**Data 624 Project 2 Group 5**

By Noori Selina, Gavriel Steinmetz-Silber, Yina Qiao, and Zach Rose

7/13/2024

# Table of Contents

* [Introduction](#_h0q3b41ad6vk)
* [Exploratory Data Analysis (EDA)](#_ro2s5zhcgehv)
  + [Data Preprocessing](#_ba6gf1xylxfm)
  + [Class Distribution](#_ys9tf6f0mp2i)
  + [pH Levels by Brand Code](#_mtrw0dytm4e3)
  + [Low Variance Variables](#_1wwtjihuu39u)
  + [Handle Implausible Values](#_iw4lq5qal1io)
  + [Balling](#_eosrjs43139u)
  + [Missing Values](#_hb02nkw7u5wi)
  + [Imputing Missing Values](#_lk23gfobsul0)
  + [Handling Outliers](#_ugeu8et9mbed)
  + [Interpretation of Box Plots](#_amm5ttdsws2n)
  + [Histograms](#_q9gskm7oen5k)
  + [Interpretation of Histograms](#_poupakej44pz)
  + [Transformations](#_nc1tfgp9cw66)
  + [Correlations](#_qeuapnxkhzu)
* [Models and Evaluation](#_gqlzmazg3dmt)
  + [Evaluating Models on the Evaluation Set](#_dzkb8kee42a2)
  + [Training and Evaluating a Random Forest Model](#_w06nkg1eqldl)
* [Predictions](#_s3ln86u5nwwb)
  + [Evaluate Predictions](#_rxusv2itro0v)
* [Conclusion](#_6hgvu0exytv6)

# Introduction

In the beverage manufacturing industry, maintaining the correct pH level is crucial as it directly impacts product quality, safety, and consumer satisfaction. The pH level, a measure of acidity or alkalinity, must conform to a critical range to ensure the beverage's taste, stability, and shelf life. As a Key Performance Indicator (KPI), pH is vital for quality control and regulatory compliance.

This project aims to predict the pH levels in beverage production using various machine learning models. By accurately forecasting pH, the company can take proactive measures to maintain optimal production standards, reduce waste, and ensure product consistency.

The dataset provided consists of 2,571 rows and 33 columns, capturing various production parameters. Additionally, a scoring set with 267 cases is provided, which excludes the target variable (pH). The goal is to develop a predictive model using the training data and apply it to the scoring set to generate pH predictions.

# Exploratory Data Analysis (EDA)

### Data Preprocessing

Data preprocessing involves cleaning and preparing the dataset for analysis. This step is critical to ensure the accuracy and reliability of the predictive models.

```{r}

library(tidyverse)

library(readxl)

```

```{r}

train = read\_csv("https://raw.githubusercontent.com/gsteinmetzsilber/DATA-624-Project-2/main/Data/StudentData%20-%20TO%20MODEL.xlsx%20-%20Subset.csv")

test = read\_csv("https://raw.githubusercontent.com/gsteinmetzsilber/DATA-624-Project-2/main/Data/StudentEvaluation-%20TO%20PREDICT.xlsx%20-%20Subset%20(2).csv")

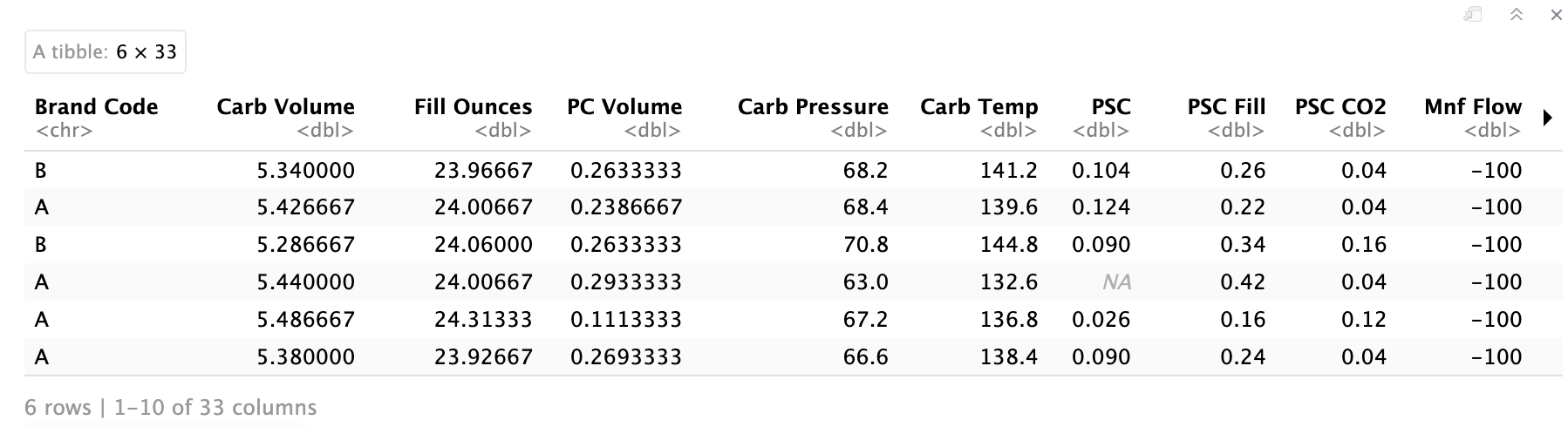
```

Now, let's get a sense of the data by examining the first few rows and generating summary statistics.

```{r}

head(train)

```



This reveals that the first column is categorical, while the others are numeric. Next we will run the summary statistics to get an overview of the data set.

```{r}

summary(train)

table(train$`Brand Code`)

```

Brand Code Carb Volume Fill Ounces PC Volume Carb Pressure Carb Temp PSC

Length:2571 Min. :5.040 Min. :23.63 Min. :0.07933 Min. :57.00 Min. :128.6 Min. :0.00200

Class :character 1st Qu.:5.293 1st Qu.:23.92 1st Qu.:0.23917 1st Qu.:65.60 1st Qu.:138.4 1st Qu.:0.04800

Mode :character Median :5.347 Median :23.97 Median :0.27133 Median :68.20 Median :140.8 Median :0.07600

Mean :5.370 Mean :23.97 Mean :0.27712 Mean :68.19 Mean :141.1 Mean :0.08457

3rd Qu.:5.453 3rd Qu.:24.03 3rd Qu.:0.31200 3rd Qu.:70.60 3rd Qu.:143.8 3rd Qu.:0.11200

Max. :5.700 Max. :24.32 Max. :0.47800 Max. :79.40 Max. :154.0 Max. :0.27000

NA's :10 NA's :38 NA's :39 NA's :27 NA's :26 NA's :33

PSC Fill PSC CO2 Mnf Flow Carb Pressure1 Fill Pressure Hyd Pressure1 Hyd Pressure2

Min. :0.0000 Min. :0.00000 Min. :-100.20 Min. :105.6 Min. :34.60 Min. :-0.80 Min. : 0.00

1st Qu.:0.1000 1st Qu.:0.02000 1st Qu.:-100.00 1st Qu.:119.0 1st Qu.:46.00 1st Qu.: 0.00 1st Qu.: 0.00

Median :0.1800 Median :0.04000 Median : 65.20 Median :123.2 Median :46.40 Median :11.40 Median :28.60

Mean :0.1954 Mean :0.05641 Mean : 24.57 Mean :122.6 Mean :47.92 Mean :12.44 Mean :20.96

3rd Qu.:0.2600 3rd Qu.:0.08000 3rd Qu.: 140.80 3rd Qu.:125.4 3rd Qu.:50.00 3rd Qu.:20.20 3rd Qu.:34.60

Max. :0.6200 Max. :0.24000 Max. : 229.40 Max. :140.2 Max. :60.40 Max. :58.00 Max. :59.40

NA's :23 NA's :39 NA's :2 NA's :32 NA's :22 NA's :11 NA's :15

yd Pressure3 Hyd Pressure4 Filler Level Filler Speed Temperature Usage cont Carb Flow

Min. :-1.20 Min. : 52.00 Min. : 55.8 Min. : 998 Min. :63.60 Min. :12.08 Min. : 26

1st Qu.: 0.00 1st Qu.: 86.00 1st Qu.: 98.3 1st Qu.:3888 1st Qu.:65.20 1st Qu.:18.36 1st Qu.:1144

Median :27.60 Median : 96.00 Median :118.4 Median :3982 Median :65.60 Median :21.79 Median :3028

Mean :20.46 Mean : 96.29 Mean :109.3 Mean :3687 Mean :65.97 Mean :20.99 Mean :2468

3rd Qu.:33.40 3rd Qu.:102.00 3rd Qu.:120.0 3rd Qu.:3998 3rd Qu.:66.40 3rd Qu.:23.75 3rd Qu.:3186

Max. :50.00 Max. :142.00 Max. :161.2 Max. :4030 Max. :76.20 Max. :25.90 Max. :5104

NA's :15 NA's :30 NA's :20 NA's :57 NA's :14 NA's :5 NA's :2

Density MFR Balling Pressure Vacuum PH Oxygen Filler Bowl Setpoint

Min. :0.240 Min. : 31.4 Min. :-0.170 Min. :-6.600 Min. :7.880 Min. :0.00240 Min. : 70.0

1st Qu.:0.900 1st Qu.:706.3 1st Qu.: 1.496 1st Qu.:-5.600 1st Qu.:8.440 1st Qu.:0.02200 1st Qu.:100.0

Median :0.980 Median :724.0 Median : 1.648 Median :-5.400 Median :8.540 Median :0.03340 Median :120.0

Mean :1.174 Mean :704.0 Mean : 2.198 Mean :-5.216 Mean :8.546 Mean :0.04684 Mean :109.3

3rd Qu.:1.620 3rd Qu.:731.0 3rd Qu.: 3.292 3rd Qu.:-5.000 3rd Qu.:8.680 3rd Qu.:0.06000 3rd Qu.:120.0

Max. :1.920 Max. :868.6 Max. : 4.012 Max. :-3.600 Max. :9.360 Max. :0.40000 Max. :140.0

NA's :1 NA's :212 NA's :1 NA's :4 NA's :12 NA's :2

Pressure Setpoint Air Pressurer Alch Rel Carb Rel Balling Lvl

Min. :44.00 Min. :140.8 Min. :5.280 Min. :4.960 Min. :0.00

1st Qu.:46.00 1st Qu.:142.2 1st Qu.:6.540 1st Qu.:5.340 1st Qu.:1.38

Median :46.00 Median :142.6 Median :6.560 Median :5.400 Median :1.48

Mean :47.62 Mean :142.8 Mean :6.897 Mean :5.437 Mean :2.05

3rd Qu.:50.00 3rd Qu.:143.0 3rd Qu.:7.240 3rd Qu.:5.540 3rd Qu.:3.14

Max. :52.00 Max. :148.2 Max. :8.620 Max. :6.060 Max. :3.66

NA's :12 NA's :9 NA's :10 NA's :1

A B C D

293 1239 304 615

From the summary statistics, several key observations emerged. First, there appears to be a class imbalance in terms of brand codes, which may necessitate the use of over/undersampling techniques to address potential bias. Additionally, the dataset contains missing values, including for the predictor variable pH, which will require imputation or other handling methods to ensure model accuracy. Some variables, such as fill ounces, exhibit low variance and will need a thorough check to determine their relevance. Implausible values were detected, particularly in the Balling scale, where negative values are not logically consistent with the measurement of sugar concentration. Furthermore, various pressure columns showed negative values, prompting the need for closer inspection to understand their validity within the production context. Overall, the distributions of the variables must be examined to ensure they are suitable for modeling. Outliers need to be identified and appropriately managed to prevent skewed results. Finally, multicollinearity among the variables should be addressed to avoid redundancy and ensure the robustness of the predictive models.

To begin our analysis, we explored the dataset to understand its structure and identify potential issues. We started by examining the distribution of the Brand Code variable, analyzing missing values, and looking at pH levels by brand code.

#### 

#### 

### Class Distribution

We visualized the distribution of brand codes to identify any class imbalances.

```{r}

table(train$`Brand Code`)

ggplot(train, aes(x = `Brand Code`)) +

geom\_bar(fill = "blue", color = "black") +

ggtitle("Distribution of Brand Codes") +

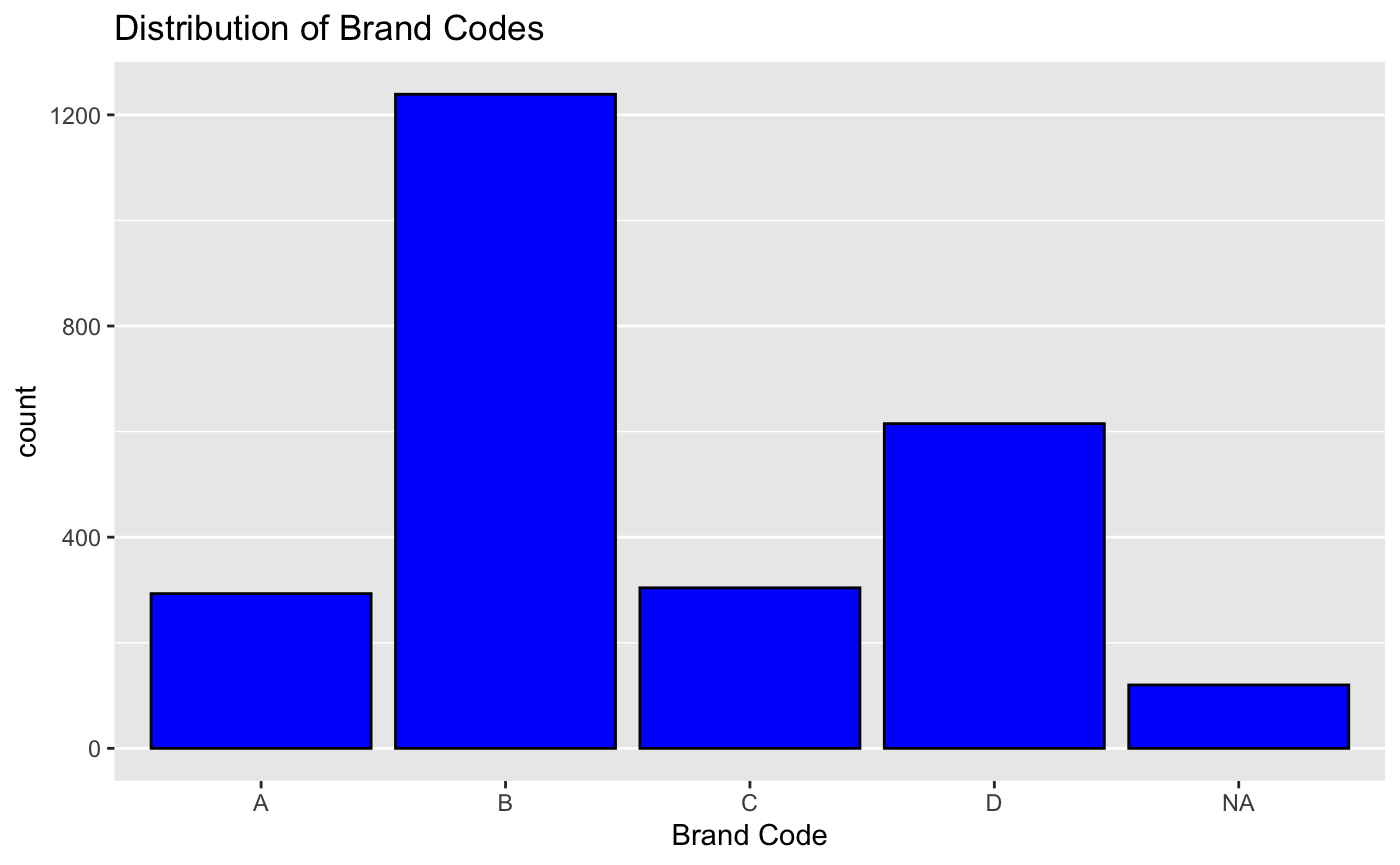
theme(

panel.grid.major.x = element\_blank(),

panel.grid.minor.x = element\_blank()

)

```



Brand B dominates the dataset, indicating a class imbalance that we may need to address.

We examined the missing values to see if there's a pattern related to brand codes.

```{r}

# Calculating the percentage of missing data for each brand code

missing\_percentage\_by\_class = train %>%

gather(key = "variable", value = "value", -`Brand Code`) %>%

group\_by(`Brand Code`, variable) %>%

summarize(missing\_count = sum(is.na(value)), total\_count = n()) %>%

mutate(missing\_percentage = (missing\_count / total\_count) \* 100) %>%

ungroup()

# Visualizing using a heatmap

ggplot(missing\_percentage\_by\_class, aes(x = `Brand Code`, y = variable, fill = missing\_percentage)) +

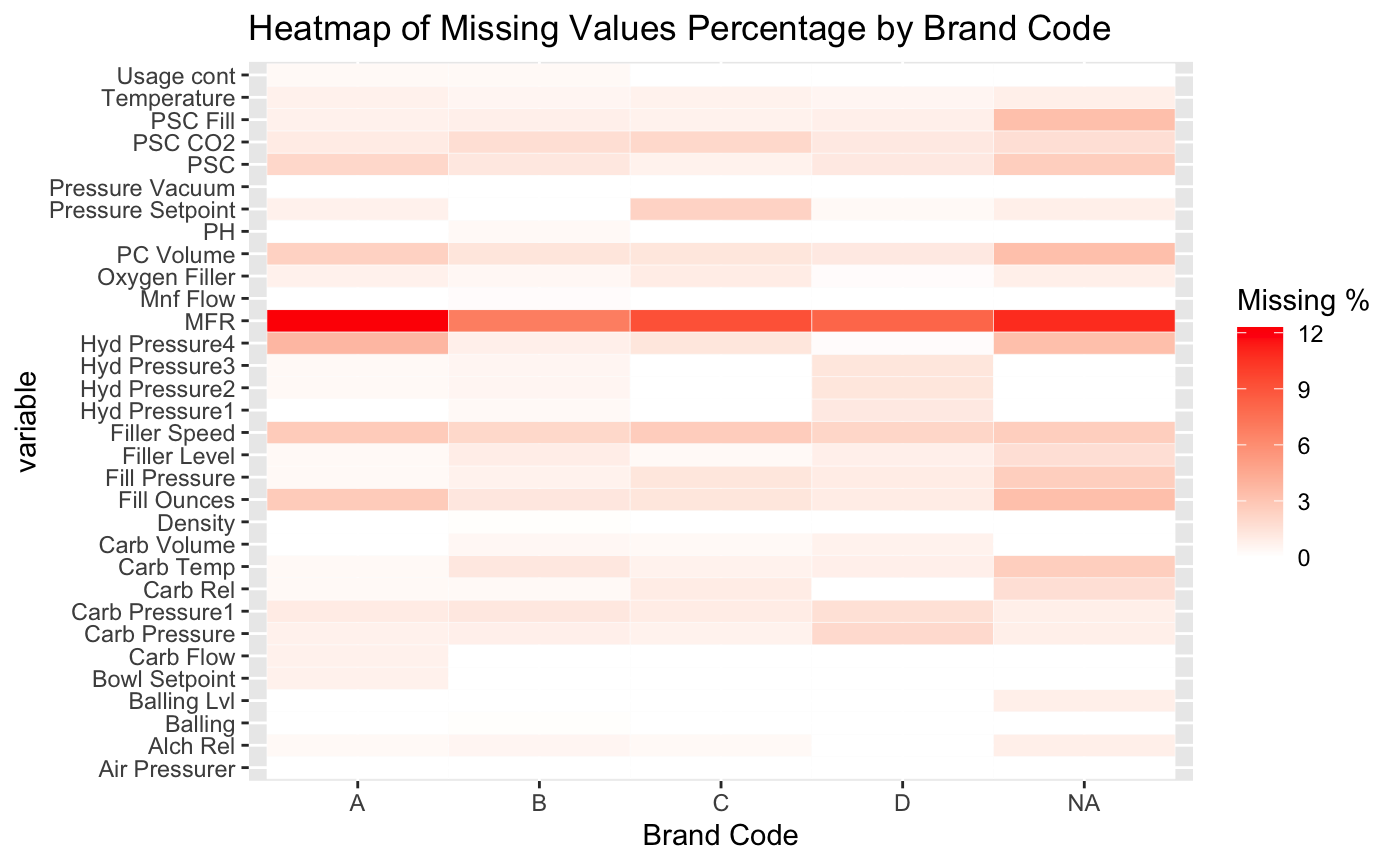
geom\_tile(color = "white") +

scale\_fill\_gradient(low = "white", high = "red") +

ggtitle("Heatmap of Missing Values Percentage by Brand Code") +

labs(fill = "Missing %")

```



The heatmap shows no significant pattern between brand codes and missing data, though some variables, like MFR, have substantial missing values across all brand codes.

#### 

### pH Levels by Brand Code

We explored pH levels by brand code to understand its impact.

```{r}

# First, we create summary statistics of PH by class

ph\_by\_class = train %>%

group\_by(`Brand Code`) %>%

summarize(mean\_PH = mean(PH, na.rm = TRUE),

sd\_PH = sd(PH, na.rm = TRUE),

median\_PH = median(PH, na.rm = TRUE),

min\_PH = min(PH, na.rm = TRUE),

max\_PH = max(PH, na.rm = TRUE))

print(ph\_by\_class)

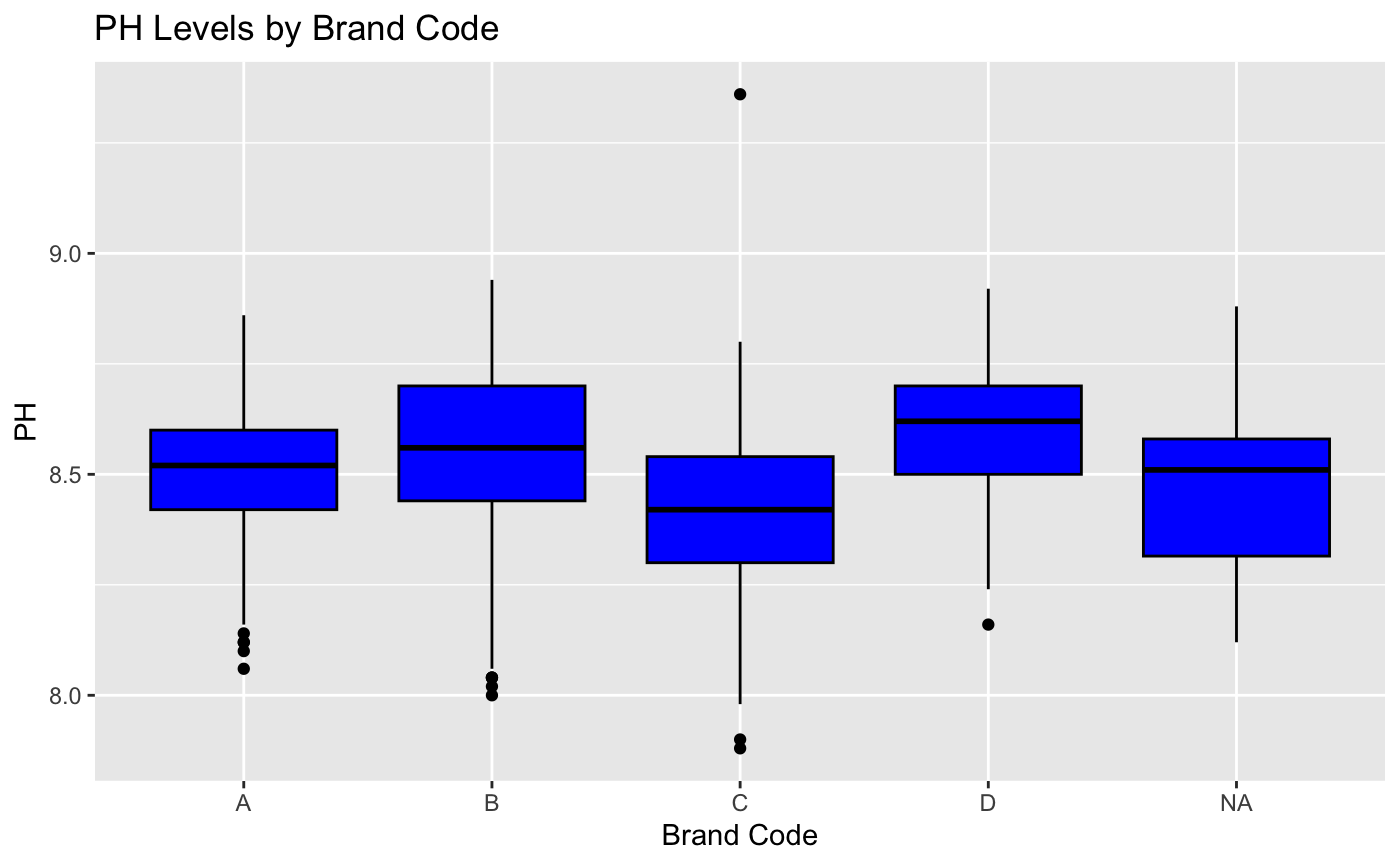
# And we visualize with a boxplot

ggplot(train, aes(x = `Brand Code`, y = PH)) +

geom\_boxplot(fill = "blue", color = "black") +

ggtitle("PH Levels by Brand Code")

```



The boxplot shows that while pH levels vary by brand code, the variation is not drastic. Brand code should be included as a categorical variable in the model, but balancing the brand codes is not a major concern.

These exploratory steps provided valuable insights into the dataset, guiding our data preprocessing and modeling stages.

### Low Variance Variables

Next, we checked for low or no variance variables. Zero-variance predictors are those that have only one unique value, which do not contribute to the predictive model.

```{r}

zero\_variance = function(column) {

return(length(unique(column)) == 1)

}

print(names(train)[sapply(train, zero\_variance)])

```

character(0)

As expected, there were no zero-variance predictors in our dataset.

Next, following the method by Johnson and Kuhn, we looked for near-zero variance predictors. These occur when the fraction of unique values over the sample size is low and when the ratio of the most prevalent value's frequency to the second most prevalent value's frequency is large.

```{r}

near\_zero\_variance = function(column) {

unique\_values = length(unique(column))

most\_frequent\_value\_count = max(table(column))

second\_most\_frequent\_value\_count = sort(table(column), decreasing = TRUE)[2]

fraction\_unique\_values = unique\_values / length(column)

ratio\_most\_to\_second = most\_frequent\_value\_count / second\_most\_frequent\_value\_count

return(fraction\_unique\_values < 0.10 & ratio\_most\_to\_second > 20)

}

print(names(train)[sapply(train, near\_zero\_variance)])

```

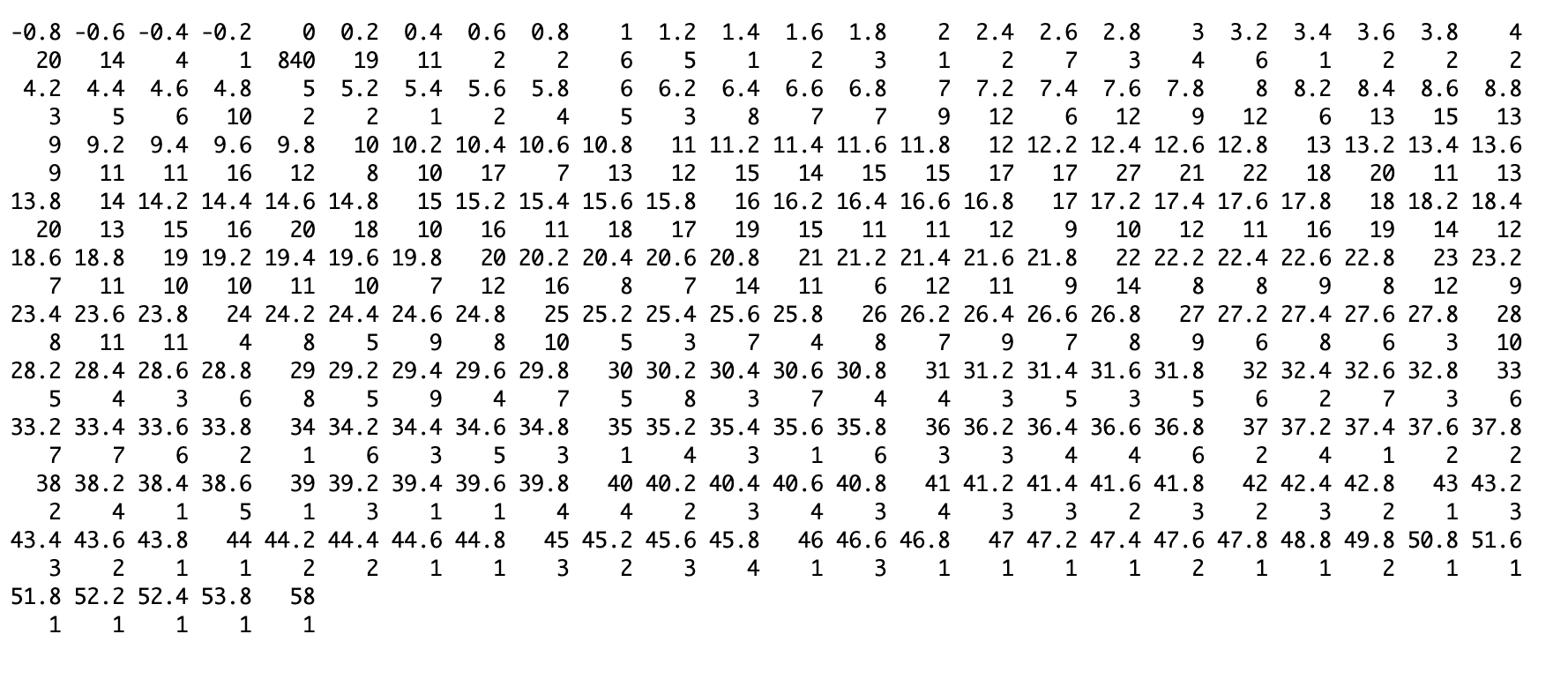
[1] "Hyd Pressure1"

From this analysis, we identified the Hyd Pressure1 variable as having near-zero variance. We further examined its frequencies:

```{r}

table(train$`Hyd Pressure1`)

```



We found that the value 0 occurs 840 times, much more frequently than any other value. This high frequency indicates that Hyd Pressure1 does not provide much variability and is not likely to be useful in our predictive model.

In addition to the formulaic approach, we computed the variance of each column to identify columns with incredibly low variance that might still pass Johnson and Kuhn's test. We examined columns with a variance of less than 0.01:

```{r}

numeric\_vars = train %>% select\_if(is.numeric)

variances = sapply(numeric\_vars, var, na.rm = TRUE)

print(names(variances[variances < 0.01]))

```

[1] "Fill Ounces" "PC Volume" "PSC" "PSC CO2" "Oxygen Filler"

The variance analysis revealed that the following predictor variables have very low variance:

* Fill Ounces
* PC Volume
* PSC
* PSC CO2
* Oxygen Filler

The variance of these predictor variables is very low, and so they likely contribute little to the model's predictive power. Removing these low variance predictors can simplify the model without sacrificing performance, especially since there are many other predictor variables.

We will remove:

* Hyd Pressure1
* PC Volume
* PSC
* PSC CO2
* Oxygen Filler

We removed these low variance columns from both the training and test datasets:

```{r}

low\_var\_cols = c("Hyd Pressure1", "PC Volume", "PSC", "PSC CO2", "Oxygen Filler")

train = train %>% dplyr::select(-all\_of(low\_var\_cols))

test = test %>% dplyr::select(-all\_of(low\_var\_cols))

```

By performing these steps, we ensured that our model would not be bogged down by variables that do not provide meaningful information, thereby improving its efficiency and performance.

