MovieLens Recommendation System

Ruth Parras

10/20/2021

---

1 Introduction

1.1 Objective

This project builds an algorithm that predicts the ratings for a movie that a user has not yet seen, using ratings given by other people. This type of algorithm is known as "recommendation system" because predicted ratings are used to recommend other movies that a user might like. Recommendation systems are widely used not only in entertainment to rate movies or music but also in eCommerce to propose new items to add to a shopping cart.

1.2 Dataset Description

To develop our algorithm, we use the `edx` dataset which is a random 90% partition of the observations in the `MovieLens 10M` file that can be downloaded here: https://grouplens.org/datasets/movielens/10m/

`edx` contains 9000055 ratings of 10677 movies by 69878 users. Ratings range from 0.5 to 5, with half point increments. Additional attributes include the title of the movie, genres, and timestamp of the rating.

The remainder 10% of the observations (the `validation` set) is used at the end of the project to measure the performance of the algorithm using the residual mean squared error (RMSE).

1.3 Key Activities

Following is an outline of the key steps performed in this project:

- After downloading and cleaning the data, we visualize relationships between movies, users, genres and time. We observe significant variations (biases or effects) in ratings from movie-to-movie, user-to-user, and across genres and time.
- We use these insights to build an algorithm that estimates the rating for a movie as the average across all movies and users after adjusting for the variations that we observe above. Further, we also adjust for noisy estimates due to low number of ratings by using regularization.
- To develop and train our model we use 90% of the observations in `edx` selected at random (the `train_set`), while we set aside the remainder 10% (the `test_set`) to evaluate the algorithm using the residual mean squared error (RMSE).

- Next, we use hierarchical clustering to uncover similarities in rating patterns by groups of users and movies. We leverage the Recommenderlab package and in particular the matrix factorization SVD algorithm to further improve our model.
- Lastly, we use the `validation` set to run our final test. Results are then presented followed by conclusions and recommended next steps.

2 Analysis

2.1 Data Cleaning and Preparation

We start by downloading the MovieLens 10M ratings file, which includes ratings.dat and movies.dat

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

Each line in ratings.dat represents one rating in the form UserID::MovieID::Rating::Timestamp. Likewise, each line of movies.dat represents one movie as MovieID::Title::Genres. After parsing the attributes into two tables, we join them into a dataframe, but exclude observations without ratings.

Finally, we randomly partition the observations in MovieLens into two data sets:

- `edx` containing 90% of the observations, used to build and test our algorithm
- `validation` with the other 10%, used to conduct the final test at the end of the project

2.2 Data Exploration and Visulization

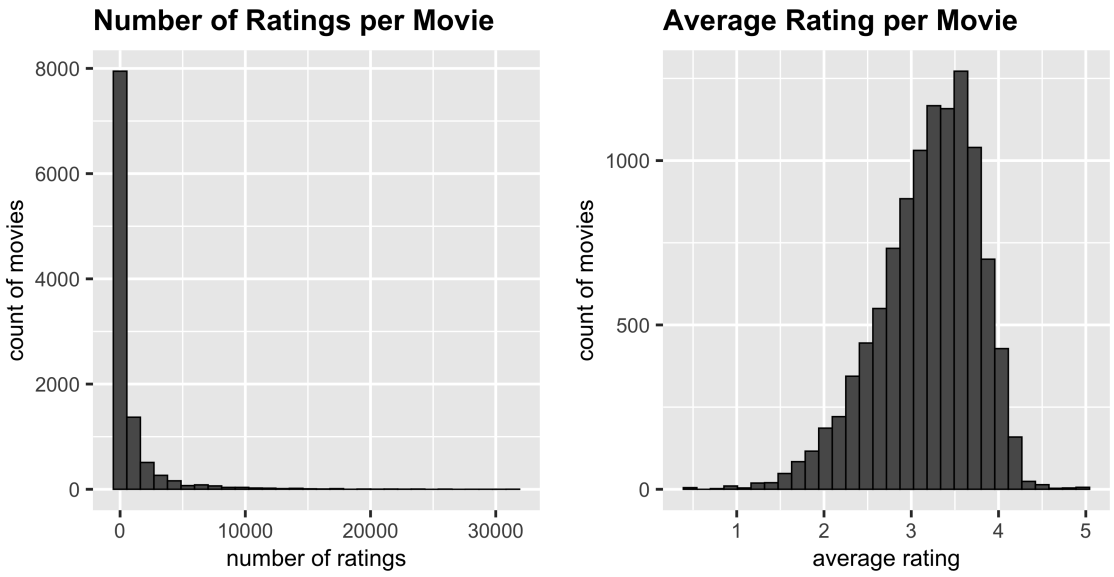
The `edx` set is a 90% random split of the $10M observations in `MovieLens`. Each observation consists on the rating of one movie, by one user, at a given time. The first few rows of `edx` look like this:

