Restaurant Rating Prediction

Mike Gessner

5/20/2019

Introduction and background

The objective of this study is to predict the rating of a restaurant by a user in the restaurant and consmer data set from UCI. The dataset is avialable at the following URL: https://archive.ics.uci.edu/ml/datasets/Restaurant+%26+consumer+data. I used the rating_final.csv file which contains 1,161 observations. The goal is to minimize the root mean square error (RMSE) through a variety of machine learning techniques. RMSE is calculated as the average difference between the actual rating and predicted rating squared, and then take the square root, or in r:

```
RMSE = function(true_ratings, predicted_ratings){
        sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

The following steps are performed to meet the stated goal. First, we load the necessary libraries and data. Next, we explore the data structure and the data itself to look for any patterns. The final step is to work through some models to see which meet or exceed our stated goal. We start with some simple models and then progress there to regularization, then to recommender models, and finally matrix factorization. While this is not an exhaustive examination of potential models, most will meet the stated goal. In the end, we provide a summary of the results of the modeling.

The first step is to load the necessary libraries.

```
#knitr::opts_chunk$set()
library(caret)
library(caTools)
library(tidyverse)
library(data.table)
library(h2o)
library(recosystem)
library(recommenderlab)
library(ggplot2)
library(kableExtra)
```

Load the data

In this section, we load the data for the project. The code reads in the data and ultimately splits the data into a training set and validation set (80% and 20%, respectively) using the sample.split function. In addition, there is one data transformation done to the userID variable. In the original data, the user is identified with a 'U' and 4 numbers. The transformation drops the 'U' from the variable and converts the userID to a numeric. That is the extent of the data transformation for this data.

```
#read in the data
restaurant_cons = read.csv('~/Documents/edx_courses/harvard_capstone/RCdata/rating_final.csv')

#data transformation for userID, to drop the U and make numeric
restaurant_cons$userID = substring(restaurant_cons$userID, 2,5)
restaurant_cons$userID = as.numeric(restaurant_cons$userID)
```

```
#will need to partition the data into a training set and validation (test) set
#split 80% training, 20% test/validation

set.seed(35)
sample = sample.split(restaurant_cons, SplitRatio = .8)
train_set = subset(restaurant_cons, sample==TRUE)
test_set = subset(restaurant_cons, sample==FALSE)
```

Exploring the data

Next, we take a look at the data in the training set (train_set).