### Handle Implausible Values

We now move to handle implausible values. Because we removed some low variance predictors, there are fewer problematic values to address:

This variable likely represents the rate at which a drink flows through a system during production. We previously observed negative values, which seem highly implausible. Let's take a closer look:

```{r}

train %>% ggplot(aes(x = `Mnf Flow`)) +

geom\_histogram(binwidth = 10, fill = "blue", color = "black") +

ggtitle("Histogram of MNF Flow") +

labs(x = "MNF Flow", y = "Frequency") +

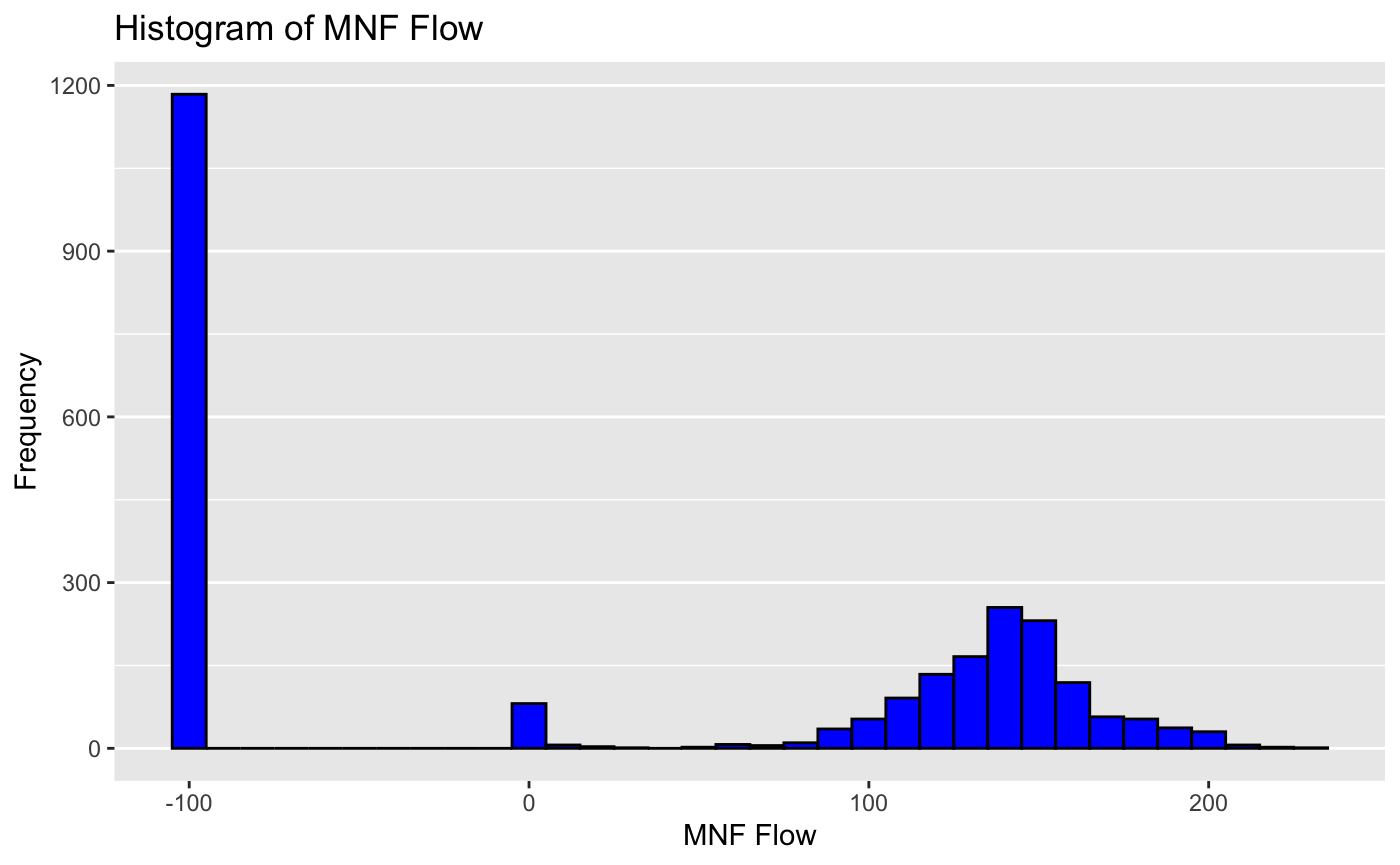
theme(

panel.grid.major.x = element\_blank(),

panel.grid.minor.x = element\_blank()

)

```



Upon inspecting the distribution of MNF Flow, we observed an unusual pattern. The distribution is nearly normal on the right side with a small peak around 0. However, approximately half of the values are close to -100, which is implausible and likely reflects something other than the true MNF Flow. To address this, we created a categorical variable that indicates whether MNF Flow is positive or negative.

```{r}

train = train %>%

mutate(`Mnf Flow Positive` = ifelse(`Mnf Flow` > 0, "Yes", "No"))

test = test %>%

mutate(`Mnf Flow Positive` = ifelse(`Mnf Flow` > 0, "Yes", "No"))

```

Next, we examined whether this newly created categorical variable (Mnf Flow Positive) is predictive of pH levels. We used a box plot to visualize the distribution of pH values based on the Mnf Flow Positive categories.

```{r}

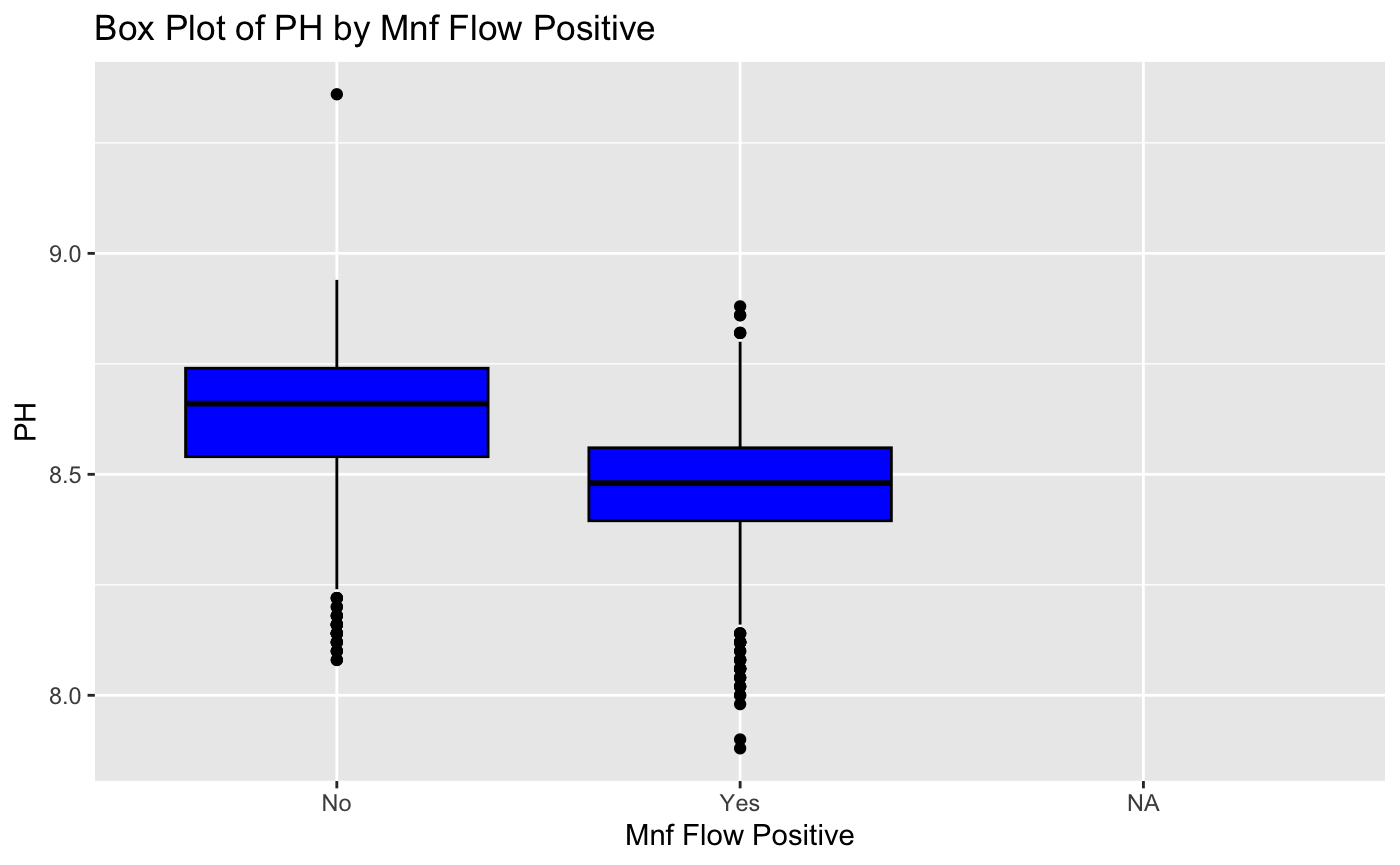
ggplot(train, aes(x = `Mnf Flow Positive`, y = PH)) +

geom\_boxplot(fill = "blue", color = "black") +

ggtitle("Box Plot of PH by Mnf Flow Positive") +

labs(x = "Mnf Flow Positive", y = "PH")

```

​​

The box plot indicates that the Mnf Flow Positive variable is meaningful. It shows a distinct difference in pH levels between the categories, suggesting that this variable could be a good predictor for our model.

Given this insight, we will drop the original Mnf Flow variable. Converting the negative values to NAs would result in around half of the data being missing, making the original variable less useful.

```{r}

train = train %>% dplyr::select(-`Mnf Flow`)

test = test %>% dplyr::select(-`Mnf Flow`)

```

By removing the original MNF Flow variable, we ensure that our dataset remains as complete as possible while retaining the meaningful information provided by the Mnf Flow Positive categorical variable. This preprocessing step helps improve the quality and reliability of our subsequent analyses and models.

### Balling

As aforementioned, Balling refers to the Balling scale, which reflects the concentration of sugar. Negative values for this variable are not realistic. We will replace negative values with 0.

```{r}

train = train %>%

mutate(Balling = ifelse(Balling < 0, 0, Balling))

test = test %>%

mutate(Balling = ifelse(Balling < 0, 0, Balling))

```

#### 

Initially, there were four variables related to "Hyd Pressure." We removed one of those variables due to low variance. Hyd Pressure likely refers to the hydraulic pressures at different points in the manufacturing process. We observed some negative values, which are unlikely to be realistic. We can imagine that there might be a slight vacuum at certain points, so we'll set the absolute lowest value possible for these variables to -1.2.

The variables are:

* Hyd Pressure2
* Hyd Pressure3
* Hyd Pressure4

```{r}

train = train %>%

mutate(

`Hyd Pressure2` = ifelse(`Hyd Pressure2` < -1.2, -1.2, `Hyd Pressure2`),

`Hyd Pressure3` = ifelse(`Hyd Pressure3` < -1.2, -1.2, `Hyd Pressure3`),

`Hyd Pressure4` = ifelse(`Hyd Pressure4` < -1.2, -1.2, `Hyd Pressure4`)

)

test = test %>%

mutate(

`Hyd Pressure2` = ifelse(`Hyd Pressure2` < -1.2, -1.2, `Hyd Pressure2`),

`Hyd Pressure3` = ifelse(`Hyd Pressure3` < -1.2, -1.2, `Hyd Pressure3`),

`Hyd Pressure4` = ifelse(`Hyd Pressure4` < -1.2, -1.2, `Hyd Pressure4`)

)

```

### Missing Values

As discussed, there are missing values. Let's get a sense of the percentage missing for each variable:

```{r}

missing\_values = train %>%

summarize\_all(~mean(is.na(.))) %>%

gather(key = "column", value = "missing\_percent") %>%

mutate(missing\_percent = missing\_percent \* 100)

# Now let's visualize

ggplot(missing\_values, aes(x = reorder(column, -missing\_percent), y = missing\_percent)) +

geom\_bar(stat = "identity", fill = "blue") +

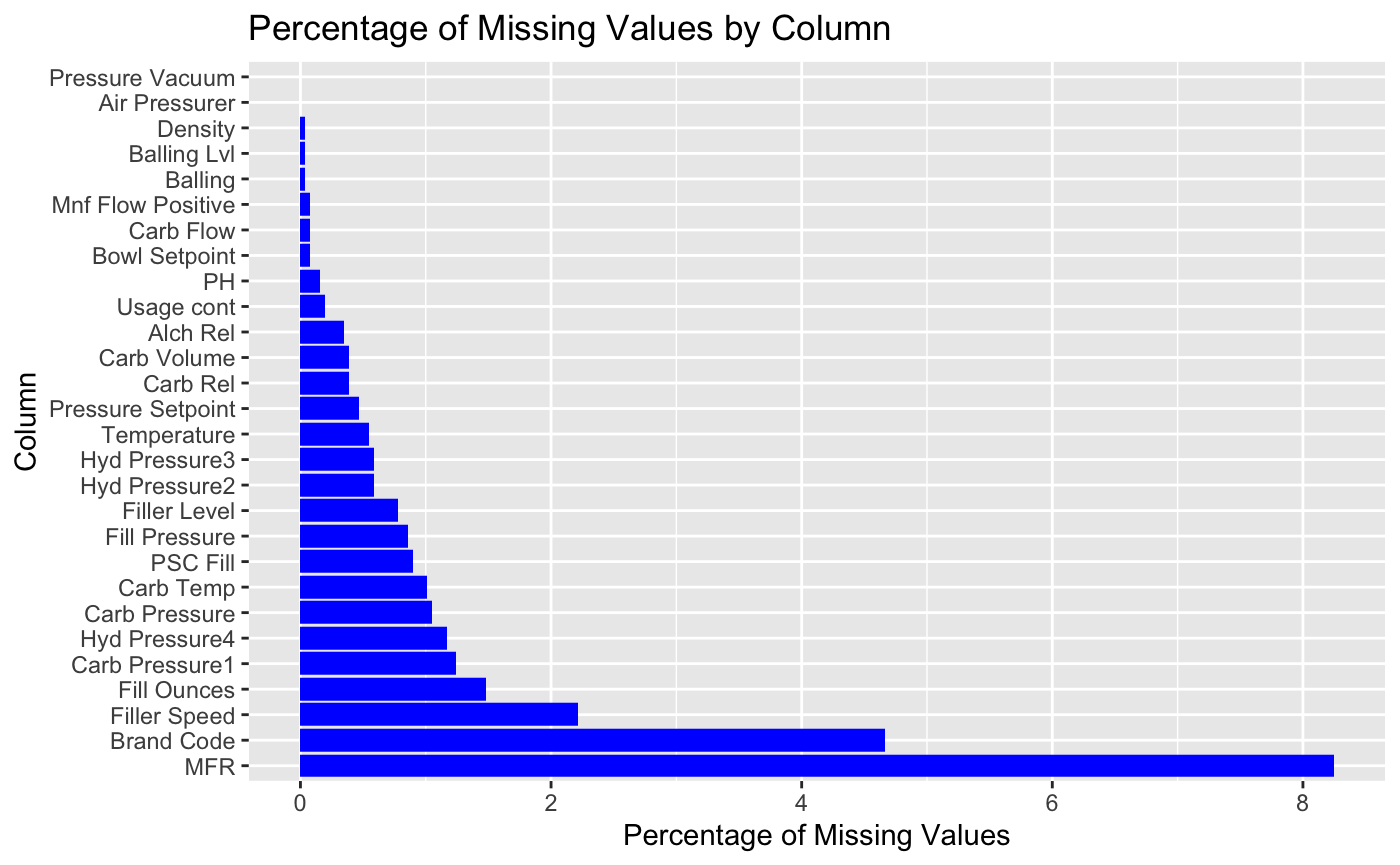
coord\_flip() +

xlab("Column") +

ylab("Percentage of Missing Values") +

ggtitle("Percentage of Missing Values by Column")

```



First, rows missing the PH value are problematic since that is the response variable. We will simply remove those 4 rows:

```{r}

train = train %>% filter(!is.na(PH))

```

Next, we don’t have any deeply problematic variables in terms of missing data. The variable MFR has the highest percentage of missing values, with over 8% of its entries missing. However, before we invest significant effort into imputing the missing values for MFR, it's important to determine if it is even a strong predictor of the PH value. If MFR is not highly predictive, we might decide to exclude it from our analysis instead of dealing with its missing values.

This step ensures that our focus remains on the most impactful variables, optimizing the effectiveness of our predictive model while maintaining data integrity.

```{r}

ggplot(train, aes(x = `MFR`, y = `PH`)) +

geom\_point() +

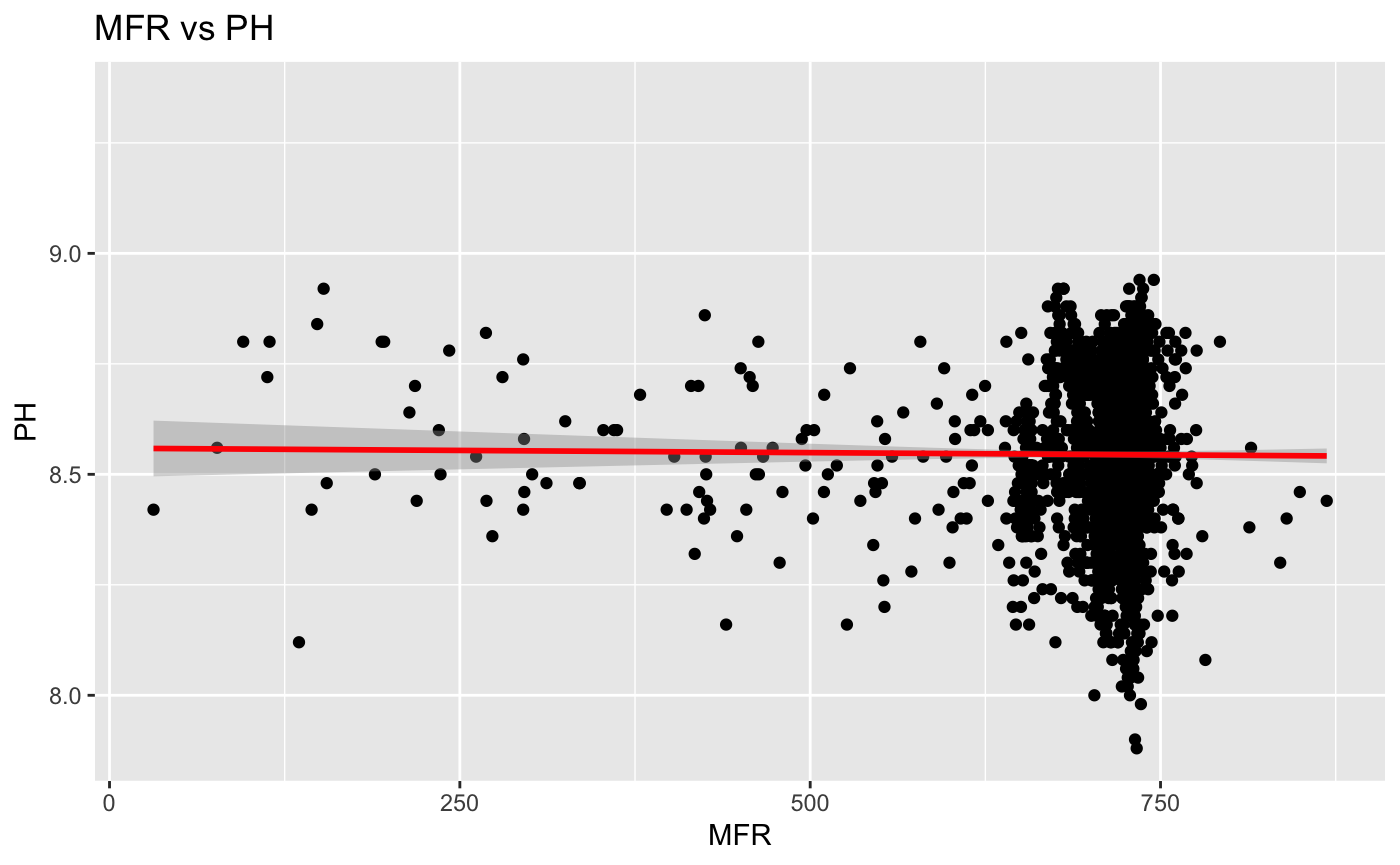
geom\_smooth(method = "lm", col = "red") +

ggtitle("MFR vs PH") +

xlab("MFR") +

ylab("PH")

```



By looking at our graph we can see that is not predictive, there appears to be many outliers, and it is missing a good chunk of its values. We will just drop the column altogether.

```{r}

train = train %>% dplyr::select(-`MFR`)

test = test %>% dplyr::select(-`MFR`)

```

#### Imputing Missing Values

At this point, we'd like to impute the other missing values. We will first split the training set into training and evaluation sets.

```{r}

library(caret)

set.seed(7)

trainIndex = createDataPartition(train$PH, p = 0.8, list = FALSE) # 80/20 split

train\_split = train[trainIndex, ]

eval\_split = train[-trainIndex, ]

# Confirming the split worked as planned:

dim(train\_split)

dim(eval\_split)

```

Step 1: Preparing Data for Imputation

``{r}

train\_split$`Brand Code` = as.factor(train\_split$`Brand Code`)

eval\_split$`Brand Code` = as.factor(eval\_split$`Brand Code`)

test$`Brand Code` = as.factor(test$`Brand Code`)

# just in case the tibble structure messes things up

train\_split\_df = as.data.frame(train\_split)

eval\_split\_df = as.data.frame(eval\_split)

test\_df = as.data.frame(test)

numeric\_vars\_train = train\_split\_df %>% select\_if(is.numeric)

numeric\_vars\_eval = eval\_split\_df %>% select\_if(is.numeric)

numeric\_vars\_test = test\_df %>% select\_if(is.numeric)

# The test set doesn't have PH and we're not missing values anyways so let's exclude it

numeric\_vars\_train = numeric\_vars\_train %>% dplyr::select(-PH)

numeric\_vars\_eval = numeric\_vars\_eval %>% dplyr::select(-PH)

factor\_vars\_train = train\_split\_df %>% select\_if(is.factor)

factor\_vars\_eval = eval\_split\_df %>% select\_if(is.factor)

factor\_vars\_test = test\_df %>% select\_if(is.factor)

```

Step 2: Imputing Numeric Variables with RANN

```{r}

library(RANN)

# kNN imputation with k=5

preProc = preProcess(numeric\_vars\_train, method = "knnImpute", k = 5)

# we want to get an imputation model only on the training set to avoid data leakage

train\_numeric\_imputed = predict(preProc, numeric\_vars\_train)

# Now we can apply the same model to the evaluation and test sets

eval\_numeric\_imputed = predict(preProc, numeric\_vars\_eval)

test\_numeric\_imputed = predict(preProc, numeric\_vars\_test)

```

Step 3: Predicting Brand Code with Logistic Regression

# First, we want to combine numeric and factor variables

train\_combined = cbind(train\_numeric\_imputed, train\_split\_df %>% dplyr::select(`Brand Code`))

eval\_combined = cbind(eval\_numeric\_imputed, eval\_split\_df %>% dplyr::select(`Brand Code`))

test\_combined = cbind(test\_numeric\_imputed, test\_df %>% dplyr::select(`Brand Code`))

# Defining function to fit a logistic regression model to predict Brand Code

impute\_brand\_code = function(data) {

missing = is.na(data$`Brand Code`)

# if no missing values, return column as is

if (sum(missing) == 0) {

return(data$`Brand Code`)

}

# defining a logistic regression formula using all features to predict Brand Code

formula = as.formula(`Brand Code` ~ .)

model = glm(formula, data = data[!missing, ], family = binomial)

return(model)

}

# Fit model on the training set

model = impute\_brand\_code(train\_combined)

```{r}

# Now defining a function to predict missing Brand Code values using the fitted model

predict\_brand\_code = function(model, data) {

missing = is.na(data$`Brand Code`)

if (sum(missing) == 0) {

return(data$`Brand Code`)

}

predictions = predict(model, newdata = data[missing, ], type = "response")

predicted\_levels = ifelse(predictions > 0.5, levels(data$`Brand Code`)[2], levels(data$`Brand Code`)[1])

data$`Brand Code`[missing] = predicted\_levels

return(data$`Brand Code`)

}

# Applying to all sets

train\_combined$`Brand Code` = predict\_brand\_code(model, train\_combined)

eval\_combined$`Brand Code` = predict\_brand\_code(model, eval\_combined)

test\_combined$`Brand Code` = predict\_brand\_code(model, test\_combined)

```

Step 4: Final Combination

```{r}

train\_split\_imputed = cbind(train\_combined, train\_split\_df %>% dplyr::select(`PH`, `Mnf Flow Positive`))

eval\_split\_imputed = cbind(eval\_combined, eval\_split\_df %>% dplyr::select(`PH`, `Mnf Flow Positive`))

test\_imputed = cbind(test\_combined, test\_df %>% dplyr::select(`Mnf Flow Positive`)) # Test set doesn't have PH column to impute

```

### 

### 

### 

### Handling Outliers

First, let's visualize the numeric variables to identify potential outliers. We will create boxplots to visualize this.

```{r}

numeric\_cols = colnames(train\_split\_imputed)[sapply(train\_split\_imputed, is.numeric)]

