PH125.9x - Predicting User Ratings Using the Movielens 10M Dataset

Brendan Hawk

2020-06-01

Introduction

In 1992, computer and data scientists John Riedl[1] and Paul Resnick [2] began work implementing what was to become known as GroupLens[3] - a recommender system computing predicted ratings of Usenet news articles. This became the foundation for the research lab of the same name at the Department of Computer Science and Engineering at the University of Minnesota[4]. This lab went on to publish the MovieLens dataset - a set of now-ubiquitous data comprised of "26,000,000 ratings and 750,000 tags applied to 45,000 movies by 270,000 users"[5].

In this report, the MovieLens 10M Dataset - so named for being 10 million records long - will be used to predict ratings of movies for given users. Predicting how much a given person will like a given product is not a novel concept. As long as there has been an economy of trade, people have needed to find ways to predict that market's response to new things in order to maximize profits. The more accurately these predictions can be made, the more successful the product or service can be. On a technical level, the goal here is to generate models that minimize the difference between predicted ratings and the known ratings for a given set of movies and users. In the real world the models generated would be applied to sets of data where the actual rating is unknown.

[1] https://en.wikipedia.org/wiki/John_T._Riedl
[2] https://en.wikipedia.org/wiki/Paul_Resnick
[3] https://en.wikipedia.org/wiki/GroupLens
[4] https://grouplens.org
[5] https://en.wikipedia.org/wiki/MovieLens#Datasets

Analysis

The Data

The data used in this analysis comes from the MovieLens 10M dataset, and has already been separated into a training dataset named `edx` and a validation dataset named `validation`. The `validation` dataset will not be touched until the very end, where it will be used to take a final measurement of the accuracy of our models. Some summary views of the data can demonstrate how the training dataset came out of the split:

```
edx[
  ,
  list(
    `Total Movies` = uniqueN(movieId),
    `Total Users` = uniqueN(userId),
    `Total Genres` = uniqueN(genres)
  )
]
```

```
##    Total Movies Total Users Total Genres
## 1:        10677       69878          797
```

We can see that of the total dataset, we are going to train our models with 10677 movies rated by 69878 users. We also have 797 genre tags, however it will remain to be seen if those are going to be useful for our analysis.

For analysis, it is prudent to have a set of training data set aside to validate against, without having to rely on the actual `validation` data as that can lead to overfitting. We will repeat the same exact process used in the script provided in the courseware to take a `testing` dataset of 10% of our `training` dataset, removing that data from the `training` dataset to prevent overfitting there as well.

```
# Split a chunk of data off to use as test data to validate
# training as we go.
testing.index <- createDataPartition(
  edx$rating,
  times = 1,
  p = 0.1,
  list = FALSE
)
training <- edx[-testing.index,]
testing.temp <- edx[testing.index,]

# Make sure userId and movieId in testing set are also in training set
# using as_tibble here and below because some joins are
# not compatible with dtplyr backend and data.tables,
# so applying them to lazy_dt's fails
testing <- testing.temp %>%
  as_tibble() %>%
  semi_join(training, by = "movieId") %>%
  semi_join(training, by = "userId")

# Add rows removed from testing set back into training set
removed <- anti_join(
  as_tibble(testing.temp),
```

```
  testing,
  by = names(testing)
)
training <- rbind(training, removed)

# Clean up memory
remove(edx, testing.index, testing.temp, removed)

# We will be relying heavily on data.table being much
# more performant and easier to manipulate than data.frame
training <- as.data.table(training)
testing <- as.data.table(testing)
```

We can now summarise the `training` and `testing` datasets to see that the partitioning has kept the number of movies, users, and genres as consistent as possible.

```
training[
  ,
  list(
    `Total Movies` = uniqueN(movieId),
    `Total Users` = uniqueN(userId),
    `Total Genres` = uniqueN(genres)
  )
]
```

```
##    Total Movies Total Users Total Genres
## 1:        10677       69878          797
```

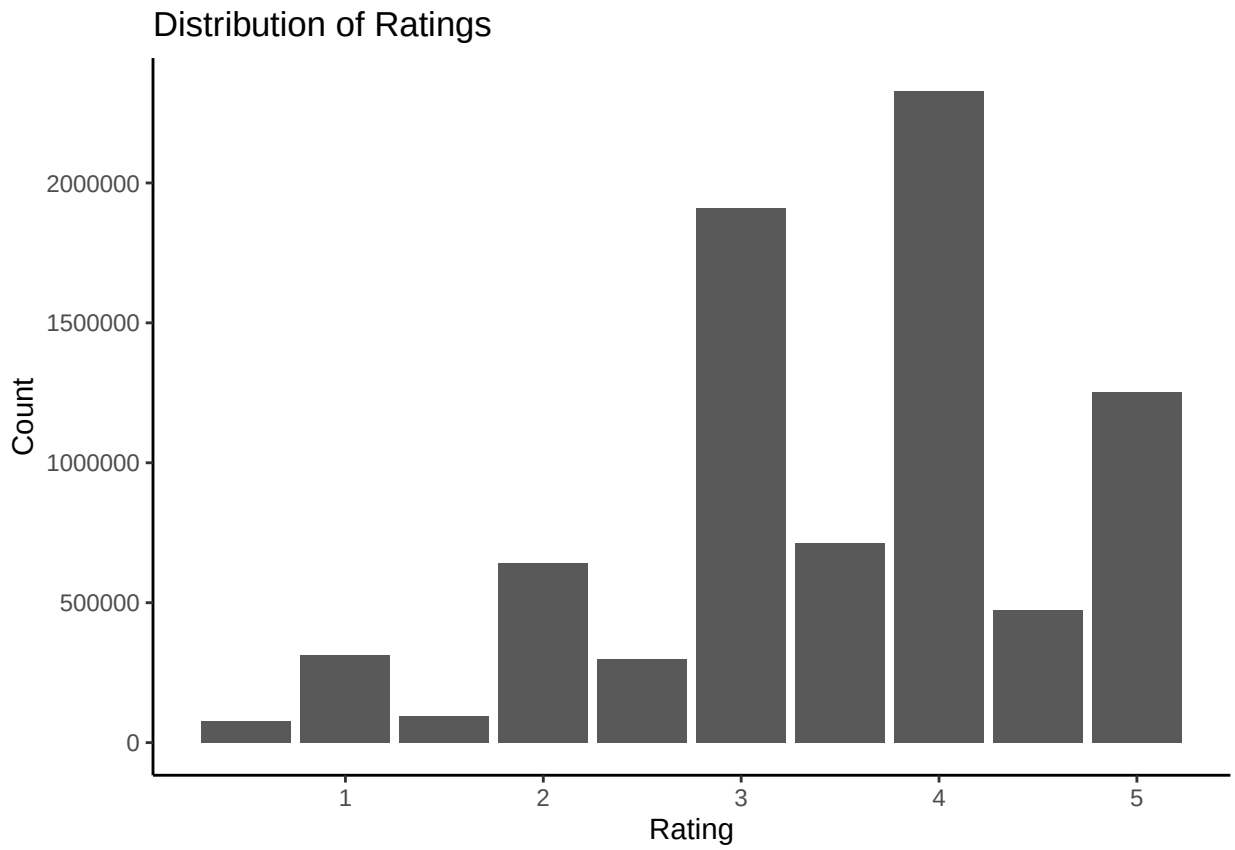
```
testing[
  ,
  list(
    `Total Movies` = uniqueN(movieId),
    `Total Users` = uniqueN(userId),
    `Total Genres` = uniqueN(genres)
  )
]
```

```
##    Total Movies Total Users Total Genres
## 1:         9736       68110          765
```

