Introduction

In the following study, we will utilize the Rstudio environment to tackle one of the classic machine learning datasets. The MovieLens dataset, published by the MovieLens recommendation service, is commonly used for benchmarking new machine learning algorithms, especially recommendation systems, and for educational purposes. The full dataset contains 27 million movie ratings applied to 58,000 movies and 280,000 users collected over time. For our purposes, we will be utilizing a subset of this data. This smaller dataset contains 10 million ratings for 10,000 movies and 72,000 users contained in a 63MB set. For each entry, a movieId and userId are included which uniquely identify the movie and user providing the rating, respectively.

The goal of the study is simple: we would like to find a machine learning algorithm that is best able to predict a user's rating for a given movie based upon the available data, i.e. their previous movie ratings and previous ratings from other users for the movie(s) of interest.

Methods and Analysis

We will begin by importing the data and cleaning it. Import involves reading the data in from a URL and then extracting the relevant columns from the ratings and movie tables. We will then clean the data by merging the data tables and ensure each column is in the format that will be most useful to us. In a later section, we will break up the genres column into a factor table that will allow us to use this for prediction as well, as well as extract the release year from the title. We conclude this initial preprocessing step by splitting the data into training and validation sets.

Exploratory data analysis techniques are then utilized to view the relevant features of the dataset, including its overall size, distributions of ratings, genres, and reviews per film. We quickly examine the timestamp column to determine if it is worth using this as a predictive feature using correlation and a simple plot. A short discussion of PCA and SVA is included, though these techniques are not explored here due to the nature of the data and the relatively small number of features in comparison to much higher-dimensional genomics data.

In the main section, we validate several machine learning algorithms on the training data and tune relevant hyperparameters. However, given the large size of the dataset and the lack of access to high-performance computing clusters, we do not take full advantage of the diversity of models available in the caret package, instead providing some relevant code commented out so the user can explore further if desired and if resources are available. We restrict ourselves primarily to regression methods, beginning by building linear regression predictions based on available features and finally add in regularization to improve performance. We do not use R's built-in lm() function since it requires allocation of several GB of memory, which is not available. Similar constraints prevent us from using quadratic or linear discriminant approaches. Temporal constraints prevent us from using other methods, such as k-nearest neighbors, random forests, or matrix factorization, but code for these is included.

Based upon the results from the training set, we select the model which shows the smallest root mean square error and then use the held-out validation set to quantify the final model performance.

Results

1. Data Loading and Cleaning, Formation of Training and Validation Sets

We begin our exploration of the data by loading the data from a specified file, parsing the ratings and movies, and splitting the data into a training and validation set. Note that the following block of code was provided by the EdX course website and was designed to work with R 3.6 or later. We'll first load in the tidyverse, caret, data.table, lubridate, ggplot2, and recosystem packages to be used later.

```
############################################################
# Create edx set, validation set (final hold-out test set)
############################################################
```

```r
# Note: this process could take a couple of minutes

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

```
## Loading required package: tidyverse
```

```
## -- Attaching packages --------------------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.0
## v tidyr   1.1.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

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

```
## Loading required package: caret
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

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

```
## Loading required package: data.table
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:dplyr':
##
##     between, first, last
```

```
## The following object is masked from 'package:purrr':
##
##     transpose
```

```r
if(!require(lubridate)) install.packages("lubridate", repos = "http://cran.us.r-project.org")
```

```
## Loading required package: lubridate
```

```
##
## Attaching package: 'lubridate'
```

```
## The following objects are masked from 'package:data.table':
##
##     hour, isoweek, mday, minute, month, quarter, second, wday, week,
##     yday, year
```

```
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
```

```
if(!require(ggplot2)) install.packages("ggplot2", repos = "http://cran.us.r-project.org")
if(!require(recosystem)) install.packages("recosystem", repos = "http://cran.us.r-project.org")
```

```
## Loading required package: recosystem
```

```
options(tidyverse.quiet = TRUE)
library(tidyverse)
library(caret)
library(data.table)
library(lubridate)
library(ggplot2)
library(recosystem)
```

Next, we'll download the dataset and extract the relevant columns. The link is provided in the comment as well as in the download.file() command. We download the dataset into a temporary file before parsing. We then unzip the ratings file, which is structured with four elements in each row separated by "::". Rows are separated by tab. The four elements are the userId, movieId, rating, and timestamp.

The movies table is similar, with columns movieId, title, and genres using the same separator. We then convert the movies object to a data.frame and ensure that movieId, title, and genres are numeric, character, and character elements, respectively. Lastly, we join the two tables by movieId with the left_join() function, ensuring that only ratings which have movie information are present.

```
# MovieLens 10M dataset:
# https://grouplens.org/datasets/movielens/10m/
# http://files.grouplens.org/datasets/movielens/ml-10m.zip

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")

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")
```

We will now split the movielens object into training and validation sets by rating, using the createDataPartition() function. The validation set will be 10% of the data, and the training set will be the remaining 90%. The set.seed() method used here is for versions of R of 3.6 or later and can be used to compare results.

