HarvardX: PH125.9x

Data Science: Capstone

MovieLens recommendation system

Franz Gruber

06/06/2020

Introduction

Many companies use **recommendation systems** (or **recommender systems**[1]) to predict what *items* (e.g. movies, groceries, music, ...) a *user* might prefer, based on previous user ratings. In 2006, the streaming platform Netflix offered $1,000,000 in a challenge to enhance their recommendation algorithm (called *Cinematch*) by 10% or more[2]. Here, we want to generate a competetive recommendation system (a machine learning model) to predict a user's movie rating. For this task a smaller subset of the MovieLens dataset generated by the GroupLens research lab[3] will be used. This subset includes 10 million ratings, of 10,000 Movies by 72,000 users (release date 01/2009[4]). We will use the root mean squared error (**RMSE**), as read-out to compare and improve our models. We start building a simple regression model, add several effects (or *biases*), apply regularization to add a penaltiy to our *biases* and finally try out a more sophisticated approach using matrix factorization.

Methods

The full code is provided in the *movielens.R* file. Only show essential code snippets will be shown here.

Dataset generation

This section is addapted from *HarvardX: PH125.9x*

The *MovieLens 10M* dataset was downloaded from https://grouplens.org/datasets/movielens/10m. A `ratings` table containing *userId, movieId, rating, timestamp* columns can be merged with a `movies` table containing *movieId, titles, genres* on *movieId* columns. The resulting `data.frame` is partitioned into `edx` (90 % of the data; used for model training) and `validation` (10 % of data; used for final model evaluation) using the `createDataPartition` function of the `caret` package.

Data cleaning

Both, `edx` as well as the `validation` contain a *timestamp* column, which contains *UNIX* time format. This can be transformed into human readable format using the `as_datetime` function of the `lubridate` package.

[1] https://en.wikipedia.org/wiki/Recommender_system

[2] more information on: https://en.wikipedia.org/wiki/Netflix_Prize

[3] Department of Computer Science and Engineering at the University of Minnesota, Twin City; https://grouplens.org/about/what-is-grouplens

[4] https://grouplens.org/datasets/movielens/10m

In addition we will generate a new column *week* using the `round_date` function of the `lubridate` package, setting the parameter `unit = 'week'`.

Table 1: 'edx' after data cleaning

userId	movieId	rating	timestamp	title	genres	week
1	122	5	1996-08-02 11:24:06	Boomerang (1992)	Comedy|Romance	1996-08-04
1	185	5	1996-08-02 10:58:45	Net, The (1995)	Action|Crime|Thriller	1996-08-04
1	292	5	1996-08-02 10:57:01	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller	1996-08-04
1	316	5	1996-08-02 10:56:32	Stargate (1994)	Action|Adventure|Sci-Fi	1996-08-04
1	329	5	1996-08-02 10:56:32	Star Trek: Generations (1994)	Action|Adventure|Drama|Sci-Fi	1996-08-04
1	355	5	1996-08-02 11:14:34	Flintstones, The (1994)	Children|Comedy|Fantasy	1996-08-04

Data Exploration

Our `edx` dataset consists of 9000055 *rows* (or user ratings) and 7 *columns*. There are no missing values in any of the columns:

```
is.na(edx) %>% any()
```

