings, rmse_results, week_fit, experimental_set, day_avgs,
   genre_avgs, user_avgs, movie_avgs, lambda, mu)
```

Now that the case for the BASIC model has been built and explained, the corresponding `caret` implementation `cFBasic` model is listed below. See the documentation guide for Using Your Own Model with the `caret` package. The following listing defines the `cFBasic` model including: hyper-parameters, tune grid, train and predict functions so that it can be used as part of the standard `caret` functions for calibration (e.g. cross validation), train and prediction. We can see that the following implementation essentially componentizes the previous listing into separate functions for train and predict:

```r
#############################################################################################
## Create the CF BASIC model integrated with the caret package. The model is called
## cfBasic standing for Collaborative Filtering basic method which provides some extensions
## and improvements over the model described in the lectures and book.
##
## Extensions over the book model:
## - Models the genres effect by computing the regularized rating genres effects.
## - Models the time effects using the day of the week or month for a rating.
## - Models the time effects using the number of week blocks since the movie was released.
##
## This caret model specification follows the implementation details described here:
## https://topepo.github.io/caret/using-your-own-model-in-train.html
```

```r
#####################################################################################

# trigger garbage collection and free some memory if possible
gc(TRUE, TRUE, TRUE)
```

```
##             used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells  11275786  602.2   17369902  927.7  11275786  602.2
## Vcells 150148140 1145.6  318281139 2428.3 150148140 1145.6
```

```r
# Define the model cFBasic (Collaborative Filtering basic)
cFBasic <- list(type = "Regression",
                library = c("lubridate", "stringr"),
                loop = NULL,
                prob = NULL,
                sort = NULL)

# Five different parameters are supported:
# @param lambda the regularizaton parameter applied to the different effects
# @param span the span parameter applied to loess for smoothing the week elapsed time effects
# @param degree the degree parameter applied to loess for smoothing the week elapsed time effects
# @param weekSpan the number of weeks to bin the data with.
# @param dayType whether "dayOfTheWeek" e.g. 1-7 or "dayOfTheMonth" 1-31
#
cFBasic$parameters <- data.frame(parameter = c("lambda", "span", "degree", "weekSpan",
                                               "dayType"),
                                 class = c(rep("numeric", 4), "character"),
                                 label = c("Lambda", "Loess Span", "Loess Degree",
                                           "Week Span", "Day Type"))

# Define the required grid function, which is used to create the tuning grid (unless
# the user gives the exact values of the parameters via tuneGrid)
cFBasic$grid <- function(x, y, len = NULL, search = "grid") {
  lambda <- seq(2, 5, 0.1) # like in the book
  span <- c(0.05, 0.1, 0.3, 0.5, 0.75)
  degree <- c(1, 2)
  weekSpan <- c(1, 2, 3, 4, 5, 6, 7)
  dayType <- c("dayOfTheWeek", "dayOfTheMonth")

  # to use grid search
  out <- expand.grid(lambda = lambda,
                     span = span,
                     degree = degree,
                     weekSpan = weekSpan,
                     dayType = dayType)

  if(search == "random") {
    # random search simply random samples from the expanded grid
    out <- out %>%
      sample_n(100)
  }
  out
}

# Define the fit function so we can fit our model to the data
```

```r
cFBasic$fit <- function(x, y, wts, param, lev, last, weights, classProbs, ...) {
  # check whether we have a correct x
  stopifnot("userId" %in% colnames(x))
  stopifnot("movieId" %in% colnames(x))
  stopifnot("timestamp" %in% colnames(x))
  stopifnot("genres" %in% colnames(x))
  stopifnot("rating" %in% colnames(x))
  stopifnot("min_ts" %in% colnames(x))
  stopifnot(all(x$rating == y))

  # compute global mean
  mu <- mean(x$rating)

  # compute movie effects b_i
  movie_avgs <- x %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n() + param$lambda))

  # compute user effects b_u
  user_avgs <- x %>%
    left_join(movie_avgs, by='movieId') %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - mu - b_i)/(n() + param$lambda))

  # compute genre effects b_g
  genre_avgs <- x %>%
    left_join(movie_avgs, by='movieId') %>%
    left_join(user_avgs, by='userId') %>%
    group_by(genres) %>%
    summarize(b_g = sum(rating - (mu + b_i + b_u))/(n() + param$lambda))

  # add the day feature to model temporal effects
  if (param$dayType == "dayOfTheWeek") {
    x <- x %>%
      mutate(day = as.factor(wday(as_datetime(timestamp))))
  } else {
    stopifnot(param$dayType == "dayOfTheMonth")
    x <- x %>%
      mutate(day = as.factor(day(round_date(as_datetime(timestamp), unit = "day"))))
  }