```
# Validation set will be 10% of MovieLens data
set.seed(1, sample.kind="Rounding")
```

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
## used
```

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

We now need to check to make sure that the userId and movieId values found in the validation set are also

found in the training set, or else our algorithm will have a difficult time predicting values for these untrained users or films. We do this using the semi_join() function, which returns rows of the first argument found in the second argument (temp and edx, respectively). We then use the anti_join() function to put the removed data back into the training data set, so we can still use it. Lastly, we remove the objects we no longer need.

```
# 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)
```

```
## Joining, by = c("userId", "movieId", "rating", "timestamp", "title", "genres")
```

```
edx <- rbind(edx, removed)

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

Another step that we will perform is to re-format the genre information. The current format of the genres information is a single column in which each genre to which a movie belongs is separated with a vertical bar (|). To be useful in predictions, it would be more suitable if this information were binary, and each movie did or did not belong to a given genre. To accomplish this, we utilize tidyverse tools.

```
# which genres are unique?
genre <- edx$genres %>%
  unique() %>%
  str_split("\\|") %>%
  unlist() %>%
  unique() %>%
  sort()
genre_validation <- validation$genres %>%
  unique() %>%
  str_split("\\|") %>%
  unlist() %>%
  unique() %>%
  sort()

# confirm same sets of genres present in training and validation sets
identical(genre, genre_validation)
```

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

```
# make new columns for all genres
for (i in 1:length(genre)) {
  edxmov <- edx$movieId %in% edx$movieId[grep(genre[i], edx$genres)]
  valmov <- validation$movieId %in% validation$movieId[grep(genre[i], validation$genres)]
  edx <- edx %>%
    mutate(!!genre[i] := edxmov)
  validation <- validation %>%
    mutate(!!genre[i] := valmov)
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
```

We can confirm that this process did as expected through some simple checks.

```
head(edx$genres[edx$Thriller])
```

```
## [1] "Comedy|Mystery|Thriller" "Horror|Thriller"
## [3] "Horror|Thriller"         "Comedy|Thriller"
## [5] "Action|Romance|Thriller" "Comedy|Thriller"
```

```
head(edx$genres[edx$Comedy])
```

```
## [1] "Comedy|Mystery|Thriller" "Comedy|Thriller"
## [3] "Comedy"                  "Comedy"
## [5] "Comedy|Drama|Romance"    "Action|Children|Comedy"
```

```
head(edx$genres[edx$Romance])
```

```
## [1] "Drama|Romance"           "Action|Romance|Thriller"
## [3] "Comedy|Drama|Romance"    "Action|Drama|Romance"
## [5] "Comedy|Drama|Romance"    "Comedy|Drama|Romance"
```

Lastly, we can extract the movie release year from the title using regular expressions.

```
edx <- edx %>%
  mutate(year = str_replace_all(str_extract(title, "\\(\\d{4}\\)$"), "\\(|\\)", "")) %>%
  mutate(year = as.numeric(year))

validation <- validation %>%
  mutate(year = str_replace_all(str_extract(title, "\\(\\d{4}\\)$"), "\\(|\\)", "")) %>%
  mutate(year = as.numeric(year))
```

2. Exploratory Data Analysis

We will now explore the training dataset created above (edx) in order to provide some insight into the dimensionality of the dataset and the spread of relevant features, steps that will be useful in designing our different machine learning models in the following subsections.

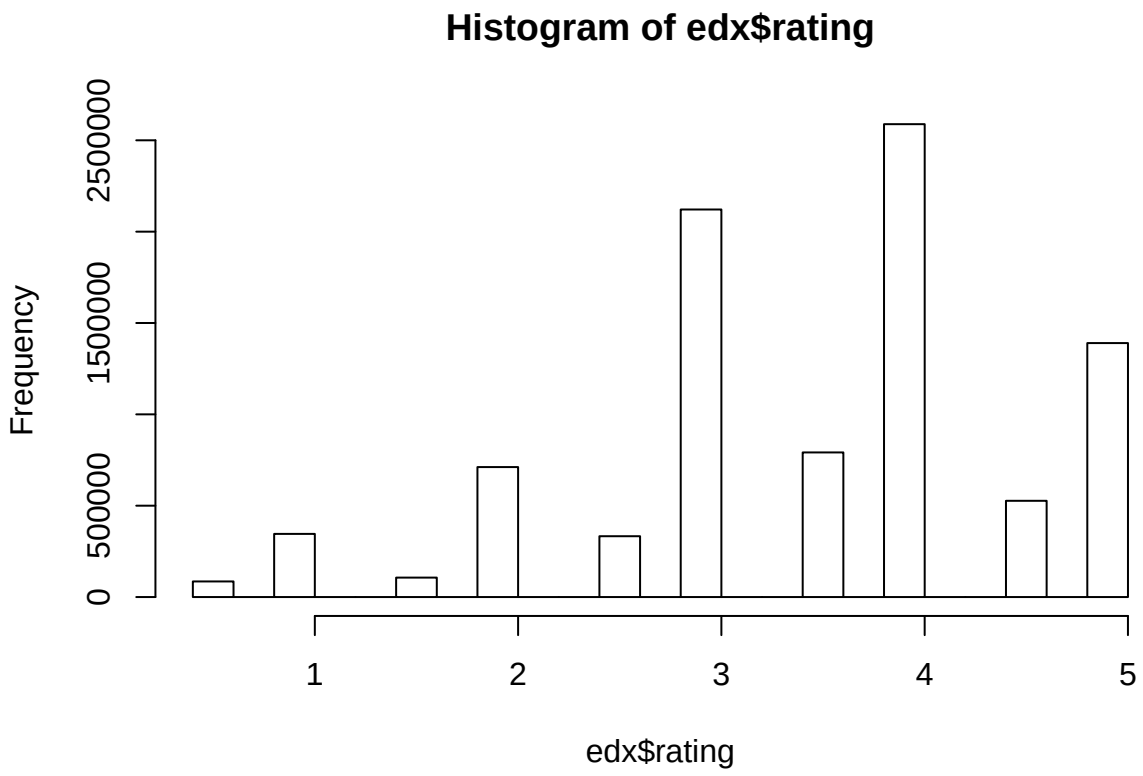
```
library(tidyverse)

