Predicting Climbing Route Quality

PSTAT 131 Final Project

Daniel Ledvin

2024-02-12

Introduction

The goal of this project is to build a model that will predict the quality rating of a climbing problem based on its different characteristics.

What is climbing?

Climbing refers to the sport of rock climbing. This sport is pretty self-explanatory, it just consists of climbing rocks. However, there are many different ways to climb and different types of routes, or the predetermined path of a climb. In this project, I will be focusing on lead climbing, which means that a climber clips their rope into protection devices as they climb. The two subcategories of lead climbing are trad, or traditional, and sport. Trad climbing means that one must set and remove their own protections along the route as they go. Sport means that the protections are already anchored into the wall, so climbers can just clip into them as they go. Another important metric in climbing is climb rating, or sometimes called grade. Rating refers to the difficulty of the climb. The grades increase numerically from 5.0 to 5.9, which are considered beginner climbs. Once the rating hits 5.10, a letter from a-d is added to the end, where a climb rated as a 5.10d is more difficult than a climb rated as a 5.10b.

Motivation

On REI's website, MountainProject.com, there are hundreds of thousands of user documented routes from all over the world. Entries on this website include information about each climb, as well as a user quality rating. Users that complete the climb are allowed to rate the quality of the climb from 0 to 4 stars, 4 being the best and 0 being the worst. This raises the question of whether there is some kind of relationship between the quality rating of a climb and its respective difficulty, among other characteristics like route length, type, and location. Therefore, I would like to test if it is possible to create a model that may predict what the quality rating of a climb may be based on these characteristics.

Data Source

I will be using data from the Kaggle data set, "Mountain Project Rotues and Forums", scraped from REI's Mountain Project by user Patricia Degner.

Exploratory Data Analysis

Loading and Exploring the Raw Data

```
initdata <- read.csv('data/mp_routes.csv') %>%
  clean_names()
dim(initdata)
```

```
## [1] 116700     14
```

Since the data set is so large, with 116700 routes and 14 variables, it is most likely true that are some missing values in the data as some climbs may have characteristics that were unmeasured when they were submitted to the website.

```
names(which(colSums(is.na(initdata)) > 0))
```

```
## [1] "length"
```

```
sum(is.na(initdata))
```

```
## [1] 15232
```

We can see that the only column missing data is length, with 15,232 missing values. Since the dataset is so large, we can remove the routes that have missing length values without hurting our model.

Tidying the Data

To improve the data, I will make some changes to it.

```
set.seed(132)

initboulder <- initdata %>%
  na.omit() %>%
  filter(num_votes > 10) %>%
  filter(grepl('^5', rating)) %>%
  sample_n(1000)

write.csv(initboulder, '/Users/danielledvin/Downloads/R/PSTAT 131 Statistical Machine Learning/Final Pr
```

Firstly, I removed all missing values. Due to the size of the data, there is really no point to try to impute missing values.

After looking through the raw data, we can also see that there are some routes that have thousands of votes, while some have as little as one vote on quality rating. Since it is easily possible that a climber finished a route but had a bad day and chose to take out their anger on a climb, giving a route that might be considered four stars in quality only one star. If that climber was one of two people that rated the route, it would drop the rating significantly. To account for this, I chose to exclude routes with less than ten votes.

Another issue with the data is that it includes routes of different classes. Climbing class essentially refers to how steep the climb is. A class one climb is just hiking. Class two and three climbing is scrambling,

which means that the mountain is steep, so one might use their hands to help climb up the hill. Class four is considered climbing, but it is very easy. A rope is typically used as the climb could be high, but it is irrelevant to this model since we are looking at lead climbing, which is only really done for class 5 climbs. Since we are only looking at class 5 routes, I removed all routes that do not fall in that category.

Finally, Since our data set is so massive, it would be hard to work with the entire thing. This will especially become an issue when tuning models to the data later. Since I do not want to wait several weeks for a random forest model to finish tuning, I chose to take a random sample of 1000 routes from the refined data frame and save it as `boulder.csv` to make it easier to load in the future.

```
boulder <- read.csv('data/boulder.csv') %>%
  select(-c(x, route, location, url, protection))
```

Of the 14 variables in our dataset, several of them are irrelevant to our model. Route name is one of these since each route name is unique and serves no purpose in predicting the quality of climb (but maybe if I named a terrible route "Best Route Ever" it might convince some people). I also removed location since we have the gps coordinates, which are both more accurate and easier to work with. URL being removed should be pretty easy to understand why. Finally, I removed protection since there are so many different types of protections between the climbs that it makes the variable almost unique in each one.

Tidying Route Ratings

One of the problems in this dataset is found in the values of the route ratings. Each route has a rating as described before, but some routes have an additional piece of information after the difficulty rating. Typically this second value describes the position of holds or where the protections are. These values will be removed since they are not really relevant to the route rating that we need for this model.

Some routes may also be denoted with a plus or minus, i.e. 5.7+, which means that route is slightly harder relative to other 5.7 climbs. I chose to remove this as well because this difference in difficulty is marginal.

