Capstone Project: Abalone Age Prediction

Roxie Trachtenberg

2023-12-05

Introduction

The abalone dataset, downloaded from the `AppliedPredictiveModeling` R package (which can be accessed at this link (https://github.com/cran/AppliedPredictiveModeling/tree/master/data)), contains 4,177 instances of physical and qualitative observations of this type of marine snail. The data originated from a 1995 study conducted by UC Irvine and contains the following data:

Variable Name	Variable Type	Units	Description
Sex	Categorical	NA	Split into 3 categories: M (Male), F (Female), and I (Immature)
LongestShell	Continuous	millimeters	Abalone length measured from ends of longest shell
Diameter	Continuous	millimeters	Abalone shell diameter (perpendicular to longest shell length)
Height	Continuous	millimeters	Abalone height, including meat within shell
WholeWeight	Continuous	grams	Whole abalone weight
ShuckedWeight	Continuous	grams	Weight of abalone flesh (not including exoskeleton or shell)
VisceraWeight	Continuous	grams	Gut weight after bleeding
ShellWeight	Continuous	grams	Shell weight only (after being dried)
Rings	Integer	NA	Number of abalone rings observed, used to derive age in years

According to the researchers that piloted the 1995 study, the age of an abalone is determined by cutting and staining the shell, followed by analysis under microscope, which is a cumbersome task. All information regarding the origin of this data can be found in the UC Irvine Machine Learning Repository archives, here (http://archive.ics.uci.edu/dataset/1/abalone). The goal of this project is to see if the measurements already a part of this dataset can be used to predict the age instead of using the aforementioned time-consuming and invasive procedures.

Load Libraries

First, we need to install packages and load libraries of those that are needed to complete the project and run all of the project code.

```
# Install packages
package_names <- c("AppliedPredictiveModeling",
                   "ggplot2", "dplyr", "tidyverse", "caret",
                   "randomForest", "corrplot", "glmnet", "rmarkdown",
                   "RColorBrewer", "xgboost", "DAAG", "Metrics")
# Load libraries
libraries_to_load <- c("AppliedPredictiveModeling",
                       "ggplot2", "dplyr", "tidyverse", "caret",
                       "randomForest", "corrplot", "glmnet", "rmarkdown",
                       "RColorBrewer", "xgboost", "DAAG", "Metrics")
for (lib in libraries_to_load) {
  library(lib, character.only = TRUE)
}
```

```
# Load and examine abalone dataset
data(abalone)
dim(abalone)
```

```
## [1] 4177    9
```

```
str(abalone)
```

```
## 'data.frame':    4177 obs. of  9 variables:
##  $ Type         : Factor w/ 3 levels "F","I","M": 3 3 1 3 2 2 1 1 3 1 ...
##  $ LongestShell : num  0.455 0.35 0.53 0.44 0.33 0.425 0.53 0.545 0.475 0.55 ...
##  $ Diameter     : num  0.365 0.265 0.42 0.365 0.255 0.3 0.415 0.425 0.37 0.44 ...
##  $ Height       : num  0.095 0.09 0.135 0.125 0.08 0.095 0.15 0.125 0.125 0.15 ...
##  $ WholeWeight  : num  0.514 0.226 0.677 0.516 0.205 ...
##  $ ShuckedWeight: num  0.2245 0.0995 0.2565 0.2155 0.0895 ...
##  $ VisceraWeight: num  0.101 0.0485 0.1415 0.114 0.0395 ...
##  $ ShellWeight  : num  0.15 0.07 0.21 0.155 0.055 0.12 0.33 0.26 0.165 0.32 ...
##  $ Rings        : int  15 7 9 10 7 8 20 16 9 19 ...
```

Upon initial observation, it looks like the dataset has 4,177 observations and 9 columns, with variable information that generally matches what is in the documentation.

Data Cleaning

Now that we have our libraries and dataset loaded without error, we can inspect the structure and contents of the data and identify any parts that may require cleaning, formatting, and/or handling of outliers or missing data.

First, we created a new column for “Age” (which is just the number of rings + 1.5), removed the “Rings” column to avoid confusion and issues with multicollinearity (Age and Rings would have a perfect correlation of 1 if the “Rings” variable were left in), and renamed the “Type” variable to “Sex” for clarity.

```
abalone <- abalone %>% mutate(Age = Rings + 1.5) %>% rename(Sex = Type) %>% select(-Rings)
```

Next, we checked the minimum of each variable (ignoring the result of the categorical variable, “Sex”), to ensure that there were no “0” measurements.

```
apply(abalone, 2, min)
```

```
##           Sex  LongestShell      Diameter        Height   WholeWeight
##           "F"       "0.075"       "0.055"       "0.000"      "0.0020"
## ShuckedWeight VisceraWeight   ShellWeight           Age
##      "0.0010"      "0.0005"      "0.0015"        " 2.5"
```

It looks like the “Height” variable has at least one value of 0 that should be removed. It does not make sense that an abalone would not have a measurement for height and this could skew our results down the line when we apply models to the data.

In addition, we know from a logical perspective that the shucked and viscera weight cannot be larger than whole weight, so any observations that show this are likely erroneous.

```
sum(abalone$ShuckedWeight >= abalone$WholeWeight)
```

```
## [1] 4
```

```
# there are 4 values here that should be removed

sum(abalone$VisceraWeight >= abalone$WholeWeight)
```

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

```
# There are no values here that need to be removed
```

Now that we know which values need to be removed, let’s generate a clean dataset:

```
abalone_clean <- abalone %>% filter(ShuckedWeight <= WholeWeight) %>% filter(Height != 0)

dim(abalone_clean)
```

```
## [1] 4171    9
```

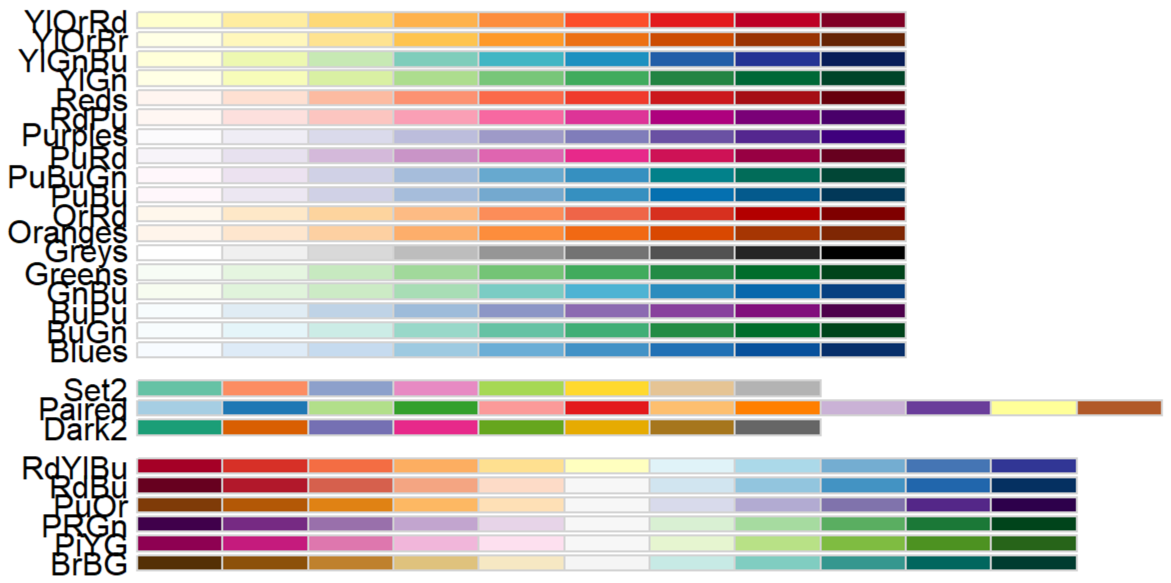
The clean dataset has 4,171 observations and 9 columns, meaning we successfully removed the 6 erroneous observations.

Exploratory Data Analysis (EDA)

Plot Formatting

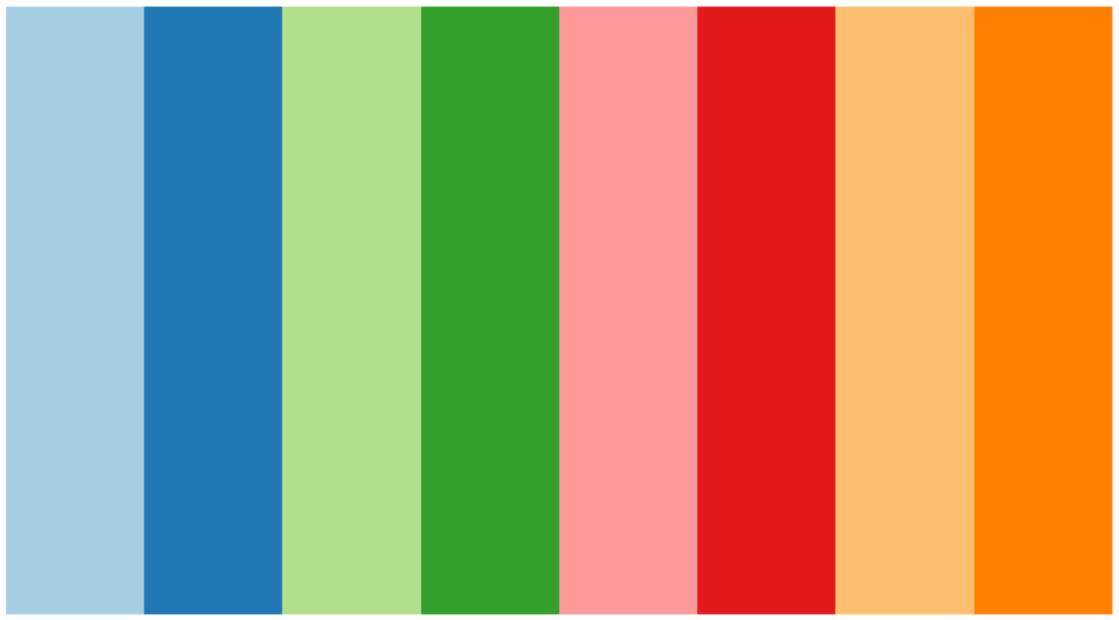
Now that we have a clean dataset, let's get a feel for the data and how it's distributed throughout and within features. Since we will be doing a decent amount of plotting and graphing, let's first look into a color-blind friendly palette to use throughout our analysis using the RColorBrewer package.

```
display.brewer.all(colorblindFriendly = TRUE)
```