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

2.2.1 Relationship between Movies and Ratings

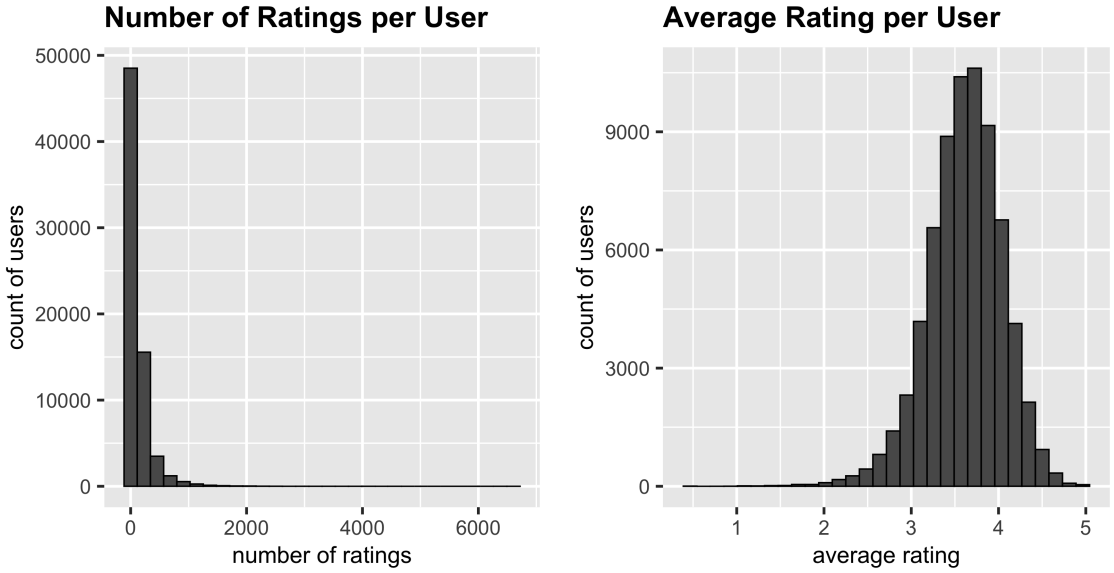
The first plot represents the number of ratings per movie. It is skewed towards lower counts, indicating that the majority of movies are infrequently rated.

The second plot shows large variations in average rating per movie, and is slightly skewed towards higher values. Likely the result of some “bias” in ratings favoring “blockbusters” or “main stream” productions.



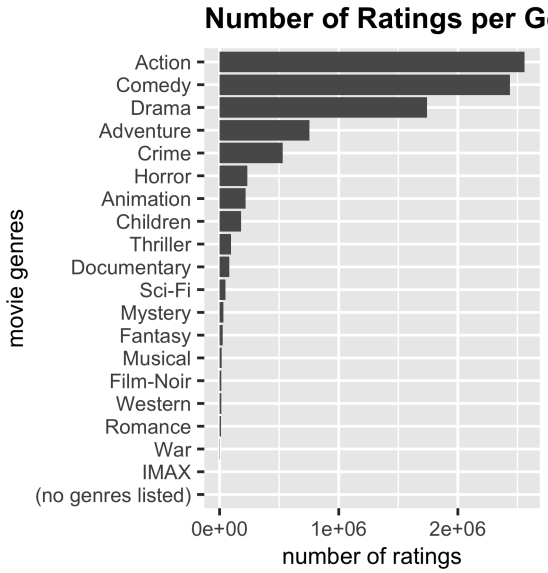
2.2.2 Relationship between Users and Ratings

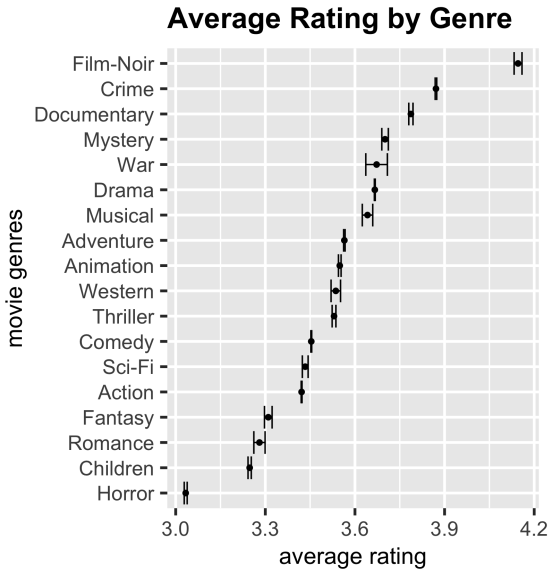
Similarly, we display the number of ratings given by each user and its average. Again, the first plot is skewed towards lower counts, suggesting infrequent ratings, while the second chart is skewed towards higher averages, indicating some users are very generous in their evaluations.



2.2.3 Relationship between Genre and Ratings

Using a boxplot this time, we observe again high variability in ratings, with "horror" movies getting the lowest averages, while "film-noir" gets the best evaluations.



