HarvardX - PH125.9x Data Science: Capstone - Movie Lens

Emmanuel Rialland - https://github.com/Emmanuel_R8

October 04, 2019

Contents

Chapter 1

Introduction

This report is a story of failing human intuitions and data science success. In brief, it demonstrates that statistical learning brings insights otherwise unavailable, and eventually achieves and RMSE of *0.

This project is the first of two final projects of the *HarvardX - PH125.9x Data Science* course.

Its purpose is the development of a recommender system for movie ratings using the Movie Lens dataset.[1] Recommender systems are a class of statistical learning systems that analyse individual past choices and/or preferences to propose relevant information to make future choices. Typical systems would be propose additional items to purchase knowing past shopping activity, searches (e.g. Amazon), choice of books (e.g. GoodRead) or movies (Netflix).

Broadly, recommender systems fall into two categories:

- *collaborative filtering* (user-based) which attempts to pool similar users together and guide a user's recommendation given the pool's preference.

- *content-based filtering* which attempts to pool similar contents (e.g. shopping carts, movie ratings) together and guide a user's recommendation within a simila pools of content.

In practice, those two approaches are mixed together. A general overview is available on Wikipedia[2] and in the course materials.[3]

Being given a training and a validation dataset, we will attempt to minimise the Root Mean Sqared Error (RMSE) of predicted ratings for pairs of user/movie below `0.8649`.

We note that Netflix organised a competition spanning over several years to improve a recommender system which shares many similarities with this project (Bennett, Lanning, and others 2007). Papers published by teams who participated in that competition have guided some of this report. (Bennett, Lanning, and others 2007) (Bell, Koren, and Volinsky 2007) (Bell, Koren, and Volinsky 2008) (Koren 2009) (Töscher, Jahrer, and Bell 2009) (Piotte and Chabbert 2009) (Gower 2014)

This report is organised as follows. In Section 2, we describe the dataset and add a number of possibly relevant predictors. Section 3 provides a number of visualistions. Section 4 proposes three

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

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

[3] https://rafalab.github.io/dsbook/large-datasets.html#recommendation-systems

models that will show to be poor performers. Section 5 is dedicated to a low-rank matrix factorisation estimated with a stochastic gradient descent.

Chapter 2

Data Summary and Processing

Unless specified, this section only uses a portion (20%) of the dataset for performance reasons.

2.1 Description of the dataset

The data provided is a list of ratings made by anonymised users of a number of movies. The entire training dataset is a table of 9000055 rows and 6 variables. Note that the dataset is extermely sparse: if each user had rated each movie, the dataset should contain 54000330 ratings, i.e. 85 times more.

Each row represents a single rating made by a given user regarding a given movie.

The complete dataset includes 10677 unique movies, rated by 69878 unique users. No user rated the same movie twice.[1] Importantly, the dataset is fully and properly populated: no missing or abnormal value was found. However, a few movies were rated before the movie came out: the date of such ratings falls in the year before the one in brackets in the title. In such case, the date of first screening is brought to the date of the first rating.

The reduced data set includes 10225 unique movies, rated by 69750 unique users. That is, very few users or movies are missed by restricting the dataset.

The dataset variables are:

[1] See source code.

Name	Format	Description
‘userId‘	Numerical	Unique numerical identifier for anonymity purposes
‘movieId‘	Numerical	Unique numerical identifier
‘rating‘	Numerical	Possible ratings are 0, 0.5, 1, ..., 4.5 and 5.0. No movie is rated 0.
‘timestamp‘	Numerical	Unix epoch of the date/time of the rating (i.e. number of seconds since 1-Jan-1970.
‘title‘	Character string	String of characters of the movie title _AND_, in brackets, of the year the movie came out.
‘genres‘	Character string	String of characters listing the genres to which the movie belongs. There are 20 possible categories. Each movie can belong to several categories (e.g. Action and Comedy). If there are several categories, there are listed separated by a vertical bar.

2.2 Description of the variables.

2.2.1 Intuitive description of the pre-processing requirements

The dataset needs to be preprocessed to add more practical information. Some steps are necessary to make available information usable: this is the case for splitting the genres and extracting the year a movie came out. Other changes are driven by the following considerations.

All users are resource-constrained. Watching a movie requires time and money, both of which are in limited supply. The act of taking the time to watch a movie, by itself, is an act of choice. The choice of which movie to watch results from a selection process that already biases a spectator towards movies he/she feels likely to enjoy. In other words, at least on an intuitive level, the pairs user/movie are not random: users did not select a movie randomly before rating it.

It is common knowledge that:

- A movie screened for the first time will sometimes be heavily marketed: the decision to watch this movie might be driven by hype rather than a reasoned choice; the choice to watch it is not a rational choice and will lead to possible disappointments.
- In the medium term after first screening, movie availability could be relevant. Nowadays, internet gives access to a huge library of recent and not so recent movies. This was definitely not the case in the years at which ratings started to be collected (mid-nineties).
- The decision to watch a movie that came out decades ago is a very deliberate process of choice. There is a *survival effect* in the sense that time sieved out bad movies. We could expect old movies, e.g. *Citizen Kane*, to be rated higher on average than recent ones.
- In the short term, just a few weeks would make a difference on how a movie is perceived. But whether a movie is 50- or 55-year old would be of little impact. In other words, some sort of rescaling of time, logarithmic or other, need considering.

- If a movie is very good, more people will watch it and rate it. In other words, we should see some correlation between ratings and numbers of ratings. Again, some sort of rescaling of time, logarithmic or other, need considering.

Whether this additional information is actually useful will be analysed later in this report.

2.2.1.1 Changes related to the movies:

- Split the `genres` tags into separate logical variable, i.e. 1 variable per individual genre. Each individual tags is a `-1` or `1` numerical value, with `1` indicating that a movie belongs to that genre. The reasons for using numerical values are:
  - On a more intuitive level, movie are not all-or-nothing of a particular genre: a movie is not funny or not-funny; it could be a little bit funny or extremely funny. We could imagine a dataset where that movie would be a 20% or a 95% Comedy, or -50% anti-funny movie, possibly by extracting information from movies reviews.
  - We could also encode with 0,1 instead of -1,1. Modeling has shown to be more effective with the -1,1 encoding.
  - Key algorithms for recommender system involve dimension reduction which requires all variable to be numerical (no factors).
- Dimension reduction require variable scaling: for a given movie, all the ratings received by that movie are centered and scaled into a z-score. If a movie only received a single rating, the standard deviation is assumed to be 1 to avoid any missing value.
- The date a movie came out is extracted from the title of the movie. The date is always a year, which we convert into January, 1st of that year (to avoid any rating being dated before).

2.2.1.2 Changes related to the users:

- As for the movies, for a given user, ratings given by a particular user are centered and scaled using the mean and standard deviation of all the ratings given by that particular user.

2.2.1.3 Changes related to the dates:

- Timestamps cannot be readily understood. All dates (including the date a movie came out) are converted to number of `proper`Lubridate‘ date objects. Difference between dates are expressed in days.
- As we will see, ratings for older movies tend to be higher. Time lapsed until a movie is rated seems of interest (later analysis will show to which extent). The dataset is completed by there time lapses: looking at the date of a particular rating, how many days have passed since:
  - the movie came out;
  - the movie received its first rating;
  - the user gave its first rating.
- All dates are also in [logarithmic / square root scale].

2.2.2 Summary of the steps

Once the pre-processing is carried out, the dataset variables are:

```
##  [1] "userId"               "movieId"              "rating"
##  [4] "title"                "date_rating"          "rating_z"
##  [7] "movie_nRating"        "movie_nRating_log"    "movie_mean_rating"
## [10] "movie_sd_rating"      "movie_first_rating"   "movie_z"
## [13] "movie_year_out"       "movie_date_out"       "Action"
## [16] "Adventure"            "Animation"            "Children"
## [19] "Comedy"               "Crime"                "Documentary"
## [22] "Drama"                "Fantasy"              "FilmNoir"
## [25] "Horror"               "Musical"              "Mystery"
## [28] "Romance"              "SciFi"                "Thriller"
## [31] "War"                  "Western"              "user_nRating"
## [34] "user_nRating_log"     "user_mean_rating"     "user_sd_rating"
## [37] "user_first_rating"    "user_z"               "time_since_out"
## [40] "time_since_out_log"   "time_movie_first"     "time_movie_first_log"
## [43] "time_user_first"      "time_user_first_log"
```

Chapter 3

Visualisation

This review is focused on the training set, and excludes the validation data. We are working on the same extract of the full dataset as in the previous section.

The purpose of the review is to give a high level sense of what the presented data is and some indicative research avenues for modelling.

We first review individual variables. Then we reviews variables by pairs.

We have described the `Data Preparation` section the list of variables that were originally provided, as well as reformatted information.

3.1 Summary analysis of individual variables

3.1.1 Users

All users are identified by a single numerical ID to ensure anonymity.[1]

There are 69750 unique users in the training dataset. Most of them have rated few movies.

The following plot shows a log-log plot of number of ratings per user. Recall that the *Movie Lens* dataset only includes users with 20 or more ratings.[2] However, since we are plotting a reduced dataset (20%), we can see users with less than 20 ratings.

However, plotting the cumulative sum the number of ratings (as a a number between 0% and 100%) reveals that most of the ratings are provided by a minority of users.

We note the movielens data only includes users who have provided at least 20 ratings.

3.1.2 Ratings

3.1.2.1 Ratings are not continuous

All ratings are between 0 and 5, say, *stars* (higher meaning better), using only a whole or half number. A user cannot rate a movie 2.8 or 3.14159. The following code shows that all available ratings *apart* from 0 have been used.

[1] Note that in the case of the Netflix challenges, researchers succeeded in de-anonymising part of the dataset by cross-referencing with IMDB information. See (Narayanan and Shmatikov 2006).

[2] See the `README.html` file provided by GroupLens in the zip file.

