Code

Predicting the Capture Rate of Pokemon

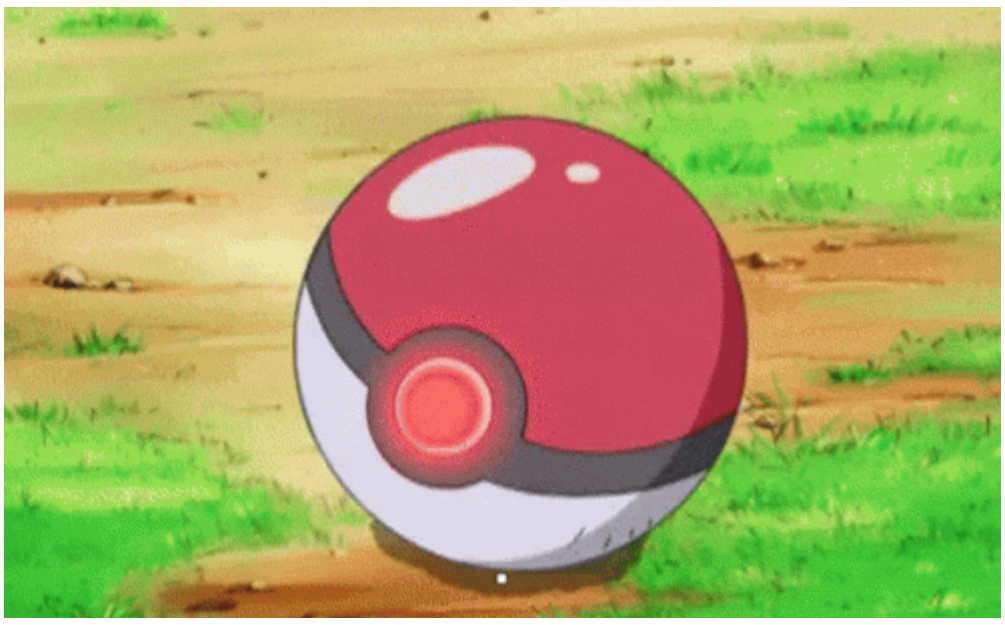
PSTAT 131 Final Project

Shivani Kharva

2022-12-11

Introduction

The purpose of this project is to develop a model that will predict the capture rate of a Pokemon.



What Are Pokemon?

Pokemon (or “pocket monsters”) are creatures from the Japanese Nintendo video game, Pokemon. Pokemon is a role-playing game in which players play as trainers who capture the Pokemon using Poke Balls and train them to compete in battles with other trainers. As they train and gain more experience through battle, Pokemon evolve and grow stronger. There are over 800 Pokemon with 18 types (i.e. fire, water, etc,), and they have numerous abilities (both common and unique).

Pokemon in battle!

What Are We Trying to Do?

However, a question I’d like to pose is “How hard is it to capture a Pokemon in the first place depending on their existing statistics?” It seems quite reasonable to hypothesize that a Pokemon’s capture rate would vary depending on variables like their type, base statistics, generation, etc. So, I would like to test if it is possible to develop a model that can predict the capture rate of a Pokemon based on those qualities.

Pokemon stats screen.

Why?

Players, including me, are often only able to see the aforementioned qualities of a Pokemon and not their actual capture rate. So, it may be useful to have a specific model that can accurately predict the capture rate of a Pokemon so a player may see how difficult it may be to capture that Pokemon. The player can then adjust their strategy if a Pokemon is harder to catch by using more advanced equipment, such as using an Ultra Ball instead of a Poke Ball or giving the Pokemon a fruit before attempting to capture them to slightly lower their guard. So, let's see if the variation in the capture rate of a specific Pokemon can be explained by these qualities!

Loading Packages and Data

First, let's load in all of our packages and the raw Pokemon data.

Code

abilities <chr>	against_bug <dbl>	against_dark <dbl>	against_
['Overgrow', 'Chlorophyll']	1.00	1	
['Overgrow', 'Chlorophyll']	1.00	1	
['Overgrow', 'Chlorophyll']	1.00	1	
['Blaze', 'Solar Power']	0.50	1	
['Blaze', 'Solar Power']	0.50	1	
['Blaze', 'Solar Power']	0.25	1	

6 rows | 1-4 of 41 columns

This data was taken from the Kaggle data set, "Pokemon Stats Dataset (https://www.kaggle.com/datasets/hasanarcas/pokemon-stats-dataset)," and it was scraped from the official Pokemon Database (https://pokemondb.net/pokedex/all) by user Hasan Arcas.

Exploring and Tidying the Raw Data

Variable Selection

Let's mess around with the data a little bit to see what we're currently working with!

Code

```
## [1] 801  41
```

The data set contains 801 rows and 41 columns. This means we have 801 Pokemon and 41 variables. One of those variables, `capture_rate` , is our response, but that leaves us with 40 predictor variables! We're definitely going to have to narrow that down!

From the data, each of the variables that specify each Pokemon's skill against a certain type of Pokemon (i.e. `against_bug` , `against_dark` , etc.) do not appear to be as relevant as other attributes in deciding whether specifc features about a Pokemon affect its capture rate. We can also assume that the `attack` variable gives us an overall estimate of each Pokemon's attack skill so we would not need to go into such great detail as finding out how the Pokemon perform against every other type of Pokemon. For that reason, these 18 columns can be dropped.

At first, it seems that the `classification` variable may be helpful; however, let us see how many unique classifications of a Pokemon there are.

Code

n
<int>
588

1 row

There are 588 unique classifications! That seems a little too high for our analysis so we will just stick to using the `type` of the Pokemon instead.

Furthermore, for the `abilities` variable, we can see that there is no set amount for how many abilities a Pokemon has or a measure to figure out which abilities are more powerful than others. For these reasons, it may be best to avoid including this column into our analysis.

The `base_egg_steps` variable is a variable not used in the original Pokemon games so, to keep the data consistent with the game it was taken from, this variable should also be omitted.

Finally, `pokedex_number` and `japanese_name` are unique to each Pokemon and are mainly used to identify the Pokemon; however, since we already have the `name` variable, it would not be useful to include these variables.

Now, let's filter out all those unwanted variables and our dataset will be ready for further tidying!

Code

Tidying the Outcome

Upon further inspection, we can see that our outcome, `capture_rate` , is actually a character variable, which is not what we want. By looking into the values, we see that one of the Pokemon, Minior, actually has two capture rates.

Code

name	capture_rate
<chr>	<chr>
Minior	30 (Meteorite)255 (Core)

1 row

By looking at Bulbapedia (https://bulbapedia.bulbagarden.net/wiki/Minior_(Pok%C3%A9mon)), a community Pokemon encyclopedia, we can see that Minior has a relatively low capture rate at 30 when it is "Meteorite," meaning its shell has not been broken. However, once Minior's shell is broken ("Core"), the Pokemon shoots up to a capture rate of 255, which is the highest capture rate there is. Therefore, it would make sense to keep the capture rate of Minior when it is still Meteorite.

Left: Minior with its shell (Meteorite). Right: Minior without its shell (Core).

Code

name <chr>	capture_rate <int>
Minior	30

1 row

Now the capture rate for Minior is fixed and our outcome is an integer we can use in regression!

Tidying the Abilities Variable

One last thing we need to do before our exploratory data analysis is tidy the `abilities` variable because we currently have a column made up of lists of each Pokemon's abilities. However, it is not very useful to analyze 801 unique lists of abilities. It may be better to prioritize the abilities that are the most common among the Pokemon. Let's look at the top 5 most common abilities and their capture rates.

Code

abilities <chr>	count <int>	capture_rate <dbl>
Sturdy	41	102.2683
Swift Swim	38	110.6579
Keen Eye	37	132.0270
Chlorophyll	35	123.7143
Levitate	33	85.0303

5 rows

The top 5 abilities appear to have varying capture rates that are not too close to one another. So, it might prove meaningful to include whether a Pokemon has one of the top 5 most common abilities or not. Now, we must add in 5 more variables indicating whether a Pokemon has each of these abilities or not.

Code

Our dataset is now tidied and ready for some exploratory data analysis!

Exploratory Data Analysis

Loading Data and Converting Factors

First, we need to read and load in the data and convert any categorical variables to factors.