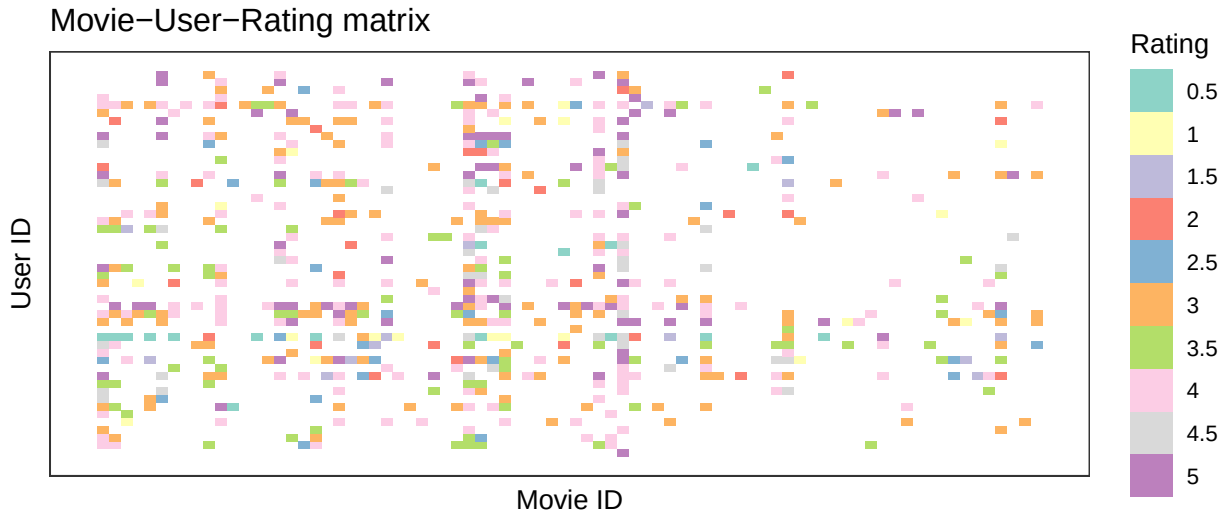
```
## [1] FALSE
```

The number of users and movies are summarized below. We can see that if we would multiply $users \times movies$, we would get a much higher number of user ratings, than we acutally have. This shows that now every user rated every movie.

Table 2: User/Movie Summary

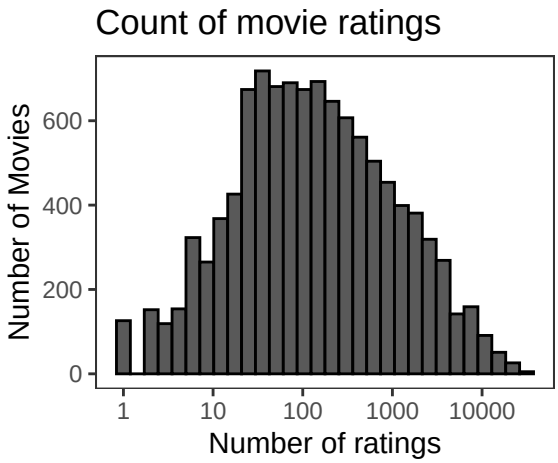
movies	users	movie_times_users
10677	69878	746087406

We can visualize this by selecting 50 random users and any *movieId* below 100:



Data in the `edx` set contains user ratings from 1995-01-09 11:46:49 to 2009-01-05 05:02:16.

The number of movie ratings is visualized as a histogram below. We can see that some movies get very few ratings (some only 1 rating), while other movies get $>$ 10,000 ratings.

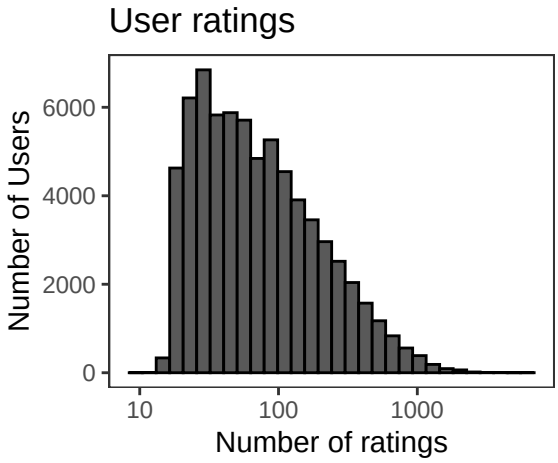


The `median` and `mean` number of movie ratings are listed below. The median is more robust against movies with extremely high number of ratings.

```
edx %>%
  count(movieId) %>%
  summarize(mean = mean(n),
            median = median(n))
```

```
## # A tibble: 1 x 2
##    mean median
##   <dbl>  <int>
## 1  843.    122
```

We can also summarize the number of user ratings per user as a histogram and see that some users rate more movies than others:



The `median` and `mean` number of user ratings are listed below.

```
edx %>%
  count(userId) %>%
  summarize(mean = mean(n),
            median = median(n))
```