We will be using the "Paired" and "Set2" palettes, so let's note their hexadecimal color names for future use using the following code:

```
display.brewer.pal(n = 8, name = 'Paired')
```



Paired (qualitative)

```
brewer.pal(n = 8, name = "Paired")
```

```
## [1] "#A6CEE3" "#1F78B4" "#B2DF8A" "#33A02C" "#FB9A99" "#E31A1C" "#FDBF6F"
## [8] "#FF7F00"
```

```
display.brewer.pal(n = 8, name = 'Set2')
```

Set2 (qualitative)

```
brewer.pal(n = 8, name = "Set2")
```

```
## [1] "#66C2A5" "#FC8D62" "#8DA0CB" "#E78AC3" "#A6D854" "#FFD92F" "#E5C494"
## [8] "#B3B3B3"
```

Finally, let’s define a shortcut for applying good general formatting to plots to save processing time later.

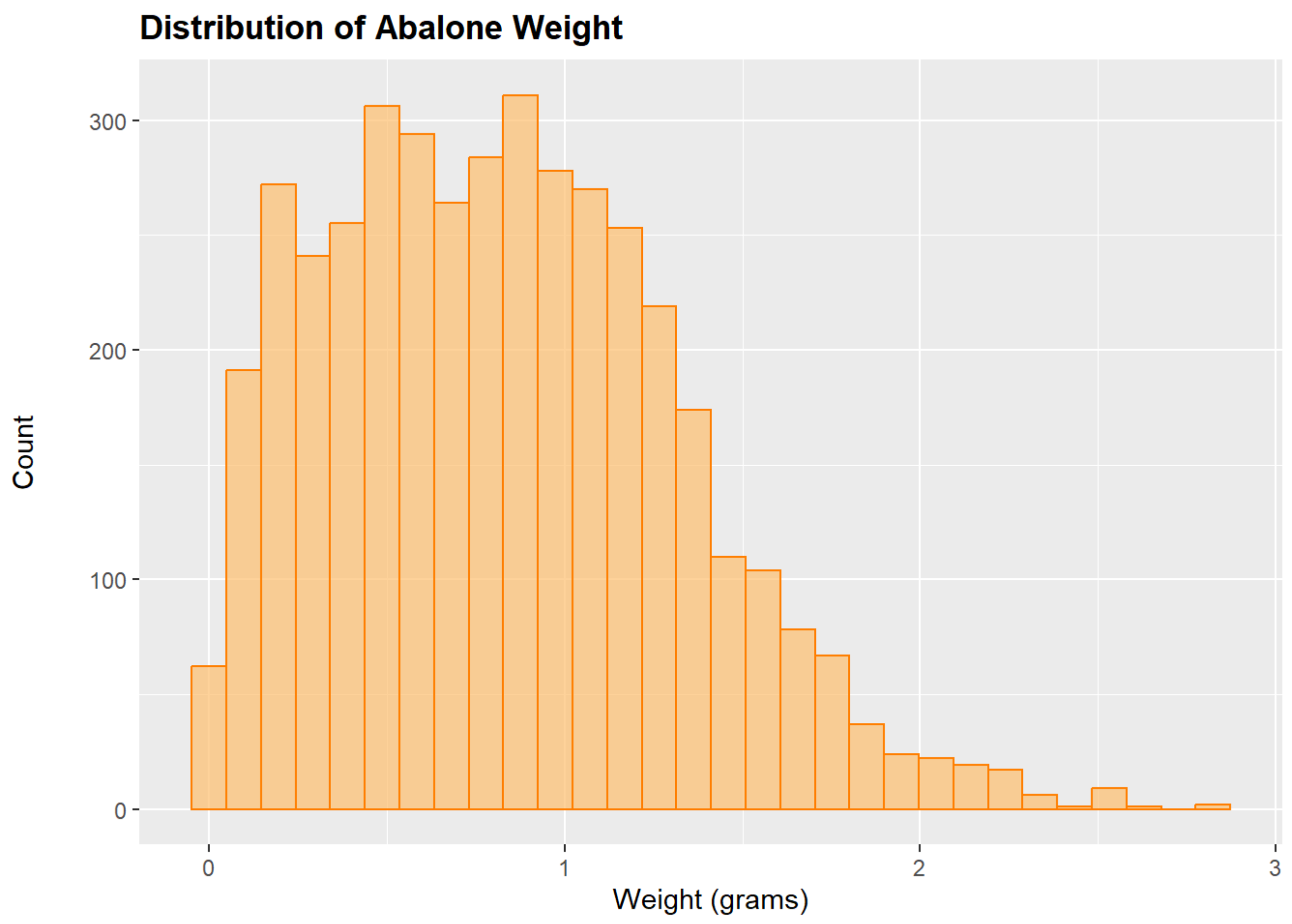
```
gen_formatting <- theme(plot.title = element_text(face = "bold"),
                        axis.title.y = element_text(margin = unit(c(0, 20, 0, 0), "pt"))
)
```

Univariate EDA

First, let’s visualize the distribution of abalone weights in the full dataset:

```
plot1 <- ggplot(abalone_clean, aes(x = WholeWeight)) +
  geom_histogram(fill = "#FDBF6F", color = "#FF7F00", alpha = 0.7) +
  labs(title = "Distribution of Abalone Weight",
       x = "Weight (grams)",
       y = "Count") +
  gen_formatting

plot1
```

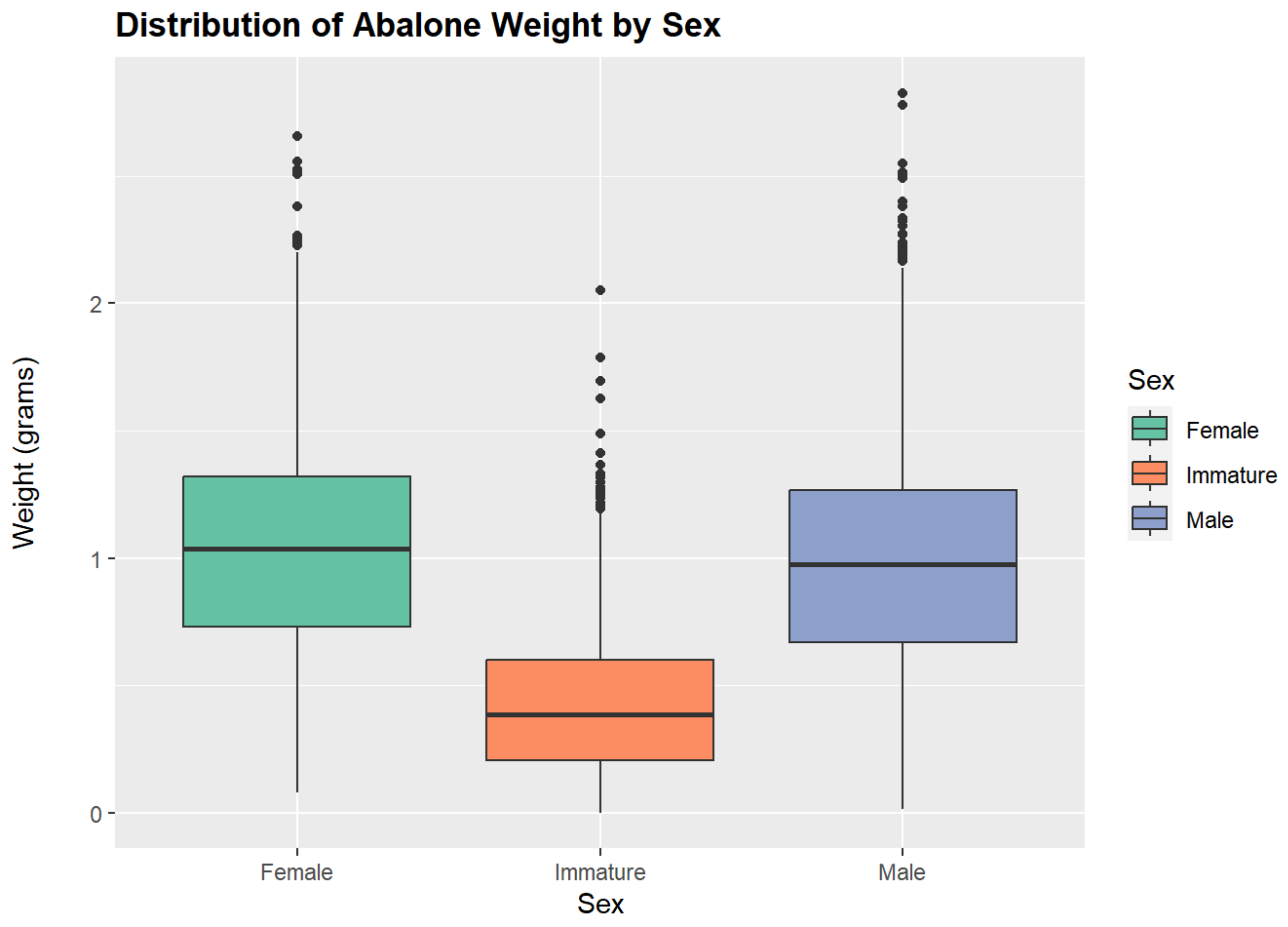
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



It looks like whole weight of abalones in our dataset fall generally below 3 grams, with a majority clustered between 0 and 1.5 grams. This looks like a fairly normal distribution of weights throughout the dataset.

