Code

Building a Model To Predict a Movie's IMDB Rating

Arjun Sahasrabuddhe

Introduction

Purpose

The goal of this project is to build a model that effectively predicts the IMDB rating of a movie. I want to better understand which variables play an important role in predicting our IMDB rating. I've always found myself checking a movie's IMDB rating after I finish watching it to see if I agree, but also before to see if a movie is worth watching. I'm curious to know whether an IMDB rating is a metric that can be predicted using a machine learning model.

What is an IMDB Rating?

You may have heard of rating systems such as "Rotten Tomatoes" or "Metacritic". These ratings are based on users who have rated the movie. Another rating system just like these is known as IMDB. Most of all the movies and TV shows that you have seen have an IMDB rating assigned to them. "IMDB", which stands for Internet Movie Database, is a database that consists of statistics for movies, TV shows, and much more. Each movie/TV show has an IMDB rating assigned to it, and this number is calculated based on IMDB registered users who cast a vote from 1-10 (1 being the most hated, 10 being the best ever) for the movie (assuming they actually watched it and are honest about their reviews). No rating system is perfect, but IMDB ratings is one that I use often before watching movies in the theater or even at home. It has been seen to be the most representative for today's movie watchers according to many experts as they come solely from users and are calculated using a consistent formula.

Hide

```
knitr::include_graphics("IMDB_Logo_2016.png")
```

Choosing a Dataset

The dataset that I chose for modeling includes information for over 5000 movies across 66 countries over the span of 100 years, including their IMDB ratings, movie producers, and more. I will be obtaining the data through Kaggle, and it's titled "IMDB 5000 Movie Dataset". The original dataset was from Dataworld, and it consists of 5043 observations and 28 predictors. I will be working with numeric as well as categorical variables.

Loading Necessary Packages

Let's start by loading in necessary packages for our project.

Hide

```
library(tidyverse) # includes multiple packages for data visualization and more
library(tidymodels) # includes multiple packages for data visualization and more
tidymodels_prefer() # for dealing with packages having the same name, uses tidymode
ls functions before others.
library(corrplot) # for creating correlation plots
library(naniar) # for dealing with missing variables
library(klaR)
library(poissonreg)
library(discrim)
library(corrr)
library(vip) # for our variable importance plots
```

Exploratory Data Analysis

Loading in Our Data

Let's begin by loading in our data into R studio.

Hide

```
imdb_data = read.csv('movie_metadata.csv') # reading in our data
```

Now that we have loaded the important packages as well as our dataset, we can start by exploring our data a little. Let's start by observing our data's dimensions:

Hide

```
dim(imdb_data)
```

```
## [1] 5043   28
```

We have 5043 rows and 28 columns of data! This is a great sign, as we have more than enough data to work with. We can even consider removing some data later if appropriate. But first, let us see how much missing data we actually have (in other words, it is the number of missing values we have in our dataset).

Dealing With Missing Values

Hide

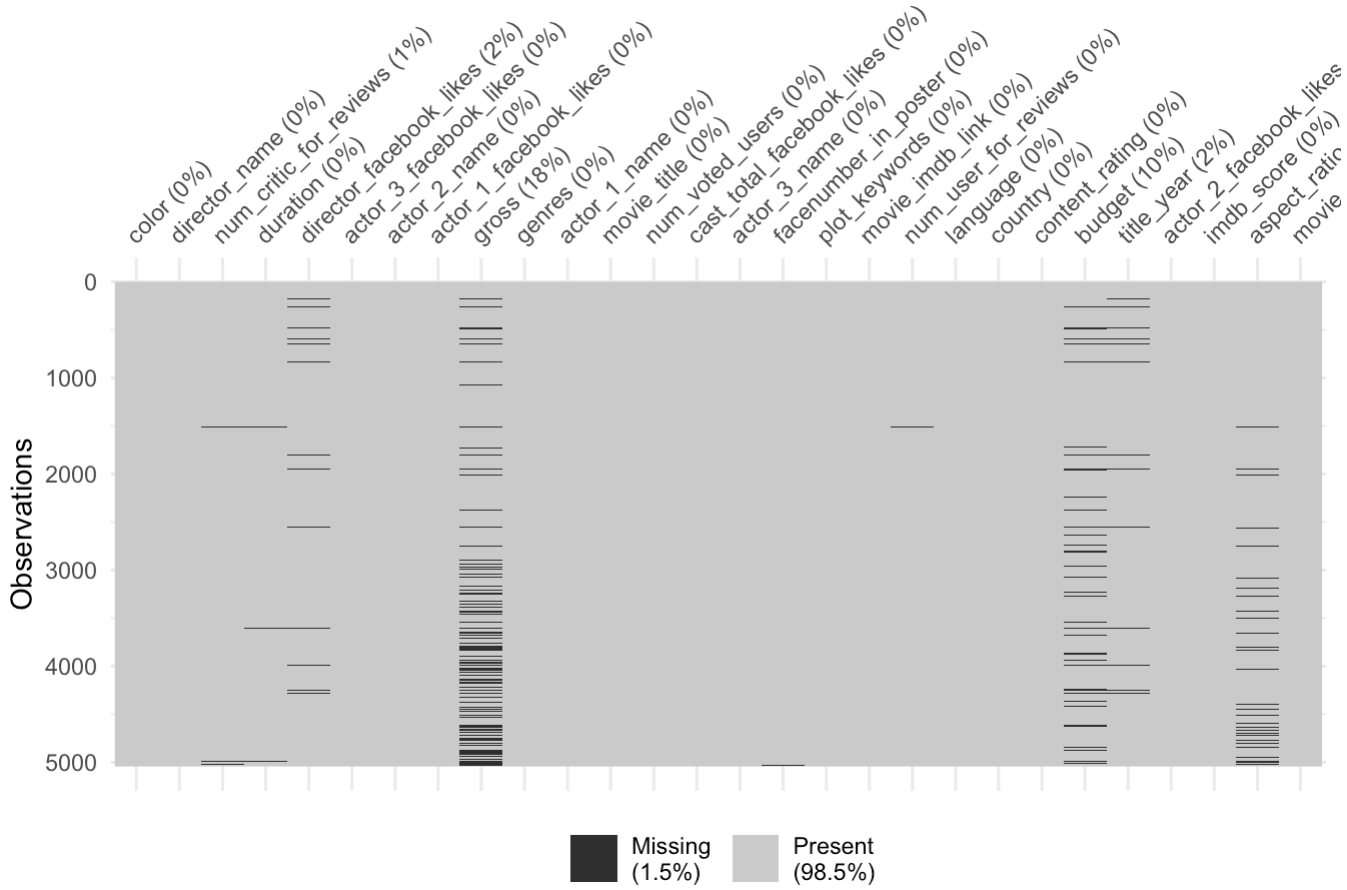
```
imdb_data %>%
n_miss()
```

```
## [1] 2059
```

We have 2059 missing values. This may seem like a lot at first glance, but if we look at what percentage of missing values this is when compared to our entire dataset, we have 2059/(5049*28), giving us 0.015, or 1.5%. That means we only have 1.5% of our values as missing. If we want to confirm this, we can create a missingness plot as displayed below.