# dimensions
paste("Rows:", nrow(edx),", Columns:", ncol(edx))
```

```
## [1] "Rows: 9000055 , Columns: 26"
```

```
# ratings
paste("Number of zeros:", sum(edx$rating == 0.0), ", Number of threes:", sum(edx$rating == 3.0))
```

```
## [1] "Number of zeros: 0 , Number of threes: 2121240"
```

```
hist(edx$rating)
```




```
# numbers of distinct features
paste("Number of unique movies:", length(unique(edx$movieId)), ", Number of unique users:", length(uniq
```

```
## [1] "Number of unique movies: 10677 , Number of unique users: 69878"
```

```
# ratings per genre
paste("Drama:", length(grep("Drama", edx$genres)), "Comedy:", length(grep("Comedy", edx$genres)),
"Thriller:", length(grep("Thriller", edx$genres)), "Romance:", length(grep("Romance", edx$genres)))
```

```
## [1] "Drama: 4151718 Comedy: 2962038 Thriller: 1485456 Romance: 1312948"
```

```
# ratings per movie
edx %>%
  group_by(movieId) %>%
  summarize(title = title, n = n()) %>%
  distinct() %>%
  ungroup() %>%
  arrange(desc(n)) %>%
  head()
```

```
## `summarise()` regrouping output by 'movieId' (override with `.groups` argument)
```

```
## # A tibble: 6 x 3
##   movieId title                                n
##     <dbl> <chr>                            <int>
## 1     296 Patton (1970)                    31362
## 2     356 Amityville Horror, The (1979)    31079
## 3     593 Blue in the Face (1995)          30382
## 4     480 Guantanamera (1994)              29360
## 5     318 Being There (1979)               28015
## 6     110 American President, The (1995)   26212
```

```
# most common ratings
edx %>%
  group_by(rating) %>%
  summarize(n = n()) %>%
  distinct() %>%
  ungroup() %>%
  arrange(desc(n)) %>%
  head()
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 6 x 2
##   rating       n
##    <dbl>   <int>
## 1    4   2588430
## 2    3   2121240
## 3    5   1390114
## 4    3.5  791624
## 5    2    711422
## 6    4.5  526736
```

```
# half-star vs. whole-star ratings
edx %>%
  mutate(star = rating %% 1) %>%
  group_by(star) %>%
  summarize(n = n())
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 2
##    star       n
##   <dbl>   <int>
## 1   0   7156885
## 2   0.5 1843170
```

3. Machine Learning Models

A comment on dimensionality reduction

We will explore a number of different machine learning models in order to predict rating based on movie and user combinations. An initial assumption is required to select the features (columns) which will ultimately be used to train our different models from the available set of columns. The features present in the dataset are: userId, movieId, rating, timestamp, title, and the available genres. We can likely guess that title and timestamp shouldn't provide much excess information to our model, since movieId captures the former, and as long as there are no temporal trends in the data, timestamp should not matter. However, we will still consider it to see if it could reduce our error even slightly. In a larger dataset with many more features, such as those found in genomics contexts, methods such as principal component analysis (PCA) or surrogate

variable analysis (SVA) can be used to select features or identify batch effects, respectively. As a simpler check for our earlier assumption that date provides little useful information, we will compute the correlation between the timestamp column and the rating column. We will also group the dates by week and plot the average to further confirm the lack of a significant time effect.

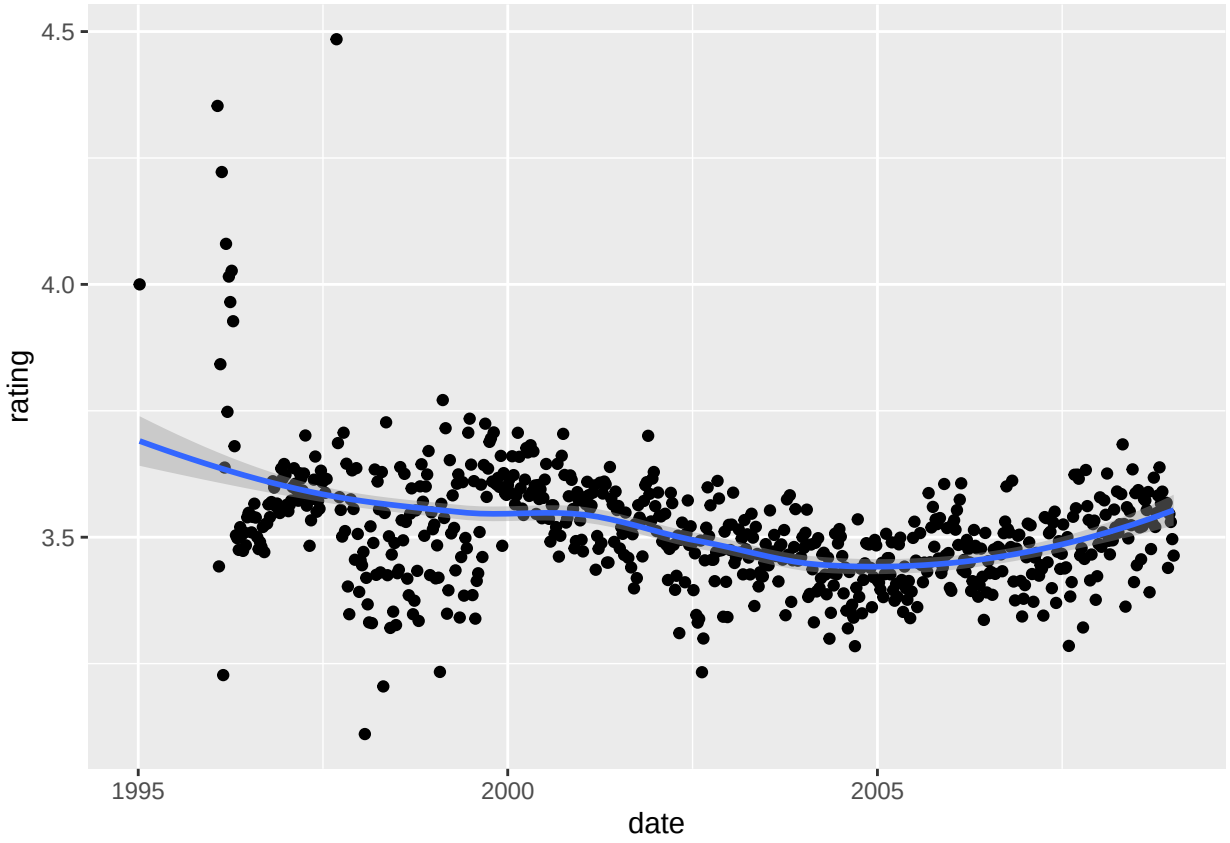
```
cor(edx$timestamp, edx$rating)
```