Now, let’s explore whether there are significant weight differences by Sex:

```
# Visualize distribution of abalone weights by Sex
plot2 <- ggplot(abalone_clean, aes(x = Sex, y = WholeWeight, fill = Sex)) +
  geom_boxplot() +
  labs(title = "Distribution of Abalone Weight by Sex",
       x = "Sex",
       y = "Weight (grams)") +
  scale_fill_brewer(palette = "Set2", labels = c("Female", "Immature", "Male")) +
  scale_x_discrete(labels = c("Female", "Immature", "Male")) +
  gen_formatting

plot2
```



Looking at this, it's pretty clear that there is a significant weight difference from Immature to mature abalones, but the male/female difference is not immediately clear. Let's use a t-test to see if there is a significant difference in weight between these two groups:

```
# Conduct independent t-test: Are females significantly heavier than males?

# First, test that homogeneity of variance is achieved (prerequisite for t-test)
abalone_test <- abalone_clean %>% filter(Sex %in% c("M", "F"))
res <- var.test(WholeWeight ~ Sex, data = abalone_test)
res
```

```
##
##  F test to compare two variances
##
## data:  WholeWeight by Sex
## F = 0.83619, num df = 1306, denom df = 1527, p-value = 0.0008249
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  0.7533735 0.9285163
## sample estimates:
## ratio of variances
##          0.8361923
```

The ratio of variances is around 0.84, showing that variance is similar between male and female group and that we can move forward with our t-test.

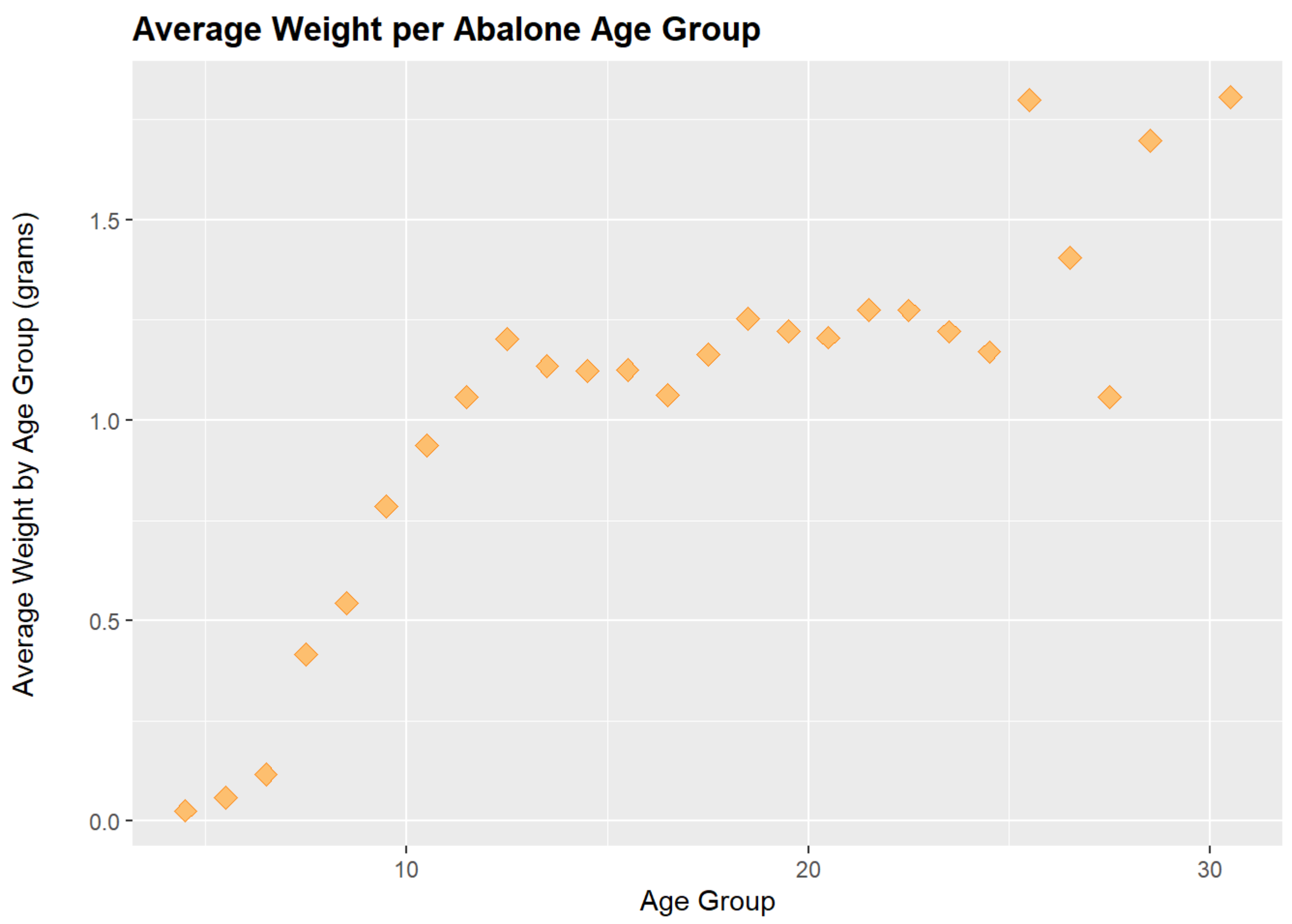
```
t.test(WholeWeight ~ Sex, var.equal = TRUE, data = abalone_test)
```

```
##
##  Two Sample t-test
##
## data:  WholeWeight by Sex
## t = 3.2305, df = 2833, p-value = 0.00125
## alternative hypothesis: true difference in means between group F and group M is not equal to
0
## 95 percent confidence interval:
##  0.02164586 0.08849956
## sample estimates:
## mean in group F mean in group M
##       1.0465321       0.9914594
```

The t-value is 3.2305. A higher absolute t-value indicates a larger difference between the groups. This represents the number of standard deviations that the sample mean (mean in group F minus mean in group M) is from the null hypothesis mean (0, assuming no difference between the groups). In summary, the low p-value (0.00125) and the 95% confidence interval (not including 0) suggest that there is evidence to reject the null hypothesis and that there is a significant difference between sample means in abalone female weight versus male weight.

We know there are differences in weight by Sex, but what about Age? Let’s now focus on our target variable, Age, and visualize weight by age group:

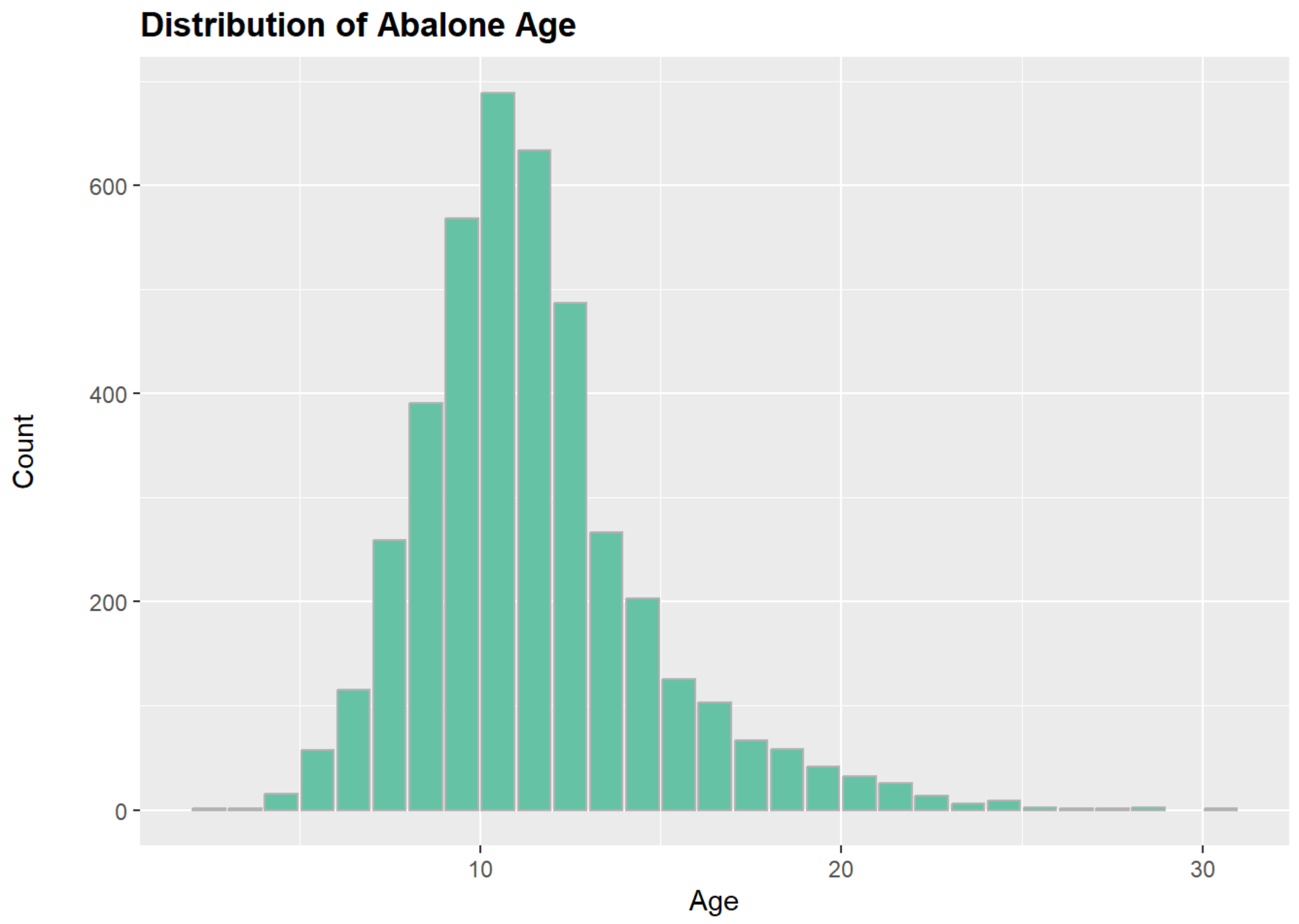
```
# Visualize average abalone weight by age group
plot3 <- abalone_clean %>% filter(Sex %in% c("M", "F")) %>% group_by(Age) %>%
  summarise(avg_weight = mean(WholeWeight)) %>%
  ggplot(aes(x = Age, y = avg_weight)) +
  geom_point(size = 3, shape = 23, color = "#FF7F00", fill = "#FDBF6F") +
  labs(title = "Average Weight per Abalone Age Group",
       x = "Age Group",
       y = "Average Weight by Age Group (grams)") +
  gen_formatting

plot3
```



