EdX Data Science Capstone Project

William Mee

2022-08-17

Summary

This is the machine learning capstone project for EdX Data Science online course.

I elected to implement a recommendation system based on the Movielens 10 million dataset. I began by following the bias models we explored in coursework, took this further by also using genre as a feature, attempted to use PCR and then switched to investigate some of the more advanced models of the R recommenderlab package.

The best result I obtained for the entire dataset used singular value decomposition.

Baseline: mean of all ratings

I began by establishing a simple root mean square error (RMSE) function and calculating a baseline RMSE by taking a simple mean of all the ratings: this value is **1.061202**.

Model #	Description	RMSE
0	Baseline	1.061202

```
rmse <- function(true_ratings, predicted_ratings) {
  sqrt(mean((true_ratings - predicted_ratings)^2))
}

mu_hat <- mean(edx$rating)
pred_0 <- rep(mu_hat, nrow(validation))
rmse(validation$rating, pred_0)
```

```
## [1] 1.061202
```

Bias models

I proceeded to calculate the bias successively of movies and users, so using the following to calculate prediction Y for movie i.

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

and for movie i and user u

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

where

- μ is the mean of all ratings
- b_i is the bias for movie i
- b_u is the bias for user u
- ϵ represents the residual i.e. the difference between the predicted and actual rating.

Code snippets are below. Using this approach, I obtained the following RMSE values, which got progressively better.

Model #	Description	RMSE
1	Movie Bias	0.9439087
2	Movie/User Bias	0.8653488

Movie Bias

```
movie_bias <- edx %>%
  group_by(movieId) %>%
  summarize(biasMovie = mean(rating)-mu_hat)

pred_1 <- validation %>%
  left_join(movie_bias, by='movieId') %>%
  pull(biasMovie)

pred_1 <- pred_1 + mu_hat

# Prediction 1: 0.9439087
rmse(validation$rating, pred_1)
```

```
## [1] 0.9439087
```

Movie/User Bias

```
user_bias <- edx %>%
  left_join(movie_bias, by='movieId') %>%
  group_by(userId) %>%
  summarize(biasUser = mean(rating - mu_hat - biasMovie))

pred_2 <- validation %>%
  left_join(movie_bias, by='movieId') %>%
  left_join(user_bias, by='userId') %>%
  mutate(prediction = mu_hat + biasMovie + biasUser) %>%
  pull(prediction)

# Prediction 2: 0.8653488
rmse(validation$rating, pred_2)
```

```
## [1] 0.8653488
```

Using regularization

In both cases, using regularization on the bias models improved results.

The training data `edx` was partitioned to obtain an optimal lambda parameter.

