Predicting Movie Ratings

Why Matrix Factorization Beats Additive Models

Moritz C. Schaaf

2020-04-22

Contents

Introduction

This report is part of the HarvardX Data Science Series[1] capstone project. The R and R Markdown codes are also available on GitHub[2].

In this report I will be creating a movie recommendation system. Recommendation systems - sometimes also called recommender platform or engine - try to quantify the interest a particular user is going to have for a particular item. Well known applications are for example the suggestions of YouTube, Spotify or Netflix. However, even online newspapers or public libraries[3] have started to incorporate recommendation engines. In 2006, Netflix raised public awareness when it held an open competition[4] for the best algorithm to predict movie ratings. The grand price of US$1,000,000 was awarded three years later, as a team managed to outperform Netflix's own algorithm by over 10%. But why bother? The gain for providers like Netflix is enormous: Not only do they increase the sales, but such systems do also decrease the effort for shopping (or selection of the desired item in general). Therefore, both customer and corporation benefit from sophisticated recommendation systems.

The assignment for this report is similar to the Netflix challenge: A target prediction accuracy has to be beaten without using the dataset for the final evaluation of the created prediction algorithm.

Methods: Data and Pre-processing

The data used is from MovieLens, a research lab of the University of Minnesota, and constists of about 10 million ratings[5]. This is only a subset of the full dataset[6], which also is publicly available for download.

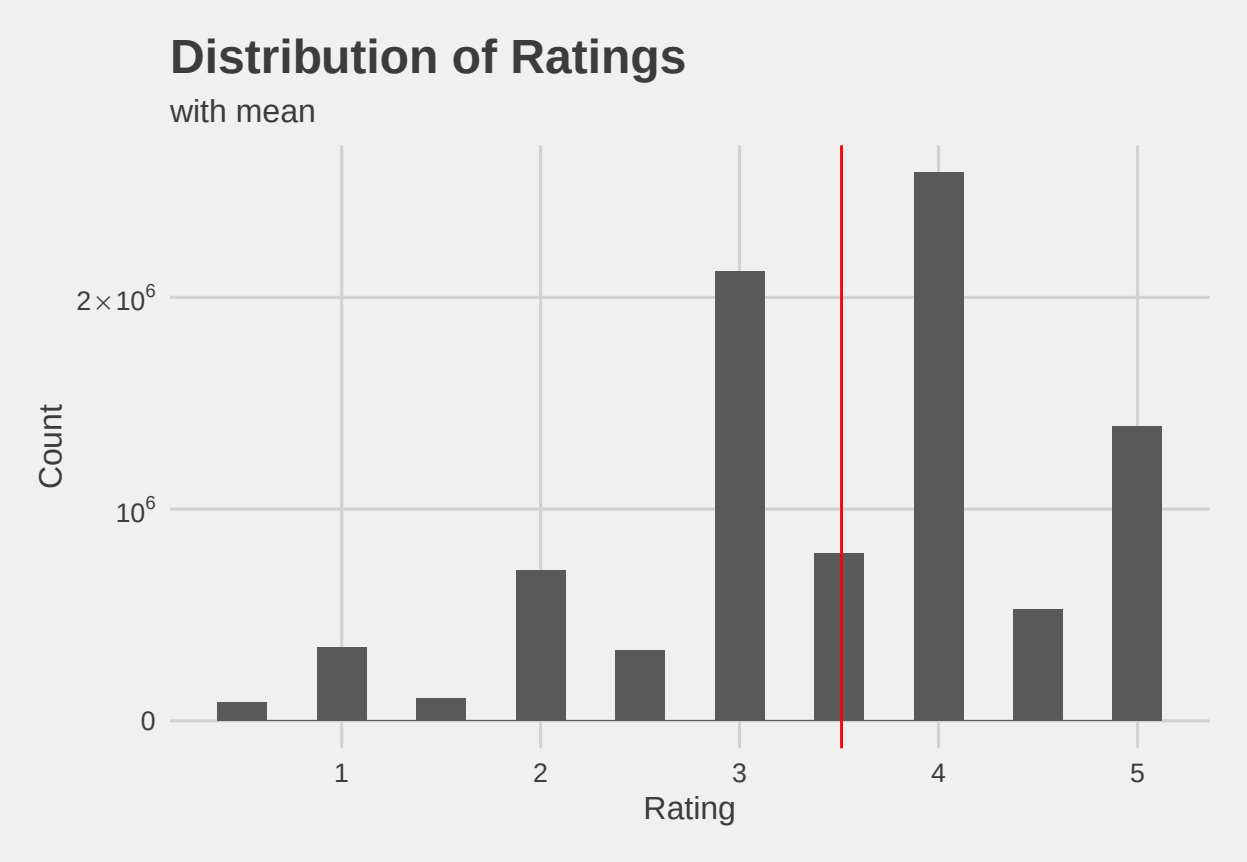
The dataset includes ratings of 69878 users and 10677 movies. 10 percent of the dataset is used to create the dataset for the final validation. After ensuring that all users and movies in the validation dataset are also in the one used to develop the recommendation system, the number of ratings totals to 9000055. For each rating, information about the user ID, the time of the rating, the title of the movie and the genre is given:

userId	movieId	rating	timestamp	title	genres
1	122	5	838985046	Boomerang (1992)	Comedy|Romance
1	185	5	838983525	Net, The (1995)	Action|Crime|Thriller
1	292	5	838983421	Outbreak (1995)	Action|Drama|Sci-Fi|Thriller
1	316	5	838983392	Stargate (1994)	Action|Adventure|Sci-Fi
1	329	5	838983392	Star Trek: Generations (1994)	Action|Adventure|Drama|Sci-Fi
1	355	5	838984474	Flintstones, The (1994)	Children|Comedy|Fantasy

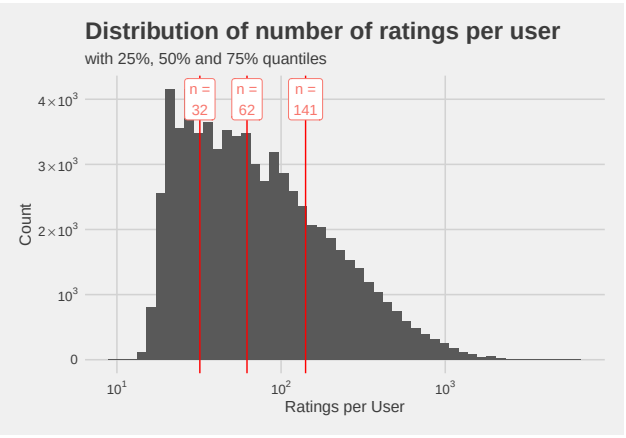
We notice two things: First, the `title` column includes information on the release year of the movie, which we could possibly use. Second, the `genre` column list all genres that apply to this particular movie, separated by "|". This could cause problems later on, so we will inspect this later on. However, first we will take a look at the distribution of our variables:

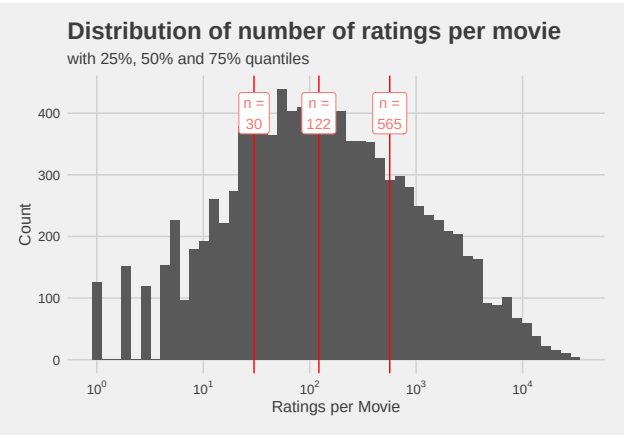
Ratings can range from 5 *stars* (indicating a excellent rating) to 0.5 *stars* (indicating a terrible movie). However, the ratings are not normaly distributed and half-star ratings appear to be very rare.