This plot shows a general increasing trend and direct relationship between age group and average weight. However, there are some funky-looking values in the top right-hand corner of the plot (for age groups of around 18 yrs and above). Average is really only a good central tendency measure when the data in each age group is normally distributed and has a healthy sample size. It is possible that the distribution of weight data in the higher age groups is not normally distributed or has low sample size.

Let’s investigate distribution of age in the data:

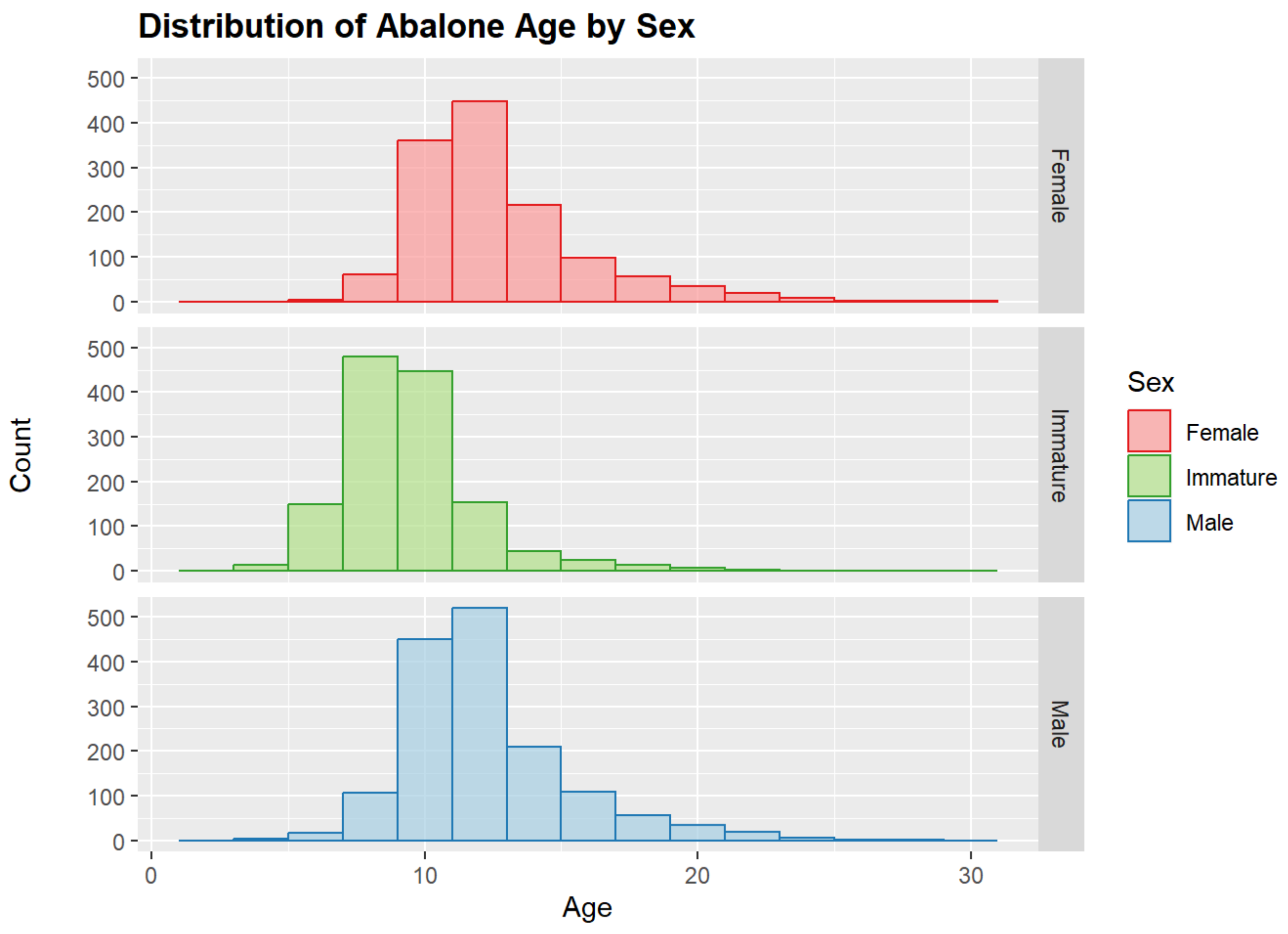
```
# Visualize age distribution within the abalone dataset:
plot4 <- abalone %>% select(-Sex) %>% group_by(Age) %>%  ggplot(aes(Age)) +
  geom_bar(fill = "#66C2A5", color = "#B3B3B3") +
  labs(title = "Distribution of Abalone Age",
       x = "Age",
       y = "Count") +
  gen_formatting

plot4
```



Interesting - it looks like our initial hunch about low sample size was correct and our dataset suffers from survivorship bias, in that there are fewer abalones that live to older age and therefore, there is less data to reliably predict abalones of older age. We will keep this unbalanced dataset issue in mind as we run machine learning models.

Is there at least a somewhat even distribution of males and females in each age group, even if sample size is lower in older age groups?

```r
# Visualize distribution of abalone age by Sex:
facet_colors_fill <- c("#A6CEE3", "#FB9A99", "#B2DF8A")
facet_colors_color <- c("#1F78B4", "#E31A1C", "#33A02C")

plot5 <- ggplot(abalone_clean, aes(x = Age, fill = as.factor(Sex), color = as.factor(Sex))) +
  geom_histogram(binwidth = 2, alpha = 0.7) +  # Custom bar color
  scale_fill_manual(values = setNames(facet_colors_fill, unique(abalone_clean$Sex)),
                    name = "Sex",
                    labels = c(F = "Female", I = "Immature", M = "Male")) +
  scale_color_manual(values = setNames(facet_colors_color, unique(abalone_clean$Sex)),
                     name = "Sex",
                     labels = c(F = "Female", I = "Immature", M = "Male")) +
  facet_grid(Sex ~ ., labeller = as_labeller(c(F = "Female", I = "Immature", M = "Male"))) +
  labs(title = "Distribution of Abalone Age by Sex",
       x = "Age",
       y = "Count") +
  gen_formatting

plot5
```