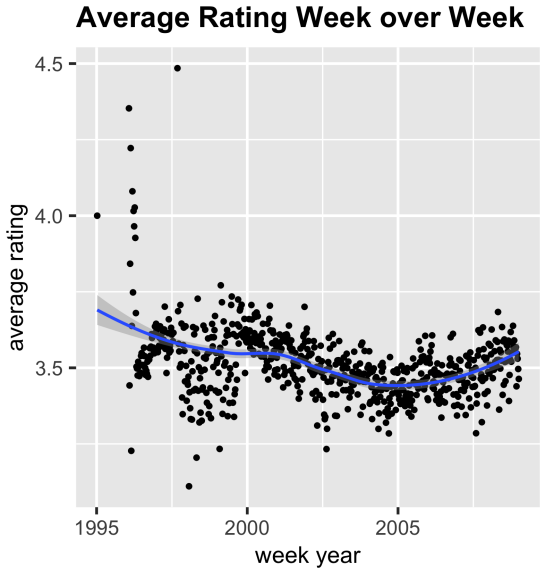


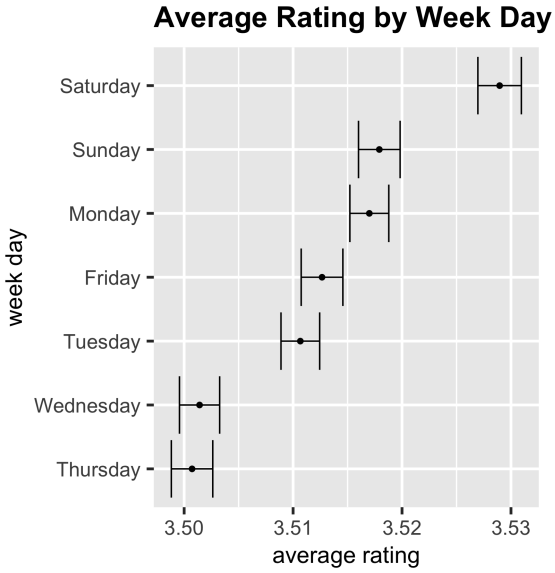
2.2.4 Relationship between Time and Ratings

Lastly, we plot rating averages week-over-week and by day-of-the-week.

The first plot shows that average ratings have been trending down slowly since circa 1995 but started to bounce back around 2005.

The second chart shows ratings being slightly higher around weekends, suggesting that users moods (arguably, people tend to be more relaxed and happier on weekends) have an effect on ratings.





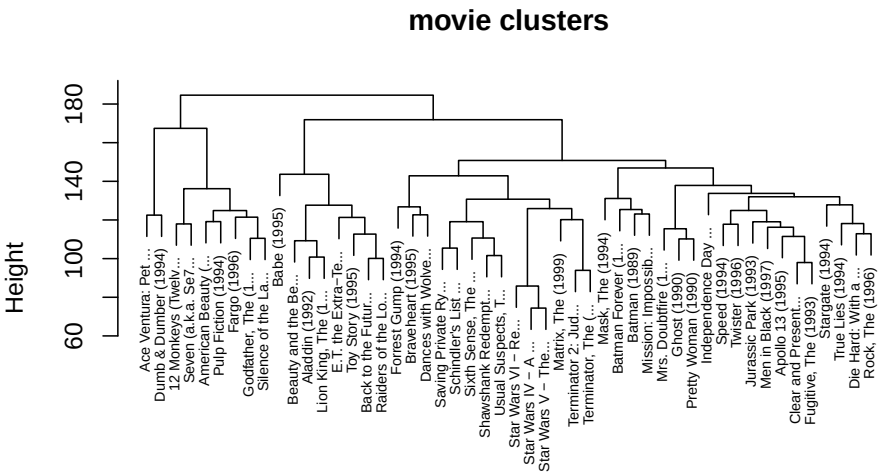
2.2.5 Group Patterns

To explore if there are groups of users or movies with similar rating patterns, we use a subset of the data, `small_edx`, consisting of the 50 most rated movies and the users that have rated at least 25 of them.

2.2.5.1 Clusters of Movies with similar rating patterns

After normalizing `small_edx` by removing row and column averages, we calculate the distance between observations and use **hierarchical clustering** to group movies that are close together. Next, we summarize the information with a dendrogram

```
h <- dist(small_edx) %>% hclust()
```



Ratings among users in the same dendrogram branch are closer together. In fact, if we cut the tree into 10 clusters and display group 3 and 5, for example, we can see similarities.

```
groups <- cutree(h, k = 10)  # generate  10 groups
names(groups)[groups==3]   # family movies
```

```
## [1] "Aladdin (1992)"       "Back to the Futur..." "Beauty and the Be..."
## [4] "E.T. the Extra-Te..." "Lion King, The (1..." "Raiders of the Lo..."
## [7] "Toy Story (1995)"
```

