MovieLens Project Report

Tang Li

Feb 01, 2022

Contents

Note: All the links are colored in violet without underline, feel free to click!

Overview

About this project

This machine learning project is about creating a movie recommendation system, using the MovieLens 10M dataset. The goal is to develop a machine learning algorithm that can predict movie ratings by users, with rmse (root-mean-square-error) as low as possible.

Using recosystem library which supports parallel matrix factorization, my recommendation algorithm is able to predict ratings (from unknown validation set) with a rmse of 0.786.

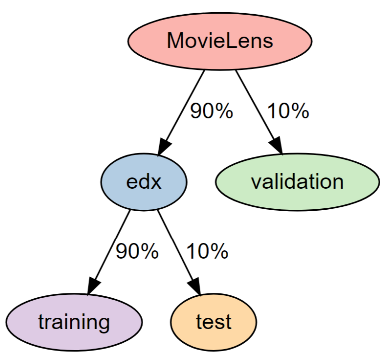
Describe MovieLens dataset

Note: This section is just a brief overview. I'll go into more detail in the Methods section.

From the MovieLens readme site, the MovieLens dataset contains 10000054 ratings of 10681 movies by 71567 users.

Each row of dataset contains:

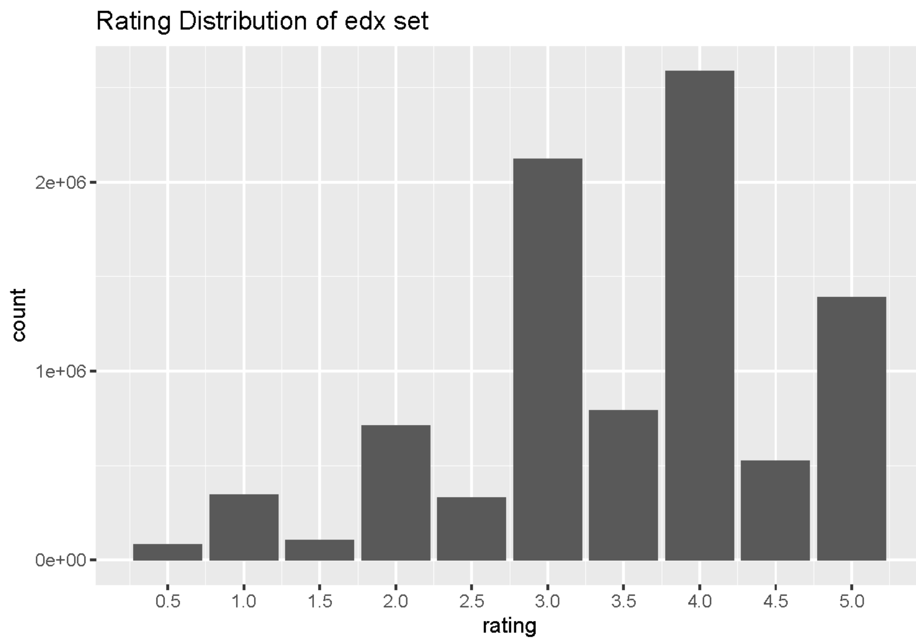
- `rating`: can be half/whole stars ranging from 0.5 to 5
- `userId`: represent a user, unique
- `movieId`: represent a movie, unique
- `title`: title of the movie, with year in parenthesis
- `genres`: genres of the movie in a pipe-separated list, selected from a total of 18 genres
- `timestamp`: time when the user rated the movie



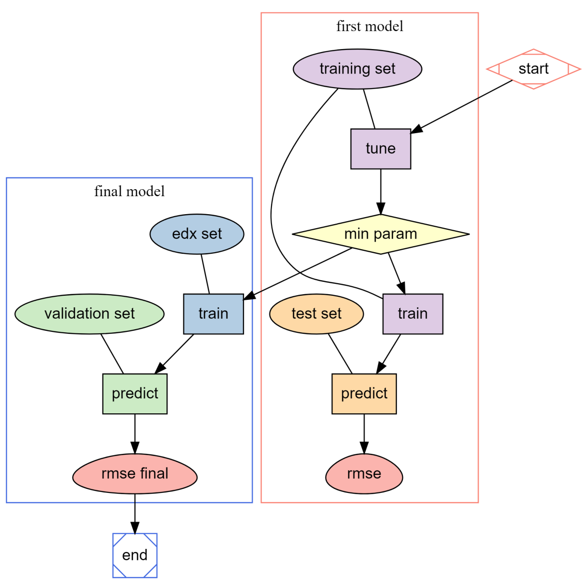
Immediately after downloading the MovieLens dataset, data is split into edx (90%) and validation (10%) sets.

- edx set is used in model development
- validation set is NOT used anywhere except when **testing the final model**, so it's hidden data

Just to give a preview of data visualization in this overview section, here is a histogram that shows the ratings distribution of edx set. We can see that whole star ratings are more common than half star ratings.



Key steps performed



1. download and split MovieLens dataset into edx and validation sets
2. explore and visualize edx set
3. develop the **first** model

- split edx set into training and test sets
- using training set

    - **tune** to find the best tuning opts that achieve min. rmse
    - **train** with best tuning opts

- using test set

    - **predict** ratings
    - **evaluate** rmse

  See Modeling approach section for more explanation.

  In recosystem, a model has 2 parts: best tuning opts and training data. Final model differs with first model by **using edx instead of training set**, because it has the full data.

  First model: training set + best tuning opts

  Final model: edx set + best tuning opts

4. develop the **final** model

- using edx set

    - **train** the final model with best tuning opts

- using validation set

    - **predict** ratings
    - **evaluate** rmse

Methods

Step 1: Prepare dataset

First, let's download and split MovieLens dataset into edx and validation sets. We will download the libraries that are required to split dataset, as well as recosystem library used in the recommendation algorithm. By using different joins, there won't be any user/movie in `validation` that's not in edx.

Note: This download code is provided by the course.

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

library(tidyverse)
library(caret)
library(data.table)
```

```r
library(recosystem)

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

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

movies <- str_split_fixed(
  readLines(unzip(dl, "ml-10M100K/movies.dat")),
  "\\::", 3
)
colnames(movies) <- c("movieId", "title", "genres")

# if using R 3.6 or earlier:
if (as.integer(R.version$major) <= 3 & as.double(R.version$minor) <= 6) {
  movies <- as.data.frame(movies) %>%
    mutate(
      movieId = as.numeric(levels(movieId))[movieId],
      title = as.character(title),
      genres = as.character(genres)
    )
}

# if using R 4.0 or later:
if (as.integer(R.version$major) >= 4) {
  movies <- as.data.frame(movies) %>%
    mutate(
      movieId = as.numeric(movieId),
      title = as.character(title),
      genres = as.character(genres)
    )
}

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

# if using R 3.5 or earlier, use `set.seed(1)`
if (as.integer(R.version$major) <= 3 & as.double(R.version$minor) <= 5) {
  set.seed(1)
} else {
  set.seed(1, sample.kind = "Rounding")
}

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

```r
# Make sure userId and movieId in validation set are also in edx set
validation <- temp %>%
  semi_join(edx, by = "movieId") %>%
  semi_join(edx, by = "userId")

# Add rows removed from validation set back into edx set
removed <- anti_join(temp, validation)
edx <- rbind(edx, removed)

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

Because I will only use `userId`, `movieId`, `rating` columns, we keep these 3 columns and remove the rest. Lastly, we save edx and `validation` dataframes into rda files, and remove `validation` to save memory. The validation set rda file can be loaded back in future when we use it.