Similarly, some routes may be considered between grades, notated with something like 5.10c/d. This is similar to the plus or minus notation, meaning that a route may be relatively harder than other 5.10c's but easier than 5.10d. There is a common practice in climbing called sandbagging, which means that routes are marked as easier than they may actually seem. I chose to implement this in my model by taking climbs markes as a 5.10c/d and inputting it as a 5.10c. This is reasonable since the difference in difficulty between a 5.10c/d and a 5.10c is not that major, so it will not significantly impact my model.

```
boulder$rating <- gsub(' PG13| R| X| Easy Snow| A0| C1| V2',
                       '', as.character(boulder$rating))
boulder$rating <- gsub('5.7[+]', '5.7', as.character(boulder$rating))
boulder$rating <- gsub('5.8[-]|5.8[+]', '5.8', as.character(boulder$rating))
boulder$rating <- gsub('5.9[+]|5.9[-]', '5.9', as.character(boulder$rating))
boulder$rating <- gsub('[-]|a/b', 'a', as.character(boulder$rating))
boulder$rating <- gsub('[+]|c/d', 'c', as.character(boulder$rating))
boulder$rating <- gsub('b/c', 'b', as.character(boulder$rating))
boulder$rating <- gsub('^5.10$', '5.10b', as.character(boulder$rating))
boulder$rating <- gsub('^5.11$', '5.11b', as.character(boulder$rating))
boulder$rating <- gsub('^5.12$', '5.12b', as.character(boulder$rating))

rate_lvl <- c('5.3', '5.4', '5.5', '5.6', '5.7', '5.8', '5.9', '5.10a', '5.10b', '5.10c', '5.10d', '5.1

# 5.2, 5.13b, and 5.13d only have one value, which will cause issue with model tuning
boulder <- boulder %>%
  filter(if_any(rating, ~ !(.x %in% c('5.2', '5.13d', '5.13b')))) %>%
  mutate(rating = factor(rating, levels = rate_lvl))
```

Tidying Route Types

Similarly to the ratings, there are also different ways to climb routes besides lead climbing. In the dataset, every route is marked as trad, sport, or both, which means that they can be lead climbed. However, some routes can also be marked with values like 'alpine' or 'tr'. Since these are not relevant to my model, I have removed them.

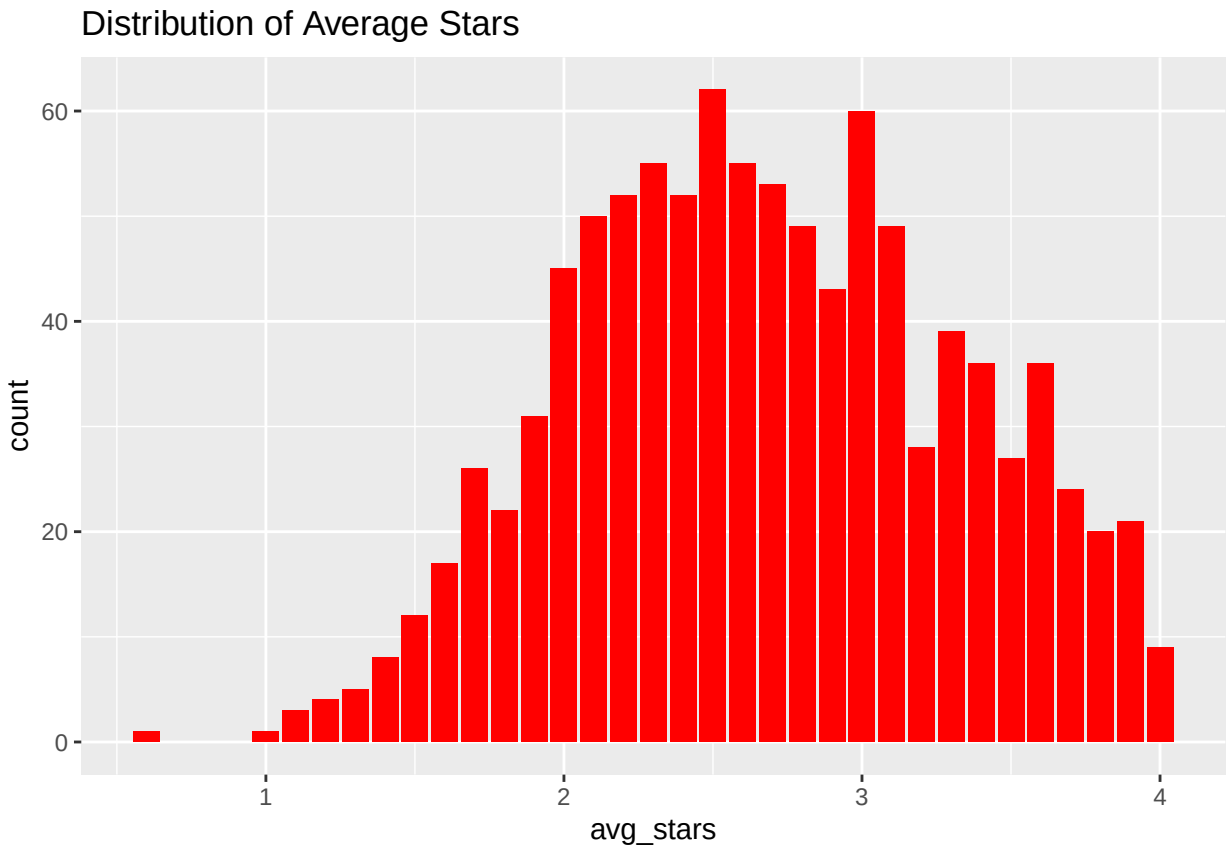
```
boulder$route_type <- gsub('Trad, Sport', 'Both',
                           as.character(boulder$route_type))
boulder <- boulder %>%
  mutate(route_type = case_when(
    startsWith(route_type, 'Sport') ~ 'sport',
    startsWith(route_type, 'Trad') ~ 'trad',
    startsWith(route_type, 'Both') ~ 'both',
    TRUE ~ as.character(boulder$route_type)
  )) %>%
  mutate(route_type = factor(route_type, levels = c('sport', 'trad', 'both')))
```

Visualizing Data

Before we begin fitting our models, let's look visualize some of our data to gain a better understanding of how the variables work and how they relate to each other.