Exploration

Some cursory exploration shows that the ratings in this dataset are somewhat normally distributed, with integer ratings being far more common.

```
# distribution of ratings
training[, list(count = .N), by = rating] %>%
  ggplot(
    aes(x = rating, y = count)
  ) +
  geom_bar(stat = "identity") +
  labs(
    title = "Distribution of Ratings",
    x = "Rating",
    y = "Count"
  )
```



There is also a large discrepancy between a few users who have reviewed MANY movies, and most users who have only reviewed a few.

```
# Aggregate the counts of ratings per user
reviews.per.user <- training[, list(reviews = .N), by = userId]

# Summarise this data
summary(reviews.per.user$reviews)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     8.0    29.0    56.0   115.9   127.0  5968.0
```

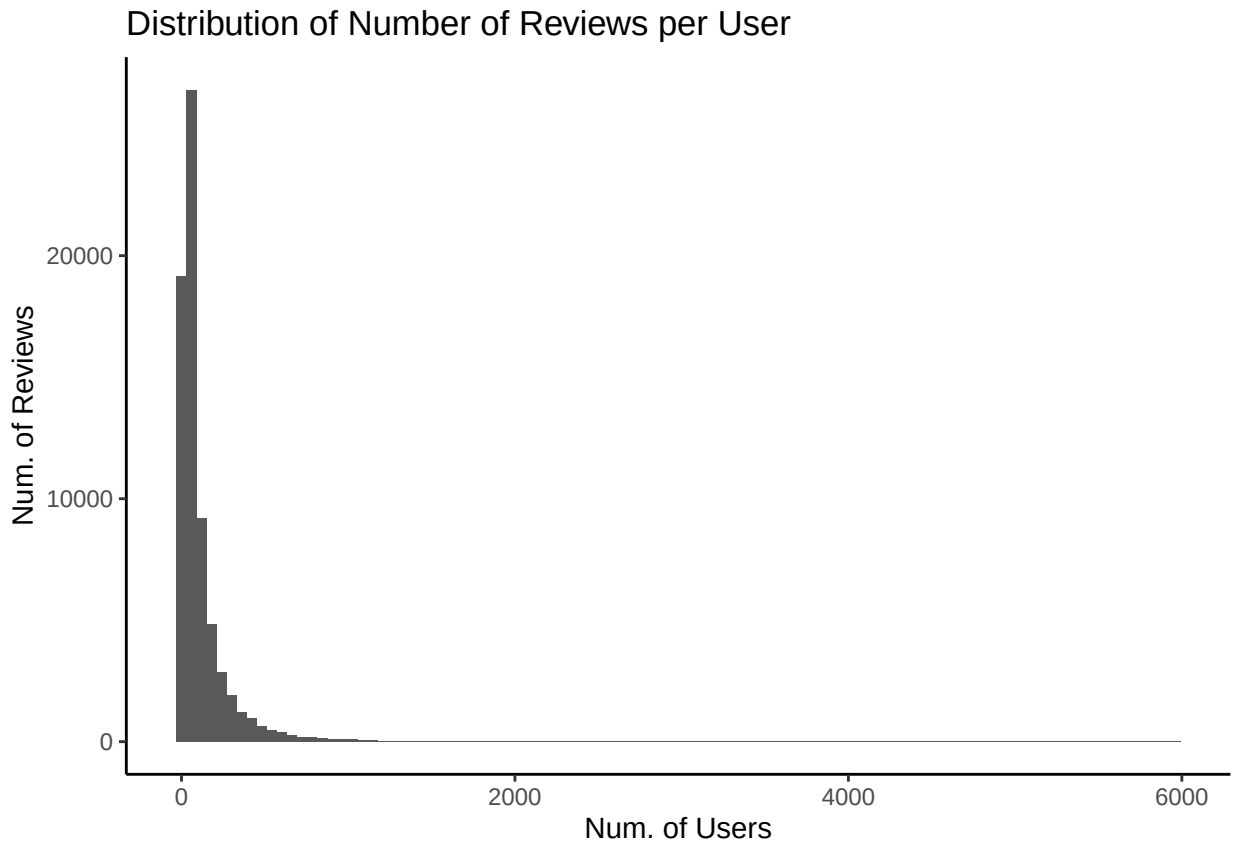
```r
sd(reviews.per.user$reviews)
```

```
## [1] 175.6065
```

```r
# What percent of users have less than the mean number of reviews ?
reviews.per.user[, list(sum(reviews <= mean(reviews.per.user$reviews))/nrow(reviews.per.user))]
```