[1] https://www.edx.org/professional-certificate/harvardx-data-science
[2] https://github.com/mc-schaaf/edX-Harvard-DataScience-Capstone
[3] http://www.bibtip.com/en
[4] https://en.wikipedia.org/wiki/Netflix_Prize
[5] https://grouplens.org/datasets/movielens/10m/
[6] https://grouplens.org/datasets/movielens/latest/

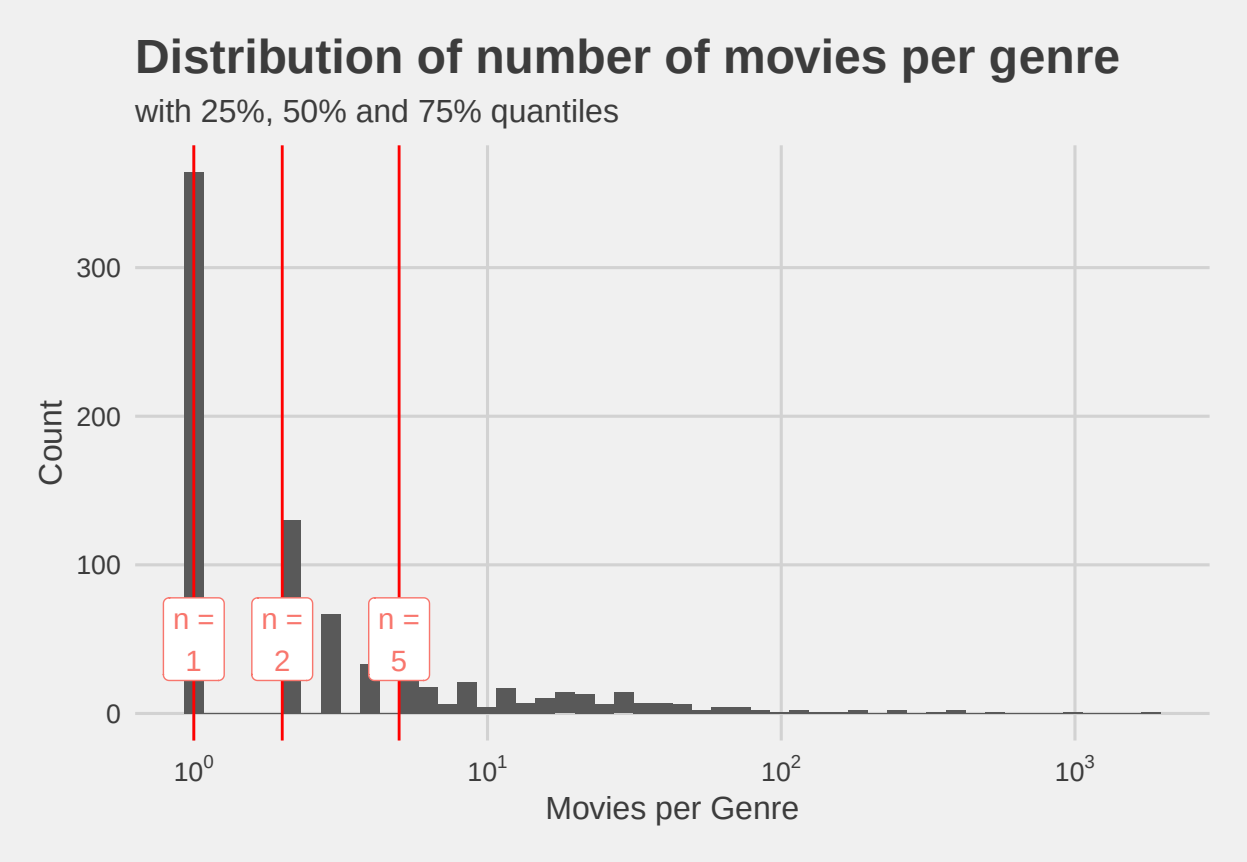


These ratings are also not evenly distributed between users or movies. Blockbuster movies may get tens of thousands ratings whilst half the movies have less than 125 ratings. The same applies for users. Some are very active and produce tousands of ratings, but most have rated less than one hundred movies. However, after closer inspection we notice that whilst the number of ratings per movie seems log-normally distributed - including movies with only one or two ratings - the number of ratings per user seems to be "cut off" at the lower tail. One possible explanation for this could be that only active users are included in the 10M dataset.





The `genres` column consists of 797 unique categories. Most of the categories, however, apply to only very few movies:



When inspecting the content of the `genres` that only include one movie title the reason for this becomes clear.

```
edx %>% group_by(genres, movieId) %>% count() %>%
group_by(genres) %>% count() %>% filter(n < 2) %>% head()
```

genres	n
(no genres listed)	1
Action|Adventure|Animation|Children|Comedy|Fantasy	1
Action|Adventure|Animation|Children|Comedy|IMAX	1
Action|Adventure|Animation|Children|Fantasy	1
Action|Adventure|Animation|Children|Sci-Fi	1
Action|Adventure|Animation|Comedy|Drama	1

Multiple genres are applied to most movies but separated by "|". To be able to use the information of the genre, this has to be converted to an easier format.

Considering the first impressions above, we have to do a bit of data pre-processing. We want to extract information about the release date from the `title` variable.

```
# make a small subset that contains only 1 rating per movie
```

```
one_per_film <- edx %>% group_by(title) %>% top_n(1, timestamp)

# set up extraction pattern
year_pattern <- "\\([0123456789]{4}\\)$"

# Use pattern
one_per_film <- one_per_film %>% ungroup() %>% mutate(
  year = str_extract(title, year_pattern) %>% str_remove_all("\\(|\\)") %>% as.integer()
  ,title = str_remove(title, paste0(" ", year_pattern))
)

# merge back together
edx <- edx %>% select(-title)
one_per_film <- one_per_film %>% select(-c(userId, timestamp, genres, rating))
edx <- full_join(edx, one_per_film, by = c("movieId"))

edx %>% head()
```

userId	movieId	rating	timestamp	genres	title	year
1	122	5	838985046	Comedy|Romance	Boomerang	1992
1	185	5	838983525	Action|Crime|Thriller	Net, The	1995
1	292	5	838983421	Action|Drama|Sci-Fi|Thriller	Outbreak	1995
1	316	5	838983392	Action|Adventure|Sci-Fi	Stargate	1994
1	329	5	838983392	Action|Adventure|Drama|Sci-Fi	Star Trek: Generations	1994
1	355	5	838984474	Children|Comedy|Fantasy	Flintstones, The	1994

Also, we want to wrangle the **genres** column into a matrix indicating whether a certain genre applies to a movie. Additionally, we will create a column which takes the first genre listed and defines it as the main genre. However, to keep the dataset manageable in size, we will not yet merge the matrix back to the original data frame:

```
# make a small subset that contains only 1 rating per unique sting in the genres-column
one_per_genres <- edx %>% group_by(genres) %>% top_n(1, timestamp)
unique_genres <- one_per_genres$genres %>% str_split("\\|") %>% unlist() %>% unique()

# Maybe the first genre listed is the main genre?
one_per_genres <- one_per_genres %>%
  mutate(main_genre = str_split_fixed(genres, "\\|", n = 2)[1])

# produce a logical matrix-like object if genre XY is in the genres column
for (i in 1:length(unique_genres)) {
  one_per_genres$V1 <- str_detect(one_per_genres$genres, unique_genres[i])
  one_per_genres <- one_per_genres %>% rename(!!unique_genres[i] := "V1")
}

# shorten data frames
one_per_genres <- one_per_genres %>%
  select(-c(userId, movieId, rating, timestamp, title, year))
```

```
# show the structure of the first few columns
one_per_genres[,1:6] %>% head()
```

genres	main_genre	Animation	Crime	Mystery	Action
Animation|Crime|Mystery	Animation	TRUE	TRUE	TRUE	FALSE
Action|Adventure|Comedy|Crime|Romance|Thriller	Action	FALSE	TRUE	FALSE	TRUE
Action|Animation|Comedy|Horror	Action	TRUE	FALSE	FALSE	TRUE
Documentary|Drama|War	Documentary	FALSE	FALSE	FALSE	FALSE
Action|Adventure|Horror	Action	FALSE	FALSE	FALSE	TRUE
Adventure|Children|Fantasy|Western	Adventure	FALSE	FALSE	FALSE	FALSE