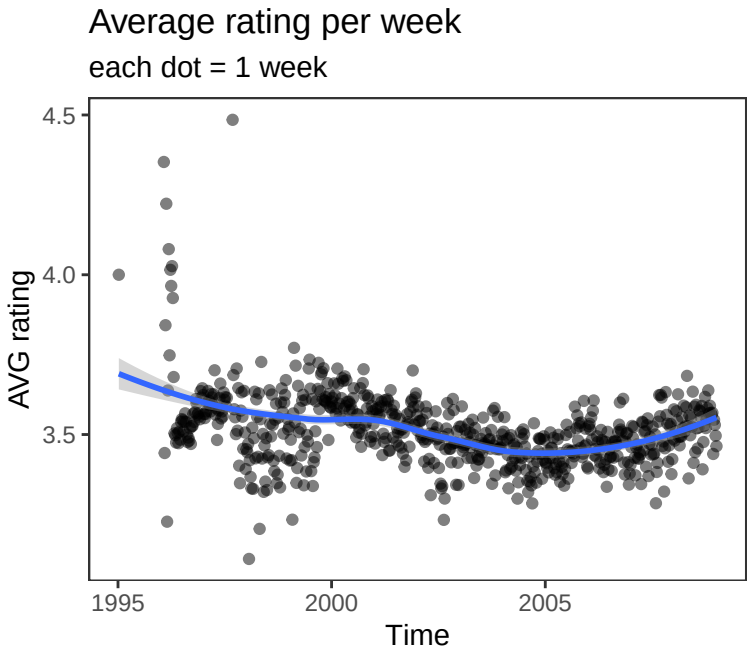
```
## # A tibble: 1 x 2
##    mean median
##   <dbl>  <dbl>
## 1  129.     62
```

Looking at the top 10 movies with the highest number of ratings, we can see *Pulp Fiction* with the most user ratings, amongst other popular movies.

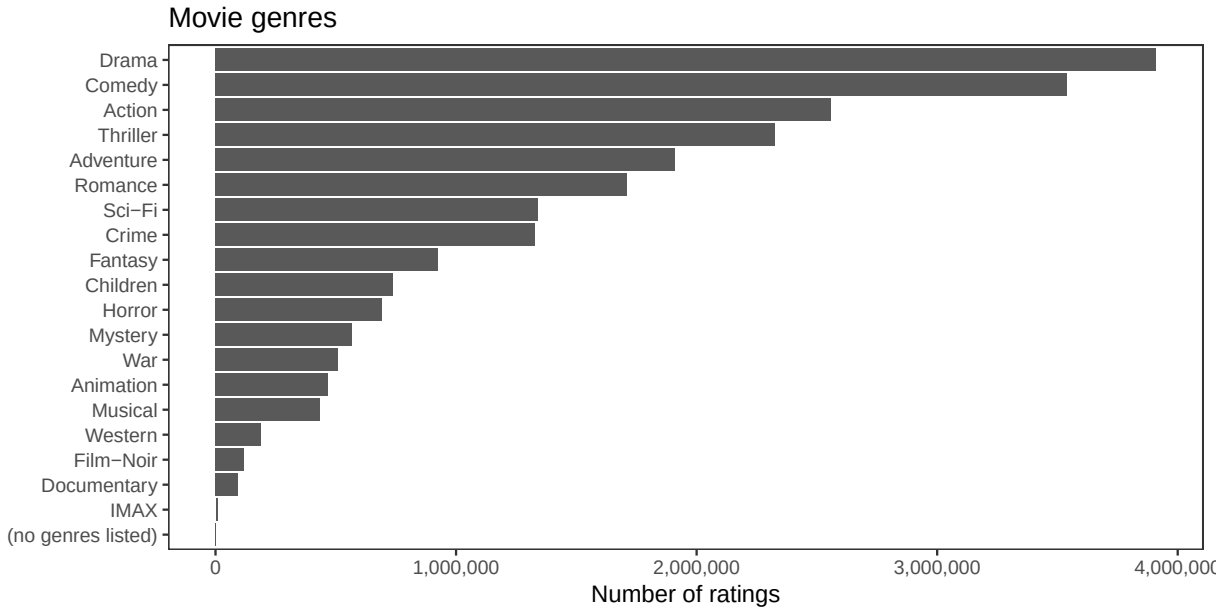
Table 3: Top 10 Movies with highest number of ratings

movieId	title	n
296	Pulp Fiction (1994)	31362
356	Forrest Gump (1994)	31079
593	Silence of the Lambs, The (1991)	30382
480	Jurassic Park (1993)	29360
318	Shawshank Redemption, The (1994)	28015
110	Braveheart (1995)	26212
457	Fugitive, The (1993)	25998
589	Terminator 2: Judgment Day (1991)	25984
260	Star Wars: Episode IV - A New Hope (a.k.a. Star Wars) (1977)	25672
150	Apollo 13 (1995)	24284

Next we want to see, how the weekly average user rating changed over time between 1995-2009. We can see that there are some variation in weekly average user ratings over time.