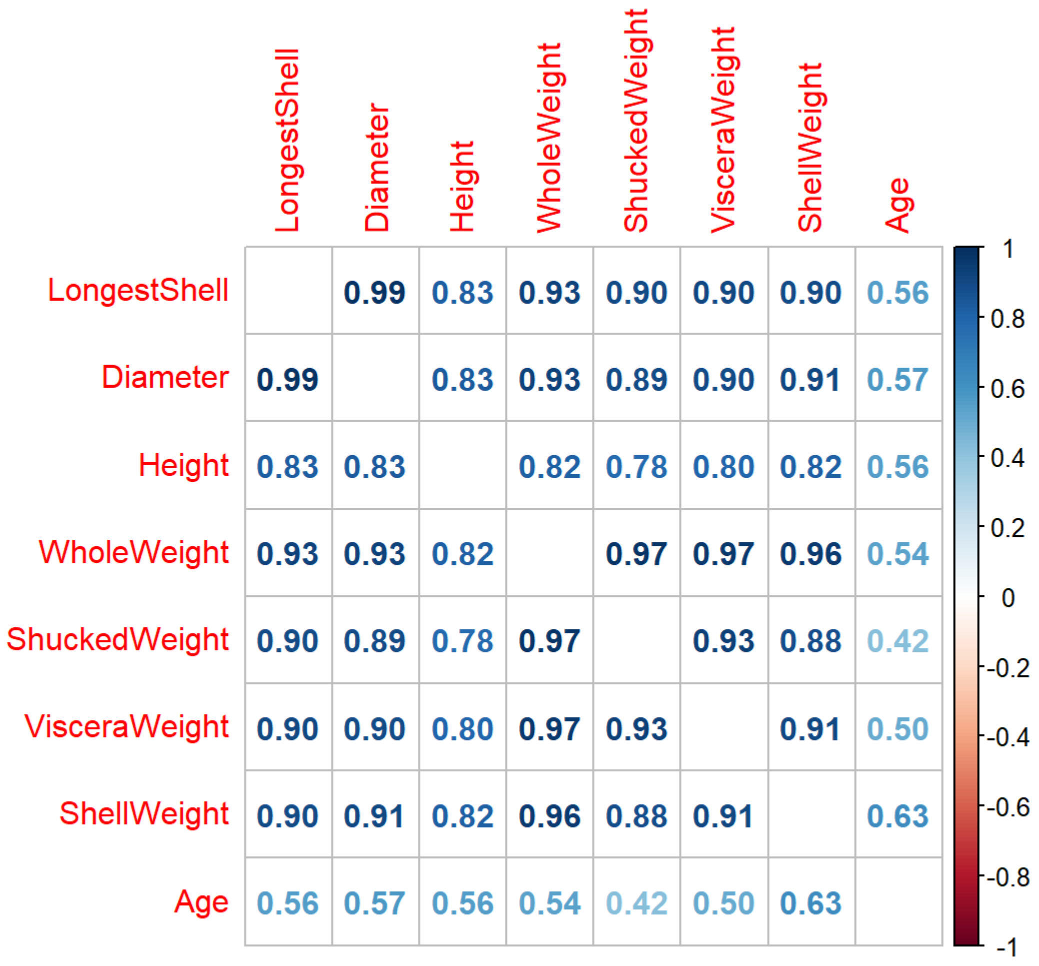


It looks like there is a normal distribution of grown males and females in each age group. Immature abalones show more of a right-skewed distribution.

Multivariate EDA

Now that we have made observations about some key variables in our dataset, such as, age, sex, and weight, let's explore their associations with one another through a correlation plot:

```
cor_table = abalone_clean %>% select(-Sex) %>% cor()
corplot1 <- corrplot(cor_table, method = 'number', diag = FALSE) # colorful number
```



	LongestShell	Diameter	Height	WholeWeight	ShuckedWeight	VisceraWeight	ShellWeight	Age
LongestShell		0.99	0.83	0.93	0.90	0.90	0.90	0.56
Diameter	0.99		0.83	0.93	0.89	0.90	0.91	0.57
Height	0.83	0.83		0.82	0.78	0.80	0.82	0.56
WholeWeight	0.93	0.93	0.82		0.97	0.97	0.96	0.54
ShuckedWeight	0.90	0.89	0.78	0.97		0.93	0.88	0.42
VisceraWeight	0.90	0.90	0.80	0.97	0.93		0.91	0.50
ShellWeight	0.90	0.91	0.82	0.96	0.88	0.91		0.63
Age	0.56	0.57	0.56	0.54	0.42	0.50	0.63	

This plot indicates that we will need to proceed cautiously regarding multicollinearity when running machine learning algorithms. However, features that are highly correlated with the age (e.g., ShellWeight, Diameter) might be good candidates for inclusion in our machine learning models over others. Viscera weight and whole weight, among other measurements like this, are highly correlated, so this may provide a good case for using interaction terms or dimensionality reduction prior to running machine learning models. Finally, it is important to note that higher correlations do not necessarily imply causation or better predictive power in a machine learning context.

Machine Learning Models

Linear Regression

Now that we have a good sense of our cleaned up data, let's start running some models!

First, let's define our RMSE function, set the seed (so that anyone that reviews this will get the same indices and therefore model results that I do when running my code below), and split our data into training and test sets:

```
# Define RMSE function
RMSE <- function(true_ratings, predicted_ratings) {
  sqrt(mean((true_ratings - predicted_ratings)^2))
}

set.seed(100) # Set the seed

train_index <- createDataPartition(abalone_clean$Age, p = 0.8, list = FALSE)
train_set <- abalone_clean[train_index, ]
test_set <- abalone_clean[-train_index, ]
```

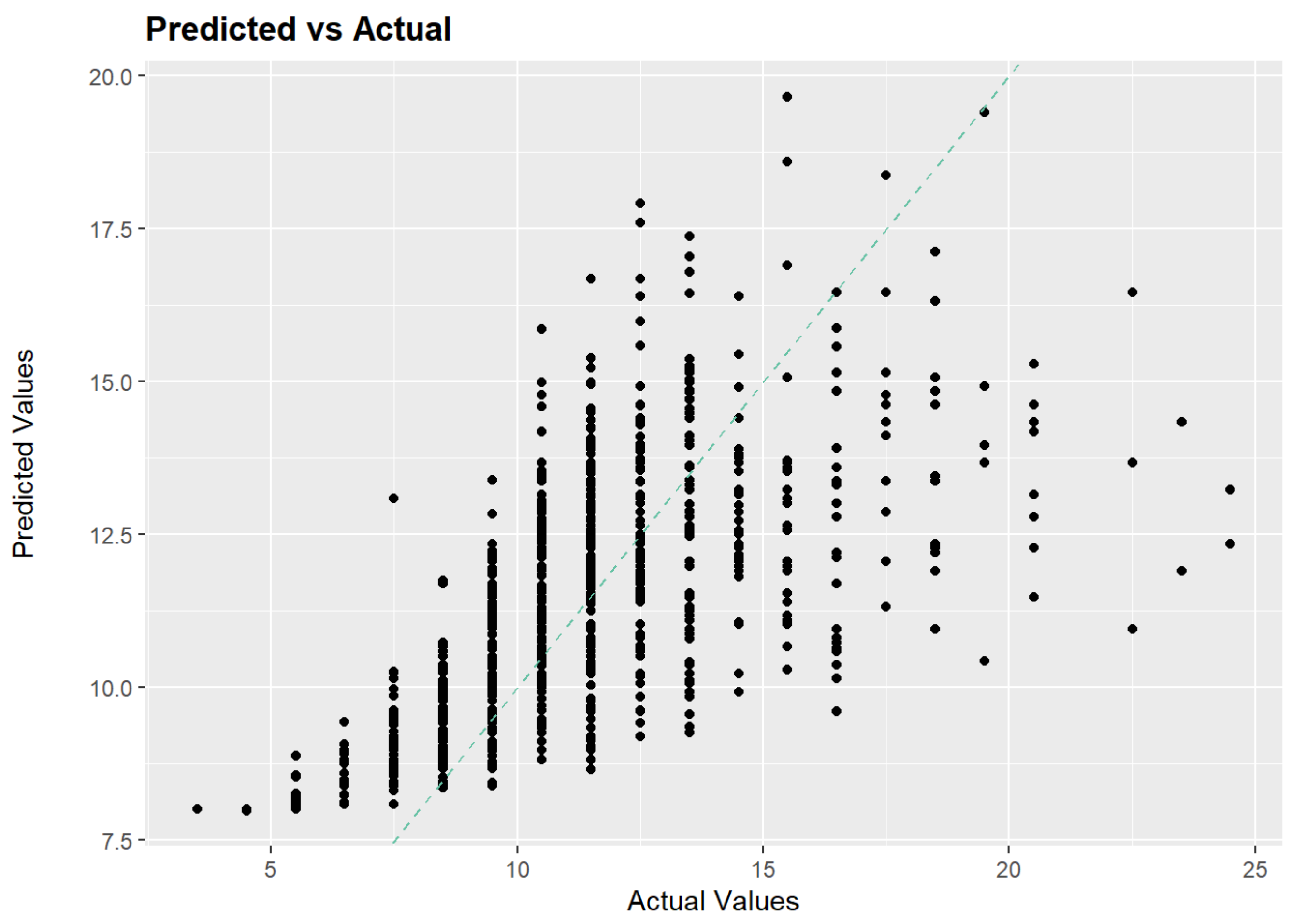
Let's run a simple linear regression model (to use primarily for comparison) with just ShellWeight (our variable most highly correlated with Age according to the correlation plot above) as a predictor:

```
# Prepare the model
lm_fit <- train_set %>% lm(Age ~ ShellWeight, data = .)

# Make predictions using the fitted model
pred <- predict(lm_fit, test_set)

# Generate root mean squared error (RMSE) value
rmse_1 <- RMSE(test_set$Age, pred)

# Save RMSE value in a table to track progress later
rmse_summary <- tibble(Model = "Linear Reg - Shell Weight", RMSE = round(rmse_1, 3))
rmse_summary
```

```
## # A tibble: 1 × 2
##   Model                      RMSE
##   <chr>                     <dbl>
## 1 Linear Reg - Shell Weight   2.4
```

Let's visualize our actual vs predicted values:

```
# Visualize predicted vs actual values
plot_data <- data.frame(Predicted_value = pred,
                        Observed_value = test_set$Age)

ggplot(plot_data, aes(x = Observed_value, y = Predicted_value)) +
  geom_point() +
  geom_abline(intercept = 0, slope = 1, color = "#66C2A5", linetype = "dashed") +
  labs(title = "Predicted vs Actual",
       x = "Actual Values",
       y = "Predicted Values") +
  gen_formatting
```



As we can see, the model has a harder job predicting the younger (likely immature abalones) and older age groups correctly. We will experiment with using weights to fix this issue of unbalanced data in the random forest section. For now, let's try linear regression using all of the variables (aside from Age) as predictors to see if we can improve our result.

```
lm_fit <- train_set %>% lm(Age ~ ., data = .)
pred <- predict(lm_fit, test_set)
rmse_2 <- RMSE(test_set$Age, pred)

rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Linear Reg - All Predictor Variables",
                                 RMSE = round(rmse_2, 3)))
rmse_summary
```

```
## # A tibble: 2 × 2
##   Model                                 RMSE
##   <chr>                                <dbl>
## 1 Linear Reg - Shell Weight             2.4
## 2 Linear Reg - All Predictor Variables  2.12
```