# Plotting box plots for each numeric variable

for (col in numeric\_cols) {

p = ggplot(train\_split\_imputed, aes(x = factor(0), y = .data[[col]])) +

geom\_boxplot() +

ggtitle(paste("Boxplot of", col)) +

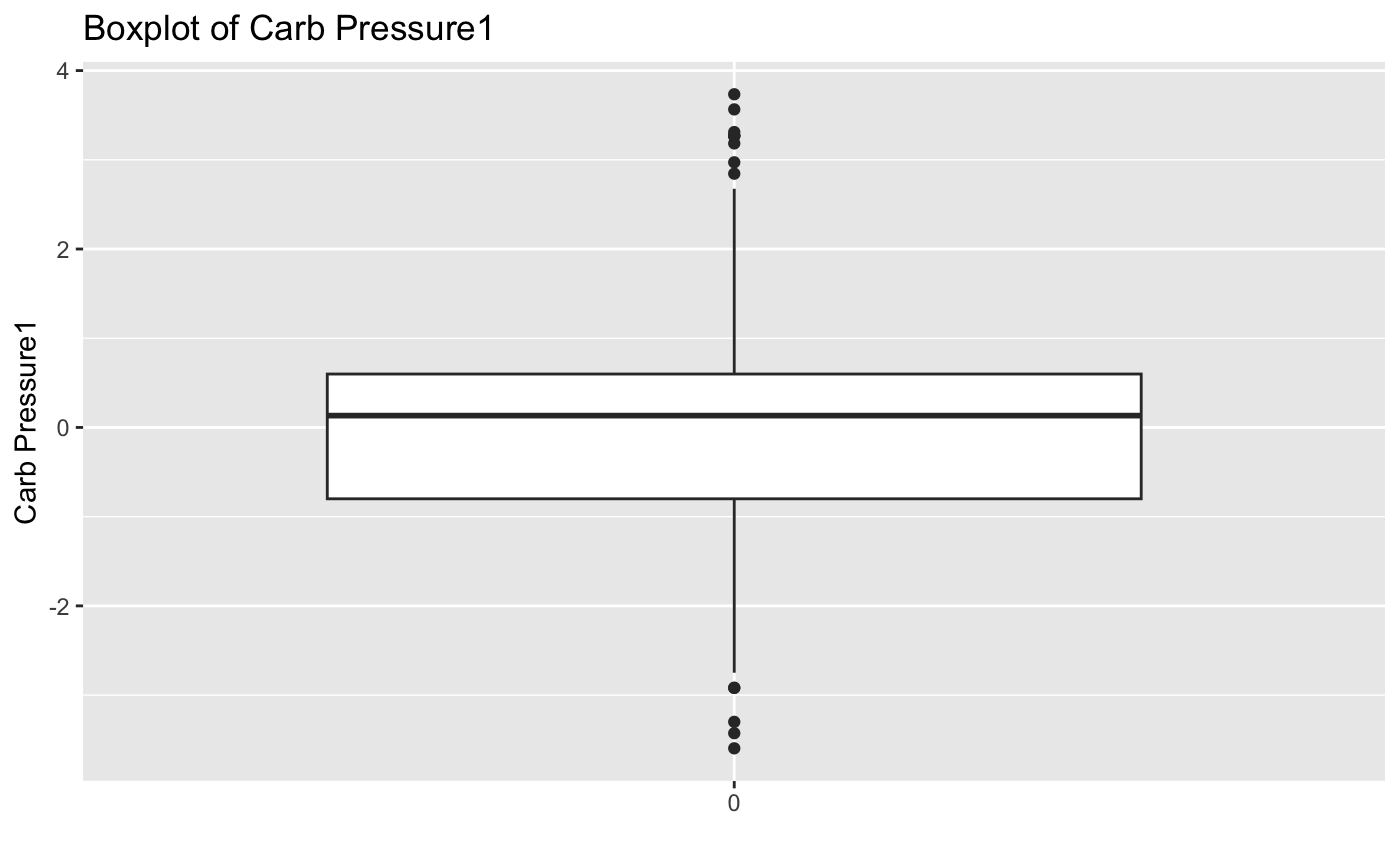
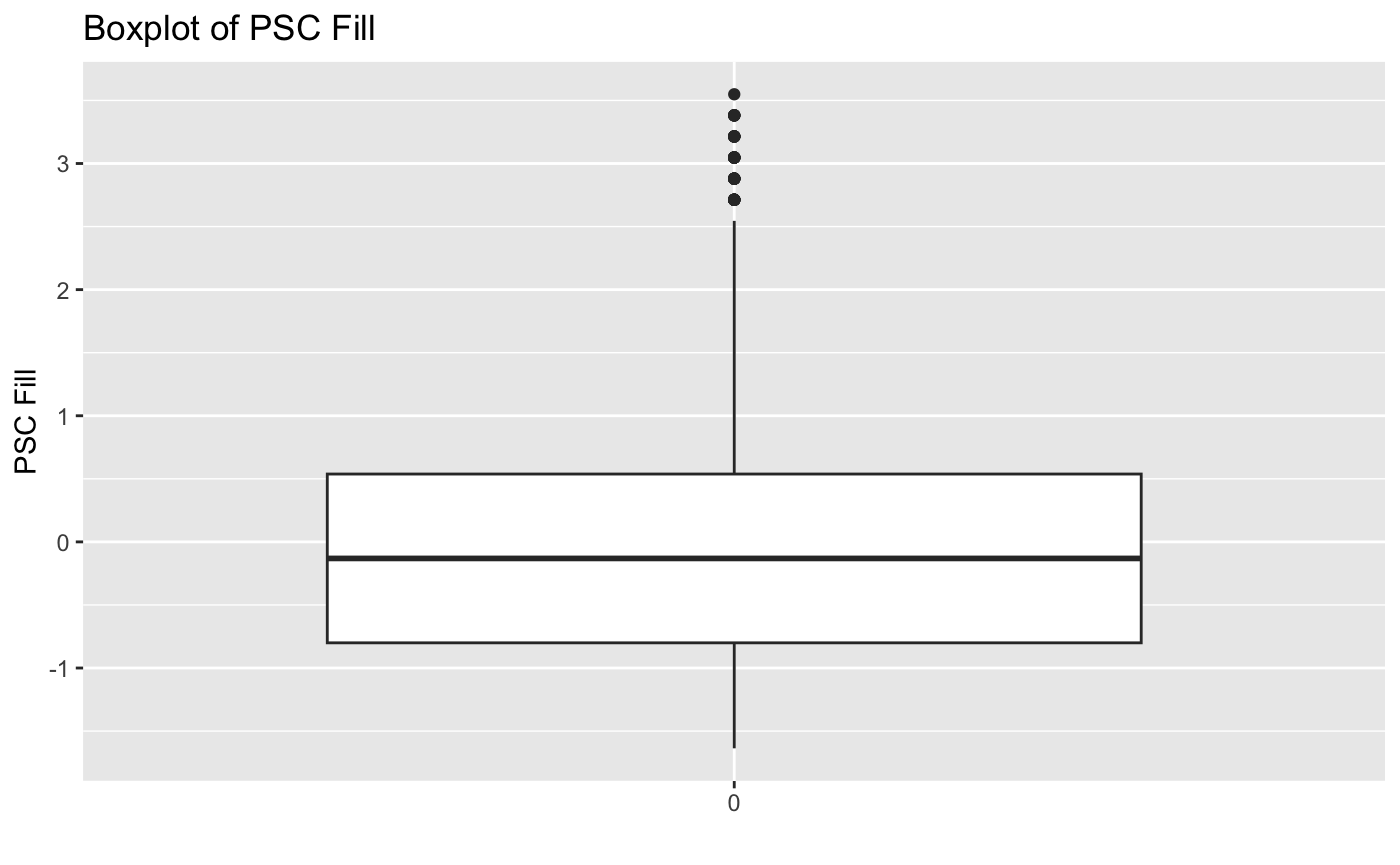
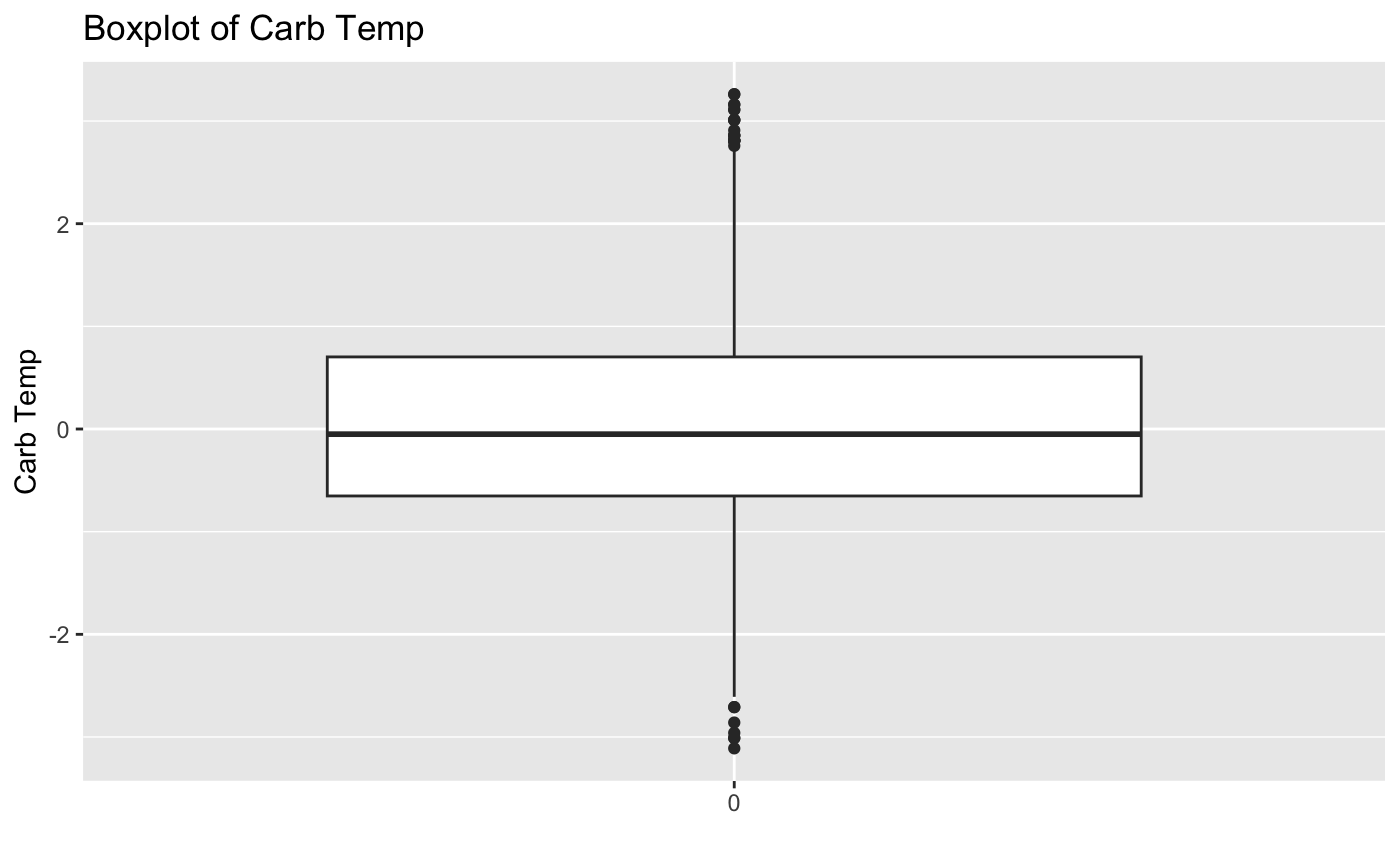
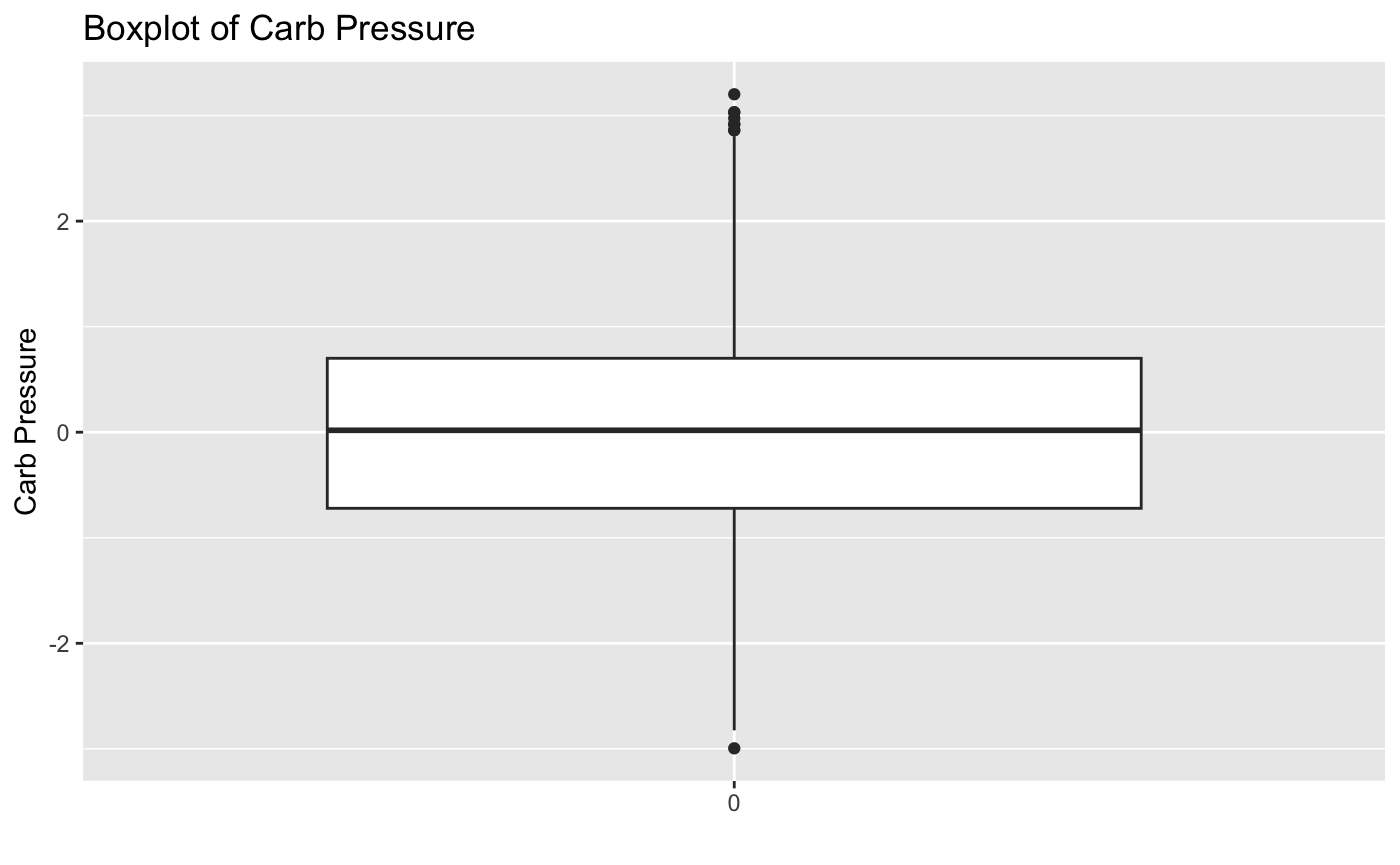
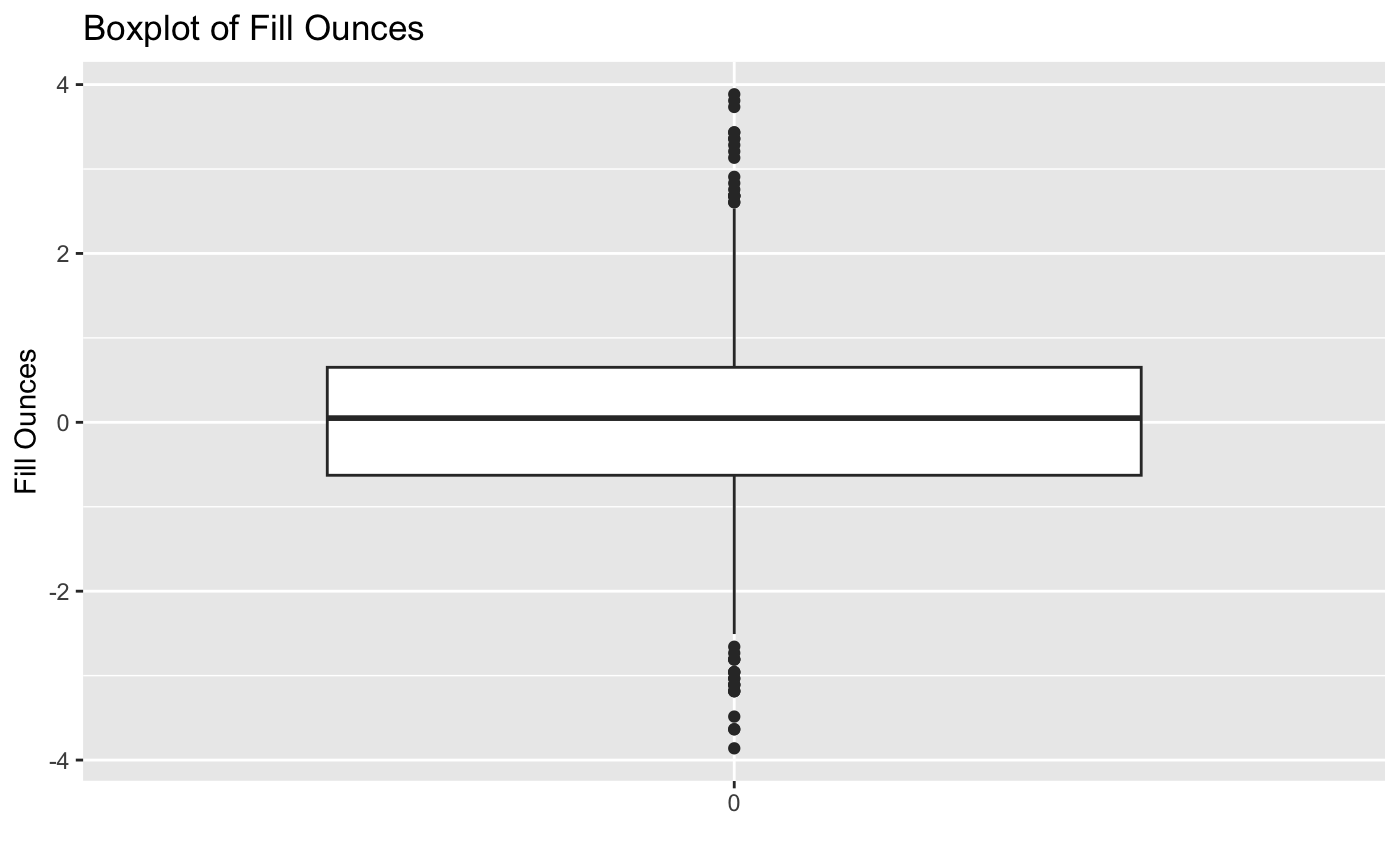
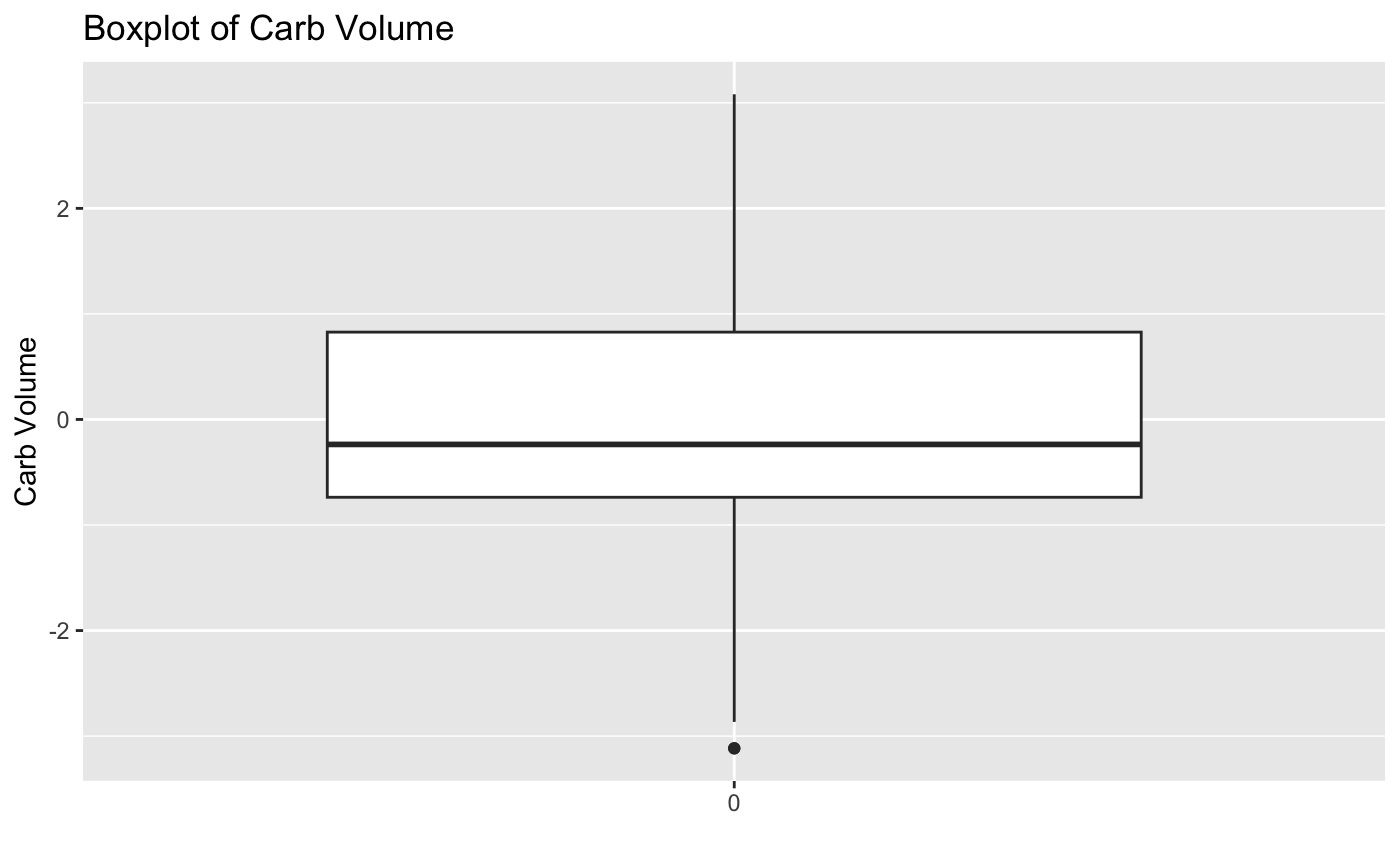
xlab("") +