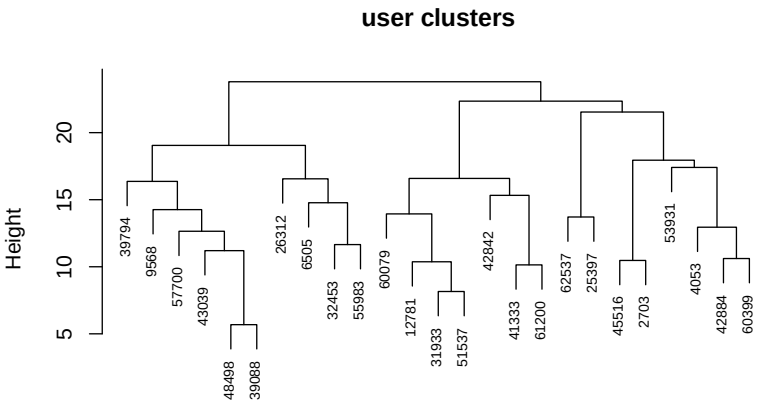
```
names(groups)[groups==5]   # blockbusters
```

```
##  [1] "Apollo 13 (1995)"     "Clear and Present..." "Die Hard: With a ..."
##  [4] "Fugitive, The (1993)" "Independence Day ..." "Jurassic Park (1993)"
##  [7] "Men in Black (1997)"  "Rock, The (1996)"     "Speed (1994)"
## [10] "Stargate (1994)"      "True Lies (1994)"     "Twister (1996)"
```

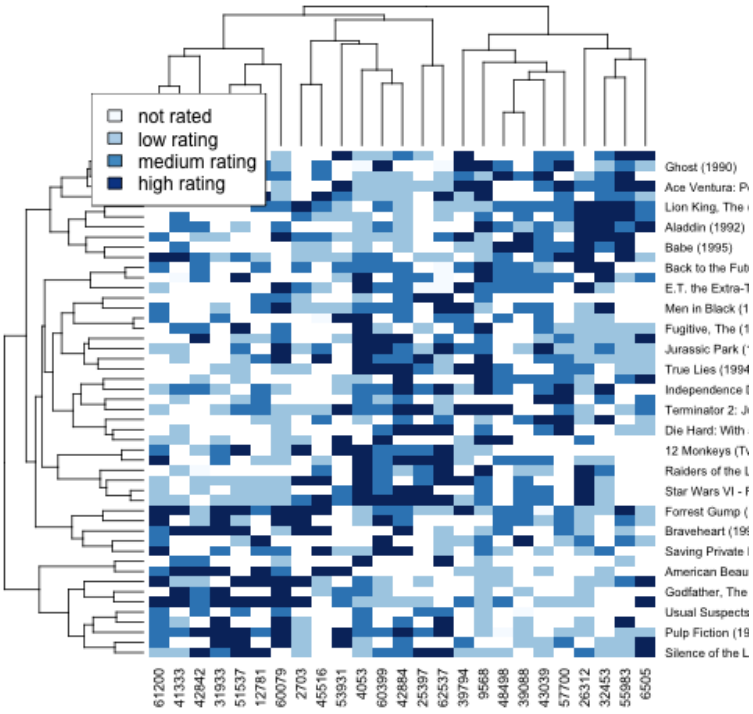
2.2.5.2 Clusters of Users with similar rating patterns

This time, we only include the 25 users with the highest variability in ratings for whom movies are not the same. We transpose `small_edx` to calculate the distance between ratings and cluster users into groups with similar patterns.

```
h2 <- dist(t(small_edx)) %>% hclust()
```



2.2.5.3 Clusters of movies and users Finally, we visualize the combined clusters using a heatmap. Areas with the same shade represent groups of users and movies with similar rating patterns. Note large number of missing ratings (no color).



2.3 Insights

Data exploration has proven very effective in uncovering rating patterns for movies, users and related attributes. Here is a summary of the insights:

1. Rating averages are not uniform across movies, users, genres or time. Instead we observe strong variations or biases.
    - Some movies are widely perceived as better and receive higher evaluations (**movie effect**)
    - Some users have a tendency to give higher or lower ratings (**user effect**)

- Preferences in genre such as "action movies", that consistently score higher ratings (**genre effect**)
- Variations in ratings based on the day of the week, and over time (**time effect**)

2. Most movies have been infrequently rated and most users have given very few evaluations, which could skew predictions and lead to noisy estimates.
3. There are groups of movies and groups of users with similar rating patterns, as validated using hierarchical clustering.

In the next section we use these insights to build and improve our recommendation algorithm.

3 Modeling Approach

We start by leveraging the insight that rating averages are not uniform across movies, users, genres and time, and build a model that adjusts for the different biases or effects. Next, we adjust for infrequent (noisy) ratings using regularization. Finally, we use the **Recommenderlab** package to account for similarities in rating patterns for groups of users and movies.