```
## [1] -0.03473968
```

```
edx %>% mutate(date = round_date(as_datetime(timestamp), unit = "week")) %>%
    group_by(date) %>%
    summarize(rating = mean(rating)) %>%
    ggplot(aes(date, rating)) +
    geom_point() +
    geom_smooth()
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



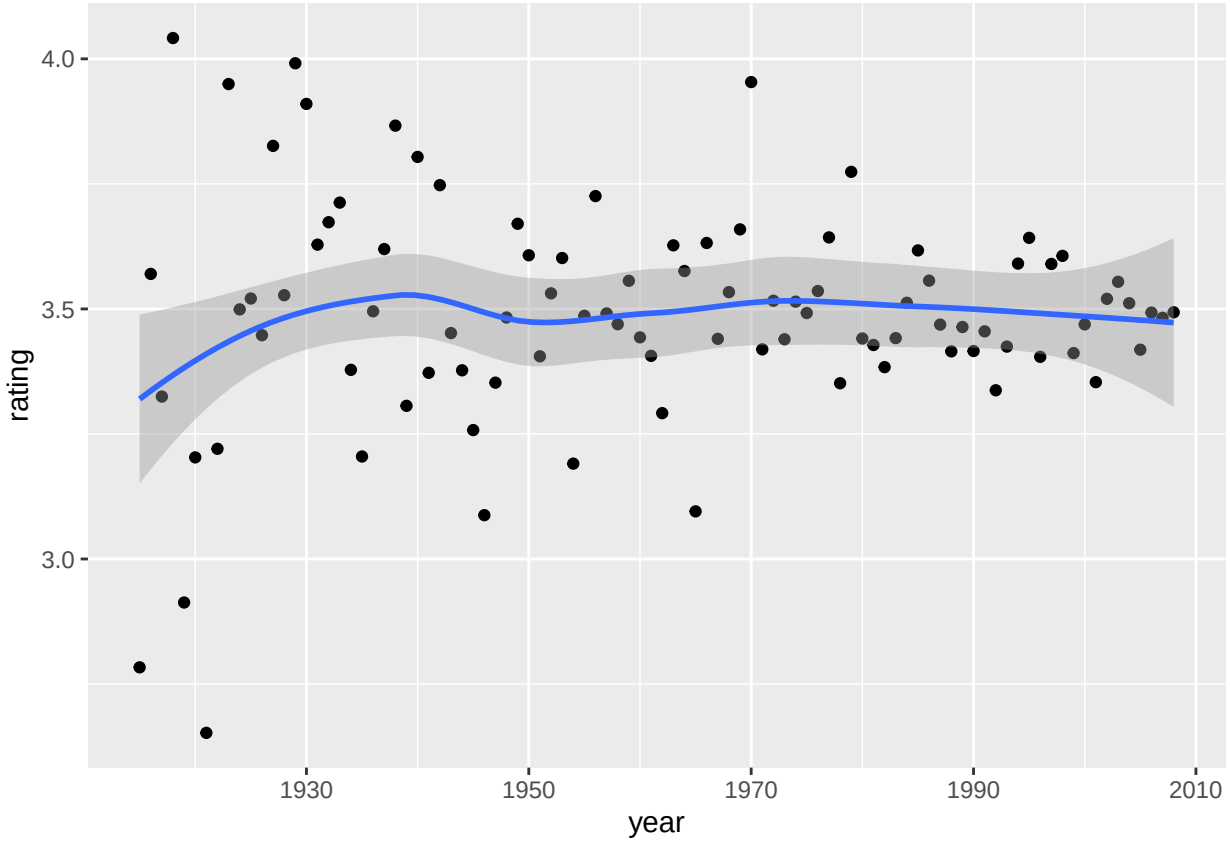
We can perform a similar test for the release year variable that we extracted from the title. Our results are similar as well- the correlation is extremely small though there is some slight trend in the data, especially for very old movies.

```
idx <- which(!is.na(edx$year))
cor(edx$year[idx], edx$rating[idx])
```

```
## [1] -0.00765108
```

```
edx %>%
    group_by(year) %>%
    summarize(rating = mean(rating)) %>%
    ggplot(aes(year, rating)) +
    geom_point() +
    geom_smooth()
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
## Warning: Removed 1 rows containing non-finite values (stat_smooth).
## Warning: Removed 1 rows containing missing values (geom_point).
```