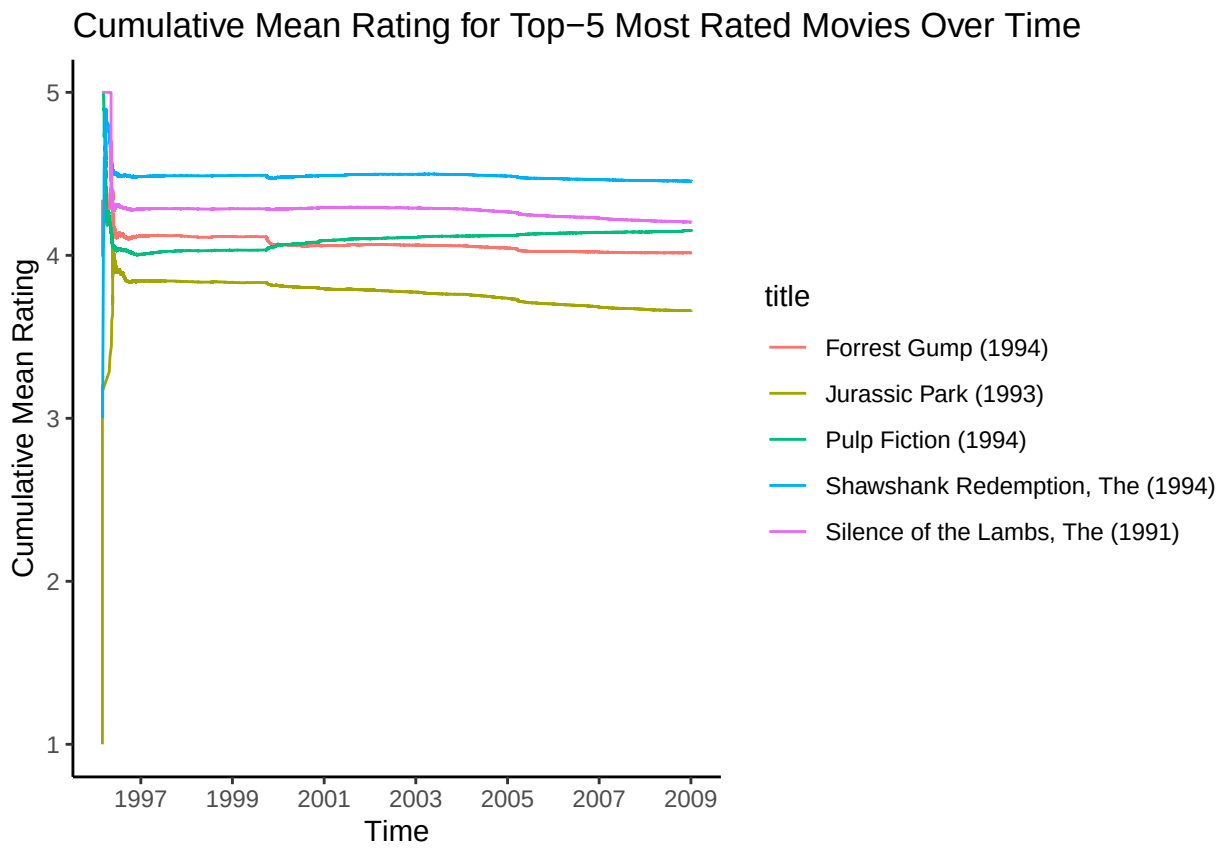
```
##           V1
## 1: 0.7270529
```

```r
# Discrepancies this large can be hard to conceptualise
# a visualisation can help.
reviews.per.user %>%
  ggplot(aes(x = reviews)) +
  geom_histogram(bins = 100) +
  labs(
    title = "Distribution of Number of Reviews per User",
    x = "Num. of Users",
    y = "Num. of Reviews"
  )
```



There are two other dimensions to this data that I felt warranted exploration, the timestamp of each review, and the genres assigned to each movie. The first question worth answering was whether or not the average review for a given movie changed over time. For this exploration, it's worthwhile to limit our processing to just a few of the most reviewed movie.

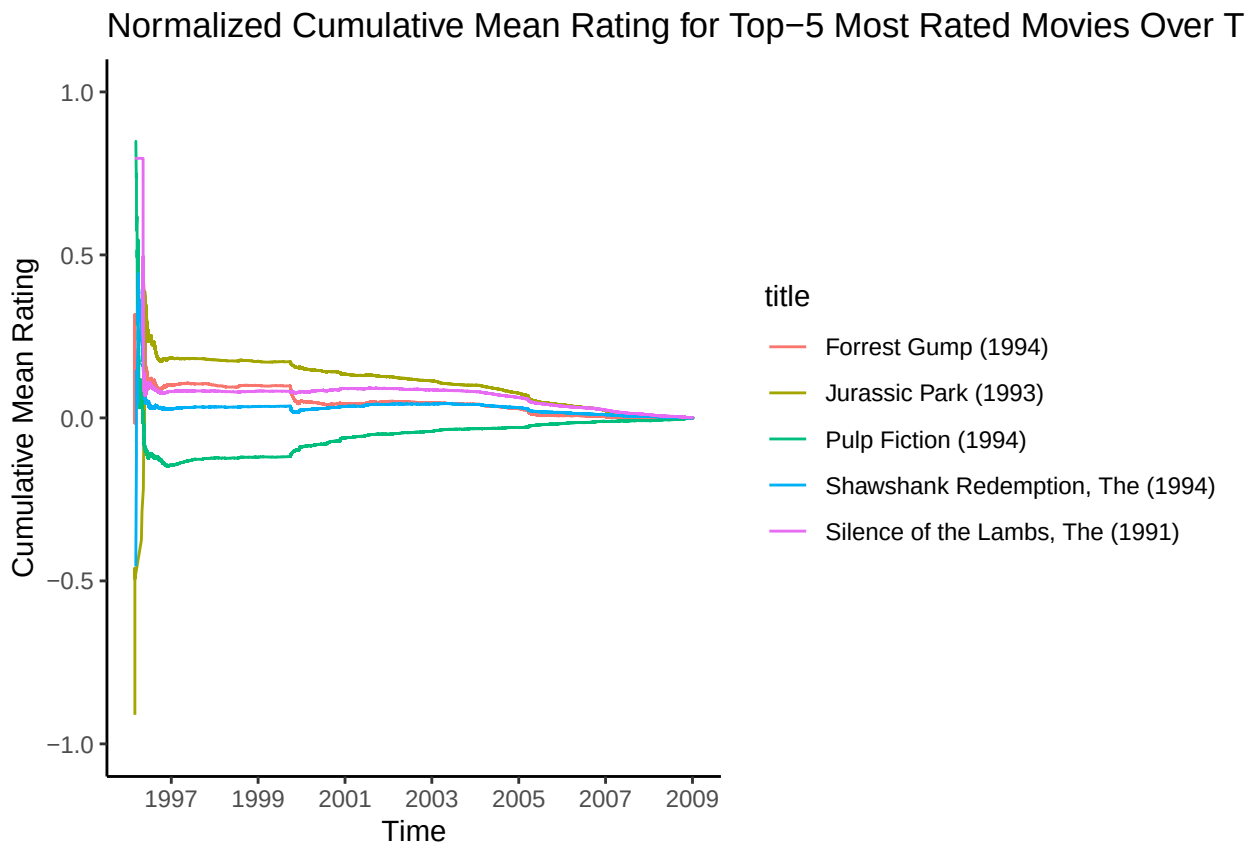
```
most.rated.movies <- lazy_dt(training) %>%
  count(movieId, sort = TRUE) %>%
  as.data.table() %>%
  head(5)

most.rated.movies <- merge(
  training,
  most.rated.movies
)
# grab the overall means for later
most.rated.movies.means <- lazy_dt(most.rated.movies) %>%
  group_by(title) %>%
  summarise(mean.rating = mean(rating)) %>%
  as.data.table()

# plot cumulative average over time
most.rated.movies <- lazy_dt(most.rated.movies) %>%
  group_by(title) %>%
  arrange(timestamp) %>%
  summarise(
    date = as_datetime(timestamp),
    cum.mean.rating = cummean(rating)
  ) %>%
  as.data.table()

most.rated.movies %>%
  ggplot(aes(x = date, color = title)) +
  geom_line(aes(y = cum.mean.rating)) +
  scale_x_datetime(
    date_breaks = "2 years",
    date_labels = "%Y"
  ) +
  labs(
    title = "Cumulative Mean Rating for Top-5 Most Rated Movies Over Time",
    x = "Time",
    y = "Cumulative Mean Rating"
  )
```