  # compute the day effects b_d
  stopifnot("day" %in% colnames(x))
  day_avgs <- x %>%
    left_join(movie_avgs, by='movieId') %>%
    left_join(user_avgs, by='userId') %>%
    left_join(genre_avgs, by='genres') %>%
    group_by(day) %>%
    summarize(b_d = sum(rating - (mu + b_i + b_u + b_g))/(n() + param$lambda))

  # add the week feature to model temporal effects
  x <- x %>%
    mutate(week = ceiling(as.duration(as_datetime(min_ts) %--%
```

```
                                         as_datetime(timestamp)) / dweeks(param$weekSpan)))

  stopifnot("week" %in% colnames(x))
  week_fit <- x %>%
    left_join(movie_avgs, by='movieId') %>%
    left_join(user_avgs, by='userId') %>%
    left_join(genre_avgs, by='genres') %>%
    left_join(day_avgs, by='day') %>%
    group_by(week) %>%
    summarise(rating_residual=mean(rating - (mu + b_i + b_u + b_g + b_d))) %>%
    loess(rating_residual~week, data=., span=param$span, degree=param$degree)

  # return the model fit as a list
  list(mu=mu,
       movie_avgs=movie_avgs,
       user_avgs=user_avgs,
       genre_avgs=genre_avgs,
       day_avgs=day_avgs,
       week_fit=week_fit,
       params=param)
}

# Define the predict function that produces a vector of predictions
cFBasic$predict <- function(modelFit, newdata, preProc = NULL, submodels = NULL) {
  # add the day feature to model temporal effects
  if (modelFit$params$dayType == "dayOfTheWeek") {
    newdata <- newdata %>%
      mutate(day = as.factor(wday(as_datetime(timestamp))))
  } else {
    stopifnot(modelFit$params$dayType == "dayOfTheMonth")
    newdata <- newdata %>%
      mutate(day = as.factor(day(round_date(as_datetime(timestamp), unit = "day"))))
  }

  # add the week feature to model temporal effects
  newdata <- newdata %>%
    mutate(week = ceiling(as.duration(as_datetime(min_ts) %--% as_datetime(timestamp)) /
                          dweeks(modelFit$params$weekSpan)))

  stopifnot("day" %in% colnames(newdata))
  stopifnot("week" %in% colnames(newdata))
  predicted <- newdata %>%
    left_join(modelFit$movie_avgs, by='movieId') %>%
    left_join(modelFit$user_avgs, by='userId') %>%
    left_join(modelFit$genre_avgs, by='genres') %>%
    left_join(modelFit$day_avgs, by='day') %>%
    mutate(pred = modelFit$mu + b_i + b_u + b_g + b_d + predict(modelFit$week_fit, .)) %>%
    pull(pred)