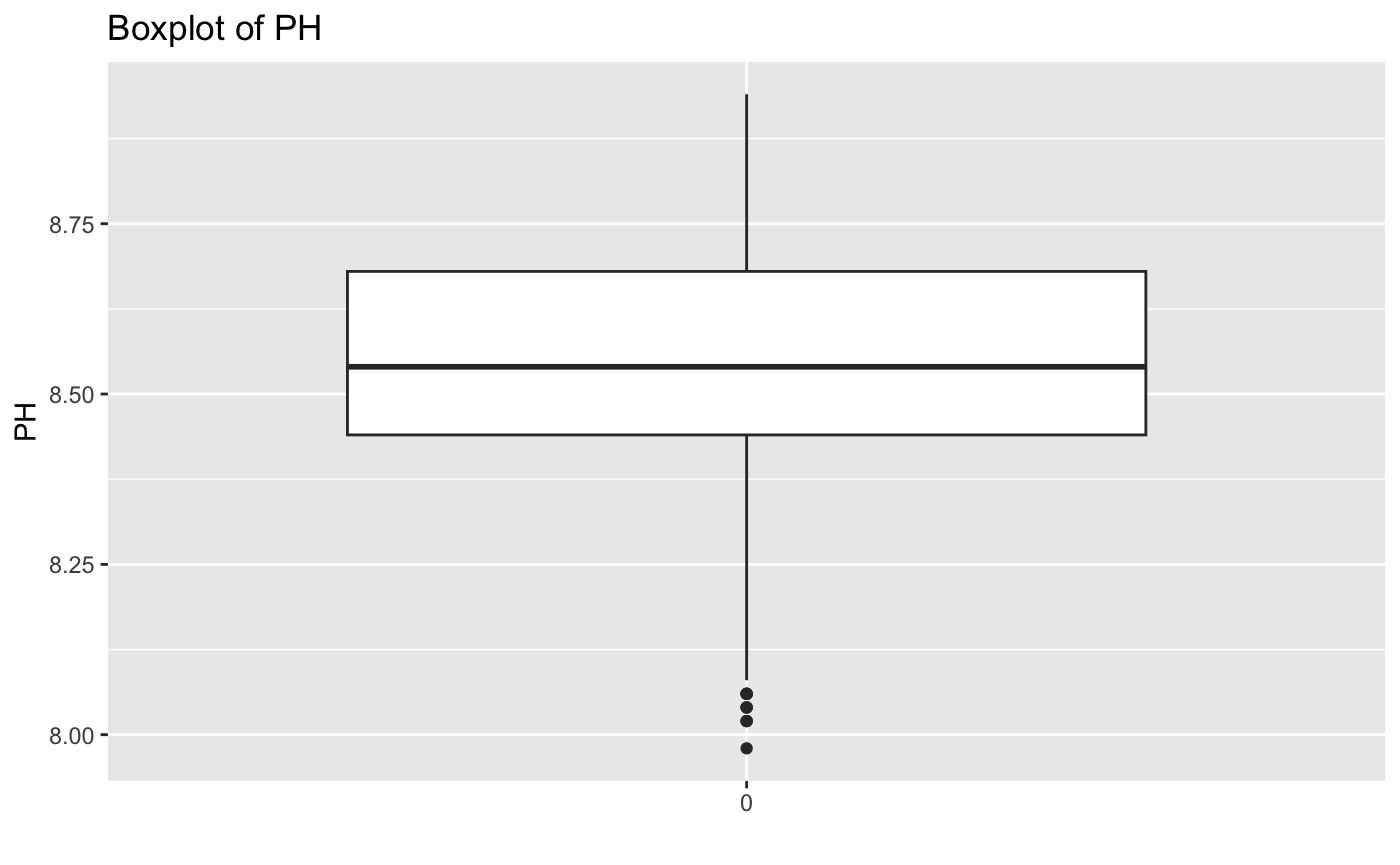
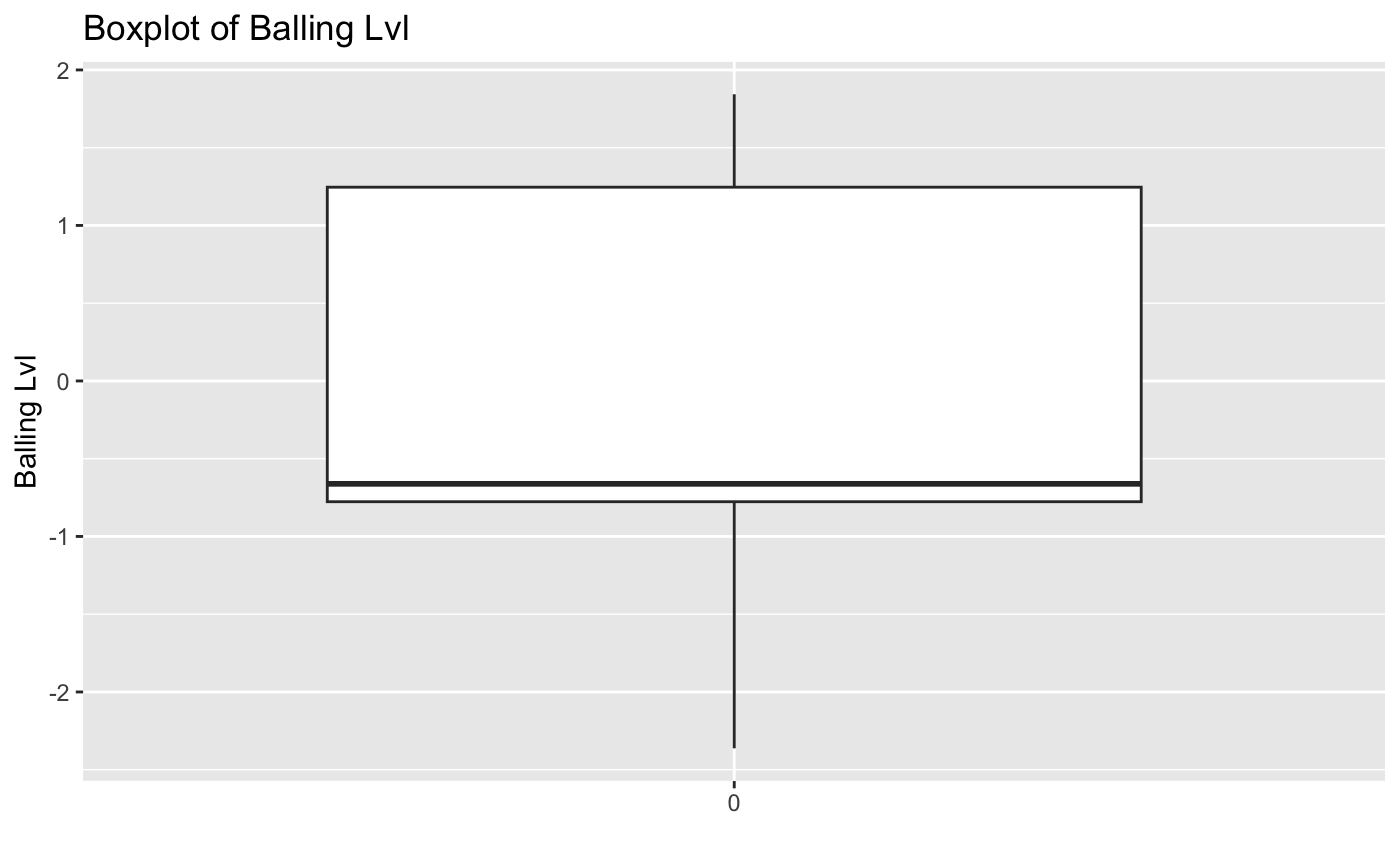
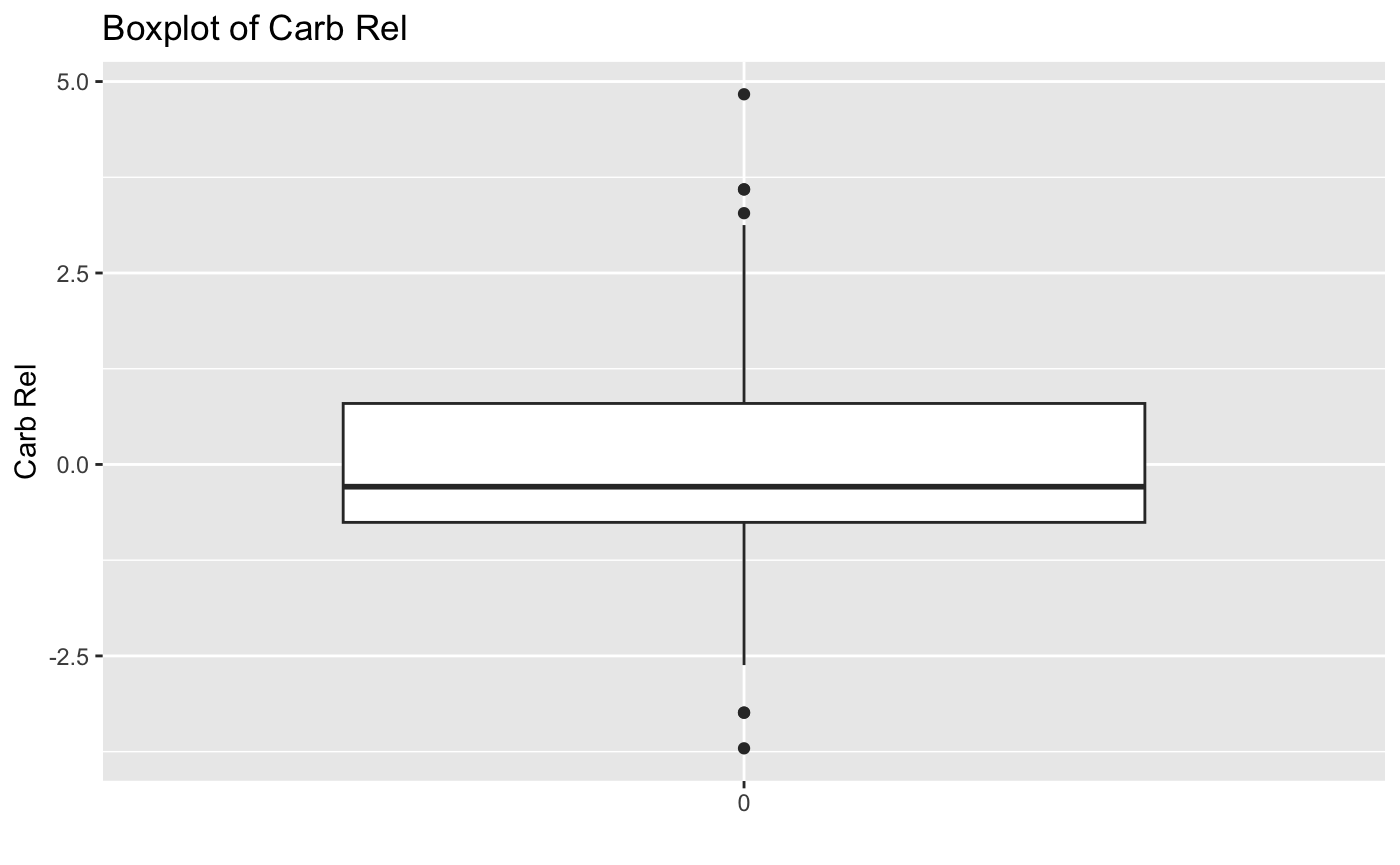
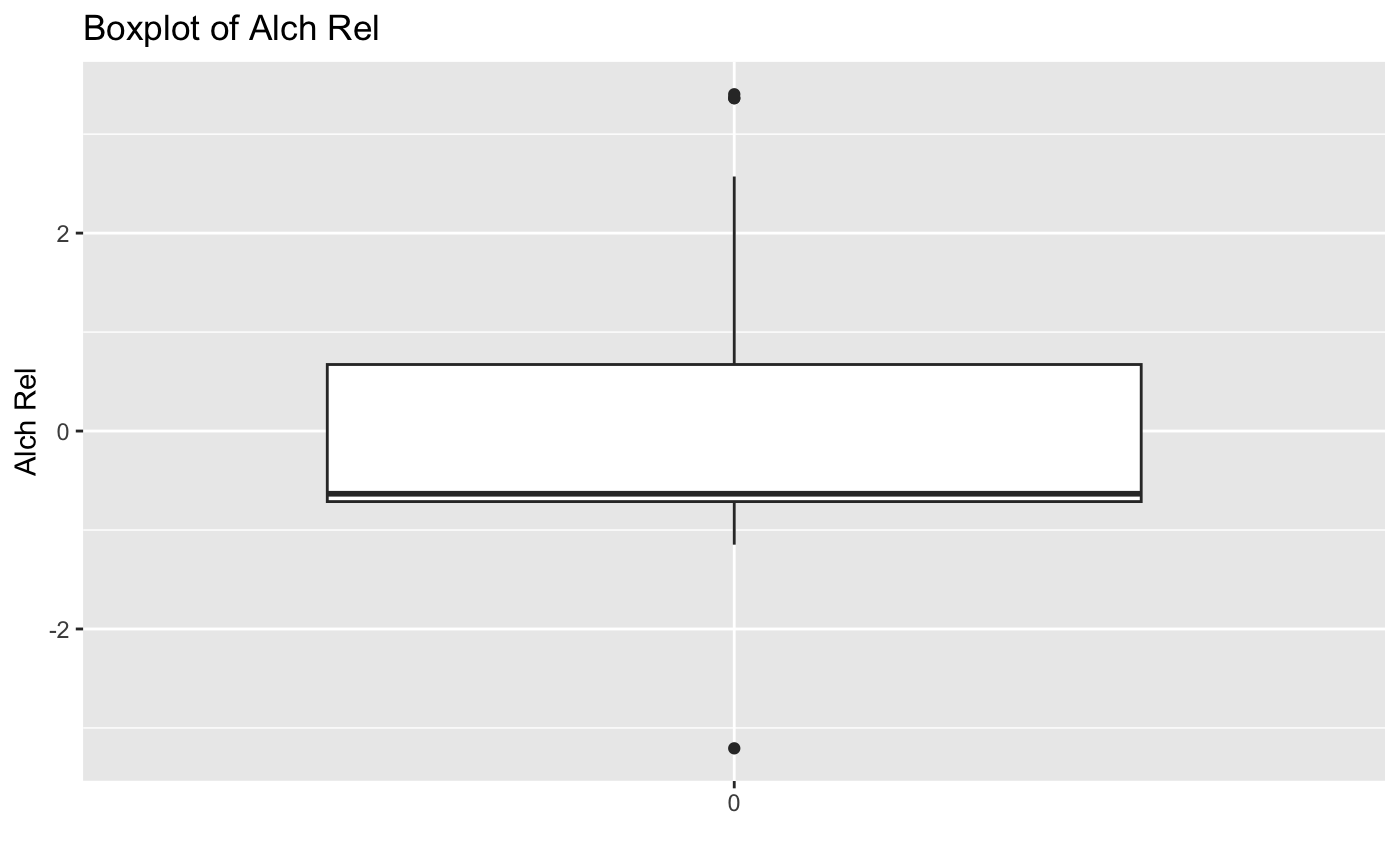
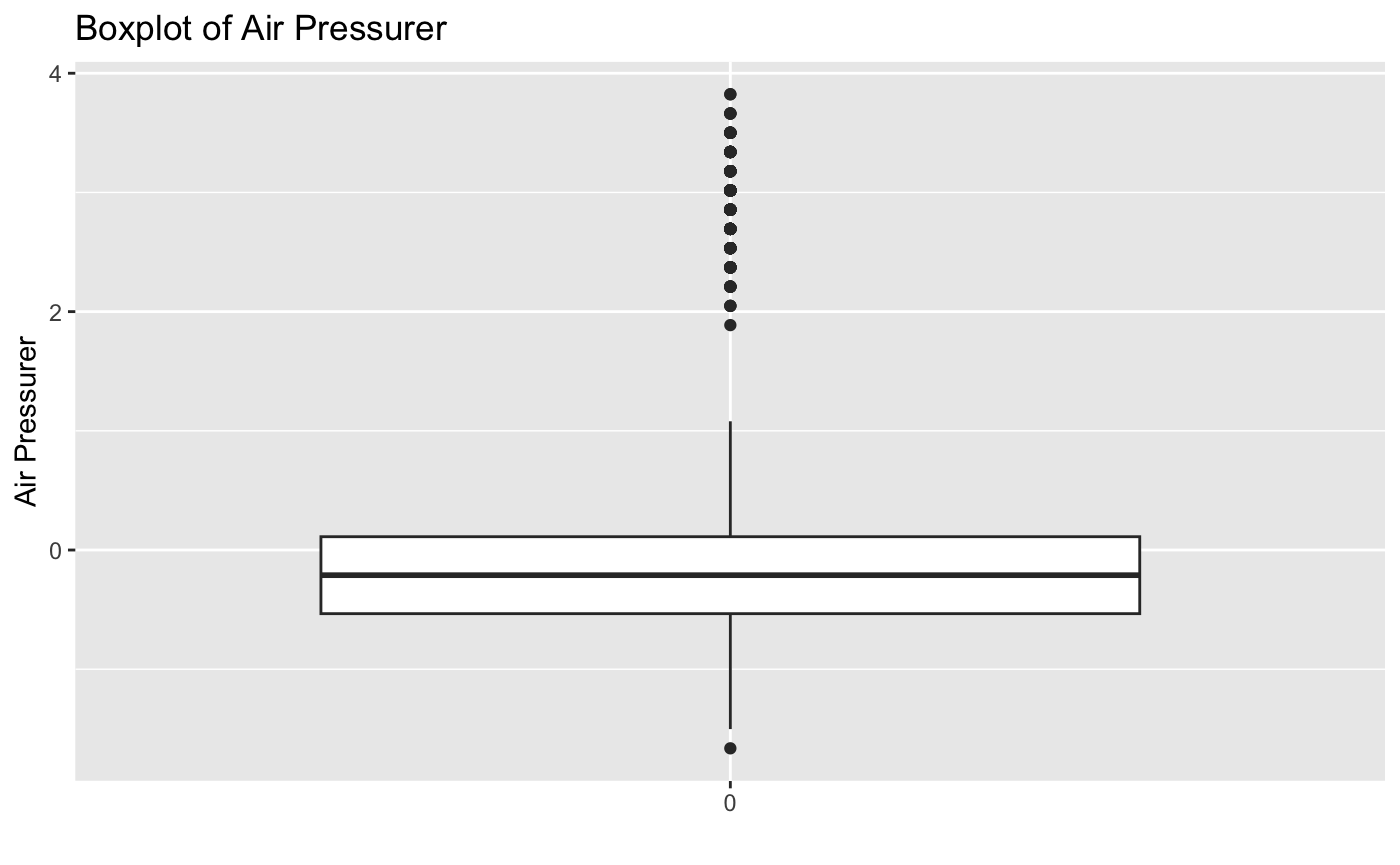
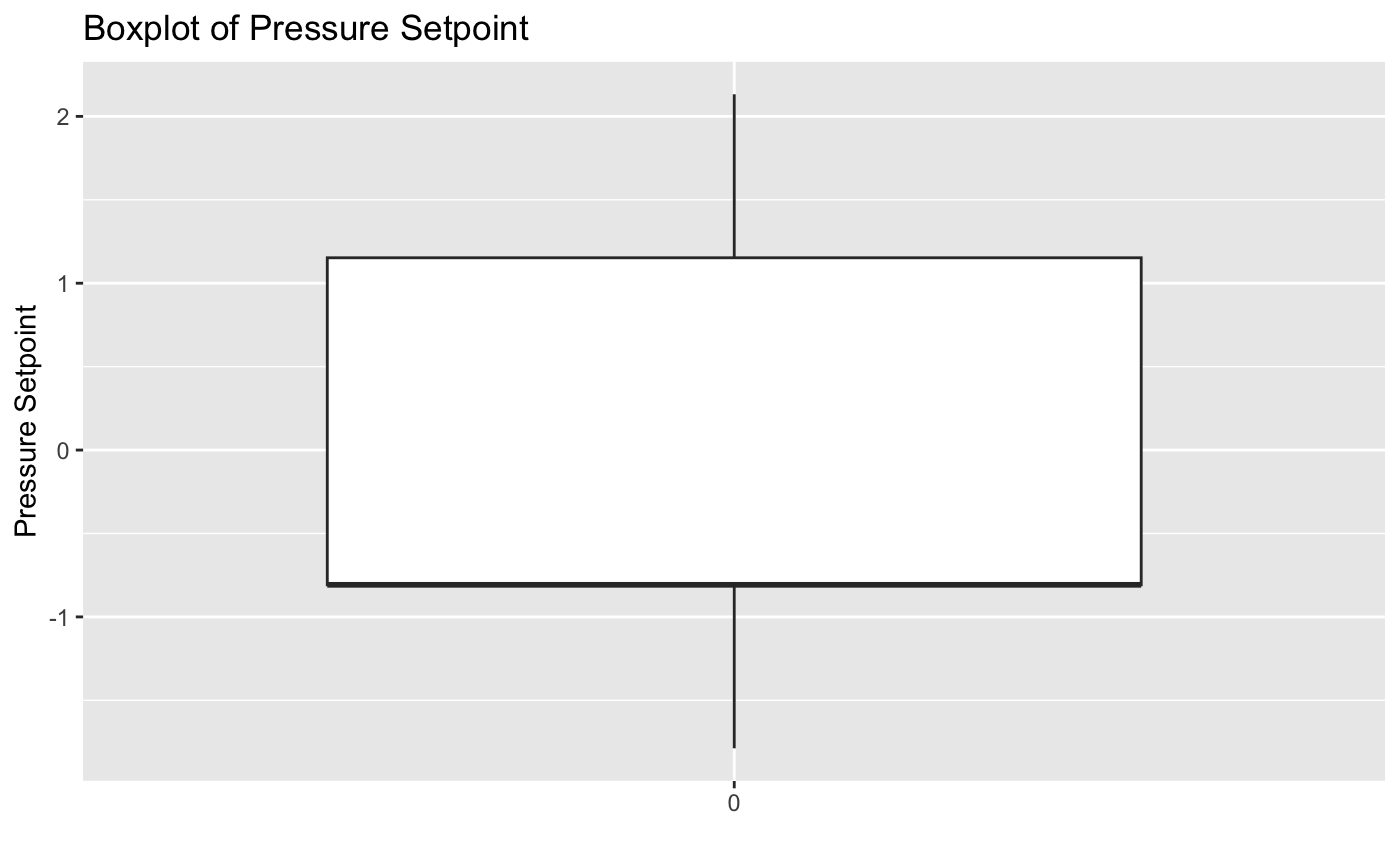
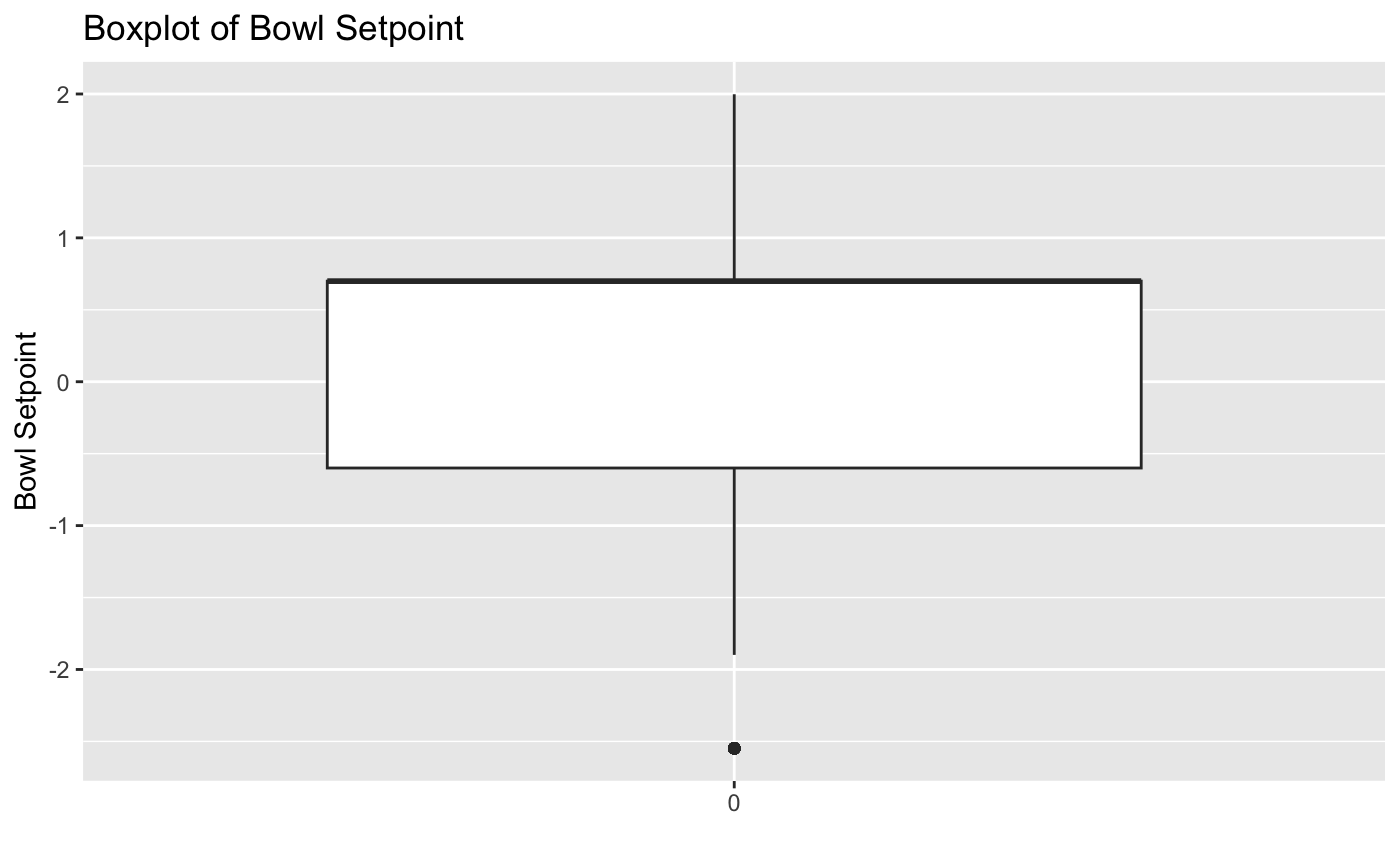
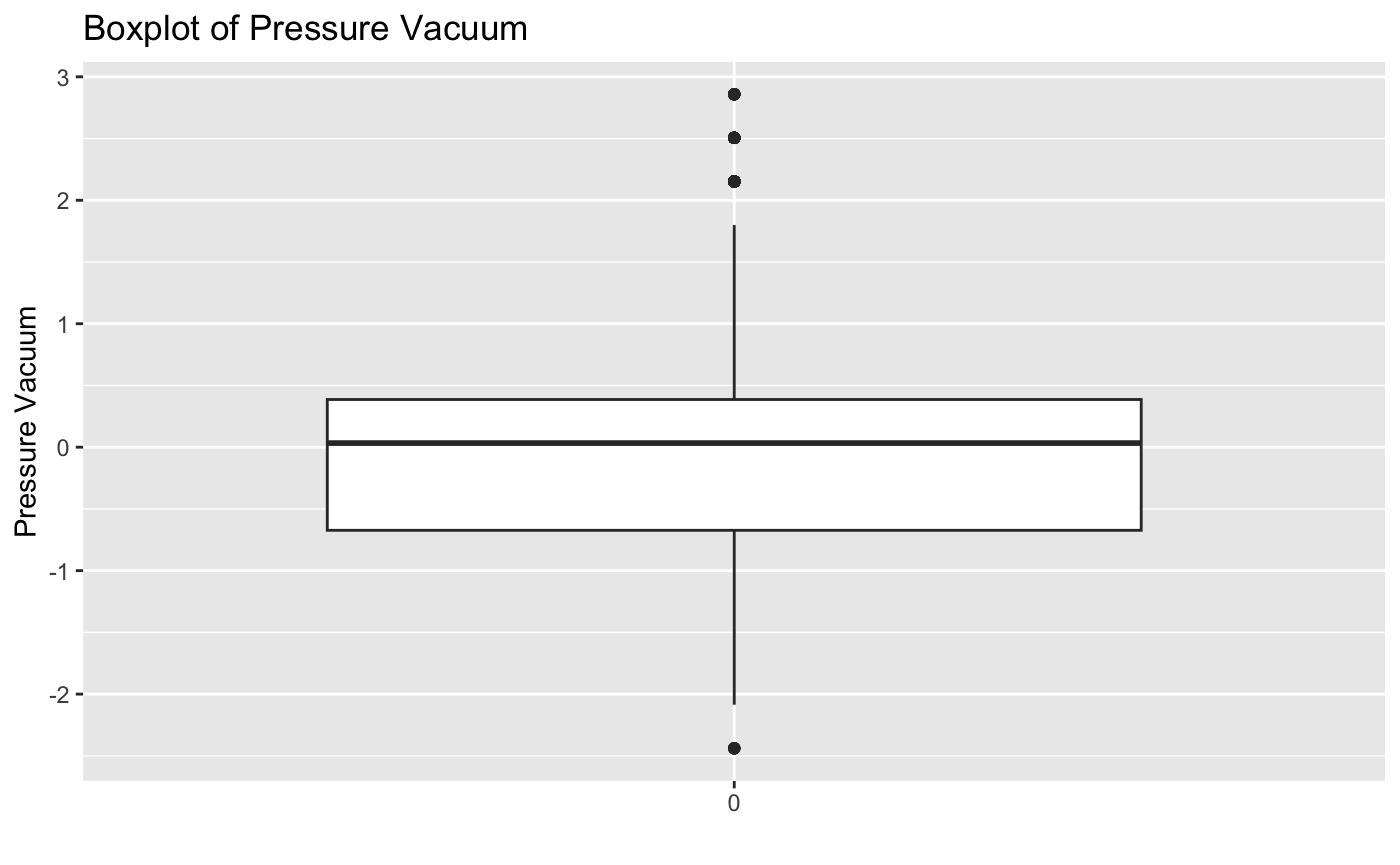
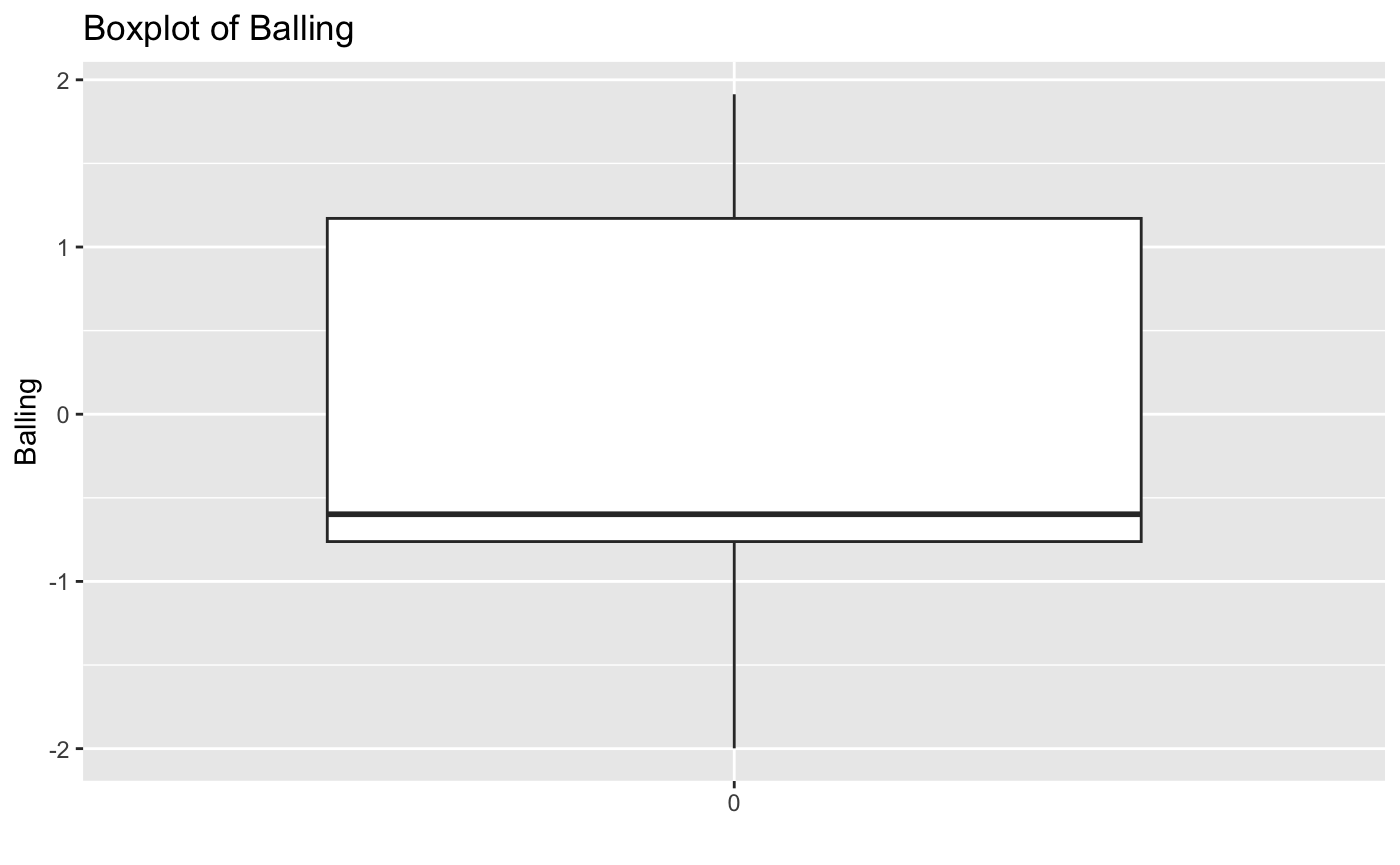
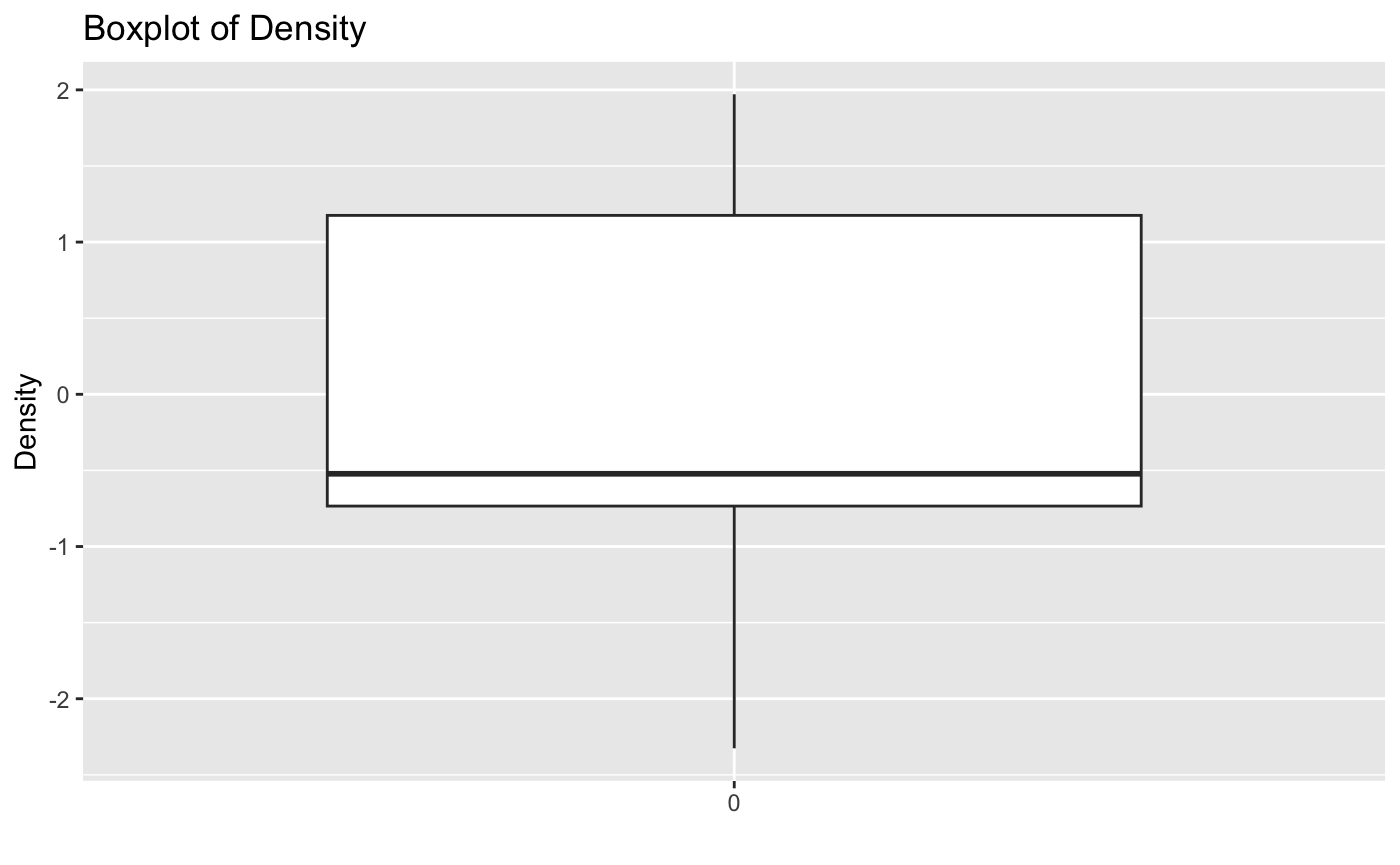
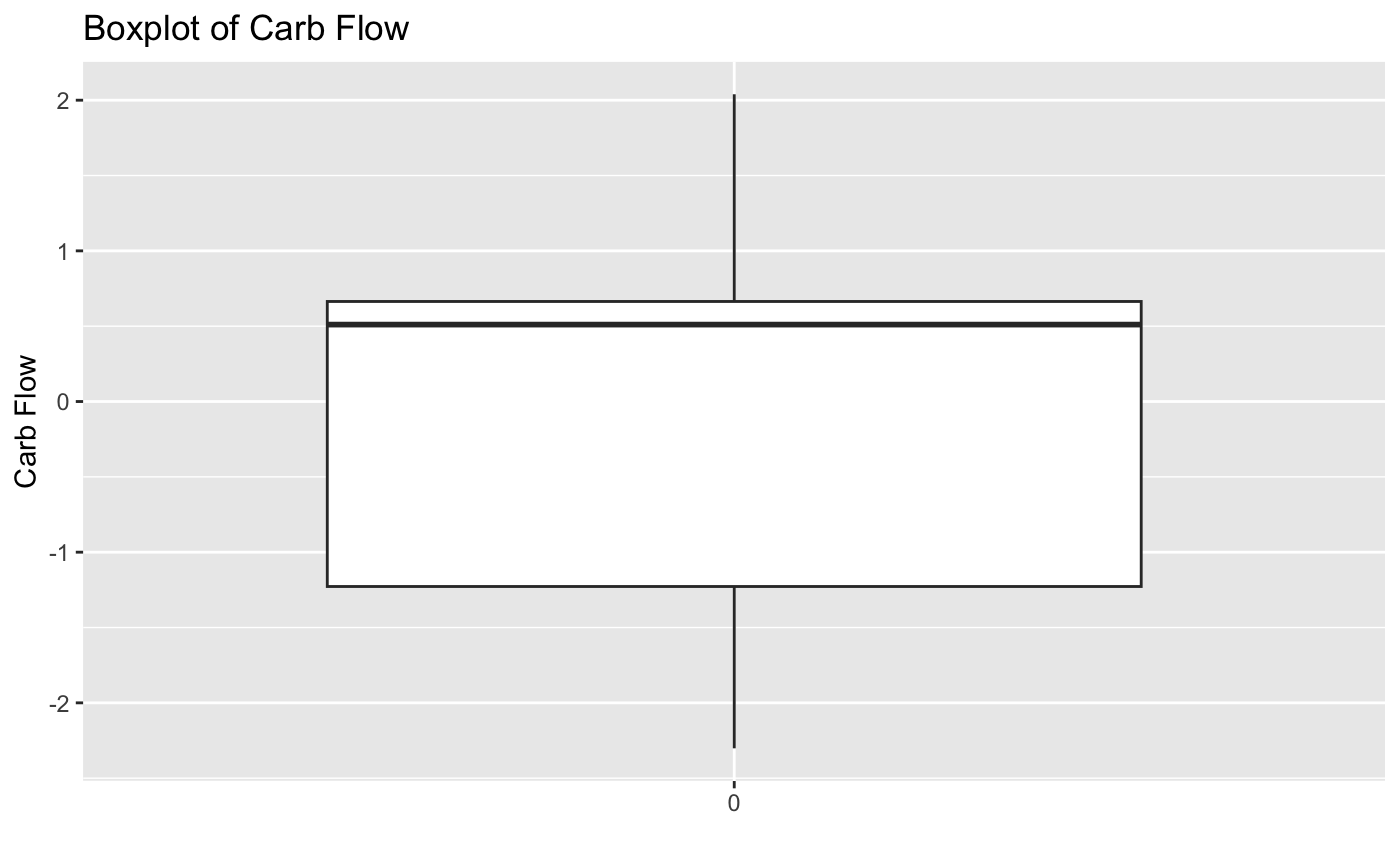
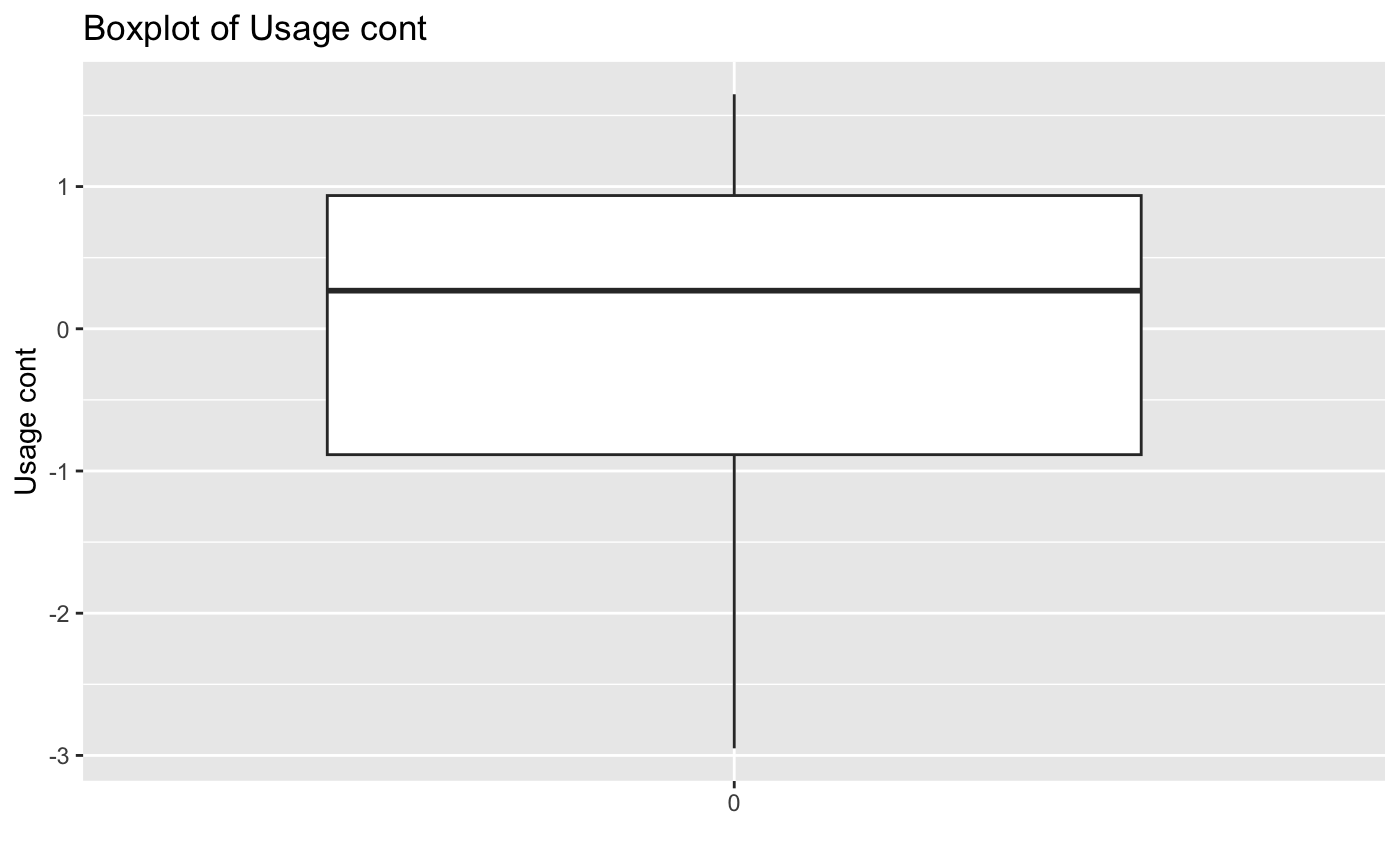
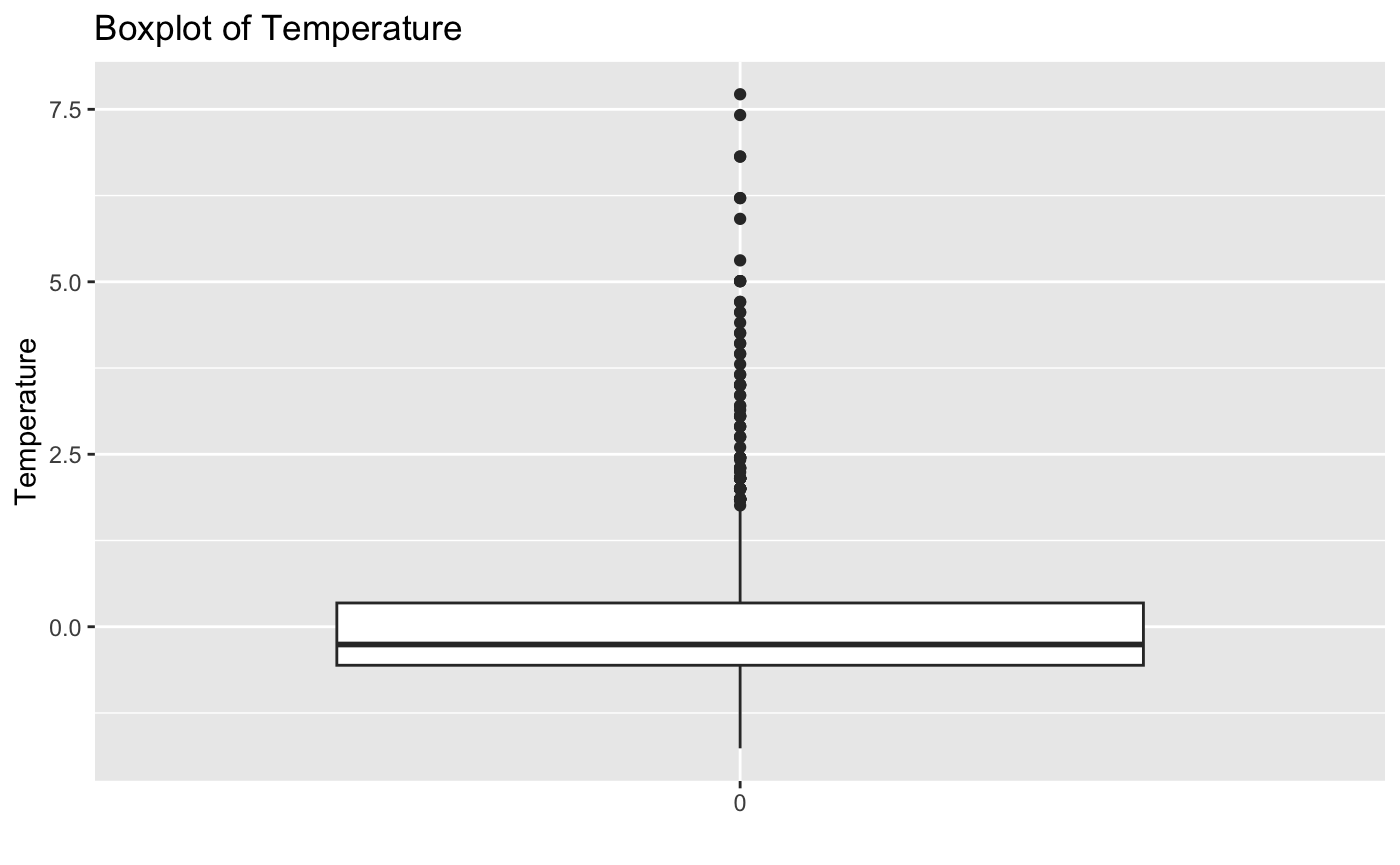
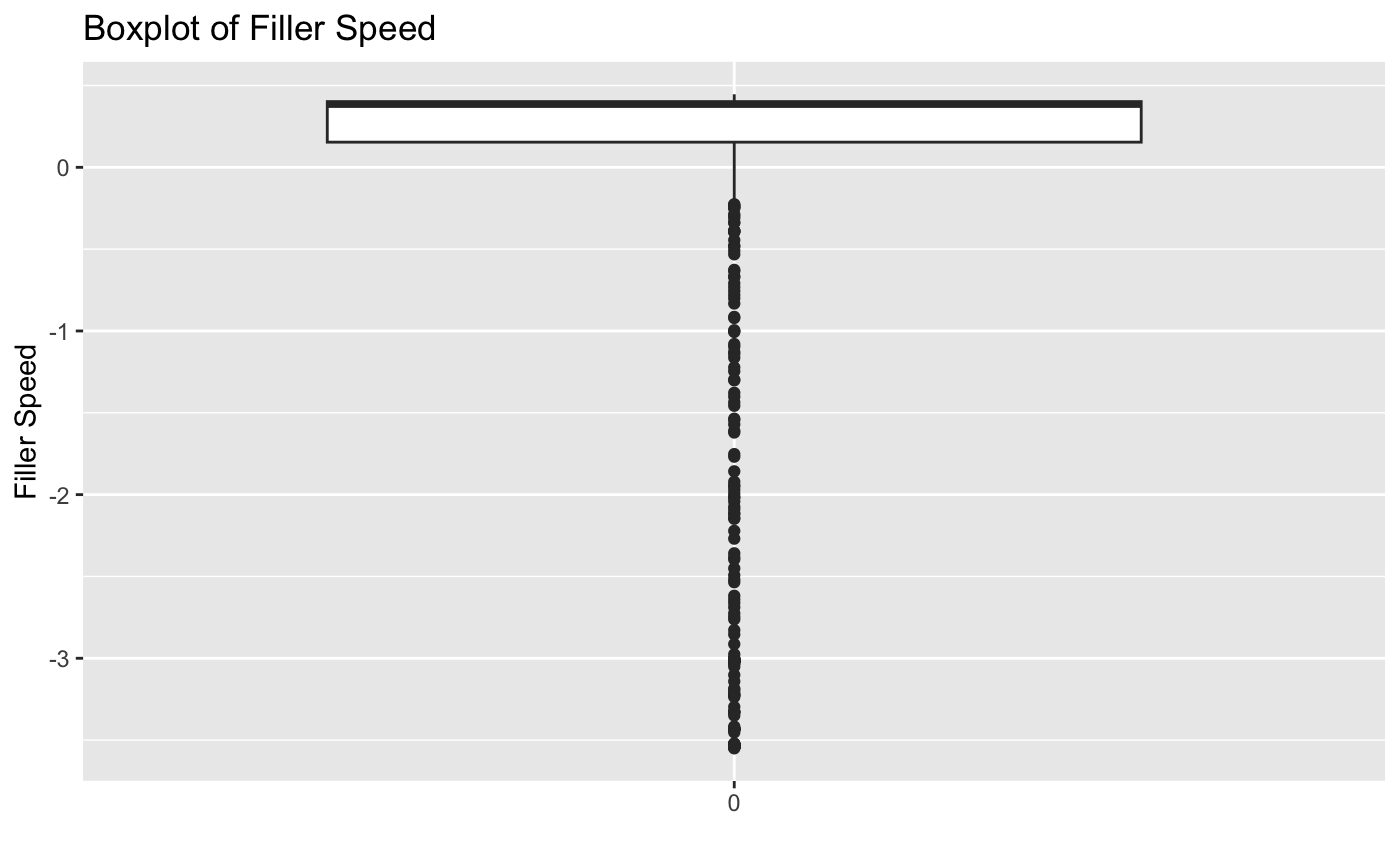
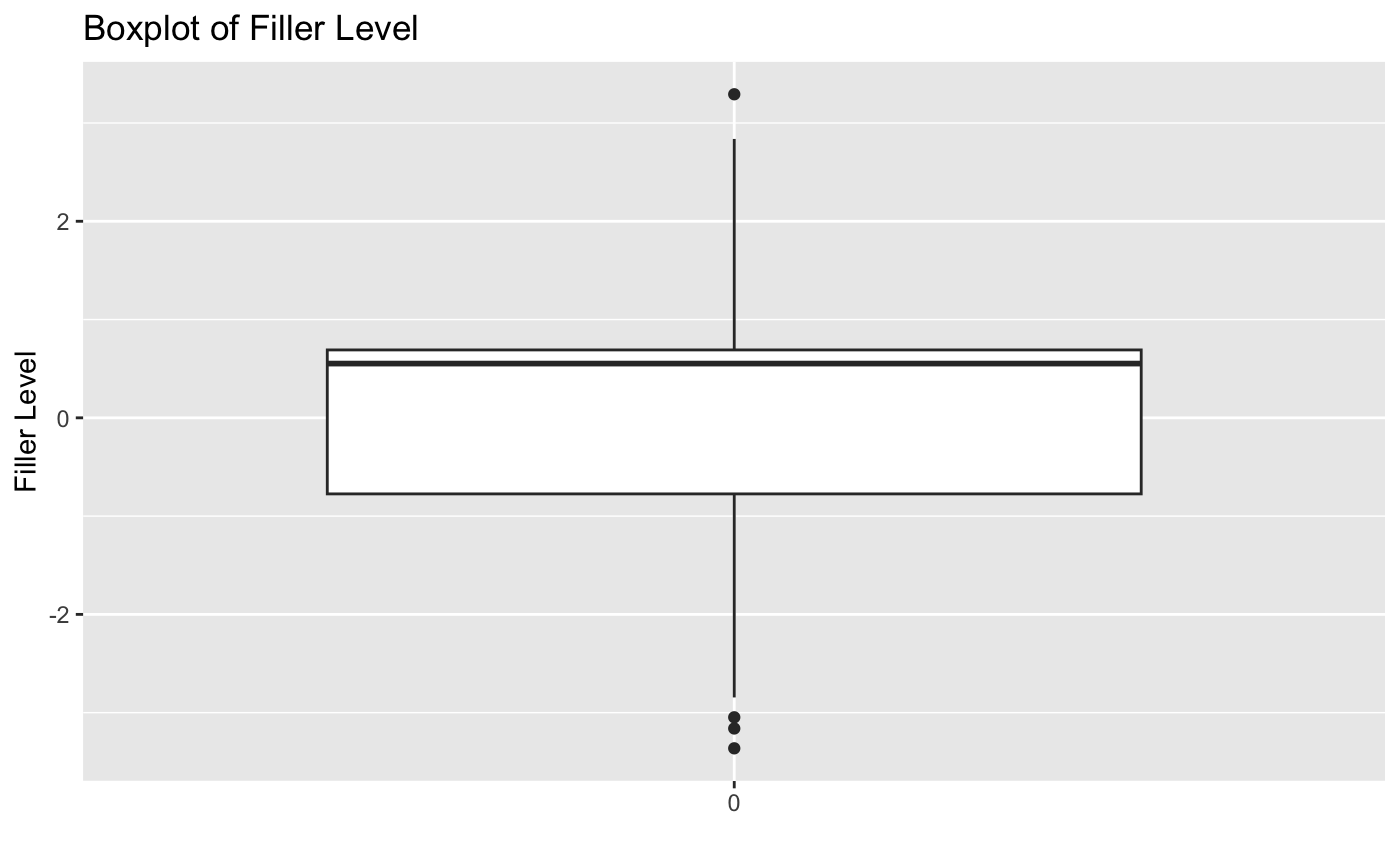
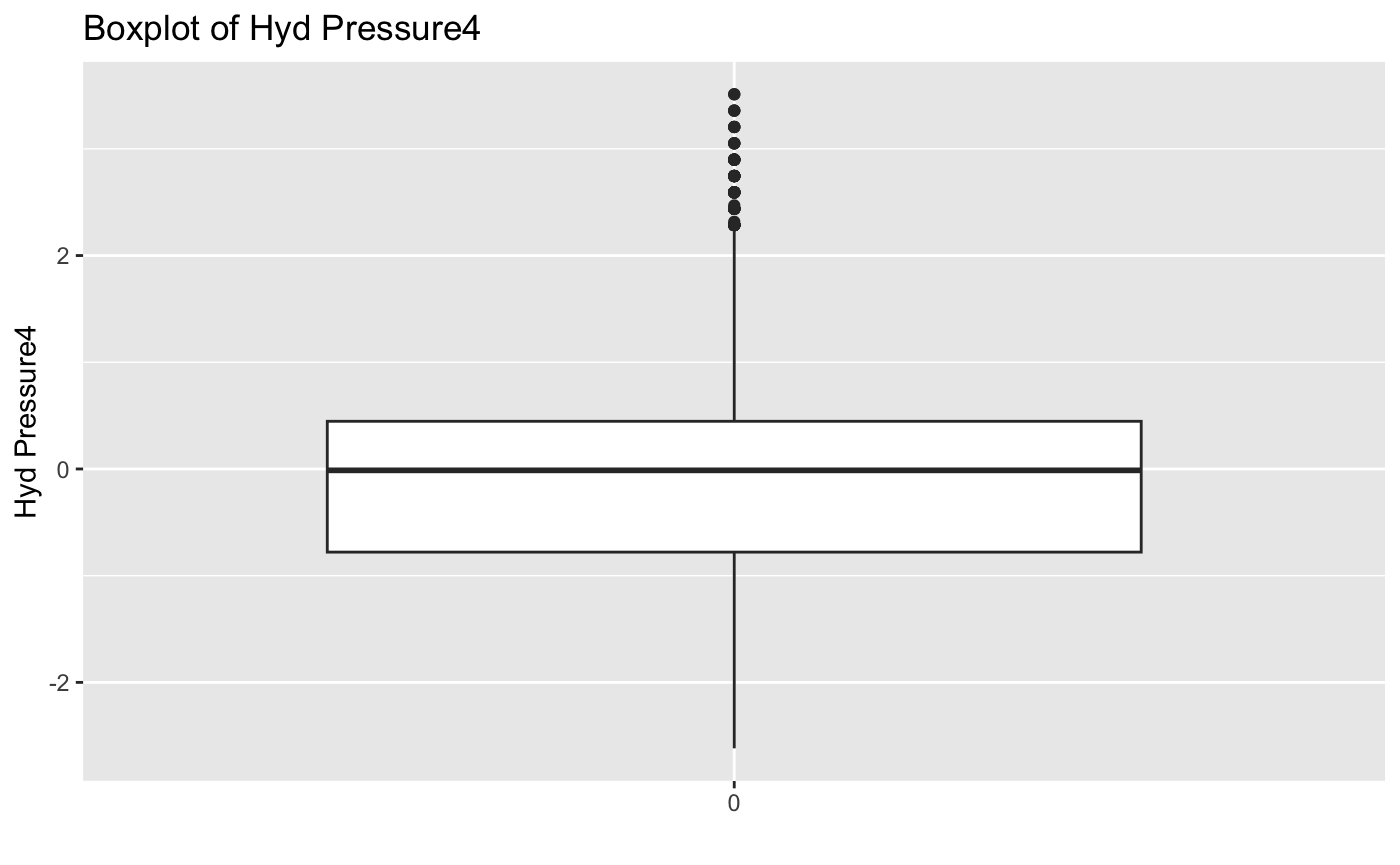
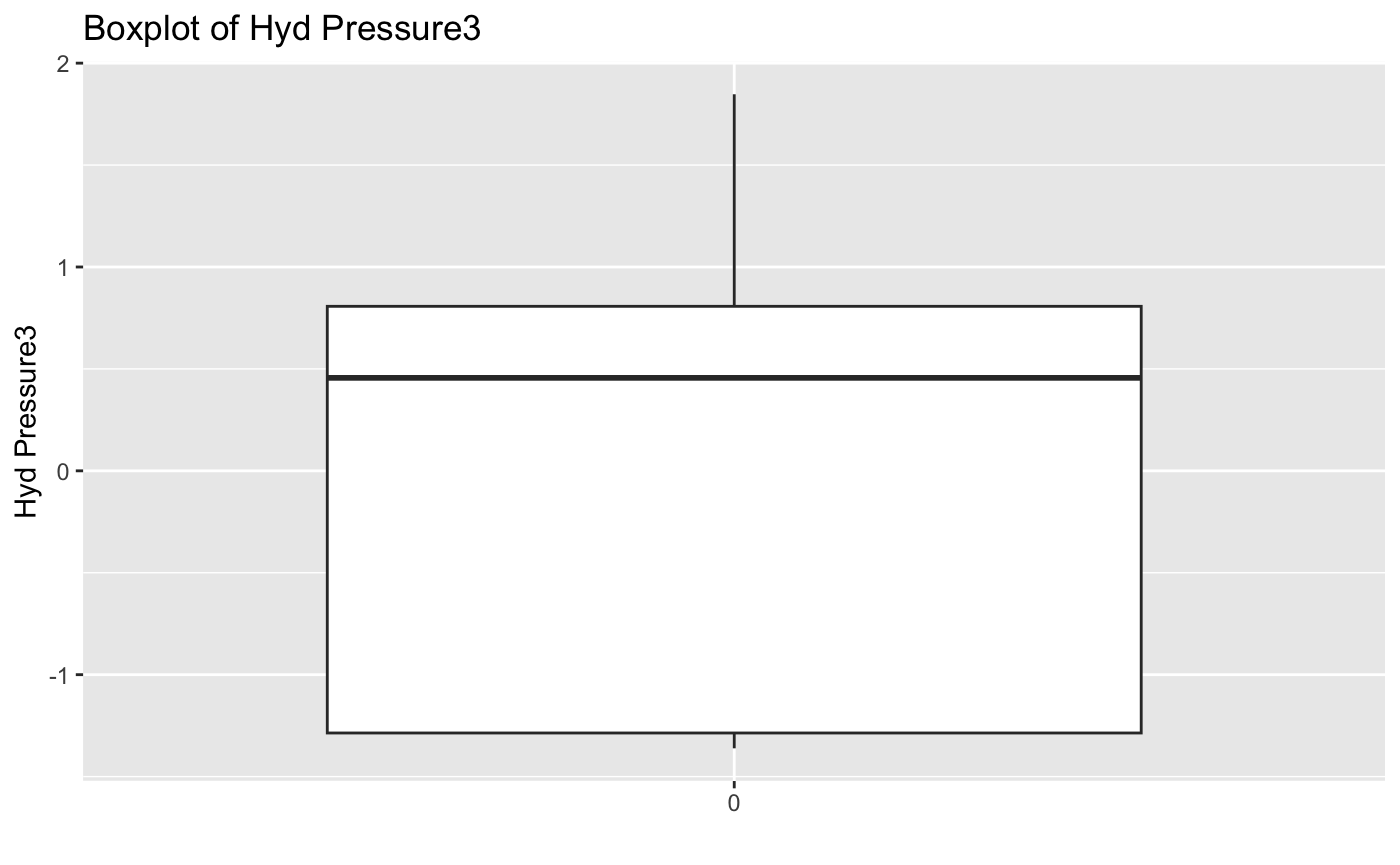
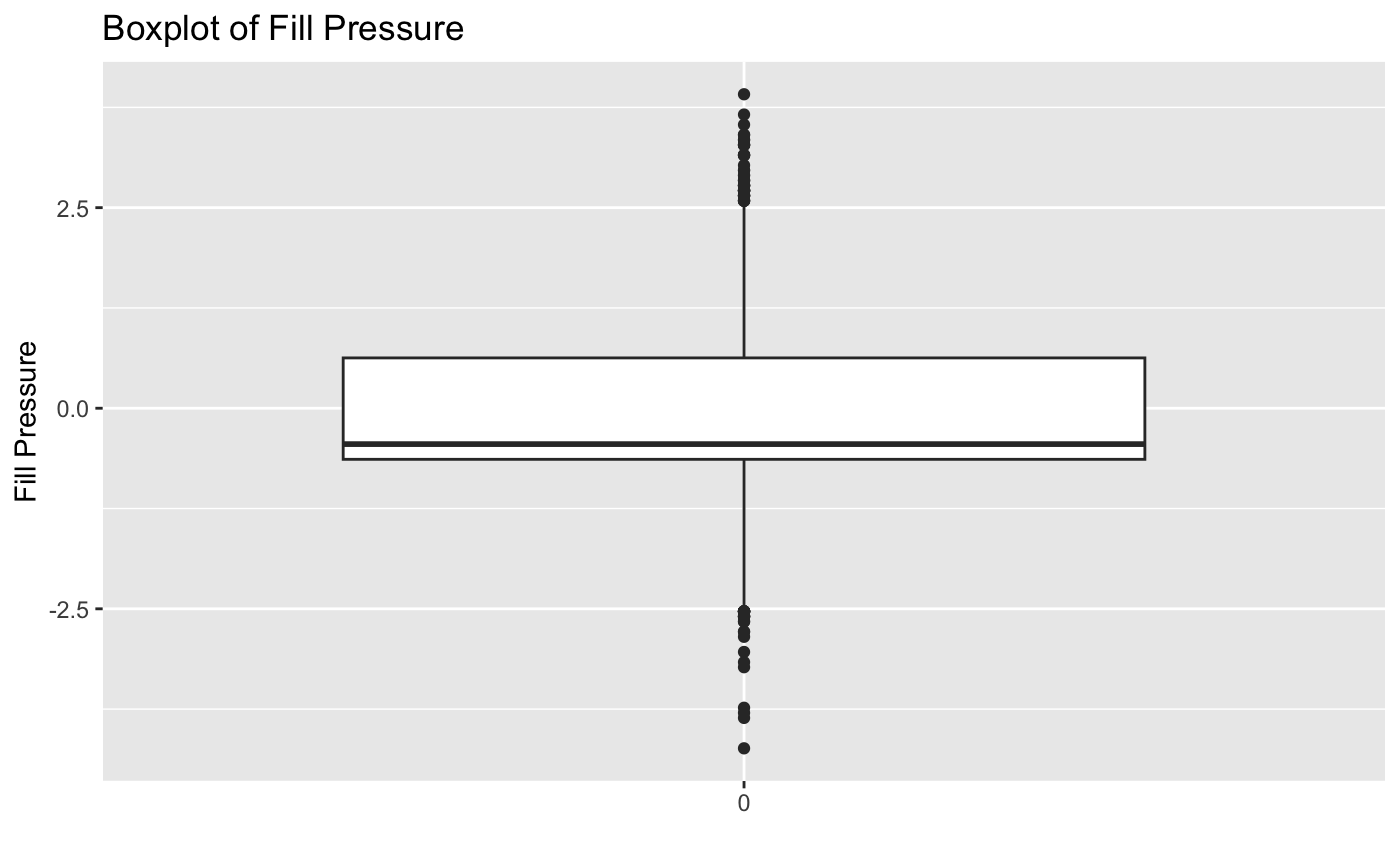
ylab(col)