```
#take a look at the train_set
glimpse(train_set)
```

```
## Observations: 929
## Variables: 5
## $ userID         <dbl> 1077, 1077, 1077, 1068, 1068, 1068, 1068, 1067,...
## $ placeID        <int> 135085, 135038, 132825, 135104, 132740, 132663,...
## $ rating         <int> 2, 2, 2, 1, 0, 1, 0, 2, 1, 1, 1, 1, 1, 0, 1, 1,...
## $ food_rating    <int> 2, 2, 2, 1, 0, 1, 0, 2, 1, 2, 0, 0, 2, 0, 2, 2,...
## $ service_rating <int> 2, 1, 2, 2, 0, 1, 0, 2, 1, 2, 1, 0, 1, 2, 0, 2,...
```

The dataset we start with has 5 variables and 929 observations. Each observation is for a user who rates a restaurant (0, 1, or 2). Additional information include the food and service rating (same scale as rating). Our variable of interest for the project is the rating variable.We want to predict the rating variable given some information (e.g., userID and placeID).

In particular, the userID variable is a unique identifier for a user. The placeID variable is also a unique identifier, but for a restaurant. The outcome variable is the rating. The value of the rating can be bvetween 0 and 2 (whole numbers). Additionally, there is a food rating and service rating variable

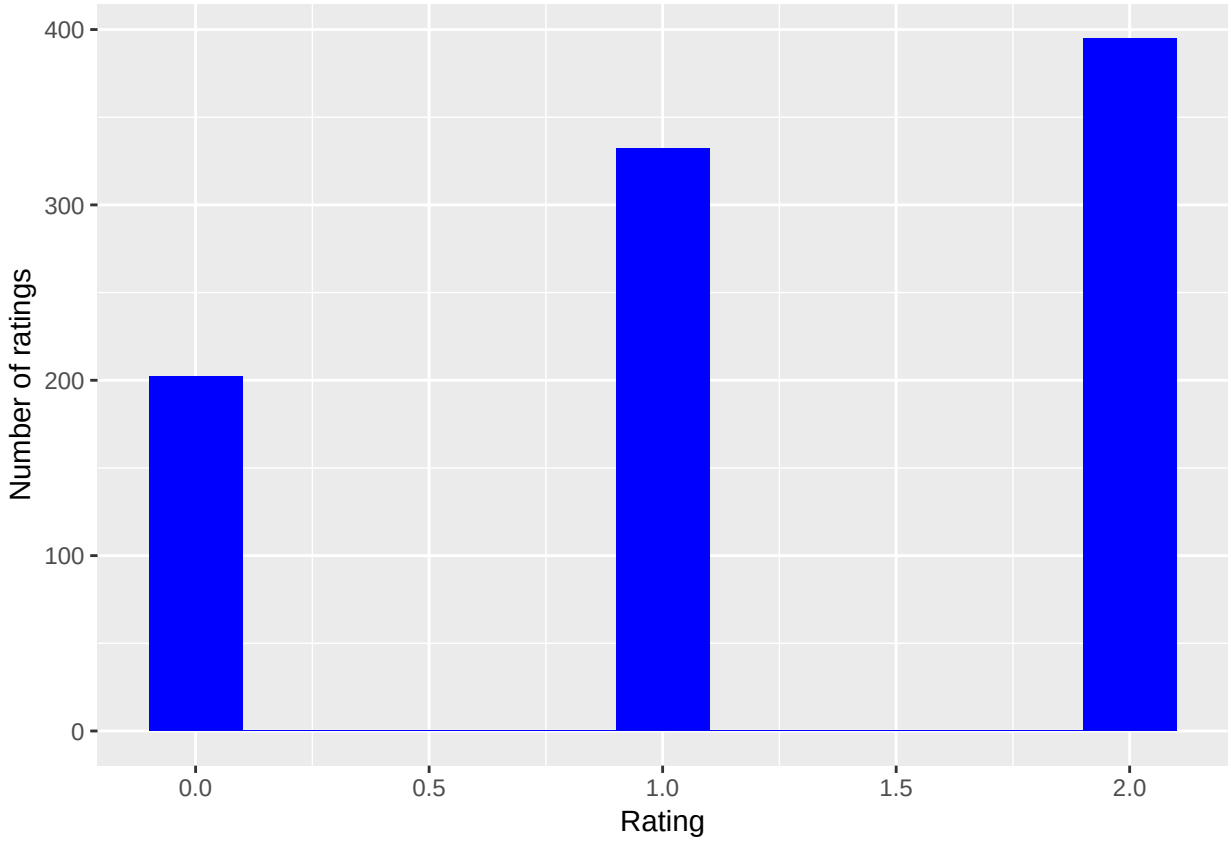
Now, let's take a look at the data.

```
summary(train_set)
```

```
##      userID       placeID           rating       food_rating
##  U1106  : 15   Min.   :132560   Min.   :0.000   Min.   :0.00
##  U1061  : 14   1st Qu.:132856   1st Qu.:1.000   1st Qu.:1.00
##  U1134  : 13   Median :135032   Median :1.000   Median :1.00
##  U1024  : 12   Mean   :134204   Mean   :1.208   Mean   :1.21
##  U1003  : 11   3rd Qu.:135059   3rd Qu.:2.000   3rd Qu.:2.00
##  U1016  : 11   Max.   :135109   Max.   :2.000   Max.   :2.00
##  (Other):853
##  service_rating
##  Min.   :0.000
##  1st Qu.:0.000
##  Median :1.000
##  Mean   :1.094
##  3rd Qu.:2.000
##  Max.   :2.000
##
```

```
#take a look at the total rating (variable of interest)
train_set %>% ggplot(aes(x=rating)) +
```

```
        geom_histogram(binwidth = .2, fill='blue') +
        labs(x = 'Rating', y = 'Number of ratings')
```



We can see from the histogram of ratings in the data that there of the three ratings possible, a 2 is most common and 0 is the least common.

We can also create a table of the top restaurants in the dataset.