3.1 Modeling for Effects

3.1.1 Average of Ratings

We start with a basic model Y_{ui} that predicts the same value mu for each movie i and user u

$$\boldsymbol{Y_{ui}} = \boldsymbol{mu} + e_{ui}$$

with e_{iu} the random error that explains variations and mu the average of all known ratings:

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

3.1.2 Adjusting for Movie and User Effects

Next, we adjust for systemic differences (effects) in ratings by movie and users, represented as:

$$\boldsymbol{Y_{ui}} = \boldsymbol{mu} + \boldsymbol{b_i} + \boldsymbol{b_u} + \boldsymbol{e_{ui}}$$

with b_i accounting for movie-to-movie variations in ratings, and b_u user-to-user differences. After making predictions on the test_set, the RMSE is 0.86468. Here is the code:

```
bis <- train_set %>%     # with bis the vector of bi (movie-to-movie variations)
            group_by(movieId) %>%
            summarize(bi = mean(rating- mu))

bus <- train_set %>%   # with bus the vector of bu (user-to-user variations)
            left_join(bis, by="movieId") %>%
            group_by(userId) %>%
            summarize(bu = mean(rating- mu- bi))

# make predictions for unknown ratings on the test_set using: Yui = mu + bi + bu
predictions<- test_set %>%    #
```

```
  left_join(bis, by="movieId") %>%
  left_join(bus, by="userId") %>%
  summarize(pred= mu + bi+ bu) %>%
  pull(pred)

RMSE<- sqrt(mean((test_set$rating - predictions)^2))  # calculate RMSE
```

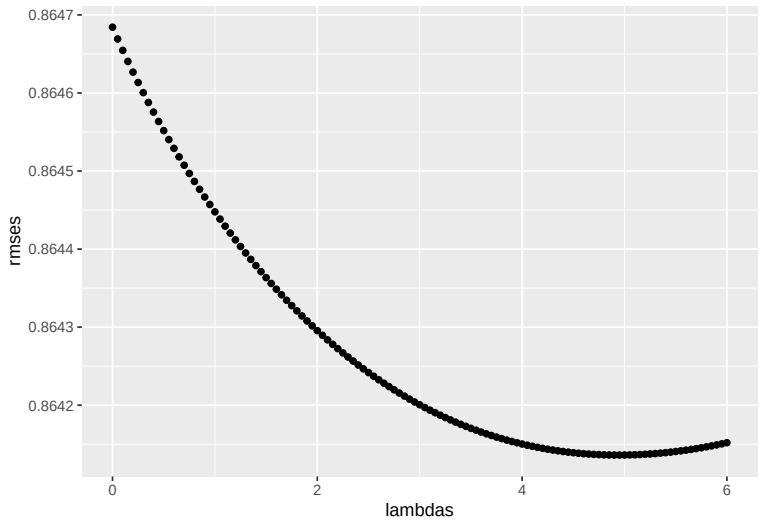
3.1.3 Adjusting for Small Samples: Regularization

As previously discussed, we need to penalize large estimates formed with a small number of ratings. This is achieved by adding a "penalty" *lambda* that shrinks estimates with few ratings, and by calculating the "regularized" *bi* and *bu* that minimize the RMSE. The model looks like this:

$$Y_{ui} = mu + reg\_b_i + reg\_b_u + e_{ui}$$

and the lambda is estimated using cross-validation:

```
lambdas <- seq(0, 6, 0.05)

rmses <- sapply(lambdas, function(l){
  mu<-mean(train_set$rating)

  reg_bis <- train_set%>%
    group_by(movieId) %>%
    summarize(reg_bi = sum(rating - mu)/(n()+l))

  reg_bus <- train_set %>%
    left_join(reg_bis, by="movieId") %>%
    group_by(userId) %>%
    summarize(reg_bu = sum(rating - reg_bi - mu)/(n()+l))

  predictions <-test_set %>%
    left_join(reg_bis, by = "movieId") %>%
    left_join(reg_bus, by = "userId") %>%
    mutate(pred = mu + reg_bi + reg_bu) %>%
    pull(pred)

  return(sqrt(mean((test_set$rating - predictions)^2)))
})
```



The plot above shows that the lambda that minimizes the RMSE is 4.95, and the RMSE is 0.86414

3.1.4 Adjusting for Genre and Time effects

Finally, we adjust our model for systemic variations in ratings by genre (g_{ui}) and day of the week (d_{ui}). The augmented model is:

$$\boldsymbol{Y_{ui} = mu + reg\_b_i + reg\_b_u + g_{ui} + d_{ui} + e_{ui}}$$

After making predictions on the test_set, the RMSE is 0.86367.

```
guis <- train_set %>%   # vector of genre-to-genre variations in rating (gui)
           left_join(reg_bis, by="movieId") %>%
           left_join(reg_bus, by="userID")
           group_by(userId) %>%
           summarize(gui = mean(rating- mu- bi))

duis<- train_set %>%    # vector of day-of-the-week variations in rating (dui)
           left_join(reg_bis, by="movieId") %>%
           left_join(reg_bus, by="userId") %>%
           left_join(guis, by="genre1")  %>%
           group_by(weekday) %>%
           summarise(dui = mean(rating- mu- reg_bi- reg_bu- gui))

# predictions for unknown ratings using: Yui = mu + reg_bi + reg_bu + gui + dui
predictions<- test_set %>%
  mutate(date = as_datetime(timestamp),
         week=round_date(date, "week")) %>%
  left_join(reg_bis, by="movieId") %>%
  left_join(reg_bus, by="userId") %>%
  left_join(guis, by="genres") %>%
  left_join(duis, by="week") %>%
  summarize(pred= mu + reg_bi+ reg_bu + gui + dui) %>%
  pull(pred)

RMSE<- sqrt(mean((test_set$rating - predictions)^2))  # RMSE of the predictions
```