print(p)

}

```





#### 

#### Interpretation of Box Plots

Upon visualizing the box plots for each numeric variable, we can see several outliers. It is extremely difficult to know how to work with these outliers because we are simply not familiar with the nuances of these variables. One thing that is especially concerning, though, is the outliers for pH. The lower outliers seem legitimate, but the high outlier looks like it could be a data entry error.

To address this, we will remove the row with the high outlier for pH:

```{r}

# identify row with the high outlier

max\_ph\_index = which.max(train\_split\_imputed$PH)

# Remove the row

train\_split\_imputed = train\_split\_imputed[-max\_ph\_index, ]

```

As for the other variables, again we lack context. But some of the boxplots (e.g., Fill Pressure) suggest that while some of the observations are technically outliers, their characterization as such might be a function of the fact that the data isn't normally distributed.

### Histograms

Let's take a look at the distributions:

```{r}

# histograms for each numeric variable

numeric\_cols = colnames(train\_split\_imputed)[sapply(train\_split\_imputed, is.numeric)]

for (col in numeric\_cols) {

p = ggplot(train\_split\_imputed, aes(x = .data[[col]])) +

geom\_histogram(fill = "blue") +

ggtitle(paste("Distribution of", col)) +

xlab(col) +

ylab("Frequency")

print(p)

}

```

#### 

#### 

#### 

#### Interpretation of Histograms

Upon examining the histograms, it is evident that very few of the variables are normally distributed. Because we're already reluctant to do anything drastic with the other outliers (outside of PH), we can just try to make the distributions more normal with transformations.

Next, we will consider appropriate transformations to handle these non-normal distributions.

### 

### Transformations

We already saw that some of the distributions are not normal. But there are a few different types of issues. Let's take Pressure Point: there are clear groupings. As such, we'll convert it to a categorical variable. We will do the same for "Alch Rel" and "Bowl Setpoint."

```{r}

# duplicating dataframes

train\_transformed = train\_split\_imputed %>%

mutate(

Pressure\_Setpoint\_Trans = `Pressure Setpoint`,

Bowl\_Setpoint\_Trans = `Bowl Setpoint`,

Alch\_Rel\_Trans = `Alch Rel`

)

eval\_transformed = eval\_split\_imputed %>%

mutate(

Pressure\_Setpoint\_Trans = `Pressure Setpoint`,

Bowl\_Setpoint\_Trans = `Bowl Setpoint`,

Alch\_Rel\_Trans = `Alch Rel`

)

test\_transformed = test\_imputed %>%

mutate(

Pressure\_Setpoint\_Trans = `Pressure Setpoint`,

Bowl\_Setpoint\_Trans = `Bowl Setpoint`,

Alch\_Rel\_Trans = `Alch Rel`

)

```

Next, we will categorize the transformed setpoints and alcohol release variables:

```{r}

# Function to categorize Setpoints (based on histogram above)

categorize\_setpoint = function(data, column\_name) {

data[[column\_name]] = cut(data[[column\_name]],

breaks = c(-Inf, -1, 0, 1, 2, Inf),

labels = c("< -1", "-1 to 0", "0 to 1", "1 to 2", "> 2"))

return(data)

}

# Function to categorize Alch Rel (based on histogram above)

categorize\_alch\_rel = function(data, column\_name) {

data[[column\_name]] = cut(data[[column\_name]],

breaks = c(-Inf, -1, 0, 1, 2, Inf),

labels = c("< -1", "-1 to 0", "0 to 1", "1 to 2", "> 2"))

return(data)

}

# Applying the categorization functions to new dataframes

train\_transformed = categorize\_setpoint(train\_transformed, "Pressure\_Setpoint\_Trans")

train\_transformed = categorize\_setpoint(train\_transformed, "Bowl\_Setpoint\_Trans")

train\_transformed = categorize\_alch\_rel(train\_transformed, "Alch\_Rel\_Trans")

eval\_transformed = categorize\_setpoint(eval\_transformed, "Pressure\_Setpoint\_Trans")

eval\_transformed = categorize\_setpoint(eval\_transformed, "Bowl\_Setpoint\_Trans")

eval\_transformed = categorize\_alch\_rel(eval\_transformed, "Alch\_Rel\_Trans")

test\_transformed = categorize\_setpoint(test\_transformed, "Pressure\_Setpoint\_Trans")

test\_transformed = categorize\_setpoint(test\_transformed, "Bowl\_Setpoint\_Trans")

test\_transformed = categorize\_alch\_rel(test\_transformed, "Alch\_Rel\_Trans")

```

Now, we will remove the original variables since we have the categorical versions:

```{r}

# Now removing the original variables since we have the categorical:

train\_transformed = train\_transformed %>%

dplyr::select(-`Pressure Setpoint`, -`Bowl Setpoint`, -`Alch Rel`)

eval\_transformed = eval\_transformed %>%

dplyr::select(-`Pressure Setpoint`, -`Bowl Setpoint`, -`Alch Rel`)

test\_transformed = test\_transformed %>%

dplyr::select(-`Pressure Setpoint`, -`Bowl Setpoint`, -`Alch Rel`)

```

Next, let's find the best transformations for the remaining numeric variables:

```{r}

library(bestNormalize)

# Numeric columns excluding PH

numeric\_cols = colnames(train\_transformed)[sapply(train\_transformed, is.numeric)]

numeric\_cols = setdiff(numeric\_cols, "PH")

print(numeric\_cols)

```

[1] "Carb Volume" "Fill Ounces" "Carb Pressure" "Carb Temp" "PSC Fill" "Carb Pressure1"

[7] "Fill Pressure" "Hyd Pressure2" "Hyd Pressure3" "Hyd Pressure4" "Filler Level" "Filler Speed"

[13] "Temperature" "Usage cont" "Carb Flow" "Density" "Balling" "Pressure Vacuum"

[19] "Air Pressurer" "Carb Rel" "Balling Lvl"

```{r}

# NOTE: reusing code from my DATA 622 work as reference

# Lists storing transformation objects

transformations = list()

# Apply the best normalization method to each numeric column in the training set

for (col in numeric\_cols) {

bn\_object = bestNormalize(train\_transformed[[col]])

train\_transformed[[col]] = predict(bn\_object, train\_transformed[[col]])

transformations[[col]] = bn\_object

}

```

We can also assess the current skewness:

```{r}

library(e1071)

skewness\_values = sapply(train\_transformed[numeric\_cols], skewness, na.rm = TRUE)

print(skewness\_values)

```

Carb Volume Fill Ounces Carb Pressure Carb Temp PSC Fill Carb Pressure1 Fill Pressure

9.277512e-05 -5.516583e-02 5.151163e-04 -9.012069e-03 9.276538e-02 -7.211951e-05 5.041984e-03

Hyd Pressure2 Hyd Pressure3 Hyd Pressure4 Filler Level Filler Speed Temperature Usage cont

4.749877e-01 2.454308e-01 1.726694e-02 -9.363297e-04 1.166212e-03 7.710042e-01 -1.743610e-03

Carb Flow Density Balling Pressure Vacuum Air Pressurer Carb Rel Balling Lvl

2.814228e-03 -2.267721e-04 4.482989e-03 -2.744623e-01 1.522034e+00 2.919010e-03 8.705519e-03

Finally, we can look at the shape of the new distributions:

```{r}

for (col in numeric\_cols) {

p = ggplot(train\_transformed, aes(x = .data[[col]])) +

geom\_histogram(fill = "blue") +

ggtitle(paste("Distribution of", col)) +

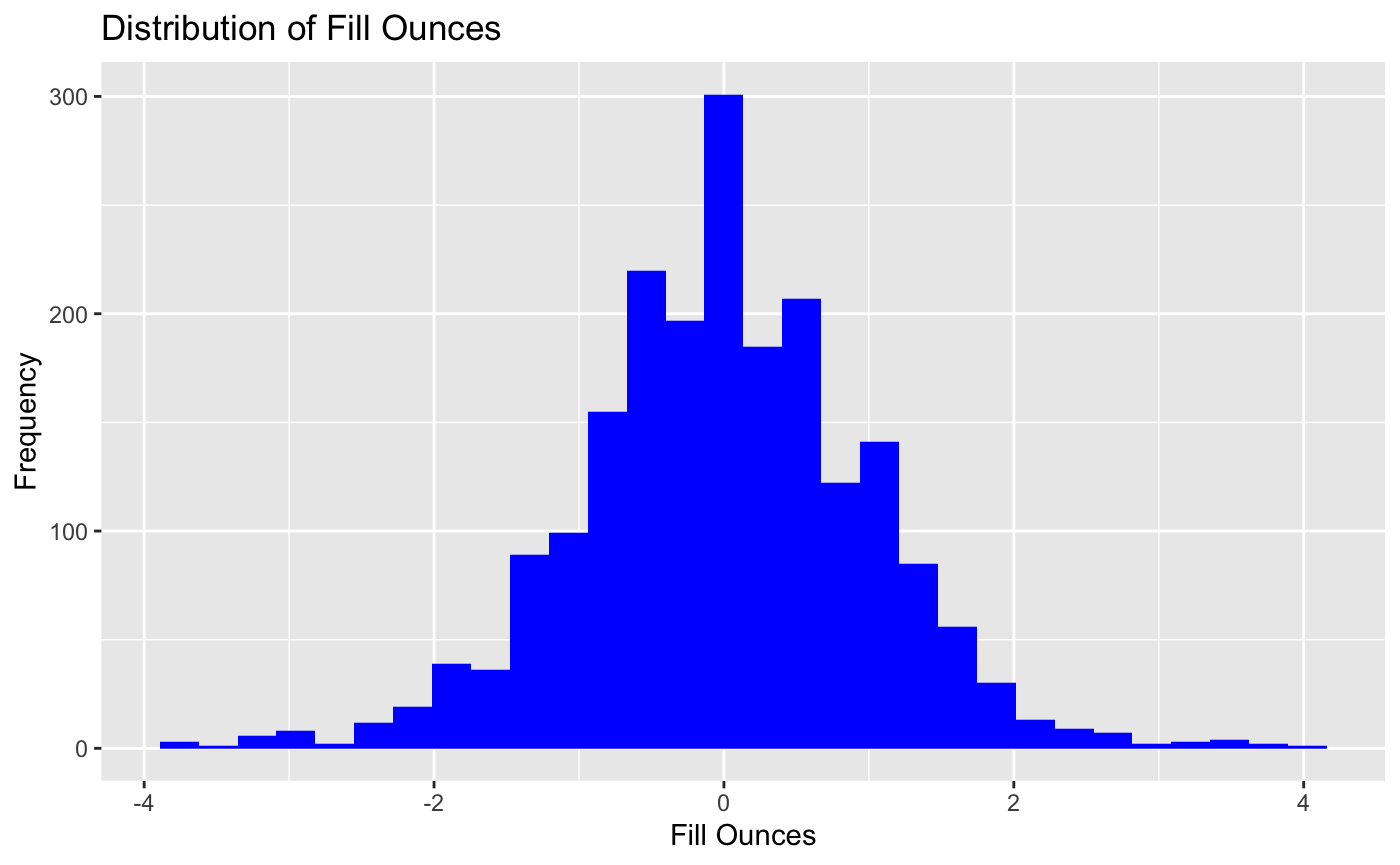
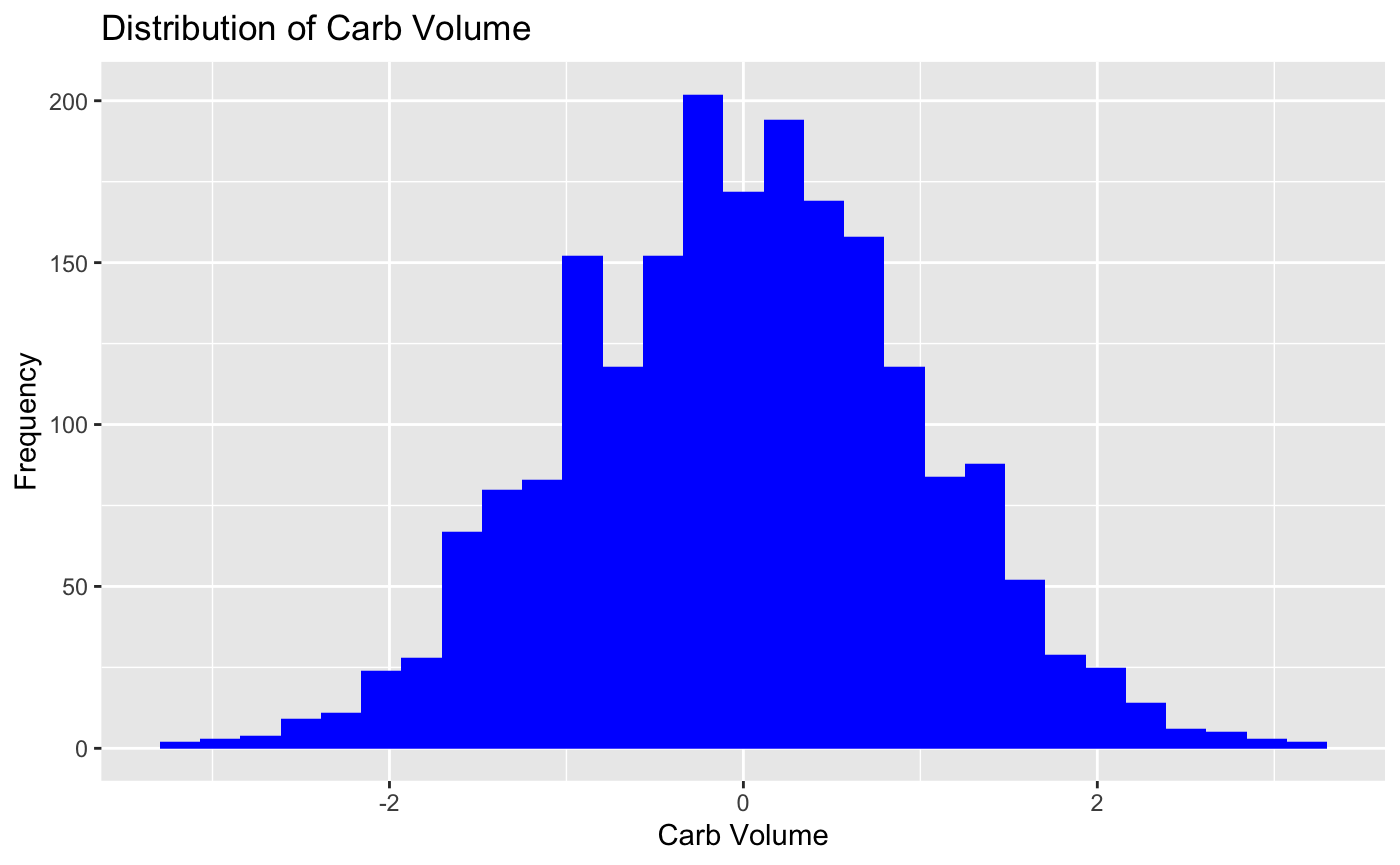
xlab(col) +