Code

abilities <chr>	attack <dbl>	base_happiness <dbl>	base_total <dbl>
['Overgrow', 'Chlorophyll']	49	70	318
['Overgrow', 'Chlorophyll']	62	70	405
['Overgrow', 'Chlorophyll']	100	70	625
['Blaze', 'Solar Power']	52	70	309
['Blaze', 'Solar Power']	64	70	405
['Blaze', 'Solar Power']	104	70	634

6 rows | 1-5 of 23 columns

Missing Data

Before we move on to exploring the variables in our data, we must check for any missing data because that could potentially cause issues.

Code

```
##   abilities            attack       base_happiness     base_to
tal
##  Length:801         Min.   :  5.00   Min.   :  0.00   Min.   :1
80.0
##  Class :character   1st Qu.: 55.00   1st Qu.: 70.00   1st Qu.:3
20.0
##  Mode  :character   Median : 75.00   Median : 70.00   Median :4
35.0
##                     Mean   : 77.86   Mean   : 65.36   Mean   :4
28.4
##                     3rd Qu.:100.00   3rd Qu.: 70.00   3rd Qu.:5
05.0
##                     Max.   :185.00   Max.   :140.00   Max.   :7
80.0
##
##   capture_rate       defense       experience_growth    height_
m
##  Min.   :  3.00   Min.   :  5.00   Min.   : 600000   Min.   :
0.100
##  1st Qu.: 45.00   1st Qu.: 50.00   1st Qu.:1000000   1st Qu.:
0.600
##  Median : 60.00   Median : 70.00   Median :1000000   Median :
1.000
##  Mean   : 98.68   Mean   : 73.01   Mean   :1054996   Mean   :
1.164
##  3rd Qu.:170.00   3rd Qu.: 90.00   3rd Qu.:1059860   3rd Qu.:
1.500
##  Max.   :255.00   Max.   :230.00   Max.   :1640000   Max.   :1
4.500
##                                                     NA's   :20
##  percentage_male        hp             name             sp_atta
ck
##  Min.   :  0.00   Min.   :  1.00   Length:801         Min.   :
10.00
##  1st Qu.: 50.00   1st Qu.: 50.00   Class :character   1st Qu.:
45.00
##  Median : 50.00   Median : 65.00   Mode  :character   Median :
65.00
##  Mean   : 55.16   Mean   : 68.96                      Mean   :
71.31
##  3rd Qu.: 50.00   3rd Qu.: 80.00                      3rd Qu.:
91.00
##  Max.   :100.00   Max.   :255.00                      Max.   :1
94.00
##  NA's   :98
##    sp_defense         speed            type1       weight_kg
generation
##  Min.   : 20.00   Min.   :  5.00   water  :114   Min.   :  0.10
1:151
##  1st Qu.: 50.00   1st Qu.: 45.00   normal :105   1st Qu.:  9.00
2:100
##  Median : 66.00   Median : 65.00   grass  : 78   Median : 27.30
3:135
##  Mean   : 70.91   Mean   : 66.33   bug    : 72   Mean   : 61.38
4:107
##  3rd Qu.: 90.00   3rd Qu.: 85.00   psychic: 53   3rd Qu.: 64.80
5:156
##  Max.   :230.00   Max.   :180.00   fire   : 52   Max.   :999.90
6: 72
##                                    (Other):327   NA's   :20
7: 80
##  is_legendary has_sturdy has_swift_swim has_keen_eye has_chloro
phyll
##  0:731        0:763      0:763          0:765        0:766
##  1: 70        1: 38      1: 38          1: 36        1: 35
##
##
##
##
##
##  has_levitate
##  0:768
##  1: 33
##
##
##
```

```
##
##
```

We can see that `height_m`, `percentage_male`, and `weight_kg` all have missing values: 20, 98, and 20, respectively. It appears that the variables with missing data relate more so to the physical characteristics of the Pokemon and not as much to the capabilities of the Pokemon. Because these are not very high proportions of missing data, we can deal with this issue by imputing the missing values using a linear regression of our other variables. This will be done later in the recipe creation!

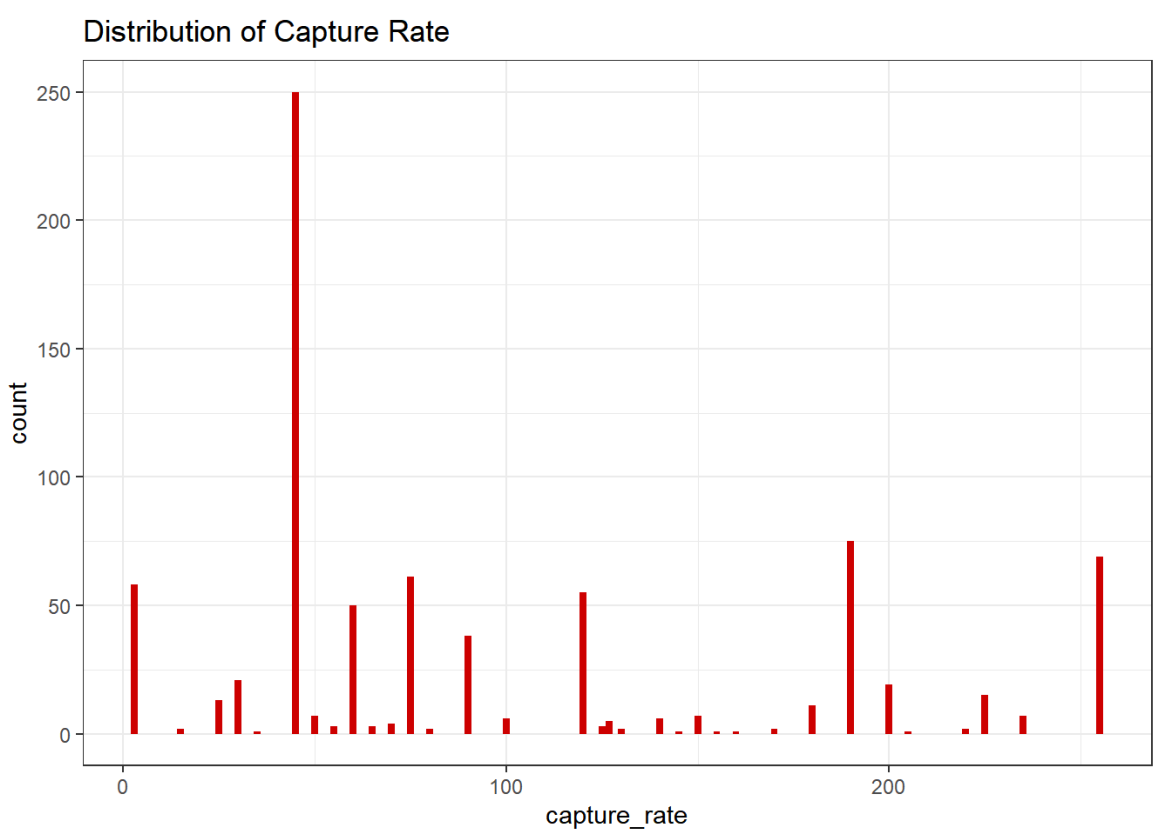
Visual EDA

Now, we will explore the relationships between select variables with the outcome as well as with each other!

Capture Rate

First, let's explore the distribution of our outcome, `capture_rate`.