3.2 Using Recommenderlab to account for Group Patterns

Thus far, we have built a model that adjusts for different biases in ratings across movies, users, genres and time. Next, we leverage the insight that there are groups of movies and users with similar rating patterns and augment our model as follows:

$$\boldsymbol{Y_{ui} = mu + reg\_b_i + reg\_b_u + g_{ui} + d_{ui} + r_{ui} + e_{ui}}$$

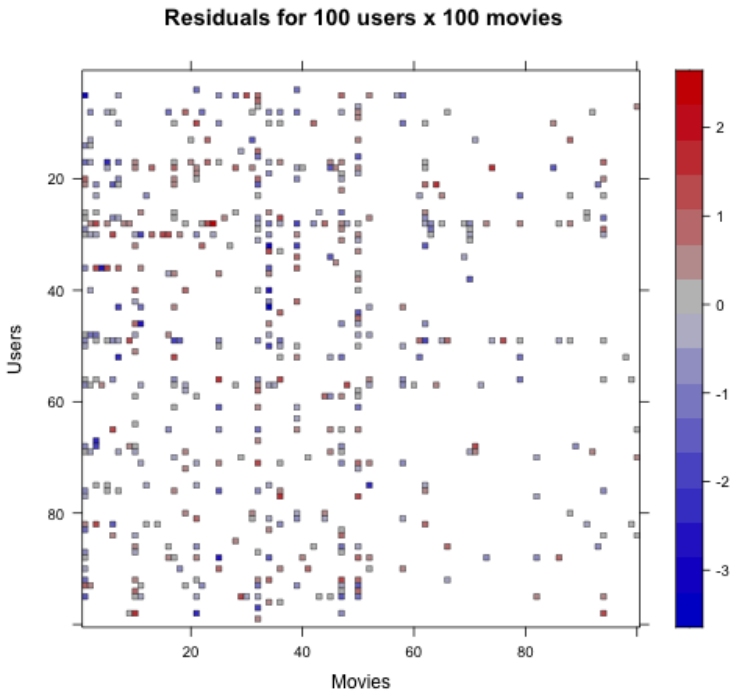
with $r_{ui} = p_u q_i$ the residuals for user u and movie i, given by similarities in user (p_u) and movie (q_i) ratings

We use **RecommenderLab's** implementation of the SVD approximation to decompose de original rating matrix using factorization, so that we can find the closest neighbors by similarities in ratings.

We start by adjusting ratings and removing all effects discussed in previous sections. The resulting "residuals" are transformed into a RealRatingMatrix, as required by Recommenderlab.

```
# calculate the residuals for known ratings
residual <- edx %>%
                mutate(date = as_datetime(timestamp),week=round_date(date, "week")) %>%
                left_join(reg_bis, by="movieId") %>%
                left_join(reg_bus, by= "userId") %>%
                left_join(guis, by="genres") %>%
                left_join(duis, by="week")
                summarize(res= rating - (mu + reg_bi + reg_bu + gui + dui)) %>%
                pull(res)

 # transform into sparse matrix which is more efficient storing sparse data
edx_matrix <- sparseMatrix(i = edx$userId, j = edx$movieId,  x = residual)

# convert to realRatingMatrix as required by Recommenderlab
edx_realMatrix <- as(edx_matrix, "realRatingMatrix")
```

Here are the residuals for the first 100 users and movies. Note the sparsity (missing ratings) of the matrix



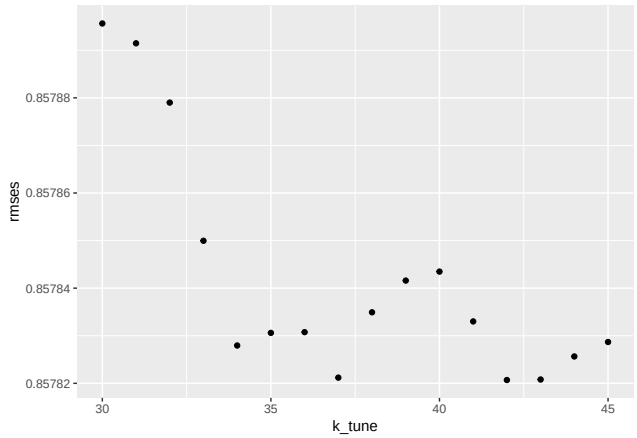
To improve the performance of Recommenderlab, we train our algorithm with users who have rated at least 50 movies, under the assumption that "frequent users" give more robust ratings that lead to better predictions.