Hide

```
vis_miss(imdb_data) # visualize data missingness with a plot.
```



Like we guessed, 1.5% of the values in our dataset are missing. But, one thing to note here is that a majority of our missing values are in one of two columns, gross and budget. I do believe that these two columns are very important in predicting our IMDB score, especially gross. At least it would seem so at first glance. So it wouldn't be beneficial to us if we included movies that don't have either a gross or budget value reported. We have no missing values for our IMDB score, which is great. So, let us remove all observations with missing data and look at the new dimensions of our modified "IMDB_data" dataset.

Hide

```
imdb_data = imdb_data %>% na.omit() # remove missing observations

dim(imdb_data)
```

```
## [1] 3801   28
```

Now we have successfully removed all missing values, and are still left with 3801 observations and 28 columns which is still plenty to work with. We overwrite our "IMDB_data" containing no missing values as seen above. But using 27 columns as predictors (1 being our outcome) is quite a lot. A lot of these columns at first glance are not going to be making an impact to our IMDB scores with our model, for instance the movie title. This is something that we will need to address in our recipe later, so let's not worry about it for now. But before we start with that, it can really be helpful to look at our predictors visually with graphs and charts, which is what we will do below.

Tidying our Data

Let's convert our categorical variables to factors and this will create n number of levels for us. We need to be able to distinguish one from another numerically as we will be using these in our formula, so let's do this below.

Hide

```
imdb_data$color = factor(imdb_data$color)
imdb_data$aspect_ratio = factor(imdb_data$aspect_ratio)
imdb_data$content_rating = factor(imdb_data$content_rating)
imdb_data$title_year = factor(imdb_data$title_year)
imdb_data$country = factor(imdb_data$country)
imdb_data$language = factor(imdb_data$language)
```

Now if we look at our dataset, we see that these variables are of type factor which is what we wanted.

Many of our categorical variables have an unequal distribution of values (I have created multiple plots to confirm this skewness in data, but it would be unneccessary to add all those here). For example, a large majority of our movies are made in English. For these variables, it may be smarter to create a new category called other, and lump any movies that aren't produced in English into that category which is what we will do below. We do this for almost all of our categorical variables.

Hide

```
imdb_data = imdb_data %>%
  mutate(country = fct_lump_n(country, n = 1)) %>%
  mutate(language = fct_lump_n(language, n = 1)) %>%
  mutate(color = fct_lump_n(color, n = 1)) %>%
  mutate(title_year = fct_lump_n(title_year, n = 30)) %>%
  mutate(aspect_ratio = fct_lump_n(aspect_ratio, n = 2)) %>%
  mutate(content_rating = fct_lump_n(content_rating, n = 4))
```

Visual EDA

We have familiarized ourselves with our variables and modified our datasets so far. But we haven't seen our data visually yet. Let's first take a look at our response variable more closely, imdb_score.

Hide

```
summary(imdb_data$imdb_score)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.600   5.900   6.600   6.466   7.200   9.300
```

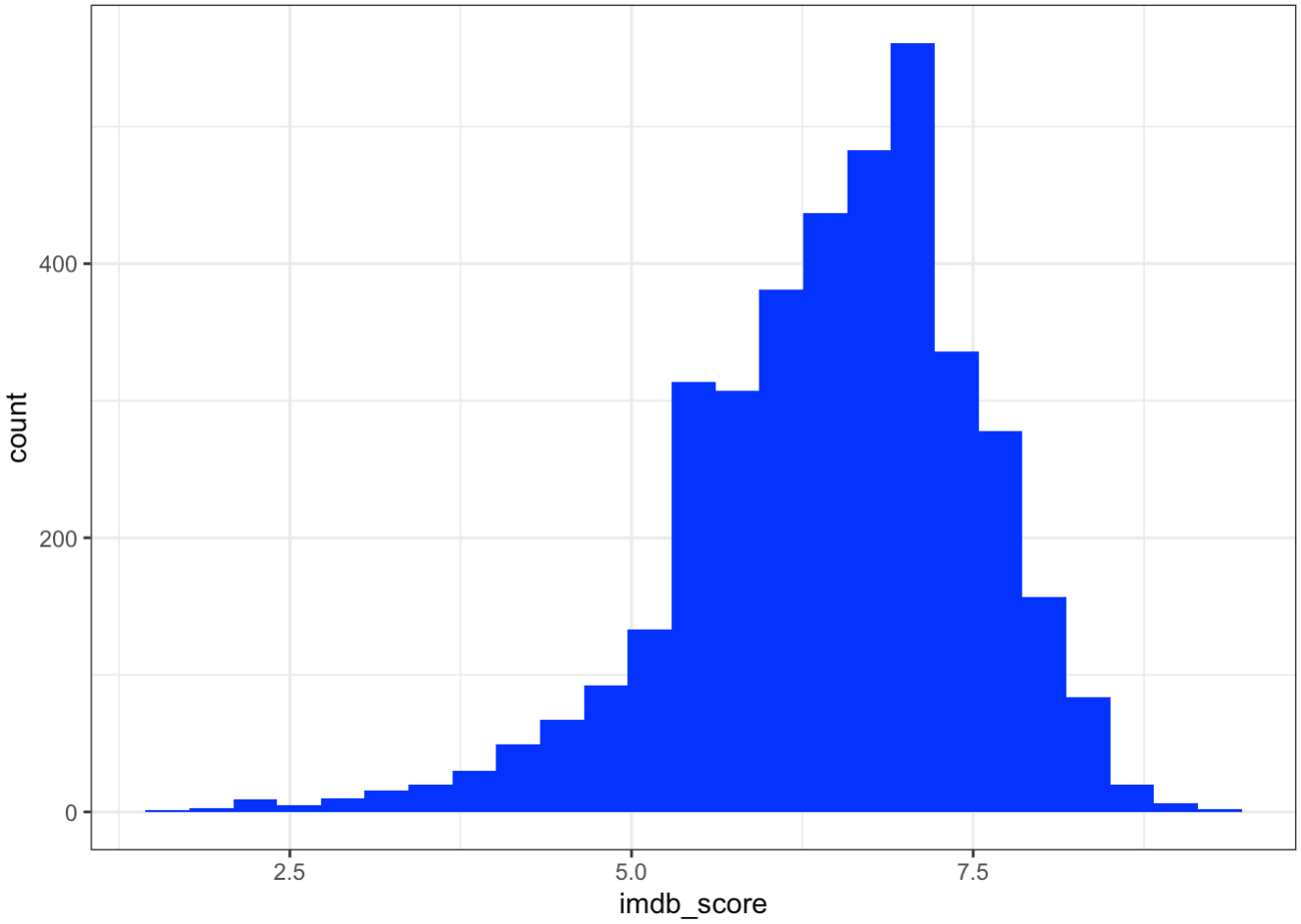
Hide

```
sd(imdb_data$imdb_score)
```

```
## [1] 1.057642
```