```r
edx <- edx %>% select(userId, movieId, rating)
validation <- validation %>% select(userId, movieId, rating)

if (!dir.exists("rdas")) {
  dir.create("rdas")
}
save(edx, file = "rdas/edx.rda")
save(validation, file = "rdas/validation.rda")

rm(validation)
```

Let's view edx by sampling a few rows.

```r
set.seed(1)
edx %>% slice_sample(n = 6)
```

userId	movieId	rating
19193	3019	3.5
26746	6333	4.5
40919	1136	3.0
65123	3310	4.0
14692	3197	2.0
64379	8807	2.5

Step 2: Describe dataset

edx rating

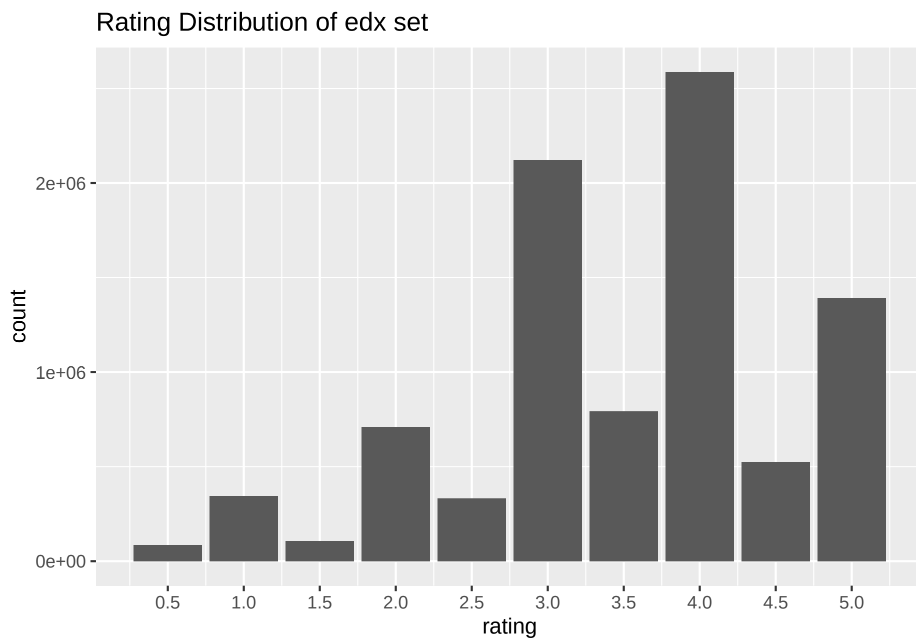
Let's see the 5-number summary (min, Q1, median, Q3, max) of `edx$rating`.

```r
summary(edx$rating)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.50    3.00    4.00    3.51    4.00    5.00
```

We can see that the minimum rating is 0.5, and half of the ratings are between 3 and 4 stars, so overall users are kind in their ratings!

Here is a ratings histogram, which shows that whole star ratings are more common than half star ratings.



Users and movies

There are 69878 users and 10677 movies are in edx set, and the sparsity is 1.21%.

```
n_distinct(edx$userId)
```

```
## [1] 69878
```

```
n_distinct(edx$movieId)
```

```
## [1] 10677
```

```
nrow(edx) / (n_distinct(edx$userId) * n_distinct(edx$movieId))
```

```
## [1] 0.0121
```

Step 3: Develop the first model

Create training and test sets

We can split edx set into `train` set (90%) and `test` set (10%) using the caret package. By using different joins, there won't be any user/movie in `test` that's not in `train`.

```r
set.seed(1)
test_index <- createDataPartition(
  y = edx$rating, times = 1, p = 0.1, list = FALSE
)
train <- edx[-test_index, ]
temp <- edx[test_index, ]

# Make sure userId and movieId in test set are also in training set
test <- temp %>%
  semi_join(train, by = "movieId") %>%
  semi_join(train, by = "userId")

# Add rows removed from test set back into training set
removed <- anti_join(temp, test)
train <- rbind(train, removed)

rm(edx, test_index, temp, removed)
```

Modeling approach

What is a model? To create a model means developing an algorithm by analyzing a known dataset. Machine learning is about using this model so that it can predict useful information of a new/unknown dataset.

Tune: generate the best recosystem algorithm Previously in Key Steps we described that a model in recosystem has 2 parts: training data + best tuning opts. Training data is the known dataset, and best tuning opts is the algorithm part.

Although we don't know how recosystem does matrix factorization under the hood, we can **create different recosystem algorithms by putting in different tuning parameters**. This process is the tuning step, and the goal is to find best tuning opts: the optimal param and option values that result in min rmse, and thus generate the **best recosystem algorithm** *for our data*.

> The emphasis is on "for our data" because different tuning opts work best for different data. What we consider the best recosystem algorithm for movie recommendation is unlikely to work for song recommendation.

Train: create the model Once we are satisfied with the rmse resulted by best tuning opts, we execute the algorithm on training data to create a model. The model will contain matrices necessary for the predicting step.

Predict: use the model to predict the unknown The predict step uses the matrices in the model to predict ratings in the unknown set. After getting the predicted ratings list, we can calculate the rmse by comparing with the list of true ratings.

recosystem functions Tune, train, and predict functions are stored in the model object, using and updating information in the model. Each step is further explained in the next sections.

create a new model	`model <- Reco()`
tune	`model$tune()`
train	`model$train()`
predict	`model$predict()`

Create the first model

Using `Reco()` function, we create a new model object.

```
first_model <- Reco()
```

But we are not yet done the setup because there is a data conversion step.

To put our data `train` and `test` in recosystem, they need to be **converted from dataframes to DataSource** objects. We can use `data_memory` function to convert. The DataSource objects will contain same data as the original dataframes.

Let's add the **`_data`** suffix to DataSource variable names to avoid mix up. Here's a table that shows the variable names, and the syntax to access user, movie, and rating columns.

Name	Type	Access user	Access movie	Access rating
train	dataframe	$userId	$movieId	$rating
train_data	DataSource	@source[1]	@source[2]	@source[3]
test	dataframe	$userId	$movieId	$rating
test_data	DataSource	@source[1]	@source[2]	@source[3]

Note: The `index1` argument below in `data_memory` function asks whether the ids start with 0 or 1. They start with 1, so we set `index1 = TRUE`.