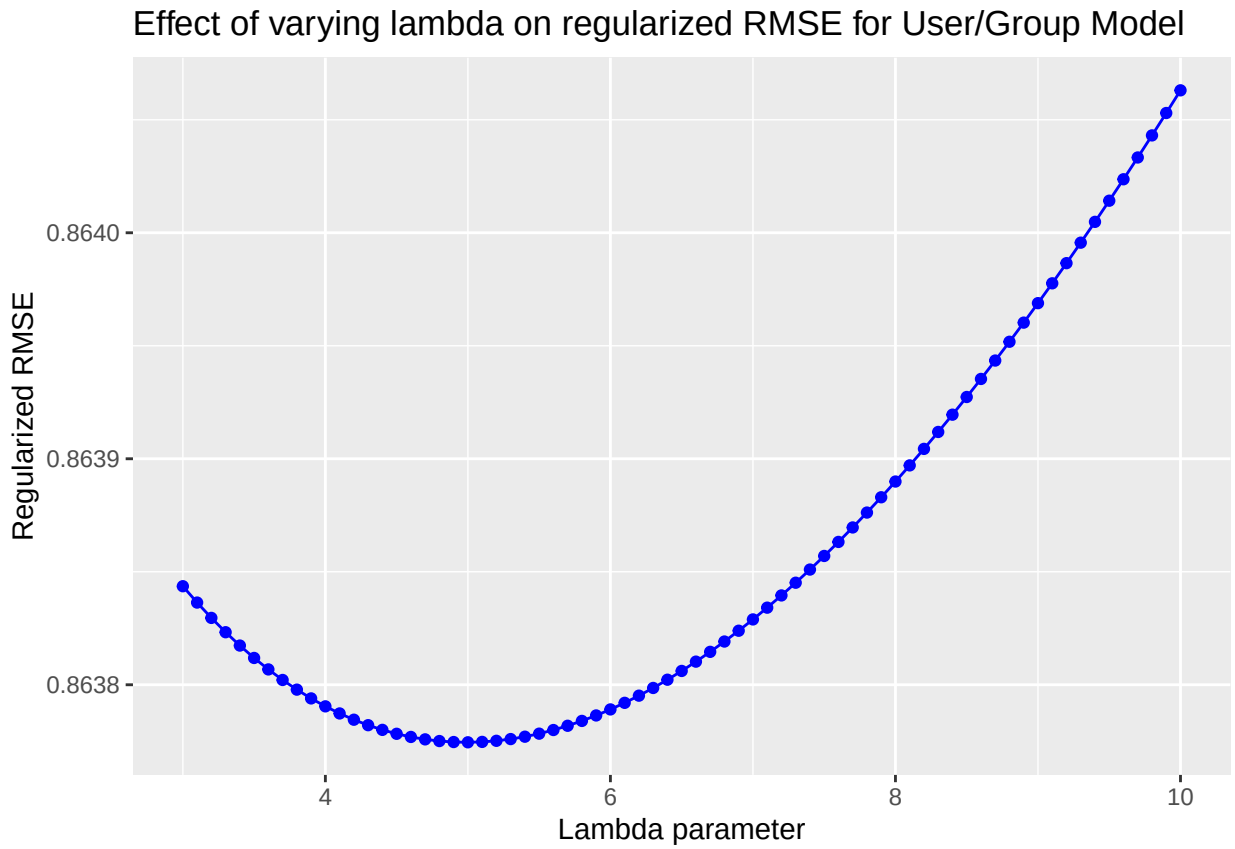
```
set.seed(51, sample.kind='Rounding')
test_index <- createDataPartition(y = edx$rating, times = 1, p = 0.1, list = FALSE)
edx_train <- edx[-test_index,]
temp <- edx[test_index,]

edx_test <- temp %>%
  semi_join(edx_train, by = 'movieId') %>%
  semi_join(edx_train, by = 'userId')

removed <- anti_join(temp, edx_test, by=c('movieId', 'userId'))
edx_train <- rbind(edx_train, removed)
```

```
## [1] 0.9438542
```

The code for movie and user bias is shown below (the movie-only version is a simplified subset of this)

```
regularized_prediction_movie_user <- function(lambda, train, test) {
  reg_mu <- mean(train$rating)

  movie_bias_regularized <- train %>%
    group_by(movieId) %>%
    summarize(biasMovie = sum(rating - reg_mu)/(lambda + n()))

  user_bias_regularized <- train %>%
    left_join(movie_bias_regularized, by='movieId') %>%
    group_by(userId) %>%
    summarize(biasUser = sum(rating - reg_mu - biasMovie)/(lambda + n()))

  pred <- test %>%
    left_join(movie_bias_regularized, by='movieId') %>%
    left_join(user_bias_regularized, by='userId') %>%
    mutate(pred = reg_mu + biasUser + biasMovie) %>%
    pull(pred)

  rmse(test$rating, pred)
}

lambdas <- seq(3, 10, 0.1)
rmses_regularized_movie_user <- sapply(lambdas, function(x) {
  regularized_prediction_movie_user(x, edx_train, edx_test)
})
```

I selected the lambda parameter programmatically, by iterating through values and taking the one with the minimum RMSE.




```
lambda <- lambdas[which.min(rmses_regularized_movie_user)]
lambda
```

```
## [1] 5
```

```
# Prediction 4: regularization with movie and user and optimal lambda
regularized_prediction_movie_user(lambda, edx, validation)
```

```
## [1] 0.8648177
```

The table below summarizes these results, and repeats the earlier unregulated models for comparison.