Code

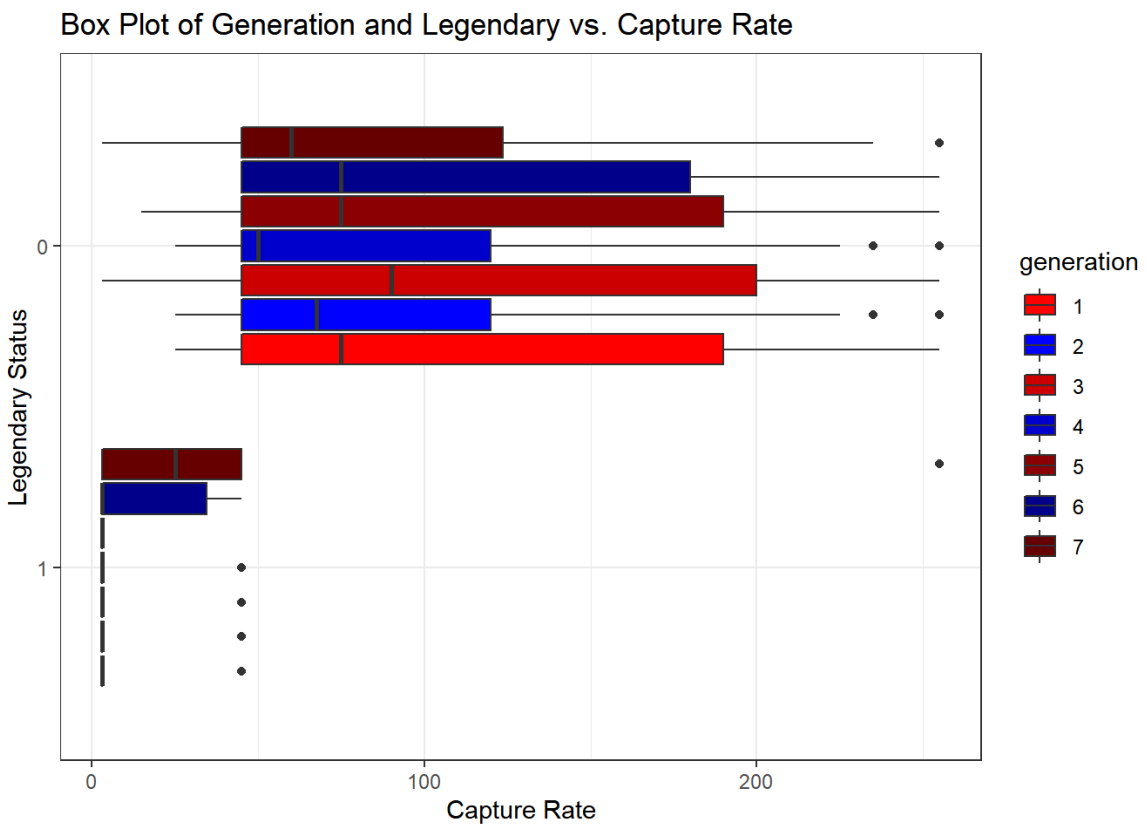


The possible values of `capture_rate` range from 0 to 255 and the `capture_rate` of the Pokemon in our data appears to range from 3 to 255. There is also a peak at about 45, meaning the most common capture rate for a Pokemon in our data is ~45.

Generation

The data contains 7 different generations of Pokemon. The generations contain Pokemon from each of the 7 generations of Pokemon games. This poses the question of whether there are specific generations that have higher capture rates overall than any other generations. Also, legendary Pokemon are especially powerful Pokemon that players rarely encounter. From this definition, it seems likely that it would be harder to capture a Pokemon that is legendary than one that is not. Let's check to see how these variables relate to one another and to the capture rate!

Code



From this plot, we can see that the medians, maxima, minima, and first quartiles of the capture rates for each of the generations are the same or quite close to one another. However, the third quartile of generations 2, 5, and 7 appear to be less than the other generations. This does not seem as if it will make much different as the rest of the overall distribution is quite similar between generations. So, it appears that `capture_rate` does not vary much depending on specifically `generation` alone.

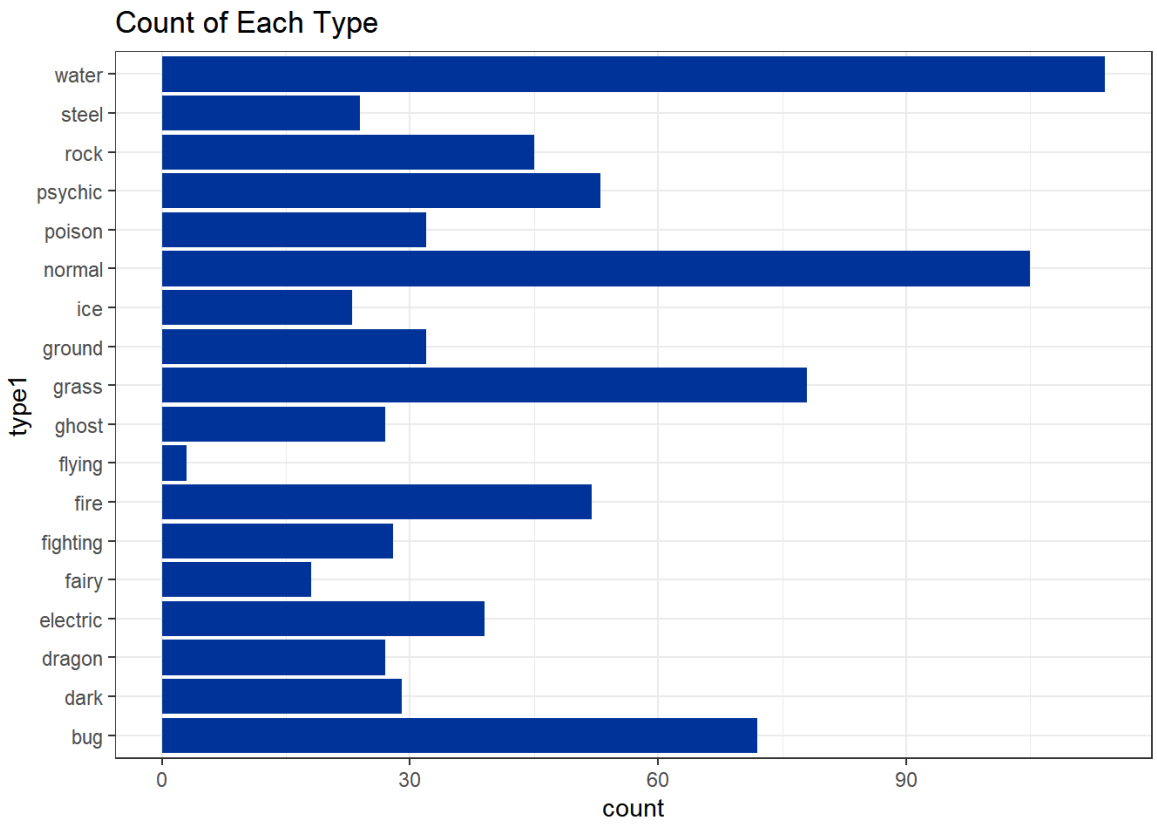
It is also apparent that there are very few Pokemon, if any, of legendary status in any generations other than 6 and 7. However, the distribution of generations 6 and 7 across legendary Pokemon still fit the aforementioned observations.

As for legendary status, from the plot, we can see that, although there are still non-legendary Pokemon in the lower end of the capture rate, there are no legendary Pokemon with a capture rate above 50. All legendary Pokemon capture rates appears to be less than all first quartiles of non-legendary Pokemon capture rates. The data implies that legendary Pokemon genearally have a lower capture rate than non-legendary Pokemon, which is what was expected.

Type 1

In our analysis, we are only using Type 1 of each Pokemon since not all Pokemon have a Type 2. However, how many types of Pokemon are there?