```
top_restaurant = train_set %>%
        group_by(placeID) %>%
        summarize(count = n()) %>%
        top_n(20, count) %>%
        arrange(desc(count))

top_restaurant
```

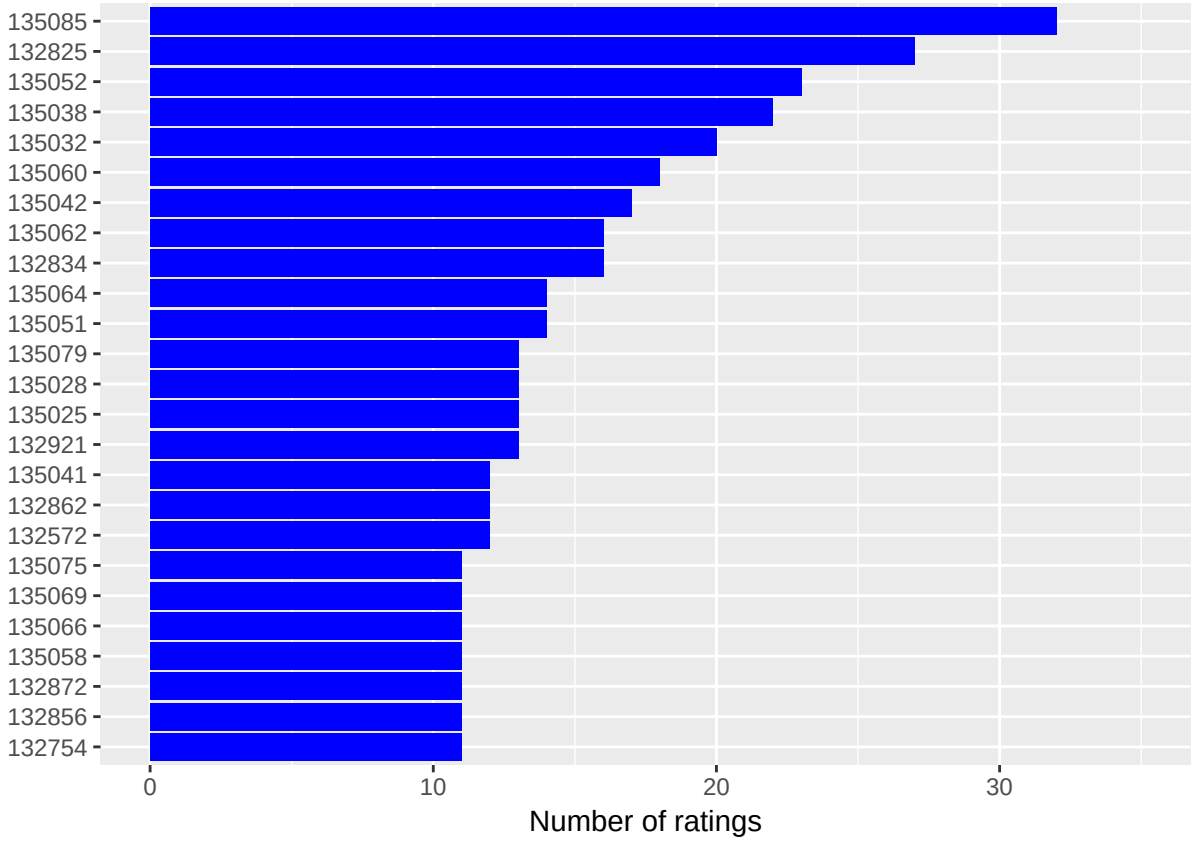
```
## # A tibble: 25 x 2
##    placeID count
##      <int> <int>
##  1  135085    32
##  2  132825    27
##  3  135052    23
##  4  135038    22
##  5  135032    20
##  6  135060    18
##  7  135042    17
##  8  132834    16
##  9  135062    16
```