```
min(train$userId)
```

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

```
min(train$movieId)
```

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

```
train_data <- data_memory(
  user_index = train$userId,
  item_index = train$movieId,
  rating = train$rating,
  index1 = TRUE
)

test_data <- data_memory(
  user_index = test$userId,
  item_index = test$movieId,
  rating = test$rating,
```

```
  index1 = TRUE
)

rm(train, test)
```

Tuning

The `$tune()` function uses k-fold cross validation to tune the model's parameters.

input

- `train_data`: training data
- `opts`: parameters and options
  - params are tuned, tune will choose the best value for each param
  - options are NOT tuned, think of these as settings

output

- `min` (**best tuning opts** mentioned earlier): the optimal combination with each param assigned a value, that result in **minimum rmse**
- `res`: each combination and its rmse

Default tuning opts From the help file by typing `?tune` in console, we find recosystem default tuning opts. Default is used when calling `model$tune()` with no `opts` argument. The code chunk below shows what default looks like, and stores the tuning opts in a variable.

The tune function will look through each combination and pick the best one with min rmse, out of $2^6 = 64$ combinations (6 params, 2 choices each).

> For example, the first combination is: `dim = 10, costp_l1 = 0, costp_l2 = 0.01, costq_l1 = 0, costq_l2 = 0.01, lrate = 0.01`.

```
best_tune_opts <- list(
  # params
  dim = c(10, 20),
  costp_l1 = c(0, 0.1),
  costp_l2 = c(0.01, 0.1),
  costq_l1 = c(0, 0.1),
  costq_l2 = c(0.01, 0.1),
  lrate = c(0.01, 0.1),
  # options
  loss = "l2",
  nfold = 5,
  niter = 20,
  nthread = 1,
  nbin = 20,
  nmf = FALSE,
  verbose = FALSE,
  progress = TRUE
)
```

Here's a table about what each parameter and option means.

> P in `costp` means user, Q in `costq` means movies in this case.
> l2 is related to RMSE - the squared error, l1 is related to MAE - mean absolute error.

Parameters	What it is	Options	What it is
dim	number of factors for matrix factorization	nfold	number of folds in cross validation
costp_l1	L1 user factors regularization cost	niter	number of iterations
costp_l2	L2 user factors regularization cost	nthread	number of threads for parallel computing
costq_l1	L1 movie factors regularization cost	nbin	number of bins: must be $>$ nthread
costq_l2	L2 movie factors regularization cost	nmf	whether to perform non-negative matrix factorization
lrate	learning rate: step size in gradient descent	verbose	whether to print info in console
loss	error function: loss = l2 for rmse	progress	whether to print progress bar in console

Change the default opts The default provides a nice starting point, but we want to customize the params and options before we call tune function for the first time. Yes, we will call tune function 2 times! The reasons are below each code chunk.

```
best_tune_opts$nthread <- 16
best_tune_opts$nbin <- 128
best_tune_opts$nmf <- TRUE
best_tune_opts$verbose <- TRUE
```

Options

Option	Old value	New value	Reason for change
nthread	1	16	Take advantage of recosystem's parallel computation and speed up the process.
nbin	20	128	What nbin exactly means is unclear in recosystem's help file, but since nbin must be $>$ nthread, I just made them both bigger.
nmf	FALSE	TRUE	All our ratings are non-negative, we don't want any predicted ratings to be <0.
verbose	FALSE	TRUE	Outputs progress in RStudio console (not in report pdf, there would be many pages!)

Params

The 2 tuning performance concerns are efficiency and quality.

Tuning needs to be efficient because its intense computation can take a long time, so we want to **run as few combinations as possible**. The changed set only has $2^4 = 16$ combinations compared to the default of 64 combinations. We choose to only test 2 values for each param (picking 3 values would result in $3^4 = 81$ combinations). We should also try to improve the quality of each tuning, so we **shouldn't waste on unlikely/bad guesses** with only 2 values to test for each param.

Moreover, the goal of tuning is try to find/approach the global minimum of rmse in a few rounds of tuning, but it's easy to get stuck inside a local minimum. We can avoid this by starting out big **(test values that are farther apart) in the first call** of tuning, then we narrow down, test the best value's close neighbors in later rounds. More on this later!

> For example, the first combination is: `dim = 20, lrate = 0.1, costp_l1 = 0, costp_l2 = 0, costq_l2 = 0, costq_l2 = 0`

```r
best_tune_opts$dim <- c(20, 40)
best_tune_opts$lrate <- c(0.1, 0.2)
best_tune_opts$costp_l1 <- 0
best_tune_opts$costp_l2 <- c(0, 0.2)
best_tune_opts$costq_l1 <- 0
best_tune_opts$costq_l2 <- c(0, 0.2)
best_tune_opts
```

```
## $dim
## [1] 20 40
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0.0 0.2
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.0 0.2
##
## $lrate
## [1] 0.1 0.2
##
## $loss
## [1] "l2"
##
## $nfold
## [1] 5
##
## $niter
## [1] 20
##
## $nthread
```

```
## [1] 16
##
## $nbin
## [1] 128
##
## $nmf
## [1] TRUE
##
## $verbose
## [1] TRUE
##
## $progress
## [1] TRUE
```

Param	Old value	New value	Reason for change
dim	c(10, 20)	c(20, 40)	I believe that the more dimensions in matrix factorization, the better the result would be. Since movies have 18 movie genres, my guess is that we need at least 20 dimensions.
lrate	c(0.01, 0.1)	c(0.1, 0.2)	A learning rate step size of 0.01 is not a good guess because it's almost 0 (no gradient descent), test 0.2 instead
costp_l1, costq_l1	c(0, 0.1)	0	l1 is related to mean absolute error, not rmse, so won't tune these
costp_l2	c(0.01, 0.1)	c(0, 0.2)	l2 is related to rmse, and we test a bigger range

Tuning round 1 Now let's tune, and store the tune result in a variable.

```
set.seed(1)
tune_output <- first_model$tune(train_data, opts = best_tune_opts)
```

`min` shows the combination that yields the minimum rmse, and `res` shows rmse of all combinations. The minimum rmse is 0.801, so looks pretty good on the first try!