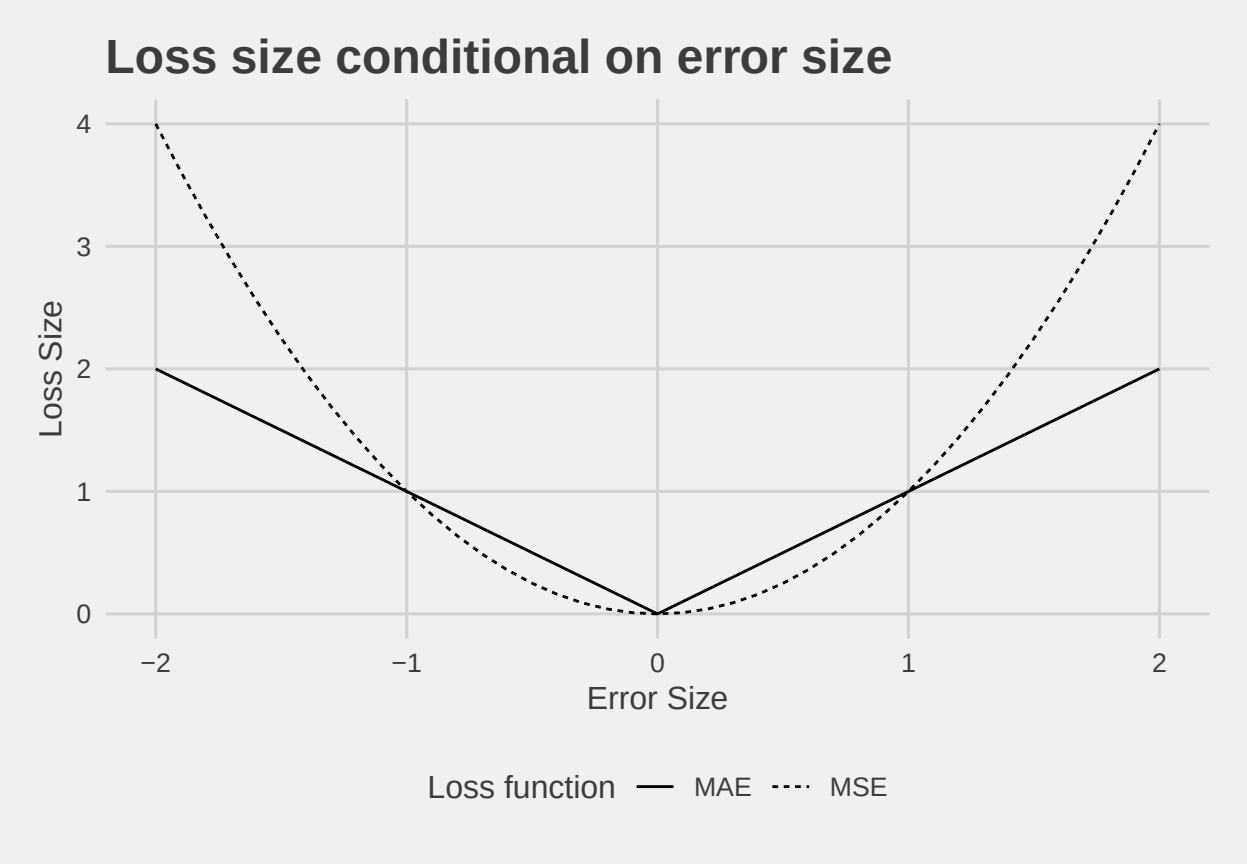
Methods: Quantification of Prediction Accuracy

For creating a well performing algorithm, one has define what a *good* or *bad* prediction is. Binarizing could be a possibility, thus splitting ratings into *likes* and *dislikes*. However, as we explored earlier, we are dealing with interval-scaled discrete ratings. Therefore, the size of the error can be taken as loss or costs. Two very commonly used loss functions are the mean absolute error (MAE) and mean sqared error (MSE). We define $y_{u,i}$ as the rating for movie i by user u with N being the number of user/movie combinations and denote our prediction with $\hat{y}_{u,i}$. The loss functions are then defined as:

$$\text{MAE} = \frac{1}{N} \sum_{u,i} |\hat{y}_{u,i} - y_{u,i}|$$

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

The difference is in the weight of error sizes: Whilst an error of size 1 results in a MAE and MSE of 1, an error of size 3 results in a MAE of 3 and MSE of 9. The MSE scales linear with the prediction errors. The MSE on the other hand weighs large errors especially high:



Whilst those two loss functions are easy to compute and have desireable mathematical properties, there are prediction tasks where loss functions might have different and asymmetric shapes. Imagine predicting the oxygen capacity a diver may need: Whilst bringing too much might result in heavier equipment, bringing too few might result in having to abort a scuba trip or even fatal accidents. However, the Netflix challenge used the typical error loss: they decided on a winner based on the root mean squared error (RMSE) on a test set. It has the same properties as the MSE (being heavily influenced by outliers for example) but is easier to interpret than the sqared version because it has the same unit as standard deviations and prediction errors. Thus, we will also use this common loss function and define as my personal project goal a RMSE of < 0.8567. This is the RMSE "BellKor's Pragmatic Chaos" achieved[7] in order to win the million-dollar-prize.

Methods: Building Different Models

Just like in the Netflix challenge, we will not touch the validation set until our final model is built. This prevents us from hillclimbing. However, to keep a "leaderboard" of the different models, we will again set aside 10 percent of our observations. The Netflix challenge used a similar approach, but mind the different semantics: I will be calling the training set `edx_train`, the leaderboard-calculation set `edx_test` and the final validation set `validation`. The training set consists of 8100065 ratings and the leaderboard set includes 899990 ratings. Let's define a function that computes our leaderboard RMSEs and do some baseline measurements!

[7] https://www.netflixprize.com/leaderboard.html

```r
fun_RMSE <- function(y_hat) {
  RSS <- sum((y_hat - edx_test$y)^2)
  RMSE <- sqrt(RSS/length(y_hat))
  return(RMSE)
}
```

Baseline Measurements

If we knew nothing at all about the data, we would have to guess every possible outcome with the same probability:

```r
suppressWarnings(set.seed(1, sample.kind = "Rounding"))
possible_outcomes <- c((1:10)/2)
y_hat <- sample(possible_outcomes, size = length(edx_test$y), replace = TRUE)
fun_RMSE(y_hat)
```

```
## [1] 1.94104
```

This results in an average error of about two stars! This is really bad, we could do way better with estimating the outcome distribution from the `train` set:

```r
suppressWarnings(set.seed(1, sample.kind = "Rounding"))
possible_outcomes <- c((1:10)/2)
probs <- sapply(possible_outcomes, function(x){mean(x == edx_train$y)})
y_hat <- sample(possible_outcomes, size = length(edx_test$y), replace = TRUE, prob = probs)
fun_RMSE(y_hat)
```

```
## [1] 1.499034
```

We already manage to decrease our RMSE by about a half star! Not bad, but we know that the expected value is equal to the arithmetic mean. So why bother with guessing when we can always predict the same rating, the mean (which we know minimizes the RSME)?

```r
mu <- mean(edx_train$y)
y_hat <- rep(mu, times = length(edx_test$y))
fun_RMSE(y_hat)
```

```
## [1] 1.060054
```

We - again - managed to decrease the RMSE by about a half star. An educated guess is way better than a random guess! However, we always predict the same and regard everything else as an error. In mathematical notation we would write this model

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

where μ is the "true" rating for all users and $\varepsilon_{i,u}$ independent errors, or "noise". However, we know that not all movies are the same. Would **you** rate *The Matrix* the same as it's succesors?

Movie and User Effects

This movie *effect* or *bias* (b) can be added to the model above. Our equation now states

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

We will estimate the movie effect b_i from the data as average deviation from the mean for each movie:

```
# estimate movie effects
movie_bs <- cbind(edx_train$x, edx_train$y) %>% rename("true_y" = "edx_train$y") %>%
  group_by(movieId) %>%
  summarize(b_movie = mean(true_y - mu))

# apply movie effects and compute RSME
predictions <- data.frame(mu = mu, b_movie = edx_test$x %>%
                            left_join(movie_bs, by = "movieId") %>%
                            pull(b_movie))
y_hat <- predictions %>% rowSums()
fun_RMSE(y_hat)
```

```
## [1] 0.9429615
```

The same holds true for users. Some tend to rate every movie a perfect 5 stars whilst others are critical and discerning. If we augment the equation to account for user effects b_u

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

and estimate it as the average deviation from the mean for each user after accounting for movie effects

```
# estimate user effects
user_bs <- cbind(edx_train$x, edx_train$y) %>%
  rename("true_y" = "edx_train$y") %>%
  left_join(movie_bs, by = "movieId") %>%
  group_by(userId) %>%
  summarize(b_user = mean(true_y - (mu + b_movie)))

# apply user effects and compute RSME
predictions$b_user <- edx_test$x %>%
  left_join(user_bs, by = "userId") %>%
  pull(b_user)
y_hat <- predictions %>% rowSums()
fun_RMSE(y_hat)
```

```
## [1] 0.8646844
```

we get a RMSE of 0.865. This is very close to the project goal and actually better than in the example from the subset included in the `dslabs` package. However, this is to be expected since we have a much larger data basis and our predictions therefore become more accurate by reducing the random "noise" from small samples.