Average Stars

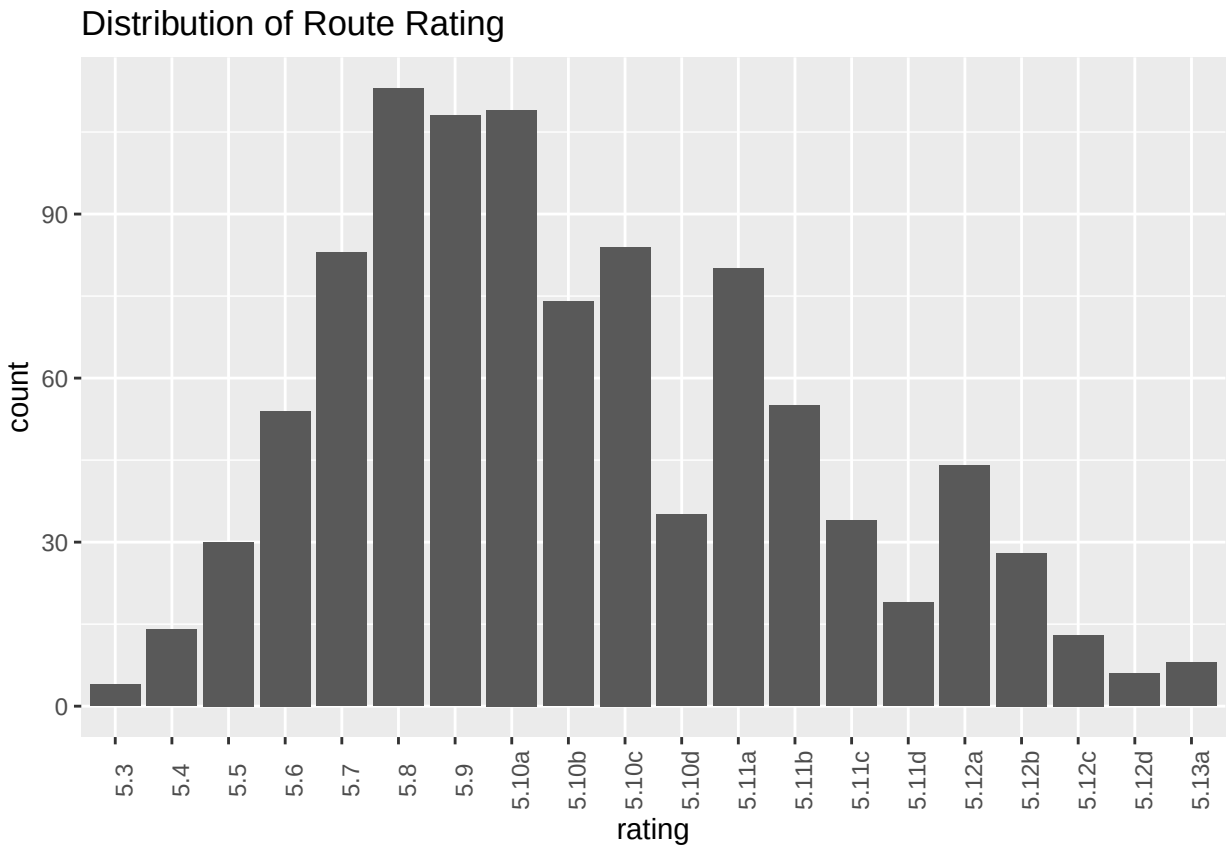
```
ggplot(boulder, aes(avg_stars)) +
  geom_bar(fill = 'red') +
  labs(title = 'Distribution of Average Stars')
```



First let's look at the distribution of the outcome variable, `avg_stars`. We can see that the variables are approximately normally distributed, with a large peak between 2 to 3 stars, meaning most climb ratings have a quality rating in the range of 2 to 3 stars.

Route Ratings

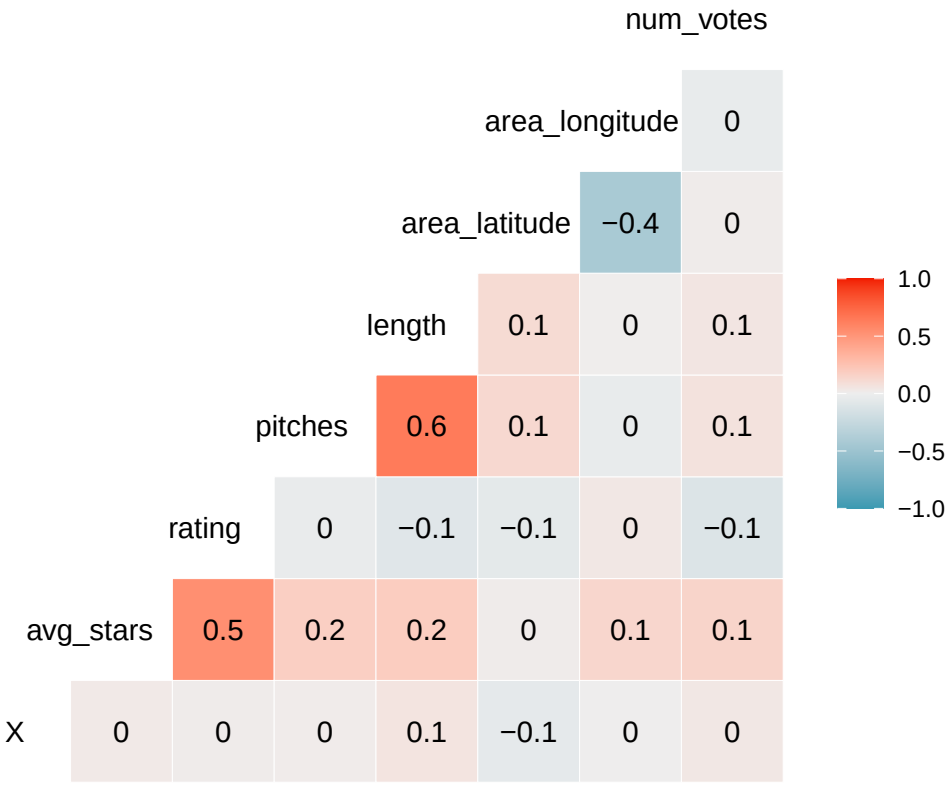
```
ggplot(boulder, aes(x = rating)) +
  geom_bar() +
  theme(axis.text.x = element_text(angle = 90)) +
  labs(title = 'Distribution of Route Rating')
```



Next, we look at the distribution of `rating`. We can see that they are also approximately distributed, with a peak around the point where route difficulties transition from beginner to intermediate, specifically at ratings 5.7 to 5.10a.