This chart begins to describe how the average rating for these movies change over time, however the range of the y scale is making it hard to see. Instead of the mean rating, we can normalize this value against each movie's overall mean, so that the line is instead demonstrating a difference from the current average.

```
# normalize each rating against it's eventual overall mean,
# so that these lines demonstrate the shift over time relative
# to their mean as if it were an asymptote
merge(
  most.rated.movies,
  most.rated.movies.means
) %>%
  lazy_dt() %>%
  mutate(cum.mean.rating = cum.mean.rating - mean.rating) %>%
  as.data.table() %>%
  ggplot(aes(x = date, color = title)) +
  geom_line(aes(y = cum.mean.rating)) +
  scale_x_datetime(
    date_breaks = "2 years",
    date_labels = "%Y"
  ) +
  scale_y_continuous(limits = c(-1, 1)) +
  labs(
    title = "Normalized Cumulative Mean Rating for Top-5 Most Rated Movies Over Time",
    x = "Time",
    y = "Cumulative Mean Rating"
  )
```

```
## Warning: Removed 4 rows containing missing values (geom_path).
```



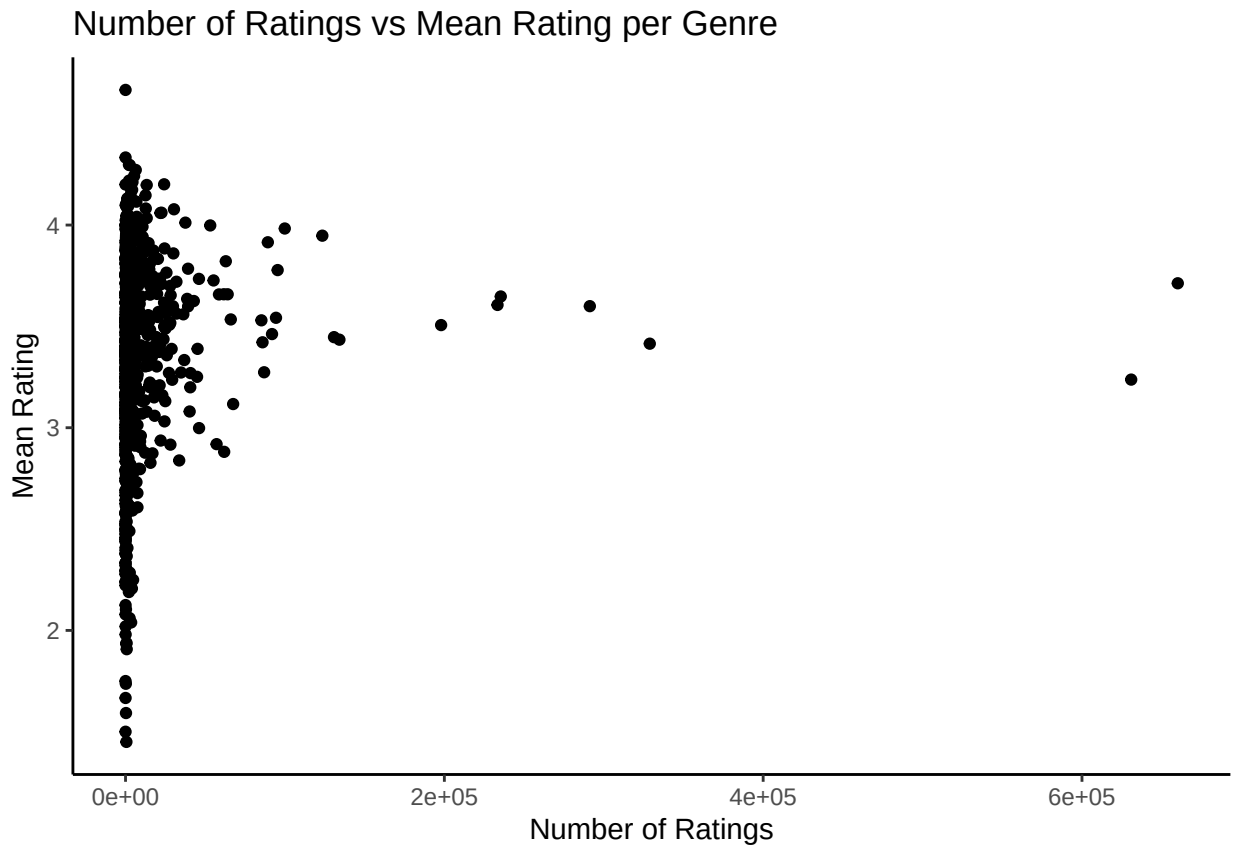

```
# clean up
remove(most.rated.movies, most.rated.movies.means)
```

We can see that the pattern is immensely consistent for these movies - a wildly unpredictable beginning but otherwise not a lot of variation over time. Further analysis will disregard this predictor, but it could be useful for approaches not applied here.

Lastly, we can explore the genres assigned to each movie. As a movie can be part of more than one genre, the values for this column in the MovieLens dataset is a concatenated list of one or more genres. Here we explore the average rating for each genre label as-is. Note that this chart doesn't try to display a legend for the genres in this chart, as there are far too many unique values and the chart would be illegible.

```
# average rating of all movies for every genre as-is
lazy_dt(training) %>%
  group_by(genres) %>%
  summarise(
    N = n(),
    rating = mean(rating)
  ) %>%
  as.data.table() %>%
  ggplot(aes(x = N, y = rating)) +
  geom_point() +
  labs(
    title = "Number of Ratings vs Mean Rating per Genre",
```

```
    x = "Number of Ratings",
    y = "Mean Rating"
  )
```