Regularized Movie and User Effects

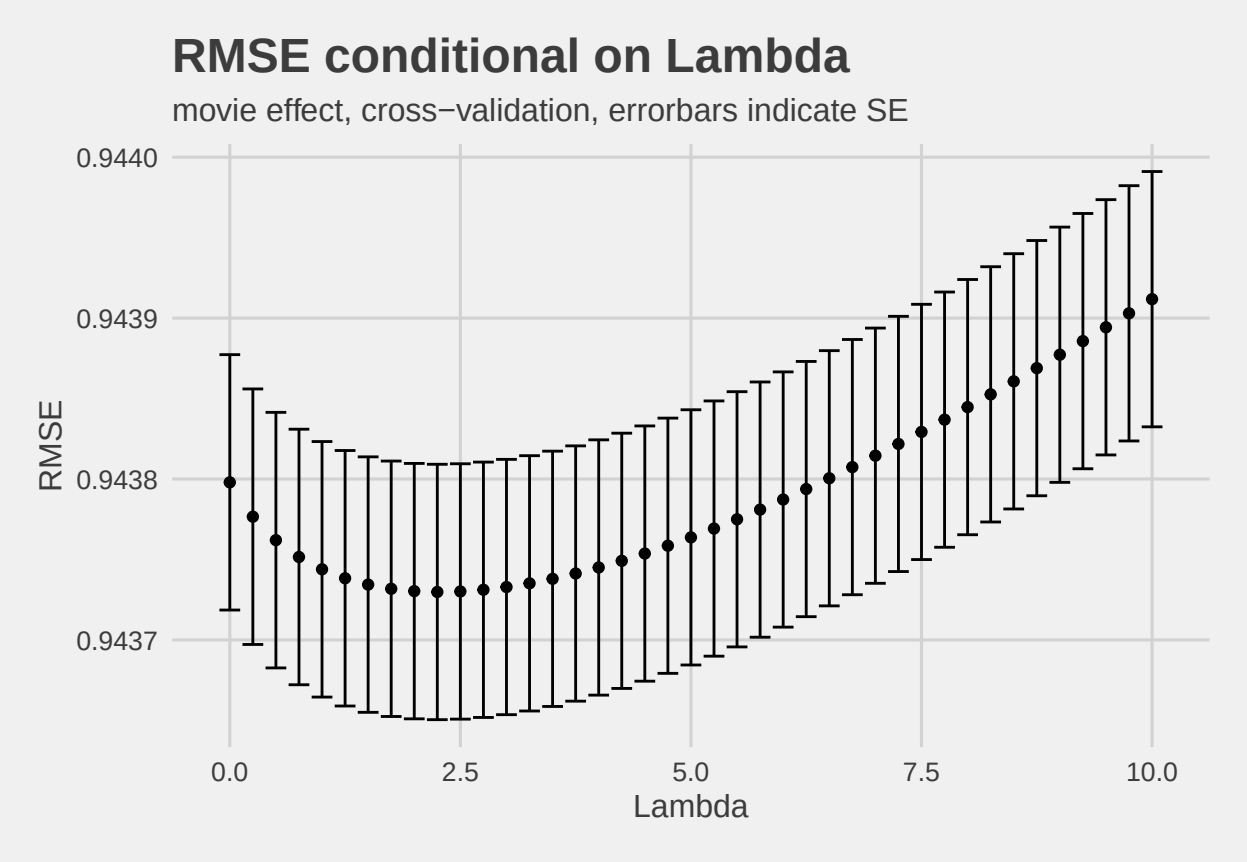
Remember when we said that one large error increases our RMSE way more than many small errors? Therefore, we should be conservative and only trust our estimates when they are stable. We can "punish" uncertain predictions (predictions that stem from few samples) by shrinking them as a function of the observations that drive them. Consider again our estimation for the movie effects

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

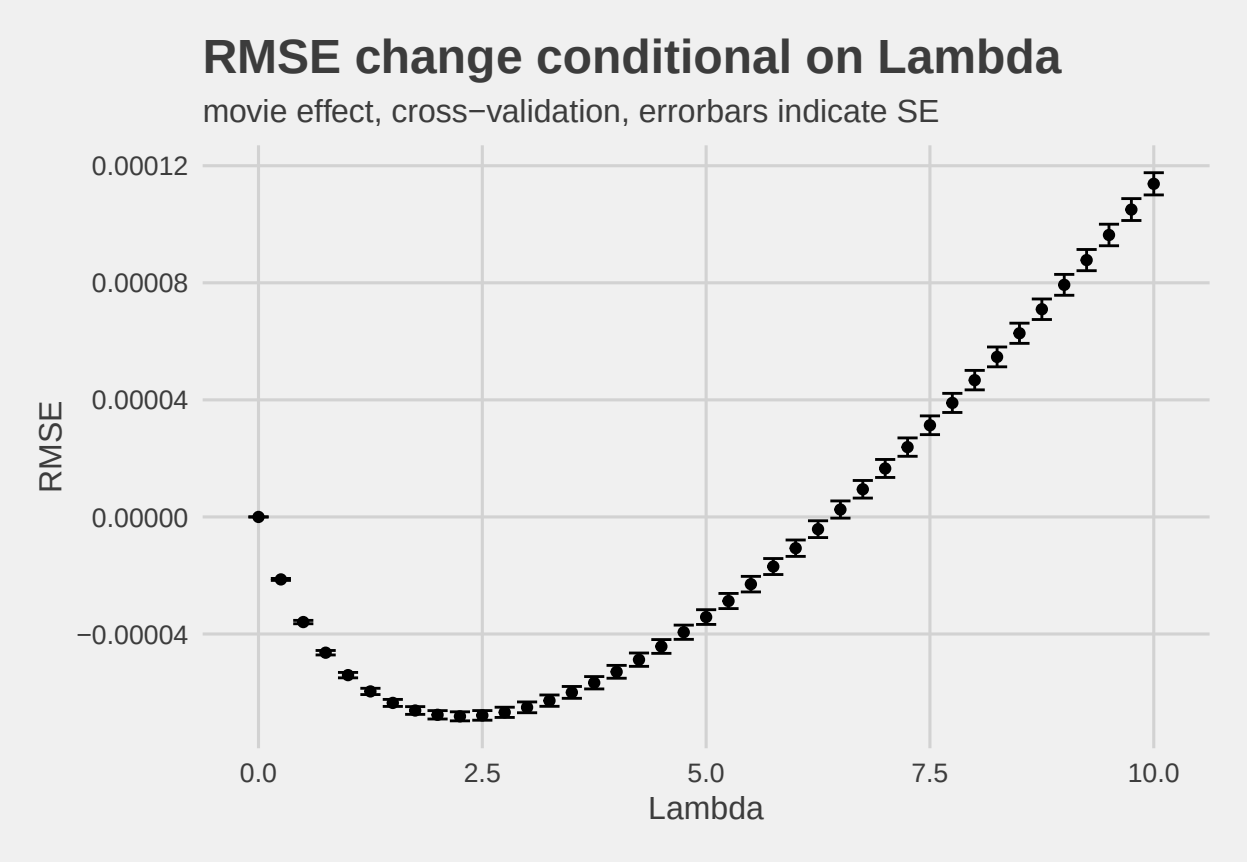
which can be "shrunken" by introducing a penalty λ

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

As we don't know what lambda ($\lambda = 0$ means penalty-free estimate) yields the best RMSE, we can use cross-validation to select the optimal lambda. This, in addition, allows us to estimate the stability of the computed RMSE.