This improves our RMSE quite a bit! Now let's see if we can refine the linear regression model further by adding interaction terms. Since the dataset has minimal features and we are relatively familiar with how each variable is related, we can incorporate interaction terms into our model by multiplying two or more predictor variables that may be correlated to capture their combined effects in the model. Interaction terms allow the model to account for situations where the relationship between one predictor variable and the response is related to the value of another predictor variable (e.g., ShellWeight and WholeWeight).

```
train_set_intxns <- train_set %>% mutate(
  physical_measurements = Height * Diameter * WholeWeight,
  internal_features = ShuckedWeight * VisceraWeight,
  shell_features = ShellWeight * Diameter)

test_set_intxns <- test_set %>% mutate(
  physical_measurements = Height * Diameter * WholeWeight,
  internal_features = ShuckedWeight * VisceraWeight,
  shell_features = ShellWeight * Diameter)

lm_fit <- train_set_intxns %>% lm(Age ~ ., data = .)
pred <- predict(lm_fit, test_set_intxns)
rmse_3 <- RMSE(test_set_intxns$Age, pred)

rmse_summary <- bind_rows(rmse_summary, tibble(
  Model = "Linear Reg - All Predictor Variables + Interaction Terms",
  RMSE = round(rmse_3, 3)))
rmse_summary
```

```
## # A tibble: 3 × 2
##   Model                                                     RMSE
##   <chr>                                                    <dbl>
## 1 Linear Reg - Shell Weight                                 2.4 
## 2 Linear Reg - All Predictor Variables                      2.12
## 3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
```

Nice! We got an even better RMSE, but this does not solve our multicollinearity issue (actually, adding in interaction terms may have exacerbated it since most of the variables are pretty correlated and combining them doesn't help with that). Let's see if we can address this and any overfitting that may be occuring with regularization techniques.

Regularization Application

Data Preparation

First, let's transform Sex to a numeric value so that all predictor variables are numeric:

```
train_set_continuous <- train_set_intxns %>%
  mutate(Sex_Integer = as.integer(train_set$Sex)) %>% select(-c(Sex))

test_set_continuous <- test_set_intxns %>%
  mutate(Sex_Integer = as.integer(test_set$Sex)) %>% select(-c(Sex))
```

Since regularization methods are sensitive to the scale of the input features, let's (a) ensure our features are on a similar scale and (b) if not, use a feature scaling technique such as standardization.

* Note that though the `glmnet` package that we are going to use to run regularized models standardizes data automatically, we will do this anyway for practice and educational purposes.

```
# Initialize the scaled data frames
train_set_scaled <- train_set_continuous
test_set_scaled <- test_set_continuous

# Ensure features are on a similar scale:
scale_names_train <- colnames(train_set_continuous)[!colnames(train_set_continuous) %in% c("Ag
e")]
scale_names_test <- colnames(test_set_continuous)[!colnames(test_set_continuous) %in% c("Age")]

# Check the means and standard deviations of the features
feature_summary <- data.frame(
  Feature = scale_names_train,
  Mean = colMeans(train_set_continuous[, scale_names_train]),
  SD = apply(train_set_continuous[, scale_names_train], 2, sd)
)

feature_summary
```

```
##                                    Feature       Mean         SD
## LongestShell                  LongestShell 0.52521120 0.11993128
## Diameter                          Diameter 0.40895297 0.09905777
## Height                              Height 0.14001947 0.04235368
## WholeWeight                    WholeWeight 0.83412433 0.49016970
## ShuckedWeight                ShuckedWeight 0.36150989 0.22171133
## VisceraWeight                VisceraWeight 0.18188212 0.10961135
## ShellWeight                    ShellWeight 0.23957070 0.13892691
## physical_measurements physical_measurements 0.06342414 0.05840722
## internal_features        internal_features 0.08838225 0.09587406
## shell_features              shell_features 0.11041915 0.08036662
## Sex_Integer                    Sex_Integer 2.05452367 0.82551401
```

It does not look like our features are on a similar scale, indicated by the wide range of means and standard deviations for each feature, so we can scale using the code below:

```
# Implement feature scaling using the scale() function
train_set_scaled[, scale_names_train] <- scale(train_set_continuous[, scale_names_train])
test_set_scaled[, scale_names_test] <- scale(test_set_continuous[, scale_names_test])

# Now, check the means and standard deviations after scaling
scaled_feature_summary <- data.frame(
  Feature = scale_names_train,
  Mean = colMeans(train_set_scaled[, scale_names_train]),
  SD = apply(train_set_scaled[, scale_names_train], 2, sd)
)

scaled_feature_summary
```

```
##                                       Feature          Mean SD
## LongestShell                     LongestShell -4.459850e-16  1
## Diameter                             Diameter  1.995691e-16  1
## Height                                 Height  1.453705e-16  1
## WholeWeight                       WholeWeight  1.006698e-16  1
## ShuckedWeight                   ShuckedWeight  1.095698e-16  1
## VisceraWeight                   VisceraWeight  6.999802e-17  1
## ShellWeight                       ShellWeight -3.797839e-17  1
## physical_measurements physical_measurements  4.854212e-17  1
## internal_features         internal_features -3.827461e-17  1
## shell_features               shell_features -7.299766e-17  1
## Sex_Integer                       Sex_Integer -3.604566e-17  1
```

The scaling process has centered the data (resulting in a mean close to zero) and standardized its spread (resulting in a standard deviation of 1) for each feature. Scaling ensures that the regularization strength (the hyperparameter controlling the extent of regularization) applies uniformly across features, so now that this is done, we can proceed with regularization.

Regularization - Ridge (L2) Regression

Ridge (L2) regression works best when most variables in the model are useful, as it shrinks parameters but does not remove them. Note that the parameter that makes this a Ridge (L2) Regression is the alpha parameter set at 0. Let's see how our ridge regression model performs:

```
# Define predictor names so that Age is not included as a predictor during model fitting

predictor_names <- colnames(train_set_scaled)[!colnames(train_set_scaled) %in% c("Age")]

# Run regularized model and use to output predictions

alpha0.fit <- cv.glmnet(x= as.matrix(train_set_scaled[,predictor_names]), y= train_set_scaled$Ag
e, family= "gaussian",
          type.measure = "mse", alpha = 0, nlambda = 100)

alpha0.predicted <- predict(alpha0.fit, s = alpha0.fit$lambda.min, newx = as.matrix(test_set_sca
led[,predictor_names]))

rmse_4 <- RMSE(test_set_scaled$Age, alpha0.predicted)

rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Linear Reg w Ridge (L2) Regression",
                                 RMSE = round(rmse_4, 3)))
rmse_summary
```

```
## # A tibble: 4 × 2
##   Model                                                    RMSE
##   <chr>                                                   <dbl>
## 1 Linear Reg - Shell Weight                                2.4 
## 2 Linear Reg - All Predictor Variables                     2.12
## 3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
## 4 Linear Reg w Ridge (L2) Regression                       2.17
```

It looks like this didn’t have quite the effect on RMSE that we were hoping for, but maybe that’s because Lasso (L1) Regression is a better fit.

Regularization - Lasso (L1) Regression

Lasso (L1) Regression works best when there are some variables in the model that are useless or redundant (sounds like our multicollinearity issue!), as it shrinks these parameters completely to zero and therefore generates a simpler model. Note that the parameter that indicates Lasso (L1) Regression is the alpha parameter set at 1. Let’s see how our lasso regression model performs:

```
# Run regularized model and use to output predictions

alpha1.fit <- cv.glmnet(x= as.matrix(train_set_scaled[,predictor_names]),
                        y= train_set_scaled$Age, family= "gaussian",
                        type.measure = "mse", alpha = 1, nlambda = 100)

alpha1.predicted <- predict(alpha1.fit, s = alpha1.fit$lambda.min, newx = as.matrix(test_set_sca
led[ , predictor_names]))

rmse_5 <- RMSE(test_set_scaled$Age, alpha1.predicted)

rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Linear Reg w Lasso (L1) Regression",
                                 RMSE = round(rmse_5, 3)))
rmse_summary
```

```
## # A tibble: 5 × 2
##   Model                                                   RMSE
##   <chr>                                                  <dbl>
## 1 Linear Reg - Shell Weight                               2.4 
## 2 Linear Reg - All Predictor Variables                    2.12
## 3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
## 4 Linear Reg w Ridge (L2) Regression                      2.17
## 5 Linear Reg w Lasso (L1) Regression                      2.06
```

The regularized model here seems to be an improvement, although it still doesn’t quite beat out the RMSE from our linear regression model with interaction terms.

Finally, let’s try elastic net regression to see if this improves our RMSE.

Regularization - Elastic Net Regression

Elastic net regression combines the the two regularization penalties (alpha and lambda) and is also good at dealing with situations when there are correlations between parameters. Note that the parameter that indicates Lasso (L1) Regression is the alpha parameter set between 0 and 1. We will use an alpha of 0.5 to get an idea of whether this method will be useful.

```
# Run regularized model and use to output predictions

alpha0.5.fit <- cv.glmnet(x= as.matrix(train_set_scaled[ , predictor_names]), y= train_set_scale
d$Age, family= "gaussian",
                        type.measure = "mse", alpha = 0.5, nlambda = 100)

alpha0.5.predicted <- predict(alpha0.5.fit, s = alpha0.5.fit$lambda.min, newx = as.matrix(test_s
et_scaled[ , predictor_names]))

rmse_6 <- RMSE(test_set_scaled$Age, alpha0.5.predicted)

rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Elastic Net Regression",
                                 RMSE = round(rmse_6, 3)))
rmse_summary
```

```
## # A tibble: 6 × 2
##   Model                                                   RMSE
##   <chr>                                                  <dbl>
## 1 Linear Reg - Shell Weight                               2.4 
## 2 Linear Reg - All Predictor Variables                    2.12
## 3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
## 4 Linear Reg w Ridge (L2) Regression                      2.17
## 5 Linear Reg w Lasso (L1) Regression                      2.06
## 6 Elastic Net Regression                                  2.06
```

It seems like the optimal combination of alpha and lambda may have yet to be found.