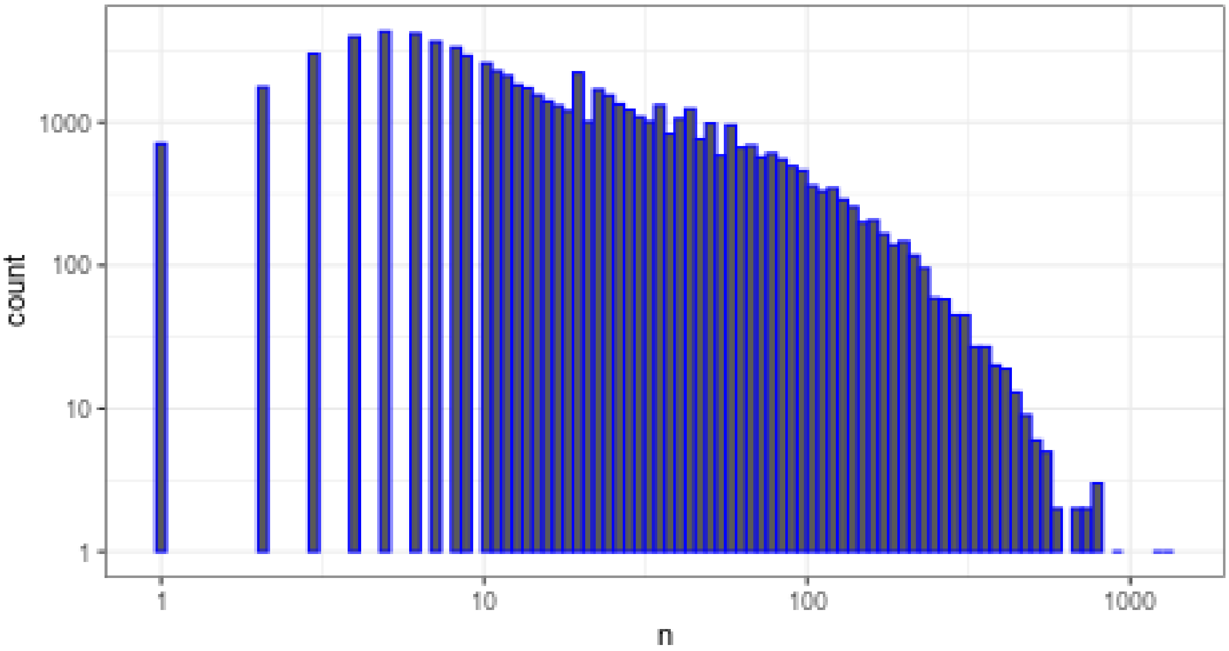


Figure 3.1: Number of ratings per users (log scale)

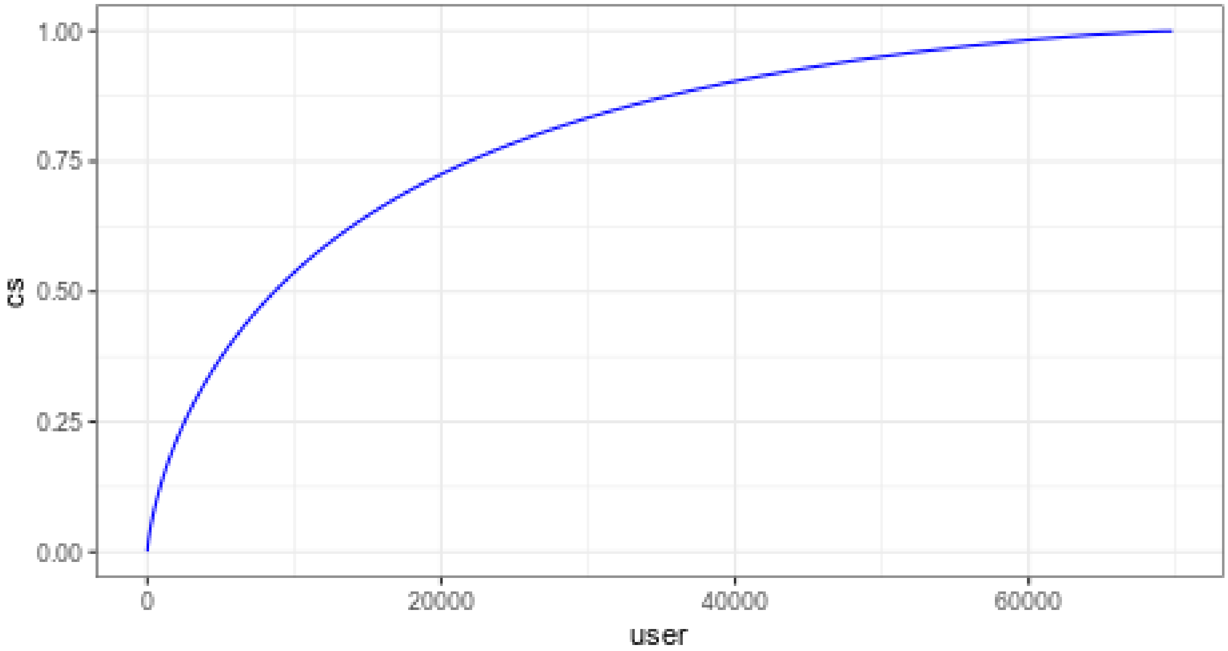


Figure 3.2: Cumulative proportion of ratings starting with most active users.

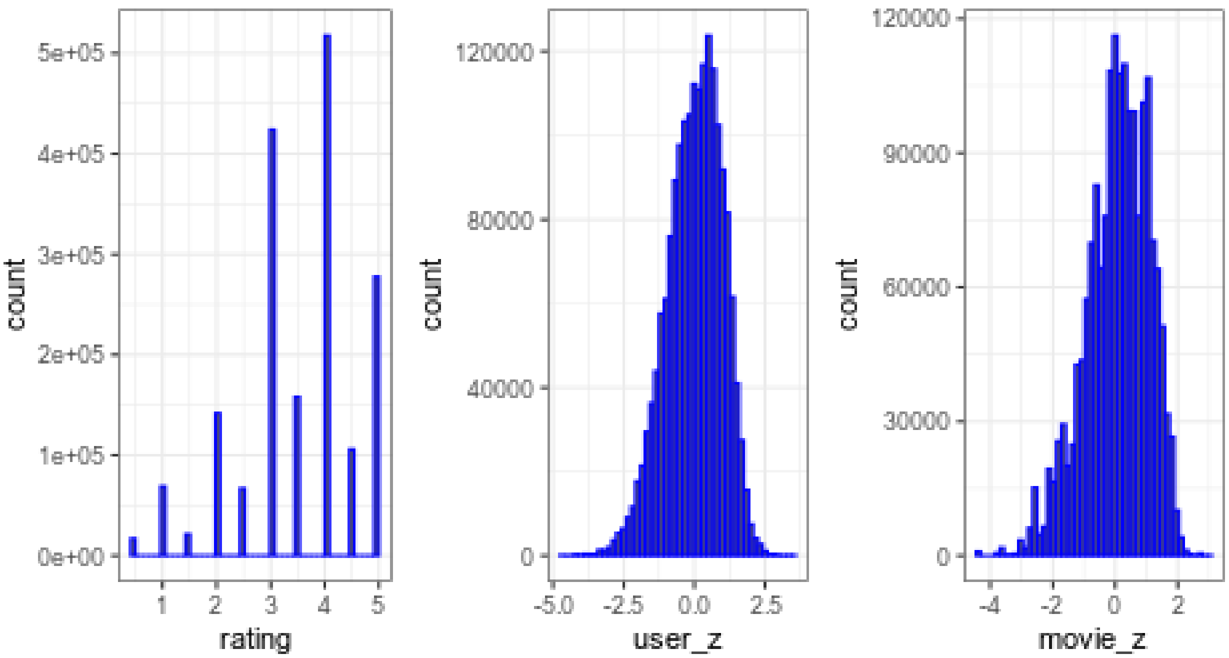


Figure 3.3: Histograms of ratings z-scores

rating	n
0.5	16942
1.0	69045
1.5	21447
2.0	142056
2.5	66953
3.0	424188
3.5	158307
4.0	517704
4.5	105809
5.0	277561

We also note that users prefer to use whole numbers instead of half numbers:

whole_or_half	n
0.0	1430554
0.5	369458

3.1.2.2 Whole ratings and z-scores

Plotting histograms of the ratings are fairly symmetrical with a marked left-skewness (3rd moment of the distribution).

3.2 Intuitive statements

We previously made a number of statements driven by intuition. Let us verify those.

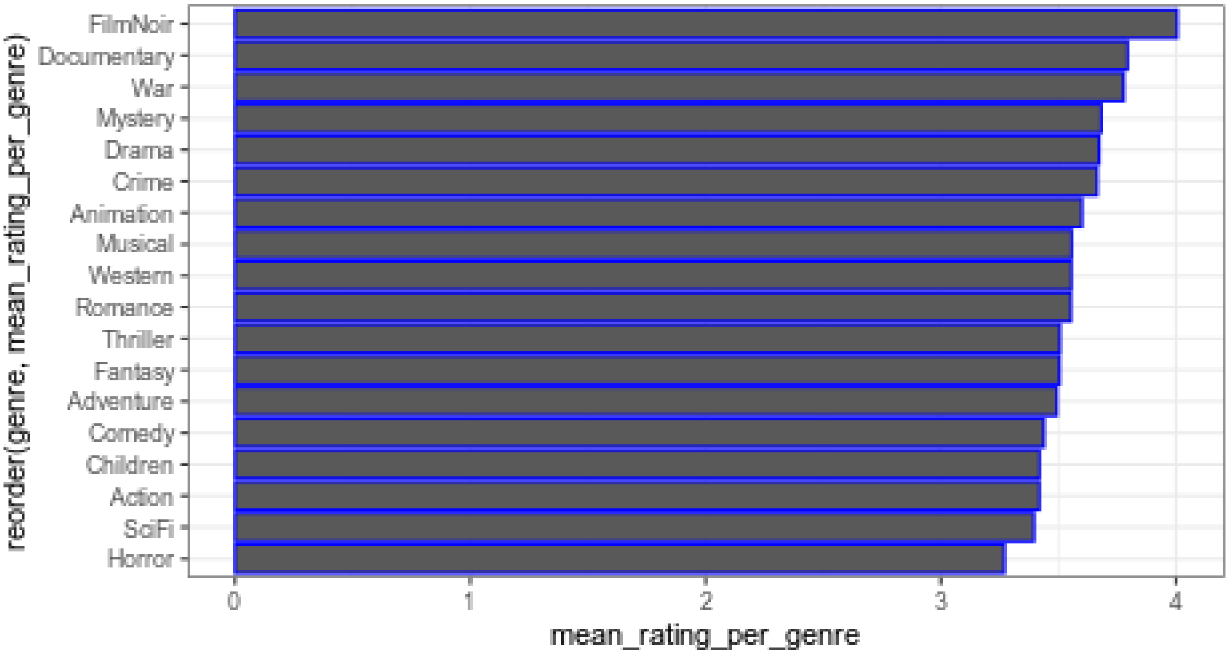


Figure 3.4: Average rating per genre

3.2.1 Statement 1

> *A movie screened for the first time will sometimes be heavily marketed: the decision to watch this movie might be driven by hype rather than a reasoned choice.*

A plot of ratings during the first 100 days after they come out seems to corroborate the statement: at the far left of the first plot, there is a wide range of ratings (see the width of the smoothing uncertainty band). As time passes by, ratings drops then stabilise.