We will test a number of different machine learning algorithms on the edx dataset in order to identify that which provides us with the best predictive power. Due to aforementioned computational constraints, we restrict ourselves to the construction of linear regression models with and without regularization, though we also briefly discuss other available algorithms that require more computational power.

Linear Regression

We will begin with the simplest model (outside of random guessing)- the only parameters we will first consider are effects due to movieId (i) and userId (u), along with the mean rating, that is:

$$r_{i,j} = \mu + b_i + b_j + \epsilon$$

The large size of our dataset is problematic for R's builtin lm() function, since it requires allocation of a several GB large vector. Instead, using knowledge of linear regression, we can compute the movie-specific and user-specific predictors as follows.

```
mu <- mean(edx$rating)
movie_avg <- edx %>%
  group_by(movieId) %>%
  summarize(b_i = mean(rating - mu))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
user_avg <- edx %>%
  left_join(movie_avg, by='movieId') %>%
  group_by(userId) %>%
  summarize(b_u = mean(rating - mu - b_i))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

Each predictive factor for the movieId is simply the mean rating of all of the entries of that factor, and the userId effect is the mean of the residuals grouped by user. We can now predict our results to obtain an estimate for the accuracy on the training set. We could also round the final predictions to the nearest 0.5, since these are the only values permitted in the ratings, but we instead quantify the RMSE without this rounding (which is commented out).

```
predictions <- edx %>%
  left_join(movie_avg, by='movieId') %>%
  left_join(user_avg, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  pull(pred)
# predictions <- round(predictions * 2)/2
sqrt(mean((predictions-edx$rating)^2))
```

```
## [1] 0.8567039
```

Thus, we see that this linear model has an accuracy of 85.7%. To try to improve on this, we can add additional factors to the model, including genre specificity. The new model is now:

$$r_{i,j} = \mu + b_i + b_j + \Sigma_{g \in genres} b_g x_i + \epsilon$$

Here, we sum over all genres, including the genre factor for the given genre only if the movie falls in that genre, i.e. x_i is 1 if and only if x is in genre g, and is 0 otherwise.

```
b_g = numeric(length(genre))
for (i in 1:length(genre)) {
  idx <- which(edx[[genre[i]]])
  movs <- edx[idx,]
  genre_effect <- movs %>%
    select(Action:Western)
  genre_effect <- as.matrix(genre_effect) %*% b_g
  resid <- movs %>%
    left_join(movie_avg, by='movieId') %>%
    left_join(user_avg, by='userId') %>%
    mutate(resid = rating - mu - b_i - b_u) %>%
    pull(resid)
  resid <- resid - genre_effect
  b_g[i] = mean(resid)
}
```

We can then compute the predictions as before, including the new genre information.

```
predictions <- edx %>%
  left_join(movie_avg, by='movieId') %>%
  left_join(user_avg, by='userId') %>%
  mutate(pred = mu + b_i + b_u) %>%
  pull(pred)
genre_effect <- edx %>%
  select(Action:Western)
genre_effect <- as.matrix(genre_effect) %*% b_g
# predictions <- round(2*(predictions + genre_effect))/2
predictions <- predictions + genre_effect
sqrt(mean((predictions-edx$rating)^2))
```

```
## [1] 0.8566902
```

As another factor, we can add in the date to determine if this has any influence. This model is:

$$r_{i,j} = \mu + b_i + b_j + \Sigma_{g \in genres} b_g x_i + b_d + \epsilon$$

Now, b_d accounts for the week at which the review was submitted.