Note, however, that the variation in the RMSE is not only due to the lambda but in a big part due to the cross-validation method. We could also inspect the variation of the lambda-effect only.



However, we need one lambda value, not a range of possible best lambda values, so this is of less interest for us than the range of RMSEs we might have with the validation set. Thus, we select 2.25 as optimal lambda value for the movie effect.

```
# estimate movie effects with regularization
movie_bs <- cbind(edx_train$x, edx_train$y) %>% rename("true_y" = "edx_train$y") %>%
  group_by(movieId) %>%
  summarize(
    s = sum(true_y - mu),
    n_i = n()) %>%
  mutate(
      b_movie = s/(n_i+movie_lambda)
      ) %>%
  select(movieId, b_movie)

# apply movie effects and compute RSME
predictions <- data.frame(mu = mu, b_movie = edx_test$x %>%
                            left_join(movie_bs, by = "movieId") %>%
                            pull(b_movie))
y_hat <- predictions %>% rowSums()
fun_RMSE(y_hat)

## [1] 0.9429389
```

The same priciple can be applied to the user effect which produces the best RMSE for $\lambda_u = 2.25$. When computing the RMSE for our leaderboard set with the new lambda,

```
# estimate user effects with regularization
user_bs <- cbind(edx_train$x, edx_train$y) %>% rename("true_y" = "edx_train$y") %>%
  left_join(movie_bs, by = "movieId") %>% mutate(
    residual = true_y - (mu + b_movie)) %>%
  group_by(userId) %>%
  summarize(
    s = sum(residual),
    n_i = n()) %>%
  mutate(
    b_user = s/(n_i + user_lambda)
  ) %>%
  select(userId, b_user)

# apply user effects and compute RSME
predictions$b_user <- edx_test$x %>%
  left_join(user_bs, by = "userId") %>%
  pull(b_user)
y_hat <- predictions %>% rowSums()
fun_RMSE(y_hat)
```

```
## [1] 0.8641563
```

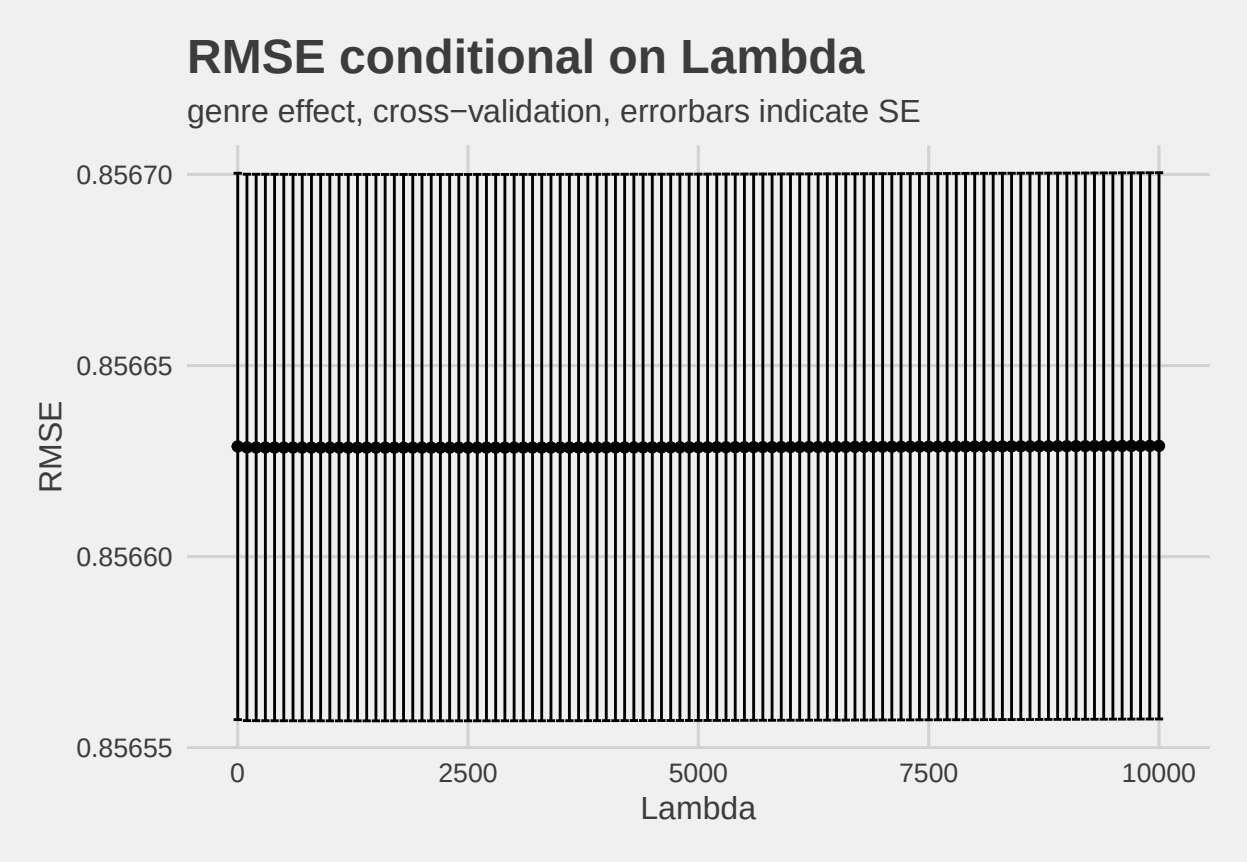
we decrease the RMSE to 0.864. This is only 0.001 better than the unregularized version. How come? As with the good performance of the unregularized effects this is due to the the large dataset. With this many ratings, there are only very few movies/ users that produce unstable estimates. However, it still produces *some* improvement, so we will keep the regularized b's instead of the unregularized.

Grouping Users Movies

The next approach that comes to one's mind is adding a term that accounts for similarity between movies. Sci-Fi movies are rated differently than Romantic comedies, aren't they? May we be able to use the information on the movies genre? A hands on approach to this is to simply define a movies i's only genre q as the first genre that is listed.

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

We will incorporate regularization directly and compute the leaderbord RMSE only with the optimal lambda.



We see that the effect of constraining the size of biases is only very limited here. Again, this is due to the (now even larger) samples that are used to compute the effects. However, again, we take whatever improvement we can get und thus end up with a lambda of 1800 for the genre effect. When applying this to the leaderboard set

```
# estimate genre effects with regularization
genre_bs <- cbind(edx_train$x, edx_train$y) %>% rename("true_y" = "edx_train$y") %>%
  left_join(movie_bs, by = "movieId") %>%
  left_join(user_bs, by = "userId") %>%
  left_join(one_per_genres, by = "genres") %>% mutate(
    residual = true_y - (mu + b_movie + b_user)) %>%
  group_by(main_genre) %>%
  summarize(
    s = sum(residual),
    n_i = n()) %>%
  mutate(
    b_genre = s/(n_i + genre_lambda)
  ) %>%
  select(main_genre, b_genre)

# apply genre effects and compute RSME
predictions$b_genre <- edx_test$x %>%
  left_join(one_per_genres, by = "genres") %>%
```

```
  left_join(genre_bs, by = "main_genre") %>%
  pull(b_genre)
y_hat <- predictions %>% rowSums()
fun_RMSE(y_hat)
```

```
## [1] 0.8640489
```