The effect is independent from movie genre (when ignoring all movies that do not have ratings in the early days).

3.2.2 Statement 2

> *In the medium term after first screening, movie availability could be relevant. Nowadays, the Internet gives access to a huge library of recent and not so recent movies. This was definitely not the case in the years at which ratings started to be collected (mid-nineties).*

For the purpose of determining whether this statement holds in some way, we need to consider:

- What happened to the number of ratings over time since a movie came out: more people would see the movie when in movie theaters, whereas later the movies would have been harder to access.
- Whether these changes in rating numbers vary if a movie is released in the eighties, nineties, and so on.

The following plot should be read as follows:

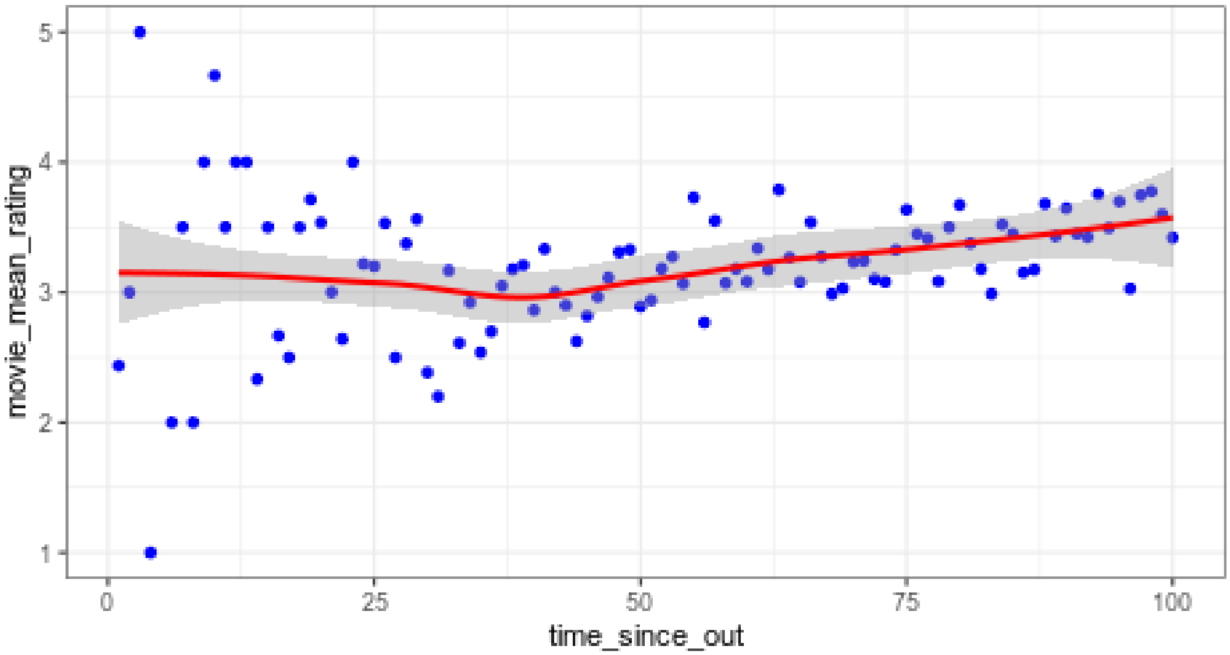


Figure 3.5: Ratings for the first 100 days

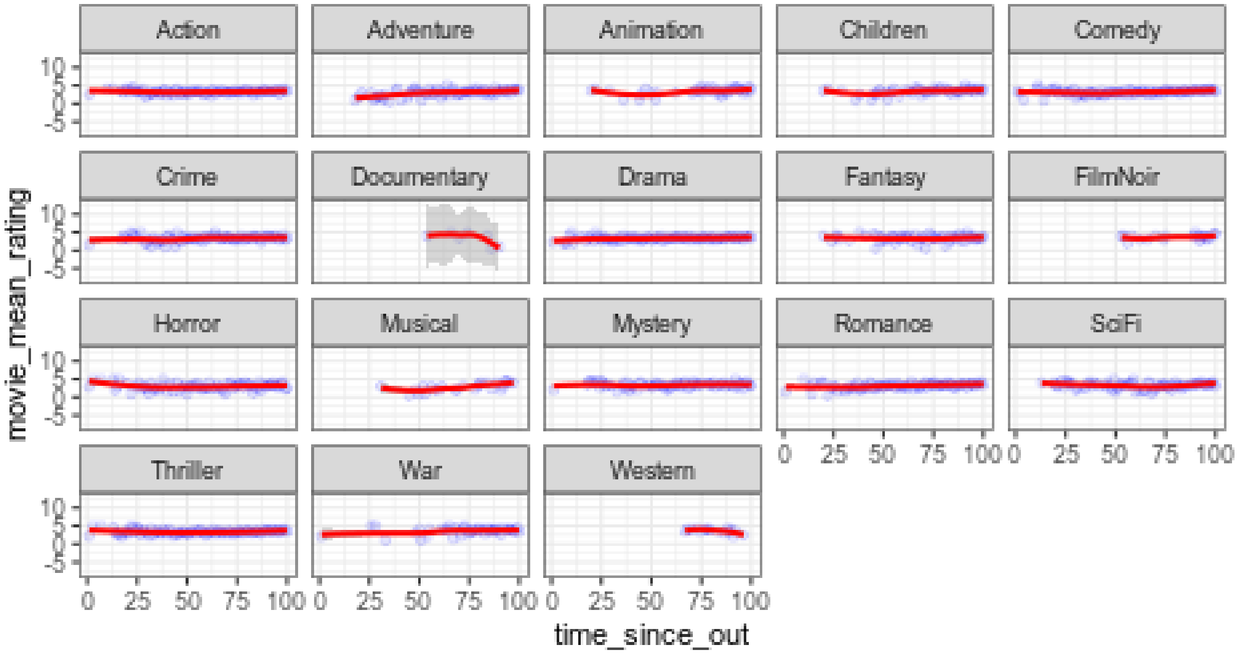


Figure 3.6: Ratings for the first 100 days by genre

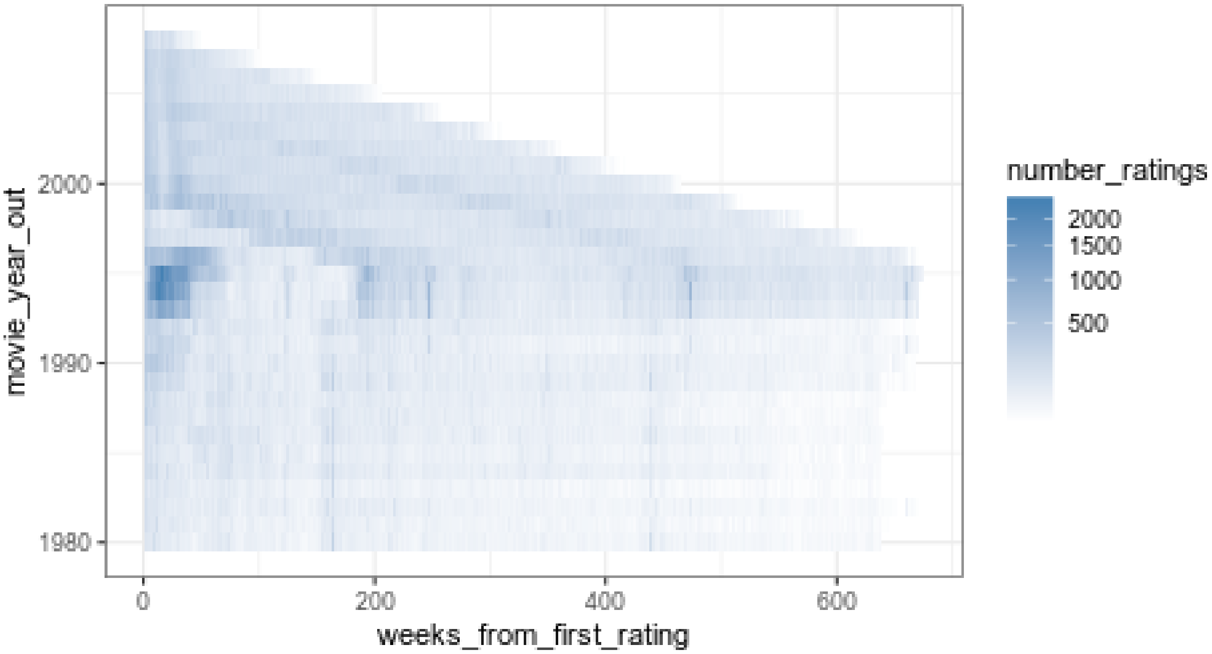


Figure 3.7: Number of ratings depending on time lapsed since premier and year of premiering

- choose year on the y-axis, and follow in a straight line from left to right;
- the colour shows the number of ratings: the darker, the more numerous;
- the first ratings only in 1988, therefore there is a longer and longer delay before the colours appear when going for later dates to older dates.

We can distinguish 4 different zones depending on the first screening date:

- Very early years before 1992: very few ratings (very pale colour) possibly since fewer people decide to watch older movies.

- Early years 1993-1996: Strong effect where many ratings are made when the movie is first screen, then very quiet period.