```
genre_effect <- edx %>%
  select(Action:Western)
genre_effect <- as.matrix(genre_effect) %*% b_g

date_effect <- edx %>%
  left_join(movie_avg, by='movieId') %>%
  left_join(user_avg, by='userId') %>%
  mutate(date = round_date(as_datetime(timestamp), "week"), genre_effect = genre_effect, resid = rating
  group_by(date) %>%
  summarize(d = mean(resid))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
predictions <- edx %>%
  left_join(movie_avg, by='movieId') %>%
  left_join(user_avg, by='userId') %>%
  mutate(date = round_date(as_datetime(timestamp), "week")) %>%
  left_join(date_effect, by = "date") %>%
  mutate(pred = mu + b_i + b_u + d) %>%
  pull(pred)
genre_effect <- edx %>%
  select(Action:Western)
genre_effect <- as.matrix(genre_effect) %*% b_g
# predictions <- round(2 * (predictions + genre_effect))/2
predictions <- predictions + genre_effect
sqrt(mean((predictions-edx$rating)^2))
```

```
## [1] 0.8565851
```

A final factor we can add to our model is the release year:

$$r_{i,j} = \mu + b_i + b_j + \Sigma_{g \in genres} b_g x_i + b_d + b_y + \epsilon$$

```
year_effect <- edx %>%
  left_join(movie_avg, by='movieId') %>%
```

```
  left_join(user_avg, by='userId') %>%
  mutate(date = round_date(as_datetime(timestamp), "week")) %>%
  left_join(date_effect, by = "date") %>%
  mutate(genre_effect = genre_effect, resid = rating - genre_effect - mu - b_i - b_u - d) %>%
  group_by(year) %>%
  summarize(y = mean(resid))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
predictions <- edx %>%
  left_join(movie_avg, by='movieId') %>%
  left_join(user_avg, by='userId') %>%
  mutate(date = round_date(as_datetime(timestamp), "week")) %>%
  left_join(date_effect, by = "date") %>%
  left_join(year_effect, by = "year") %>%
  mutate(pred = mu + b_i + b_u + d + y) %>%
  pull(pred)

# predictions <- round(2 * (predictions + genre_effect))/2
predictions <- predictions + genre_effect
sqrt(mean((predictions - edx$rating)^2))
```

```
## [1] 0.8563559
```

Regularization

While use of linear regression allows us to get relatively low errors, it is possible to improve. Regularization is a widely used technique in machine learning that seeks to address the tendency of models to overfit the training data. Adding a regularization term to the loss function reduces this overfitting by penalizing very large estimates. We introduce regularization to our final model from above, including user, movie, genre, year, and date effects, as it performed the best on the training data. We scan over a range of lambda parameters to optimize this regularization. It is worth noting that adding in cross-validation to this function could potentially improve our estimation of lambda, but would require further partitioning the training data and becomes substantially more computationally intensive, so we do not perform it here.

```
# function that re-estimates all parameters for a given lambda
# lambda is added to the length of the vector in the denominator during estimation

est_RMSE <- function(lambda) {
  paste("Evaluating lambda =", lambda)
    movie_avg_r <- edx %>%
      group_by(movieId) %>%
      summarize(b_i = sum(rating - mu)/(n() + lambda))
    user_avg_r <- edx %>%
      left_join(movie_avg_r, by='movieId') %>%
      group_by(userId) %>%
      summarize(b_u = sum(rating - mu - b_i)/(n() + lambda))
    b_g_r = numeric(length(genre))
    for (i in 1:length(genre)) {
      idx <- which(edx[[genre[i]]])
      movs <- edx[idx,]
      genre_effect <- movs %>%
        select(Action:Western)
      genre_effect <- as.matrix(genre_effect) %*% b_g_r
      resid <- movs %>%
```

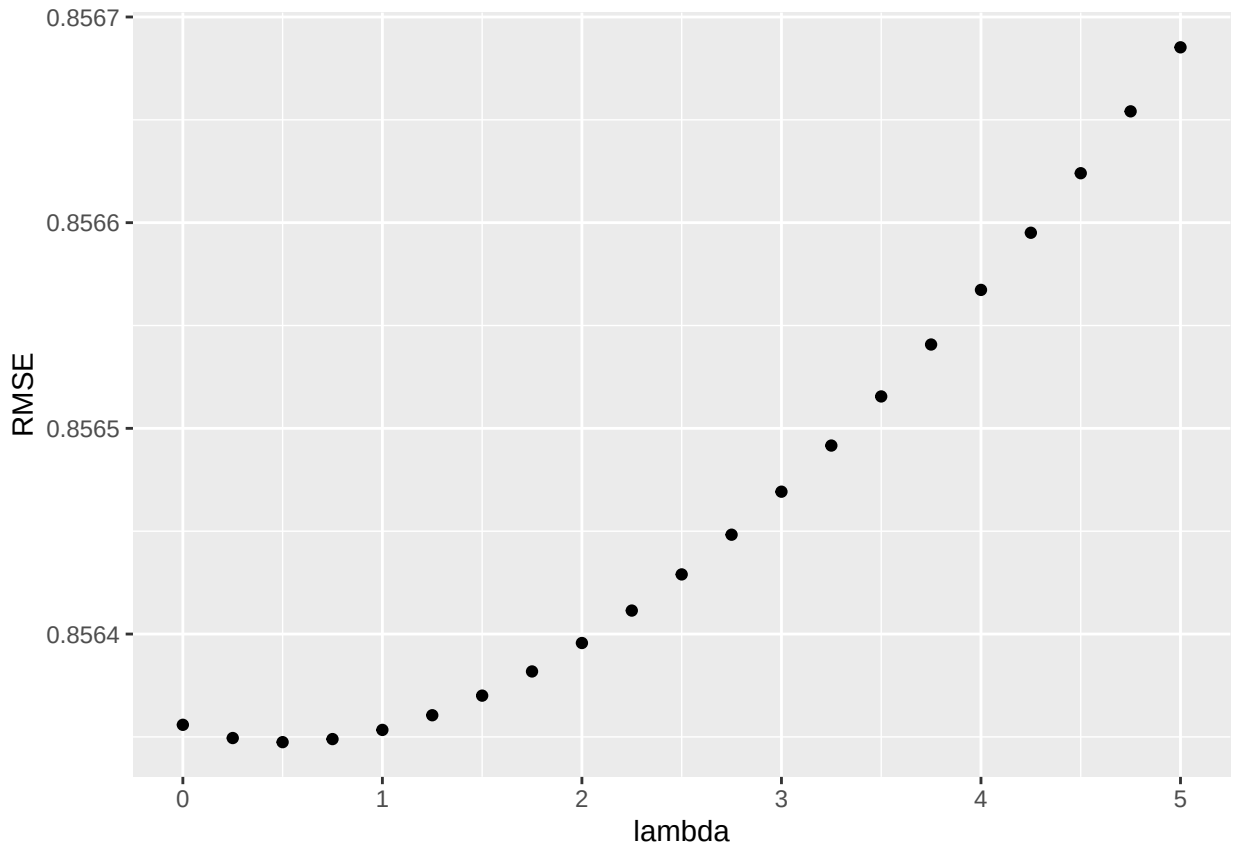
```
      left_join(movie_avg_r, by='movieId') %>%
      left_join(user_avg_r, by='userId') %>%
      mutate(resid = rating - mu - b_i - b_u) %>%
      pull(resid)
    resid <- resid - genre_effect
    b_g_r[i] = sum(resid)/(length(resid) + lambda)
  }
  genre_effect <- edx %>%
    select(Action:Western)
  genre_effect <- as.matrix(genre_effect) %*% b_g_r
  date_effect_r <- edx %>%
    left_join(movie_avg_r, by='movieId') %>%
    left_join(user_avg_r, by='userId') %>%
    mutate(date = round_date(as_datetime(timestamp), "week"), genre_effect = genre_effect, resid = ra
    group_by(date) %>%
    summarize(d = sum(resid)/(n() + lambda))
  year_effect_r <- edx %>%
    left_join(movie_avg_r, by='movieId') %>%
    left_join(user_avg_r, by='userId') %>%
    mutate(date = round_date(as_datetime(timestamp), "week")) %>%
    left_join(date_effect_r, by = "date") %>%
    mutate(genre_effect = genre_effect, resid = rating - genre_effect - mu - b_i - b_u - d) %>%
    group_by(year) %>%
    summarize(y = sum(resid)/(n() + lambda))