Next we use Recommenderlab built-in function to partition the data into a train, test and error set.

```
# train Recommenderlab with "frequent" users who have rated at least 50 movies
frequent_users <- edx_realMatrix[rowCounts(edx_realMatrix) >= 50 ]

# partition data into train, test and error sets, and withhold 30 ratings for evaluation
set.seed(123, sample.kind="Rounding")
frequent_users_split <- evaluationScheme(frequent_users, method="split", train= 0.9, given=30)
train_set<- getData(frequent_users_split, "train")
test_set <- getData(frequent_users_split, "known")
error_set <- getData(frequent_users_split, "unknown")
```

Finally, we train Recommenderlab's SVD algorithm by tuning for the k-neighbors that minimize Recommenderlab built-in RMSE function:

```
k_tune <- seq(30,45, 1)

rmses <- sapply(k_tune, function(k){
  model_svd <- Recommender(train_set, method="SVD", param=list(k=k))
  predictions <- predict(model_svd, test_set, type="ratings" )
  RMSE<- calcPredictionAccuracy(predictions, error_set)["RMSE"]
  return(RMSE)
})
```



The plot above shows that the k that minimizes RMSE is 42, and the RMSE of the model is 0.85782

4 Results

In this section we perform the final test on the validation set for the "Effects" model and the "Recommenderlab" approach. Then we discuss the modeling results and overall performance.

4.1 Final test on the Validation set

4.1.1 Effects Model

The RMSE returned by testing the predictions of our Effects model on the validation set is 0.86431. This is the value we are submitting for this project since it is below the target of 0.86490.

Here is the snipped of code used for the final test on the validation set:

```
# use the full edx set to recalculate reg_bi, reg_bu, gui and dui.
# make predictions on the validation set with Yui = mu + reg_bi + reg_bu + gui + dui

predictions<- validation %>%
  mutate(date = as_datetime(timestamp), week=round_date(date, "week")) %>%
  left_join(reg_bis, by="movieId") %>%
  left_join(reg_bus, by="userId") %>%
  left_join(guis, by="genres") %>%
  left_join(duis, by="week") %>%
  summarize(pred= mu + reg_bi+ reg_bu + gui + dui ) %>%
  pull(pred)

RMSE<- sqrt(mean((validation$rating - predictions)^2))  # RMSE on validation set
```

4.1.2 Recommenderlab SVD Model

As a stretch goal for this project, we have also developed a model that uses Recommenderlab's SVD algorithm to account for groups of movies and users with similar ratings. Surprisingly, the RMSE returned on the validation set, 0.86649, was considerably above target.

Below is a snipped of the proposed code. In practice, we had to break the users rating matrix into several chunks to optimize for memory allocation and expedite processing (code not shown for simplicity, but available in the R script).

```
# fit the SVD model for "frequent users", with k_opt that minimizes RMSE
frequent_users <- edx_realMatrix[rowCounts(edx_realMatrix) >= 50 ]
model_svd <- Recommender(frequent_users, method="SVD", param=list(k=k_opt) )

# predict unknown residuals for frequent users
preds_realMatrix <- predict(model_svd, frequent_users, type="ratingMatrix")

# transform RealRatingsMatrix into a matrix and then a dataframe.
preds_matrix <- getRatingMatrix(preds_realMatrix)
userId <-as.integer(rownames(preds_matrix))
movieId<- as.integer(colnames(preds_matrix))
residual<- as.vector(preds_matrix)

res_freq_users<- data.frame(userId, movieId, residual)

# as above for "infrequent users" who rated less than 50 movies
infrequent_users <- edx_realMatrix[rowCounts(edx_realMatrix) <50, ]
preds_realMatrix <- predict(model_svd, infrequent_users, type="ratings")

preds_matrix <- getRatingMatrix(preds_realMatrix)
userId<-as.integer(rownames(preds_matrix))
movieId<- as.integer(colnames(preds_matrix))
residual<- as.vector(preds_matrix)

res_infreq_users <- data.frame(userId, movieId, residual)

# Combine residuals from "frequent" and "infrequent" users
res<- rbind (res_freq_users, res_infreq_users)

# make predictions on validation set: Yui = mu + reg_bi + reg_bu + gui + dui + residual
predictions<- validation %>%
    mutate(date = as_datetime(timestamp),week=round_date(date, "week")) %>%
    left_join(reg_bis, by="movieId") %>%
    left_join(reg_bus, by="userId") %>%
    left_join(guis, by="genres") %>%
    left_join(duis, by="week") %>%
    left_join(res, by=c("movieId", "userId")) %>%
    mutate (pred= mu + reg_bi + reg_bu + gui + dui + residual) %>%
    pull(pred)

RMSE<- sqrt(mean((validation$rating - predictions)^2))  # RMSE on validation set
```

4.2 Modeling Results and Overall Performance

The following table presents the results of each iteration in our model, including the final tests on the validation set for the two proposed models.