Regularization - Optimize Penalty Combination

In order to understand which combination of penalties is truly best, we need to try many different values for alpha by optimizing the elastic net regression. We can do so and then run a model using the optimal alpha value using the following code:

```
fit_values <- list() # initialize empty list of fit_values

# for loop to generate models using different values of alpha
for (i in 0:10) {
  fit_name <- paste0("alpha", i/10)

  fit_values[[fit_name]] <-
    cv.glmnet(x = as.matrix(train_set_continuous[ , predictor_names]), y= train_set_continuous$A
ge, family= "gaussian",
              type.measure = "mse", alpha = i/10, nlambda = 100)
}

results <- data.frame() # initialize empty dataframe for results

# for loop to run models and showcase results
for (i in 0:10) {
  fit_name <- paste0("alpha", i/10)

  predicted <-
    predict(fit_values[[fit_name]],
            s = fit_values[[fit_name]]$lambda.min, newx = as.matrix(test_set_continuous[ , predi
ctor_names]))

  rmse <- RMSE(test_set_continuous$Age, predicted)

  temp <- data.frame(alpha = i/10, mse = rmse, fit_name = fit_name)
  results <- rbind(results, temp)
}

results
```

```
##    alpha      mse fit_name
## 1    0.0 2.171615   alpha0
## 2    0.1 2.061348 alpha0.1
## 3    0.2 2.061860 alpha0.2
## 4    0.3 2.062202 alpha0.3
## 5    0.4 2.062729 alpha0.4
## 6    0.5 2.062729 alpha0.5
## 7    0.6 2.062844 alpha0.6
## 8    0.7 2.062910 alpha0.7
## 9    0.8 2.062947 alpha0.8
## 10   0.9 2.062865 alpha0.9
## 11   1.0 2.062894   alpha1
```

```
# rename colnames of results dataframe for clarity
results <- results %>% rename(rmse = mse, alphas = alpha)

# Print the best result:
best_alpha <- results$alphas[which.min(results$rmse)] # store the best alpha in a variable

# Run elastic net regression model with the best alpha

alpha.best.fit <- cv.glmnet(x= as.matrix(train_set_scaled[ , predictor_names]), y= train_set_sca
led$Age, family= "gaussian", type.measure = "mse", alpha = best_alpha, nlambda = 100)

alpha.best.predicted <- predict(alpha.best.fit, s = alpha.best.fit$lambda.min, newx = as.matrix
(test_set_scaled[ , predictor_names]))

rmse_7 <- RMSE(test_set_scaled$Age, alpha.best.predicted)

rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Elastic Net Regression - Best Alpha",
                                 RMSE = round(rmse_7, 3)))
rmse_summary
```

```
## # A tibble: 7 × 2
##   Model                                                  RMSE
##   <chr>                                                 <dbl>
## 1 Linear Reg - Shell Weight                              2.4 
## 2 Linear Reg - All Predictor Variables                   2.12
## 3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
## 4 Linear Reg w Ridge (L2) Regression                     2.17
## 5 Linear Reg w Lasso (L1) Regression                     2.06
## 6 Elastic Net Regression                                 2.06
## 7 Elastic Net Regression - Best Alpha                    2.06
```

Even though the RMSE is similar to the RMSE we got when running the initial elastic net regression with alpha = 0.5, we will still add it to the table to show that we tested the optimized model.

Overall, the elastic net regression seemed to improve our RMSE notably, although still not as much as the linear regression with interaction terms. Let’s see if an ensemble method, which tends to be generally more advanced and robust to overfitting, can help improve our RMSE.

Random Forest Model

According to IBM (https://www.ibm.com/topics/random-forest), a random forest model is a machine learning algorithm that generates a combination, or “forest,” of decision trees, which act as a means to split the data to arrive at a final predicted result. While decision trees consider all the possible feature splits, random forests only select a subset of those features.

Let’s start by running a random forest model on our data with default parameters:

```
rf_model <- randomForest(Age ~ ., data = train_set_intxns, ntree = 500)
predictions <- predict(rf_model, test_set_intxns)
rmse_8 <- RMSE(predictions, test_set_intxns$Age)

rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Random Forest - Default Params",
                                 RMSE = round(rmse_8, 3)))
rmse_summary
```

```
## # A tibble: 8 × 2
##   Model                                                     RMSE
##   <chr>                                                    <dbl>
## 1 Linear Reg - Shell Weight                                 2.4
## 2 Linear Reg - All Predictor Variables                      2.12
## 3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
## 4 Linear Reg w Ridge (L2) Regression                        2.17
## 5 Linear Reg w Lasso (L1) Regression                        2.06
## 6 Elastic Net Regression                                    2.06
## 7 Elastic Net Regression - Best Alpha                       2.06
## 8 Random Forest - Default Params                            2.04
```

This is our best result yet! Let’s try running the same random forest model, but with incorporation of class weights (by assigning a higher weight to older age groups with a lower sample size) to address the issue of unbalanced data:

```
# Assign class weights (higher weight for age groups 15 years and older)
class_weights <- ifelse(train_set_intxns$Age >= 15, yes = 2, no = 1)

# Run the model with class weights
rf_model_weights <- randomForest(
  formula = Age ~ .,
  data = train_set_intxns,
  ntree = 500,
  classwt = class_weights
  )

predictions <- predict(rf_model_weights, test_set_intxns)
rmse_rf_wt <- RMSE(predictions, test_set_intxns$Age)
rmse_rf_wt
```

```
## [1] 2.035915
```

It doesn’t seem like this improved our RMSE from the initial random forest model, so we will leave it out of the table to avoid cluttering it up.

Note that though it is not shown in the code above, multiple values (from 2 through 10) were tried in place of the `yes = 2` argument in the code above, with no improvement in RMSE.

Let’s try tuning the model instead:

```r
# Specify the training control settings for cross-validation (with a 10-fold cross validation)
ctrl <- trainControl(method = "cv", number = 10)

# We could use the tuneGrid argument of the train() function in the caret package to do this,
# but let's try adjusting parameters using the randomForest() function instead

param_grid <- expand.grid(
  mtry = c(2, 4, 6),
  nodesize = c(1,3,5),
  ntree = c(100, 300, 500)
)

# Create an empty list to store the models
models <- list()
```

Let’s continue with our model tuning process. Note that the following code may take around 5 minutes to run, depending on the capabilities of your computer.

```r
# Iterate over each combination of parameters
# WARNING: Note that this code will take around 5 minutes to run.
for (i in 1:nrow(param_grid)) {
  params <- param_grid[i, ]

  # Train the model with the current set of parameters
  model <- randomForest(
    x = train_set_intxns[, colnames(train_set_intxns) != "Age"],
    y = train_set_intxns$Age,
    mtry = params$mtry,
    nodesize = params$nodesize,
    ntree = params$ntree
  )

  # Store the trained model
  models[[paste(params$mtry, params$nodesize, params$ntree, sep = "_")]] <- model
}

names(models)
```

```
##  [1] "2_1_100" "4_1_100" "6_1_100" "2_3_100" "4_3_100" "6_3_100" "2_5_100"
##  [8] "4_5_100" "6_5_100" "2_1_300" "4_1_300" "6_1_300" "2_3_300" "4_3_300"
## [15] "6_3_300" "2_5_300" "4_5_300" "6_5_300" "2_1_500" "4_1_500" "6_1_500"
## [22] "2_3_500" "4_3_500" "6_3_500" "2_5_500" "4_5_500" "6_5_500"
```

```
# Create an empty vector to store the RMSE values
rmse_values <- numeric(length(models))

# Evaluate each model on the test set
for (i in seq_along(models)) {
  model <- models[[i]]

  # Make predictions on the test set
  predictions <- predict(model, newdata = test_set_intxns[, colnames(test_set_intxns) != "Age"])

  # Calculate RMSE
  rmse_values[i] <- RMSE(predictions, test_set_intxns$Age)
}
```

Now, let’s find the best values and run our model again:

```
# Find the index of the model with the lowest RMSE
best_model_index <- which.min(rmse_values)

# Access the best model and parameter combination
best_model <- models[[best_model_index]]
names(models[best_model_index])
```

```
## [1] "2_5_100"
```

```
# Print the RMSE values
rmse_9 <- rmse_values[best_model_index]
rmse_9
```

```
## [1] 2.026884
```

```
rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Random Forest - Tuned Params",
                                 RMSE = rmse_9))
rmse_summary
```

```
## # A tibble: 9 × 2
##   Model                                                    RMSE
##   <chr>                                                   <dbl>
## 1 Linear Reg - Shell Weight                                2.4
## 2 Linear Reg - All Predictor Variables                     2.12
## 3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
## 4 Linear Reg w Ridge (L2) Regression                       2.17
## 5 Linear Reg w Lasso (L1) Regression                       2.06
## 6 Elastic Net Regression                                   2.06
## 7 Elastic Net Regression - Best Alpha                      2.06
## 8 Random Forest - Default Params                           2.04
## 9 Random Forest - Tuned Params                             2.03
```

It looks like using tuned parameters provided a slight improvement to our model. Let’s try another ensemble method for comparison purposes.