  predicted
}
```

ADVANCED method

The ADVANCED model builds on top of the BASIC and attempts to explain the user-movie interactions only. It does that by first covering all the discussed effects using the BASIC model prediction $\hat{r_{i,j}}$. The implemented low-rank matrix factorization SGD algorithm works as follows:

1. Create the matrices P and Q and initialize them to standard normal values with zero mean and σ standard deviation parameter.
2. For `maxIter` random samples compute the gradient update derived below and update P and Q accordingly.
3. Store final P and Q as part of the fit object and use them to make predictions.

$$\epsilon_{u,i} = r_{u,i} - (\underbrace{\hat{r}_{u,i}}_{\text{prediction using BASIC}} + P_u^T Q_i)$$

$$\text{ADVANCED}_{\text{loss}} = \underset{P,Q}{\text{argmin}} \sum_{u,i} \epsilon_{u,i}^2 + \lambda_{\text{ADVANCED}} \left(\sum_u \parallel P_u \parallel^2 + \sum_i \parallel Q_i \parallel^2 \right)$$

$$\frac{\partial}{\partial P_u} = 2\epsilon_{u,i} Q_i + 2\lambda P_u = 2(\epsilon_{u,i} Q_i + \lambda P_u) \Rightarrow \Delta P_u = \gamma(\epsilon_{u,i} Q_i + \lambda P_u)$$

$$\frac{\partial}{\partial Q_i} = 2\epsilon_{u,i} P_u + 2\lambda Q_i = 2(\epsilon_{u,i} P_u + \lambda Q_i) \Rightarrow \Delta Q_i = \gamma(\epsilon_{u,i} P_u + \lambda Q_i)$$

Where γ is the learning rate. Therefore, in every SGD step the following P and Q updates are executed:

$$P_u = P_u + \gamma(\epsilon_{u,i} Q_i + \lambda P_u)$$
$$Q_i = Q_i + \gamma(\epsilon_{u,i} P_u + \lambda Q_i)$$

We see that this model has the following possible hyper-parameters: K number of latent dimensions, λ regularization that penalizes large values for P and Q, γ the learning rate and possibly σ the standard deviation corresponding to the normal distribution used to initialize P and Q. The following listing defines the `cFAdv` model including: hyper-parameters, tune grid, train and predict functions so that it can be used as part of the standard `caret` functions for calibration (e.g. cross validation), train and prediction. The γ hyper-parameter or learning rate was observed to have the following behavior. For lower values e.g. $\gamma = 0.001$ the convergence is slow but more steady whereas for larger values e.g. $\gamma = 0.2$ the convergence is initially very fast as it drops the RMSE abruptly and then reaches a plateau. An optimal variant of the SGD ADVANCED method should dynamically adapt γ depending on the convergence.

The matrices P and Q in the following listing are implemented with actual dimensions `K x N` and `K x M` respectively where N is the number of distinct users and M the number of distinct movies. The reason for this is that the default matrix ordering in R is column-major i.e. columns are stored as contiguous memory segments whereas row accesses require striding which is a costly memory access pattern. To train and run predictions using P and Q we access them by a specific user or movie therefore it was best to represent users and movies as columns in P and Q and the K latent dimensions to be represented row-wise.

Note also that the `cFAdv$fit` requires a few additional custom parameters. These parameters are concerned with controlling the number of iterations i.e. `maxIter` and accurately tracking the RMSE progress on a small subset of the training data. It's also possible to provide initial conditions for P and Q and this enables the use-case to continue training or learning P and Q from the state they were left in a previous training.

```
#############################################################################################
## Create the CF ADVANCED model integrated with the caret package. The model is called
## cfAdv standing for Collaborative Filtering advanced method which builds on top of the
## basic model and employs low-rank matrix factorization trained using SGD.
##
## This caret model specification follows the implementation details described here:
## https://topepo.github.io/caret/using-your-own-model-in-train.html
```

```
################################################################################

# free a bit of memory if possible
gc(TRUE, TRUE, TRUE)
```

```
##              used   (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells  11277427  602.3   17369902  927.7  11277427  602.3
## Vcells 150163351 1145.7  318281139 2428.3 150163351 1145.7
```

```
# Define the model cFAdv (Collaborative Filtering advanced)
cFAdv <- list(type = "Regression",
              library = c("lubridate", "stringr"),
              loop = NULL,
              prob = NULL,
              sort = NULL)

# Four different parameters are supported:
# @param K the number of latent dimensions
# @param gamma the learning rate
# @param lambda the regularizaton parameter applied to the different effects
# @param sigma the standard deviation of the initial values
#
cFAdv$parameters <- data.frame(parameter = c("K", "gamma", "lambda", "sigma"),
                               class = c(rep("numeric", 4)),
                               label = c("K-Latent Dim", "Learning rate",
                                         "Lambda", "Sigma of Init Values"))

# Define the required grid function, which is used to create the tuning grid (unless
# the user gives the exact values of the parameters via tuneGrid)
cFAdv$grid <- function(x, y, len = NULL, search = "grid") {
  K <- 2:3
  gamma <- c(0.02, 0.04, 0.06, 0.08, 0.1)
  lambda <- c(10^seq(-2, -1), 5*10^seq(-2, -1))
  sigma <- c(0.05, 0.1)

  # to use grid search
  out <- expand.grid(K = K,
                     gamma = gamma,
                     lambda = lambda,
                     sigma = sigma)

  if(search == "random") {
    # random search simply random samples from the expanded grid
    out <- out %>%
      sample_n(100)
  }
  out
}

# Define the fit function so we can fit our model to the data
# NOTE: the fit function requires the CF BASIC model as extended parameter argument
# @param P the initial P matrix.
# @param Q the initial Q matrix.
# @param maxIter maximum number of iterations or random samples.
```

```r
# @param trackConv whether to track RMSE convergence of the algorithm.
# @param perTrack percent of samples to track for RMSE convergence.
# @param iterBreaks number of steps before checking for convergence.
# @param fitCFBasic the BASIC CF fit model.
cFAdv$fit <- function(x, y, wts, param, lev, last, weights, classProbs,
                      P=NULL, Q=NULL, maxIter=1500, trackConv=FALSE, perTrack=0.01,
                      iterBreaks=100, fitCFBasic, ...) {
  # check whether we have a correct x
  stopifnot("userId" %in% colnames(x))
  stopifnot("movieId" %in% colnames(x))
  stopifnot("timestamp" %in% colnames(x))
  stopifnot("genres" %in% colnames(x))
  stopifnot("rating" %in% colnames(x))
  stopifnot("min_ts" %in% colnames(x))
  stopifnot(all(x$rating == y))