```
## 10  135051    14
## # ... with 15 more rows
```

The table shows the top restaurants. As the table shows, the restaurant with the most ratings has 32.

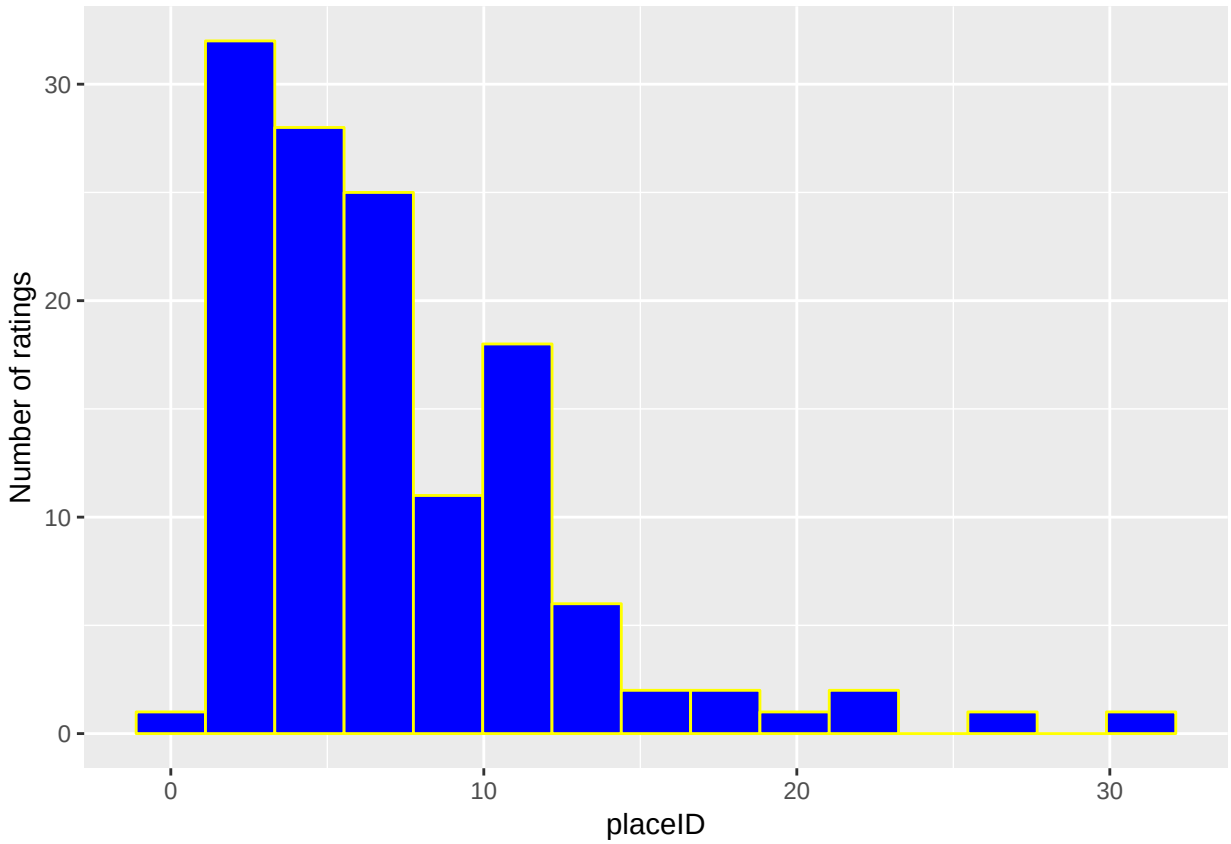
We can also look at the most popular restaurants by the number of ratings in a graph.