Code

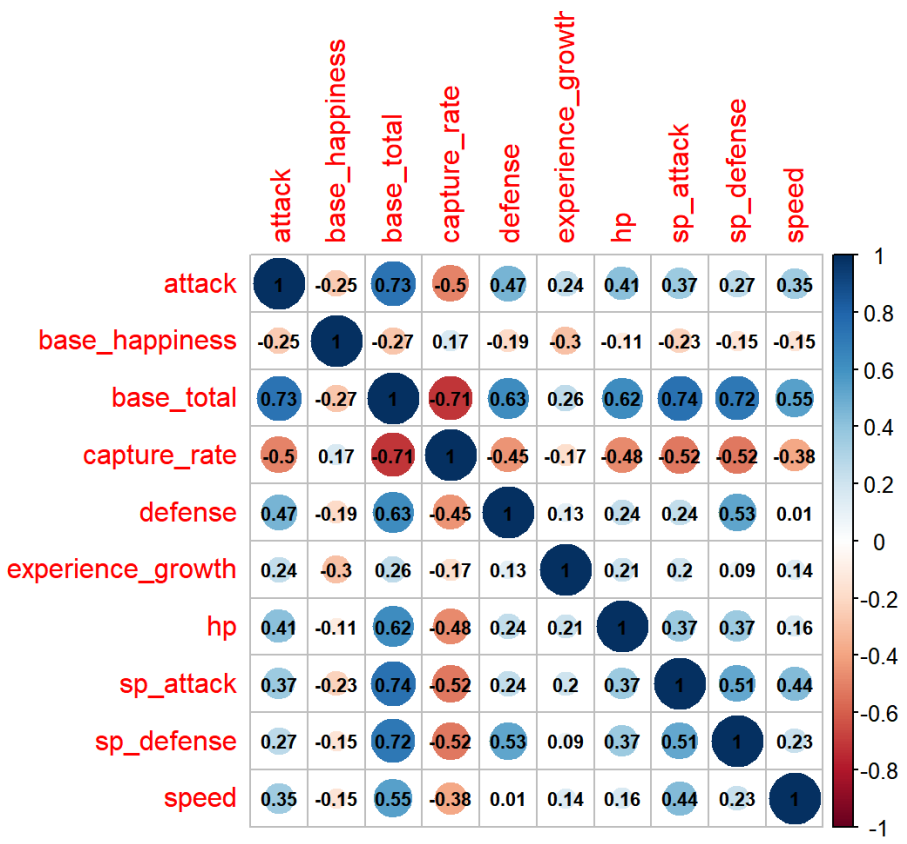


From the plot, we can see that there are 18 different types for type 1 Pokemon. 18 types is a lot of categories to include! However, the plot also reveals that there are significantly more Pokemon in some types like “water” and “normal” in comparison to other types with much fewer Pokemon like “flying” and “fairy”. So, in our analysis we should specify that we only want to prioritize the types with the most Pokemon and group the rest of the types into “Other”. This will be completed and further discussed in the recipe creation.

Correlation Plot

Now, let's explore the overall relationships between all of the continuous, non-missing variables using a correlation plot.

Code



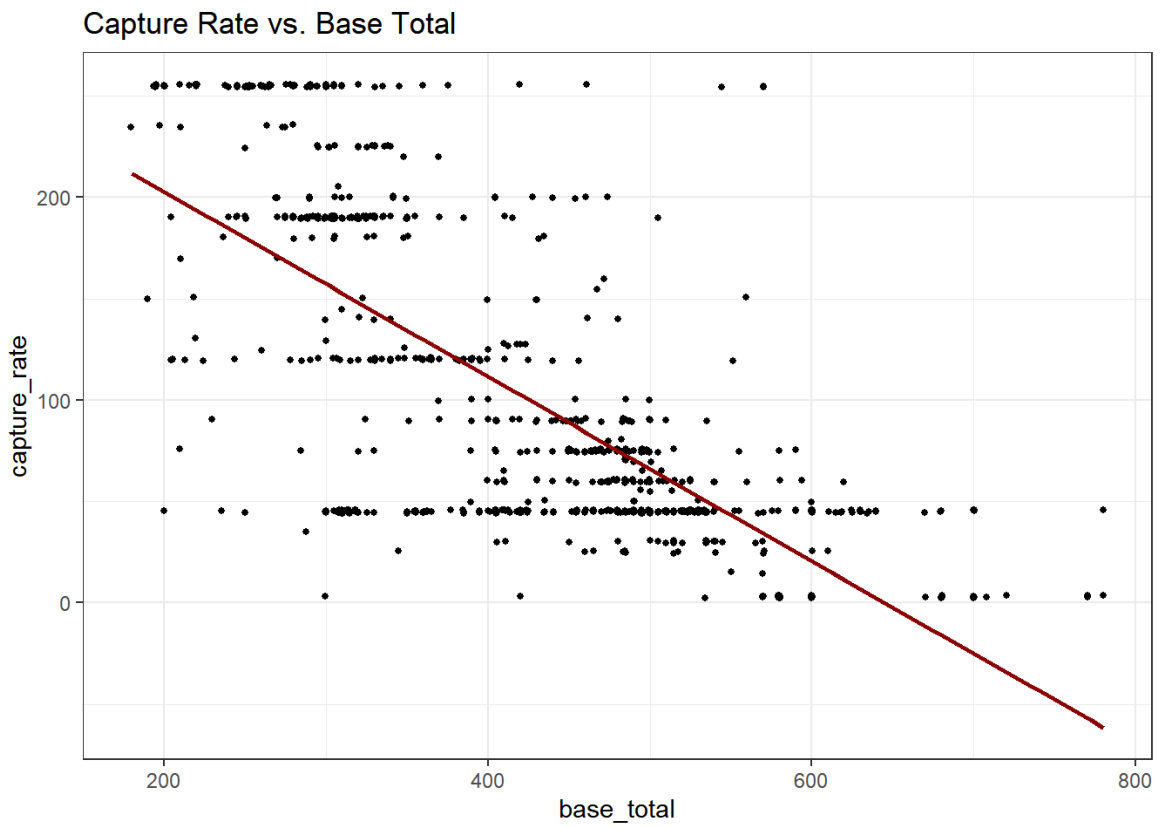
One relationship that stands out is the moderately negative correlation between `base_total` and `capture_rate` (-0.71), which indicates that it may be useful to use `base_total` in our analysis rather than the six individual battle statistics (we cannot use all of them because the battle statistics add up to the exact `base_total`). This relationship implies that, as a Pokemon has a higher total of battle statistics, it becomes more difficult to capture them (lower `capture_rate`).

Another apparent relationship is that all six battle statistics are moderately positively correlated with `base_total` and moderately negatively correlated with `capture_rate`. This implies that, as the battle statistics increase, so does their total, which makes sense as they add up to the `base_total`. Also, as the six battle statistics increase, the Pokemon becomes more difficult to capture (lower `capture_rate`). Since the battle statistics have a similar relationship with `capture_rate` as `base_total`, this further justifies using only `base_total` rather than the six individual battle statistics.

Base Total

Since we discussed the relationship between `base_total` and `capture_rate` in the correlation plot, perhaps we should further explore the relationship between the two variables.

Code



Although the relationship is not very strong, it is apparent that there is a moderately negative relationship between `capture_rate` and `base_total`. That is, a higher value of total battle statistics slightly relates to a lower capture rate. This makes contextual sense because a Pokemon with greater battle abilities would be more valuable (and

harder to capture because of its greater value). So, it seems as if it would prove more meaningful to use `base_total` rather than the six battle statistics in exploring the relationship between the predictors and the outcome, `capture_rate`.

Setting up for the Models

It's time to start fitting models to our data to see if we really can predict the capture rate of a Pokemon based on the predictors we have. However, we first have to set up our data by splitting it, creating the recipe, and creating folds for k-fold cross validation.

Data Split

The first step we have to take before fitting any models is splitting our data into a training and testing set. The training set will be used to do just that: train our models. The testing set acts as a test in the sense that our models will not be able to train on that data. So, once we fit whichever model we decide is the "best" (usually based on lowest RMSE or root mean squared error for regression) to our testing set, we will see how it truly performs on new data. By splitting our data into testing and training sets, we avoid over-fitting because the model is not using all of the available data to learn. The split I have chosen is a 70/30 split, which means 70% of the data will go towards the training set and 30% of the data will go towards the testing set. This way, most of our data is being used to train the model; however, we still have an adequate amount of data to test the model on. Also, the split is stratified on the outcome variable, `capture_rate`, to ensure that both the training and the testing data have an equal distribution of `capture_rate`.

Code

Before we move on, we should verify that the data split correctly.

Code

```
## [1] 0.6978777
```

Code

```
## [1] 0.3021223
```

The training set has about 70% of the data and the testing set has about 30% of the data. So, the data was split correctly between the training and testing sets.

Recipe Creation

Throughout our analysis, we will be using essentially the same predictors, conditions, and outcome. So, we will create one universal recipe to use for all of our models (and slightly adjust this recipe if needed). Each model will take this recipe and work with it using the methods associated with that model.

We will be using the same 14 predictors: `base_total`, `base_happiness`, `experience_growth`, `height_m`, `percentage_male`, `weight_kg`, `type1`, `generation`, `is_legendary`, `has_sturdy`, `has_swift_swim`, `has_keen_eye`, `has_chlorophyll`, and `has_levitate`. As mentioned in the exploratory data analysis, it would be best to use `base_total` instead of the individual six battle statistics because it has a higher negative correlation with the outcome and also acts as a summary of each Pokemon's six battle statistics.

In order to deal with missingness, we will impute the missing data in `percentage_male`, `height_m`, and `weight_kg` using a linear regression of all of the other predictors in the recipe.