  # read model information from the CF BASIC fit
  mu  <- fitCFBasic$finalModel$mu
  user_avgs  <- fitCFBasic$finalModel$user_avgs
  movie_avgs <- fitCFBasic$finalModel$movie_avgs
  genre_avgs <- fitCFBasic$finalModel$genre_avgs
  week_fit <- fitCFBasic$finalModel$week_fit

  K <- param$K          # number of latent dimensions
  N <- nrow(user_avgs)  # number of users
  M <- nrow(movie_avgs) # number of movies

  # randomly initialize P and Q
  if (is.null(P)) P <- matrix(rnorm(K*N, sd=param$sigma), K, N)
  if (is.null(Q)) Q <- matrix(rnorm(K*M, sd=param$sigma), K, M)

  # ensure that the columns dimension match the number of distinct
  # users and movies
  stopifnot(ncol(P) == N)
  stopifnot(ncol(Q) == M)

  # identify rows by user or movie respectively. Note that this is the
  # lookup method to match users to P and movies to Q using the userId
  # or movieId as column name key.
  colnames(P) <- user_avgs$userId
  colnames(Q) <- movie_avgs$movieId

  computeRMSE <- function(subset_samples) {
    # compute the user-movie interaction effects contained in P and Q
    pq_effects <- subset_samples %>%
      group_by(userId, movieId) %>%
      mutate(pq=(P[,u]%*%Q[,i])[1]) %>%
      select(userId, movieId, pq)

    # compute the predictions
    predicted <- subset_samples %>%
      left_join(pq_effects, by=c('userId', 'movieId')) %>%
      mutate(predicted=mu + b_i + b_u + b_g + b_w + pq) %>%
```

```r
    pull(predicted)

  if(any(is.na(predicted))) {
    stop(sprintf("train - na detected for K=%d, gamma=%.6f, lambda=%.6f, sigma=%.6f",
                 param$K, param$gamma, param$lamda, param$sigma))
  }
  rmse_val <- RMSE(predicted, subset_samples$rating)
}

# add the week feature to model temporal effects
x <- x %>%
  mutate(week = ceiling(as.duration(as_datetime(min_ts) %--% as_datetime(timestamp)) /
                          dweeks(fitCFBasic$finalModel$params$weekSpan)))

# select random samples corresponding to the number of iterations parameter
samples <- x %>%
  left_join(movie_avgs, by='movieId') %>%
  left_join(user_avgs, by='userId') %>%
  left_join(genre_avgs, by='genres') %>%
  mutate(i=as.character(movieId), u=as.character(userId),
         b_w = predict(week_fit, .), residual=rating - (mu + b_i + b_u + b_g + b_w)) %>%
  sample_n(maxIter)

# use a subset of the samples to test for convergence
subset_samples <- samples %>%
  sample_n(nrow(samples)*perTrack)

rmse_val <- computeRMSE(subset_samples)

if (trackConv) {
  rmse_hist <- tibble(k=0, rmse=rmse_val)
} else {
  rmse_hist <- NULL
}

for (k in 1:nrow(samples)) {
  i <- as.character(samples[k,]$movieId)
  u <- as.character(samples[k,]$userId)

  # compute the residual
  residual <- samples[k,]$residual - (P[,u]%*%Q[,i])[1]

  # update the latent vectors
  P[,u] <- (P[,u] + param$gamma*(residual*Q[,i] - param$lambda*P[,u]))
  Q[,i] <- (Q[,i] + param$gamma*(residual*P[,u] - param$lambda*Q[,i]))

  # track convergence every "iterBreaks" steps
  if (trackConv && k %% iterBreaks == 0) {
    # check rmse
    rmse_val <- computeRMSE(subset_samples)
    cat(sprintf('the RMSE at k=%d is %.9f\n', k, rmse_val))
    rmse_hist <- rbind(rmse_hist, tibble(k=k, rmse=rmse_val))
  }
```

```r
  }

  if(any(is.na(P)) || any(is.na(Q))) {
    stop(sprintf("na detected in P or Q for K=%d, gamma=%.6f, lambda=%.6f, sigma=%.6f",
                 param$K, param$gamma, param$lamda, param$sigma))
  }