Correlation plot

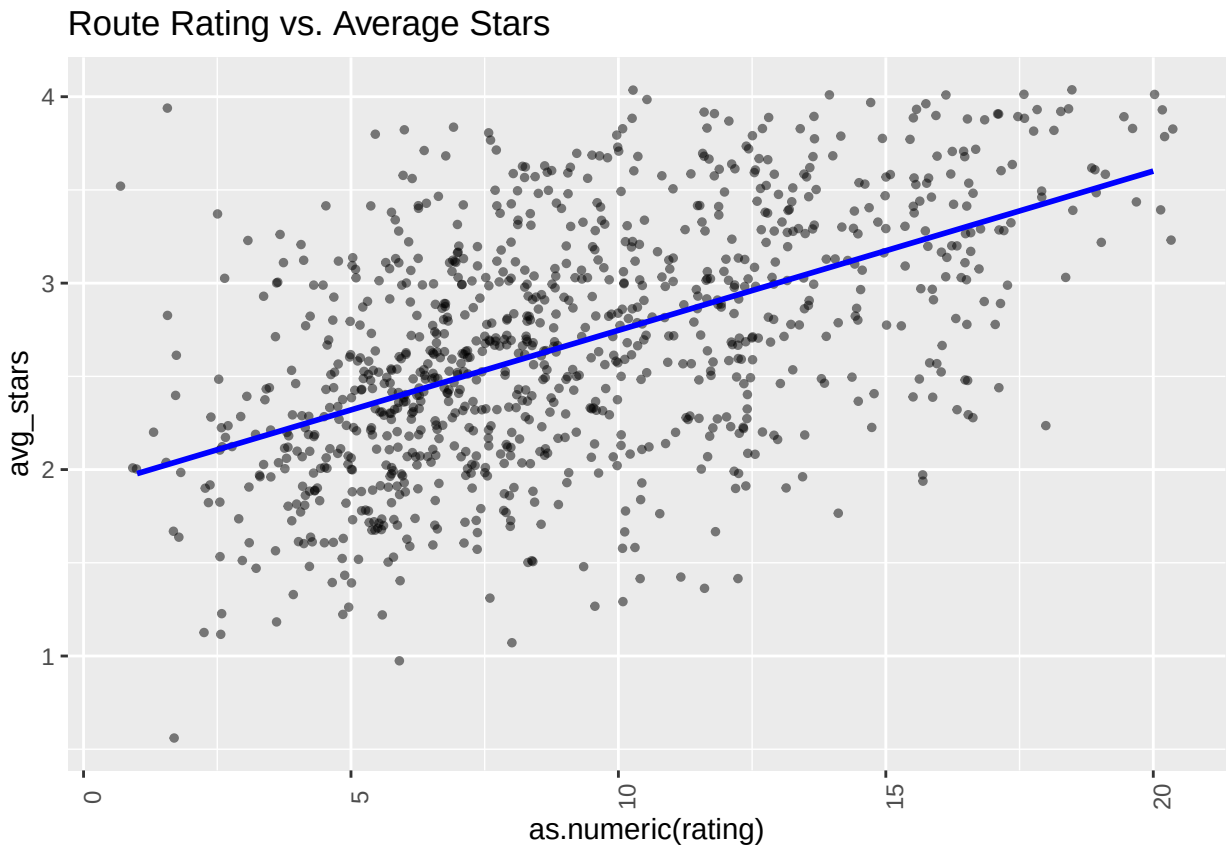
```
boulder %>%
  mutate(rating = as.numeric(rating)) %>%
  ggcorr(hjust = 0.75, label = T, layout.exp = 1)
```



Now we explore the relationships of the variables with each other. From the correlation plot, we can see that our outcome `avg_stars` is positively correlated with `rating`. This tells us that the quality rating of a route does tend to be higher for more difficult climbs. 'avg_stars' also has a slight positive correlation with `pitches` and `length`, which are highly positively correlated with each other. Additionally, `area_latitude` and `area_longitude` are negatively correlated.

Route Rating and Average Stars

```
ggplot(boulder, aes(x = as.numeric(rating), y = avg_stars)) +
  geom_jitter(width = .5, alpha = 0.5, size = 1) +
  geom_smooth(method = lm, se = F, col = 'blue') +
  labs(title = 'Route Rating vs. Average Stars') +
  theme(axis.text.x = element_text(angle = 90))
```



To further explore the relationship between `avg_stars` and `rating`, I have created a plot of route difficulty rating against its average quality rating. We can see from the plot that `rating` has a positive linear relationship with `avg_stars`. This relationship is shown by the life of best fit in the plot.

Describing Variables

- `route_type`: The type of climb the route is. It can either be sport, trad, or both.
- `rating`: The difficulty of the route.
- `pitches`: How many pitches the route is. A pitch is the length between anchor points in a climb, which is no longer than the length of rope used. Routes with multiple pitches typically require multiple ropes.
- `length`: How tall the climb is.
- `area_latitude`: GPS latitude of the climb.
- `area_longitude`: GPS longitude of the climb
- `num_votes`: How many people voted on the quality of the climb.

Setting Up Models

Now that the data is cleaned up and we understand how behavior of the data, we can start fitting our models.

Data Split

Our first step in fitting the models is splitting the data into a training and testing set. The training set will be used to create, or train, the model. After fitting our models, the testing set will be used to test each model's performance on new data. I chose to go with a 70/30 split, meaning the testing data will contain

70% of the data and the testing set will have the other 30% of the data. By splitting our data in this way, we can avoid overfitting since the model is not using all of the available data to learn. The split is stratified on the outcome variable, `avg_stars`, to ensure that the training and testing data accurately represents the ourcome variable.

```
# Set seed for reproducability
set.seed(131)

# Splitting the data
boulder_split <- initial_split(boulder, prop = 0.7, strata = avg_stars)
boulder_train <- training(boulder_split)
boulder_test <- testing(boulder_split)

# Make sure the data is split correctly
nrow(boulder_train) + nrow(boulder_test) == nrow(boulder)
```

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

```
dim(boulder_train)
```

```
## [1] 694  10
```

```
dim(boulder_test)
```

```
## [1] 301  10
```

We can see that the training set has 694 observations and the testing set has 301 observations, meaning the data was split correctly.