Finally, we will have a look at movie genres. We will separate the *genres* column using the `separate_rows` function of the `tidyr` package. As with the number of movie ratings and user ratings, we can see that some genres are more popular than others:



Insights gained

This section contained typical tasks of a data science project: gathering, cleaning and exploration of data. The `Movielens` dataset is available online, so there was no need to perform any web scraping or pdf parsing. The data set comes in tidy format[5]. For our purpose, we only had to transform the *UNIX* time into a human readable format using the `lubridate` package, and split the *genres* column into single genres.

After data cleaning, we explored the data using plots and summary tables. We saw that some movies get more ratings than others; some users rate more than others; there seems to be fluctuations in weekly average movie ratings over time; some genres receive more ratings than others. We can use some of this information gained for our main goal, which is to generate a competitive movie recommendation system.

Loss function

Our aim is to create a movie recommendation system. We will use a modeling approach and for this we require a loss function, which informs us about our prediction error. Furthermore, we use the information of our loss function to improve our model.

Here, we will use the residual mean squared error (RMSE), which is a popular choice in the field of machine learning. The RMSE is defined as:

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

where N is the number of user/movie combinations, $\hat{y}_{u,i}$ is the predicted rating of user u on movie i and $y_{u,i}$ is the actual prediction of user u on movie i. Our aim is to minimize the RMSE during model optimization.

Alternatives to the RMSE would be the *mean absolute error* (MAE), which will not be used here.

[5]https://www.jstatsoft.org/article/view/v059i10

Data splitting for modelling

Before we start building our movie recommendation model, we have to split our `edx` dataset again into a train set and a test set, which we will use to generate our model. We will randomly split our dataset using 80% for the train set and 20% for the test set using the `createDataPartition` function of the `caret` package. We also need to filter out users and movies from our test set, which are not part in our train set. Our final model will then be trained on the full `edx` set and will use the `validation` set to evaluate the model's performance.

Modeling approach

Average rating model

The simplest model would be to predict the same rating for all movies independet of users. Such a model would look like this:

$$Y_{u,i} = \mu + \epsilon_{u,i}$$

where $Y_{u,i}$ is the rating by user **u** of movie **i**, μ is the *true* rating for all movies and $\epsilon_{u,i}$ are independent errors sampled from the same distribution centered around 0. The estimate which minimizes the RMSE would be μ. We will use the average of all user ratings $\hat{\mu}$.

Such a model is not expected to be very reliable and needs to be improved using insights gained from the data exploration section.

Movie Bias Model

We have seen that some movies get rated more often than others. This is intuitive as some movies are more popular than others. We can include such an effect (or *bias*) in our model and see how it improves prediction:

$$Y_{u,i} = \mu + b_i + \epsilon_{u,i}$$

where b_i is the movie bias.

We will estimate the movie bias b_i using the least square estimate $\hat{b}_i$, which is the average over all movie ratings minus the estimated mean: $Y_{u,i} - \hat{\mu}$.

In order to calculate our movie prediction, we have to add our estimated movie effect, $\hat{b}_i$, to the average rating $\hat{\mu}$.

User Bias

We will also add another bias term. We observed that user ratings can differ as well - some users are more active than others. We will include a user effect (or *bias*) into our model:

$$Y_{u,i} = \mu + b_i + b_u + \epsilon_{u,i}$$

where b_u is the user bias.

Time Bias

We saw that the average weekly ratings was time dependent. We can add a time effect to our model, using $d_{u,i}$, which is the day for the rating of *user* u on *movie* i. We will use a smoothing function $f(d_{u,i})$, which we will define as b_t and average over a full week to include this term to our model:

$$Y_{u,i} = \mu + b_i + b_u + b_t + \epsilon_{u,i}$$

Regularization

Regularization allows to penalize large estimates from small sample size. One example are movies, which have only one rating. Their ratings without being penalized, would lead to errors and over-fitting of our model.