- Medium years 1996-1998: Very pale in early weeks getting abit darker from 1999 (going down in a diagonal from top-left to bottom right follows a constant year). We can give any intuitive for this, apart from democratisation of the Internet. This is pure conjecture.

- Recent years 2000 to now: More or less constant colour.

3.2.3 Statement 3

> *The decision to watch a movie that came out decades ago is a very deliberate process of choice. There is a* survival effect *in the sense that time sieved out bad movies. We could expect old movies, e.g.* Citizen Kane*, to be rated higher on average than recent ones.*

There is clearly an effect where the average rating goes down. More striking is that recent movies are more likely to receive a bad rating, where the variance of ratings for movies before the early seventies is much lower.

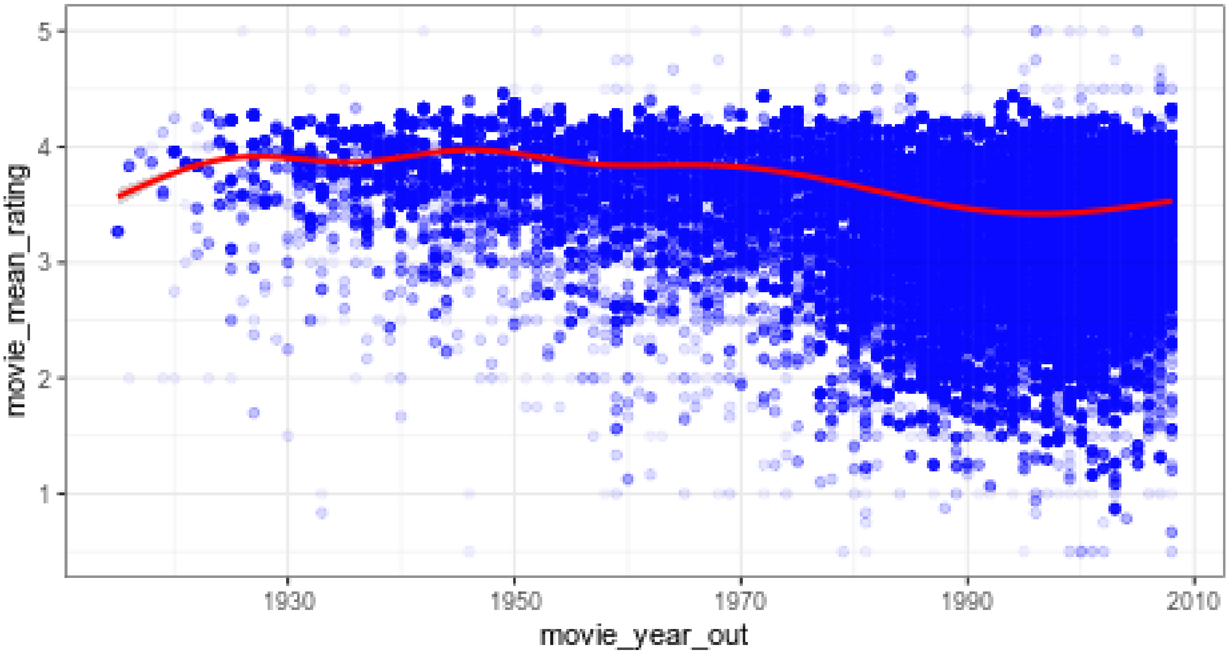
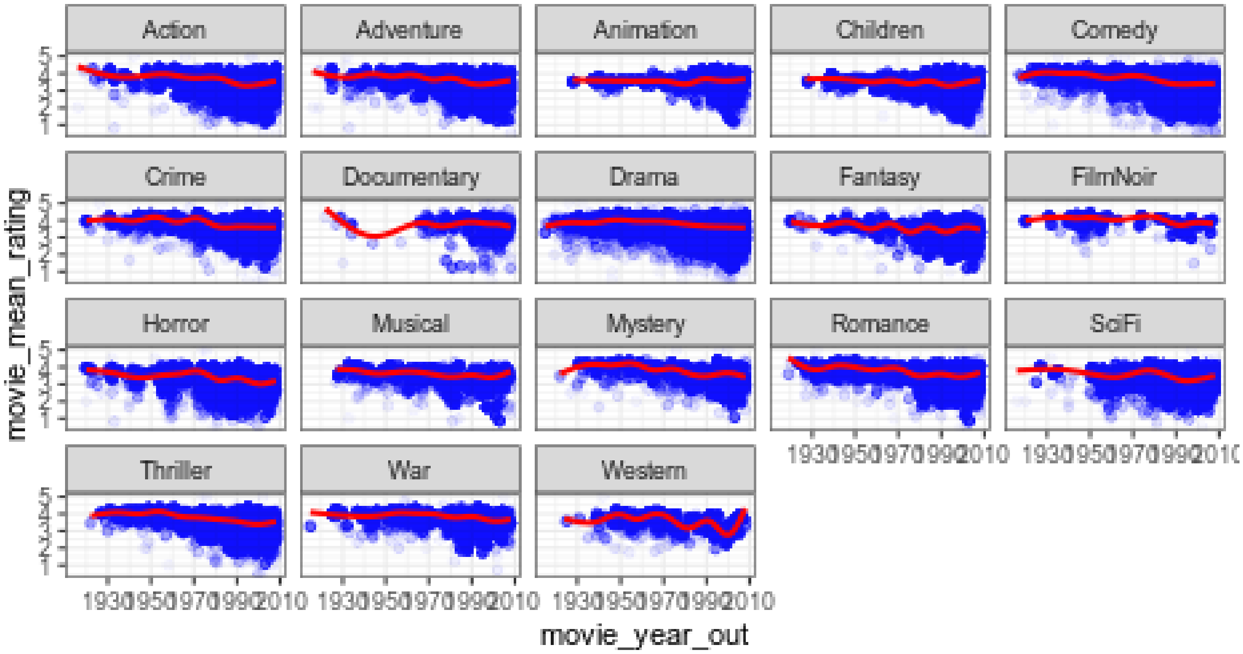


Figure 3.8: Average rating depending on the premiering year

This being said, the impact on average movie ratings is fairly small: it goes from just under 4 to mid-3.

The statement broadly holds on a genre by genre basis. However, this is clearly not the case for (1) Animation/Children movies (whose quality has dramatically improved and CGI animation clearly caters to a wider audience) and (2) Westerns who have become rarer in recent times and possibly require very strong story/cast to be produced (hence higher average ratings).



3.2.4 Statement 4

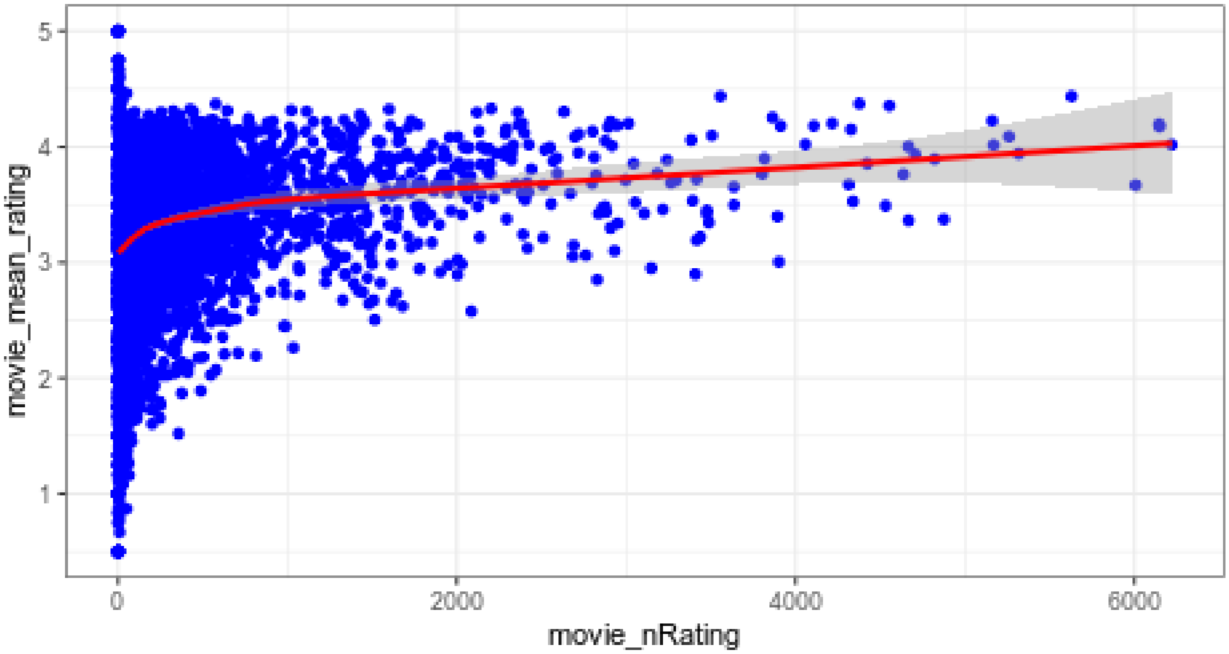
> *In the short term, just a few weeks would make a difference on how a movie is perceived. But whether a movie is 50- or 55-year old would be of little impact. In other words, some sort of rescaling of time, logarithmic or other, need considering.*

More generally, ratings are more variable in early weeks than later weeks. See Statement 1 plot.