Gradient Boosting

According to Amazon Web Services documentation (https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost-HowItWorks.html), gradient boosting trees attempt to predict a target variable by combining estimates of a set of simpler models. Although gradient boosted models can be prone to overfitting, they have been known to perform better than random forests in some cases due to the fact that they combine results along the way and iterate upon results throughout the selection process, as opposed to combining results at the end of the process as a random forest algorithm would.

Let’s attempt to use a gradient boosted model with the `xgboost` package.

First, we need to do a bit of pre-processing to get the data into a format that is expected by `xgboost` :

```
# Separate out x and y variables

x_train <- subset(train_set_continuous, select = -c(Age))
y_train <- train_set_continuous$Age
x_test <- subset(test_set_continuous, select = -c(Age))
y_test <- test_set_continuous$Age

# Convert data to the format expected by xgboost
dtrain <- xgb.DMatrix(data = as.matrix(x_train), label = y_train)
dtest <- xgb.DMatrix(data = as.matrix(x_test), label = y_test)
```

Now, let’s set some standard hyperparameters and run the boosted model:

```
# Set hyperparameters (we may need to tune these)
params <- list(
  objective = "reg:squarederror",  # Regression task
  eval_metric = "rmse",            # Evaluation metric (Root Mean Squared Error)
  max_depth = 2,                 # Maximum depth of a tree
  eta = 0.1,                # Learning rate
  nrounds = 400     # Number of boosting rounds
)

# Train the xgboost model
xgb_model <- xgboost(data = dtrain, params = params, nrounds = 100)
```

Now, let’s make some predictions and evaluate RMSE:

```
# Make predictions on the training set (you can use a separate test set for predictions)
predictions <- predict(xgb_model, as.matrix(x_test))
rmse_10 <- RMSE(predictions, y_test)

rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Gradient Boosted Model - Standard Params",
                                 RMSE = round(rmse_10, 3)))
rmse_summary
```

```
## # A tibble: 10 × 2
##    Model                                               RMSE
##    <chr>                                              <dbl>
##  1 Linear Reg - Shell Weight                           2.4 
##  2 Linear Reg - All Predictor Variables                2.12
##  3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
##  4 Linear Reg w Ridge (L2) Regression                  2.17
##  5 Linear Reg w Lasso (L1) Regression                  2.06
##  6 Elastic Net Regression                              2.06
##  7 Elastic Net Regression - Best Alpha                 2.06
##  8 Random Forest - Default Params                      2.04
##  9 Random Forest - Tuned Params                        2.03
## 10 Gradient Boosted Model - Standard Params            2.08
```

This isn't looking quite as good as the random forest model, but let's try tuning parameters anyway:

```
# Define hyperparameter grid
param_grid <- expand.grid(
  eta = c(0.01, 0.1, 0.3),
  max_depth = c(3, 6, 9),
  gamma = c(0, 0.1, 0.2),
  subsample = c(0.8, 1.0),
  colsample_bytree = c(0.8, 1.0),
  min_child_weight = c(1, 3, 5)
)

# Function to train and evaluate model for a given set of hyperparameters

train_and_evaluate <- function(params, train_data, test_data) {
  # Convert non-numeric columns to numeric using one-hot encoding
  train_data_matrix <- model.matrix(~ . - 1, data = train_data)
  test_data_matrix <- model.matrix(~ . - 1, data = test_data)

  # Create DMatrix for training and testing data
  dtrain <- xgb.DMatrix(data = train_data_matrix, label = train_data$Age)
  dtest <- xgb.DMatrix(data = test_data_matrix, label = test_data$Age, missing = NA, nthread =
1)

  # Train the model
  xgb_model <- xgboost(params = params, data = dtrain, nrounds = 100, verbose = 0)

  # Make predictions on the test set
  predictions <- predict(xgb_model, newdata = dtest)

  # Evaluate the model
  rmse <- sqrt(mean((predictions - test_data$Age)^2))

  return(data.frame(params, rmse))
}

# Apply the function to all combinations of hyperparameters
# WARNING: This portion of the code may take 1-5 mins to run
# depending on your machine capabilities
results <- apply(param_grid, 1, function(row) {
  params <- as.list(row)
  train_data <- train_set_continuous
  test_data <- test_set_continuous
  train_and_evaluate(params, train_data, test_data)
})


# Combine results into a data frame
results_df <- do.call(rbind, results)
```

Now, let’s use the best parameter selection and run another model:

```
# Find the best hyperparameters
best_params <- results_df[which.min(results_df$rmse), ]
print(best_params)
```

```
##     eta max_depth gamma subsample colsample_bytree min_child_weight       rmse
## 275 0.1         6     0       0.8                1                5 0.02739996
```

```
# Let's try running our model again with these "best" parameters
params <- list(
  objective = "reg:squarederror",  # Regression task
  eval_metric = "rmse",            # Evaluation metric (Root Mean Squared Error)
  max_depth = 3,                  # Maximum depth of a tree
  eta = 0.1,
  gamma = 0,
  subsample = 0.8,
  colsample_bytree = 1,
  min_child_weight = 1
)

xgb_model <- xgboost(data = dtrain, params = params, nrounds = 100)
```

Finally, let's use the optimized model to make predictions and evaluate performance:

```
predictions <- predict(xgb_model, as.matrix(x_test))
rmse_11 <- RMSE(predictions, y_test)
rmse_11
```

```
## [1] 2.070017
```

```
rmse_summary <- bind_rows(rmse_summary,
                          tibble(Model = "Gradient Boosted Model - Tuned Params",
                                 RMSE = rmse_11))
rmse_summary
```

```
## # A tibble: 11 × 2
##    Model                                                    RMSE
##    <chr>                                                   <dbl>
##  1 Linear Reg - Shell Weight                                2.4
##  2 Linear Reg - All Predictor Variables                     2.12
##  3 Linear Reg - All Predictor Variables + Interaction Terms  2.06
##  4 Linear Reg w Ridge (L2) Regression                       2.17
##  5 Linear Reg w Lasso (L1) Regression                       2.06
##  6 Elastic Net Regression                                   2.06
##  7 Elastic Net Regression - Best Alpha                      2.06
##  8 Random Forest - Default Params                           2.04
##  9 Random Forest - Tuned Params                             2.03
## 10 Gradient Boosted Model - Standard Params                 2.08
## 11 Gradient Boosted Model - Tuned Params                    2.07
```

The tuned boosted model did improve RMSE slightly over the initial boosted model; however, a gradient boosted model in general may have been too granular for our limited dataset since the more generalized random forest models performed slightly better overall.

Summary

Within the scope of this project, our tuned random forest model is the winner!

Let’s take a look at a sorted version of our RMSE Summary table:

```
rmse_summary_sorted <- rmse_summary %>%
  arrange(RMSE)

rmse_summary_sorted
```

```
## # A tibble: 11 × 2
##    Model                                                   RMSE
##    <chr>                                                  <dbl>
##  1 Random Forest - Tuned Params                            2.03
##  2 Random Forest - Default Params                          2.04
##  3 Elastic Net Regression - Best Alpha                     2.06
##  4 Linear Reg w Lasso (L1) Regression                      2.06
##  5 Elastic Net Regression                                  2.06
##  6 Linear Reg - All Predictor Variables + Interaction Terms  2.06
##  7 Gradient Boosted Model - Tuned Params                   2.07
##  8 Gradient Boosted Model - Standard Params                2.08
##  9 Linear Reg - All Predictor Variables                    2.12
## 10 Linear Reg w Ridge (L2) Regression                      2.17
## 11 Linear Reg - Shell Weight                               2.4
```

We can likely assume that the random forest model was the best fit for our data due to plenty of factors; however, overall, the abalone dataset is a relatively simple one with only nine features (including the target variable), and may have been generalized better with a random forest model or linear regression (with all variables and adjusted inputs) over a more complex boosted model. In addition, most of the data in the abalone dataset seemed to have linear relationships, giving random forest and linear regression the edge over a gradient boosted model. Obviously, the simple linear regression model was too simple (likely underfit the data). Since we had issues with multicollinearity in the dataset and therefore, some of the variables were redundant and not contributing much to the predictive power, elastic net’s ability to perform variable selection may have been advantageous over ridge regression’s approach to simply shrinking all variable coefficients.

Reflections and Opportunities for Improvement

I went about this problem assuming Age was a continuous variable. However, since the age range of abalones in the dataset is 2.5 - 30.5, incremented by 0.5 for each age group, this could have easily been a classification problem and machine learning models could have been tailored to a classified result. For example, using cluster methods such as kNN would have been a good application to assigning an age “category” to an abalone using the other eight features in the dataset as predictors. If I had used this approach, I would have been able to run confusion matrices, which may have also provided better metrics of how my model was performing than just using RMSE as an indicator.

I was tentative to remove more observations from the dataset after cleaning since it already only had around 4,000 observations and 9 features, but another opportunity for improvement could have been better identification and adjustment of outliers in the dataset, especially since linear relationships were observed throughout. Removing outliers could have helped to reduce issues associated with noise (such as overfitting) and improve predictive power of my models.

Overall, the abalone data sample size was likely the the biggest bottleneck to achieving better machine learning models and results. More than anything, I think my algorithms and any future algorithms run on this dataset would benefit highly from more observations to predict from. It would be interesting to look into whether there is a similar quality dataset (more recent than 1995) with greater observations and features of abalone traits.