Hide

```
imdb_data %>%
  ggplot(aes(x=imdb_score)) +
  geom_histogram(bins=25, fill = 'blue') +
  theme_bw()
```

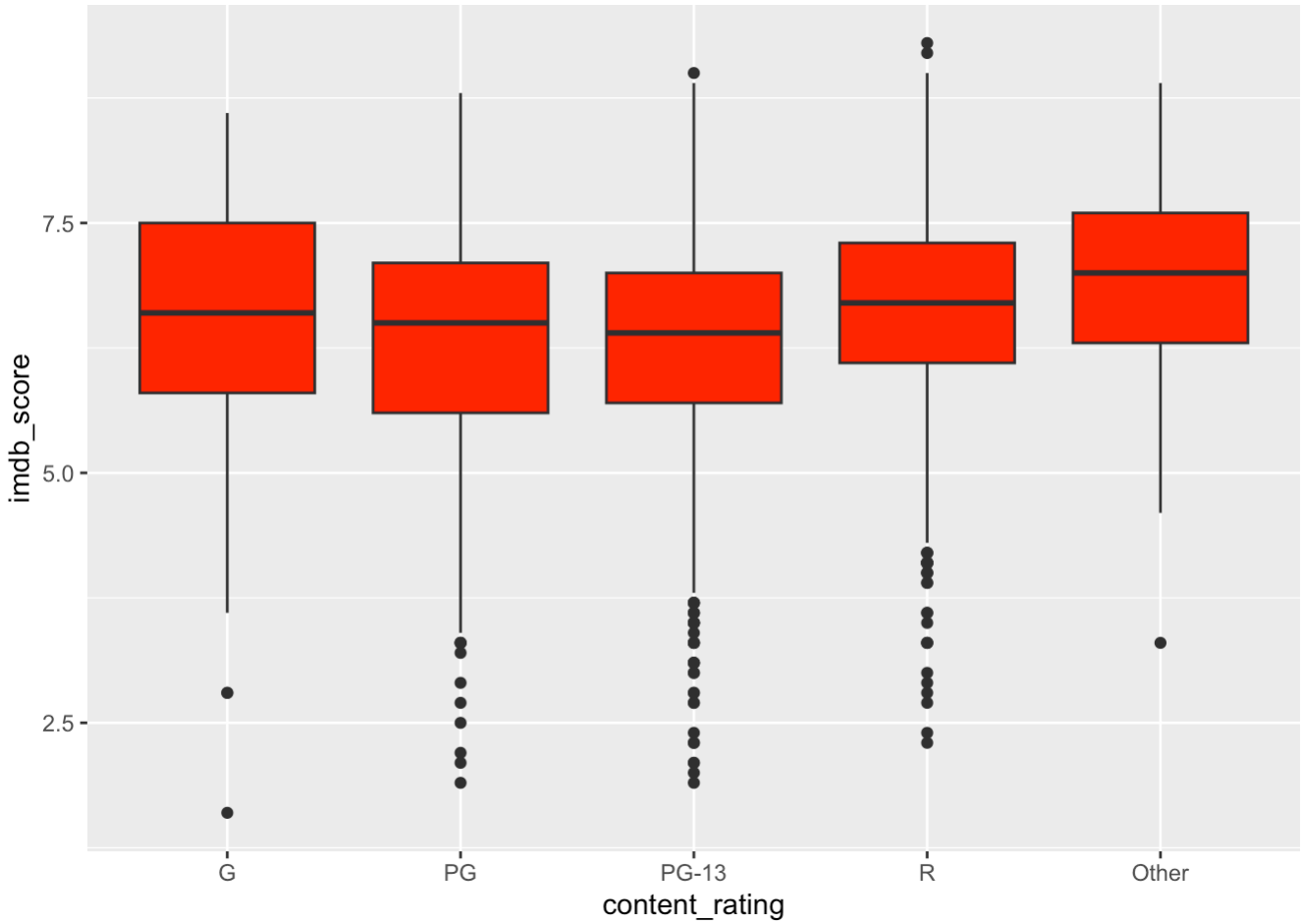


By looking at this bar chart, we can see that most of our imdb_scores are centered from about 6 to 7. I shouldn't be too surprised by this as this is likely the average rating of movies in general, even though the movies I watch tend to be a bit higher. If we look at our summary statistics above that, we see that the mean is 6.466, and the first quartile to third quartile is 5.9-7.2.

I was also curious as to whether the content rating of a movie had any effect on our response. My guess would be that G rated movies would have the highest imdb ratings on average since there would likely be parents rating the movies, and they would be careful to show their children only the best movies.