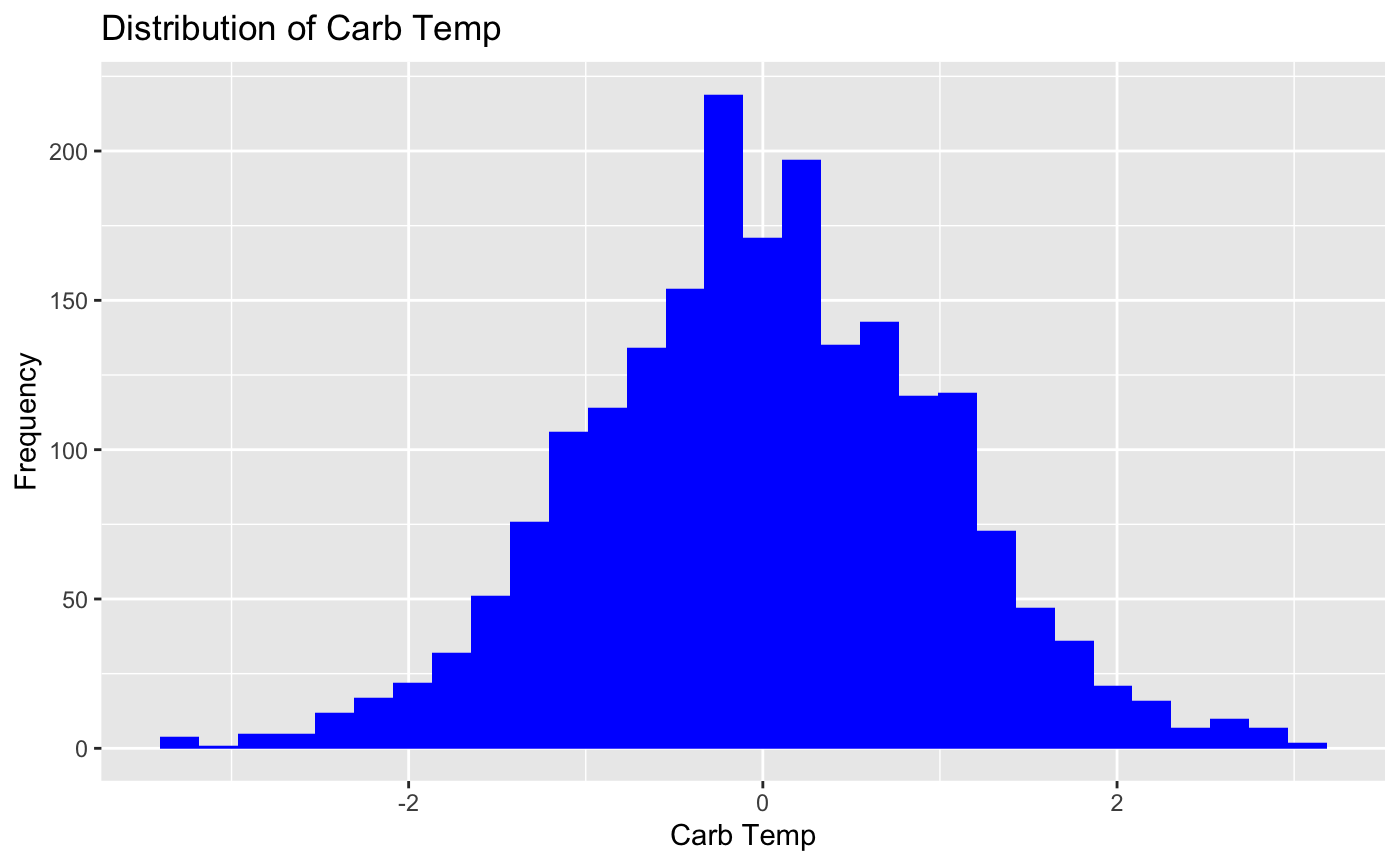
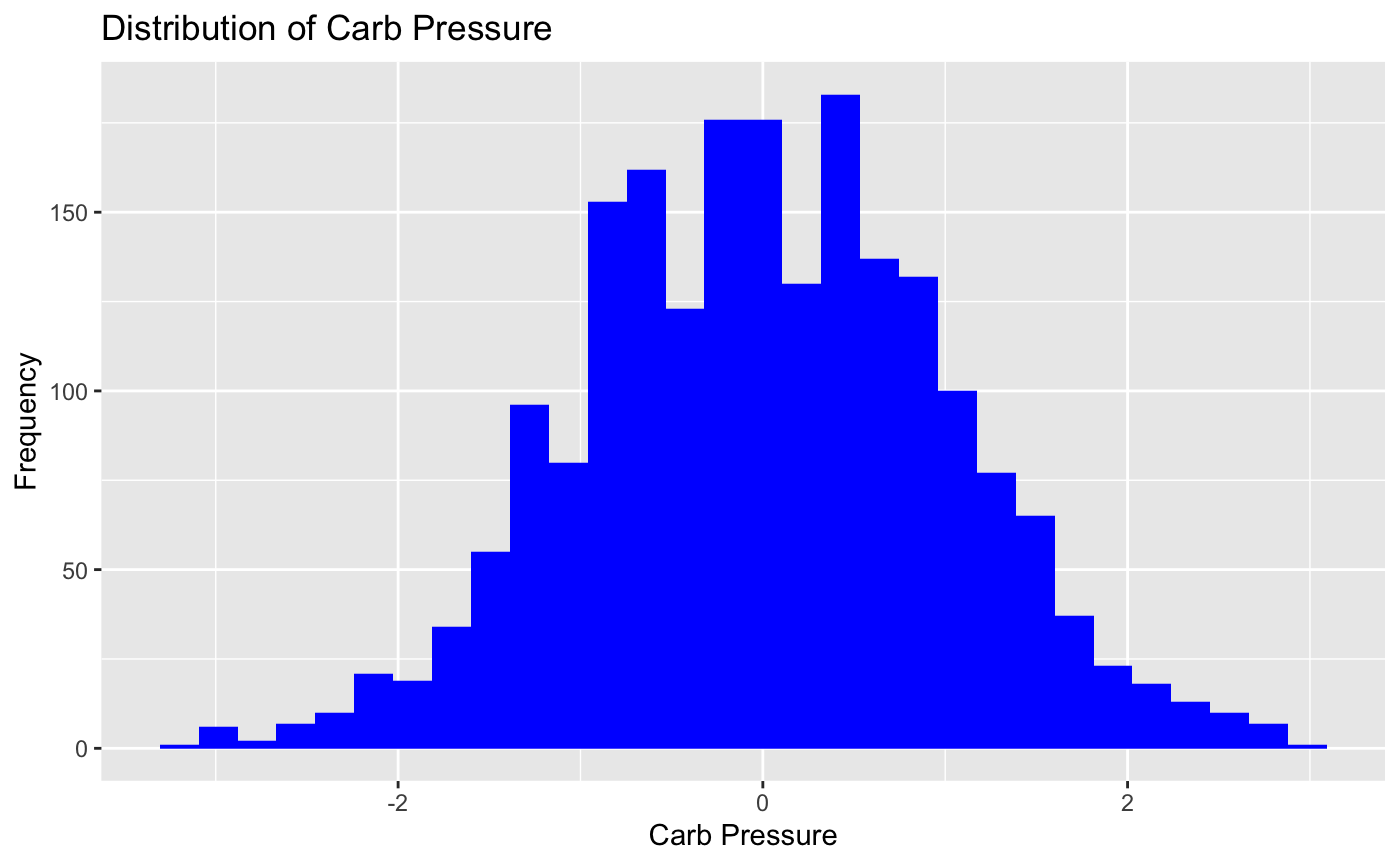
ylab("Frequency")

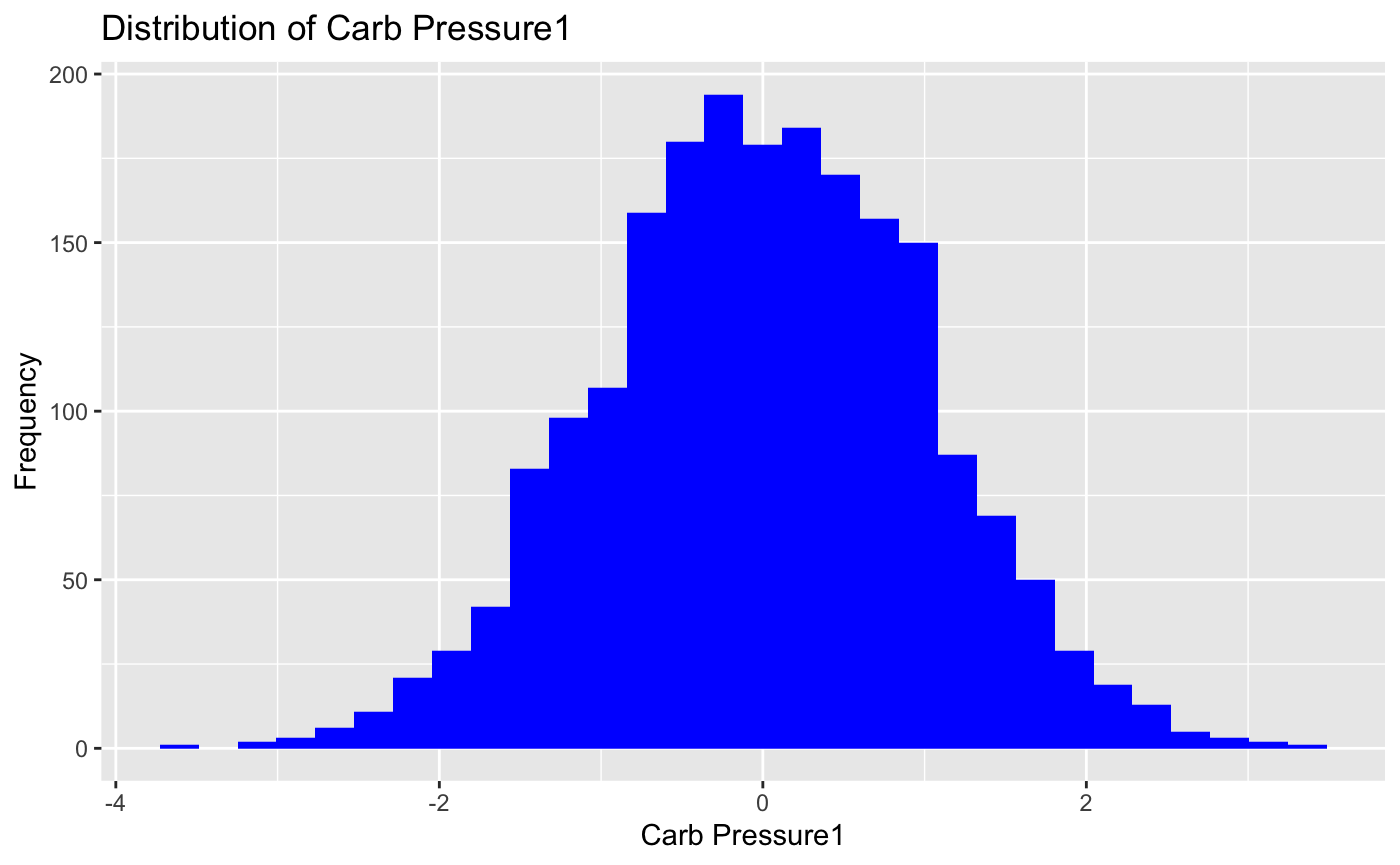
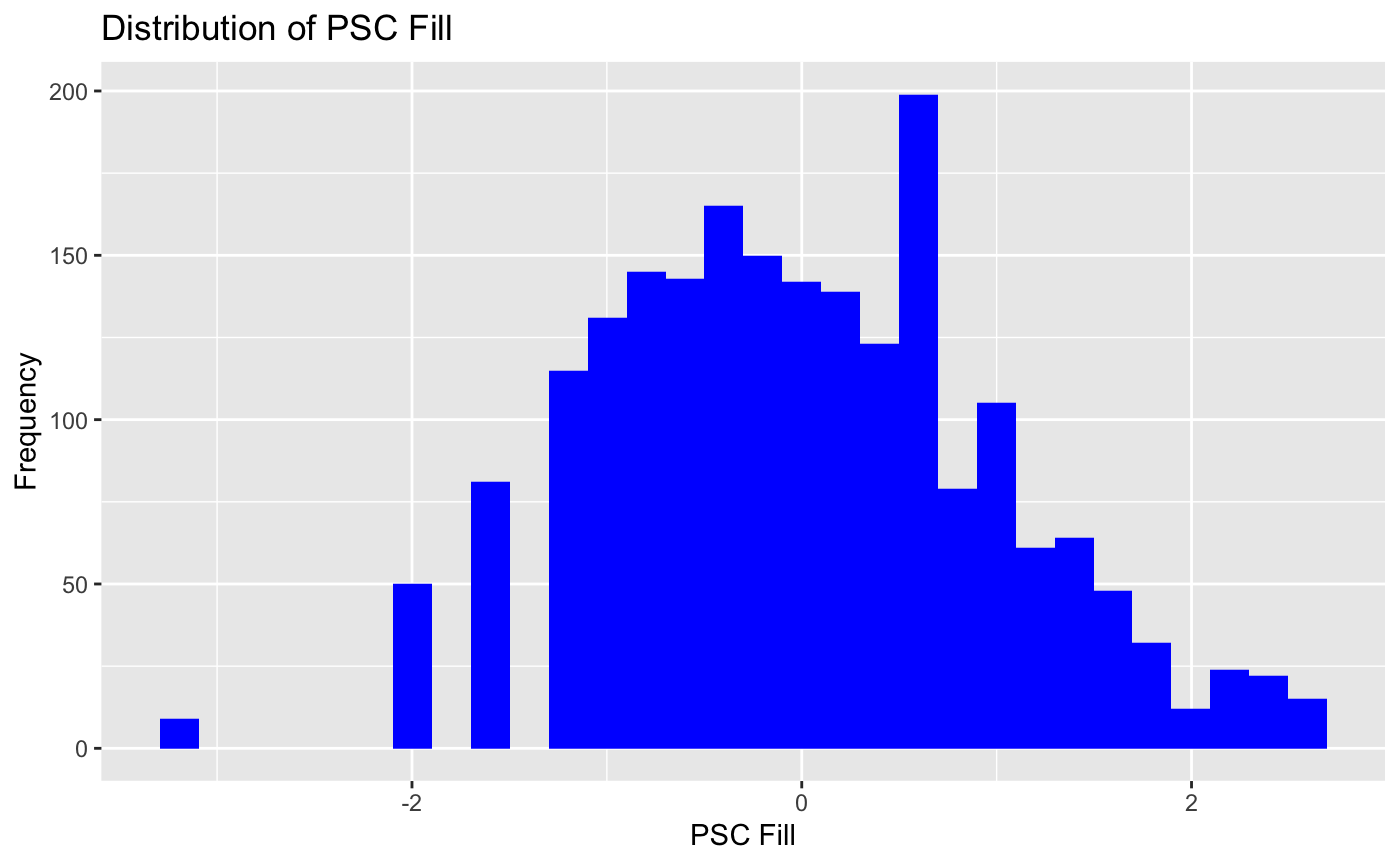
print(p)

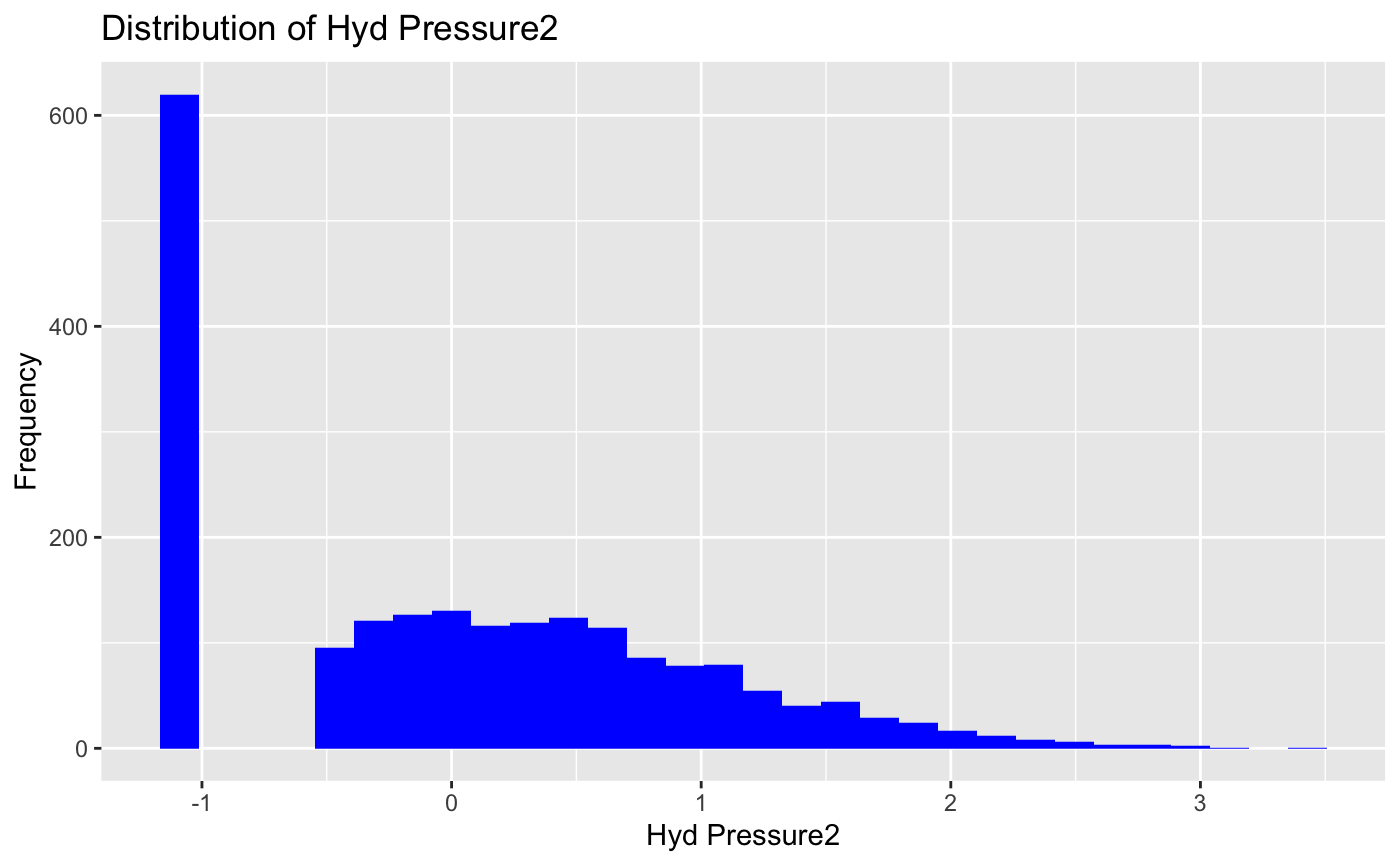
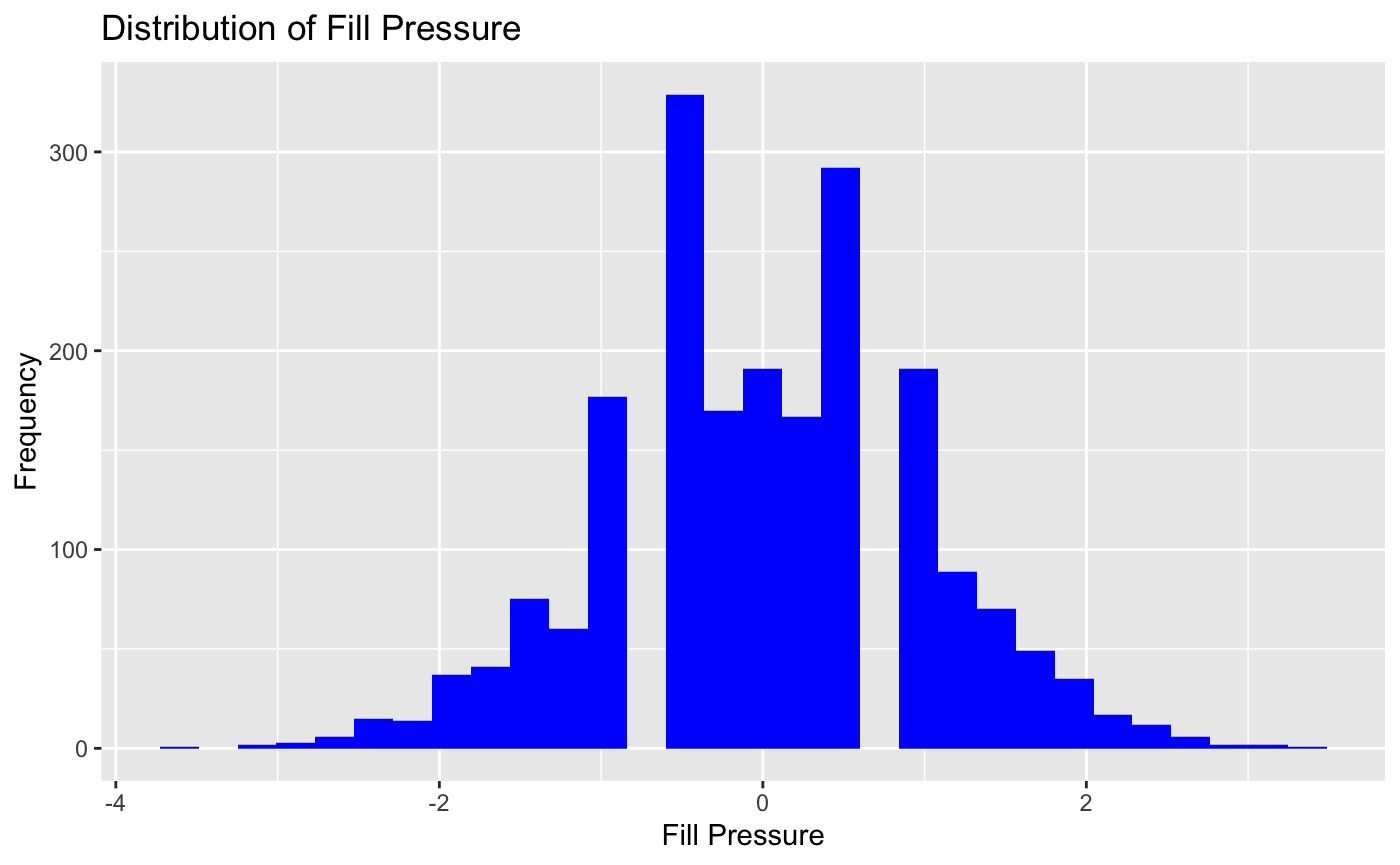
}