```
tune_output$min
```

```
## $dim
## [1] 40
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0
##
## $costq_l1
## [1] 0
```

```
##
## $costq_l2
## [1] 0.2
##
## $lrate
## [1] 0.1
##
## $loss_fun
## [1] 0.802
```

```
tune_output$res
```

dim	costp_l1	costp_l2	costq_l1	costq_l2	lrate	loss_fun
20	0	0.0	0	0.0	0.1	0.831
40	0	0.0	0	0.0	0.1	0.851
20	0	0.2	0	0.0	0.1	0.825
40	0	0.2	0	0.0	0.1	0.832
20	0	0.0	0	0.2	0.1	0.811
40	0	0.0	0	0.2	0.1	0.802
20	0	0.2	0	0.2	0.1	0.878
40	0	0.2	0	0.2	0.1	0.878
20	0	0.0	0	0.0	0.2	0.839
40	0	0.0	0	0.0	0.2	0.873
20	0	0.2	0	0.0	0.2	0.836
40	0	0.2	0	0.0	0.2	0.846
20	0	0.0	0	0.2	0.2	0.810
40	0	0.0	0	0.2	0.2	0.807
20	0	0.2	0	0.2	0.2	0.881
40	0	0.2	0	0.2	0.2	0.883

Analysis We can visualize the full result better with 2 raster plots.

dim-lrate plot: Using tidyverse library, we can `group_by` dim and lrate (4 combinations), but because there are 4 rmses for each combination (since l2 factors are not fixed), we take the minimum rmse in `summarize`. The code chunk below shows the dataframe which can then be piped to ggplot.

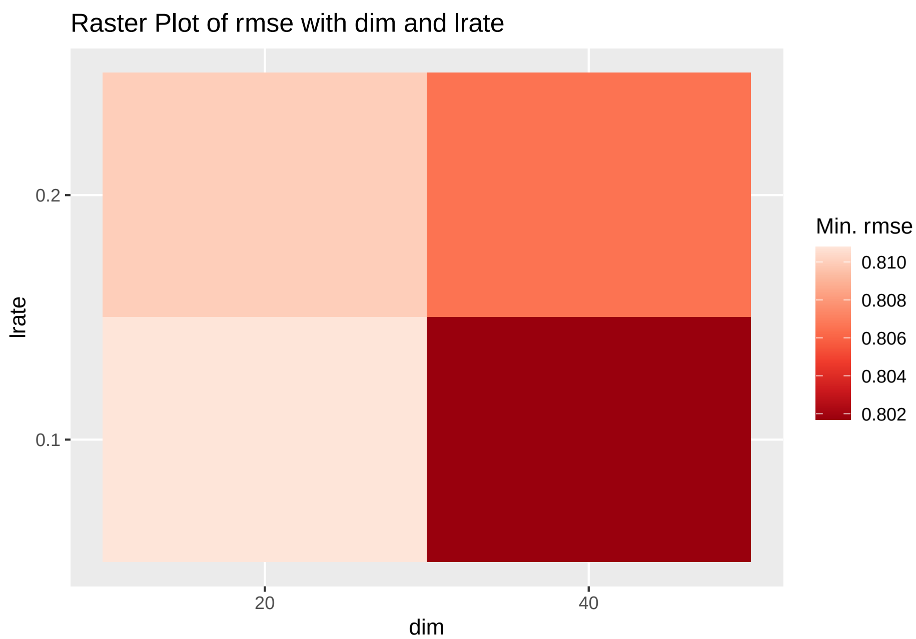
l2 factors plot: similar to first plot, `group_by` by costp_l2 and costq_l2, and `summarize` min rmse

```
tune_output$res %>%
  group_by(dim = factor(dim), lrate = factor(lrate)) %>%
  summarize(rmse = min(loss_fun))
```

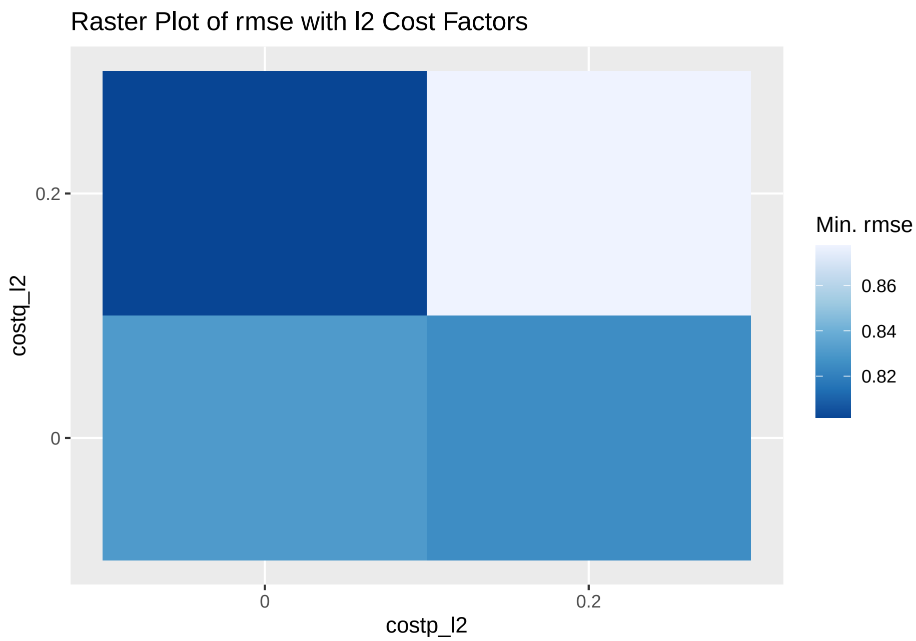
dim	lrate	rmse
20	0.1	0.811
20	0.2	0.810
40	0.1	0.802
40	0.2	0.807

```
tune_output$res %>%
  group_by(costp_l2 = factor(costp_l2), costq_l2 = factor(costq_l2)) %>%
  summarize(rmse = min(loss_fun))
```

costp_l2	costq_l2	rmse
0	0	0.831
0	0.2	0.802
0.2	0	0.825
0.2	0.2	0.878



Looking at the color bar legend in the dim-lrate raster plot, rmses roughly ranges from 0.802 to 0.81, and don't vary much from each other. This suggests that we further tuning dim and lrate won't be much of an improvement.



However, the color bar legend in the l2 cost factors plot shows that rmses roughly ranges from 0.8 to 0.88, which is a big difference. Notably it's `costq_l2` that significantly changes the rmses. Looking at the 2 tiles in the first column `costp_l1 = 0`, changing `costq_l2` from 0 to 0.2 reduces rmse by about 0.03. We will further tune `costq_l2` (the movie regularization cost factors) in the next round.

Before starting tuning round 2, let's make sure our tuning list variable is synced up with this round's min result.

```
best_tune_opts$dim <- tune_output$min$dim
best_tune_opts$lrate <- tune_output$min$lrate
best_tune_opts$costp_l2 <- tune_output$min$costp_l2
best_tune_opts$costq_l2 <- tune_output$min$costq_l2
```

Tuning round 2 In tuning round 1's analysis section, we chose to further tune `costq_l2`, the movie cost factors. We can verify whether `costq_l2 = 0.2` result in the true minimum of rmse, or if there is a better neighbor of 0.2 that results in an even lower rmse.

We choose the neighborhood of $[0.05, 0.3]$ with a step size of 0.05. The expanded list is 0.05, 0.1, 0.15, 0.2, 0.25, 0.3. This time we can afford to test 6 values, because we are only tuning 1 param, so there are only 6 combinations! The code chunk below updates `costq_l2` and displays our tuning list variable.