  # return the model fit as a list
  list(mu=mu,
       user_avgs=user_avgs,
       movie_avgs=movie_avgs,
       week_fit=week_fit,
       genre_avgs=genre_avgs,
       P=P,
       Q=Q,
       rmse_hist=rmse_hist,
       params=c(param, weekSpan=fitCFBasic$finalModel$params$weekSpan))
}

# Define the predict function that produces a vector of predictions
cFAdv$predict <- function(modelFit, newdata, preProc = NULL, submodels = NULL) {
  if(any(is.na(modelFit$P)) || any(is.na(modelFit$Q))) {
    stop(sprintf("predict - na in P or Q for K=%d, gamma=%.6f, lambda=%.6f, sigma=%.6f",
                 param$K, param$gamma, param$lamda, param$sigma))
  }

  # add the week feature to model temporal effects
  newdata <- newdata %>%
    mutate(week = ceiling(as.duration(as_datetime(min_ts) %--% as_datetime(timestamp)) /
                            dweeks(modelFit$params$weekSpan)))

  # compute the user-movie interaction effects contained in P and Q
  pq_effects <- newdata %>%
    group_by(userId, movieId) %>%
    mutate(i=as.character(movieId), u=as.character(userId),
           pq=(modelFit$P[,u]%*%modelFit$Q[,i])[1]) %>%
    select(userId, movieId, pq)

  # compute the predictions
  predicted <- newdata %>%
    left_join(modelFit$movie_avgs, by='movieId') %>%
    left_join(modelFit$user_avgs, by='userId') %>%
    left_join(modelFit$genre_avgs, by='genres') %>%
    left_join(pq_effects, by=c('userId', 'movieId')) %>%
    mutate(b_w=predict(modelFit$week_fit, .),
           predicted=modelFit$mu + b_i + b_u + b_g + b_w + pq) %>%
    pull(predicted)

  if(any(is.na(predicted))) {
    stop(sprintf("predict - na detected for K=%d, gamma=%.6f, lambda=%.6f, sigma=%.6f",
                 modelFit$params$K, modelFit$params$gamma, modelFit$params$lamda,
                 modelFit$params$sigma))
  }
```

```
  predicted
}
```

At this point we need another set to calibrate and fit the two model implementations listed above `cFBasic` and `cFAdv`. The ideal calibration set should be a small subset of the training edx set and still be dense enough i.e. contain enough ratings per movie and user so we learn high quality hyper-parameters. The calibration set is then built as follows:

```
#########################################################################################
## Build the calibration (cross validation) subset of the training set.
#########################################################################################

tic("collecting the calibration set of 2k samples, making sure it's a bit dense")
# set the seed again
portable.set.seed(1)
```

```
## Warning in set.seed(seed, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
calibration_set <- train_set %>%
  group_by(movieId) %>%           # group by movieId
  filter(n() > 3600) %>%          # pick movies having at least 3.6k ratings
  ungroup() %>%
  group_by(userId) %>%            # group by userId
  filter(n() > 400) %>%           # pick users within those movies having at least 400 ratings
  ungroup() %>%
  sample_n(2000)                  # choose 2k samples randomly
toc()
```

```
## collecting the calibration set of 2k samples, making sure it's a bit dense: 1.436 sec elapsed
```

```
# how many distinct movies and users in the calibration set?
cat(sprintf("The calibration set contains %d unique movies and %d unique users\n",
            length(unique(calibration_set$movieId)), length(unique(calibration_set$userId))))
```

```
## The calibration set contains 596 unique movies and 337 unique users
```

Result

At this point we're ready to calibrate and fit the models with the full training data set. Note that the order of the report doesn't exactly match the R code script order. The models are explained and implemented in the previous Methods and Analysis section and here we calibrate (cross-validate), train and compute the predictions.

BASIC method

We calibrate the BASIC model as shown in the listing below. We see that the optimal parameters found and stored in the resulting calibration fit are: λ=5, `loess span=0.3`, `loess degree=2`, `weekSpan=2` and `dayType="dayOfTheWeek"`. Recall that week span is the number of weeks or week block size to compute the elapsed time in number of week blocks (e.g. two-week blocks) since the movie release date to the day of the rating. The `dayType` chooses between considering day of the week or day of the month for the corresponding temporal day effect:

```r
###########################################################################################
## Calibrate the CF BASIC model on the calibration set (subset of the training set). Here
## I look for the best hyper-parameters that fit the basic model on a small subset of the
## training data.
###########################################################################################

# register number of cores for parallelizing the calibration or cross validation
registerDoMC(6)

cat(sprintf('The BASIC tunning grid contains %d hyper-parameter permutations.',
            nrow(cFBasic$grid())))
```