Model #	Description	RMSE	Lambda	RMSE w/out regularization
3	Movie Bias reg	0.9438542	1.75	0.9439087
4	Movie/User Bias reg	0.8648177	5.0	0.8653488

Genre bias

The Movielens data associates zero or more genres with a movie. I explored use of this as an additional feature. I began by creating a movie/genre dataframe and then calculating a genre bias for heavily-used genres (associated with over 100000 reviews), using the model

$$Y_{u,i} = \mu + b_i + b_u + \eta \sum_{g=1}^{k} b_g \theta_{g,i} + \epsilon_{u,i}$$

where

- η is a multiplier parameter for the genre biases
- b_g is the genre bias for genre g
- $\theta_{g,i}$ is a binary multiplier with a value of 1 if the movie i is associated with genre g and 0 otherwise.

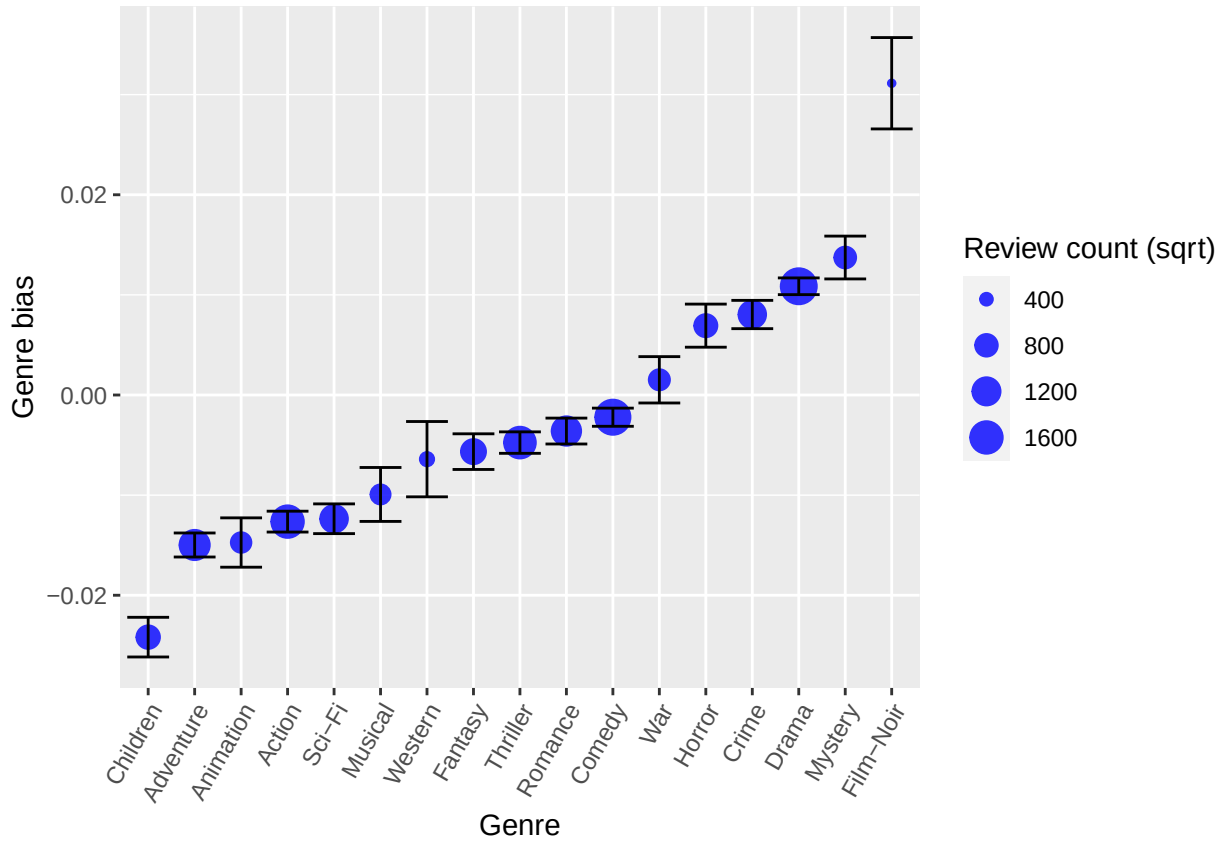
Using this model (together with regularization) gave a very slight improvement over the user/movie bias model.