Hide

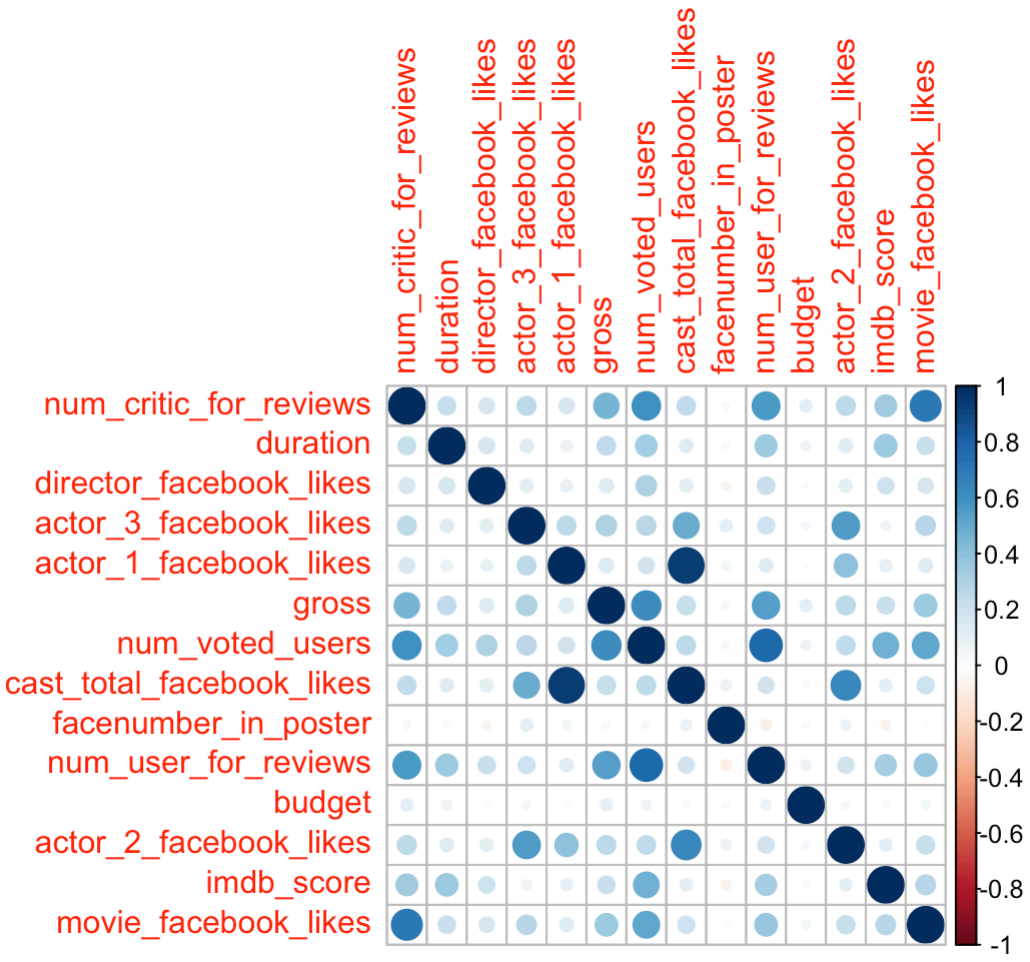
```
imdb_data %>%
  ggplot(aes(x=content_rating, y=imdb_score)) +
  geom_boxplot(fill='red')
```



But as we see, there isn't any real difference between the mean, or 1st to 3rd quartile ranges. There are not a whole lot of outliers either (our G rating movies only have 2 outliers). Also, one thing to notice is that there are many more lower outliers than higher outliers. At least visually, it doesn't appear like our content rating will have much of an effect on the imdb_score.

Hide

```
imdb_data %>%
  select(where(is.numeric)) %>%
  cor() %>%
  corrplot()
```



Here we create a correlation plot with the numeric variables. To my surprise, IMDB_score didn't have a strong positive correlation with gross nor budget. In fact, budget had essentially zero correlation and gross had a very slight correlation of about 0.2. There was only a significant enough positive correlation of IMDB_score with num_voted_users at around 0.5. This can be explained as the more one likes a movie, the higher chance they will rate the movie. The other significant correlations are self explanatory, like cast_total_facebook_likes, and actor_2_facebook likes. One thing worth noting is that actor_1_facebook_likes and cast_total_facebook_likes are perfectly correlated with one another (1 correlation). We can use a linear model by using one of them to fully predict the other with 100% accuracy. But unfortunately it's not going to be that easy for us.

Setting Up Our Models

Now, we are almost ready to start fitting our models to try to predict IMDB scores. But before we start, we must split our data, create a recipe, and perform k fold cross validation. I will explain these further below.

Splitting Our Data

I have chosen a 70/30 split for our data. That is, 70% of our data will be used to train our model, and the other 30% will be used to test our model. The proportion split varies for different datasets, but the training data must have plenty of data as this is what our model will be learning from. Our training data also must be more than our testing data.

Hide

```
set.seed(0205)
imdb_split = initial_split(imdb_data, prop = 0.7, strata = imdb_score)
imdb_train = training(imdb_split)
imdb_test = testing(imdb_split)
```

As we can see, 70% of our data consists of our training data (2660/3801 obs.), and the remaining 30% consists of our testing data (1141/3801 obs.) and both sets have been stratified on the variable we are trying to predict, imdb_score.

Our goal is to not touch this testing data. It's reserved for us only at the end for when we have chosen the best model and want to test this model on new, unseen data. In other words, our model shouldn't have access to any of this data when learning and building itself.

Now we will proceed into creating our recipe.

Creating Our Recipe

If we look at our columns, we can remove the director_name, actor_1_name, actor_2_name, actor_3_name, and movie_name columns as our model is not going to understand which actors or movies are popular. The names themselves are not going to make an impact on our imdb_score. We can also remove actor 1, 2, 3, and director facebook likes since we are adding the total cast_total_facebook_likes variable. We have also removed the genre of movie column since there are too many different categories that also have a lot of overlap, so we can't really distinguish them.

Think of a recipe like a formula. We are trying to make a dish, and we have many more items on our shelf than we really need. Our dish only calls us to use some of our ingredients at home, not all of them. We only want to include the 14 predictors that we specified in our codebook.