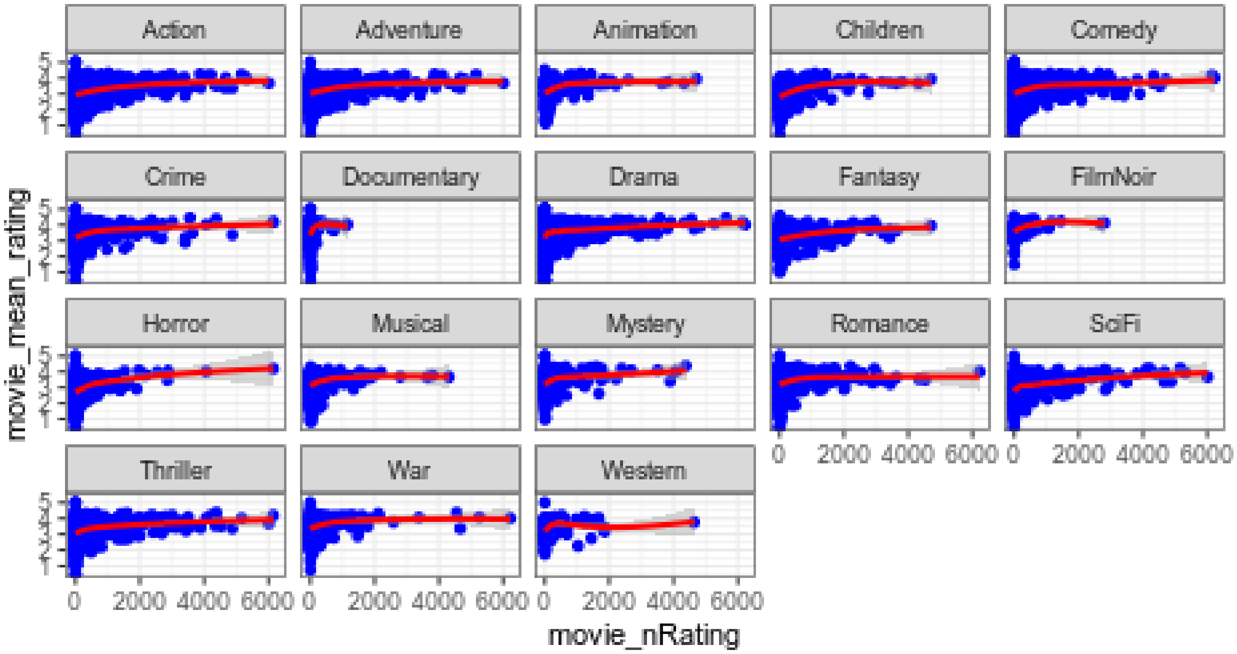
3.2.5 Statement 5

> *If a movie is very good, many people will watch it and rate it. In other words, we should see some correlation between ratings and numbers of ratings. Again, some sort of rescaling of time, logarithmic or other, need considering.*

The effect of good movies attracting many spectators is noticeable. It is also very clear that movies with few spectators generate extremely variable results.

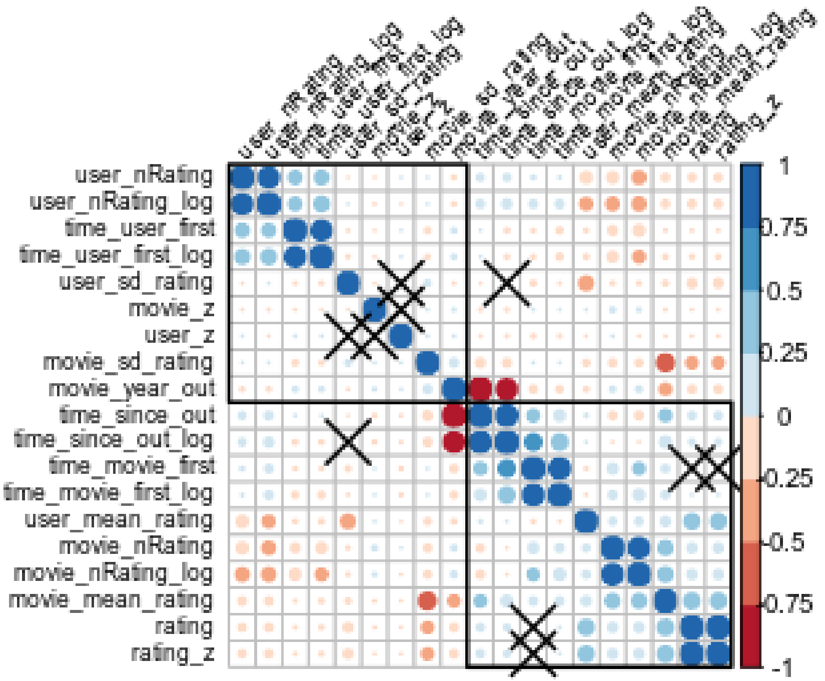


This effect remains on a genre by genre basis.

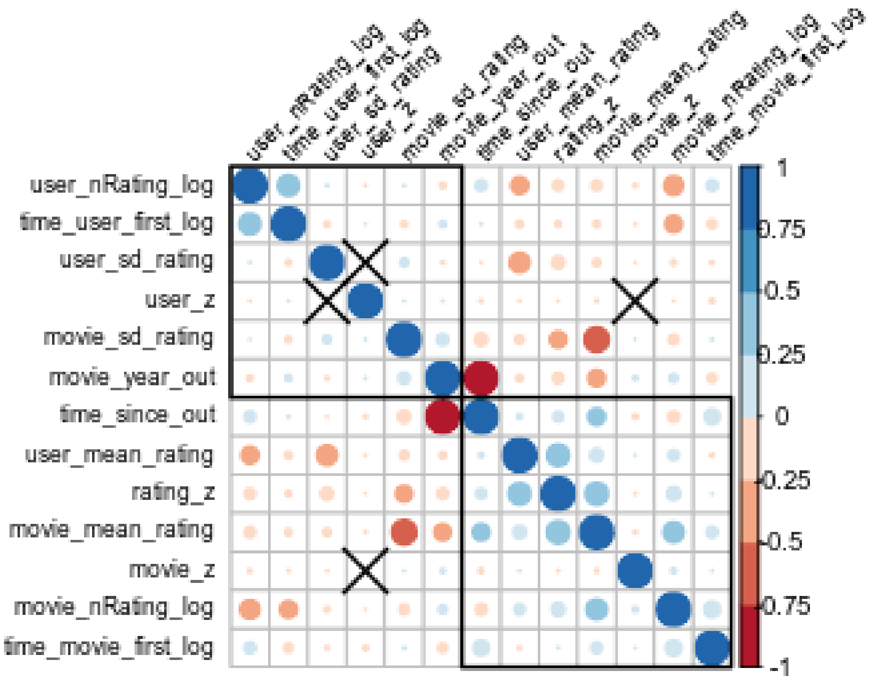


3.2.6 Correlations

We plotted variable-to-variable correlations. Nothing striking appears: strongly correlated variables are where they chould be (e.g. a variable and its z-score). All interesting correlations are in line with the intuitive statements proposed above.



On a reduced set of variables, the plot becomes:



Chapter 4

Model

From the previous sections, the following variables `list_features` have been shown to be possibly relevant:

```
##  [1] "rating"               "movie_nRating_log"    "movie_z"
##  [4] "movie_mean_rating"    "movie_sd_rating"      "user_nRating_log"
##  [7] "user_z"               "user_mean_rating"     "user_sd_rating"
## [10] "movie_year_out"       "time_since_out"       "time_movie_first_log"
## [13] "time_user_first_log" "Action"               "Adventure"
## [16] "Animation"            "Children"             "Comedy"
## [19] "Crime"                "Documentary"          "Drama"
## [22] "Fantasy"              "FilmNoir"             "Horror"
## [25] "Musical"              "Mystery"              "Romance"
## [28] "SciFi"                "Thriller"             "War"
## [31] "Western"
```

In this section, we used the reduced and full dataset. However, on all full dataset training attempts, RStudio crashed running out of memory (exceeding 32 GB).

```
# Datasets used for training.
# edx_training is either an extract or the full dataset. See source code.

x <- edx_training %>% select(one_of(list_features)) %>% as.matrix() # 2.1 GB on full set
y <- edx_training %>% select(rating) %>% as.matrix() #
```

The following helper functions:

- Make a prediction given a fitted model and return the validation dataset with squared error of each prediction.
- Appends the validation RMSE to a table that will include the 3 models RMSEs.

```
# Squared error of predictions in descending order
square_fit <- function(fit_model){

  predictions <- fit_model %>% predict(edx_test)

```

```r
  return (edx_test %>%
            cbind(predictions) %>%
            mutate(square_error = (predictions - rating)^2) %>%
            arrange(desc(square_error))
  )
}


RMSEs <- tibble(Model = "Target", RMSE = 0.8649)

add_rmse <- function(name, fit) {
  rm <- sqrt(sum(fit$square_error) / nrow(fit))
  rw <- tibble(Model = name, RMSE = rm)
  RMSEs %>% rbind(rw)
}
```

4.1 Linear regression

The following runs a linear regression on the training data using the predicting variables listed above.

```r
set.seed(42, sample.kind = "Rounding")
start_time <- Sys.time()

fit_lm <- train(rating ~ .,
                data = x,
                method = "lm")

# Make predictions
square_lm <- square_fit(fit_lm)
RMSEs     <- add_rmse("lm", square_lm)
worst_lm  <- square_lm %>% filter(square_error >= 1.5^2)


end_time <- Sys.time()
print(end_time - start_time)

# Results
#  reduced dataset = 0.8946755
#  full dataset = CRASH
```

4.2 Generalised Linear regression

The following runs a generalised linear regression on the training data using the predicting variables listed above.

```r
set.seed(42, sample.kind = "Rounding")
start_time <- Sys.time()
```

```

fit_glm <- train(rating ~ .,
                 data = x,
                 method = "glm")

 # Make predictions
square_glm <- square_fit(fit_glm)
RMSEs      <- add_rmse("glm", square_glm)
worst_glm <- square_glm %>% filter(square_error >= 1.5^2)


end_time <- Sys.time()
print(end_time - start_time)


# Results
#  reduced dataset = 0.9486
#  full dataset = CRASH
```

4.3 LASSO regression

The following runs a regularised linear regression on the training data using the predicting variables listed above.

LASSO stands for Least Absolute Shrinkage and Selection Operator. The regularisation operates in two ways:

- The absolute values of the coeeficients is minimised.
- Values below a certain threshold are nil-led, effectively removing predictors.

```
# save(fit_lasso, square_lasso, worst_glm, file = "datasets/model_lasso.rda")
# load("datasets/model_lasso.rda")

set.seed(42, sample.kind = "Rounding")

lambda <- 10^seq(-3, 3, length = 10)

fit_lasso <- train(
  rating ~.,
  data = x,
  method = "glmnet",
  trControl = trainControl("cv", number = 10),
  tuneGrid = expand.grid(alpha = 1, lambda = lambda)
  )

# Model coefficients
coef(fit_lasso$finalModel, fit_lasso$bestTune$lambda)

```

```r
# Make predictions
square_lasso <- square_fit(fit_lasso)
RMSEs        <- add_rmse("lasso", square_lasso)
worst_lasso <- square_lasso %>% filter(square_error >= 1.5^2)

end_time <- Sys.time()
print(end_time - start_time)


# Results
#  reduced dataset = 0.94837
#  full dataset = CRASH
```

4.4 Conclusion

Those models, although initially promising, do fail to meet our expectations:

- They reach an RMSE which is good but not below the threshold of 0.8649. The linear regression model performed best with an RMSE = 0.8946.
- More importantly, the training and validation on a very small sample of the datasets (20%). The computational resources required to do anything with more data or more sophisticated models has been out of reach (RStudio has crashed numerous times in the process).