Model #	Description	RMSE
5	Movie/User/Genre Bias reg	0.8647045

Implementation of this approach in R was done as follows:

```
genres <- edx %>%
  select(movieId, genres) %>%
  distinct() %>%
  separate_rows(genres, sep='\\|') %>%
  rename(genre = genres)

residuals <- edx %>%
  left_join(movie_bias, by='movieId') %>%
  left_join(user_bias, by='userId') %>%
  mutate(residual = rating - (mu_hat + biasMovie + biasUser)) %>%
  select(userId, movieId, residual)

genre_bias <- residuals %>%
  left_join(genres, by = 'movieId') %>%
  group_by(genre) %>%
  summarize(
    biasGenre = mean(residual),
    biasGenreSd = sd(residual),
    biasGenreSe = sd(residual)/sqrt(n()),
    genreCount = n(),
    biasGenreReg = sum(residual)/(genreCount + lambda)
  ) %>%
  filter(genreCount>=100000)
```

Using the approach we took in course-work, I confirmed that the bias based on the genre was not random by plotting with 95% confidence intervals showed .



Matrix-based approaches

In the second part of the project, I looked beyond the simple bias models to more sophisticated approaches, such as PCR. For this, I converted the test and training datasets to matrices, and used the R recommenderlab package

Conversion of data to Matrices

I converted the test and training data to matrix form using the Tidyverse *pivot_wider* function. I added in missing columns (movies) to the test data to ensure both had the same column dimension, giving

Matrix	Dimension (rows x columns i.e. users x movies)	Sparsity (NA ratio)
Train	69878 x 10677	0.987937
Test	68534 x 10677	0.9986334

It's worth noticing that the test matrix has almost the same number of rows (users) but is considerably more sparse.

```
train_matrix <- edx %>%
  select(userId, movieId, rating) %>%
  pivot_wider(names_from = 'movieId', values_from = 'rating') %>%
  as.matrix()
```

```
# add row names
rownames(train_matrix) <- train_matrix[,1]
train_matrix <- train_matrix[,-1]
# 69878 x 10677 users x movies
dim(train_matrix)
```

```
## [1] 69878 10677
```

```
test_matrix <- validation %>%
  select(userId, movieId, rating) %>%
  pivot_wider(names_from = 'movieId', values_from = 'rating') %>%
  as.matrix()

rownames(test_matrix) <- test_matrix[,1]
test_matrix <- test_matrix[,-1]
# 68534 x 9809 users x movies
dim(test_matrix)
```

```
## [1] 68534  9809
```

```
movie_id_delta <- setdiff(unique(edx$movieId), unique(validation$movieId))

test_matrix_ext <- matrix(data=NA, nrow=dim(test_matrix)[1], ncol=length(movie_id_delta))
colnames(test_matrix_ext) <- movie_id_delta
rownames(test_matrix_ext) <- rownames(test_matrix)
test_matrix <- cbind(test_matrix, test_matrix_ext)
# ensure column order is the same in both matrices
test_matrix <- test_matrix[,colnames(train_matrix)]

# 68534 x 10677
dim(test_matrix)
```

```
## [1] 68534 10677
```

I used the following simple function to determine matrix **sparsity** (ratio of NA entries)

```
matrix_sparcity <- function(matrix) {
  matrix_dim <- dim(matrix)
  sum(is.na(matrix))/(matrix_dim[1] * matrix_dim[2])
}
```

Use of naive PCA

Use of the R *prcromp* function to calculate PCA of the training matrix was so slow that I could not get it to complete. I abandoned this approach.

```
pca <- prcomp(train_matrix)
```

Recommenderlab

Recomenderlab is an R package for evaluating recommendation systems which implements several different algorithms. It uses sparse matrix representation and implements a set of different recommendation algorithms for easy comparison.

Setup and Workstation

To use recommenderlab, my original machine (MacOS/2.3 GHz quad core Intel i7, 16GB RAM) repeatedly ran out of memory giving a `vector memory exhausted` message, despite me configuring additional virtual memory via `R_MAX_VSIZE`. I switched to a more powerful modern machine (MacOS/Apple M1 Max, 64GB RAM) and was able to proceed.