```
best_tune_opts$costq_l2 <- seq(0.05, 0.3, 0.05)
```

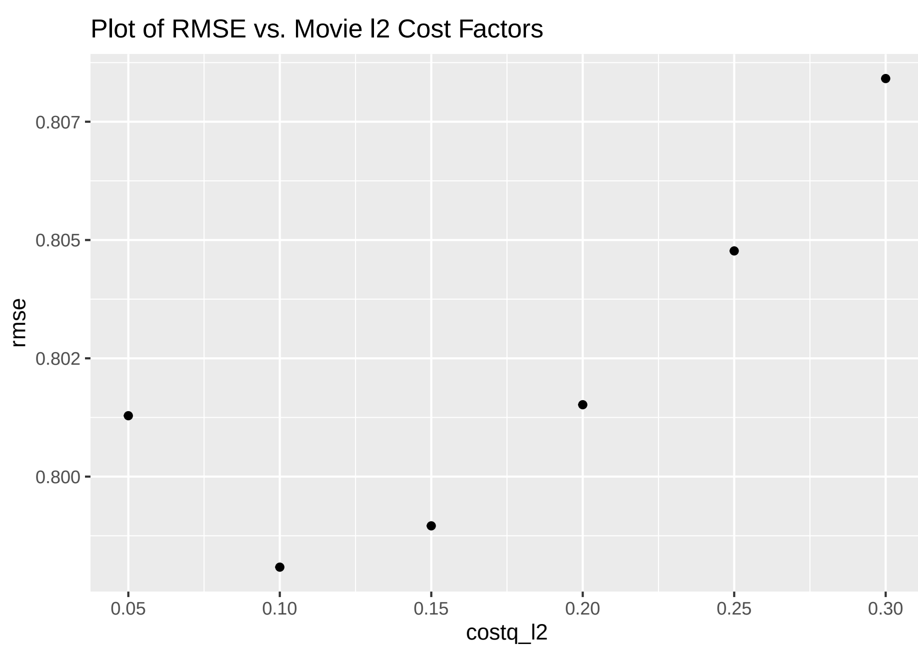
Now let's tune, and see the result!

```
set.seed(1)
tune_output <- first_model$tune(train_data, opts = best_tune_opts)
```

```
tune_output$min
```

```
## $dim
## [1] 40
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.1
##
## $lrate
## [1] 0.1
##
## $loss_fun
## [1] 0.798
```

Analysis The min rmse is 0.797, with `costq_l2 = 0.1`. We can plot the full result using `geom_point`.



We won't tune costq_l2 again, because there isn't much room for improvement. Here the lowest rmse is only 0.01 lower than highest rmse. Update and print the best tuning opts variable, and the tuning stage is now complete!

```
best_tune_opts$costq_l2 <- tune_output$min$costq_l2
best_tune_opts
```

```
## $dim
## [1] 40
##
## $costp_l1
## [1] 0
##
## $costp_l2
## [1] 0
##
## $costq_l1
## [1] 0
##
## $costq_l2
## [1] 0.1
##
## $lrate
## [1] 0.1
##
## $loss
## [1] "l2"
##
## $nfold
## [1] 5
##
## $niter
## [1] 20
##
## $nthread
## [1] 16
##
## $nbin
## [1] 128
##
## $nmf
## [1] TRUE
##
## $verbose
## [1] TRUE
##
## $progress
## [1] TRUE
```

```
rm(tune_output)
```

Train

The `$train()` function will read from training data, and create a model that contains matrices necessary for prediction later. We'll be using `train_data` as our training data.

input

- `train_data`: training data
- `out_model`: where the model is stored, set NULL for storing in memory
- `opts`: tuning params and options

output

- There is no return value, but `$model` will be populated.

The code chunk output below shows that `tr_rmse` (training rmse) gradually decreases over `niter = 20` training iterations.

```
set.seed(1)
first_model$train(train_data, opts = best_tune_opts)
```

```
## iter      tr_rmse          obj
##    0       0.9936   1.0918e+07
##    1       0.8813   8.5772e+06
##    2       0.8535   7.8576e+06
##    3       0.8330   7.3705e+06
##    4       0.8164   7.0081e+06
##    5       0.8033   6.7288e+06
##    6       0.7925   6.5092e+06
##    7       0.7834   6.3320e+06
##    8       0.7755   6.1793e+06
##    9       0.7686   6.0496e+06
##   10       0.7625   5.9340e+06
##   11       0.7569   5.8328e+06
##   12       0.7519   5.7453e+06
##   13       0.7473   5.6642e+06
##   14       0.7432   5.5917e+06
##   15       0.7394   5.5247e+06
##   16       0.7358   5.4629e+06
##   17       0.7326   5.4090e+06
##   18       0.7295   5.3552e+06
##   19       0.7267   5.3071e+06
```

Analysis How does the model generates and improves this `tr_rmse`, and why this rmse is so much better than the 0.797 we got in tuning? The model probably uses cross validation and gradient descent, though we don't really know what recosystem does in its source code.

If the more iterations we train, the lower the `tr_rmse`, why not try a lot more iterations like 100 iterations? However, this is NOT a good idea because **we don't want to over-train** the model with training data, because eventually the model will be evaluated against the unknown test data.

Predict

The `$predict()` function predicts unknown ratings in the testing data. We'll be using `test_data` as our testing data.

input

- `test_data`: testing data
- `out_pred`: where to store the predicted ratings, set `out_memory()` to store in memory

output

- a list of predicted ratings

  Note: The true ratings stored in `test_data` will be ignored by the predict function. Only user and movie ids are used. From help file by typing `?predict`:

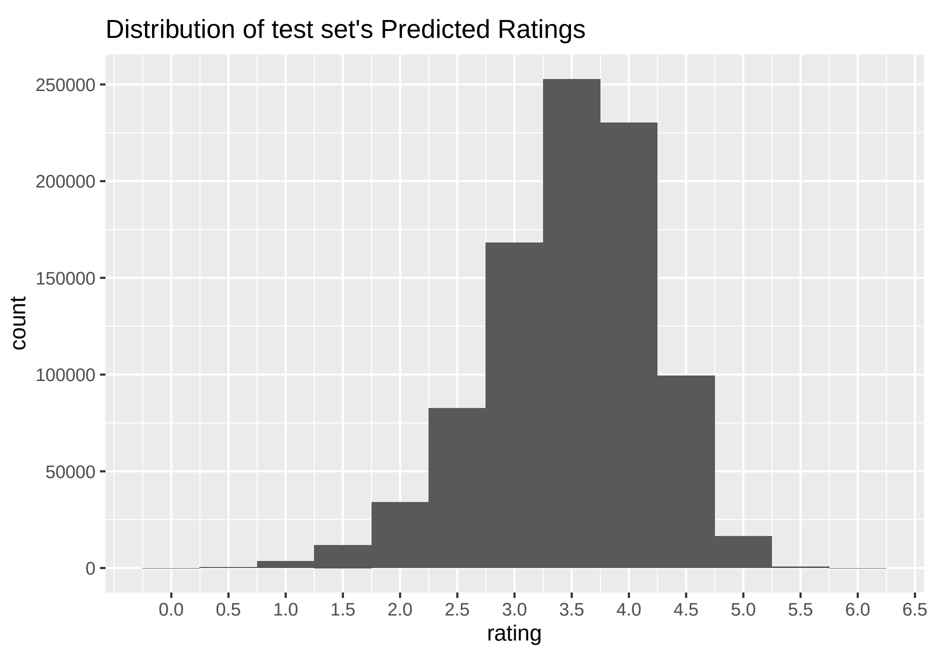
  In `$predict()` … the testing data have the same format as training data, except that the value (rating) column is not required, and **will be ignored if it is provided**.

Let's predict the ratings and see the 5-number summary.