  predictions <- edx %>%
    left_join(movie_avg_r, by='movieId') %>%
    left_join(user_avg_r, by='userId') %>%
    mutate(date = round_date(as_datetime(timestamp), "week")) %>%
    left_join(date_effect_r, by = "date") %>%
    left_join(year_effect_r, by = "year") %>%
    mutate(pred = mu + b_i + b_u + d + y) %>%
    pull(pred)

  # predictions <- round(2 * (predictions + genre_effect))/2
  predictions <- predictions + genre_effect
  sqrt(mean((predictions - edx$rating)^2))
}

# plot errors vs. lambdas and extract best estimate
lambdas <- seq(0, 5, 0.25)
RMSEs <- sapply(lambdas, est_RMSE)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
df <- data.frame(lambda = lambdas, RMSE = RMSEs)
ggplot(data = df, aes(lambda, RMSE)) + geom_point()
```




```
lambda_best <- lambdas[which.min(RMSEs)]
```

Other Models

We will now consider how other models could be used to accomplish this task, but will not execute the code since not enough RAM is available for execution in a short enough period of time. For all models, we will configure the parameters to use 5-fold cross-validation with the caret package to select the optimal parameters and tune the grid of available hyperparameters.

```
# set control parameters for cross-validation
# Control <- trainControl(method = "cv", number = 5, p = .9, returnData = FALSE, trim = TRUE)
# use range of complexity parameters (cp)
# cp <- seq(0, 0.05, 0.005)
# fit_rpart <- train(rating ~ movieId + userId, data = edx, method = "rpart", tuneGrid = data.frame(cp =
#
# plot results, extract best cp, view confusion matrix, plot final results
# plot(fit_rpart)
# cp[which.max(fit_rpart$results$Accuracy)]
# predictions_rpart <- predict(fit_rpart, validation)
# confusionMatrix(predictions_rpart, validation$rating)$overall[["Accuracy"]]
# plot(fit_rpart$finalModel, margin = 0.1)
# text(fit_rpart$finalModel, cex = 0.75)
```

Related to the classification tree is the random forest, which we will explore next.

```
# scan over a range of number of variables available to split at each node
# mtry <- seq(1, 7, 2)
#
# fit model with 50 trees per forest
# fit_rf <- train(rating ~ movieId + userId, data = edx, method = "rf", tuneGrid = data.frame(mtry = mt
#
# plot results, extract best parameters, and examine confusion matrix and variable importance
# plot(fit_rf)
# mtry[which.max(fit_rf$results$Accuracy)]
# predictions_rf <- predict(fit_rf, validation)
# confusionMatrix(predictions_rf, validation$rating)$overall[["Accuracy"]]
# varImp(fit_rf)
```

Next, we'll investigate the use of a k-nearest neighbors model. Note that this is an extremely slow model so would take even more power to run

```
# fit_knn <- train(rating ~ movieId + userId, data = edx, method = "knn", trControl = Control, tuneGrid
```

Finally, we will examine a method that has become a favorite for recommender systems, matrix factorization. We will accomplish this using the recosystem package. But first, we need to process are data a bit. Matrix factorization seeks to decompose the input into a product of two matrices of users and items (movies). Thus, rather than have our long input format used above, where each row contains a single user-movie pair, we'll instead widen our data such that a given user only has one row and each movie has its own column with the rating found at the (userId, movieId) location. This can be done with the spread function in dplyr, but once again, we run into memory issues dealing with the extremely large size of the resultant matrix (5.6 GB), so we once again do not execute these commands.