Since `type1` has 18 types, we will prioritize the types made up of the most Pokemon and group the rest of the Pokemon into type 'Other'.

We will make `type1`, `generation`, `is_legendary`, `has_sturdy`, `has_swift_swim`, `has_keen_eye`, `has_chlorophyll`, and `has_levitate` into dummy variables because they are categorical and not continous.

Finally, we normalize our variables by centering and scaling.

Code

K-Fold Cross Validation

We will create 10 folds to conduct k-fold (10-fold in our case) stratified cross validation. This means that R is taking the training data and assigning each observation in the training data to 1 of 10 folds. For each fold, a testing set is created consisting of that fold

and the remaining k-1 folds will be the training set for that fold. At the end, we end up with k total folds.

K-fold cross validation is done by splitting the data into k folds as described above with each fold being a testing set with the other k-1 folds being the training set for that fold. Then, whichever model we are fitting is fit to each training set and tested on the corresponding testing set (each time, a different fold should be used as a validation set). Then, the average accuracy is taken from the testing set of each of the folds to measure performance (other metrics can be taken as well such as standard error).

We use k-fold cross validation rather than simply fitting and testing models on the entire training set because cross validation provides a better estimate of the testing accuracy. It is better to take the mean accuracy from several samples instead of just one accuracy from one sample because, as n increases, we reduce variation.

We stratify on the outcome, `capture_rate` , to make sure the data in each fold is not imbalanced.

Code

Model Building

It is now time to build our models! Since the models take a very long time to run, the results from each of the models has been saved to avoid rerunning the models every time. I have chosen Root Mean Squared Error (RMSE) as my metric because it works as an overall metric for all models. The RMSE is one of the most commonly used measures for evaluating the performance of regression models by showing how far the model's predictions are from the true values using Euclidian distance. So, a lower RMSE is better since that means the predicted values have a smaller distance from the actual values. Since RMSE measures distance, it is important that we normalize our data, which we did in the recipe. Also, I have fit 8 models to the Pokemon data; however, we will only be conducting further analysis on the 4 best-performing models. Let's get to building our models!

Fitting the models

Each of the models had a very similar process. I will detail it below and include the code for each of the models under that step (however, the code will not be evaluated here to save time).

For each of the models, you must conduct these steps to fit them:

1. Set up the model by specifying the model you wish to fit, the parameters you want to tune, the engine the model comes from, and the mode (regression or classification) if necessary.

   Code

2. Set up the workflow for the model and add the model and the recipe.

   Code

3. Create a tuning grid to specify the ranges of the parameters you wish to tune as well as how many levels of each.

   Code

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

   Code

5. Save the tuned models to an RDS file to avoid rerunning the model.

Code

6. Load back in the saved files.

Code

7. Collect the metrics of the tuned models, arrange in ascending order of mean to see what the lowest RMSE for that tuned model is, and slice to choose only the lowest RMSE. Save the RMSE to a variable for comparison.

Code

Model Results

It's finally time to compare the results of all of our models and see which ones performed the best!

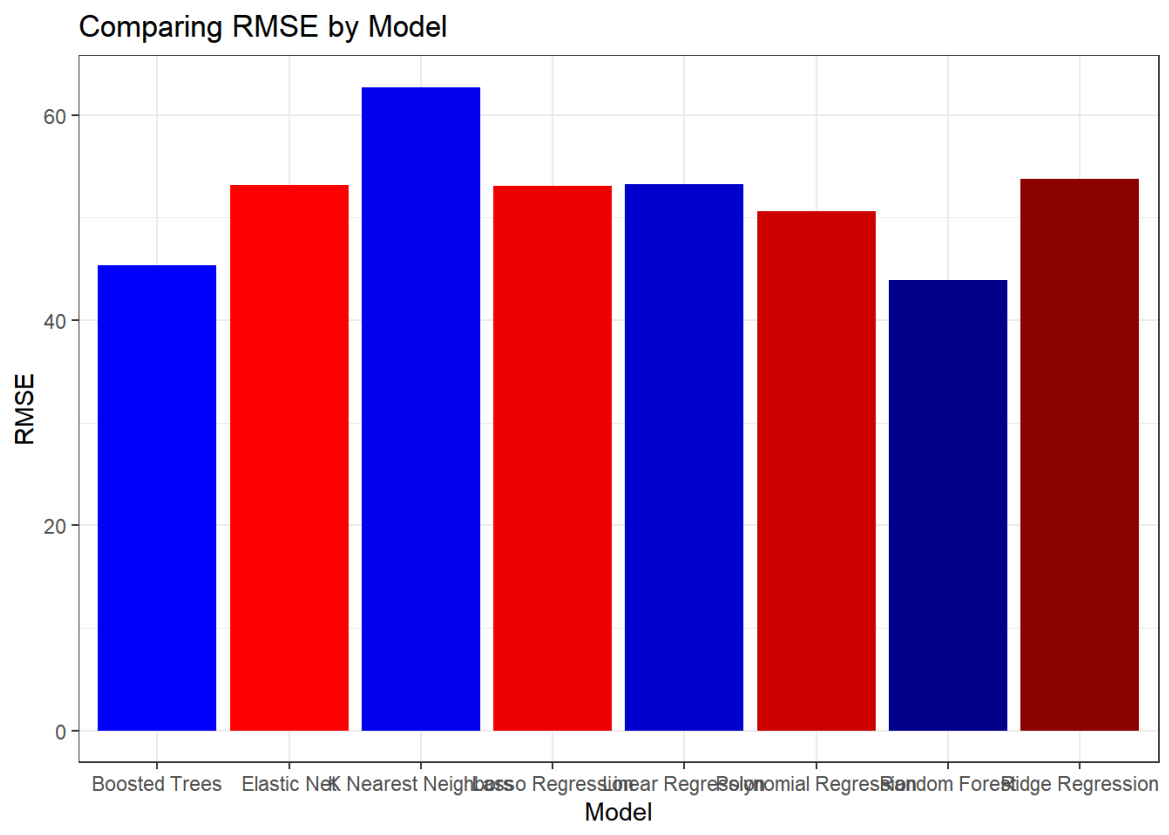
Code

Model <chr>	**RMSE** <dbl>
Random Forest	43.89281
Boosted Trees	45.36916
Polynomial Regression	50.60433
Lasso Regression	53.14522
Elastic Net	53.21883
Linear Regression	53.30941
Ridge Regression	53.74845
K Nearest Neighbors	62.74350

8 rows

Here is a visualization of these results:

Code



Note: the colors in this bar plot do not represent anything and are simply for aesthetic preference.

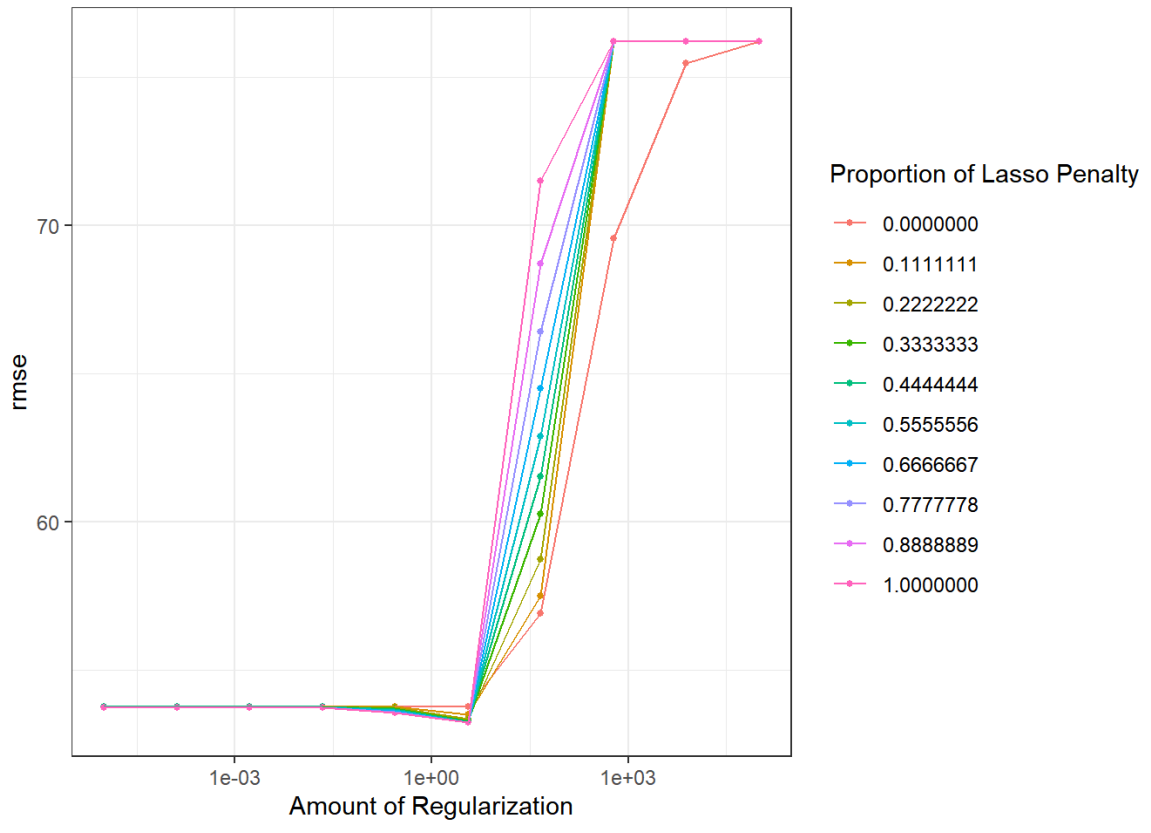
From the performance of the models on the cross-validation data, we can see that the random forest performed the best! The other three models we will focus on as well are boosted trees, polynomial regression, and elastic net because those placed right after the random forest. One key thing to note from these results is that it appears the linear and simpler models did the worst, which indicates that our data is probably nonlinear.