```
set.seed(1)
pred_ratings <- first_model$predict(test_data, out_memory())

summary(pred_ratings)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.19    3.06    3.56    3.49    3.99    6.21
```

The predicted ratings range falls outside of the star range $[0.5, 5]$. Plotting a histogram, we can see that it's a left-skewed bell curve. Unlike edx set distribution we've seen in the data exploration section, where there are a lot more whole star ratings, here our predicted ratings **don't favor whole stars ratings over half stars**.

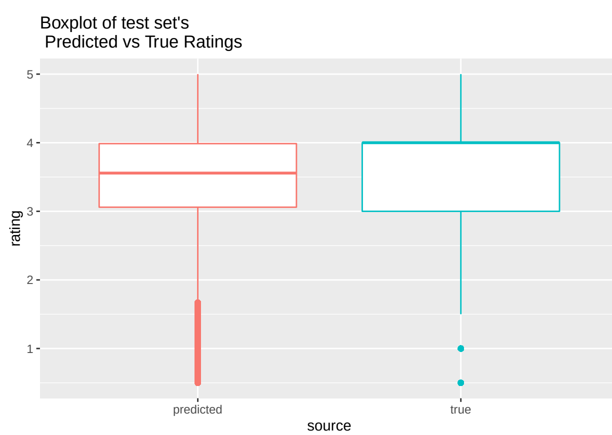


Using the `bound_rating()` function, we can bound the predicted ratings inside the $[0.5, 5]$ by setting any rating $<$ 0.5 to 0.5, and any rating $>$ 5 to 5. As we've seen in the data conversion section, test set's true ratings are stored as `test_data@source[[3]]`. We call `evaluate_rmse()` with true and predicted ratings. The evaluated rmse is 0.789.

```
bound_rating <- function(ratings) {
  sapply(ratings, function(r) {
    if (r > 5) return(5)
    if (r < 0.5) return(0.5)
    return(r)
  })
}

evaluate_rmse <- function(true, pred) {
  sqrt(
    mean((true - pred)^2)
  )
}

pred_ratings <- bound_rating(pred_ratings)

rmse_test <- evaluate_rmse(test_data@source[[3]], pred_ratings)
rmse_test
```

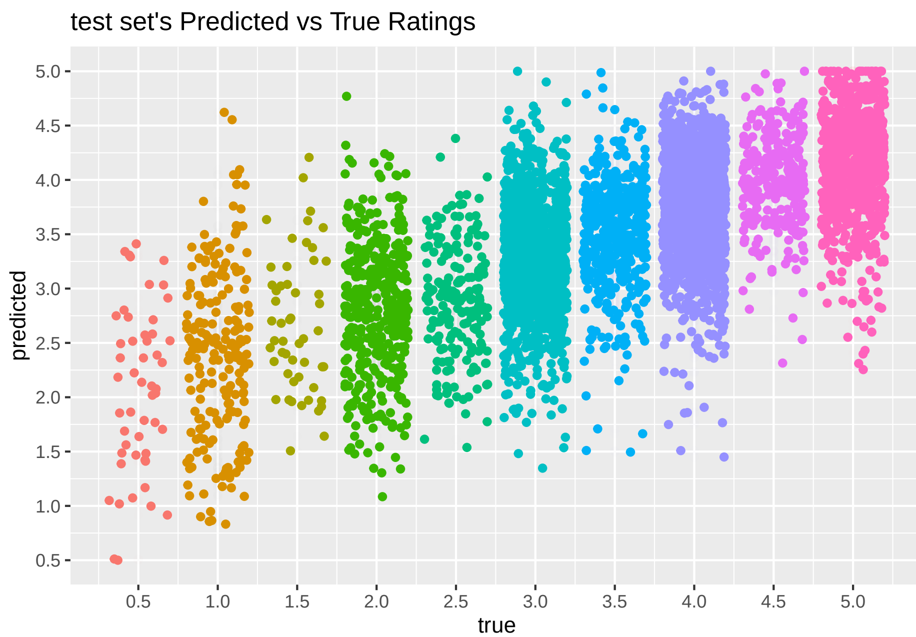
```
## [1] 0.789
```

Analysis Besides rmse evaluation, there is much more information for us to discover. For instance, we can generate a boxplot comparing the predicted and true ratings. The model predicted ratings with nearly the same IQR (interquartile range/middle half) as the true ratings. **However, the model is a little cautious on predicting higher ratings.** The predicted median about 3.5 is lower than the true median of 4.



We can use a jitter point plot to see for each true star rating how far off our predicted ratings are.

> The plot is generated by sampling 5000 points (for the code to finish on time). The alpha is 0.005, so if readers see one solid colored dot, it's the result of an overlap of hundreds of points.



Predicted ratings are continuous, but true ratings are whole/half stars. Could we then round the continuous ratings to star ratings? Similar to the standard decimal rounding rules, $<$ 0.25 rounds to 0, $[0.25, 0.74)$ rounds to 0.5, and $>$ 0.75 rounds to 1.

If we choose to round ratings to whole/half stars, then:

- Rounding one predicted rating **farther** from the true rating contributes to a **larger error** compared with no rounding. For example, true rating = 3.5, predicted rating = 3.2, rounded rating = 3
- Rounding one predicted rating **closer** to the true rating contributes to a **smaller error** compared with no rounding. For example, true rating = 4, predicted rating = 3.8, rounded rating = 4

Since **rmse penalizes larger errors much more than smaller errors**, making 1 bad rounding of large error might need many good rounding of small error. Rounding is beneficial when many points are concentrated close around true rating, and only few points are far away. We are NOT seeing this shape in our jitter plot since many points are far away, so we won't round.

Finally we clean up the variables and prepare for step 4, the final model.