```
# edx_wide <- edx %>%
#   select(userId, movieId, rating) %>%
#   spread(movieId, rating)
#
# r = Reco()
#
# opts = r$tune(edx_wide, opts = list(dim = c(10, 20, 30), lrate = c(0.1, 0.2), costp_l1 = 0, costq_l1 =
```

```
#
# r$train(edx_wide, opts = c(opts$min, nthread = 1, niter = 20))
#
# predictions <- r$predict()
```

Final Validation

With our model selected, we will now validate our regularized linear regression model using all available features on the left-out validation set to quantify our final results. We will first have to re-estimate the coefficients with the previously determined optimal lambda value.

```
lambda = lambda_best
movie_avg_r <- edx %>%
      group_by(movieId) %>%
      summarize(b_i = sum(rating - mu)/(n() + lambda))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
    user_avg_r <- edx %>%
      left_join(movie_avg_r, by='movieId') %>%
      group_by(userId) %>%
      summarize(b_u = sum(rating - mu - b_i)/(n() + lambda))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
    b_g_r = numeric(length(genre))
    for (i in 1:length(genre)) {
      idx <- which(edx[[genre[i]]])
      movs <- edx[idx,]
      genre_effect <- movs %>%
        select(Action:Western)
      genre_effect <- as.matrix(genre_effect) %*% b_g_r
      resid <- movs %>%
        left_join(movie_avg_r, by='movieId') %>%
        left_join(user_avg_r, by='userId') %>%
        mutate(resid = rating - mu - b_i - b_u) %>%
        pull(resid)
      resid <- resid - genre_effect
      b_g_r[i] = sum(resid)/(length(resid) + lambda)
    }
    genre_effect <- edx %>%
      select(Action:Western)
    genre_effect <- as.matrix(genre_effect) %*% b_g_r
    date_effect_r <- edx %>%
      left_join(movie_avg_r, by='movieId') %>%
      left_join(user_avg_r, by='userId') %>%
      mutate(date = round_date(as_datetime(timestamp), "week"), genre_effect = genre_effect, resid = ra
      group_by(date) %>%
      summarize(d = sum(resid)/(n() + lambda))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
    year_effect_r <- edx %>%
      left_join(movie_avg_r, by='movieId') %>%
      left_join(user_avg_r, by='userId') %>%
      mutate(date = round_date(as_datetime(timestamp), "week")) %>%
```

```
      left_join(date_effect_r, by = "date") %>%
      mutate(genre_effect = genre_effect, resid = rating - genre_effect - mu - b_i - b_u - d) %>%
      group_by(year) %>%
      summarize(y = sum(resid)/(n() + lambda))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

Now we'll make our predictions on the validation set and quantify our final results.

```
genre_effect_v <- validation %>%
     select(Action:Western)
   genre_effect_v <- as.matrix(genre_effect_v) %*% b_g_r
predictions <- validation %>%
     left_join(movie_avg_r, by='movieId') %>%
     left_join(user_avg_r, by='userId') %>%
     mutate(date = round_date(as_datetime(timestamp), "week")) %>%
     left_join(date_effect_r, by = "date") %>%
     left_join(year_effect_r, by = "year") %>%
     mutate(pred = mu + b_i + b_u + d + y) %>%
     pull(pred)

   # predictions <- round(2 * (predictions + genre_effect))/2
   predictions <- predictions + genre_effect_v
   sqrt(mean((predictions - validation$rating)^2))
```

```
## [1] 0.8648923
```

With this model, we obtain a final validation error of 0.86489. Certainly, further steps could be taken to improve this accuracy, including the use of cross-validation within the edx set during parameter optimization, use of a finer lambda grid, and use of different, more computationally intense models.

Conclusion

In this study, we have examined the 10M MovieLens dataset and constructed a regularized regression algorithm to predict ratings for user-movie ratings pairs. The model can be trained relatively quickly and is capable of achieving an accuracy below 0.86490 on a held-out validation dataset. We began by extracting data from downloaded files, parsing strings to extract release years and genres, and building a matrix with numerous features that can be used to improve accuracy. We performed some exploratory data analysis to get a feeling for the spread of our data, the number and distribution of features, and how useful each feature may be in predicting rating. We built a regression model beginning with a simple userId and movieId-dependent prediction and included genre, date, and release year information to marginally improve accuracy. We then included regularization and optimzed the lambda value to further reduce error on the training data. We briefly discussed some more advanced machine learning algorithms that could be used to solve these problems, but were computationally unfeasible due to the large number of observations (9 million) in our dataset. These include k-nearest neighbors, classification and regression trees, random forests, and matrix factorization. Other algorithms exist as well, including neural networks, Bayesian classifiers, and support vector machines. Given higher computational power, any one of these may outperform our regression scheme, but are out of the scope of this project.