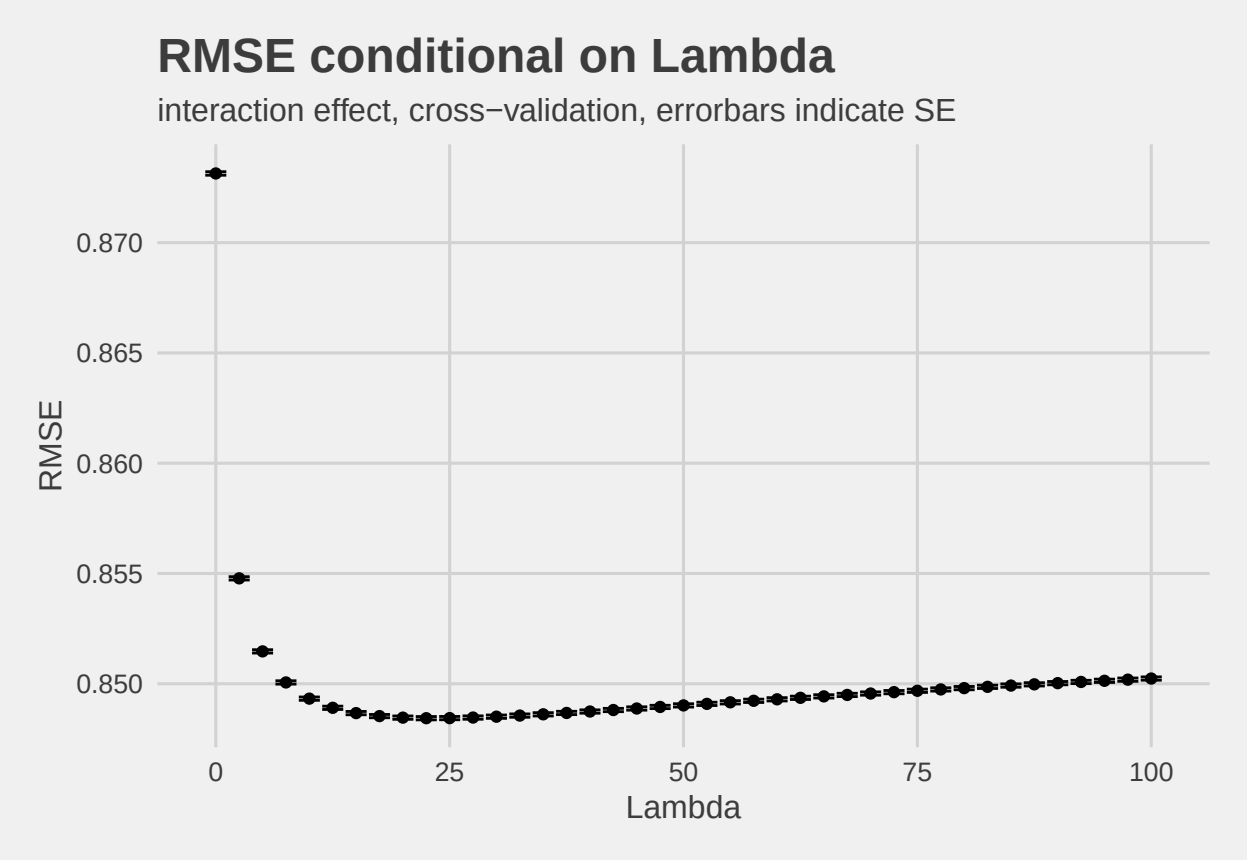
we have a RMSE of 0.864. This is surprising, isn't it?

User-Genre-Interaction: Actually recommending

Actually, not at all. We adjusted for the movie effect when computing the genre effect. Thus, the genre gives close to no *additional* information above the movie ratings. The effect we had in mind is actually not a genre effect, but agenre-user *interaction* effect. Some users like particular genres better whilst disliking others. We try to predict the entertainment value of a movie for a person, given the rating of similar movies. In our formula we have to add this interaction as

$$Y_{u,i} = \mu + b_i + b_u + p_u q_i + \varepsilon_{u,i}$$

with q_i the hands-on variable we alrady introduced before and p_u a persons particular liking of the genres. However, note that this is the first time we actually recommend different movies to different users. Before, we always recommended the same movies to everyone, but predicted different ratings. Does the RMSE reflect this extension of validity? Again, we compute the RMSE with a cross-validated lambda.



This plot looks promising and shows us that the interaction without regularization would not have improved but actually worsened our RMSE. When computing the RMSE on the loaderboard set with the optimal lambda of 22.5,

```
# estimate genre-user interaction with regularization
IA_bs <- cbind(edx_train$x, edx_train$y) %>% rename("true_y" = "edx_train$y") %>%
  left_join(movie_bs, by = "movieId") %>%
  left_join(user_bs, by = "userId") %>%
  left_join(one_per_genres, by = "genres") %>%
  left_join(genre_bs, by = "main_genre") %>% mutate(
    residual = true_y - (mu + b_movie + b_user + b_genre)) %>%
  group_by(userId, main_genre) %>%
  summarize(
    s = sum(residual),
    n_i = n()) %>%
  mutate(
    b_IA = s/(n_i + IA_lambda)
  ) %>%
  select(userId, main_genre, b_IA)

# apply genre-user interaction and compute RSME
predictions$b_IA <- edx_test$x %>%
  left_join(one_per_genres, by = "genres") %>%
  left_join(IA_bs, by = c("main_genre", "userId")) %>%
  mutate(b_IA = ifelse(is.na(b_IA), 0, b_IA)) %>%
  pull(b_IA)
y_hat <- predictions %>% rowSums()
fun_RMSE(y_hat)
```

```
## [1] 0.8541135
```

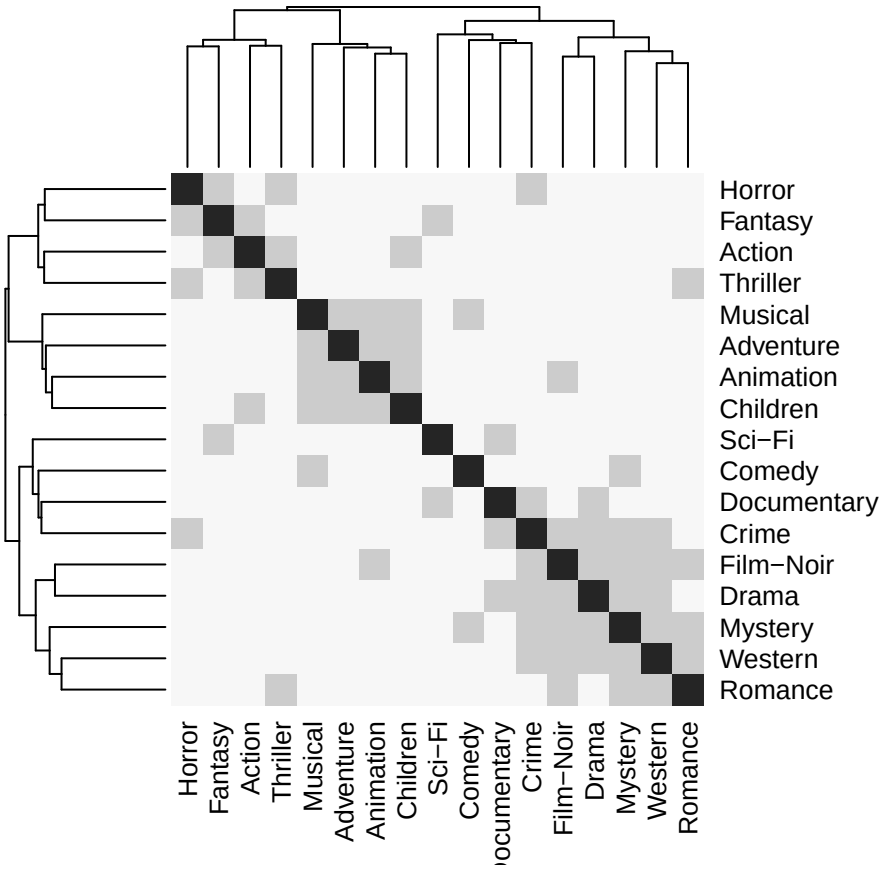
we see that the RMSE is 0.854 which is below the project goal. However, can we do even better?