Here we want to minimize the following equation:

$$\frac{1}{N}\sum_{u,i} = (y_{u,i} - \mu - b_i - b_u - b_t + \lambda(\sum_i b_i^2 + \sum_u b_u^2 + \sum_t b_t^2)$$

The first part of this equation is the least squares estimate, while the second part of the equation penalizes each bias effect.

We can calculate $\hat{b}_i$ as

$$\hat{b}_i(\lambda) = \frac{1}{\lambda + n_i}\sum_{u=1}^{n_i}(Y_{u,i} - \hat{\mu})$$

$\hat{b}_u$ as

$$\hat{b}_u(\lambda) = \frac{1}{\lambda + n_i}\sum_{u=1}^{n_i}(Y_{u,i} - \hat{\mu} - \hat{b}_i)$$

$\hat{b}_t$ as

$$\hat{b}_t(\lambda) = \frac{1}{\lambda + n_i}\sum_{u=1}^{n_i}(Y_{u,i} - \hat{\mu} - \hat{b}_i - \hat{b}_u)$$

and we will use cross-validation to pick a λ, which gives us the smallest RMSE value.

Matrix factorization

The winning team of the Netflix prize used a model that was exploiting *matrix factorization*. We saw in the data exploration section that not every user rated every movie. We transformed our `edx` data frame into a matrix with user ratings as rows and movies as columns and observed a **sparse matrix** with missing entries (user ratings) - not every user rated every movie.

Matrix factorization allows to infer information from high-dimensional data e.g. genres are rated better than other genres, or a certain user preference for certain genres. There are several R packages available e.g. `recommenderlab` or `recosystem`. We will use the `recomendersystem` package here for convenience (vignette of the `recosystem` package[6]) as this package could be run on a laptop with 8 GB RAM relatively

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

effciently. This package is a wrapper of the `LIBMF` library[7], which uses parallel matrix factorization and is open source. The underlying idea is to predict missing values in a rating matrix $R_{m\times n}$ with m users and n movies as an approximation of two matrices of lower dimension $P_{k\times m}$ and $Q_{k\times n}$, such that

$$R \approx \mathbf{P}^\top Q$$

.

Here, we will follow the vignette of the `recosystem` package and use the same values for parameter optimization.

Results

We will now look at the performance of the different models using first our train and test sets generated from `edx`, and then compare the results side-by-side, using the full `edx` for model training and the `validation` set generated at the very beginning for model validation. Our ultimate aim is to get an RMSE below 0.8649.

Average only Model

As expected, our simple model has a very weak performance.

```
mu <- train_set$rating %>% mean()
mu
```

```
## [1] 3.512386
```

```
rmse_df <- tibble(
  Method = 'Average only',
  RMSE = RMSE(test_set$rating, mu)
)
```

Method	RMSE
Average only	1.059913

We will include several effects (*biases*) to our model in the following sections and see if and how an added effect improves the RMSE of our model.

Movie Bias Model

The first effect to include will be the movie effect:

```
movie_avg <- train_set %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu))


predictions <- test_set %>%
  left_join(movie_avg, by = 'movieId') %>%
  mutate(pred = mu + b_i)
```

[7] http://www.csie.ntu.edu.tv/~cjlin/libmf

```
rmse_df <-  rmse_df %>%
  rbind(tibble(
    Method = 'Movie Effect Model',
    RMSE = RMSE(test_set$rating, predictions$pred)
    ))
```

Method	RMSE
Average only	1.0599132
Movie Effect Model	0.9437213

We see that this improves our model by ~10%.

User Bias Model

The second effect to include will be the user effect:

```
user_avg <- train_set %>%
  left_join(movie_avg, by = 'movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i))

predictions <- test_set %>%
  left_join(movie_avg, by = 'movieId') %>%
  left_join(user_avg, by = 'userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  pull(pred)

rmse_df <- rmse_df %>%
  rbind(tibble(
    Method = 'Movie + User Effect Model',
    RMSE = RMSE(test_set$rating, predictions)
    ))
```

Method	RMSE
Average only	1.0599132
Movie Effect Model	0.9437213
Movie + User Effect Model	0.8660626

As we can see this reduced our RMSE further.