Hide

```
imdb_recipe = recipe(imdb_score ~ color + num_critic_for_reviews + duration + aspec
t_ratio +
                       cast_total_facebook_likes + content_rating + country +
                       language + budget + gross + title_year + movie_facebook_likes
+
                       num_user_for_reviews + num_voted_users, data = imdb_train) %
>%

  step_dummy(color, aspect_ratio, content_rating, country, language, title_year)

imdb_recipe %>%
  prep() %>%
  bake(new_data = imdb_train)
```

```
## # A tibble: 2,660 × 48
##    num_critic_for_reviews duration cast_total_facebook_likes    budget     gross
##                     <int>    <int>                     <int>     <dbl>     <int>
##  1                    377      131                     26679 209000000  65173160
##  2                    378      165                      3988 210000000 245428137
##  3                    436      123                     17657 200000000 116593191
##  4                    367      158                      2144 200000000 166112167
##  5                    384      127                     47334 176000000  47375327
##  6                     85      106                     15870 170000000 113745408
##  7                    186       96                    108016 175000000 100289690
##  8                    250      118                     26683 175000000 150167630
##  9                    286      120                      3233 165000000 102315545
## 10                    248      206                     24598 155000000  34293771
## # ℹ 2,650 more rows
## # ℹ 43 more variables: movie_facebook_likes <int>, num_user_for_reviews <int>,
## #   num_voted_users <int>, imdb_score <dbl>, color_Other <dbl>,
## #   aspect_ratio_X2.35 <dbl>, aspect_ratio_Other <dbl>,
## #   content_rating_PG <dbl>, content_rating_PG.13 <dbl>,
## #   content_rating_R <dbl>, content_rating_Other <dbl>, country_Other <dbl>,
## #   language_Other <dbl>, title_year_X1988 <dbl>, title_year_X1989 <dbl>, …
```

Here we have used our 14 predictors in our formula, where the 6 categorical variables are dummy coded. We also prep and bake our recipe to make sure that it is running smoothly.

K-Fold Cross Validation

Earlier we had mentioned that we were going to perform k fold cross validation. Here, we create 10 folds of data from our training set. Essentially, we are creating k number of folds, treating each fold as a testing set and using the remaining k-1 folds for training that fold. They are mini training and testing sets (created from the original training data) and this process is repeated for all of the k folds.

This is beneficial since we are exposing our model to different parts of the training set to learn from. We use this instead of fitting the entire training set since k-fold is a better estimate of how our testing data will perform. We will use these folds to choose the best model based on RMSE, and then fit this model to the entire training set afterwards. It is exactly what the name means, validation. Our model isn't really learning from this data but merely helping us choose a model.

Hide

```
imdb_folds = vfold_cv(imdb_train, v = 10, strata = imdb_score)
```

Fitting Our Models

Finally we are ready to dive into our full course meal. It's time to fit our models! We will fit a total of 4 different types of models which will include Linear Regression, K-nearest neighbors, Random Forests, and Boosted Trees. I have chosen to use Root Mean Squared Error (RMSE) as a metric for assessing how well our models fit the data, as it is one of the most widely used metrics for regression problems.

We will follow a five step process here.

1. Specify all the models we want to fit, including flagging parameters than need to be tuned later.

Hide

```
lm_model = linear_reg() %>% # linear regression
  set_engine('lm')

knn_model = nearest_neighbor(neighbors = tune()) %>%   # K nearest neighbors
  set_engine('kknn') %>%
  set_mode('regression')

rf_model = rand_forest(mtry = tune(),           # random forest
                       trees = tune(),
                       min_n = tune()) %>%
  set_engine('ranger', importance = 'impurity') %>%
  set_mode('regression')

bst_model = boost_tree(mtry  = tune(),
                       trees = tune(),
                       learn_rate = tune()) %>%     # boosted trees
  set_engine('xgboost') %>%
  set_mode('regression')
```

2. Set up workflows for each model, adding the model and the recipe together.

Hide

```
lm_wflow = workflow() %>%
  add_model(lm_model) %>%
  add_recipe(imdb_recipe)

knn_wflow = workflow() %>%
  add_model(knn_model) %>%
  add_recipe(imdb_recipe)

rf_wflow = workflow() %>%
  add_model(rf_model) %>%
  add_recipe(imdb_recipe)

bst_wflow = workflow() %>%
  add_model(bst_model) %>%
  add_recipe(imdb_recipe)
```

3. Create tuning grids for the models that will be tuned. Here we will be able to decide how many different variations of a model we want to fit.

Hide

```
knn_grid = grid_regular(neighbors(range = c(1, 10)), levels = 10)

rf_grid = grid_regular(mtry(range = c(2, 12)), trees(range = c(50, 300)), min_n(ran
ge = c(10, 20)), levels = 5)

bst_grid = grid_regular(mtry(range = c(2, 12)), trees(range = c(50, 300)), learn_ra
te(range = c(-10, -1)),
levels = 5)
```

4. Tune our models using our tuning grids on our folded data (k-fold).

Hide

```
knn_tune = tune_grid(knn_wflow,     # KNN tuned
                resamples = imdb_folds,
                    grid = knn_grid,
                 metrics = metric_set(rmse))


lm_tune = fit_resamples(lm_wflow, resamples = imdb_folds, metrics = metric_set(rms
e)) # no tuning, but we fit our folded data to our linear workflow

rf_tune = tune_grid(rf_wflow,       # Random Forest tuned
                 resamples = imdb_folds,
                    grid = rf_grid,
                metrics = metric_set(rmse))

bst_tune = tune_grid(bst_wflow,     # Boosted Trees tuned
                 resamples = imdb_folds,
                    grid = bst_grid,
                metrics = metric_set(rmse))
```