Model Autoplots

The autoplot in R allows us to visualize the effect of each of the tuned parameters on the performance of each model. In the autoplots, the performance of the models is measured by the RMSE (the lower the RMSE, the better the model has performed).

Elastic Net Plot

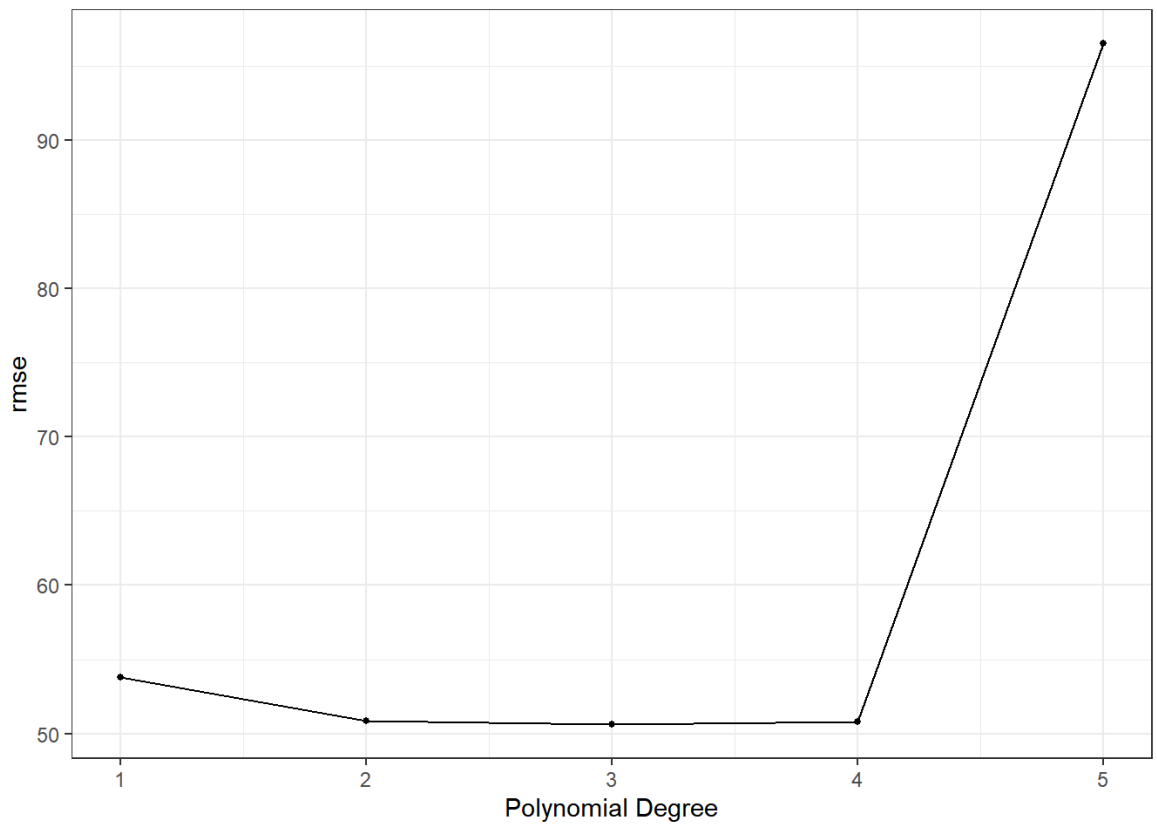
Code



For the elastic net, we tuned penalty and mixture at 10 different levels. From the plot, it seems that lower to mid-range penalty values appear to do the best. As the penalty term becomes too large, the model does worse because the coefficients of the predictors are being reduced to very small values (due to the penalty), which makes it much more difficult for the model to predict. Interestingly, although the RMSE appears to be lowest at a penalty value slightly greater than one (which matches the overall trend), the lowest RMSE appears to have a high mixture value. It seems that lower values of mixture only perform better than higher values of mixture when the penalty term is very high. Otherwise, when the penalty term is low to mid-range, higher values of mixture appear to perform better. This lines up with our overall model results as the lasso regression performed closer to the elastic net than the ridge regression (since lasso regression has a mixture value of 1 and ridge regression has a mixture value of 0).

Polynomial Regression Plot

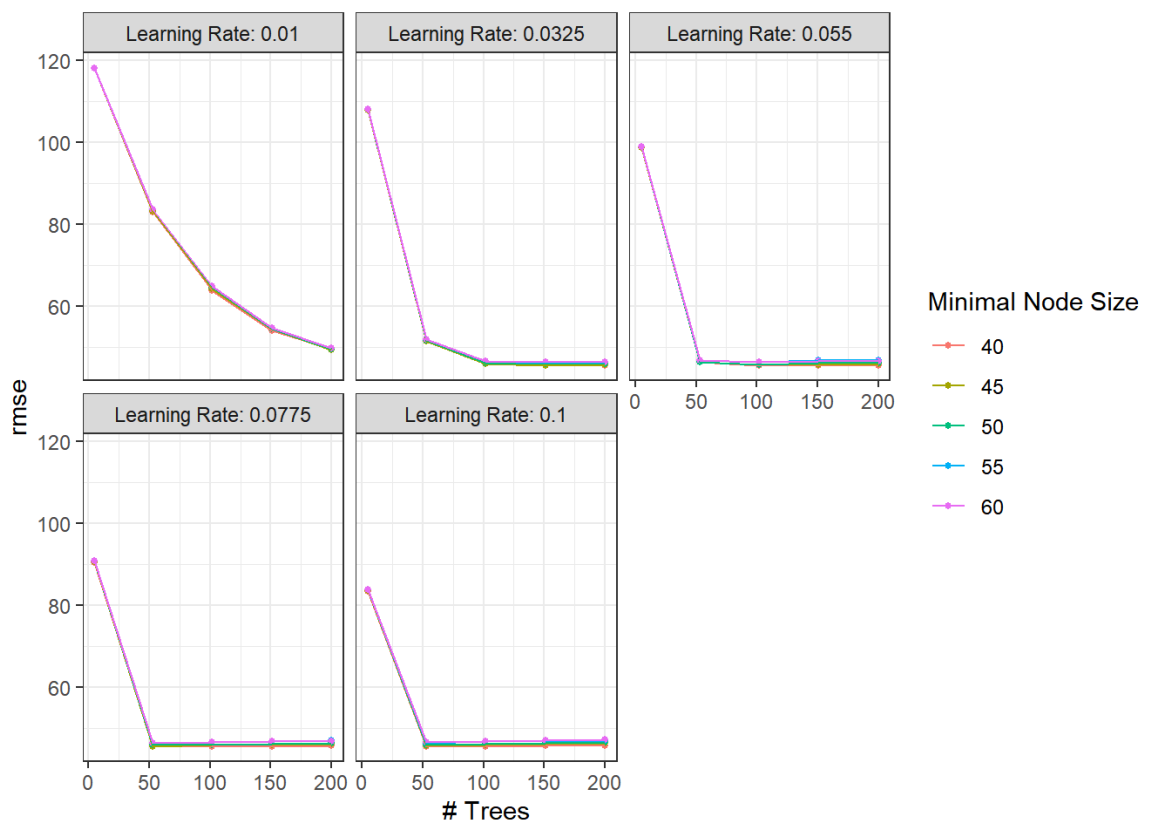
Code



For the polynomial regression, we tuned the degree of all continuous variables at 5 different levels. As we can see from the plot, the model performs better when a degree is added to the predictors (except when the degree gets too high). This further affirms our initial idea that the data is nonlinear. From the plot, it appears that the model does the best at a polynomial degree of 3 and does substantially worse after a polynomial degree of 4.