MODEL	RMSE
Average	1.06005
Movies Effects	0.94296
Movies + Users Effects	0.86468
Reg Movies + Reg Users Effects	0.86414
Reg Movies + Reg Users + Genres Effects	0.86381
Reg Movies + Reg Users + Genres + Time Effects	0.86367
Recommenderlab SVD Model	0.85782
VALIDATION test: Effects model	0.86431

MODEL	RMSE
VALIDATION test: Recommenderlab SVD Model	0.86649

We started with a basic model that predicted the same value calculated as the average of all known ratings, which had a RMSE of 1.06005 on the test set. As we adjusted for movie and user biases, we saw progressive but smaller improvements (-0.11709 and -0.07828, respectively) .

Surprisingly, regularization of small sample sizes for movies and users led to a very minor improvement of the RMSE , down -0.00054. Even less remarkable were the adjustments for genres and time effects: -0.00033 and -0.00014, respectively.

Combined, the overall RMSE for all effects was 0.86367 on the test set and 0.86431 on the validation set, both below the required target of 0.86490. As expected, the two values were very close, suggesting that there was no over-training.

The RMSE we are submitting for this project is 0.86431, as measured on the validation set

As a stretch goal, we have used Recommenderlab's SVD algorithm to model for groups of users and movies with similarities in ratings. We have removed previously identified effects from known ratings before processing the data with Recommenderlab to predict the remaining "residuals".

While the RMSE calculated on the testing set during the development phase was encouraging (0.85782), the final test on the validation set returned 0.86649, well above target.

This regression in performance (+0.00867) was likely the result of having had to reduce the size of the edx matrix so that it could be processed by Recommenderlab, rather than using all available ratings.

A faster PC with more memory and compute power that could handle the entire edx matrix might have led to better results.

5 Conclusion

5.1 Summary

We have built an algorithm that predicts ratings for a movie that a user has not yet seen, using ratings given by other people.

Data exploration has showcased that ratings are not uniform. Instead, they are subjected to strong effects associated with perceptions (some movies are widely perceived as better), user behavior (some users have a tendency to give higher or lower ratings), preferences (such as "action" movies vs. "comedies") and time.

Modeling for effects has proven not only very effective but also very insightful in confirming that movies are not rated at random. It led to a RMSE of 0.86431 on the validation set, below the targeted 0.86490

Surprisingly, our observation of groups of movies and users with similar rating patterns, and hence the assumption that users that agree on the ratings for some movies might also agree on their evaluation for others, has not led to better predictions.

We have used **RecommenderLab**'s implementation of the **SVD approximation** to find the closest neighbors by similarities in ratings, after adjusting for all previously uncovered effects.

Although the performance on the test set calculated by Recommenderlab's built-in RMSE function was promising (RMSE was 0.85782), we have observed a significant deterioration when testing on the validation set (RMSE of 0.86649). Therefore, we did not submit this value as the resulting RMSE for this project. Instead, we are proposing several opportunities for improvement in the next sections.

5.2 Limitations

Modeling for effects has proven to be very effective not only from a performance perspective as measured by RMSE, but also in terms of speed and memory use.

In contrast, Recommenderlab is easy to implement, but it is computationally expensive, and does not seem to work well with very large and sparse matrices.

To make predictions for a group of users, the algorithm needs to process all movies, whether rated or not. Tuning for k-neighbors, which requires the evaluation of each pair of users and movies was very time consuming.

To train the algorithm on a regular computer with 16 GB memory, we had to reduce the size of the rating matrix before it could be processed by Recommenderlab.

Rather than selecting observations at random (which preserves the distribution of the original population), we chose to focus on frequent users (those that have rated at least 50 movies) under the assumption that they give more robust ratings that lead to better predictions.

But this left us with a set of infrequent users, for which ratings were more sparse, and therefore the performance of Recommenderlab was far worse.

Later, in order to obtain the final predictions on the validation set, we had to further break the matrices of frequent and infrequent users into several chunks so that they could be processed by Recommenderlab. Again, very inefficient and time consuming.

5.3 Future Work

There are opportunities to improve results by using a more powerful computer that can process the entire edx set using Recommenderlab.

Alternatively, we could use random sampling or tuning Recommenderlab for the cutoff for user frequency (for this project, the limit was preset at users with at least 50 ratings) or a combination or both.

Further, we could improve scalability by reducing dimensions using Pricipal Component Analysis (PCA).

In addition to the SVD implementation that was used here, there are opportunities to compare the performance of other algorithms in Recommenderlab such as user-based collaborative filtering (UBCF), item-based collaborative filtering (IBCF) or Funk-SVD.

Also of interest would be to compare results using other libraries in R for recommender systems such as rrecsys or recosystem (see https://gist.github.com/talegari/77c90db326b4848368287e53b1a18e8d)

6 References

- edx course reference book: https://rafalab.github.io/dsbook/
- Recommenderlab: https://cran.r-project.org/web/packages/recommenderlab/recommenderlab.pdf
- Recommenderlab vignettes: https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf
- Libraries for recommender systems: https://gist.github.com/talegari/77c90db326b4848368287e53b1a18e8d