```
#graph of top restaurants
top_restaurant %>%
        ggplot(aes(x=reorder(placeID, count), y=count)) +
        geom_bar(stat='identity', fill='blue') +
        coord_flip(y=c(0,35)) +
        labs(x = "", y='Number of ratings')
```



Now, let's see the distribution of the number of ratings by restaurant.

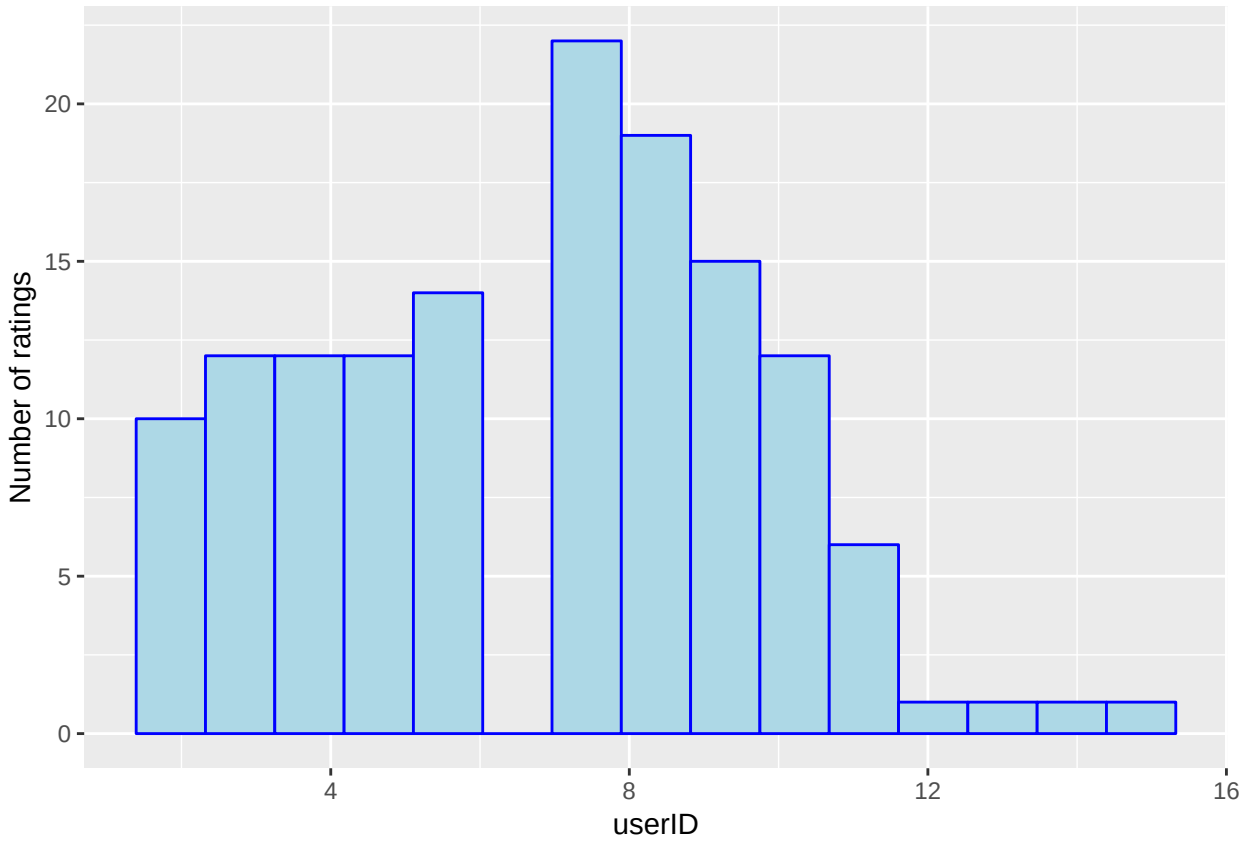
```
#histogram of number ratings by placeID
train_set %>% count(placeID) %>%
        ggplot(aes(n)) +
        geom_histogram(bins=15, color='yellow', fill='blue') +
        labs(x = 'placeID',
             y = 'Number of ratings')
```



The histogram shows that most restaurants have fewer than 10 reviews. There are some outliers that have more than 20 reviews.

Let's also examine the number of ratings by userID.

```
#histogram of ratings by userID
train_set %>% count(userID) %>%
        ggplot(aes(n)) +
        geom_histogram(bins=15, color='blue', fill='lightblue') +
        labs(x = 'userID',
             y = 'Number of ratings')
```



What can we tell from the two histograms? We can see that some restaurants get rated more than others (also from the bar plot above) and that some users are more active in rating restaurants than others. This indicates that there may be some restaurants and user effects for ratings.

Modeling and results

We start with the simple approach using regression models. I'll ignore the basic approach of using just the average rating in the dataset to predict ratings.

Restaurant effects

The first step is to create mu, the average of all ratings in the data. This simple modeling approach uses the average rating in the data and accounts for restaurant specific effects (b_i). In other words, the predicted rating is a function of the average rating in the data plus a restaurant-specific effect (and error).

```
#calculate the mean rating first
mu = mean(train_set$rating)
```

We find mu to be 1.2077503.

Then, calculate b_i of the training set as the average of the difference between the restaurant's rating and the average rating in the data:

```
#calculate b_i for training set
rest_avgs = train_set %>%
        group_by(placeID) %>%
        summarize(b_i = mean(rating - mu))
```

Next, use the above to calculate the predicted ratings

```
#predict ratings
predicted_ratings_bi = mu + test_set %>%
        left_join(rest_avgs, by='placeID') %>%
        .$b_i
```

Restaurant and user effects

Now onto the restaurant and user effects. In this case, we also account for user effects in the model (b_u). The histogram showing the ratings and users does show differences by users. So let's account for those effects.

Like above, this model adds an adjustment for the user, so that a predicted rating is a function of the average rating in the data, restaurant effects, and user effects. The code is very similar to the restaurant effect (b_i) above.