Boosted Trees Autoplot

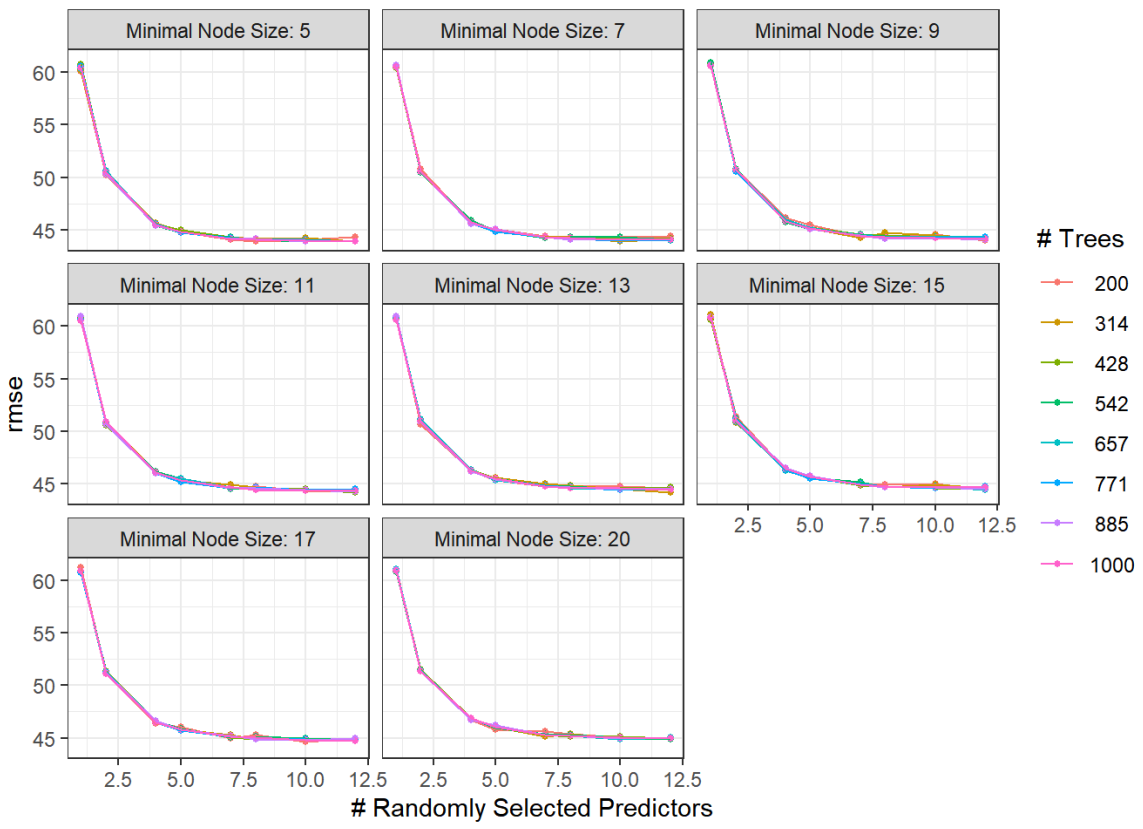
Code



For the boosted trees, we tuned the learning rate, number of trees, and minimal node size with 5 different levels. It may be useful to note that some tweaking of the model was done prior to finalizing the ranges for these parameters. The model did substantially worse when the minimal node size was a lower number, which is why the range 40 to 60 was chosen. Also, the model did not do as well when the learning rate was too high. A high learning rate causes the model to learn faster, but it also trains the model less and makes it less generalized. So, the range 0.01 to 0.1 was chosen to see which smaller learning rate value performs the best. From the starting points of each of the plots, it appears that the model does better at a higher learning rate. This means that the model does better when it is learning faster. Also, it appears that generally, once there are more than ~50 trees, the number of trees does not make much of a change on the performance of the model. However, the model appears to do worse when there are less than ~50 trees. The minimal node size does not seem to have much of an effect on the performance of the model.

Random Forest Autoplot

Code



For the random forest, we tuned the the minimal node size, the number of randomly selected predictors, and the number of trees. It may be important to note that the range of the number of randomly selected predictors for this model goes up to 12 rather than 14 (all predictors) to avoid creating a bagging model. The problem with a bagging model is that, because all the predictors are used, each tree may make the same first split. If all the trees have the same first split, every tree would no longer be independent from one another, which is an important assumption we make. And so, the range of the number of randomly selected predictors should be less than the total number of predictors, which is why I chose a slightly lower value at 12. From the plots, we can see that the number of trees does not seem to have much effect on the performance of the model. The minimal node size does not appear to have a huge effect on the performance, but the smaller values of `min_n` appear to have slightly lower RMSE values (at least judging by the starting point). The optimal node size appears to be at 5 (which lines up with the aforementioned observation). The number of predictors appears to have a greatest effect on performance. From the plots, it appears that, overall, a greater number of predictors renders a better performance. It seems that including 10-12 predictors gives the lowest RMSE here. Based on the RMSE values in the plots, this model definitely performs the best!

Results of the Best Model

Performance on the Folds

So, the random forest performed the best out of all 8 of our models. But which tuned parameters were chosen as the best random forest model?

Code

m...	trees	mi...	.metric	.estimator	mean	n	std_err	.config
<int>	<int>	<int>	<chr>	<chr>	<dbl>	<int>	<dbl>	<chr>
10	542	5	rmse	standard	43.89281	10	2.698086	Preprocess

1 row

Random forest #31 with 10 predictors, 542 trees, and a minimal node size of 5 performed the best with an RMSE of 43.89281!

Fitting to Training Data

Now, we will take that best model from the tuned random forest and fit it to the training data. This will train that random forest one more time on the entire training data set. Once we have fit and trained the random forest on the training data, it will be ready for testing!

Code

Testing the Model

Now, it's time to test our random forest model to see how it performs on data that it has not been trained on at all: the testing data set.

Code

Code

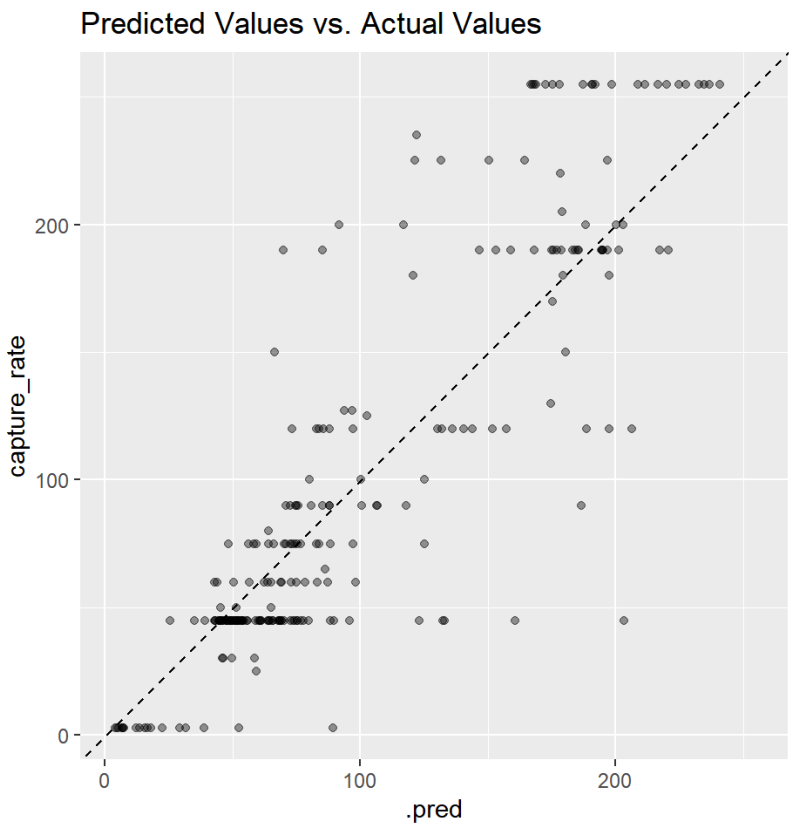
.metric	.estimator	.estimate
<chr>	<chr>	<dbl>
rmse	standard	37.78122

1 row

Our random forest actually performed better on the testing set than on the cross-validation folds with an RMSE of 37.78122! Since RMSE is measured in relation to the values the outcome may take on, our testing RMSE of 37.78122 in relation to the outcome range from 0 to 255 is not a great RMSE, but it is not too bad either. So, our model did not perform horribly and still managed to explain some variation in the outcome!