Time Bias Model

Next we will add a time effect to our model:

```
weeks_avg <- train_set %>%
  left_join(movie_avg, by = 'movieId') %>%
  left_join(user_avg, by = 'userId') %>%
  group_by(week) %>%
  summarize(b_t = mean(rating - mu - b_i - b_u))
```

```
predictions <- test_set %>%
  left_join(movie_avg, by = 'movieId') %>%
  left_join(user_avg, by = 'userId') %>%
  left_join(weeks_avg, by = 'week') %>%
  mutate(pred = mu + b_i + b_u + b_t)

rmse_df <- rmse_df %>%
  rbind(tibble(
    Method = 'Movie + User + Time Effect Model',
    RMSE = RMSE(test_set$rating, predictions$pred)
  ))
```

Method	RMSE
Average only	1.0599132
Movie Effect Model	0.9437213
Movie + User Effect Model	0.8660626
Movie + User + Time Effect Model	0.8659901

We can see that the improvement our RMSE is only ~0.1%, which is not that great.

Model regularization

We will now apply regularization to the model with the lowest RMSE, including all effects. We will use a wrapper function, which we can reuse later for the `validation` set. In this wrapper function we will perform cross-validation to find a λ which gives the lowest RMSE.

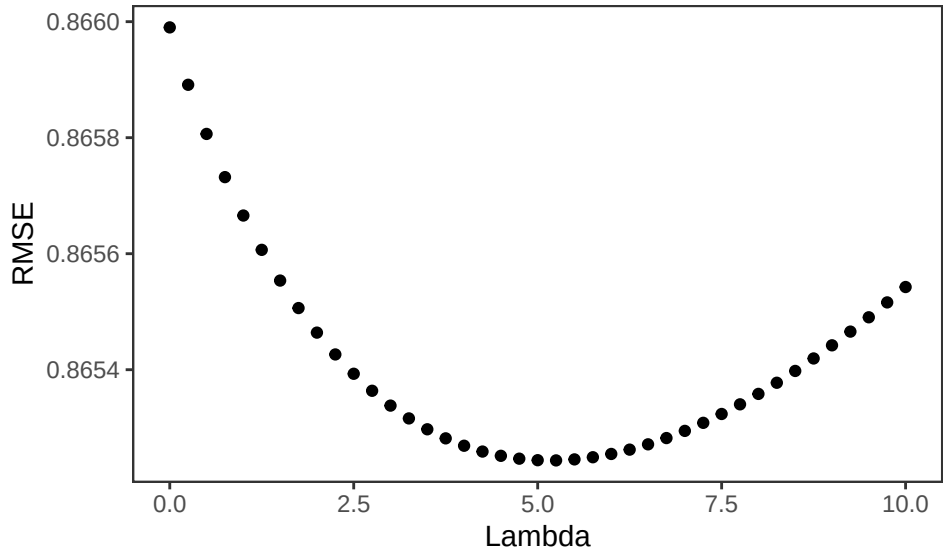
```
lambdas <- seq(0, 10, 0.25)

regularization_wrapper <- function(trainSet, testSet, lambda = lambdas) {

  rmses <- sapply(lambda, function(l){

  mu <- trainSet$rating %>% mean()

  b_i <- trainSet %>%
    group_by(movieId) %>%
    summarize(b_i = sum(rating - mu)/(n() + l))

  b_u <- trainSet %>%
    left_join(b_i, by = 'movieId') %>%
    group_by(userId) %>%
    summarize(b_u = sum(rating - b_i - mu)/(n() + l))

  b_t <- trainSet %>%
    left_join(b_i, by = 'movieId') %>%
    left_join(b_u, by = 'userId') %>%
    group_by(week) %>%
    summarize(b_t = sum(rating - mu - b_i - b_u)/ (n() + l))

  predictions <- testSet %>%
    left_join(b_i, by = 'movieId') %>%
```

```
    left_join(b_u, by = 'userId') %>%
    left_join(b_t, by = 'week') %>%
    mutate(pred = mu + b_i + b_u + b_t) %>%
    pull(pred)

  return(RMSE(testSet$rating, predictions))

  })

}
```

We can apply our wrapper to the train and test sets:

```
reg_rmses <- regularization_wrapper(train_set, test_set, lambdas)


rmse_df <- rmse_df %>%
  rbind(tibble(
    Method = 'Regularization Movie/User/Time effect',
    RMSE = min(reg_rmses)))
```

and visualize the how different values of lambda perform:



We see that a λ of 5.25 solves our model with the lowest RMSE.