Matrix Factorization Motivation

The question remains if the hands-on approach to grouping the genres was ideal. If it was, the b's should be independent from each other. Otherwise, we could use this similarity between genres to further improve our prediction. For demonstration purposes, we will look at the residuals from the leaderbard-set without the interaction biases. Also, because this is only for demonstration purposes and we don't want to crash R, we will only use genres and users with many ratings.

```
# create a dataset that only contains userId, main genre and residuals
genres_small <- edx_test$x %>% left_join(one_per_genres, by = "genres") %>%
  select(userId, main_genre, genres)
genres_small$residuals <- edx_test$y - (predictions %>% select(-b_IA) %>% rowSums())

# Keep only genres and users that have many ratings
genres_small <- genres_small %>%
  group_by(userId, main_genre) %>%
```

```
  summarize(residuals = mean(residuals)) %>%
  ungroup() %>%
  mutate(main_genre = as.factor(main_genre)) %>%
  group_by(main_genre) %>%
  filter(n() >= 500)%>%
  group_by(userId) %>%
  filter(n() >= 10)

# convert it to a matrix where every row represents a user
# and every column a genre values are the residuals
y <- genres_small %>% select(main_genre, residuals, userId) %>%
  pivot_wider(names_from = main_genre, values_from = residuals) %>% as.matrix()
rownames(y)<- y[,1]
y <- y[,-1]

# Plot a heatmap of the correlation pattern
cor(y, use = "pairwise.complete.obs") %>%
  heatmap(col = RColorBrewer::brewer.pal(5, "Greys"), symm = TRUE)
```

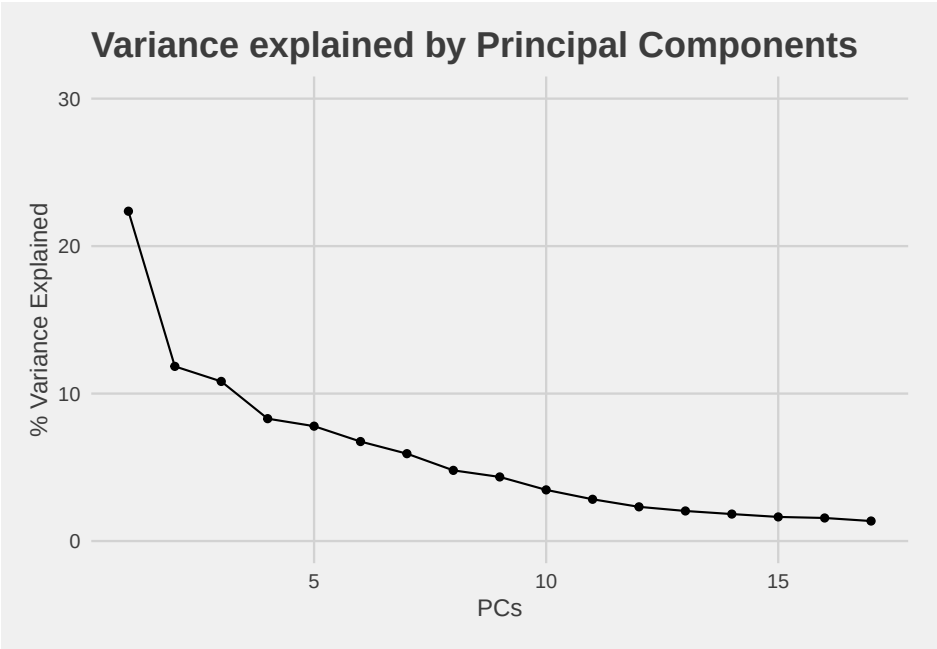


We see that there is some substantial correlation between the genres that we could possibly use. Genres are not dichotomous (Fantasy: `TRUE` or `FALSE`) but rather a continuum. Some movies incorporate aspects of both genres. However, our hands-on approach does ony allow for each movie to be in one genre. Fantasy and Sci-Fi, for example, are highly linked. Also you can have movies that are highly specific for their genre

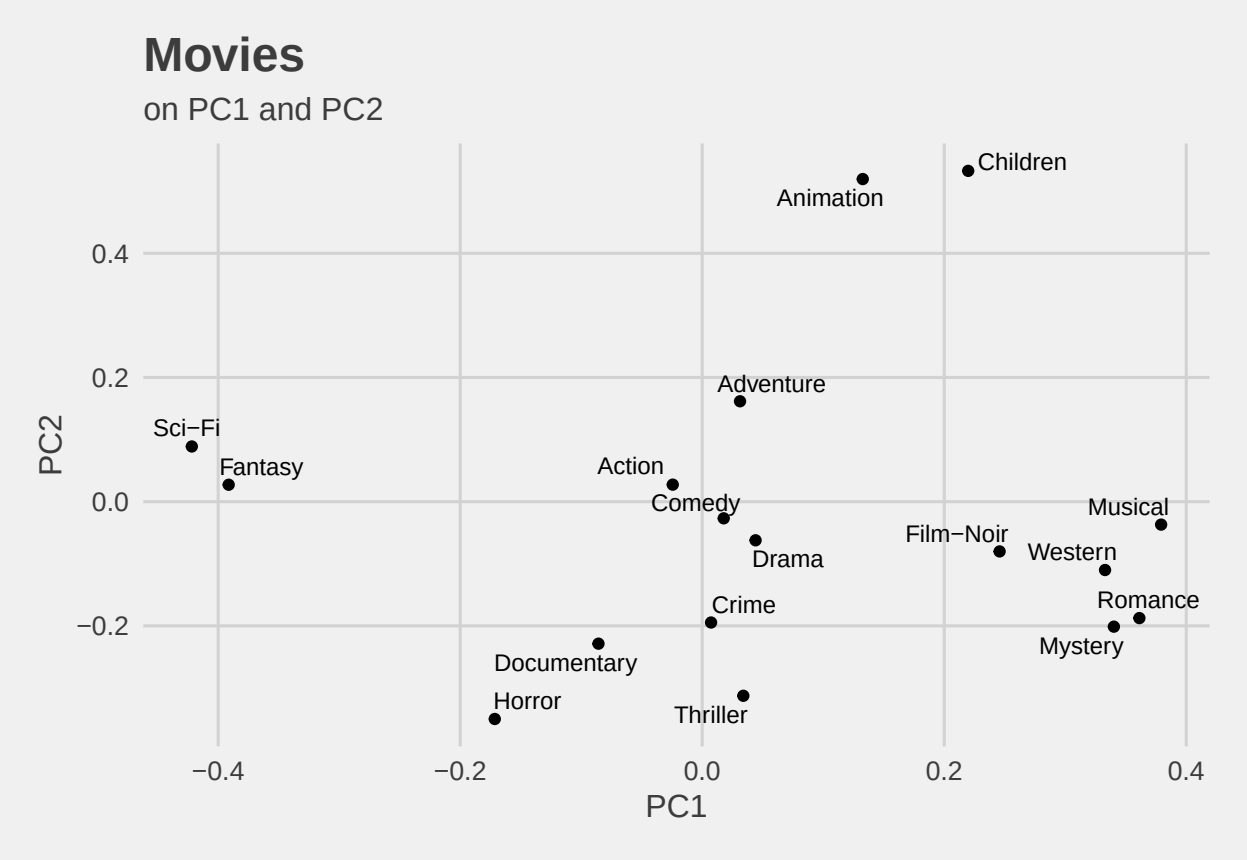
whilst others have gone out of a particular subculture into mainstream culture. We could also try to quantify the similarity of different movies, users or genres by the data itself. Principal Component Analysis comes to mind, but we face a lot of missing data even after aggregating to the main genre level:

One could use 0 to replace the missing values. However, this decreases the variance of genres that have only very few ratings. Those are the genres that we are especially interested in! For demonstration purposes we will therefore use multiple imputation to fill the missing data and compute a principal component analysis based on this generated dataset:

```
suppressWarnings(set.seed(1, sample.kind = "Rounding"))

# impute a data set
pca_md <- missMDA::imputePCA(y, ncp = 3)
y_md <- pca_md$completeObs

# compute a pca on this new data set
pca <- prcomp(y_md)

# plot the vaiability explained
ggplot(aes(1:length(pca$sdev), (pca$sdev^2 / sum(pca$sdev^2))*100), data = NULL) +
  geom_point() + geom_line() +
  scale_y_continuous(name = "% Variance Explained", limits = c(0,30)) +
  xlab("PCs") + ggtitle("Variance explained by Principal Components")
```



We see that we can wrangle the observed correlation patterns into a structure.



The first principal component seems to capture what I would call "Nerd-Favourites vs. Womans-Favourites" (Sci-Fi and Fantasy vs. Romance and Musical) whilst the second principal component captures the "Kids-Friendlieness" (Children and Animation vs. Horror and Thriller) We can see the basics of matrix factorization here. We try to explain a matrix of ratings by finding patterns and reducing the matrix to a matrix of smaller dimensions. However, we haven't used the `lm()` call throughout this course because it would crash R and instead found workarounds for our linear model with doing the *Type-I sum of squares decomposition* by hand. Doing a matrix decomposition by hand would also crash R. However, finding a workaround is beyond the scope of this report. Luckily, R packages exist for (almost) everything and so we will use the `recosystem` package to use matrix factorization on our data set.