5. Save our tuned models into RDA files to prevent them from rerunning every time. Then, load these files.

Hide

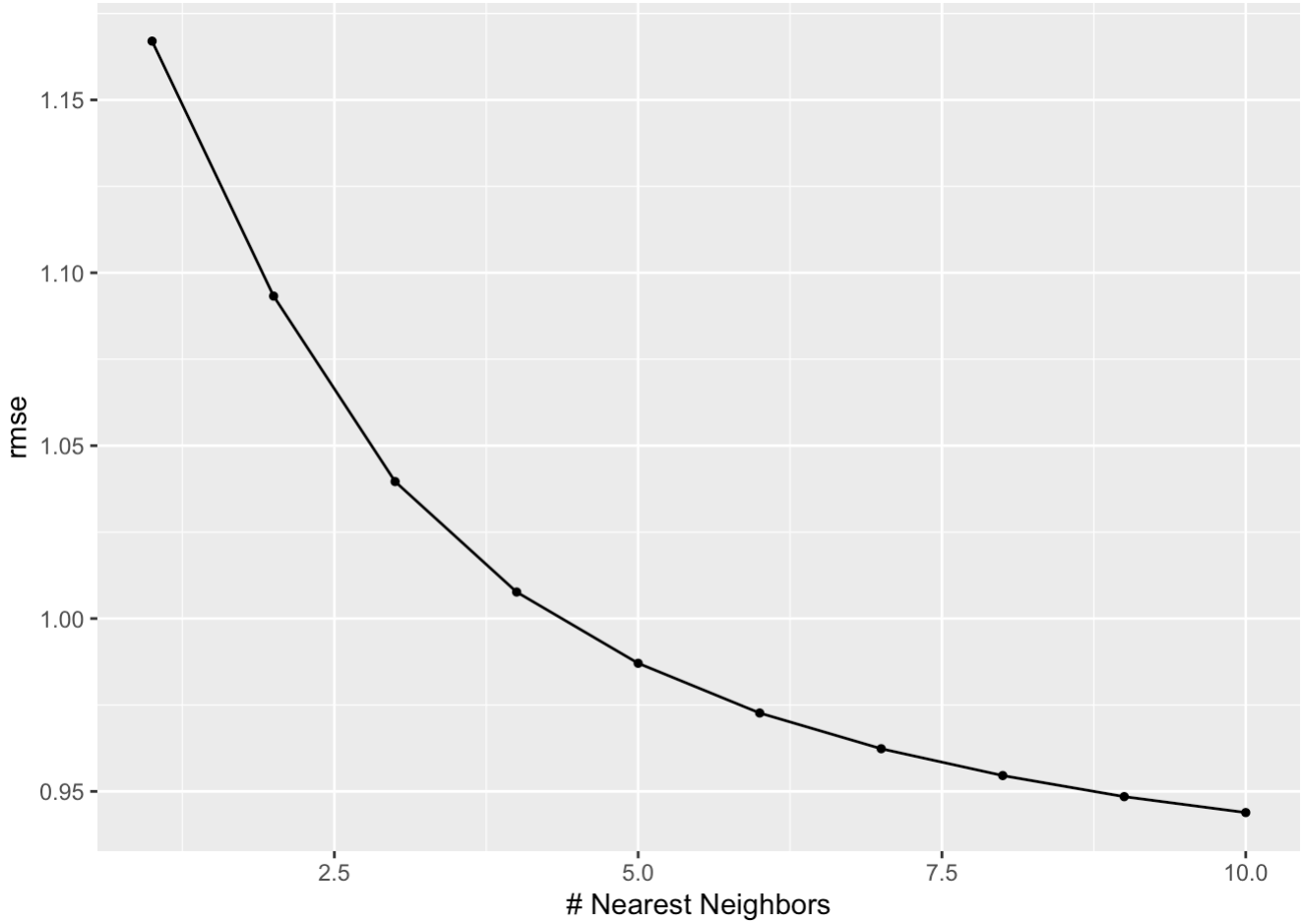
```
save(lm_tune, file = 'lm_tune.rda')
save(knn_tune, file = 'knn_tune.rda')
save(rf_tune, file = 'rf_tune.rda')
save(bst_tune, file = 'bst_tune.rda')

load('lm_tune.rda')
load('knn_tune.rda')
load('rf_tune.rda')
load('bst_tune.rda')
```

Visualizing Our Models

Let us take a look at our models visually first. We won't need to look at our linear model as there are no parameters tuned.

Hide

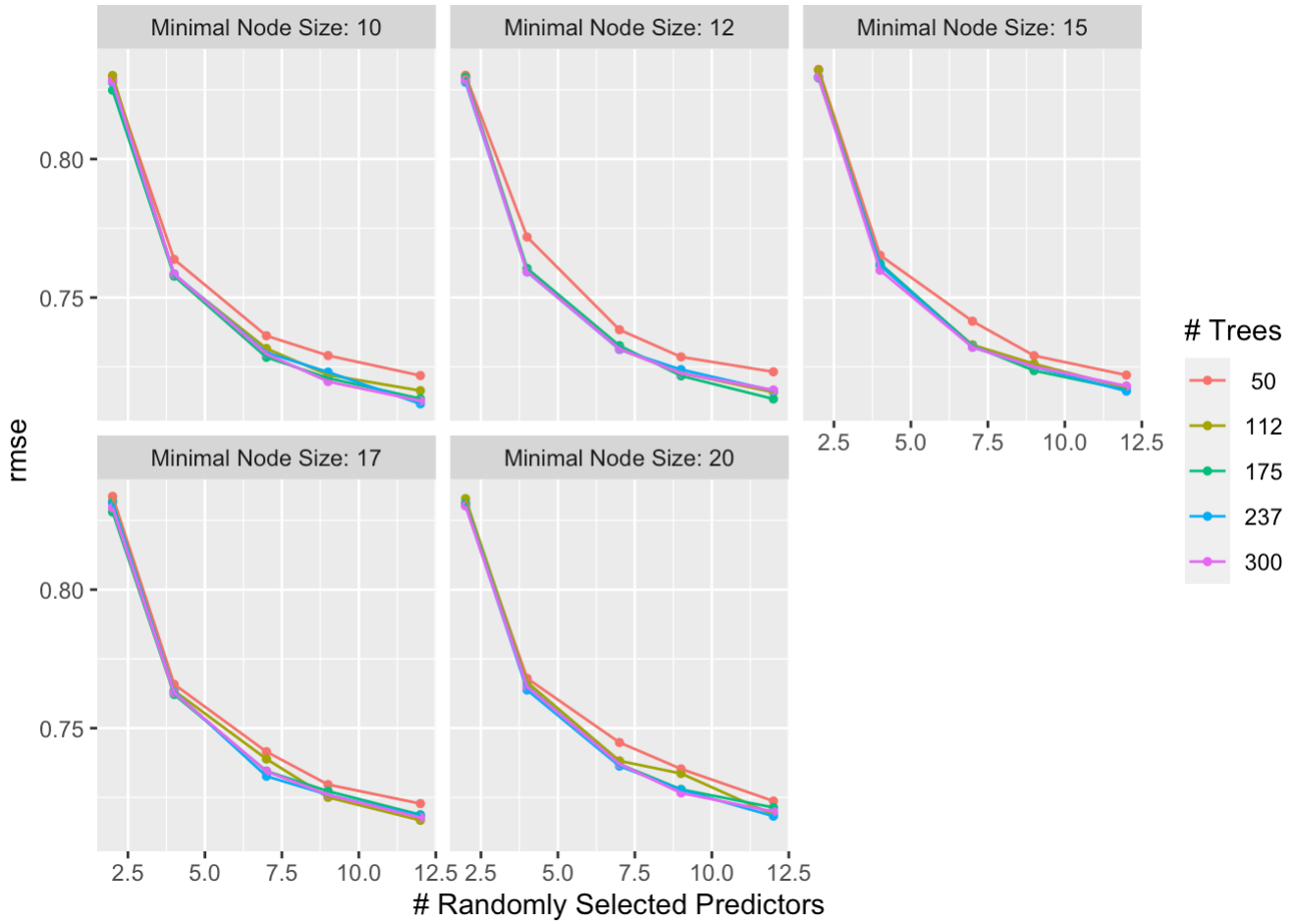
```
autoplot(knn_tune)
```



We can see that the more number of neighbors, the lower our RMSE at least up until 10. One possible explanation could be that there is random error picked up on when we use fewer neighbors. A movie that is considered a neighbor isn't really one. Even though it may mean a more flexible model, it could be too flexible.