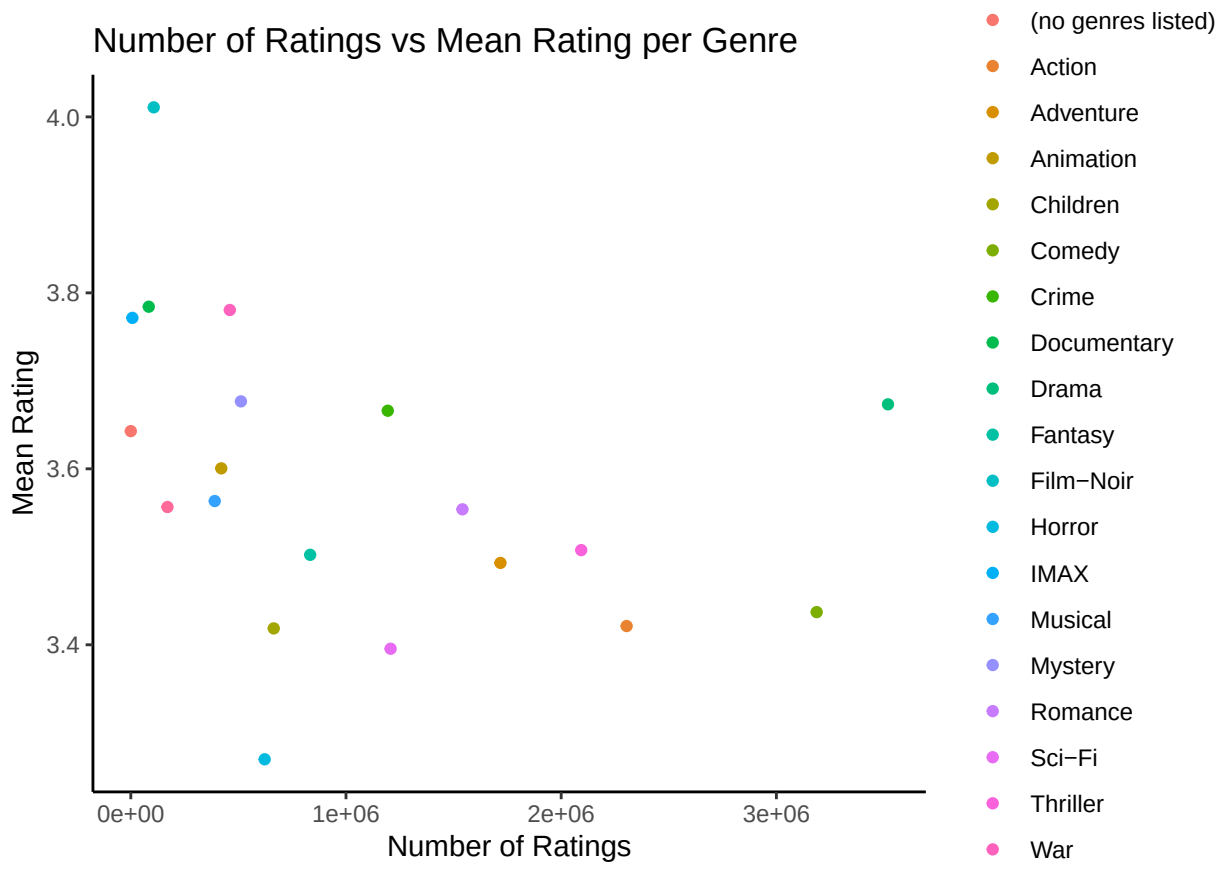
We can see that there does seem to be a correlation: the spread of mean ratings shrinks as the number of ratings for that genre increases. However, this data is FAR too confounded to make any useful conclusions. It's possible to reduce the number of dimensions we have for the Genres factor by separating the concatenated values into separate boolean columns.

```
# Create a binary matrix for each movieId (rows), denoting whether it includes
# or does not include a given genre (columns) as TRUE/FALSE
training.by.genres <- copy(training)
all.genres <- unique(training.by.genres$genres) %>%
  stringr::str_split("\\|") %>%
  unlist() %>%
  unique()

# This may take a few moments
for (genre in all.genres) {
  training.by.genres[
    ,
    ((genre)) := grepl(genre, genres, fixed = TRUE),
    by = genres
  ]
}
```

```
genre.means <- data.table()
for (genre in all.genres) {
  mean.for.genre <- training.by.genres[
    get(genre) == TRUE,
    list(genre = genre, rating = mean(rating), N = .N)
  ]
  genre.means <- rbind(
    genre.means,
    mean.for.genre
  )
}
genre.means
```

```
##                  genre   rating       N
##  1:             Comedy 3.437107 3187254
##  2:            Romance 3.553949 1541126
##  3:             Action 3.421290 2304140
##  4:              Crime 3.666038 1194210
##  5:           Thriller 3.507580 2093183
##  6:              Drama 3.673293 3518625
##  7:             Sci-Fi 3.395499 1207324
##  8:          Adventure 3.493042 1717735
##  9:           Children 3.418628  664335
## 10:            Fantasy 3.502246  833650
## 11:                War 3.780515  460087
## 12:          Animation 3.600535  420434
## 13:            Musical 3.563381  390046
## 14:            Western 3.556572  170341
## 15:            Mystery 3.676645  511788
## 16:          Film-Noir 4.010869  106681
## 17:             Horror 3.269823  622106
## 18:        Documentary 3.784253   83540
## 19:               IMAX 3.771589    7388
## 20: (no genres listed) 3.642857       7
```

```
genre.means %>%
  ggplot(aes(x = N, y = rating, color = genre)) +
  geom_point() +
  labs(
    title = "Number of Ratings vs Mean Rating per Genre",
    x = "Number of Ratings",
    y = "Mean Rating"
  ) +
  theme(legend.spacing.y = unit(8, 'pt'))
```




```
# Cleanup
remove(all.genres, genre, mean.for.genre, training.by.genres, genre.means)
```

We can see that taking all movies for a given genre drastically reduces the variability in their respective means, and even by eye we can also see that there isn't a huge correlation between a genres mean given its number of ratings. Again, this may be of use in another analysis, but for the algorithms applied in this report we will be disregarding this predictor.

Approaches and Models

Definition of Loss

Any statistical inference needs a measurement of accuracy. Here, we will use the root-mean-squared-error of our predictions versus known ratings to produce a measurement of loss.

```
loss <- function(predictions, observations) {
  sqrt(mean((observations - predictions)^2))
}
```

Naive Means

It is always useful to take a naive baseline when working towards statistical inference. In effect, it allows you to understand the fundamental question of whether or not the models applied are more or less accurate than a naive guess. For this data, it makes the most sense to use an average as a naive approach.

```r
mu <- mean(training$rating)

# Test this against the training dataset
loss(mu, training$rating)
```

```
## [1] 1.060272
```

```r
# Test this against the testing dataset
loss(mu, testing$rating)
```

```
## [1] 1.060867
```

As expected, this does not perform well. It does give us a baseline to measure further efforts against.

Modeling Movie and User Effects

A common approach to analysing this dataset is to model the effects of accounting for a given movie's average rating, or accounting for a given user's average rating on any movie. To put it simply, we can begin to answer the questions "Does the average rating for a movie help predict how a given user will feel about it?" and "Does how a given user feels about movies in general help predict how they will feel about a specific movie?". These questions are similarly naive approaches, however they are increasingly specific and can therefore begin to have a finer-grained sensitivity and yield better predictions.