```
rm(train_data, test_data, pred_ratings, first_model, rmse_test)
```

Step 4: Develop the final model

Finally we reached to the most exciting step, combining previous tuning list with edx set and form the final model! I won't put much description here since the steps are almost the same as when we created the first model.

Create final model

We convert `edx` and `validation` dataframes to recosystem DataSource objects `edx_data` and `validation_data`. We create a new model object named `final_model`.

```r
load("rdas/edx.rda")
load("rdas/validation.rda")

edx_data <- data_memory(
  user_index = edx$userId,
  item_index = edx$movieId,
  rating = edx$rating,
  index1 = TRUE
)

validation_data <- data_memory(
  user_index = validation$userId,
  item_index = validation$movieId,
  rating = validation$rating,
  index1 = TRUE
)

rm(edx, validation)

final_model <- Reco()
```

Train

We skip the tuning stage for the final model, because we are satisfied with our first model tuning opts, which forms the algorithm. We are using the same algorithm, but on the full edx set data.

The training stage generates the matrices necessary for the predicting stage.

```r
set.seed(1)
final_model$train(edx_data, opts = best_tune_opts)
```

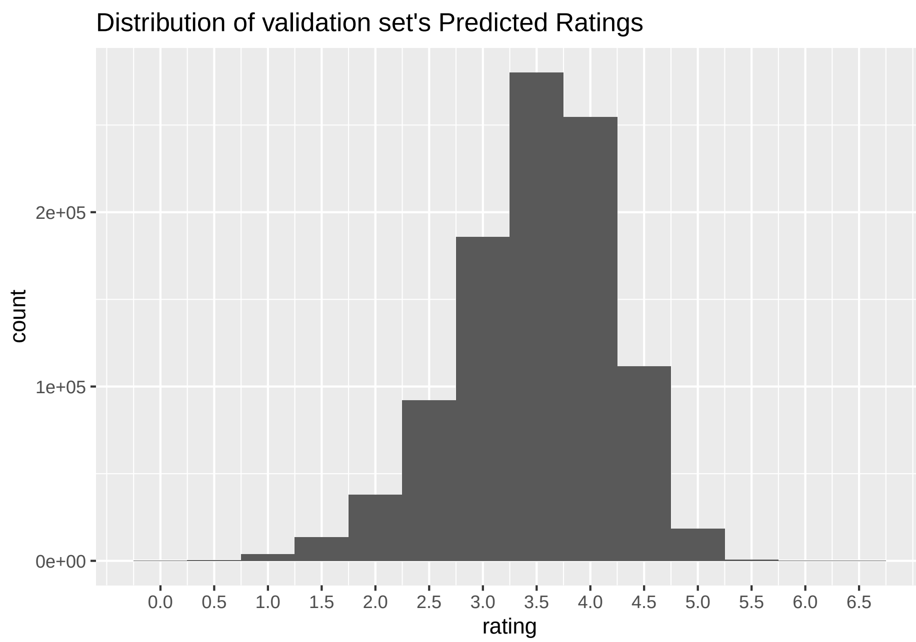
```
## iter      tr_rmse          obj
##    0       0.9830   1.1857e+07
##    1       0.8786   9.4062e+06
##    2       0.8497   8.5970e+06
##    3       0.8294   8.0712e+06
##    4       0.8133   7.6779e+06
##    5       0.8005   7.3791e+06
##    6       0.7901   7.1428e+06
##    7       0.7813   6.9510e+06
##    8       0.7739   6.7911e+06
##    9       0.7674   6.6535e+06
##   10       0.7616   6.5334e+06
##   11       0.7565   6.4275e+06
##   12       0.7518   6.3352e+06
##   13       0.7476   6.2509e+06
##   14       0.7437   6.1747e+06
##   15       0.7402   6.1047e+06
##   16       0.7369   6.0410e+06
##   17       0.7338   5.9824e+06
##   18       0.7310   5.9282e+06
##   19       0.7284   5.8779e+06
```

Predict

The predicting stage only takes the user and movie ids (ignoring ratings argument) of validation set and outputs predicted ratings. We can see the 5-number summary, and the histogram of our predicted ratings.

```
set.seed(1)
pred_ratings <- final_model$predict(validation_data, out_memory())

summary(pred_ratings)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.02    3.06    3.56    3.49    3.99    6.30
```



Since we know that ratings are in the range $[0.5, 5]$, any rating that falls outside this range can be bounded inside: set ratings $<$ 0.5 to 0.5, and ratings $>$ 5 to 5.