Hide

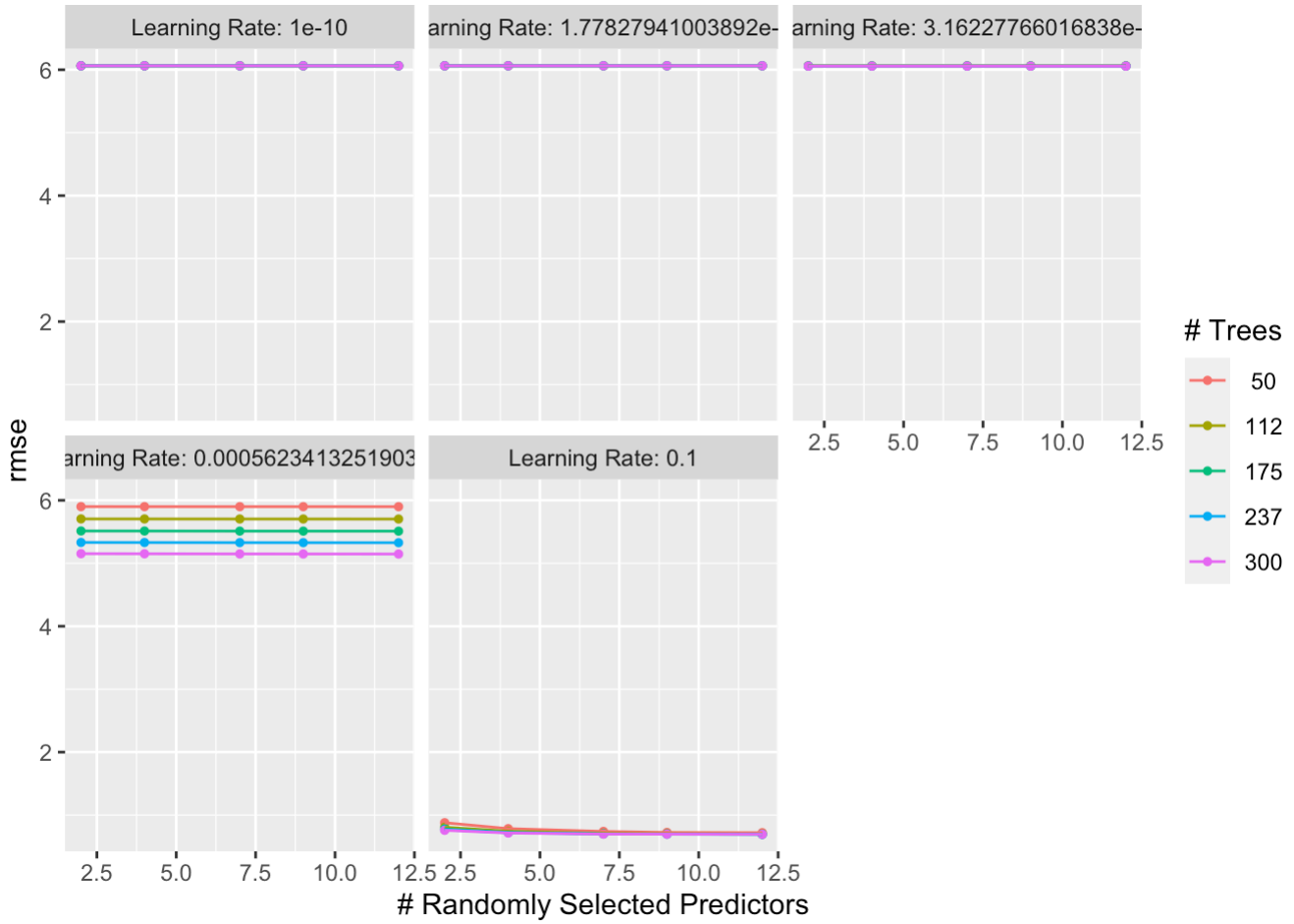
```
autoplot(rf_tune)
```



The number of trees doesn't seem to make an impact on our RMSE. However the more number of predictors used (mtry) up until 12.5 seems to reduce our RMSE.

Hide

```
autoplot(bst_tune)
```



A learning rate of 0.1 is an optimal condition for our boosted tree model. Again in this case, the number of trees doesn't play a huge role at least visually, and we will need to choose our best model to see what hyperparameters were considered best.

Highlighting Our Best Model

Let's take a look at our top tuned model for all of our models.

Hide

```
show_best(lm_tune)
```

```
## # A tibble: 1 × 6
##   .metric .estimator  mean     n std_err .config
##   <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.835    10  0.0237 Preprocessor1_Model1
```

Hide

```
show_best(knn_tune, n = 1)
```

```
## # A tibble: 1 × 7
##   neighbors .metric .estimator  mean     n std_err .config
##       <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1        10 rmse    standard   0.944    10  0.0199 Preprocessor1_Model10
```

Our best KNN model with 10 neighbors had a RMSE of 0.944. This performed worse than our linear model which had a RMSE of 0.835.

Hide

```
show_best(rf_tune, n = 1)
```

```
## # A tibble: 1 × 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1    12   237    10 rmse    standard   0.712    10  0.0139 Preprocessor1_Model0…
```

Our best random forest model had parameters mtry = 12, trees = 237, and min_n = 10, with an RMSE of 0.712.

Hide

```
show_best(bst_tune, n = 1)
```

```
## # A tibble: 1 × 9
##    mtry trees learn_rate .metric .estimator  mean     n std_err .config
##   <int> <int>      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1    12   237        0.1 rmse    standard   0.694    10  0.0132 Preprocessor1_M…
```

And finally, our boosted trees model had parameters mtry = 12, trees = 237, and learn_rate = 0.1 with an RMSE of 0.694! Our best model so far. This is the model we will decide to fit and use.