```
## The BASIC tunning grid contains 4340 hyper-parameter permutations.
```

```r
tic('BASIC - calibrating the model')
# set the seed again
portable.set.seed(1)
```

```
## Warning in set.seed(seed, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```r
control <- trainControl(method = "cv",
                        search = "grid",
                        number = 10,
                        p = .9,
                        allowParallel = TRUE,
                        verboseIter = TRUE)
calFitCFBasic <- train(x = calibration_set,
                       y = calibration_set$rating,
                       method = cFBasic,
                       trControl = control)
```

```
## Aggregating results
## Selecting tuning parameters
## Fitting lambda = 5, span = 0.3, degree = 2, weekSpan = 2, dayType = dayOfTheWeek on full training se
```

```
## Warning: Setting row names on a tibble is deprecated.
```

```r
toc()
```

```
## BASIC - calibrating the model: 381.014 sec elapsed
```

```r
## The bestTune model found is:
stopifnot(calFitCFBasic$bestTune$lambda == 5)
stopifnot(calFitCFBasic$bestTune$span == 0.3)
stopifnot(calFitCFBasic$bestTune$degree == 2)
stopifnot(calFitCFBasic$bestTune$weekSpan == 2)
stopifnot(calFitCFBasic$bestTune$dayType == "dayOfTheWeek")
```

At this point we're ready to train the `cFBasic` model on the whole training set and compute the out of sample RMSE on the validation set.

```r
###########################################################################################
## Fit the best BASIC model found on the complete edx train set.
###########################################################################################

tic('BASIC - training the model on the full training set')
fitCFBasic <- train(x = train_set,
```

```
                    y = train_set$rating,
                    method = cFBasic,
                    trControl = trainControl(method = "none"),
                    tuneGrid = calFitCFBasic$bestTune)
toc()
```

```
## BASIC - training the model on the full training set: 41.823 sec elapsed
```

```
##########################################################################################
## Finally compute the RMSE for the BASIC model on the validation set.
##########################################################################################

## VALIDATION SET ACCESS ALERT! accessing the validation set to compute RMSE.
tic('BASIC - predicting ratings')
predicted_ratings <- predict(fitCFBasic, validation_set)
rmse_val <- RMSE(predicted_ratings, validation_set$rating)
toc()
```

```
## BASIC - predicting ratings: 1.324 sec elapsed
```

```
cat(sprintf("BASIC - RMSE on validation data is %.9f", rmse_val))
```

```
## BASIC - RMSE on validation data is 0.864080283
```

```
# check that we get a reproducible result
stopifnot(abs(rmse_val - 0.864080283) < 1e-9)
```

The final RMSE obtained on the validation set using the BASIC `fitCFBasic` fit (calibrated and trained only on the full edx training set) is `RMSE=0.864080283` well below `0.8649`.

ADVANCED method

Note that the ADVANCED model takes the BASIC fit model `calFitCFBasic` or `fitCFBasic` as parameter for calibration or training respectively to do predictions that account for all the effects modeled in the BASIC model implementation. We also note that the column dimensions in P and Q correspond to the distinct number of users and movies respectively. We therefore need a small calibration set otherwise the calibration process will exhaust all the memory available and crash due to the large number of distinct users and movies in the training set. Thankfully, we did so already and built the calibration set containing a maneageable distinct number of 337 users and 596 movies and thus, help keep the size of P and Q under control.

That said, we calibrate the ADVANCED model using the following listing. We get the following best tune hyper-parameters: `K=2` latent dimensions, $\gamma = 0.06$ learning rate, $\lambda = 0.1$ and $\sigma = 0.1$.

```
##########################################################################################
## Calibrate the CF ADVANCED model on the calibration set (subset of the training edx set).
## Here I look for the best hyper-parameters that fit the advanced model on a small subset
## of the training data.
##########################################################################################

tic('ADVANCED: calibrating the model')
# set the seed again
portable.set.seed(1)
```

```
## Warning in set.seed(seed, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
control <- trainControl(method = "cv",
                        search = "grid",
                        number = 10,
                        p = .9,
                        allowParallel = TRUE,
                        verboseIter = TRUE)
calFitCFAdv <- train(x = calibration_set,
                     y = calibration_set$rating,
                     method = cFAdv,
                     trControl = control,
                     fitCFBasic = calFitCFBasic)
```

```
## Aggregating results
## Selecting tuning parameters
## Fitting K = 2, gamma = 0.06, lambda = 0.1, sigma = 0.1 on full training set