```
 #restaurant and user effect
user_avgs = train_set %>%
        left_join(rest_avgs, by='placeID') %>%
        group_by(userID) %>%
        summarize(b_u = mean(rating - mu - b_i))

#predicted ratings b_i, b_u
predicted_ratings_bu = test_set %>%
        left_join(rest_avgs, by='placeID') %>%
        left_join(user_avgs, by='userID') %>%
        mutate(pred = mu + b_i + b_u) %>%
        .$pred
```

And finally, calculate the RMSE for each model

```
#calculate RMSEs
rmse_model1 = RMSE(test_set$rating, predicted_ratings_bi)

rmse_model2 = RMSE(test_set$rating, predicted_ratings_bu)
```

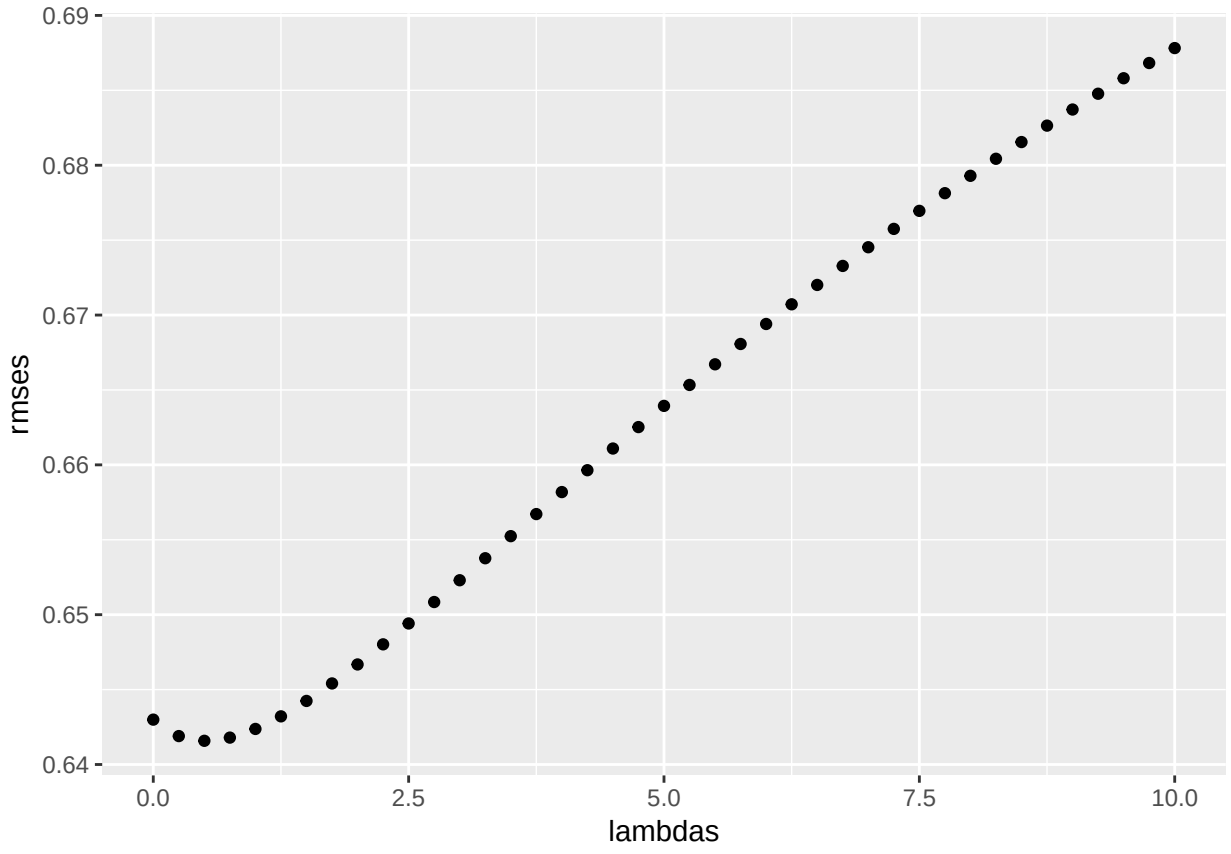
The RMSE for the restaurant effect (model 1) is 0.7982501. While adding in the user effects lowers the RMSE (model 2) to 0.6660479. This clearly shows that adding a user effect to the restaurant effects greatly improves the prediction.

Regularization model

Regularization accounts for small sample sizes by penalizing large estimates that are the result of small samples. This may come in handy as there are some restaurants in the data that don't have a large number of ratings. Essentially, the regularization model is an extension of the above simpler models by adding a regularizing term that helps avoid overfitting a model by penalizing the magnitudes of the restaurant and user effects. If there are a large number of ratings, then lambda is essentially ignored in the model. However, if n is small, then the estimate of the effect (either user or restaurant effects) is reduced towards zero and the predicted rating is closer to the mean rating in the data.

We learned that lambda is a tuning parameter and we can choose the optimal tuning parameter viw cross-validation. That optimal value of lambda is the one that minimizes the RMSE. Let's try that now.