Creating a Recipe

Since we use the same predictors, model conditions, and response variable, our next step is creating a universal recipe for all of our models to work with.

We will be using the same 7 predictors described earlier. We have already turned our categorical predictors, `rating` and `route_type`, into factors, so for the recipe we just need to turn them into dummy variables. After this, we normalize the variables by centering and scaling them.

```
boulder_recipe <- recipe(avg_stars ~ rating + route_type + pitches + length +
                           area_latitude + area_longitude + num_votes,
                         data = boulder_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
```

K-Fold Cross Validation

We will also create 10 folds to conduct k-fold stratified cross validation. We will be stratifying the folds on our response variable, `avg_stars`.

```r
boulder_folds <- vfold_cv(boulder_train, v = 10, strata = avg_stars)
```

Model Building

Now that we have split our data and created our folds for cross validation, we are ready to build our models. Since some of the models take a long time to run, I saved the tuned models to separate files to avoid having to run the models every time. I chose to use Root Mean Squared Error (RMSE) as the metric I will be using to evaluate the performance of each model. RMSE shows the the distance of model predictions from the true values. Considering this, our goal is to minimize the RMSE as a lower RMSE means that the predicted values are closer to the true values.

The models we will be fitting are Linear Regression, K-Nearest Neighbors, Elastic Net Regression, and Random Forest. Each model is fit in a fairly similar and straightforward way, so I will be describing each step of the model fitting process for all of the models.

1. Set up models by specifying what type of model we want to fit, the engine that the model comes from, the parameters we wish to tune, and the mode of each model, which will always be 'regression' as we are predicting a numerical value.

```r
# Linear Regression
lm_model <- linear_reg() %>%
  set_engine('lm')

# K-Nearest Neighbors
knn_model <- nearest_neighbor(neighbors = tune()) %>%
  set_mode('regression') %>%
  set_engine('kknn')

# Elastic Net
en_model <- linear_reg(mixture = tune(),
                       penalty = tune()) %>%
  set_mode('regression') %>%
  set_engine('glmnet')

# Random Forest
rf_model <- rand_forest(mtry = tune(),
                        trees = tune(),
                        min_n = tune()) %>%
  set_engine('ranger', importance = 'impurity') %>%
  set_mode('regression')
```

2. Set up the workflow for each model, then add the model and recipe to it.

```r
# Linear Regression
lm_wflow <- workflow() %>%
  add_model(lm_model) %>%
  add_recipe(boulder_recipe)

# K-Nearest Neighbors
knn_wflow <- workflow() %>%
  add_model(knn_model) %>%
```

```
  add_recipe(boulder_recipe)

# Elastic Net
en_wflow <- workflow() %>%
  add_model(en_model) %>%
  add_recipe(boulder_recipe)

# Random Forest
rf_wflow <- workflow() %>%
  add_model(rf_model) %>%
  add_recipe(boulder_recipe)
```

We will skip steps 3-5 for Linear Regression since it is a simpler model that does not require hyperparameter tuning.

3. Set up the tuning grid with the parameters we wish to tune for each model, specifiyng the range and levels of each.

```
# K-Nearest Neighbor
knn_grid <- grid_regular(neighbors(range = c(1, 10)),
                         levels = 10)

# Elastic Net
en_grid <- grid_regular(penalty(range = c(0.01, 0.1), trans = identity_trans()),
                        mixture(range = c(0, 1)),
                        levels = 10)

# Random Forest
rf_grid <- grid_regular(mtry(range = c(1,7)),
                        trees(range = c(200, 1000)),
                        min_n(range = c(5, 40)),
                        levels = 8)
```

4. Tune each model, specifying the workflow, k-fold cross validation folds, and the tuning grid for our chosen parameters.

```
# K-Nearest Neighbor
knn_tune <- tune_grid(
  object = knn_wflow,
  resamples = boulder_folds,
  grid = knn_grid
)

# Elastic Net
en_tune <- tune_grid(
  object = en_wflow,
  resamples = boulder_folds,
  grid = en_grid
)

# Random Forest
rf_tune <- tune_grid(
```

```
  object = rf_wflow,
  resamples = boulder_folds,
  grid = rf_grid
)
```

5. Save the tuned models to RDS files. Tuning the models can take some time, so we save them to avoid having to run them every time. We then load them back in.

```
save(knn_tune, file = 'models/knn_tune.rds')
save(en_tune, file = 'models/en_tune.rds')
save(rf_tune, file = 'models/rf_tune.rds')
```

```
load(file = 'models/knn_tune.rds')
load(file = 'models/en_tune.rds')
load(file = 'models/rf_tune.rds')
```

6. Collect the metrics of our tuned models. We save the model that had the lowest RMSE to a variable so we can call it later for comparing our models and fitting to the training data. Since some of the models, can become very complex, I chose the most simple model within one standard deviation of the best RMSE. This can help prevent overfitting and improve model generalizability for new values.

```
# Linear Regression
# Fit the Linear Regression to the folds since it was not tuned.
lm_fit <- fit_resamples(lm_wflow, resamples = boulder_folds)
lm_best <- collect_metrics(lm_fit) %>%
  slice(1)


# K-Nearest Neighbor
knn_best <- select_by_one_std_err(knn_tune,
                                  metric = 'rmse',
                                  desc(neighbors))

# Elastic Net
en_best <- select_by_one_std_err(en_tune,
                                 metric = 'rmse',
                                 penalty,
                                 mixture)

# Random Forest
rf_best <- select_by_one_std_err(rf_tune,
                                 metric = 'rmse',
                                 mtry,
                                 trees,
                                 min_n)
```

Model Results

Now that we have tuned all of our models, we can compare the results of each model to find which ones performed best.