Chapter 5

Stochastic Gradient Descent

The previous models were based on the expectation that our intuitions, confirmed by visual inspection of the dataset, would lead to better performing models. This section shows this is incorrect. We here present a more *"brute-force"* model: a *low-rank matrix factorisation* with is approximated by a *stochastic gradient descent.*

This model proves to be very efficient:

- Before any training, the validation set RMSE is 0.88516 thanks to a non-naive (i.e. not random) initialisation;
- After very little training, using the initial 3 features (explained below), the RMSE became 0.8304 which improves on the targetted RMSE.
- A few hours of training brings the RMSE down to *0.7996* with 11 features.[1] Visually, the RMSE improvements suggest that additional features may help.

5.1 Latent factor model

The approach we follow ia a Latent Factor Model. This section partly draws on part 9 of the Stanford Machine Learning course taught by Andrew Ng (which we previously completed), and a blog post by Sonya Sawtelle.[2]

In essence, this is a dimension reduction model. But two differences reduce the computational workload:

- Users and movies are coalesced into groups of similar users and similar movies. This is purely based on the triplets user / movie / rescaled rating. Information about dates, genres is ignored.
- The model is trained by Stochastic Gradient Descent (SGD). Gradient descent methods are a class of optimisation algorithms that minimise a cost function following downward gradients. SGD is a stochastic version of it where random subsets of the training set are used to converge on very large datasets.

[1] This is a white lie since many more hours went into exploring the various parameters.

[2] See https://sdsawtelle.github.io/blog/output/week9-recommender-andrew-ng-machine-learning-with-python.html

5.2 Formal description

5.2.1 Low-rank factorisation

5.2.1.1 Singular Value Decomposition

As noted in the course material (section 34.1.1), singular value decomposition (SVD) is widely used in machine learning. Wikipedia provides a general description which includes a geometric intuition.[3]

However, this approach is not feasible: the dimensions are too large, the dataset is extremely sparse.

5.2.1.2 Low-rank matrix factorisation (LRMF)

At a high level, the purpose of this assignment is to estimate a matrix R of N_{users} rows by N_{movies} columns, where each value contains the rating given by a user to a movie. This is to be estimated from a sample of values from the training set.

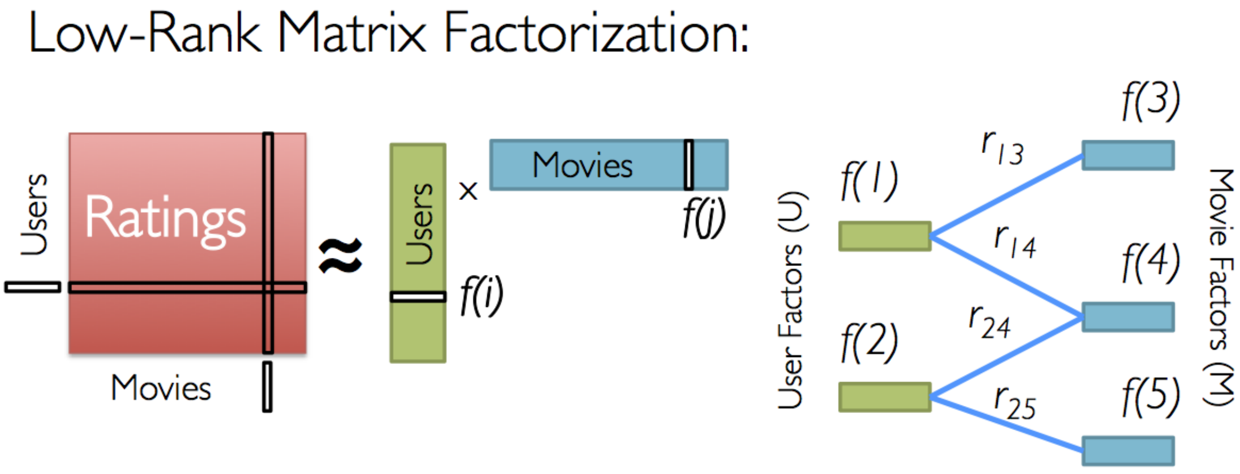
The intuitive and geometric intuition of LRMF is as follows:

- Work in a low dimensional space (k dimensions).
- In that space, give each user u and movie m coordinates in that space ($u = (u_1, ..., u_k)$ and $m = (m_1, ..., m_k)$).
- Note that the cross-product of two points in that space will be zero or close to zero if the points are in perpendicular directions. Conversely, points in a close zone in that space will have a cross-product away from zero. In that sense, movies and users can be grouped together in that space: similar movies would be in the same zone of space, different movies would be in perpendicular positions. Because movies ans users both have coordinates in that space, then can all be mixed and grouped: one can measure the similarities between movies, user or between movie and user. The dimensions are commonly called *features*.

In practice, LRMF is represented by two matrices each with k columns: P of N_{users} rows, and Q of N_{users} rows. The k columns give the k coordinates of each user and movie in the feature space. Choosing a user and a movie, the cross-product of the corresponding rows in P and Q gives $u \times m = \sum_{i=1}^{k} u_i m_i$ and should produce a rating.

The purpose of the algorithm is then to estimate P and Q so that the cross-products match that of the training sets. That is, n matrix notation, R is estimated by $PQ^\top$.

[3] https://en.wikipedia.org/wiki/Singular_value_decomposition



[4]

It is important to note that the only information used is the rating. Knowledge about the genres of the movies, timestamp of a rating, year a movie premiered is ignored.

5.2.2 Gradient Descent

SVD is not useful in our context because (1) the size of the matrices involved is too large, and (2) more importantly requires a fully populated matrix (filling out missing values is a difficult issue).

Instead, we will iteratively estimate the P and Q matrices' coefficients by gradient descent. The cost function used represents prediction error with an additional regularisation cost over those coefficients.

5.2.2.1 Cost function

Let's use the following terms:

- Ω is the set of all $(user, movie)$ pairs in the training set;
- For each (u, m) in Ω, $r_{u,m}$ is the rating in the training set.
- P is written as $p_{i,k}$, Q is written as $q_{j,k}$ with $i \in [1, ..., N_{users}]$, $j \in [1, ..., N_{movies}]$ and $k \in [1, ..., N_{features}]$.
- λ is the regularisation parameter.

Our regularised cost function is written:

$$J_{P,Q} = \sum_{(i,j)\in\Omega} \left(r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k} q_{j,k} \right)^2 + \frac{\lambda}{2} \left(\sum_{i,k} p_{i,k}^2 + \sum_{j,k} q_{j,k}^2 \right)$$

The gradient descent algorithm seeks to minimise the $J_{P,Q}$ cost function by step-wise update of each model parameter x as follows:

$$x_{t+1} \leftarrow x_t - \alpha \frac{\partial J_{P,Q}}{\partial x}$$

The parameters are the matrix coefficients $p_{i,k}$ $q_{j,k}$. α is the learning parameter that needs to be adjusted.

[4] source: https://towardsdatascience.com/large-scale-jobs-recommendation-engine-using-implicit-data-in-pyspark-ccf8df5d910e

5.2.2.2 Cost function partial derivatives

The partial derivatives of the cost function is:

$$\frac{\partial J_{P,Q}}{\partial x} = \frac{\partial}{\partial x}\left(\sum_{(i,j)\in\Omega}\left(r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k}q_{j,k}\right)^2 + \frac{\lambda}{2}\left(\sum_{i,k} p_{i,k}^2 + \sum_{j,k} q_{j,k}^2\right)\right)$$

$$\frac{\partial J_{P,Q}}{\partial x} = \sum_{(i,j)\in\Omega} 2\frac{\partial r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k}q_{j,k}}{\partial x}\left(r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k}q_{j,k}\right) + \frac{\lambda}{2}\left(\sum_{i,k} 2\frac{\partial p_{i,k}}{\partial x}p_{i,k} + \sum_{j,k} 2\frac{\partial p_{i,k}}{\partial x}q_{j,k}\right)$$

We note that $r_{i,j}$ are constants

$$\frac{\partial J_{P,Q}}{\partial x} = 2\sum_{(i,j)\in\Omega}\sum_{k=1}^{N_{features}}\left(\frac{\partial - p_{i,k}q_{j,k}}{\partial x}\left(r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k}q_{j,k}\right)\right) + \lambda\left(\sum_{i,k}\frac{\partial p_{i,k}}{\partial x}p_{i,k} + \sum_{j,k}\frac{\partial p_{i,k}}{\partial x}q_{j,k}\right)$$

If x is a coefficient of P (resp. Q), say $p_{a,b}$ (resp. $q_{a,b}$), all partial derivatives will be nil unless for $(i,j) = (a,b)$.

Therefore:

$$\frac{\partial J_{P,Q}}{\partial p_{a,b}} = -2\sum_{(i,j)\in\Omega} q_{j,b}\left(r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k}q_{j,k}\right) + \lambda p_{a,b}$$

and,

$$\frac{\partial J_{P,Q}}{\partial q_{a,b}} = -2\sum_{(i,j)\in\Omega} p_{i,b}\left(r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k}q_{j,k}\right) + \lambda q_{a,b}$$

Since $\epsilon i, j = r_{i,j} - \sum_{k=1}^{N_{features}} p_{i,k}q_{j,k}$ is the rating prediction error, this becomes:

$$\frac{\partial J_{P,Q}}{\partial p_{a,b}} = -2\sum_{(i,j)\in\Omega} q_{j,b}\epsilon_{i,j} + \lambda p_{a,b}$$

and,

$$\frac{\partial J_{P,Q}}{\partial q_{a,b}} = -2\sum_{(i,j)\in\Omega} p_{i,b}\epsilon_{i,j} + \lambda q_{a,b}$$

5.2.3 Stochastic Gradient Descent (SGD)

The size of the datasets is prohibitive to do those calculations across the entire training set.