Recommenderlab Algorithms

Recommenderlab supports multiple different algorithms and is designed to make them easy to compare. There is a good overview of the algorithms here. In the capstone, I looked at

- *RANDOM*: used for a baseline
- *UBCF*: User-based collaborative filtering
- *IBCF*: Item-based collaborative filtering (IBCF)
- *SVD*: Singular value decomposition with column-mean imputation
- *SVDF*: Funk SVD

Recommenders and RMSE for matrices with NAs

I setup recommenderlab recommenders based on the test and training matrices

```
train_rrm <- as(train_matrix, 'realRatingMatrix')
test_rrm <- as(test_matrix, 'realRatingMatrix')

rec_ubcf <- Recommender(train_rrm, method = 'UBCF')
rec_random <- Recommender(train_rrm, method = 'RANDOM')
rec_svd <- Recommender(train_rrm, method = 'SVD')
rec_svdf <- Recommender(train_rrm, method = 'SVDF')
rec_ibcf <- Recommender(train_rrm, method = 'IBCF')
```

The execution time of the recommender creation varied a lot between the algorithms; most were complete in a matter of minutes, but the *IBCF* recommender took over 24 hours to create.

Recommenderlab has its own evaluation functions, but I extended the basic algorithm to work with matrices with NAs. Some of the algrorithms (notable UBCF) also didn't provide predictions for all the test data (i.e. there were NAs in the prediction matrix which were not in the test matrix) so I also output the ratio of these failures.

```
rmse_matrix <- function(actual, predicted) {
  stopifnot(dim(actual) == dim(predicted))
  actual_cp <- actual
  predicted_cp <- predicted
  nas <- is.na(actual)
  predicted_cp[nas] <- -1
```

```
  residual_na_ratio <- matrix_sparcity(predicted_cp)
  actual_cp[nas] <- -1
  residual_nas <- is.na(predicted_cp)
  predicted_cp[residual_nas] <- -2
  actual_cp[residual_nas] <- -2
  list(rmse = sqrt(sum((actual_cp - predicted_cp)^2)/(sum(!(nas | residual_nas)))), na_ratio=residual_na
}
```

SVD and Funk SVD

The *SVD* algorithm is an implementation of singular value decomposition. I was easily able to evaluate the SVD recommender; prediction took about 3 minutes on the hardware described above.

```
system.time(pre_svd <- predict(rec_svd, test_rrm, type="ratingMatrix"))
rmse_matrix(test_matrix, as(pre_svd, "matrix"))
```

This gave me the following result, substantially better than the bias models above. The NA ratio (i.e. ratio of NAs in the prediction which weren't in test) was 0.

Model #	Description	RMSE	Prediction NA ratio
6	SVD	0.8356373	0

I couldn't create the *SVDF* recommender: R consistently reported a logical error.

User-based collaborative filtering (UBCF)

User-based collaborative filtering (UBCF) aggregates the movie ratings of users who rate movies similarly to a test user; it finds a **neighborhood** of similar users. The recommenderlab UBCF model looked to be very promising: tests against an initial small batch of users gave a substantially smaller RMSE score of **0.6703336**, but I had several issues:

1. The predict method was very slow, even on a powerful machine. I could predict on small batches of users (I was using a batch size of 1000), but each batch was taking between 30 and 90 minutes.
2. The recommendations produced NA entries for some user/movie entries which were in the test matrix; this is presumably because none of the users in the similar user neighborhood had reviewed these movies.
3. Most seriously: the UBCF predict call was triggering errors for some batches (see below) There were other reports of this online, but no clear resolution or work-around that I could find.

```
> Error in h(simpleError(msg, call)) :
   error in evaluating the argument 'x' in selecting a method for function 't': not-yet-implemented meth
```

Because of this third point, I couldn't evaluate the entire test set using UBCF.