To model these effects we start with the naive average, removing it from all ratings. Then we calculate each movie's average, and again remove it from all the ratings. Lastly, we perform the same step for each user's average. What is left can be described as a very specific residual measuring a single movie-user interaction.

```r
# first, sweep Mu out of data
training[, swept_rating := rating - mu]

# movie-mu, for modeling movie effect
mMu <- training[
  ,
  list(mMu = mean(swept_rating)),
  by = movieId
]

# sweep mMu out of data
training <- merge(training, mMu, by = "movieId") %>%
  lazy_dt() %>%
  mutate(swept_rating = swept_rating - mMu) %>%
  as.data.table()

# user-mu, for modeling user effect
uMu <- training[
  ,
  list(uMu = mean(swept_rating)),
  by = userId
]

# sweep uMu out of data
training <- merge(training, uMu, by = "userId") %>%
```

```
  lazy_dt() %>%
  mutate(swept_rating = swept_rating - uMu) %>%
  as.data.table()
```

Now that these averages are all calculated, we can take another naive approach to predict a rating by adding up `mu`, the given movie's mean from `mMu`, and the given user's mean from `uMu`.

```
# Make a prediction that Rating = Mu + mMu + uMu
loss(
  training[, list(prediction = uMu + mMu + mu)]$prediction,
  training$rating
)
```

```
## [1] 0.8562337
```

This is actually far more accurate than one might expect, however this accuracy is fragile. If we use the same means to predict on the `testing` dataset we can see that the measure of loss is noticeably greater.

```
setkey(testing, movieId)
setkey(mMu, movieId)
testing <- mMu[testing]

setkey(testing, userId)
setkey(uMu, userId)
testing <- uMu[testing]

loss(
  testing[, list(prediction = uMu + mMu + mu)]$prediction,
  testing$rating
)
```

```
## [1] 0.8657138
```

This does not give us a good enough accuracy for our purposes, and it is more than likely that the same approach on the `validation` dataset will be even worse.

From this point forward, we will use the "swept" residuals as the values we are trying to predict. Our last step before applying more complex approaches is to compute this residual and reduce our datasets to only the values we need.

```
training <- training[
  ,
  list(
    movieId,
    userId,
    rating = rating - mu - mMu - uMu
  )
]

testing <- testing[
  ,
  list(
```

```
    movieId,
    userId,
    rating = rating - mu - mMu - uMu
  )
]
```

A Formal Definition of the Problem

In essence, the goal of any algorithm applied to this problem is to fill in the gaps in a sparse matrix where each row is a user and each column is a movie - cell values are either blank (needing prediction) or filled with the rating that user has given that movie.

	Movie 1	Movie 2	Movie 3	...	Movie m
User 1	3.5	4	1	...	3.5
User 2	**?**	**?**	2	...	4
User 3	3	5	4	...	5
...	...	...	...	...	...
User u	2	5	**?**	...	2

Here, we can see that User 2 has not rated Movie 2, and our algorithm should be able to accurately predict the value that user would give that movie.

There are many similar mathematical approaches used to build models for filling in these gaps. The most common and powerful of them is a set of direct mathematical approaches called Matrix Decomposition. Effectively, it is possible to decompose or factorise the above matrix into to matrices of greatly reduced dimensions such that:

$$\tilde{R} = HW$$

where $ H $ is the matrix of users' latent factors and $ W $ is the matrix of movies' latent factors. To tune this model, the number of latent factors can be adjusted. Commonly, tuning this model uses Stochastic Gradient Descent[6] to *approximate* the minimisation of these solutions, since the "optimal" solution may not be a single unique tuning.

Two of the most common applications of matrix factorisation are Principle Component Analysis and Single-Value Decomposition. Both of these approaches factorise a high-dimension matrix of data into a set of linear vectors. As a bonus, however, PCA not only factorises the matrix of values, but also demonstrate how important each of the resulting latent factors is, allowing a data scientist to draw inferences about *how* different attributes of the data are related to the observed or predicted values.

There is a catch, however: even generating the first sparse matrix on our MovieLens dataset will require an exorbitant amount of memory. In fact, even if that weren't true, doing the resulting mathematical operations would take even more memory no matter how optimized our R code is. The basis of this approach is sound, but it is nearly impossible to implement directly on a dataset this size.

Recosystem::recommender

`Recosystem`[7] is an R wrapper to a very powerful and parallelized implementation of this approach called LIBMF[8]. Parallelization is very important in this case. As noted above, the dataset we are working 8ith yields a VERY large sparse matrix. For context, our `training` dataset would yield a matrix of 10677 rows

[6] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[7] https://cran.r-project.org/web/packages/recosystem/recosystem.pdf
[8] https://www.csie.ntu.edu.tw/~cjlin/libmf/

by 69878 users which is prohibitively large. LIBMF is designed to work around both of those limitations by computing chunks of the matrix in parallel and recombining them into a final model.

Usage of this R package is very straightforward. It is designed to work on data objects either in memory or in files, giving users flexibility to accommodate their data given the hardware they are running.

```
# Create the recommender object
recommender <- Reco()

# Create an in-memory data object with the training
# dataset. Recosystem can also use data residing in files.
trainingObject <- data_memory(
  user_index = training$userId,
  item_index = training$movieId,
  rating = training$rating,
  index1 = TRUE
)
```

The built-in training function is also parallelized, using k-fold cross validation to tune several parameters such as the learning rate and balance of cost equations used in regularizing the outcomes. Even in parallel, however, this process can take a very long time. In practice, I was able to run this using pretty substantial hardware for a consumer PC - 16 threads running at around 4GHz with 16GB of very fast RAM - and it still took several hours.

```
recoTuningResults <- recommender$tune(
  trainingObject,
  opts = list(
    dim = seq(10, 30, 10),
    costp_l1 = 0,
    costp_l2 = seq(0, 1, 0.25),
    costq_l1 = 0,
    costq_l2 = seq(0, 1, 0.1),
    lrate = 0.1,
    niter = 100,
    nfold = 5,
    nbin = 25,
    nthread = nThreads
  )
)

# The best-tuned opts can be shown here
recoTuningResults$min
```

I have duplicated the output of the call to `min` above, so that this report can be reproduced.

```
bestTune <- list(
  dim = 20,
  costp_l1 = 0,
  costp_l2 = 0,
  costq_l1 = 0,
  costq_l2 = 1,
  lrate = 0.1,
  loss_fun = 0.836
)
```

The algorithm is then tuned a final time with these options before predictions are made. Here we are creating another data object with our `testing` dataset and generating predictions for it before measuring the accuracy with our `loss` function.

```
# Tune the model with the best tuning options
recommender$train(trainingObject, opts = bestTune)

## NB in practice, you should pass the options directly from the tuning output
# recommender$train(trainingObject, opts = recoTuningResults$min)

# Create the testing dataset object in memory
testingObject <- data_memory(
  user_index = testing$userId,
  item_index = testing$movieId,
  rating = testing$rating,
  index1 = TRUE
)

# Generate our predictions
recosystem.predictions = recommender$predict(testingObject, out_memory())

# Measure our accuracy
## NB - this is being done with the "swept" ratings, however the results are
## mathematically identical to the results we would get if we "unswept" the
## mean, movie, and user effects.
loss(
  recosystem.predictions,
  testing$rating
)
```

As a measurement of accuracy, having our RMSE come out to 0.8218142 is phenomenal. It's possible finer-grained tuning could reduce this further, but at the cost of an exponential increase in tuning time.