```r
#find tuning parameter for lamnda
lambdas = seq(0, 10, .25)

rmses = sapply(lambdas, function(l) {

        mu_reg = mean(train_set$rating)

        b_i_reg = train_set %>%
                group_by(placeID) %>%
                summarize(b_i_reg = sum(rating - mu_reg) / (n()+1))

        b_u_reg = train_set %>%
                left_join(b_i_reg, by='placeID') %>%
                group_by(userID) %>%
                summarize(b_u_reg = sum(rating - b_i_reg - mu_reg) / (n() + l))

        predicted_ratings_b_i_u = test_set %>%
                left_join(b_i_reg, by='placeID') %>%
                left_join(b_u_reg, by='userID') %>%
                mutate(pred = mu_reg + b_i_reg + b_u_reg) %>%
                .$pred

        return(RMSE(test_set$rating, predicted_ratings_b_i_u))
})

qplot(lambdas, rmses)
```



```
lambda = lambdas[which.min(rmses)]

rmse_model3 = min(rmses)
```

After choosing the optimal lambda, 0.5, we calculate the minimum RMSE. We see that the regularization model has an RMSE of 0.6415814. The new RMSE is better than that of model 2 (user and restaurant effects).

Ensemble models

In this section, I use gradient boosting decision trees and rand forests to predict restaurant ratings. More details about the methods are found below. But first, I must create copies of the training and test sets and create some variables. The first variable is the number of restaurants rated by each user and the other is the number of users that rated each restaurant. Once that is done, I move onto creating the models:

```
#create copy of the train set

train_set.copy = train_set %>%
        select(userID, placeID, rating)

#create new variable that counts number of restaraunts rated by each user
train_set.copy = train_set.copy %>%
        group_by(userID) %>%
        mutate(n.rest_user = n())

#create new variable that counts number user that rated each restaurant
train_set.copy = train_set.copy %>%
        group_by(placeID) %>%
        mutate(n.users_rest = n())

#recast userID and placeID as factors
train_set.copy$userID = as.factor(train_set.copy$userID)
train_set.copy$placeID = as.factor(train_set.copy$placeID)

#do same process for test_set

test_set.copy = test_set %>%
        select(userID, placeID, rating)

#create new variable that counts number of restaraunts rated by each user
test_set.copy = test_set.copy %>%
        group_by(userID) %>%
        mutate(n.rest_user = n())

#create new variable that counts number user that rated each restaurant
test_set.copy = test_set.copy %>%
        group_by(placeID) %>%
        mutate(n.users_rest = n())

#recast userID and placeID as factors
test_set.copy$userID = as.factor(test_set.copy$userID)
test_set.copy$placeID = as.factor(test_set.copy$placeID)
```

Next, I create an h2o instance, then split the training set into a training and test set.

```
h2o.init(
        nthreads = 1,
        max_mem_size='5G'
)
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         5 hours 39 minutes
##     H2O cluster timezone:       America/Los_Angeles
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.22.1.1
##     H2O cluster version age:    4 months and 22 days !!!
##     H2O cluster name:           H2O_started_from_R_jackiechanfan_kin411
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   4.43 GB
##     H2O cluster total cores:    4
##     H2O cluster allowed cores:  1
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
##     R Version:                  R version 3.6.0 (2019-04-26)
```

```
## Warning in h2o.clusterInfo():
## Your H2O cluster version is too old (4 months and 22 days)!
## Please download and install the latest version from http://h2o.ai/download/
```

```
#remove any existing clusters
h2o.removeAll()
```

```
## [1] 0
```

```
#partition the data to create a train and test set using the train_set
splits = h2o.splitFrame(as.h2o(train_set.copy),
                        ratios =.7,
                        seed = 1)
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
train = splits[[1]]
test = splits[[2]]

#remove the progress bar
#h2o.no_progress()
```

With the set up of the h2o method complete, we can begin with the models. The first one will use the gradient boosting machine (GBM) model and use 4 explanatory variables to explain the rating variable. While the second model will look at only the placeID and userID. Details on the h2o package and the machine learning algorithims can be found here:

http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science.html https://cran.r-project.org/web/packages/h2o/h2o.pdf

```
#first model
gbdt_first = h2o.gbm(x = c('placeID', 'userID', 'n.rest_user', 'n.users_rest'),
                     y = 'rating',
                     training_frame = train,
                     nfolds = 3)
```