Instead, we will repeatedly update the model parameters on small random samples of the training set.

Chapter 14 of (Shalev-Shwartz and Ben-David 2014) gives an extensive introduction to various SGD algorithms.

We implemented a simple version of the algorithm and present the code in more detail.

5.3 SGD Code walk

The algorithm is implemented from scratch and relies on nothing but the `Tidyverse` libraries.

```r
library(tidyverse)
```

The quality of the training and predictions is measured by the *root mean squared error* (RMSE), for which we define a few helper functions (the global variables are defined later):

```r
rmse_training <- function(){
  prediction_Z <- rowSums(Matrices$P[tri_train$userN,] *
                            Matrices$Q[tri_train$movieN,])
  prediction <- prediction_Z * r_sd + r_m
  sqrt( sum((tri_train$rating - prediction)^2 / nSamples) )
}

rmse_validation <- function(){
  prediction_Z <- rowSums(Matrices$P[tri_test$userN,] *
                            Matrices$Q[tri_test$movieN,])
  prediction <- prediction_Z * r_sd + r_m
  sqrt( sum((tri_test$rating - prediction)^2) / nTest )
  }

sum_square <- function(v){
  return (sqrt(sum(v^2) / nrow(v)))
}
```

The key function updates the model coefficients. Its inputs are:

- a list that contains the P an Q matrices, the training RMSE of those matrices, and a logical value indicating whether this RMSE is worse than what it was before the update (i.e. did the update diverge).
- a `batch_size` that defines the number of samples to be drawn from the training set. A normal gradient descent would use the full training set; by default we only use 10,000 samples out of 10 million (one tenth of a percent).
- The cost regularisation `lambda` and gradient descent learning parameter `alpha`.
- A number of `times` to run the descent before recalculating the RMSE and exiting the function (calculating the RMSE is computationally expensive).

The training set used is less rich than the original set. As discussed, it only uses the rating (more exactly on the z_score of the rating). Genres, timestamps,... are discarded.

```r
# Iterate gradient descent
stochastic_grad_descent <- function(model, times = 1,
                                    batch_size = 10000, lambda = 0.1, alpha = 0.01,
                                    verbose = TRUE) {

  # Run the descent `times` times.
  for(i in 1:times) {

    # Extract a sample of size `batch_size` from the training set.
    spl <- sample(1:nSamples, size = batch_size, replace = FALSE)
    spl_training_values <- tri_train[spl,]

    # Take a subset of `P` and `Q` matching the users and
    # movies in the training sample.
    spl_P <- model$P[spl_training_values$userN,]
    spl_Q <- model$Q[spl_training_values$movieN,]

    # rowSums returns the cross-product for a given user and movie.
    # err is the term inside brackets in the partial derivatives
    # calculation above.
    err <- spl_training_values$rating_z - rowSums(spl_P * spl_Q)

    # Partial derivatives wrt p and q
    delta_P <- -err * spl_Q + lambda * spl_P
    delta_Q <- -err * spl_P + lambda * spl_Q

    model$P[spl_training_values$userN,]  <- spl_P - alpha * delta_P
    model$Q[spl_training_values$movieN,] <- spl_Q - alpha * delta_Q

  }

  # RMSE against the training set
  error <- sqrt(sum(
    (tri_train$rating_z - rowSums(model$P[tri_train$userN,] *
                                    model$Q[tri_train$movieN,]))^2)
    / nSamples )

  # Compares to RMSE before update
  model$WORSE_RMSE <- (model$RMSE < error)
  model$RMSE <- error

  # Print some information to keep track of success
  if (verbose) {
    cat("  # features=", ncol(model$P),
        "  J=",  nSamples * error ^2 +
          lambda/2 * (sum(model$P^2) + sum(model$Q^2)),
        "  Z-scores RMSE=", model$RMSE,
        "\n")
    flush.console()
  }

    return(model)
}
```

Now that the functions are defined, we prepare the data sets.

- First load the original data if not already available.

```
# Load the datasets which were saved on disk after using the course source code.
if(!exists("edx"))        edx <- readRDS("datasets/edx.rds")
if(!exists("validation")) validation <- readRDS("datasets/validation.rds")
```

- Calculate the z-score of all ratings.

```
# Creates a movie index from 1 to nMovies
r_m <- mean(edx$rating)
r_sd <- sd(edx$rating)

training_set <- edx %>%
  select(userId, movieId, rating) %>%
  mutate(rating_z = (rating - r_m) / r_sd)

test_set <- validation %>%
  select(userId, movieId, rating) %>%
  mutate(rating_z = (rating - r_m) / r_sd)
```

- We do not know if there are any gaps in the userId's and movieId's in the datasets. They cannot be used as the row numbers of the P and Q matrices. Therefore we count how many distinct users and movies there are and create an index to link a movieId (resp. userId) to its Q (resp. P) -matrix row number.

```
movieIndex <-
  training_set %>%
  distinct(movieId) %>%
  arrange(movieId) %>%
  mutate(movieN = row_number())

userIndex <-
  training_set %>%
  distinct(userId) %>%
  arrange(userId) %>%
  mutate(userN = row_number())
```

- For each movie and user, we calculate its mean rating z-score.

```r
movieMean <-
  training_set %>%
  group_by(movieId) %>%
  summarise(m = mean(rating_z))

userMean <-
  training_set %>%
  group_by(userId) %>%
  summarise(m = mean(rating_z))
```

- We can now create the training and validation sets contining the movie index (instead of the movieId), user index and ratings (original and z-score).

```r
# Training triplets with z_score
tri_train <- training_set %>%
  left_join(userIndex, by = "userId") %>%
  left_join(movieIndex, by = "movieId") %>%
  select(-userId, -movieId)

tri_test <- test_set %>%
  select(userId, movieId, rating) %>%
  left_join(userIndex, by = "userId") %>%
  left_join(movieIndex, by = "movieId") %>%
  select(-userId, -movieId) %>%
  mutate(rating_z = (rating - r_m)/r_sd,
         error = 0)
```

```r
nSamples <- nrow(tri_train)
nTest <- nrow(tri_test)

nUsers <- tri_train %>% select(userN) %>%  n_distinct()
nMovies <- tri_train %>% select(movieN) %>%  n_distinct()
```

- The P and Q matrices are defined with 3 latent factors to start with.

```r
# number of initial latent factors
nLF <- 3

LF_Model <- list( P = matrix(0, nrow = nUsers, ncol = nLF),
                  Q = matrix(0, nrow = nMovies, ncol = nLF),
                  RMSE = 1000.0,
                  WORSE_RMSE = FALSE)
```

- To speed up the training, the matrices are initialised so that the cross product is the sum of the movie average z-rating (m_{movieN}) and user z-rating (u_{userN}).

$$P \times Q^\top = \begin{bmatrix} 1 & u_1 & 0 \\ 1 & u_2 & 0 \\ \vdots & \vdots & \vdots \\ 1 & u_i & 0 \\ \vdots & \vdots & \vdots \\ 1 & u_{nUser} & 0 \end{bmatrix} \times \begin{bmatrix} m_1 & m_2 & \cdots & m_j & \cdots & m_{nMovies} \\ 1 & 1 & \cdots & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} & \vdots & \\ \cdots & u_i + m_j & \cdots \\ & \vdots & \end{bmatrix}$$

```r
# Features matrices are initialised with:
# Users: 1st column is 1, 2nd is the mean rating (centered), rest is noise
# Movies: 1st column is the mean rating (centered), 2nd is 1, rest is noise
#
# That way, the matrix multiplication will start by giving reasonable value

LF_Model$P[,1] <- matrix(1, nrow = nUsers, ncol = 1)
LF_Model$P[,2] <- as.matrix(userIndex %>%
                              left_join(userMean, by ="userId") %>% select(m))

LF_Model$Q[,1] <- as.matrix(movieIndex %>%
                              left_join(movieMean, by ="movieId") %>% select(m))
LF_Model$Q[,2] <- matrix(1, nrow = nMovies, ncol = 1)
```

- Random noise is added to all model parameters, otherwise the gradient descent has nowhere to start (zeros wipe everything in the matrix multiplications).

```r
# Add random noise
set.seed(42, sample.kind = "Rounding")
LF_Model$P <- LF_Model$P + matrix(rnorm(nUsers * nLF,
                                              mean = 0,
                                              sd = 0.01),
                                        nrow = nUsers,
                                        ncol = nLF)

LF_Model$Q <- LF_Model$Q + matrix(rnorm(nMovies * nLF,
                                              mean = 0,
                                              sd = 0.01),
                                        nrow = nMovies,
                                        ncol = nLF)
```

- We also have a list that keeps track of all the training steps and values.

```r
rm(list_results)
list_results <- tibble("alpha" = numeric(),
                       "lambda" = numeric(),
                       "nFeatures" = numeric(),
                       "rmse_training_z_score" = numeric(),
                       "rmse_training" = numeric(),
                       "rmse_validation" = numeric())
```

The main training loop runs as follows:

- We start with 3 features.
- The model is updated in batches of 100 updates. This is done up to 250 times. At each time, if the model starts diverging, the learning parameter (α) is reduced.
- Once the 250 times have passed, or if α has become incredibly small, or if the RMSE doesn't really improve anymoe (by less than 1 millionth), we add another features and start again.

```r
initial_alpha <- 0.1
for(n in 1:100){

  # Current number of features
  number_features <- ncol(LF_Model$P)

  # lambda = 0.01 for 25 features, i.e. for about 2,000,000 parameters.
  # We keep lambda proportional to the number of features
  lambda <- 0.1 * (nUsers + nMovies) * number_features / 2000000

  alpha <- initial_alpha

  cat("CURRENT FEATURES: ", number_features,
      "---- Pre-training validation RMSE = ", rmse_validation(), "\n")

  list_results <- list_results %>% add_row(alpha = alpha,
                                           lambda = lambda,
                                           nFeatures = number_features,
                                           rmse_training_z_score = LF_Model$RMSE,
                                           rmse_training = rmse_training(),
                                           rmse_validation = rmse_validation())

  for (i in 1:250){
    pre_RMSE <- LF_Model$RMSE
    LF_Model <- stochastic_grad_descent(model = LF_Model,
                                        times = 100,
                                        batch_size = 1000 * number_features,
                                        alpha = alpha,
                                        lambda = lambda)

    list_results <- list_results %>% add_row(alpha = alpha,
                                             lambda = lambda,
                                             nFeatures = number_features,
                                             rmse_training_z_score = LF_Model$RMSE,
                                             rmse_training = rmse_training(),
                                             rmse_validation = rmse_validation())

    if (LF_Model$WORSE_RMSE) {
      alpha <- alpha / 2
      cat("Decreasing gradient parameter to: ", alpha, "\n")
    }

    if (initial_alpha / alpha > 1000 |
        abs( (LF_Model$RMSE - pre_RMSE) / pre_RMSE) < 1e-6) {
      break()
      }
  }


  # RMSE against validation set:
  rmse_validation_post <- rmse_validation()
  cat("CURRENT FEATURES: ", number_features,
      "---- POST-training validation RMSE = ", rmse_validation_post, "\n")

  # if (number_features == 12){
  #   break()
  # }


  # Add k features
  k_features <- 1
  LF_Model$P <- cbind(LF_Model$P,
                         matrix(rnorm(nrow(LF_Model$P) * k_features,
                                      mean = 0,
                                      sd = sd(LF_Model$P)/100),
                                nrow = nrow(LF_Model$P),
                                ncol = k_features))

  LF_Model$Q <- cbind(LF_Model$Q,
                      matrix(rnorm(nrow(LF_Model$Q) * k_features,
                                   mean = 0,
                                   sd = sd(LF_Model$Q)/100),
                             nrow = nrow(LF_Model$Q),
                             ncol = k_features))

}
```

The following table shows the RMSE on the validation set that is obtained for a given number of features.

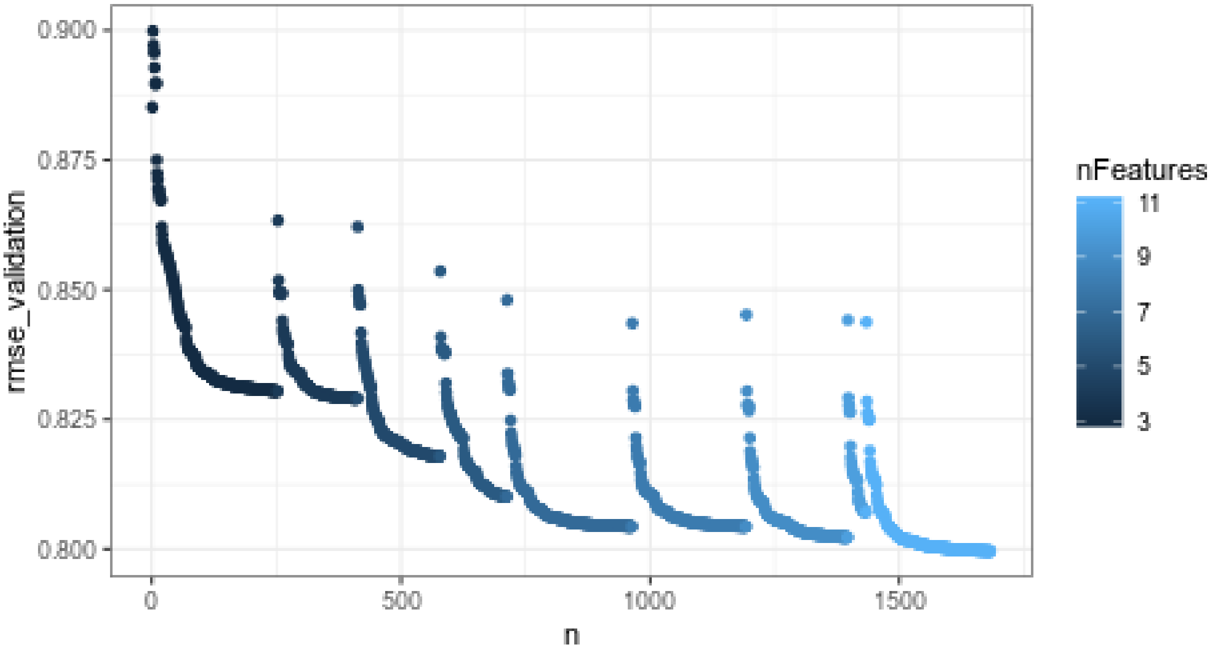


Figure 5.1: Plot of the RmSE on the validation test

nFeatures	best_RMSE
3	0.830465
4	0.829026
5	0.817902
6	0.810112
7	0.804342
8	0.804338
9	0.802270
10	0.802270
11	0.799617

This plot shows the progress of the RMSE on the validation set. It shows an overall improvement with the number of features, with little worsening spikes each time a feature seeded with random values is added.

We also developed a re-implementation in Julia (availabe on Gihub) that we used to cross-check the R implementation. It gave similar results (in much less time):

Chapter 6

Conclusion

The project developed a recommender system for a movies' set of ratings. After exploring the dataset and visually confirming a number of intuitions about movie ratings, we ran three models: a linear regression, a generalised linear model and a Lasso model (regularised linear model). All three performed poorly against the project target.

We then estimated ratings with a low-rank matrix factorisation estimated through a stochastic gradient descent. This proved very efficient and yielded a RMSE of 0.7996 against the validation dataset.

Two further work avenues are suggested:

- Convergence speed could potentially be improved by noting that the cost function is λ-strongly convex, and the SGD algorithm can be improved. See section 14.5.3 of (Shalev-Shwartz and Ben-David 2014).
- The three models proposed can also be formulated as minimisation of a cost function that can then be minimised using stochastic gradient descent, therefore being able to use the entire dataset.

Chapter 7

Appendix

7.1 Session Info

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu Eoan Ermine (development branch)
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.3.7.so
##
## Random number generation:
##  RNG:     Mersenne-Twister
##  Normal:  Inversion
##  Sample:  Rounding
##
## locale:
##  [1] LC_CTYPE=en_AU.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_AU.UTF-8        LC_COLLATE=en_AU.UTF-8
##  [5] LC_MONETARY=en_AU.UTF-8    LC_MESSAGES=en_AU.UTF-8
##  [7] LC_PAPER=en_AU.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_AU.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
##  [1] corrplot_0.84      RColorBrewer_1.1-2 kableExtra_1.1.0
##  [4] Metrics_0.1.4      gridExtra_2.3      doParallel_1.0.15
##  [7] iterators_1.0.12   foreach_1.5.1      dslabs_0.7.1
## [10] caret_6.0-84       lattice_0.20-38    lubridate_1.7.4
## [13] forcats_0.4.0      stringr_1.4.0      dplyr_0.8.3
## [16] purrr_0.3.2        readr_1.3.1        tidyr_1.0.0
## [19] tibble_2.1.3       ggplot2_3.2.1.9000 tidyverse_1.2.1
##
## loaded via a namespace (and not attached):
##  [1] httr_1.4.1         jsonlite_1.6       viridisLite_0.3.0
##  [4] splines_3.6.1      prodlim_2018.04.18 modelr_0.1.5
##  [7] assertthat_0.2.1   highr_0.8          stats4_3.6.1
## [10] cellranger_1.1.0   yaml_2.2.0         ipred_0.9-9
## [13] pillar_1.4.2       backports_1.1.4    glue_1.3.1
## [16] digest_0.6.21      rvest_0.3.4        colorspace_1.4-1
## [19] recipes_0.1.7      htmltools_0.3.6    Matrix_1.2-18
## [22] plyr_1.8.4         timeDate_3043.103  pkgconfig_2.0.3
## [25] broom_0.5.2        haven_2.1.1        scales_1.0.0
## [28] webshot_0.5.1      gower_0.2.1        lava_1.6.6
## [31] generics_0.0.2     withr_2.1.2        nnet_7.3-12
## [34] cli_1.1.0          survival_2.44-1.1  magrittr_1.5
## [37] crayon_1.3.4       readxl_1.3.1       evaluate_0.14
## [40] nlme_3.1-141       MASS_7.3-51.4      xml2_1.2.2
## [43] class_7.3-15       tools_3.6.1        data.table_1.12.2
## [46] hms_0.5.1          lifecycle_0.1.0    munsell_0.5.0
## [49] compiler_3.6.1     rlang_0.4.0        grid_3.6.1
## [52] rstudioapi_0.10    labeling_0.3       rmarkdown_1.15
## [55] gtable_0.3.0       ModelMetrics_1.2.2 codetools_0.2-16
## [58] reshape2_1.4.3     R6_2.4.0           knitr_1.25
## [61] zeallot_0.1.0      stringi_1.4.3      Rcpp_1.0.2
## [64] vctrs_0.2.0        rpart_4.1-15       tidyselect_0.2.5
## [67] xfun_0.9
```

References

Bell, Robert M, Yehuda Koren, and Chris Volinsky. 2007. "The Bellkor Solution to the Netflix Prize." *KorBell Team's Report to Netflix.*

———. 2008. "The Bellkor 2008 Solution to the Netflix Prize." *Statistics Research Department at AT&T Research* 1.

Bennett, James, Stan Lanning, and others. 2007. "The Netflix Prize." In *Proceedings of Kdd Cup and Workshop*, 2007:35. New York, NY, USA.

Gower, Stephen. 2014. "Netflix Prize and Svd." Working Paper.

Koren, Yehuda. 2009. "The Bellkor Solution to the Netflix Grand Prize." *Netflix Prize Documentation* 81 (2009): 1–10.

Narayanan, Arvind, and Vitaly Shmatikov. 2006. "How to Break Anonymity of the Netflix Prize Dataset." *CoRR* abs/cs/0610105. http://arxiv.org/abs/cs/0610105.

Piotte, Martin, and Martin Chabbert. 2009. "The Pragmatic Theory Solution to the Netflix Grand Prize." *Netflix Prize Documentation.*

Shalev-Shwartz, Shai, and Shai Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge university press. https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/index.html.

Töscher, Andreas, Michael Jahrer, and Robert M Bell. 2009. "The Bigchaos Solution to the Netflix Grand Prize." *Netflix Prize Documentation*, 1–52.