## Warning: Setting row names on a tibble is deprecated.
```

```
toc()
```

```
## ADVANCED: calibrating the model: 54.879 sec elapsed
```

```
## The bestTune model found is:
stopifnot(calFitCFAdv$bestTune$K == 2)
stopifnot(calFitCFAdv$bestTune$gamma == 0.06)
stopifnot(calFitCFAdv$bestTune$lambda == 0.1)
stopifnot(calFitCFAdv$bestTune$sigma == 0.1)
```

At this point we're ready to train the `cFAdv` model on the training edx set (note that SGD should normally require a small number of random samples) and compute the out of sample RMSE on the validation set. In this case, convergence is tracked i.e. `trackConv=TRUE` using `perTrack=0.003` or 0.3% of the 150k random samples i.e. 450 samples. We then can see a plot of how the RMSE decreases as the low-rank matrix factorization ADVANCED model is trained with more random samples.

```
##########################################################################################
## Fit the best model found to the complete training edx set (note that technically SGD
## employs a subset and not all the training data).
##########################################################################################

maxIter <- 150000
cat(sprintf("ADVANCED - training on the full edx set using %.2f%% random samples\n",
            100*maxIter/nrow(train_set)))
```

```
## ADVANCED - training on the full edx set using 1.67% random samples
```

```
tic('ADVANCED: training model on the full trainig data set')
# set the seed again
portable.set.seed(1)
```

```
## Warning in set.seed(seed, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
fitCFAdv <- train(x = train_set,
                  y = train_set$rating,
                  method = cFAdv,
                  trControl = trainControl(method = "none"),
                  tuneGrid = calFitCFAdv$bestTune,
                  maxIter = maxIter,
```

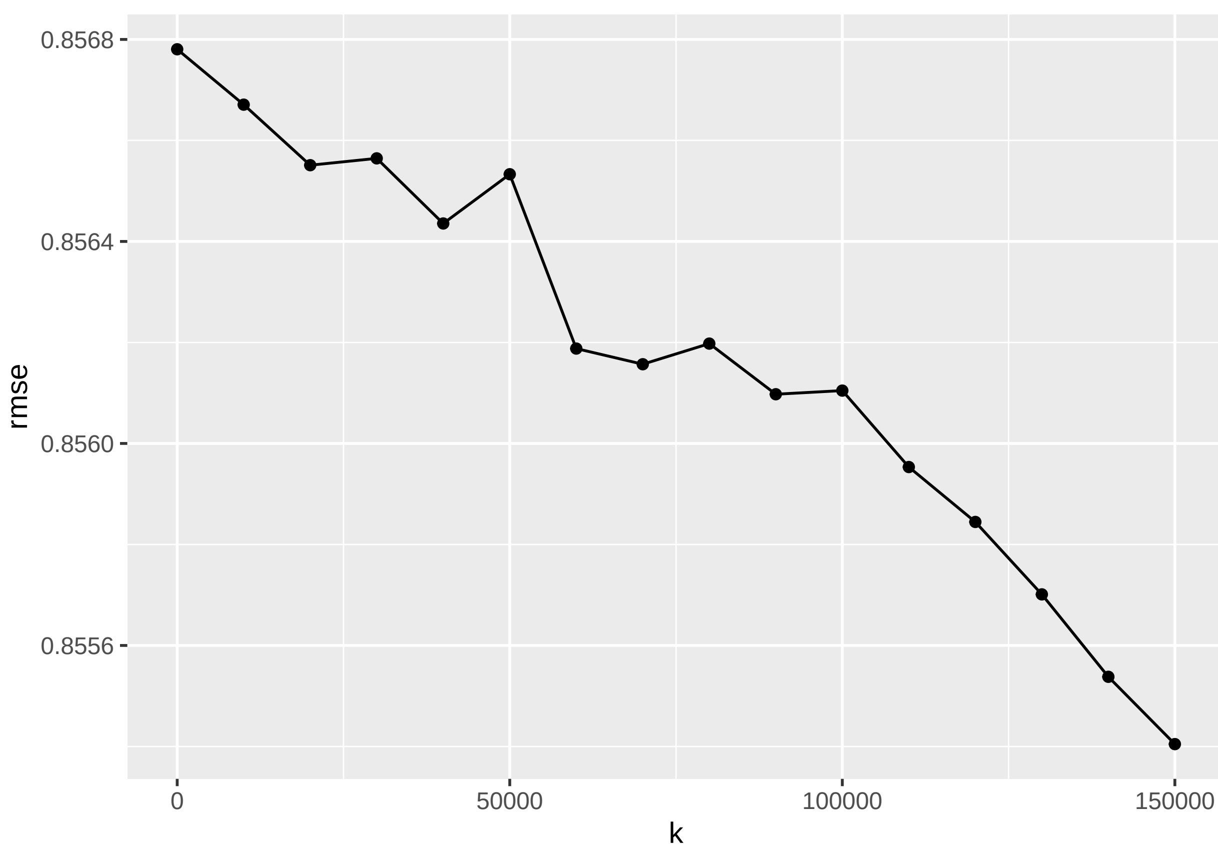
```
                  trackConv = TRUE,
                  perTrack = 0.003,
                  iterBreaks = 10000,
                  fitCFBasic = fitCFBasic)
```