Method	RMSE
Average only	1.0599132
Movie Effect Model	0.9437213
Movie + User Effect Model	0.8660626
Movie + User + Time Effect Model	0.8659901
Regularization Movie/User/Time effect	0.8652437

We can see that our RMSE is only taking small steps towards our aim of 0.8649 - can we do better?

Recosystem Matrix Factorization

We have evolved a model, which got close to our aim of an RMSE < 0.8649. To break through this barrier we will now try a more sophisticated approach using *matrix factorization*. We will use an implementation within the `recosystem` package. As with the regularization model, we will write a wrapper function, which allows to reuse it later with the `validation` set.

```
recosystem_wrapper <- function(trainSet, testSet) {

  set.seed(1986, sample.kind = "Rounding")

  train_data <- with(trainSet, data_memory(user_index = userId,
                                           item_index = movieId,
                                           rating = rating))


  test_data <- with(testSet, data_memory(user_index = userId,
                                         item_index = movieId,
                                         rating = rating))
  #initialize model object
  r <- Reco()

  #parameter optimization
  opts <- r$tune(train_data,
                 opts = list(dim = c(10,20,30),
                             lrate = c(0.1, 0.2),
                             costp_l1 = 0,
                             costq_l1 = 0,
                             nthread = 4,
                             niter = 10,
                           progress = FALSE))

  #train model using optimized parameters
  r$train(train_data, opts = c(opts$min, nthread = 1, niter = 20, verbose=FALSE))

  #generate model predictions
  reco_pred <- r$predict(test_data, out_memory())

}

recosystem_train <- recosystem_wrapper(train_set, test_set)
```

Method	RMSE
Average only	1.0599132
Movie Effect Model	0.9437213
Movie + User Effect Model	0.8660626
Movie + User + Time Effect Model	0.8659901
Regularization Movie/User/Time effect	0.8652437
Recosystem Matrix Factorization	0.7910284

Indeed the matrix factorization based model managed to get an RMSE below 0.8649! Let's move on to model validation and see, how well our model performs with the `validation` set.

Model validation

We will test the performance of our regularization model and of the Matrix factorization based model, using our wrapper functions and the `validation` set:

```
reg_rmses_val <- regularization_wrapper(edx, validation, lambdas)

recosystem_val <- recosystem_wrapper(edx, validation)
```

Method	RMSE
Average only	1.0599132
Movie Effect Model	0.9437213
Movie + User Effect Model	0.8660626
Movie + User + Time Effect Model	0.8659901
Regularization Movie/User/Time effect	0.8652437
Recosystem Matrix Factorization	0.7910284
Regularization Model Validation	0.8646938
Recosystem Model Validation	0.7836277

We see that with the `validation` set, which was data our model has not seen before, we get below 0.8649 with both the regularization model as well as the `recosystem` matrix factorization based model.

Conclusion

In this project, we have built a machine learning model, which predicts user ratings of movies. We have used two approaches: evolving a linear regression model by adding and penalizing several *bias* terms and we have explored the `recosystem` package, which uses a matrix factorization based model. We can conclude that just by using a simple linear regression approach, we would have goten close to the *BellKor's Pragmatic Chaos* winning RMSE of 0.8567[8]. Our model has some limitations and could have been evolved even further. We saw that movie genres are rated different. Furthermore, the data set contains additional information e.g. the release date. One could implement such information and try to further improve the RMSE. Finally our linaer regression model demonstrates that such a lightweight model can be used as a starting point for a machine learning (*"why bother training complicated models, if they are not better than linear regression?"*).

Using the `recosystem` package we achieved an RMSE well below our aim of 0.8649. The package showed fast performance on a slim laptop and was easy to try out. It would be interesting to test this package on other data sets in the future and see its performance. Using such a sophisticated model, demonstrates another common issue in machine learning: model interpretability. Although we achieved better performance (lower RMSE), we increased the black-box character as compared to the linear regression model.

[8] https://en.wikipedia.org/wiki/Netflix_Prize