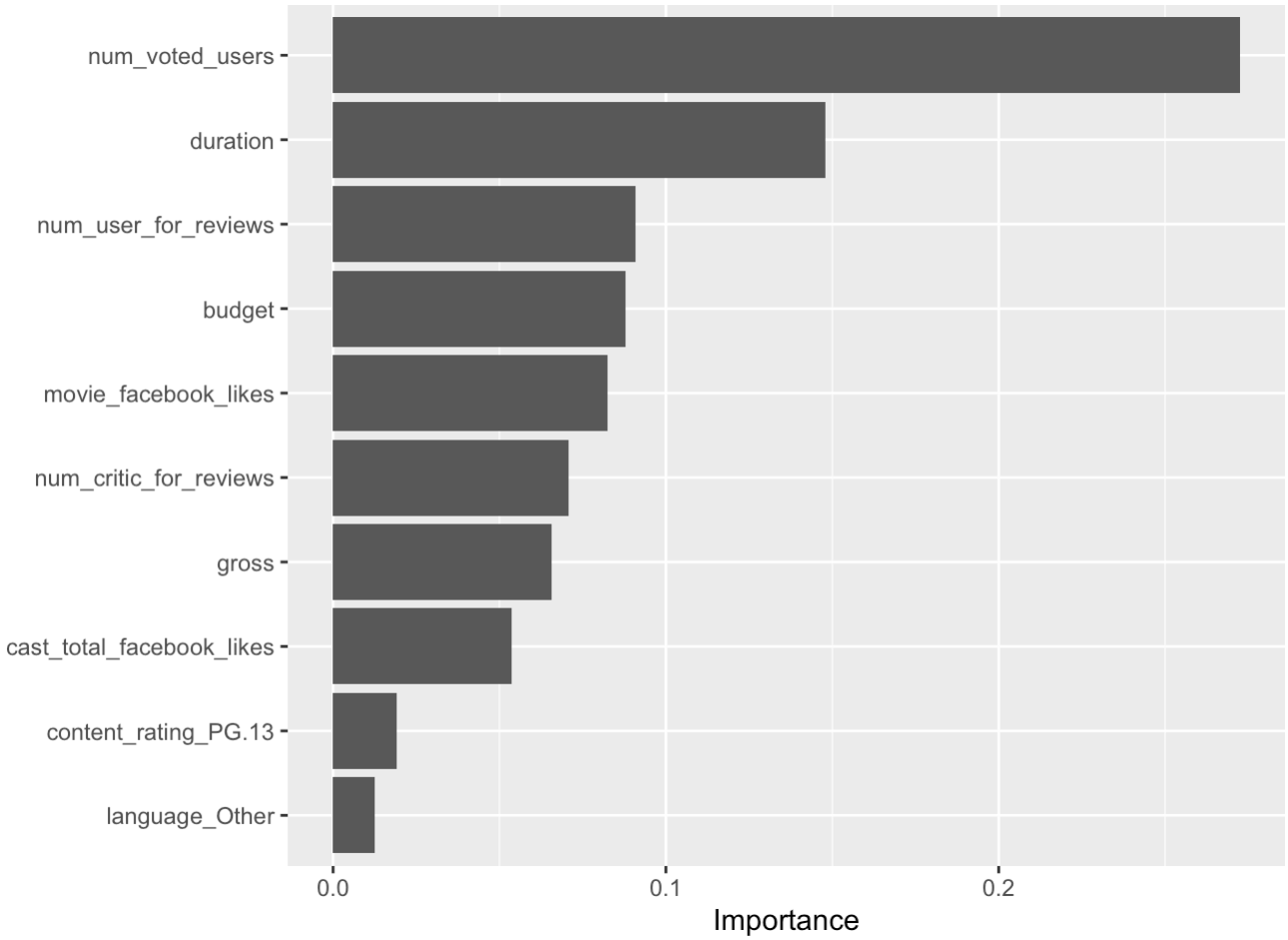
Something we can notice here is that mtry = 12, and trees = 237 were constant with both random forest and boosted tree models. Something that's definitely noteworthy.

Using Our Chosen Model

Now, we select our best model (boosted tree tuned), and fit it to our original training set that we created at the very beginning. Then we create a variable importance plot using this data to tell us which predictors the model found to be the most important in its prediction.

Hide

```
best_bst = select_best(bst_tune)

final_bst_model = finalize_workflow(bst_wflow, best_bst)
final_bst_model = fit(final_bst_model, imdb_train)

final_bst_model %>% extract_fit_parsnip() %>%
  vip()
```



As we can see, the number of voted users was the most important variable. Other than this, there wasn't a variable that stood out for us. When people see a movie they like, they are much more motivated to rate it. The gross and budget didn't play much of a role.

Fitting Our Model to the Testing Data

Now, our final step is what we've been waiting for. Fitting our chosen model to the testing set. I'm sure we are both curious to see how it will perform.

Hide

```
final_bst_model_test <- augment(final_bst_model, imdb_test)

rmse(final_bst_model_test, truth = imdb_score, .pred)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.688
```

I was a little surprised it didn't do a lot better than the k-fold cross validation but the RMSE was still lower at 0.688. This is still not a bad number and a model worth using in the future. But I am still very happy with it's performance and would call it a success!

Conclusion

It was clear that our boosted trees model performed the best among the four for our k fold data. But surprisingly, this model on our testing data performed just a little better as it did on the cross validation set, not better. The hope was that this best boosted trees model would have a lower RMSE for our testing set since it was fit on the full training set to learn from. But what we can take away from this is that our k-fold cross validation was very successful in choosing the right model.

One inaccuracy in our model was including variables such as the number of users who voted. Just because this variable was an important predictor for imdb_score doesn't mean that we should include it. This is because we will not be having this information when predicting a new movie's rating. Other factors like budget would be valid in this case. One other thing I could explore next time is observing interactions between variables, and their importance on our predictions.

Overall I felt I learned a lot in this project and this topic really furthered my interest in this field. I do feel there is room for improvement in terms of modeling, but I am certain that this won't be the last time that I will be building a machine learning model!

Sources

Link to the dataset: https://www.kaggle.com/datasets/carolzhangdc/imdb-5000-movie-dataset (https://www.kaggle.com/datasets/carolzhangdc/imdb-5000-movie-dataset)

Information regarding IMDB ratings was found on the IMDB help website: https://help.imdb.com/article/imdb/track-movies-tv/ratings-faq/G67Y87TFYYP6TWAV# (https://help.imdb.com/article/imdb/track-movies-tv/ratings-faq/G67Y87TFYYP6TWAV#)