Matrix Factorization

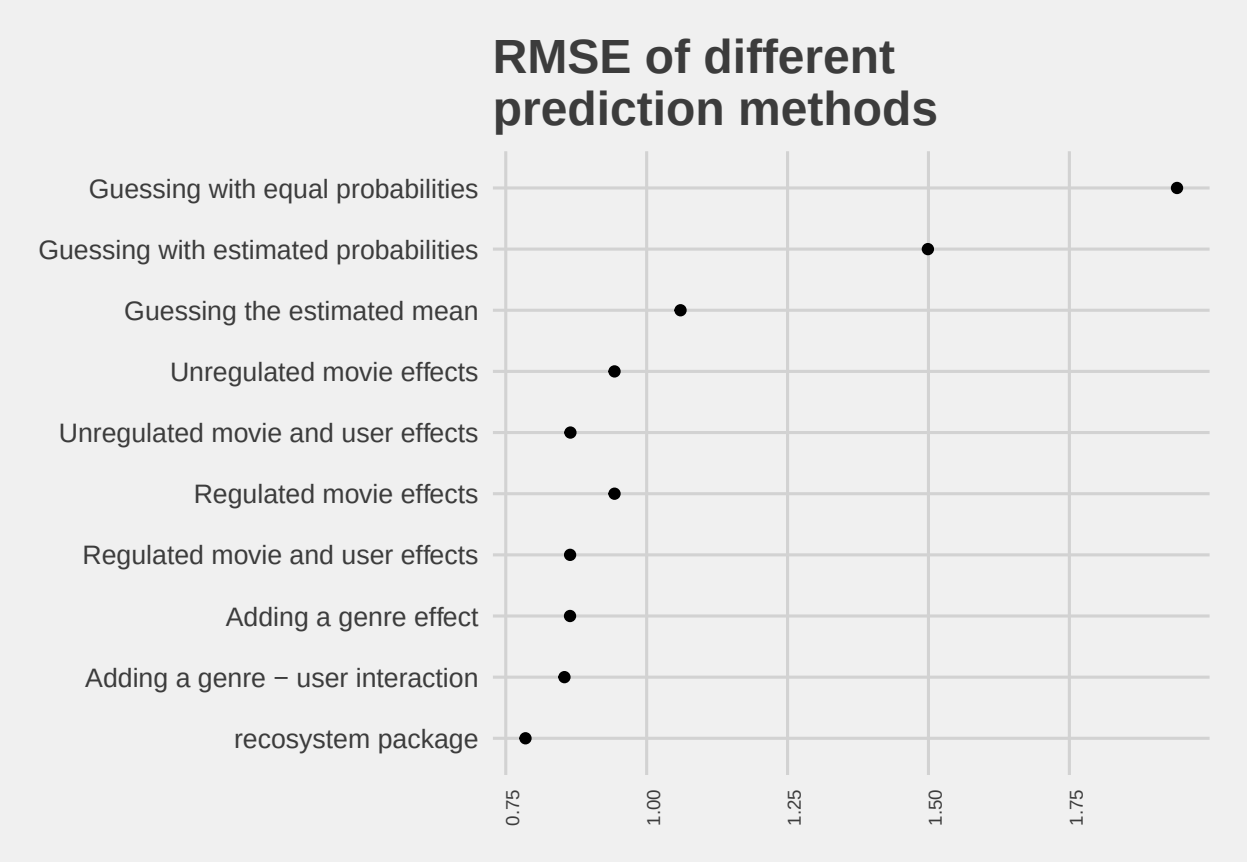
The `recosystem` package is pretty straight forward. It's purpose is to provide tools to build a recommendation system with. The operations are directly implemented in C++ and allow to write objects on the hard drive directly, thus lowering the system requirements.

```
# initialize reco
library(recosystem)
r = Reco()

# create a dataset with only rating, userId and movieId
reco_train <- edx_train$x %>% cbind(edx_train$y) %>% rename("y" = "edx_train$y") %>%
  left_join(movie_bs, by = "movieId") %>%
  left_join(user_bs, by = "userId") %>%
  left_join(one_per_genres, by = "genres") %>%
  left_join(genre_bs, by = "main_genre") %>%
  mutate(residual = y - (mu + b_movie + b_user + b_genre)) %>%
  select(userId, movieId, y)

# reco wants a special data format
reco_train <-  with(reco_train,
                    data_memory(user_index = userId, item_index = movieId,
                                rating = y, index1 = TRUE))

# try some tuning parameters, 3-fold cross validation is enough
opts = r$tune(reco_train,
              opts = list(dim = c(5, 10, 20, 50),
                          lrate = c(0.1, 0.2),
                          costp_l1 = c(0), costq_l1 = c(0),
                          costp_l2 = c(0, 0.01, 0.1, 0.3),
                          costq_l2 = c(0, 0.01, 0.1, 0.3),
                          nthread = 4, niter = 20, nfold = 3))
best_options <- opts$min

# train the model with the best parameters
suppressWarnings(set.seed(1, sample.kind = "Rounding"))
r$train(reco_train, opts = c(best_options, niter = 50, verbose = FALSE))

# create evaluation dataset
reco_test <- edx_test$x %>% select(userId, movieId)
```

```
reco_test <-  with(reco_test,
                   data_memory(user_index = userId, item_index = movieId, index1 = TRUE))

# create predictions and test them
y_hat <- r$predict(reco_test, out_memory())
fun_RMSE(y_hat)
```

```
## [1] 0.7849785
```

We see that that the `Reco()` model does substantially better than our additive model. An overview of the progress we made during this model building process is provided by the following plot:



Results: Applying the Best Model to the Validation Set

We are way below our RMSE goal so it is time to re-train the best model with the whole `edx` dataset and compute the RMSE on the `validation` set!

```
# read in validation set (I saved both sets locally)
edx <- fread("movielens_edx.csv") %>% select(userId, movieId, rating)
validation_predictors <- fread("movielens_validation.csv") %>% select(userId, movieId)
```

```
# initialize Reco and wrangle data sets
r = Reco()
edx <-  with(edx, data_memory(user_index = userId, item_index = movieId,
                              rating = rating, index1 = TRUE))
validation_predictors <-  with(validation_predictors,
                               data_memory(user_index = userId,
                                           item_index = movieId, index1 = TRUE))

# train the reco system
# consider lower number of iterations to speed things up
suppressWarnings(set.seed(1, sample.kind = "Rounding"))
r$train(edx, opts = c(best_options, niter = 1000, verbose = FALSE))

# predict ratings
y_hat <- r$predict(validation_predictors, out_memory())
```

In addition we know that ratings below 0.5 and above 5.0 can not happen, so we are rescaling them to 0.5 and 5.0, respectively, before computing the final RMSE:

```
# clip predictions
y_hat <- ifelse(y_hat > 5, 5, ifelse(y_hat < 0.5, 0.5, y_hat))

# show true values for the first time
true_y <- fread("movielens_validation.csv") %>% pull(rating)

# compute the final RMSE
caret::RMSE(y_hat, true_y)
```

```
## [1] 0.786901
```

Our final RMSE is 0.787. This surpasses the target RMSE by roughly 10%.

Conclusion

We built a recommendation system from scratch. Our first approach was simply guessing the mean for all ratings. This cut the average error by about half compaired to randomly guessing! We then started adding up different effects until we encountered the insight that we were *predicting*, but not *recommending*. We then did a very naive approach to recommending by adding an interaction term to our linear model. However, through analysis of the residuals we came to the conclusion that insights about *similarity* should be driven by the data, not by an (kind of) arbitrary genre-label. A powerful tool for sophisticated parallel matrix-factorization is the `recosystem` package[8]. With the model provided by reco we managed to achieve an even lower RMSE than the RMSE that Netflix payed 1 million dollars for!

However, the data used to create and evaluate this model is different from the data in the Netflix challenge. Sadly, due to privacy concerns[9], the data is not publicly available anymore. Therefore, compairing the RMSE is hardly feasible. Another limitation is the data we used to create the ratings: We only used `userId` and `movieId` in the final model, neglecting all other data at hand. Information about the time of the rating,

[8] https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html
[9] Narayanan, A., & Shmatikov, V. (2006). How to break anonymity of the netflix prize dataset.

release year or genres might improve our RMSE even further. Modern recommendation systems use many more predictors than our model. But such computations with a dataset this size is simply not feasible with a commodity laptop.