```
## the RMSE at k=10000 is 0.856670867
## the RMSE at k=20000 is 0.856550984
## the RMSE at k=30000 is 0.856564498
## the RMSE at k=40000 is 0.856435546
## the RMSE at k=50000 is 0.856533077
## the RMSE at k=60000 is 0.856187907
## the RMSE at k=70000 is 0.856156884
## the RMSE at k=80000 is 0.856197722
## the RMSE at k=90000 is 0.856097460
## the RMSE at k=100000 is 0.856104641
## the RMSE at k=110000 is 0.855953330
## the RMSE at k=120000 is 0.855844602
## the RMSE at k=130000 is 0.855701160
## the RMSE at k=140000 is 0.855537935
## the RMSE at k=150000 is 0.855404698
```

```
toc()
```

```
## ADVANCED: training model on the full trainig data set: 300.849 sec elapsed
```

```
# plot the convergence for the advanced model training on the full edx
fitCFAdv$finalModel$rmse_hist %>%
  ggplot(aes(k, rmse)) + geom_point() + geom_line()
```



```r
##########################################################################################
## Finally compute the RMSE for the ADVANCED model on the validation set.
##########################################################################################

## VALIDATION SET ACCESS ALERT! accessing the validation set to compute RMSE.
tic("ADVANCED: predicting ratings on the full validation set")
predicted_ratings <- predict(fitCFAdv, validation_set)
rmse_val <- RMSE(predicted_ratings, validation_set$rating)
cat(sprintf("ADVANCED - RMSE on validation data is %.9f", rmse_val))
```

```
## ADVANCED - RMSE on validation data is 0.864173163
```

```r
stopifnot(abs(rmse_val - 0.864173163) < 1e-9)
toc()
```

```
## ADVANCED: predicting ratings on the full validation set: 216.873 sec elapsed
```

The final RMSE obtained on the validation set using the ADVANCED `fitCFAdv` model (calibrated and trained only on the full edx training set) is `RMSE=0.864173163` well below `0.8649` too but not better than the BASIC approach. A possible reason for this result may be that we'd need many more iterations or random samples to reach a better RMSE as we have ran the algorithm on only 1.67% of the edx training data i.e. `maxIter=150000`.

Conclusion

In this solution to the MovieLens project two models have been explored BASIC and ADVANCED. The BASIC model builds on top of the Data Science course book approach with extensions to account for genres and temporal effects i.e. day of the week or day of the month and the changes in rating patterns over time. The most interesting finding was the b_w effect where older movies rated most recently depict a clear higher ratings pattern and the possible explanations actually make sense e.g. users rating those older movies were already positively biased towards those movies before watching them. Smoothing the b_w effect proved to be a good predictive component of the models and it was very positive to realize during calibration a span for a smoother fit (less overfitting). Both models BASIC and ADVANCED reached a very good RMSE level of `0.864080283` and `0.864173163` respectively in the out of sample validation set.

The ADVANCED method based on low-rank matrix factorization using SGD was simple to implement but harder to fine-tune for reaching better RMSE levels than the BASIC method. These are some possible improvements to reach better RMSE levels with the ADVANCED method implementation:

1. Add support for batch updates instead of one sample at the time, faster training will help reaching higher quality models (with lower RMSEs).
2. There is a great potential for fully parallelizing mutually exclusive sub-segments of P and Q for faster training.
3. Implement better convergence criteria e.g. dynamically tune the γ learning rate parameter.

The following solutions were explored without success, either calibration or training became impractical or the obtained RMSE gains weren't promising enough:

1. Generating "one hot" encoded version of the genres i.e. `splitstackshape::cSplit_e(train_set , "genres", "|", type = "character", fill = 0, drop = TRUE)` and then fitting the genres using `lm`, `glm`, etc.
2. Having the ADVANCED model learn all the effects and not only the user-movie interactions after standard scaling (z-scoring) the ratings.
3. Having the ADVANCED model learn from samples sorted by absolute residual value in descending order.