RMSE of Models

I have created a table of each model and its respective RMSE to easier assess which model had the lowest RMSE

```
compare_best_models <- tibble(Model = c('Linear Regression', 'K-Nearest Neighbors',
                                        'Elastic Net Regression', 'Random Forest'),
                              RMSE = c(lm_best$mean, knn_best$mean, en_best$mean,
                                       rf_best$mean)) %>%
  arrange(RMSE)

save(compare_best_models, file = 'data/compare_best_models.rds')
```

```
load('data/compare_best_models.rds')
compare_best_models
```

```
## # A tibble: 4 x 2
##   Model                  RMSE
##   <chr>                 <dbl>
## 1 Linear Regression     0.476
## 2 Elastic Net Regression 0.478
## 3 Random Forest         0.486
## 4 K-Nearest Neighbors   0.519
```

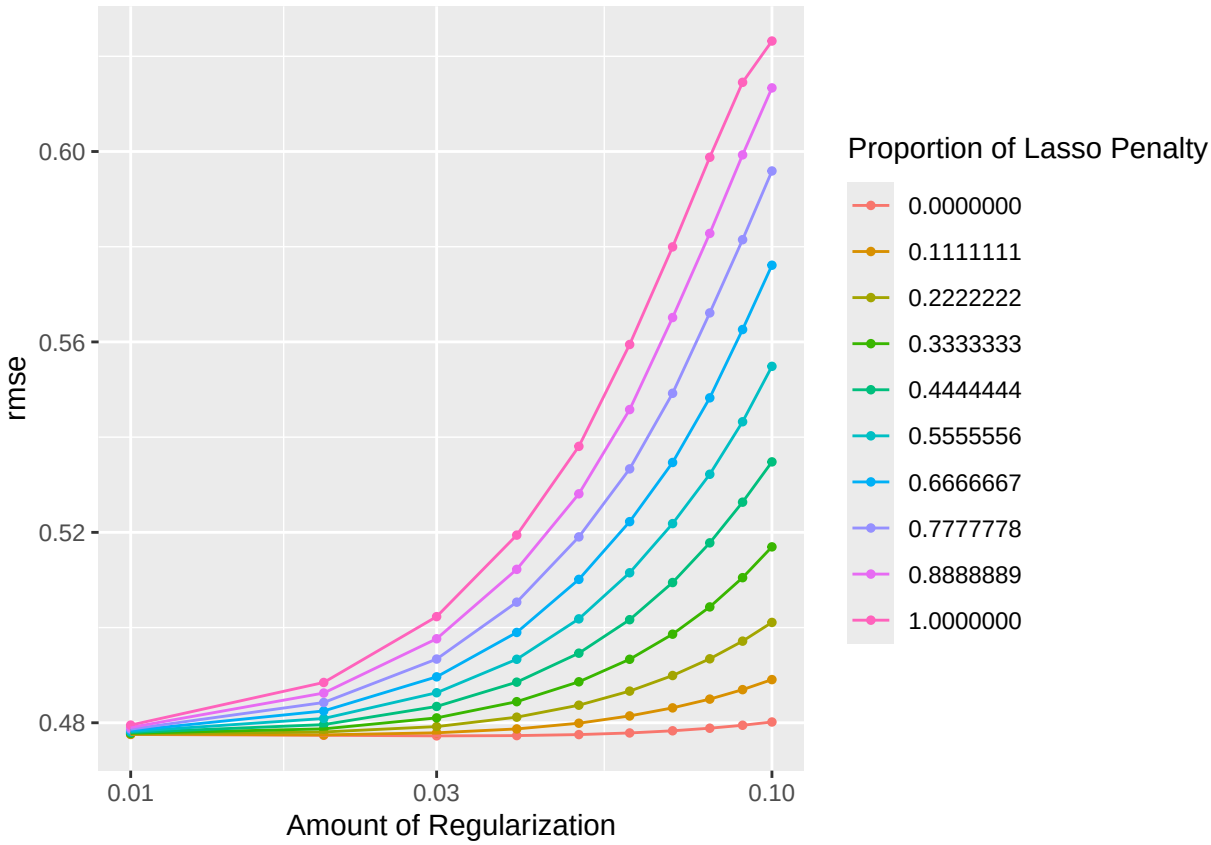
From the table, we can see that the Linear Regression and Elastic Net Regression models performed as they had the lowest RMSE. This indicated that our data is likely linear.

Model Autoplots

The `autoplot` function in R allows us to easily visualize the performance of each model. This plot will show us how each tuned parameter effects the RMSE of the model.