Item-based collaborative filtering (IBCF)

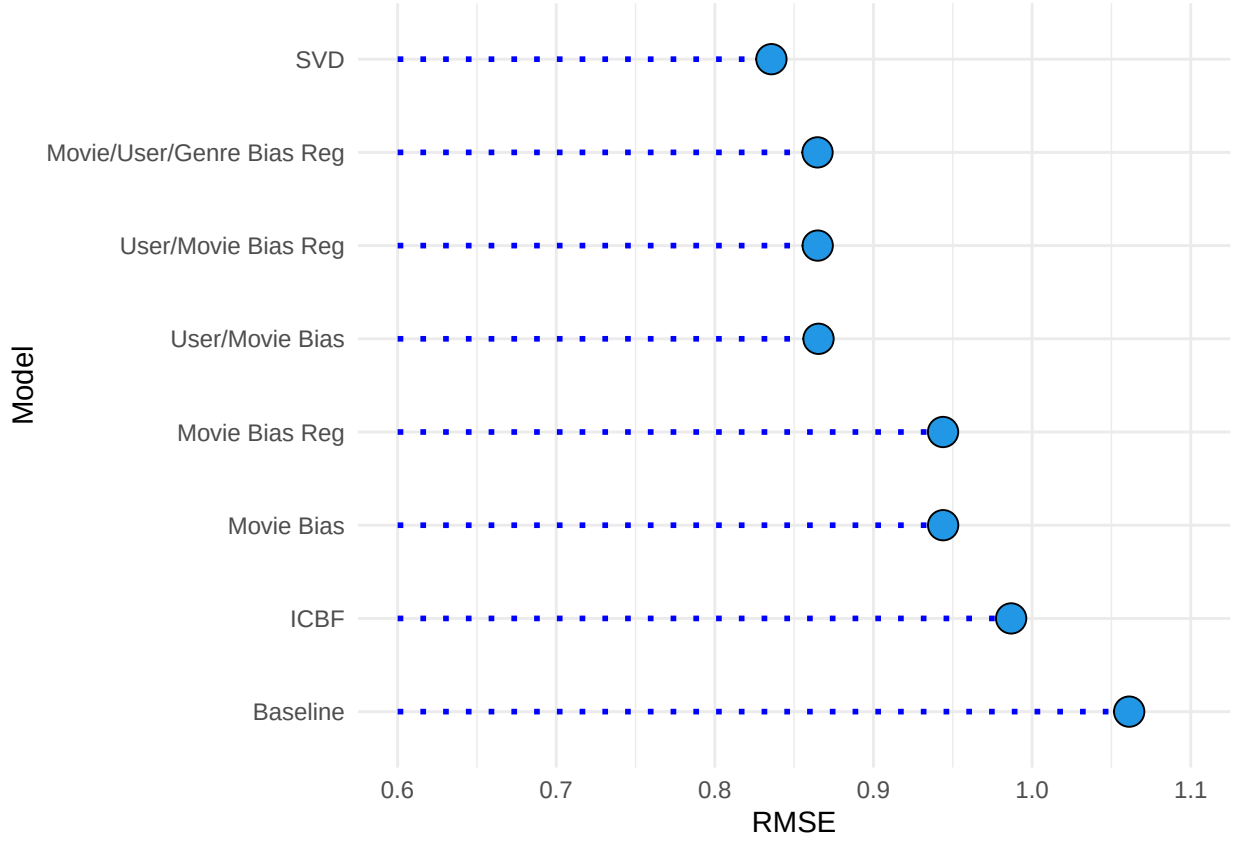
Item-based collaborative filtering (IBCF) establishes a similarity between items (in this case: movies) and makes predictions based on the user's other ratings, assuming that a user will have similar opinions about similar movies.

The IBCF recommender took a long time to create because it's creating a movie-to-movie similarity matrix for all 10677 movies; the evaluation of the test data was however very fast. However the results I got using IBCF were worse than the bias models.

Model #	Description	RMSE	Prediction NA ratio
7	Item-based collaborative filtering	0.9867809	0.001362822

Summary

An overview of the results obtained is below.



The best model I obtained used SVD, with an RMSE value of 0.8356373. I am reasonably confident that the UBCF algorithm would have provided an even lower error value, but I would have had to find an alternative implementation (or implement it myself)