```
##
  |
  |                                                                      |   0%
  |
  |====                                                                  |   6%
  |
  |======================================================================| 100%
```

```
#summary not shown in rmd documentation
#summary(gbdt_first)

#model using only place and user
gbdt_mod2 = h2o.gbm(x = c('placeID', 'userID'),
                    y = 'rating',
                    training_frame = train,
                    nfolds = 3,
                    seed = 1,
                    keep_cross_validation_predictions = TRUE,
                    fold_assignment = 'Random')
```

```
##
  |
  |                                                                      |   0%
  |
  |==                                                                    |   2%
  |
  |======================================================================| 100%
```

```
#Summary not shown in rmd docuementation
#summary(gbdt_mod2)
```

With the gbm method, there is no difference between the two models. Including variables for the restaurants rate per user and the number of users per restaurant does not add any explanatory power. Thus, we'll stick with the second, more simple model for evaluation.

```
#evaluate on the test set
h2o.performance(gbdt_mod2, test)
```

```
## H2ORegressionMetrics: gbm
##
## MSE:  0.4978679
## RMSE:  0.7055975
## MAE:  0.5188923
## RMSLE:  0.3681608
## Mean Residual Deviance :  0.4978679
```

```
predicted_ratings_gbdt = h2o.predict(gbdt_mod2, as.h2o(test_set.copy))
```

```
##
```

```
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
rmse_gbdt = RMSE(predicted_ratings_gbdt, as.h2o(test_set.copy$rating))
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

The RMSE for the gbm model is 0.7057301.

We now move to a random forest model. As above, we'll look at a more complex model and a simple model and use whichever has the best RMSE on the training data to predict ratings in the test set.

```
#look at random forest
invisible(gc())

#random forest model
rf_mod1 = h2o.randomForest(
        training_frame = train,
        x = c('placeID', 'userID',  'n.rest_user', 'n.users_rest'),
        y = 'rating',
        ntrees = 50,
        max_depth = 20
)
```

```
##
  |
  |                                                                      |   0%
  |
  |============                                                          |  18%
  |
  |======================================================================| 100%
```

```
#summary not shown in rmd documentation
#summary(rf_mod1)

#random forest model with just place and user
rf_mod2 = h2o.randomForest(
        training_frame = train,
        x = c('placeID', 'userID'),
        y = 'rating',
        nfolds = 3,
        seed = 1,
        keep_cross_validation_predictions = TRUE,
        fold_assignment = 'Random'
)
```

```
##
  |
  |                                                                      |   0%
  |
  |========                                                              |  12%
  |
  |======================================================================| 100%
```

```
#summary not shown in rmd documentation
#summary(rf_mod2)
```

From the summaries, we see that the first model is better in terms of the RMSE on the training set. So we'll use that model to predict on the test set.

```
#model 1 has lower RMSE, so let's use that--evaluate on test
h2o.performance(rf_mod1, test)
```

```
## H2ORegressionMetrics: drf
##
## MSE:  0.4601478
## RMSE:  0.6783419
## MAE:  0.5553251
## RMSLE:  0.3710902
## Mean Residual Deviance :  0.4601478
```

```
#predict the ratings on test_set.copy, evaluate RMSE
predicted_ratings_rf_mod1 = h2o.predict(rf_mod1, as.h2o(test_set.copy))
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
rmse_rf = RMSE(predicted_ratings_rf_mod1, as.h2o(test_set.copy$rating))
```

```
##
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

The RMSE for the random forest approach is 0.6781025.

Conclusion

```
rmse_results = data.frame(Method=c('Restaurant effects', 'Rest. & User effects',
                                   'Regularization', 'GBDT', 'Random forest'),
                          RMSE = c(rmse_model1, rmse_model2, rmse_model3,
                                   rmse_gbdt, rmse_rf))
```

```
kable(rmse_results) %>%
        kable_styling(bootstrap_options='striped', full_width = F, position='center') %>%
        kable_styling(bootstrap_options='bordered', full_width=F, position='center') %>%
        column_spec(1, bold=TRUE)
```

Method	RMSE
Restaurant effects	0.7982501
Rest. & User effects	0.6660479
Regularization	0.6415814
GBDT	0.7057301
Random forest	0.6781025

From the table of the RMSEs of the models, we see that the regularization model has the lowest RMSE. This might be because of the size of the dataset–other methods with more data might perform better. The best RMSE was 0.6415814. The random forest method was the best performener of the ensemble methods, but was still not better than the RMSE of the user and restaurant effects model.