Slope One Algorithm

The extraordinary memory and time cost of matrix factorisation gave me a great deal of concern when analysing this dataset. Even after discovering the Recosystem package, I felt uneasy about having an analysis done in a way I found difficult to explain. I went in search of other less costly approaches to apply by hand, in lieu of having the work done for me, and came across the Slope One algorithm[9]. This is a Collaborative Filtering approach - ie we will attempt to predict how a user feels about a movie by comparing the similarities and differences in how they feel about other movies to other users who have rated those other movies as well. The math behind this approach is astoundingly simple. The final model is calculated from a series of average differences. For example, given the following data:

User Id	Movie Id	Rating
U1	M1	4
U1	M2	3
U2	M1	2.5
U2	M2	5

We find all the combinations of each users ratings between separate movies and calculate the difference in

[9] https://en.wikipedia.org/wiki/Slope_One

ratings between them.

User Id	Movie Id 1	Movie Id 2	Difference in Rating
U1	M1	M2	4 - 3 = 1
U1	M2	M1	3 - 4 = -1
U2	M1	M2	2.5 - 5 = -2.5
U2	M2	M1	5 - 2.5 = 2.5

We then take the averages of those combinations of movies, here called `b`. Here, we keep track of how many data points went into computing this average, here called `support`.

Movie Id 1	Movie Id 2	b	support
M1	M2	mean([1, -2.5]) = -0.75	2
M2	M1	mean([-1, 2.5]) = 0.75	2

The `support` value is used to weight the calculated predictions so that predictions rely more on the given user's known opinions when there are few data points for the combination of Movie Id 1 and 2. So, to predict another user's rating for movie M2, given that they have rated movie M1:

User Id	Movie Id	Rating
U3	M1	4
U3	M2	?

We calculate the following simple equation:

$$Prediction_{U3,M2} = \frac{(Rating_{U3,M1} + b_{M1}) * support_{M1}}{support_{M1}}$$

With the given example values:

$$Prediction_{U3,M2} = \frac{(4 + -0.75) * 2}{2} = 3.25$$

In this application of Slope One, we still have the same issues with the size of our data. Similarly to the internals of Recosystem and LIBMF, I implemented Slope One for this dataset in a way that it can be done in chunks of data in parallel. This drastically reduced the memory overhead, and the final model was calculated in about 20 minutes. Each thread will process it's respective part of the model, then write to a temporary file. These files are re-aggregated after all of the data is processed into these temporary files.

```
## The function that will calculate our Slope One model.
slopeone <- function(ratings) {
  # Iterate over every user's data, calculating the difference in scores
  # between each pair of movies they've rated - Order matters in these
  # permutations, so we will get (movie1 = x, movie2 = y)
  # as well as (movie1 = y, movie2 = x) in the resulting models.
  rating.diffs <- ratings %>%
    as_tibble() %>%
    group_by(userId) %>%
    group_map(function(user.data, data.key) {
```

```
      # to keep memory overhead minimal, we will garbage collect on ever iteration
      gc()

      # First: generate every combination of movies for this user's data
      row.combinations <- as.data.table(expand.grid(
        row1 = 1:nrow(user.data),
        row2 = 1:nrow(user.data)
      ))

      # Filter out rows where the same movie is used twice
      row.combinations <- row.combinations[row1 != row2]

      # For every combination generated above, calculate a difference
      # between the two movie scores.
      data.table(
        movieId1 = user.data$movieId[row.combinations$row1],
        movieId2 = user.data$movieId[row.combinations$row2],
        diff = user.data$rating[row.combinations$row2] - user.data$rating[row.combinations$row1]
      )
    }) %>%
    rbindlist()

  # If this chunk of user data didn't end up providing us any combinations
  # of movies it means this set of users EACH only rated 1 movie.
  # We cannot leverage that data, so return an empty model.
  if (nrow(rating.diffs) == 0) {
    return(data.table(movieId1 = c(), movieId2 = c(), b = c(), support = c()))
  }

  # Finally, return the resulting model, complete with supporting data
  rating.diffs[
    ,
    list(b = mean(diff), support = .N),
    by = list(movieId1, movieId2)
  ]
}
```

In order to split the dataset up in a rational way for parallelization of Slope One, there needs to be two major concessions. First, the huge discrepancy between users who have rated many movies versus users who have rated very few can create drastically different sizes of data for each thread. Secondly, each user's data *must* be kept together. For these reasons I took a rough "binning" approach - counting how many reviews each user has done and separating the dataset into `nBins` bins so that the user data is kept together and the total size of the chunks is roughly similar.

```
nBins <- 100

# We must keep all of a given user's data together, so we will be binning
# by the aggregate number of reviews a user has given.
binData <- training[, .N, by = userId]

# Cumulatively sum these counts, breaking roughly when the sum approaches
# (total_number_of_reviews/nBins)
binData$bin <- cumsum(binData$N) %/% (ceiling(sum(binData$N) / nBins)) + 1
```

```
# Merge these bin numbers in as a column
setkey(binData, userId)
training <- binData[training]

# cleanup
remove(binData)
```

Here, I have generalized the processing step, and for the sake of this report the following code will not run. Even with parallel processing on my aforementioned hardware, this process can take upwards of an hour or two.

```
foreach(
  binIndex = 1:nBins,
  .packages = c("data.table", "dplyr", "dtplyr")
) %dopar% {
  # create the file path for this bin's data.
  filePath <- sprintf("./temp/model_%d.csv", binIndex)

  # calculate and write this data to the file
  write.table(
    slopeone(training[binIndex, .(userId, movieId, rating)]),
    filePath,
    sep = ",",
    row.names = FALSE,
    na = ""
  )
}
```

Once all the individual chunks of data are written to file, we can re-aggregate the averages into a single final model. This is the most memory intensive portion of the process, so the following code is generalized and will not run.

This process is possible without any loss of accuracy because we maintain the `support` values. Given two rows from different chunks of the Slope One model parts:

Movie Id 1	Movie Id 2	b	support
M1	M2	3	10
M1	M2	2	25

We can calculate a new, completely accurate average by multiplying the values of `b` with their given `support` value before creating a new mean:

$$b = \frac{(b_1 * support_1) + (b_2 * support_2)}{support_1 + support2}$$

Here we do this by iterating over every temp file, adding it to the model as we go, and calculating these "re-averaged" values for `b` and `support` for each combination of movies. Again, this code is generalized and will not run here.

```
# This function will do the math of re-averaging for us as the files are read.
# It will be used in the %dopar% process below as the .combine function.
reaggregate.slopeone <- function(...) {
```

```
  # Each file read in will take some memory, and the calculation
  # below will consume more memory as well, so it was helpful to
  # be zealous with running garbage collection as I went.
  gc()

  rbindlist(list(...))[
    ,
    .(b = sum(b * support)/sum(support), support = sum(support)),
    by = .(movieId1, movieId2)
  ]
}

# This will read several files at a time in parallel, and then call
# the function above to aggregate and re-average them into one model.
slopeone.model <- foreach(
  binIndex = 1:nBins,
  .packages = c("data.table", "dplyr", "dtplyr"),
  .combine = reaggregate.slopeone,
  .multicombine = TRUE
) %dopar% {
  fread(sprintf("./temp/model_%d.csv", binIndex))
}
```