```
pred_ratings <- bound_rating(pred_ratings)
```

Results

rmse

Let's evaluate the rmse for the validation set - final hold out test set. The rmse is 0.786.

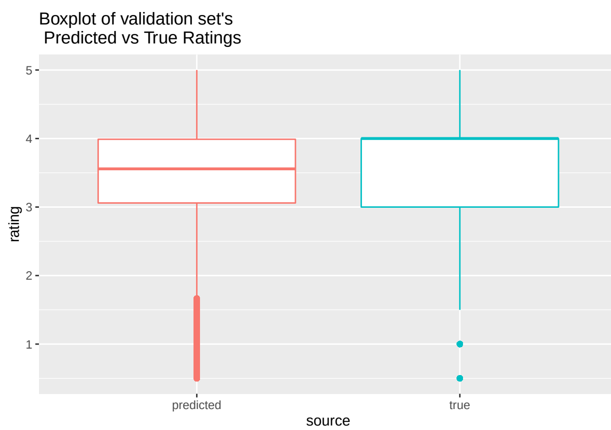
```
rmse_validation <- evaluate_rmse(
  validation_data@source[[3]], pred_ratings
)
rmse_validation
```

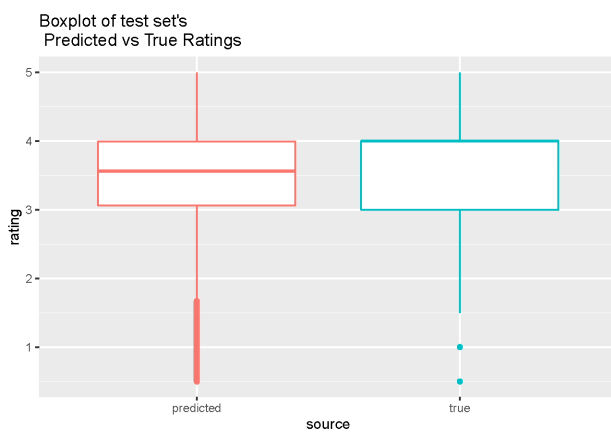
```
## [1] 0.786
```

Overall, the final model performs really well, as expected. As we learned in the Modeling approach section, getting the best tuning opts is like getting the best recosystem algorithm. In the first model, we generated best algorithm using training set, and got a rmse of 0.789 when tested on test set. Using the same algorithm on the final model, we got a rmse of 0.786 on validation set, almost the same rmse as before. This best tuning opts / our algorithm is robust.

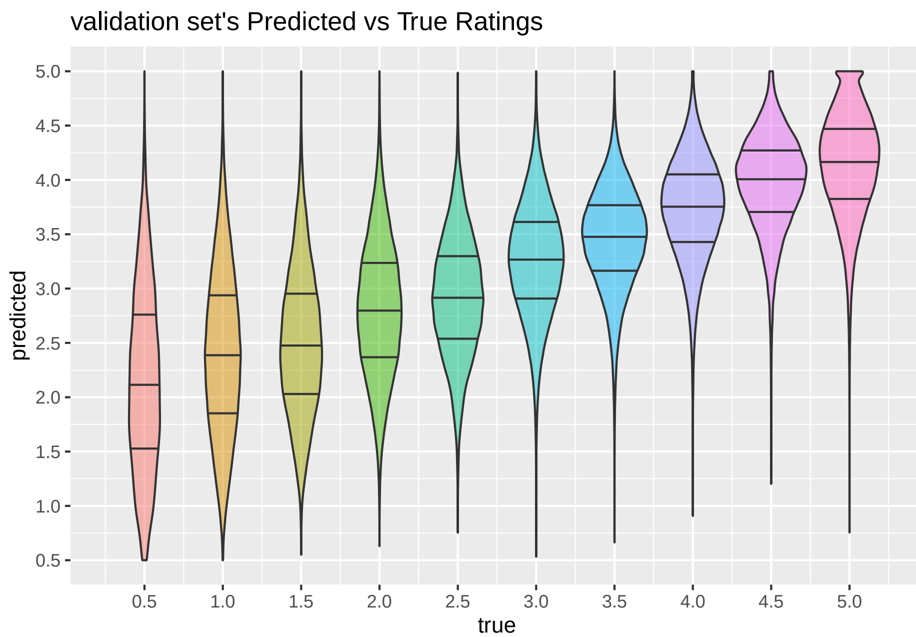
Plots

We can generate a boxplot that compares predicted vs true ratings of validation set. We can also pull up test set's boxplot generated in step 3 for comparison. The shapes look exactly the same due to using the same algorithm, but we can distinguish the plots by looking at the titles.





We can further explore the relationship of predicted vs true ratings by generating a violin plot using `geom_violin`. A violin plot is combining a boxplot and a density plot. Similar to a boxplot, it has 25%, 50%, and 75% quantile bars. However, it has **density curves** on the side, hence the "violin" shape. Interestingly, looking at the plot below, the density curves form "leaf" shapes instead of violins.



How well did our final model predict? Let's break it down.

Prediction was	Where is true rating?	Star categories
Too high	bottom 25% (min to Q1)	0.5, 1, 1.5, 2, 2.5
Good	middle half (Q1 to Q3)	3, 3.5, 4
Too low	top 25% (Q3 to max)	4.5, 5

For movies with low star ratings (from 0.5 to 2.5), the model predicted ratings to be higher than actual. In contrast, movies with high star ratings (4.5 and 5), the model predicted lower ratings than actual. The model is cautious on giving many really high or low ratings.

The model predicted well for movies with moderate ratings between 3 to 4. Fortunately, ratings between 3 and 4 constitute half of our data (see previous boxplot of "predicted vs true ratings of validation set"). So the final model aced half of the data.

Finally, let's print final rmse for one last time. Then we clean up all the variables and datasets folders (rdas and MovieLens).

```
rmse_validation
```

```
## [1] 0.786
```

```
rm(best_tune_opts, edx_data, validation_data, final_model, pred_ratings,
   bound_rating, evaluate_rmse, rmse_validation)
unlink(c("rdas", "ml-10M100K"), recursive = TRUE)
```

Conclusion

Summary

The context of this machine learning project is to create an algorithm for a movie recommendation system, using the MovieLens dataset. Specifically, this report presents an algorithm that can effectively predict movie ratings by users. The goal is to achieve a rmse as low as possible, and our final resulting rmse is 0.786.

Using recosystem library which supports parallel matrix factorization, the main focus of this report is model development and its 3 stages: tune, train, and predict. The recosystem library is flexible because we don't need to get stuck with the generic recosystem algorithm - it allows us to tune the opts (params and options) and customize a best recosystem algorithm for our data.

- In Step 1: Prepare dataset, we split MovieLens dataset into edx (known) and validation (unknown) sets.
- In Step 2: Describe dataset, we discovered that despite the huge number of ratings, edx set is in fact very sparse.
- In Step 3: Develop the first model, we further split edx set into training (known) and test (unknown) sets. Tuning with training set, we found the best recosystem algorithm to use in training stage. Then we trained the model and predicted test set's ratings, and evaluated a rmse of 0.789. We were satisfied with this best algorithm, and expected it to perform equally well with validation set.
- In Step 4: Develop the final model, we no longer needed the tuning stage, and used the same algorithm to train the final model. Finally we predicted validation set's ratings, and evaluated a rmse of 0.786.

Limitations

In the Results section we analyzed the performance of our model by looking at each true star category (from 0.5 to 5) and see if our predictions were too high/low or good. We learned that for lower star categories ($<$ 3 stars) the model overestimates the ratings, and for higher star categories ($>$ 4 stars) the model underestimates the ratings. Nevertheless the model predicts quite well for moderate star categories (between 3 and 4 stars), which is half of our data. Depends on how readers view it, it's a cup half full or cup half empty. Further analysis on which movies the model made its largest errors might tell us why the model makes these mistakes. (Is it the genres, or maybe the year? See next section!)

Future work

Incorporate more movie info

In the data cleaning step I only kept user ids, movie ids, and ratings because these are the only columns recosystem requires. However, the movie's genres and its released year are great information, since some genres/years get better ratings than others. After recosystem outputs the predicted ratings, we could add/subtract a small epsilon based on each movie's genres and year.

Using other libraries

I chose recosystem library because it supports parallel computation, but there are other libraries we could use for recommendation. For example, recommenderlab supports user and item based collaborative filtering algorithms in addition to matrix factorization we have been using. The disadvantage of recommenderlab is that the code cannot finish in a reasonable time, because the MovieLens 10M dataset is huge. If we use a smaller MovieLens dataset (like 1M), we could explore many algorithms, perhaps even create an ensemble algorithm.

Appendix

Movie Genres

- Action
- Adventure
- Animation
- Children's
- Comedy
- Crime
- Documentary
- Drama
- Fantasy
- Film-Noir
- Horror
- Musical
- Mystery
- Romance
- Sci-Fi
- Thriller
- War
- Western

References

MovieLens dataset

By GroupLens

MovieLens dataset on GroupLens: https://grouplens.org/datasets/movielens/10m/

MovieLens dataset readme: https://files.grouplens.org/datasets/movielens/ml-10m-README.html

recosystem

By Yixuan Qiu

recosystem on CRAN: https://CRAN.R-project.org/package=recosystem

recosystem readme: https://cran.r-project.org/web/packages/recosystem/vignettes/introduction.html