Elastic Net Regression

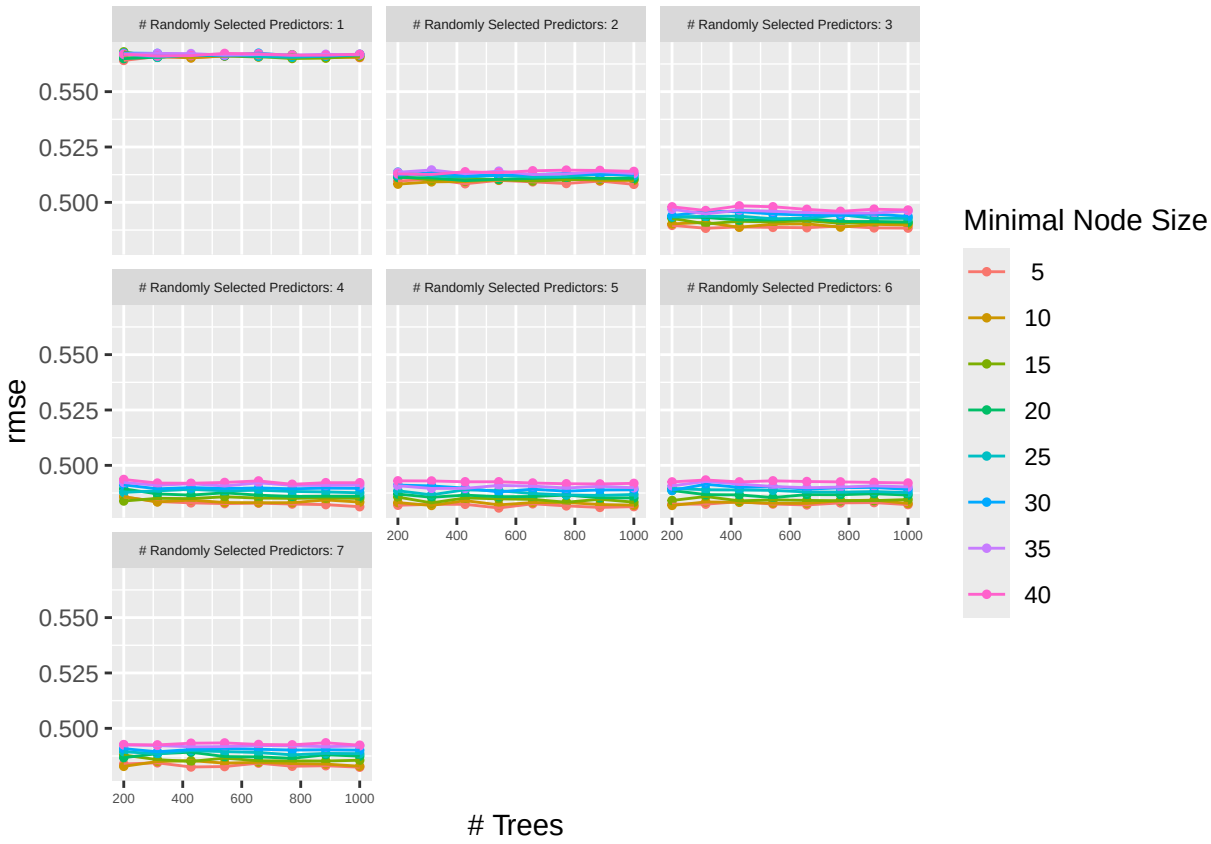
```
autoplot(en_tune, metric = 'rmse')
```



For our Elastic Net model, we tuned the parameters penalty and mixture at 10 different levels. The plot shows us that our Elastic Net model performs best at lower penalty values. As the penalty increases, this model shrinks the predictors to very small values, in turn making it harder for the model to predict. We can also see that as mixture increases, so does RMSE. The mixture value represents the proportion of lasso regression and ridge regression used in our model, with a mixture of 0 being ridge regression and a mixture of 1 being lasso regression. We can see that ridge regression performs best on our data.

Random Forest

```
autoplot(rf_tune, metric = 'rmse') +
  theme(strip.text.x = element_text(size = 5),
        axis.text.x = element_text(size = 5))
```



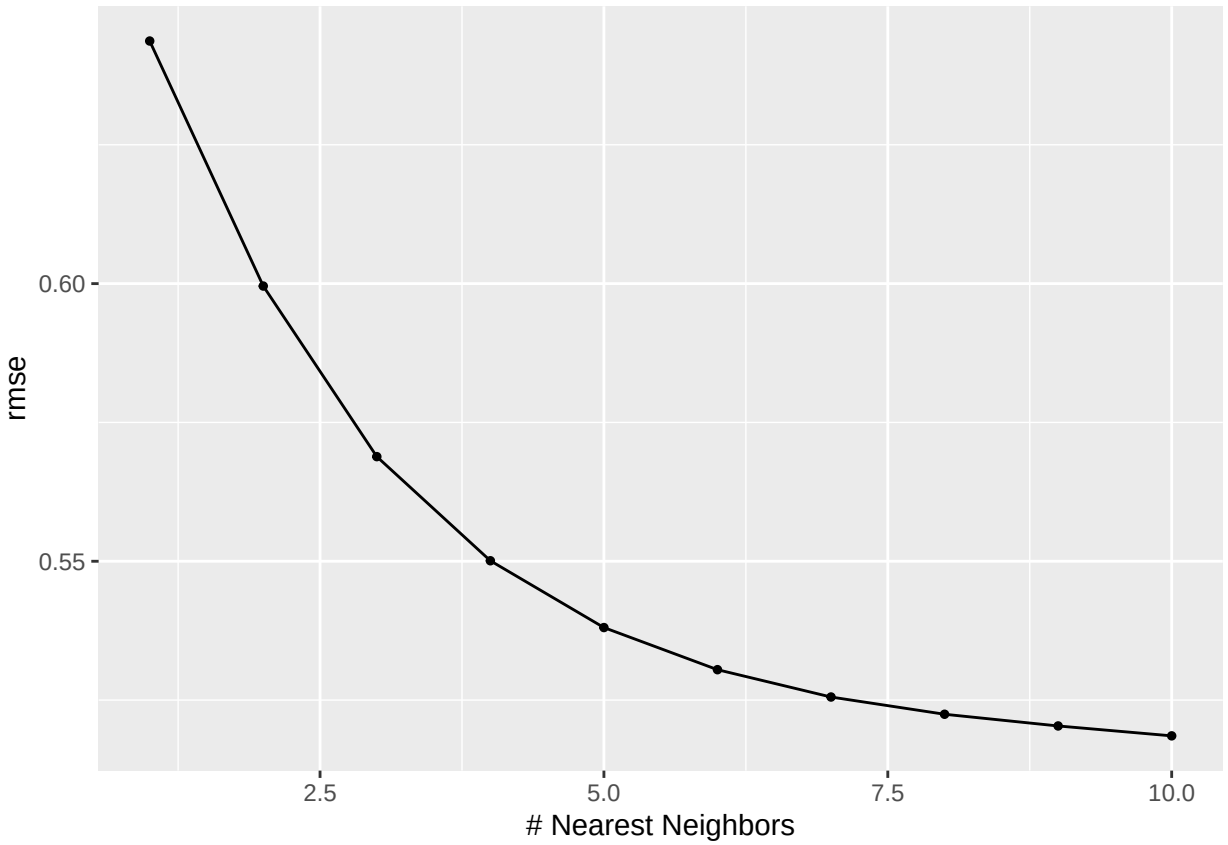
For our random forest, we tuned 3 parameters:

- mtry: The number of predictors randomly sampled for the tree to make its decisions
- trees: The number of trees grown in each forest
- min_n: The minimum number of data values needed to create another split in a tree

As the number of predictors increase, the RMSE of the model becomes lower. This is because as more predictors are used to decide how to split the tree, the more accurate the model becomes. The number of trees does not seem to have much effect on the performance of the model. The minimum node size also seems to have little effect on the performance of the model, however smaller node sizes seem to perform better. The number of predictors used seems to have the greatest affect on the performance of our Random Forest model.