We might also be interested in a plot of the predicted values versus the actual values:

Code

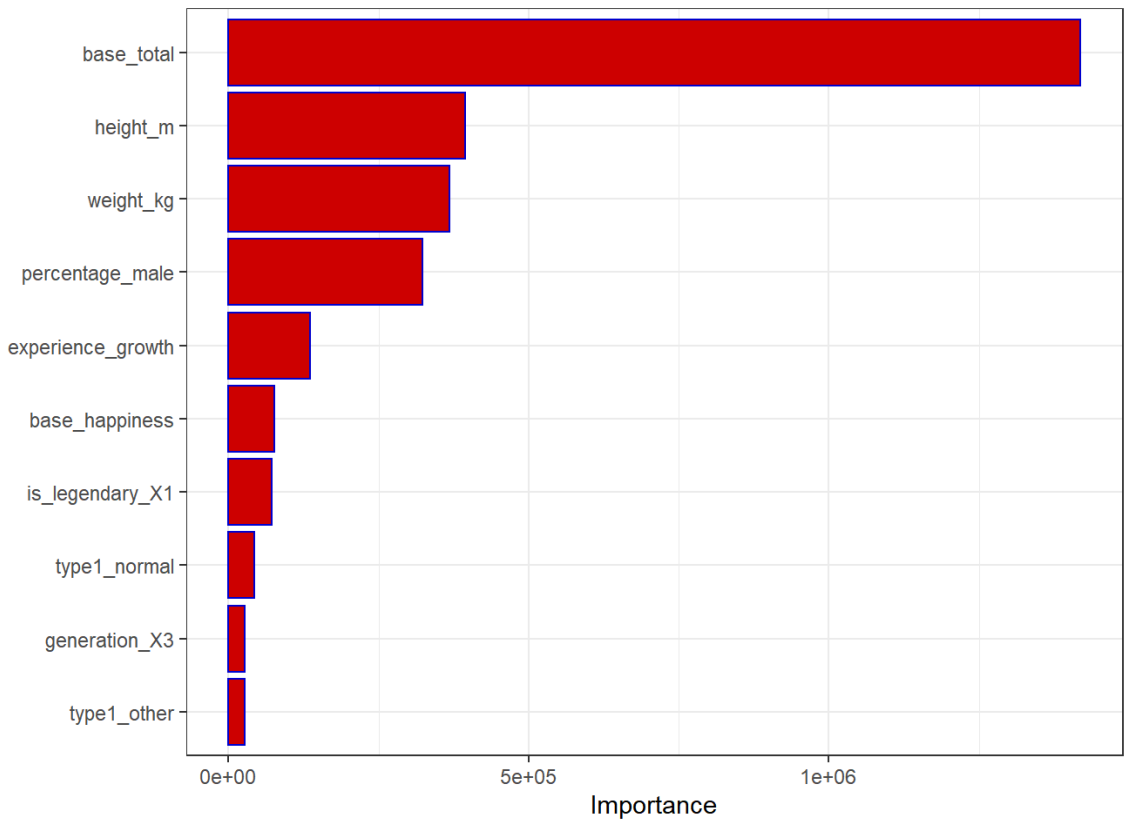


If every observation was predicted accurately, the dots would form a straight line. From the plot, we can see that very few of the dots actually fall on the line. However, it also appears that the model did not do a terrible job predicting the capture_rate because the points do seem to follow the same general pattern as the line. Also, it is a good sign that the model did not predict any negative capture rate values. So, overall, the random forest model did not do a terrible job predicting the capture rate, but there is definitely room for improvement.

Variable Importance

One cool thing about random forests is that they can also tell you which variables are the most important in predicting the outcome. This is shown in a variable importance plot (VIP).

Code



From this we can see that the variable that mattered the most in predicting the outcome was `base_total`, which is what we had believed would happen. Although some of the other variables present importance in predicting the outcome, the `base_total` is by far the most important variable. This also makes contextual sense because a Pokemon with higher battle statistics should be harder to capture because it has greater power than other Pokemon.

Conclusion

After fitting various models and conducting analysis on each of them, the best model to predict the capture rate of a Pokemon seems to be the random forest model. This is no surprise at all since the random forest tends to work well for most data because it is nonparametric and makes no assumptions about parametric forms or the outcome (more flexible). However, this model still does not do a great job predicting the capture rate (based on the RMSE).

The model that did the worst was K Nearest Neighbors (KNN). This was also not surprising as KNN tends to do worse when there are too many predictors because this means the data has too many dimensions. In a high dimensional data space, the data points are not close enough to each other for KNN to do well in predicting the outcome unless there are enough observations to make up for the high dimensional data space. Therefore, it makes sense that KNN did the worst.

Some avenues of improvement may be to look into other characteristics associated with capturing a Pokemon that may help explain more of the variability in the outcome than our current model. For example, in the Pokemon games, there are multiple Poke balls that can be used to capture a Pokemon (Great Ball, Ultra Ball, etc.). Perhaps the capture rate of some of the Pokemon in this data set was affected by which ball was used to capture it since some balls are more powerful than others. Another potential predictor could be the items used on a Pokemon before capturing it that might make it easier to capture (berries, etc.). This might have also altered the capture rate of some Pokemon.

One potential question that may be lingering is whether the choice to include the `base_total` rather than the individual six battle statistics was the correct one. You may be wondering whether the model would have done better if those were included instead. I went back and included those six battle statistics instead of the `base_total` and ran the random forest model. The model actually did slightly worse than it did with the total of the statistics. When running the variable importance plot for that fit, `height_m` actually came up as the most important variable. I believe this might be due

to the lower correlation each of the individual six statistics have with `capture_rate`. Perhaps any one of the individual six statistics alone does not contribute much to predicting the outcome. So, I believe it was justified to use `base_total` instead.

If I were to continue this project and move forward with my analysis, I would like to explore more predictors that could contribute to finding the capture rate of a Pokemon (as mentioned before). Also, another avenue for analysis might be the Pokemon GO games instead of the original Pokemon games. It would be interesting to see if the capture rate of a Pokemon differs based on the game it is in or if the ability to predict the capture rate differs by game type.

Overall, attempting to predict the capture rate of a Pokemon using this data set provided great opportunity for me to build my machine learning and data analysis skills. As I got to know the data set more, I found myself becoming more passionate about learning everything about my data and finding a model that would work the best. Although the random forest might not have been perfect, I am glad I was able to develop a model that at least explains some of the variation in the capture rate of a Pokemon!

Sources

This data was taken from the Kaggle data set, “Pokemon Stats Dataset (https://www.kaggle.com/datasets/hasanarcas/pokemon-stats-dataset),” and it was scraped from the official Pokemon Database (https://pokemondb.net/pokedex/all) by user Hasan Arcas.

The information about Minior was found on Bulbapedia (https://bulbapedia.bulbagarden.net/wiki/Minior_(Pok%C3%A9mon)), a community Pokemon encyclopedia.

General facts and definitions mentioned in the project were either found or double checked (if from prior knowledge) on the official Pokemon Database (https://pokemondb.net/). This is also the same database where the data was scraped from.