With our final model aggregated, we can make predictions. This process is not memory intensive, but does take some time. First, we set up the function that will perform our calculations. Then we need to apply it to every combination of User and Movie we want to predict. Again I leveraged parallel processing, allowing this step to complete in roughly 40 minutes. This code is also generalized and will not run for this report.

```
# set up the function that will calculate Slope One predictions
# for a given user and movie.
predict.slopeone <- function(model, target.movieId, user.ratings) {
  # Alter the input data so that we can merge with our model data
  setnames(user.ratings, 'movieId', 'movieId1')

  # Set the target movie id as movieId2
  user.ratings$movieId2 = target.movieId

  # set the data.table key for the merge
  setkey(user.ratings, movieId1, movieId2)

  # Join the model and user data
  joined <- model[user.ratings, ]

  # Filter any rows that are incomplete (any relevant column is NA)
  joined <- joined[complete.cases(joined), ]

  # If there are no cases where movieId2 = target & movieId1 = some movie this
  # target user has rated, then we cannot predict a value here. This is a limitation
  # of the Slope one algorithm.
  if (NROW(joined) == 0) {
    return(NA)
  }

  # Otherwise, compute and return the final prediction
```

```
  return(sum(joined[, (b + rating) * support]) / sum(joined[, sum(support)]))
}
```

```
# Set keys, so that joins are considerably faster.
setkey(training, userId)
setkey(slopeone.model, movieId1, movieId2)

# This prediction step also takes a very long time, even in parallel. This
# is because the calculation has to be run separately for every combination
# of movie/user, which means we are iterating over every row in
# the testing dataset.
predictions <- foreach(
  rowIndex = 1:nrow(testing),
  .packages = c("data.table", "dplyr", "dtplyr"),
  .combine = rbind,
  .multicombine = TRUE,
) %dopar% {
  data.table(
    testing[rowIndex, ],
    prediction = predict.slopeone(
      slopeone.model,
      testing$movieId[rowIndex],
      training[J(testing$userId[rowIndex]), ]
    )
  )
}
```

Normally at this point we could calculate the RMSE without adding the Mu, movie, and user effects. However, it is entirely possible that the Slope One model has generated `NA` as a prediction in some cases. As explained in the model function, this comes from an instance where there is no combination of data to line up for a given user/movie and therefore no inference can be performed. In this case, we will have to fall back on a different method. Here I have chosen to use the Naive model + Movie and User effects for simplicity's sake.

```
predictions[
  ,
  predicted_rating := ifelse(is.na(prediction), 0, prediction) + mu + mMu + uMu
][
  ,
  unswept_rating := rating + mu + mMu + uMu
]

loss(predictions$predicted_rating, predictions$unswept_rating)
```

In practice, I calculated a loss against our testing set of 0.8571771.

Results

The second set of data set aside by the provided script at the beginning of this process is now going to be used to calculate a final measure of loss. To begin, we need to read this data from file and sweep the values of Mu, Movie, and User effects we calculated from our training data.

```
# Read the validation dataset from file
validation <- fread("./data/validation.csv")

# Only these three columns are needed for the algorithms I'm applying
validation <- validation[
  ,
  .(userId, movieId, rating)
]

# Set keys, join movie and user Mu's
setkey(validation, movieId)
setkey(mMu, movieId)
validation <- mMu[validation]

setkey(validation, userId)
setkey(uMu, userId)
validation <- uMu[validation]

# Sweep Mu and Movie/User effects from ratings
validation[
  ,
  rating := rating - mu - mMu - uMu
]
```

Recosystem::recommender

Making the final predictions for the Recosystem model is straightforward. An in-memory data object is created and the model's predict function is called.

```
# create the Recosystem data object for the validation dataset
validationObject <- data_memory(
  user_index = validation$userId,
  item_index = validation$movieId,
  rating = validation$rating,
  index1 = TRUE
)

# Create the predictions as a new column on this data.table
validation <- cbind(
  validation,
  prediction = recommender$predict(validationObject, out_memory())
)

# Take a final measurement of loss
loss(
  validation[, list(prediction = prediction + mu + mMu + uMu)]$prediction,
  validation[, list(rating = rating + mu + mMu + uMu)]$rating
)
```

```
## [1] 0.8211413
```

```
# 0.8211413!
```

My final measurement of loss for this algorithm was **0.8211413**. This is by far the best RMSE I achieved in any approach.

Slope One Algorithm

The final predictions for the Slope One model are similarly easy. Also similar is the time it takes to process this - in practice about 40 or so minutes. The code below is not run as part of this report, and is generalized for simplicity's sake.

```
# Set keys on the data.table
setkey(training, userId)
setkey(slopeone.model, movieId1, movieId2)

# Make the predictions
predictions <- foreach(
  rowIndex = 1:nrow(validation),
  .packages = c("data.table", "dplyr", "dtplyr"),
  .combine = rbind,
  .multicombine = TRUE,
) %dopar% {
  data.table(
    validation[rowIndex, ],
    prediction = predict.slopeone(
      slopeone.model,
      validation$movieId[rowIndex],
      training[J(validation$userId[rowIndex]), .(userId, movieId, rating)]
    )
  )
}

# A final measurement of loss for the Slope One model
loss(
  predictions[, list(prediction = prediction + mu + mMu + uMu)]$prediction,
  predictions[, list(rating = rating + mu + mMu + uMu)]$rating
)
# 0.8568992
```

My final measurement of loss for the Slope One model was **0.8568992**. This really astounded me, as the math behind it is so much less complex than the other approaches, but the accuracy is comparable. Had I only used this method, I would still have achieved respectable results for this report.

Conclusion

In summary, we can attest that the hardest part of these predictions may well have been the size of our dataset. Indeed, most of the time I spent working on this project was trying to find an approach that balanced results with feasibility - memory and time concerns forced me to adapt many approaches or drop them altogether.

We can also conclude that Matrix Decomposition is an incredibly powerful and accurate tool for the job. Had I been running this on a machine with unlimited RAM and power, I would never have had to look for anything further. It is likely I would have spent far more time doing very fine-grained tuning on that model and seeing just how low my RMSE could get. However, the fact that Slope One came as close as it did, and is such an easy-to-implement and flexible approach, was well worth the exploration my hardware limitations forced me to do.

I intend to keep exploring ways of honing the code behind my Slope One implementation, and have been considering creating an appropriate model package for Caret[10] so that others can use it as well. There still remains the challenge of being able to use it more flexibly - allowing potential consumers of that package to change between in-memory and write-to-file approaches easily and without a lot of need to understand the code behind it. Regardless, the algorithm has been well worth learning and has proven that not all Machine Learning needs to be mathematically complex!

[10] https://topepo.github.io/caret/