K-Nearest Neighbors

```
autoplot(knn_tune, metric = 'rmse')
```



From the plot of K-Nearest Neighbors, we can see that as the number of neighbors increases, the better our model performs.

Best Model

Performance on the Folds

The Linear Regression model performed the best out of all of our models. Since there were no tuning parameters, this makes interpreting the model much easier.

Testing the Model

Now we can take our tuned linear regression model and fit it back to the training data. This will allow us to train the model again on the entire training dataset. Then, we can finalize the model and our testing data.

```
# Fit to training data
lm_final_wflow <- finalize_workflow(lm_wflow, lm_best)
lm_final_fit <- fit(lm_final_wflow, data = boulder_train)

save(lm_final_fit, file = 'models/lm_final_fit_train.rds')

final_rmse <- augment(lm_final_fit, new_data = boulder_test) %>%
  rmse(truth = avg_stars, estimate = .pred)

save(final_rmse, file = 'data/final_rmse.rds')
```

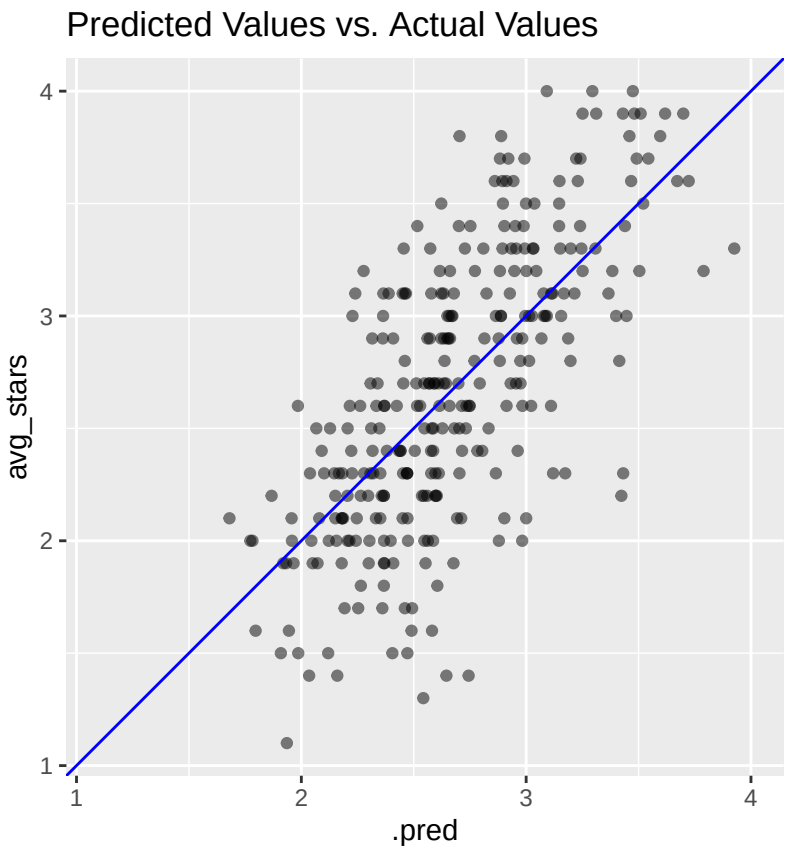
```
load('models/lm_final_fit_train.rds')
load('data/final_rmse.rds')
final_rmse
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.490
```

Our Linear Regression model performed slightly worse on the testing data, but this is to be expected. Regardless, the difference is so small that it is negligible. The testing RMSE of our model is 0.4903828. Though the RMSE is not amazing, it is also not bad.

We can plot the predicted values against the actual values of the testing data to help us visualize the accuracy of our model:

```
augment(lm_final_fit, new_data = boulder_test) %>%
  ggplot(aes(x = .pred, y = avg_stars)) +
  geom_point(alpha = 0.5) +
  geom_abline(color = 'blue') +
  coord_obs_pred() +
  labs(title = 'Predicted Values vs. Actual Values')
```



A perfect plot would show all of the dots in a perfectly straight line. Though this is not the case in our plot, the dots exhibit a linear trend, with most of them falling close to the line. This means that our model is capable of predicting a roughly accurate value of `avg_stars`.

Conclusion

Through our trial and tribulation in researching our data, fitting models, and testing them, we have come to the conclusion that the best model for predicting the quality rating of a climb is a linear regression model. This means that the data was about linear, which supports our earlier findings of route rating being significantly linearly correlated with quality rating. However, there is still room for improvement.

For this report, I only looked at lead climbing routes, but there are numerous other ways that one could climb. In a future report, it could be useful to include these other forms of climbing as predictors. Additionally, the routes on Mountain Project had descriptions of protections that are needed for each route, so including this as a predictor could be useful as well. Since some climbs may be rated higher, but require a lot of protection or other gear, they may have a different quality rating.

Overall, our model performed much better than I had expected. Going into this, a large concern of mine was that `avg_stars` would be completely uncorrelated with any characteristics of the route, making the quality rating essentially random. It was a great relief learning that the quality rating is actually possible to predict. This project taught me a lot about climbing and in a way helped reignite my enjoyment of the sport. The opportunity to research and explore the data about climbing has also ignited a desire to learn more about machine learning and building models.