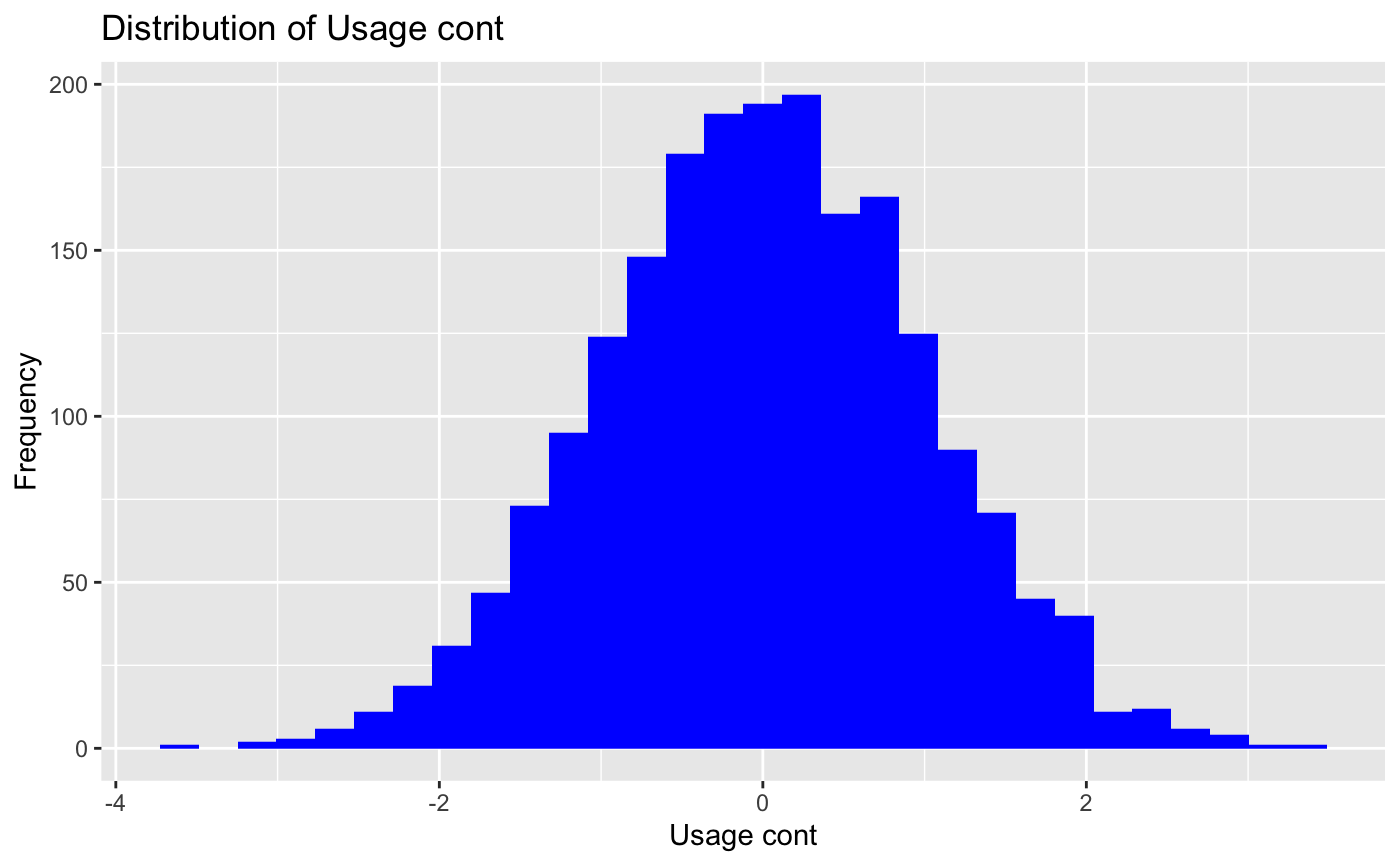
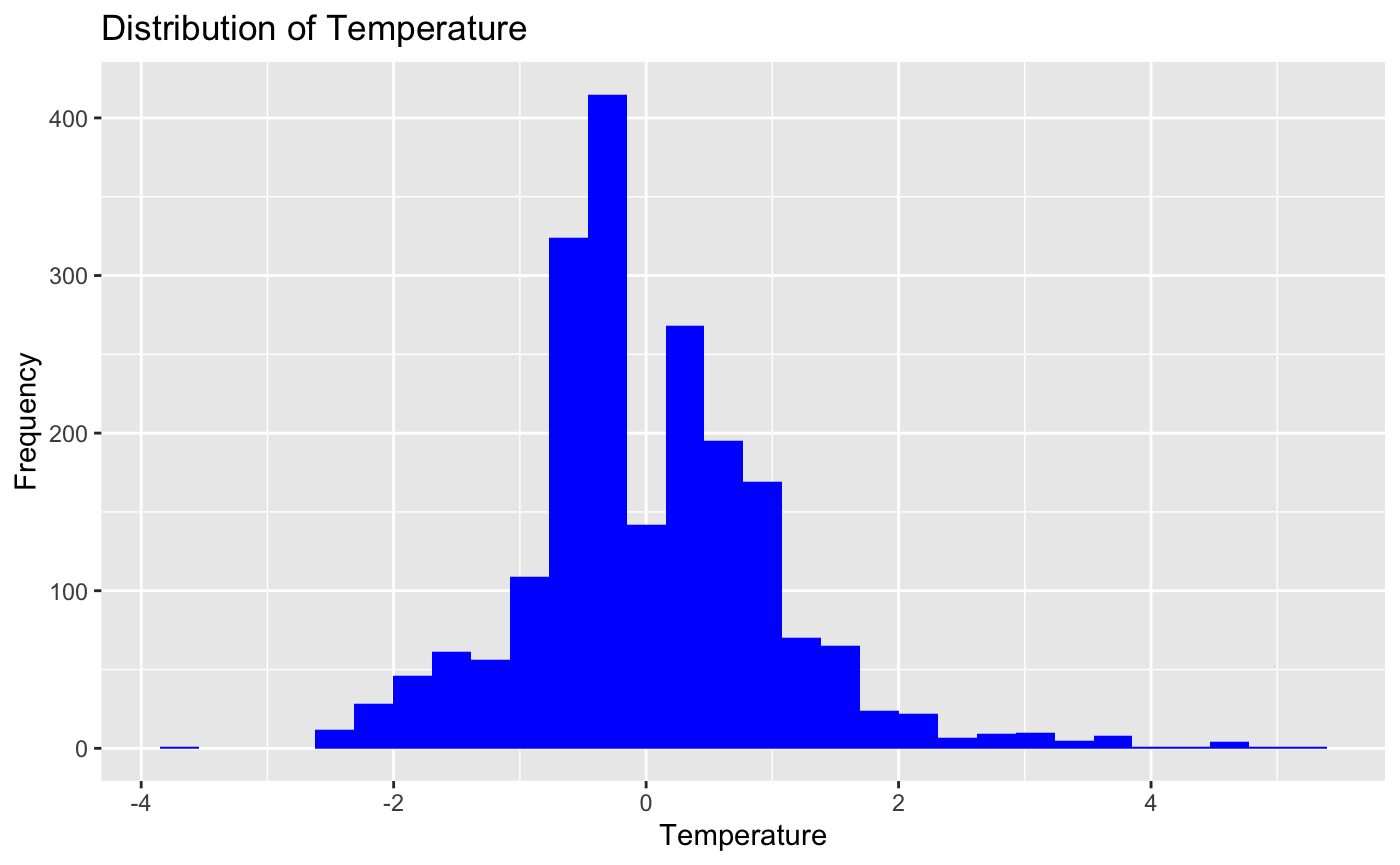
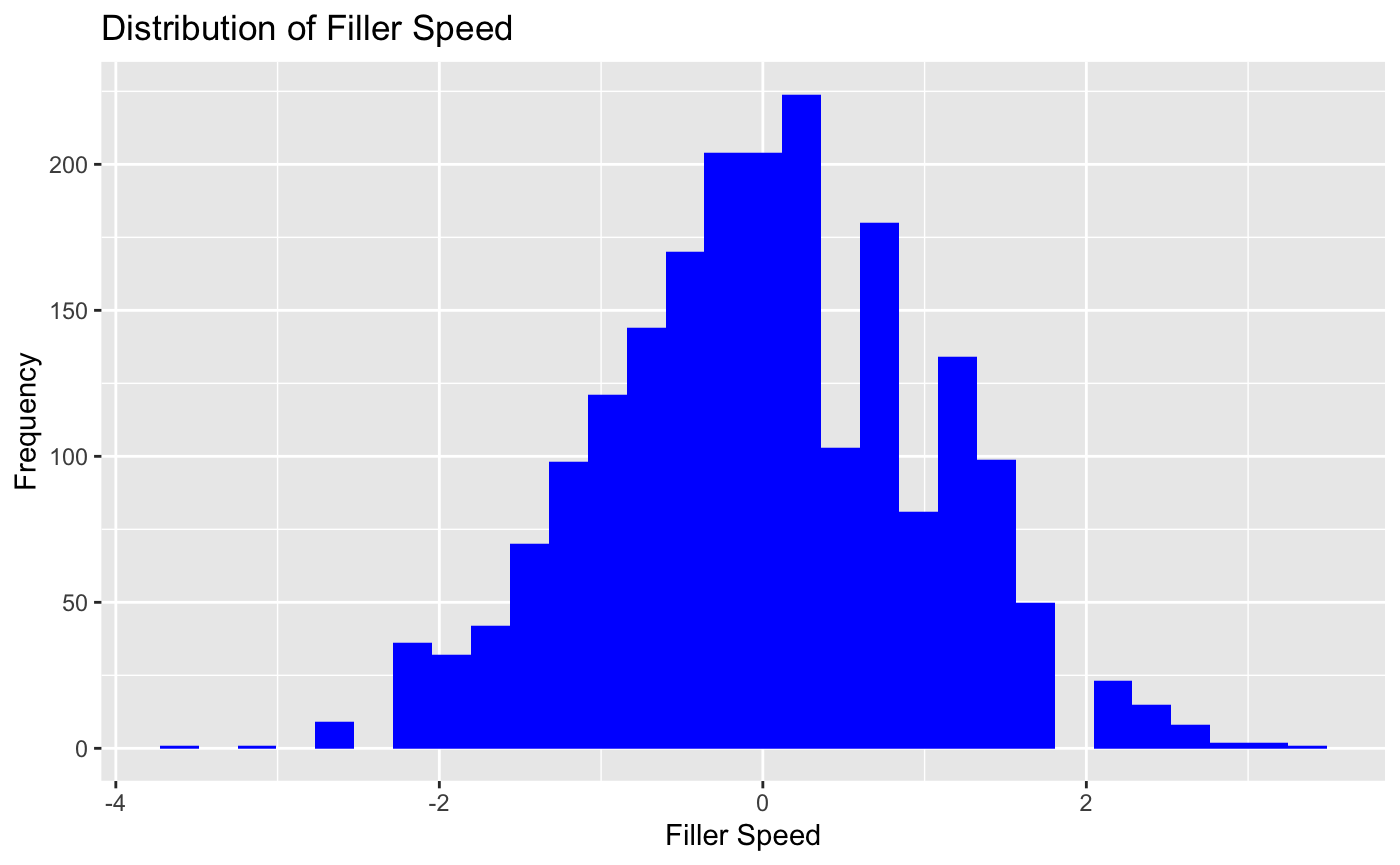
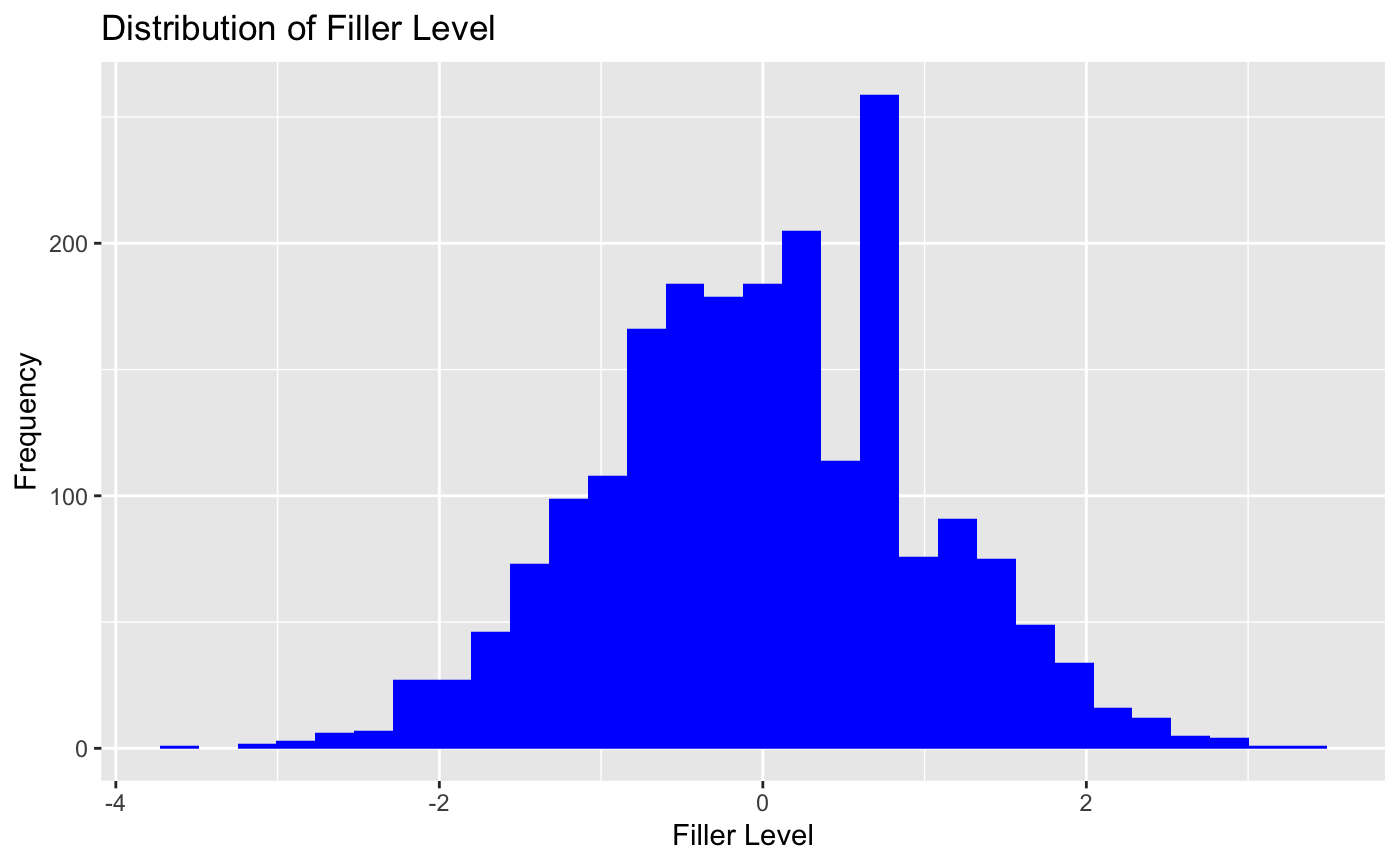
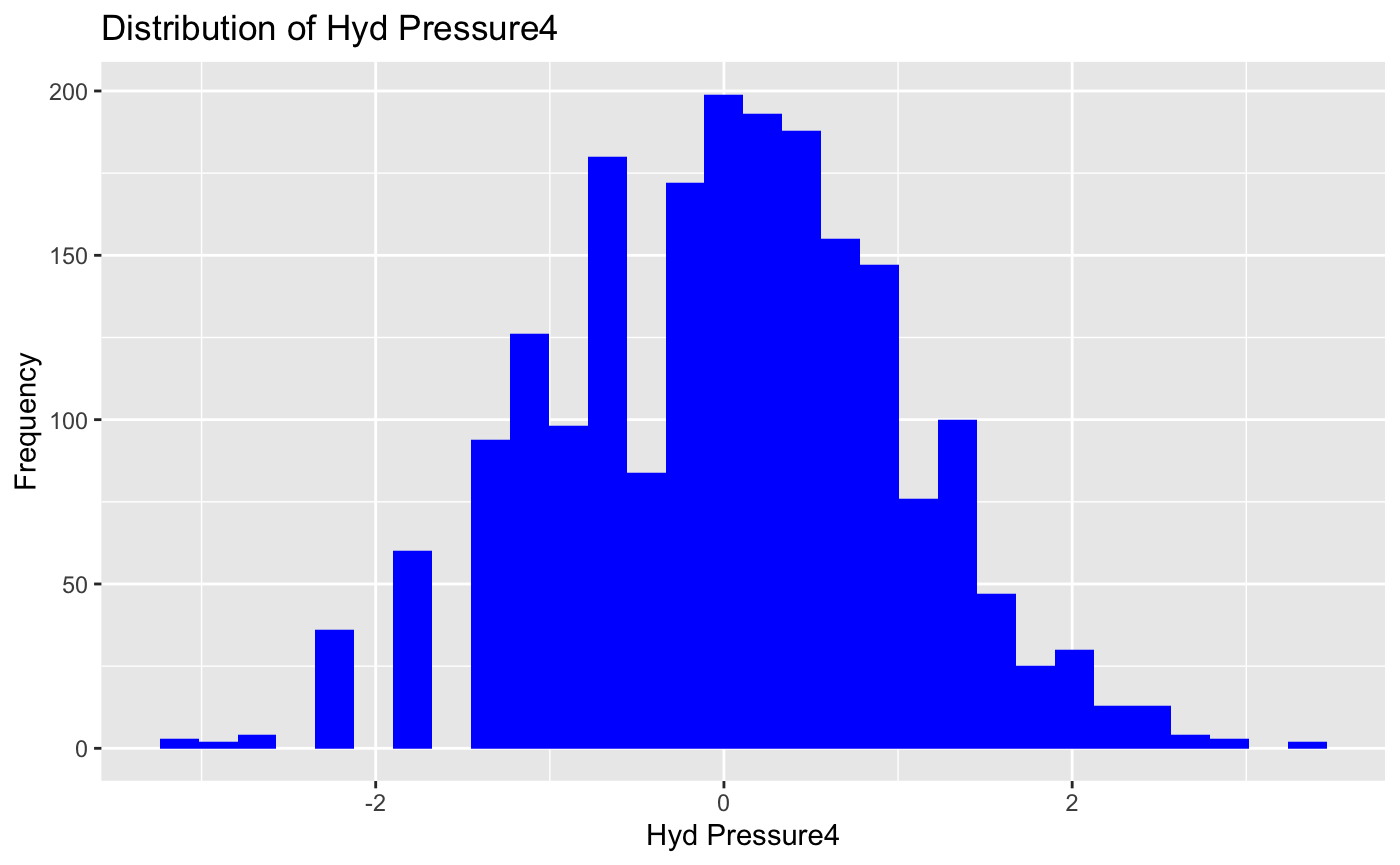
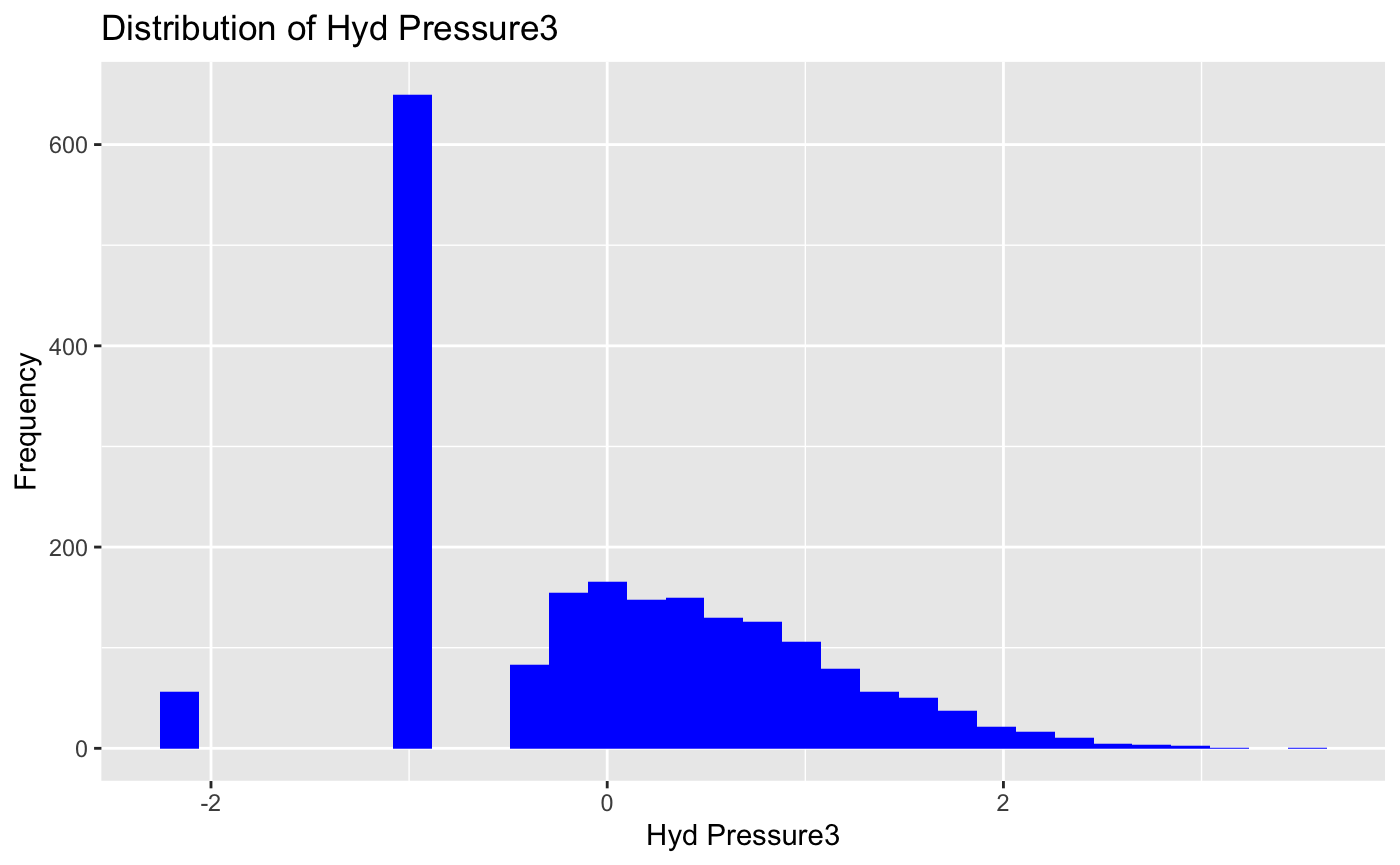
```

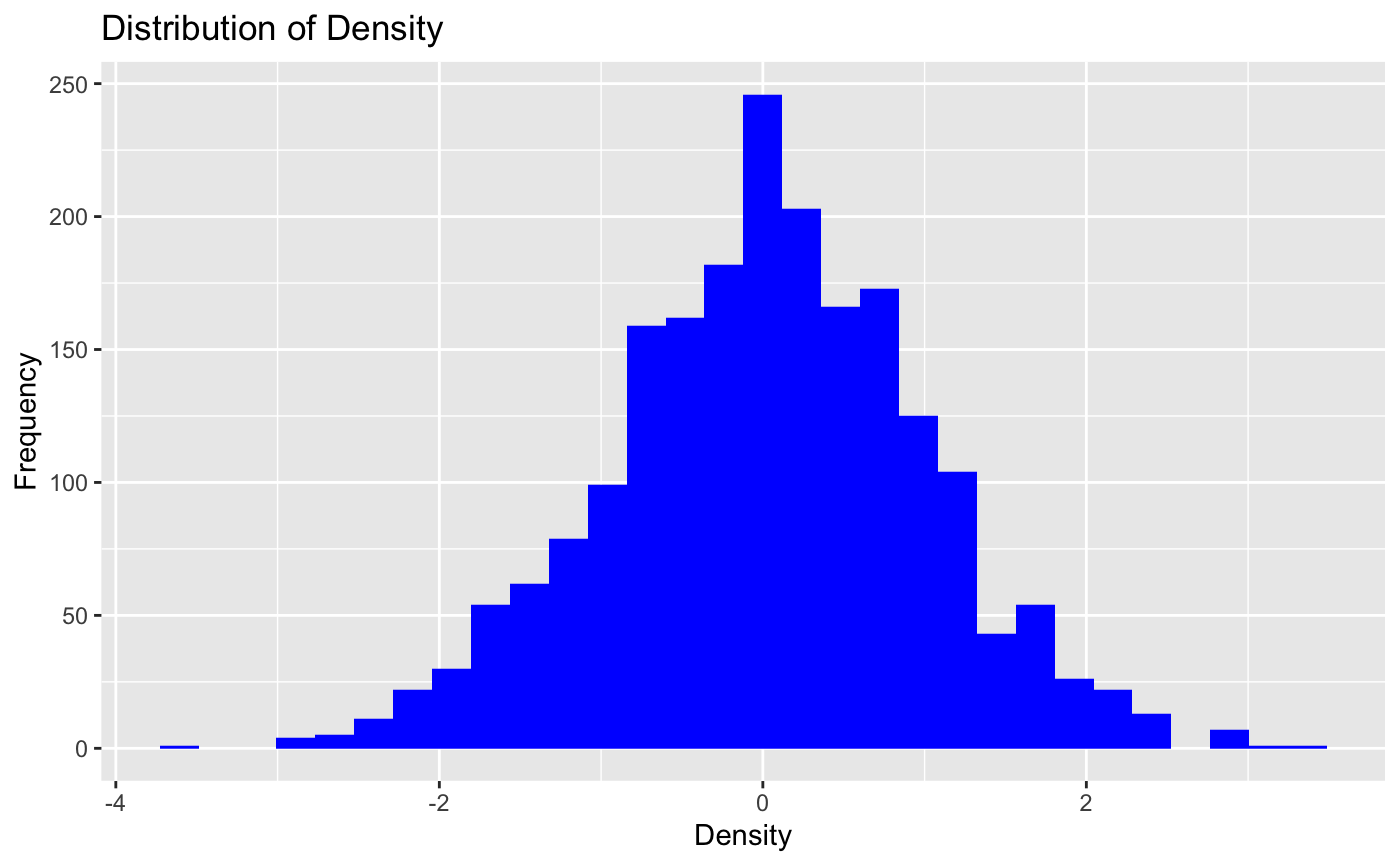
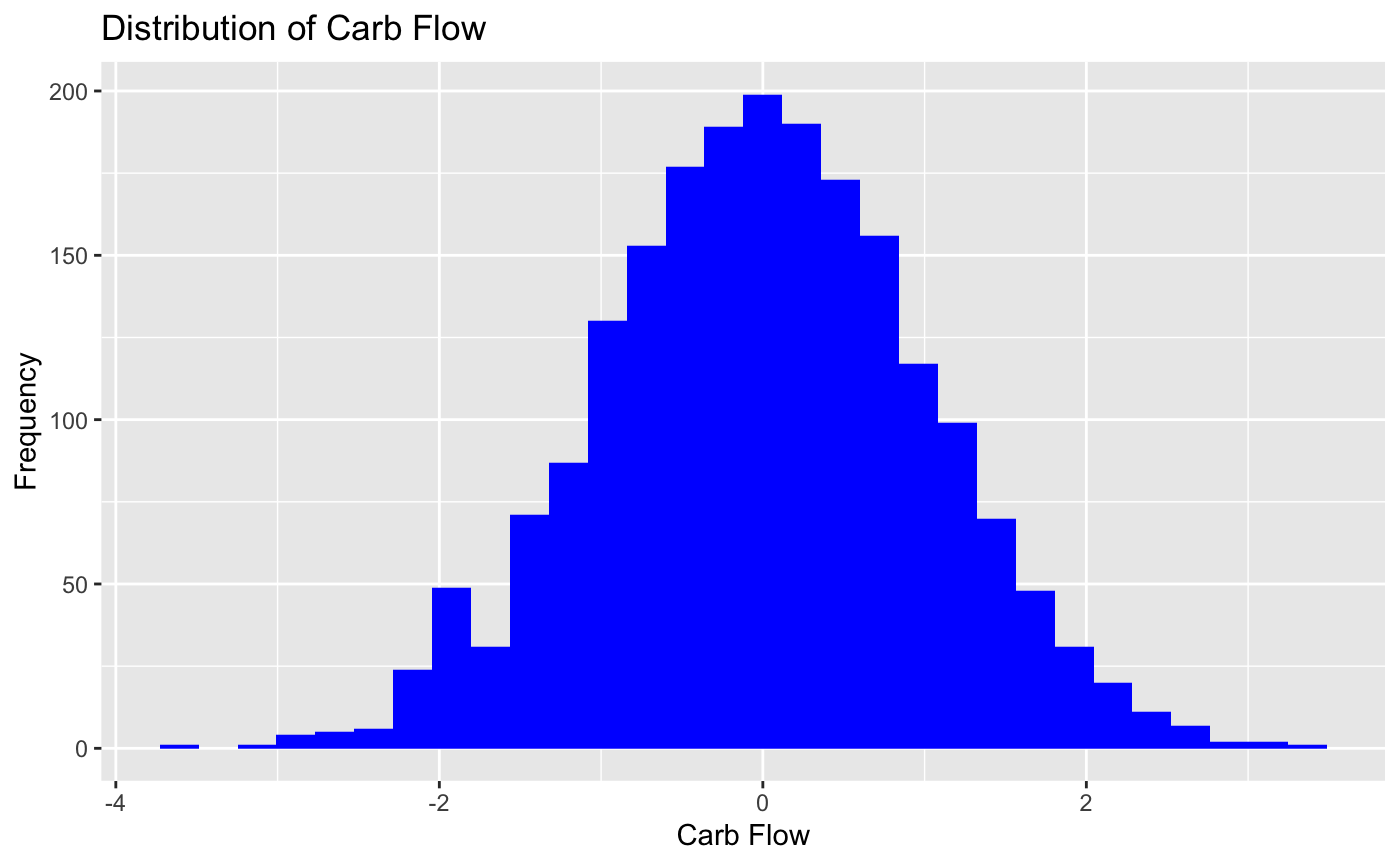


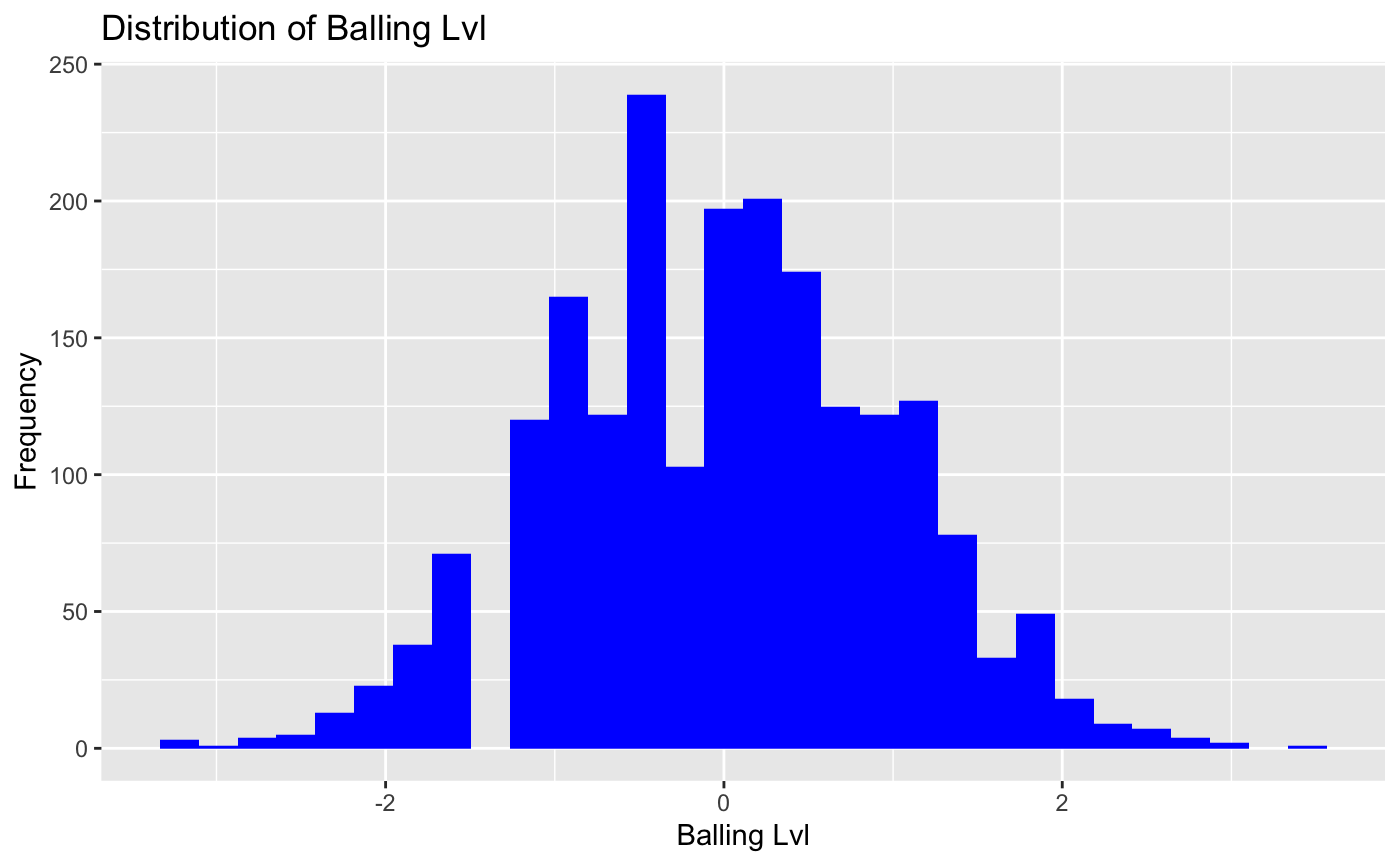
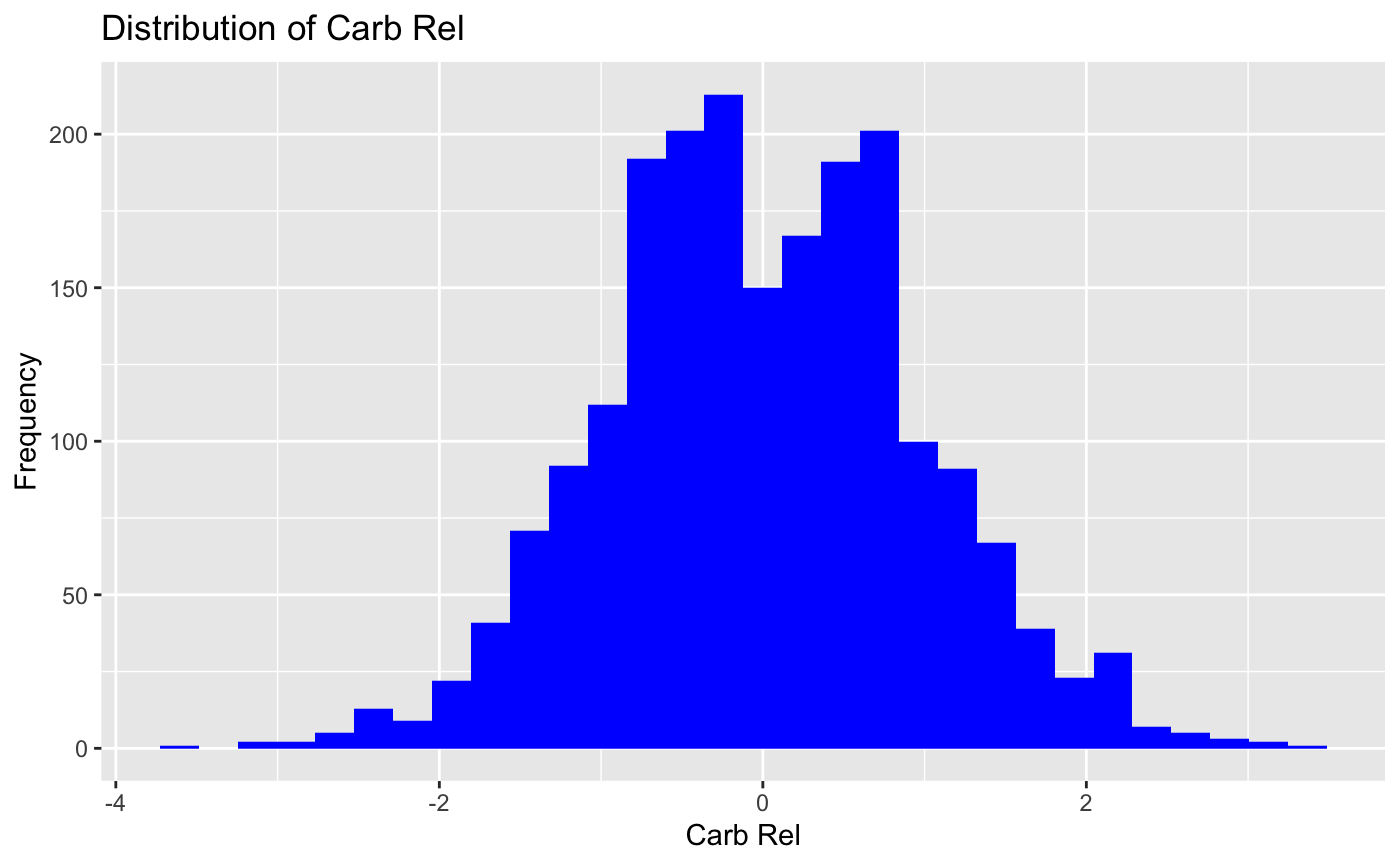
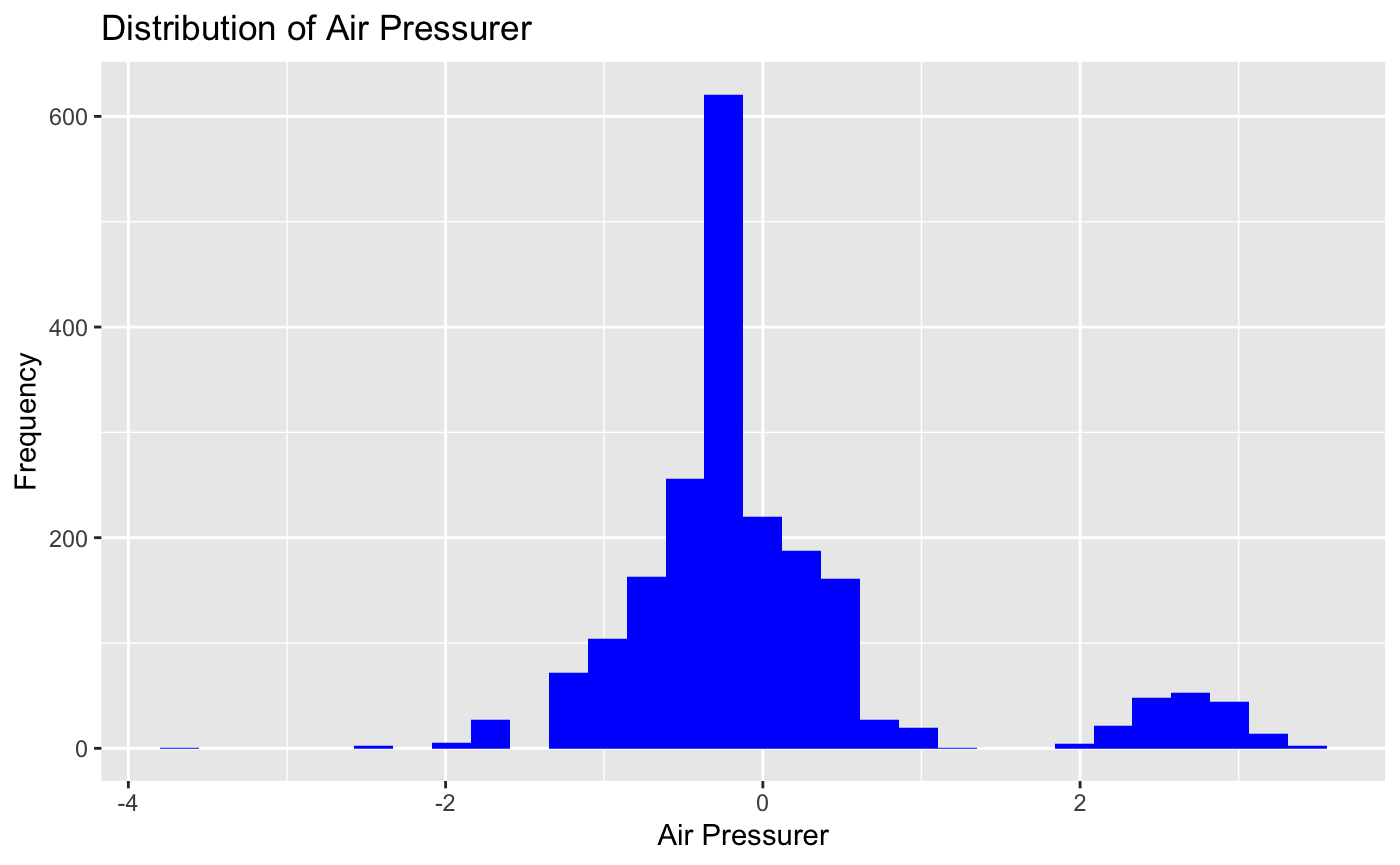
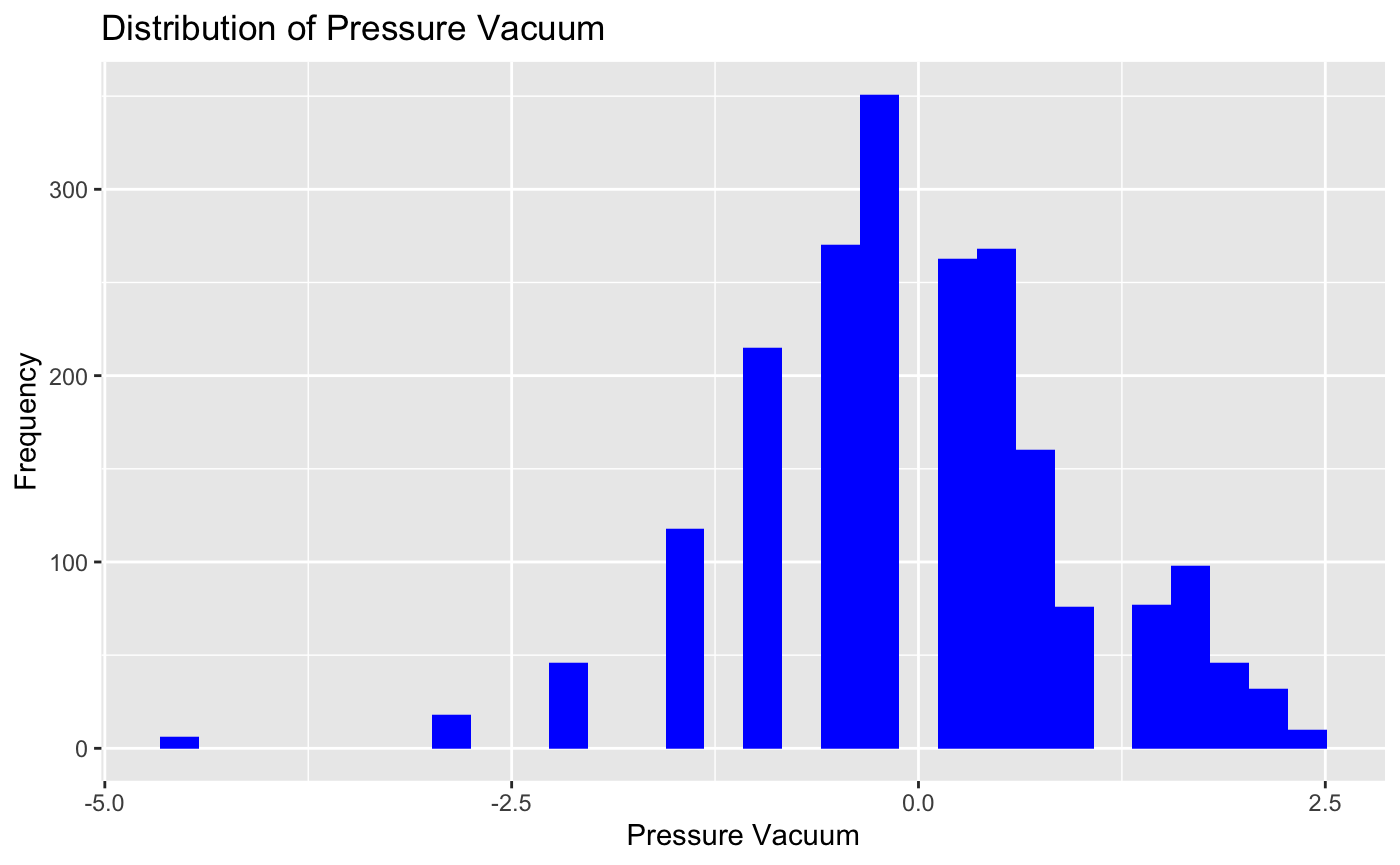
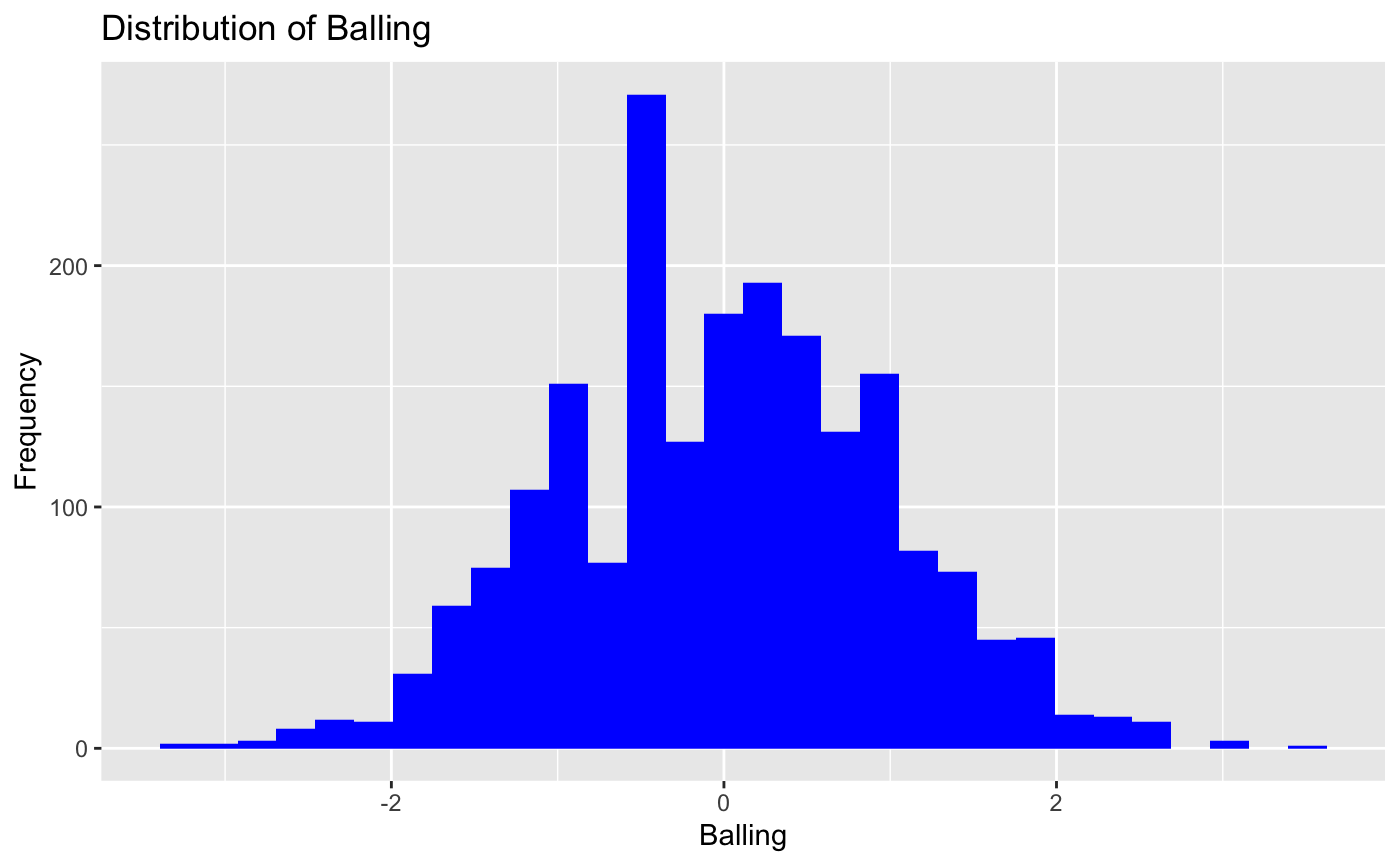












Looking at the graphs, the Hyd Pressure columns are still imperfect, but overall the data are now more normally distributed.

### 

### Correlations

To ensure the quality of our data, we need to investigate correlations between variables. High correlations can indicate redundancy, which can negatively impact our model's performance.

First, we'll visualize the correlation matrix of the numeric variables using a color-coded plot:

```{r}

library(corrplot)

numeric\_data = train\_transformed[, numeric\_cols]

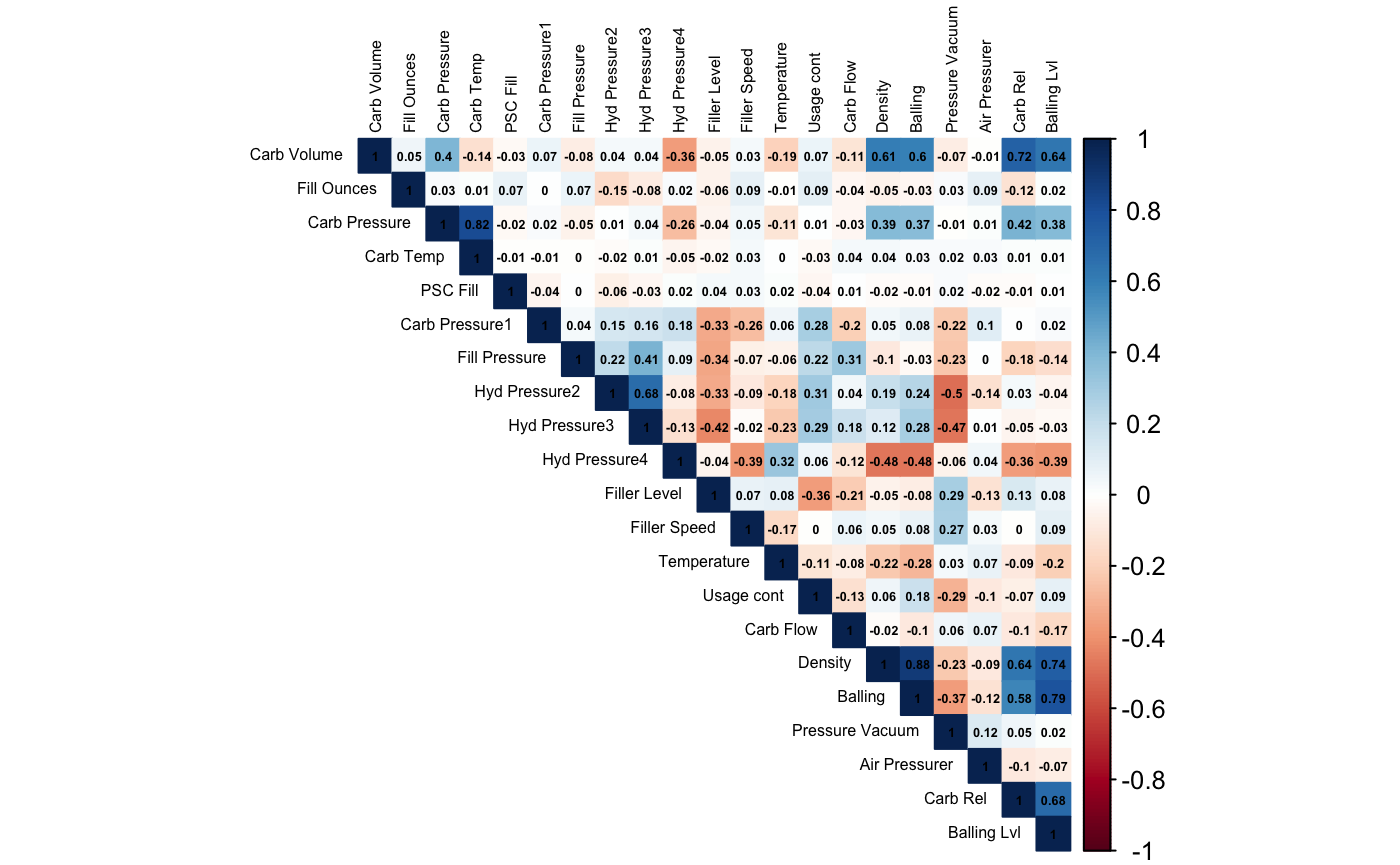
correlation\_matrix = cor(numeric\_data, use = "complete.obs")

corrplot(correlation\_matrix, method = "color", type = "upper",

tl.col = "black",

addCoef.col = "black", number.cex = 0.4, tl.cex = 0.5)

```



The correlation plot reveals several pairs of highly correlated variables. To streamline our analysis, we'll focus on pairs with correlations greater than 0.7:

```{r}

numeric\_cols = train\_transformed %>% select\_if(is.numeric)

correlation\_matrix = cor(numeric\_cols, use = "complete.obs")

# Finding correlations greater than .7

high\_correlations = which(abs(correlation\_matrix) > 0.7, arr.ind = TRUE)

# Filtering out the diagonal

high\_correlations = high\_correlations[high\_correlations[, 1] != high\_correlations[, 2], ]

# Creating df of high correlations

high\_corr\_df = data.frame(

Variable1 = rownames(correlation\_matrix)[high\_correlations[, 1]],

Variable2 = colnames(correlation\_matrix)[high\_correlations[, 2]],

Correlation = correlation\_matrix[high\_correlations]

)

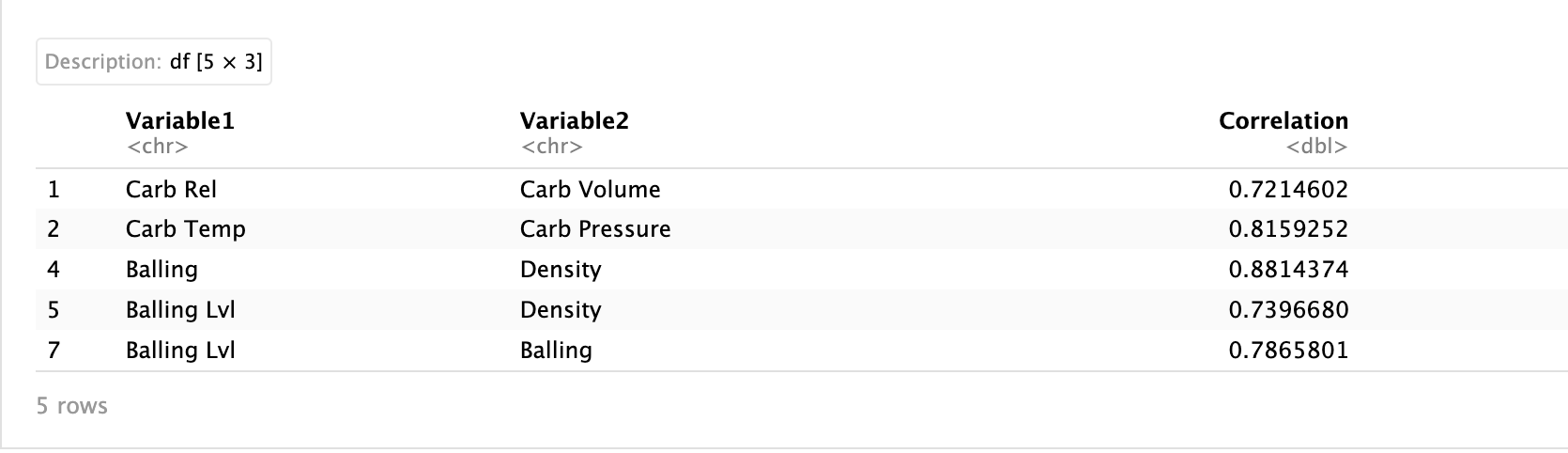
# Removing duplicate pairs

high\_corr\_df = high\_corr\_df[!duplicated(t(apply(high\_corr\_df, 1, sort))), ]

# And printing the high correlations

print(high\_corr\_df)

```



Based on this analysis, we identified several variables with high correlations. To reduce multicollinearity, we will remove the variables Carb Rel, Carb Temp, Balling, and Balling Lvl from the dataset:

```{r}

train\_transformed = train\_transformed %>%

dplyr::select(-c(`Carb Rel`, `Carb Temp`, `Balling`, `Balling Lvl`))

eval\_transformed = eval\_transformed %>%

dplyr::select(-c(`Carb Rel`, `Carb Temp`, `Balling`, `Balling Lvl`))

test\_transformed = test\_transformed %>%

dplyr::select(-c(`Carb Rel`, `Carb Temp`, `Balling`, `Balling Lvl`))

```

Although we've addressed numeric variable multicollinearity, categorical variables might still pose issues. We'll build a simple linear model and check the Variance Inflation Factor (VIF) values to detect any remaining multicollinearity:

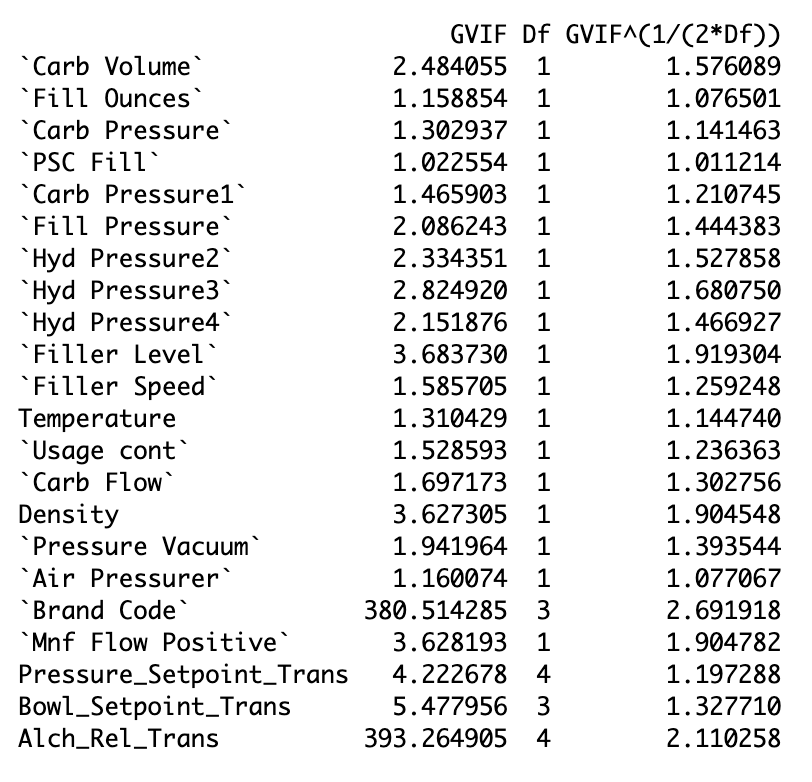
```{r}

library(car)

simple\_lm = lm(PH ~ ., data = train\_transformed)

print(vif(simple\_lm))

```



The VIF values indicate that Brand Code is highly correlated with Alch\_Rel\_Trans, likely because some brands sell alcoholic drinks. Since alcohol content is more directly related to pH than Brand Code, we will remove Brand Code:

```{r}

train\_transformed = train\_transformed %>%

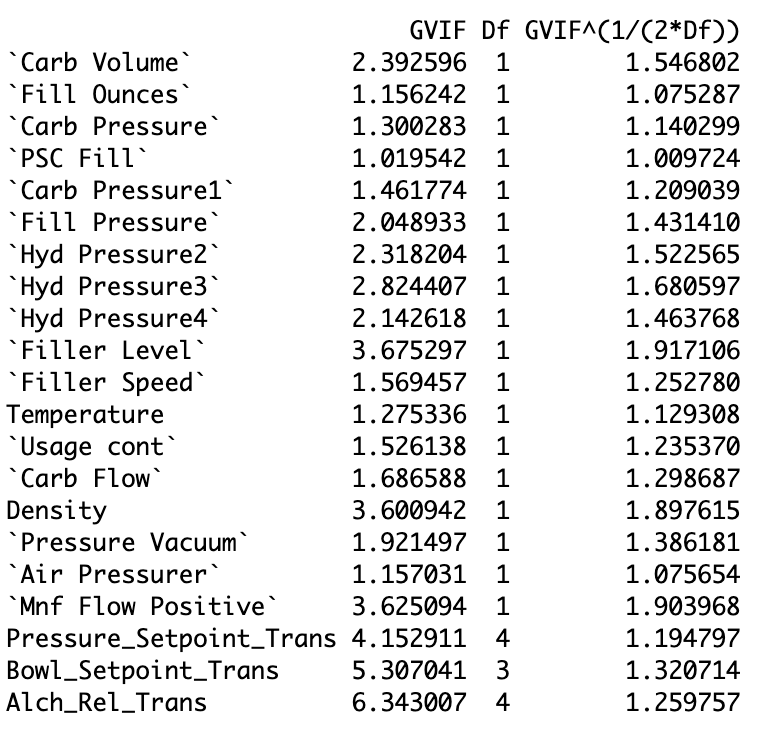
dplyr::select(-`Brand Code`)

# Recalculating VIFs

simple\_lm = lm(PH ~ ., data = train\_transformed)

print(vif(simple\_lm))

```



With the revised VIF values now at acceptable levels, we will also remove Brand Code from the evaluation and test sets:

```{r}

eval\_transformed = eval\_transformed %>%

dplyr::select(-`Brand Code`)

test\_transformed = test\_transformed %>%

dplyr::select(-`Brand Code`)

```

This concludes our step of handling correlations and multicollinearity, ensuring that our dataset is ready for robust analysis.

## Models and Evaluation

We will create and evaluate several models to determine the best approach for our dataset. To facilitate this, we will use caret's trainControl and train functions. The models we will train include:

1. Linear Regression
2. Partial Least Squares
3. Ridge Regression
4. Lasso Regression
5. Elastic Net
6. Neural Networks
7. Multivariate Adaptive Regression Splines (MARS)
8. Support Vector Machines (SVM with Radial Kernel)
9. K-Nearest Neighbors (KNN)

First, we need to center and scale our data:

```{r}

# Remove PH from the training set

train\_transformed\_without\_ph = train\_transformed %>% dplyr::select(-PH)

# Standardize data

preProcValues = preProcess(train\_transformed\_without\_ph, method = c("center", "scale"))

# Apply transformation to the training set

train\_transformed\_scaled = predict(preProcValues, train\_transformed\_without\_ph)

train\_transformed = cbind(train\_transformed\_scaled, PH = train\_transformed$PH)

```

Next, we apply the same transformations to the evaluation and test sets:

```{r}

# Remove PH from evaluation set

eval\_transformed\_without\_ph = eval\_transformed %>% dplyr::select(-PH)

# Apply transformation to evaluation and test sets

eval\_transformed\_scaled = predict(preProcValues, eval\_transformed\_without\_ph)

test\_transformed = predict(preProcValues, test\_transformed)

# Add PH back to eval

eval\_transformed = cbind(eval\_transformed\_scaled, PH = eval\_transformed$PH)

```

We also need to convert Mnf Flow Positive to a factor and then one-hot encode all the factor variables:

```{r}

train\_transformed$`Mnf Flow Positive` = as.factor(train\_transformed$`Mnf Flow Positive`)

eval\_transformed$`Mnf Flow Positive` = as.factor(eval\_transformed$`Mnf Flow Positive`)

test\_transformed$`Mnf Flow Positive` = as.factor(test\_transformed$`Mnf Flow Positive`)

```

One-hot encoding the factor variables converts them into binary columns, which makes it easier for machine learning algorithms to process them. We load the necessary libraries for modeling:

```{r}

dummies = dummyVars(~ ., data = train\_transformed[, -which(names(train\_transformed) == "PH")])

train\_transformed\_encoded = predict(dummies, newdata = train\_transformed)

eval\_transformed\_encoded = predict(dummies, newdata = eval\_transformed)

test\_transformed\_encoded = predict(dummies, newdata = test\_transformed)

# Converting the matrices to data frames

train\_transformed\_encoded = as.data.frame(train\_transformed\_encoded)

eval\_transformed\_encoded = as.data.frame(eval\_transformed\_encoded)

test\_transformed\_encoded = as.data.frame(test\_transformed\_encoded)

# Adding PH back to training, eval sets

train\_transformed\_encoded$PH = train\_transformed$PH

eval\_transformed\_encoded$PH = eval\_transformed$PH

```

```{r}

library(nnet)

library(earth)

library(glmnet)

```

Loading these libraries provides the functions and tools required for our planned machine learning models. To use Mean Absolute Percentage Error (MAPE) in our trainControl, we define a custom summary function:

```{r}

custom\_summary = function(data, lev = NULL, model = NULL) {

mape = mean(abs((data$obs - data$pred) / data$obs)) \* 100

defaultSummary(data, lev, model) %>%

c(MAPE = mape)

}

```

This custom summary function calculates MAPE, which will help us evaluate the performance of our models more effectively. We then set up the training control with cross-validation:

```{r}

train\_control = trainControl(

method = "cv",

number = 10,

summaryFunction = custom\_summary,

savePredictions = "final",

verboseIter = TRUE

)

```

The training control setup specifies that we will use 10-fold cross-validation and our custom summary function for model evaluation. If we encounter an error about zero variance, we identify the columns with zero variance:

```{r}

zero\_variance\_cols = nearZeroVar(train\_transformed\_encoded, saveMetrics = TRUE)

print(zero\_variance\_cols[zero\_variance\_cols$zeroVar, ])

```



Identifying columns with zero variance is crucial because such columns do not provide any useful information for the models and can be safely removed to improve model performance. This concludes our preparation steps for modeling, ensuring our data is properly scaled, encoded, and ready for training various machine learning models.

First, we need to remove columns with zero variance as they do not contribute useful information to the model:

```{r}

train\_transformed\_encoded = train\_transformed\_encoded %>%

dplyr::select(-`Bowl\_Setpoint\_Trans.> 2`)

eval\_transformed\_encoded = eval\_transformed\_encoded %>%

dplyr::select(-`Bowl\_Setpoint\_Trans.> 2`)

test\_transformed\_encoded = test\_transformed\_encoded %>%

dplyr::select(-`Bowl\_Setpoint\_Trans.> 2`)

```

Next, we train various machine learning models using the caret package. The models include Linear Regression, Partial Least Squares, Ridge Regression, Lasso Regression, Elastic Net, Neural Networks, Multivariate Adaptive Regression Splines, Support Vector Machines (svmRadial), and K-Nearest Neighbors.

```{r}

models = list(

lm = train(PH ~ ., data = train\_transformed\_encoded, method = "lm", trControl = train\_control),

pls = train(PH ~ ., data = train\_transformed\_encoded, method = "pls", trControl = train\_control),

ridge = train(PH ~ ., data = train\_transformed\_encoded, method = "ridge", trControl = train\_control, tuneLength = 10),

lasso = train(PH ~ ., data = train\_transformed\_encoded, method = "lasso", trControl = train\_control, tuneLength = 10),

enet = train(PH ~ ., data = train\_transformed\_encoded, method = "enet", trControl = train\_control, tuneLength = 10),

nnet = train(PH ~ ., data = train\_transformed\_encoded, method = "nnet", trControl = train\_control, trace = FALSE, linout = TRUE, tuneLength = 10),

mars = train(PH ~ ., data = train\_transformed\_encoded, method = "earth", trControl = train\_control, tuneLength = 10),

svm = train(PH ~ ., data = train\_transformed\_encoded, method = "svmRadial", trControl = train\_control, tuneLength = 10),

knn = train(PH ~ ., data = train\_transformed\_encoded, method = "knn", trControl = train\_control, tuneLength = 10)

)

results = resamples(models)

summary(results)

```

The summarized results for Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), Root Mean Squared Error (RMSE), and R-squared values are as follows:

MAE

Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

lm 0.10164139 0.10251256 0.10419940 0.10604650 0.10912448 0.11592647 0

pls 0.10096538 0.10590359 0.10982822 0.10994773 0.11470002 0.11859478 0

ridge 0.09679268 0.09942110 0.10621825 0.10571049 0.10949082 0.11528261 0

lasso 0.09851515 0.10142763 0.10741330 0.10620320 0.11026423 0.11341061 0

enet 0.09893908 0.10113967 0.10252853 0.10579411 0.10593478 0.12693192 0

nnet 0.08825363 0.08934181 0.09035573 0.09317154 0.09649000 0.10255911 0

mars 0.09650411 0.09807825 0.10000866 0.10091341 0.10302456 0.10942195 0

svm 0.07277847 0.08005055 0.08491111 0.08429567 0.09000963 0.09291149 0

knn 0.07707317 0.08253522 0.08839806 0.08812406 0.09255825 0.09786992 0

MAPE

Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

lm 1.1916679 1.2018228 1.2231824 1.2447561 1.280572 1.361662 0

pls 1.1797883 1.2412228 1.2919737 1.2906201 1.346717 1.395563 0

ridge 1.1359276 1.1684459 1.2491273 1.2408757 1.287537 1.351498 0

lasso 1.1534683 1.1912191 1.2602352 1.2467332 1.298710 1.334003 0

enet 1.1636387 1.1855545 1.2022117 1.2419606 1.243026 1.495731 0

nnet 1.0330378 1.0473051 1.0609964 1.0939122 1.133730 1.203399 0

mars 1.1321516 1.1508341 1.1723814 1.1853006 1.209202 1.287940 0

svm 0.8541687 0.9415000 0.9988711 0.9913696 1.057846 1.094357 0

knn 0.9060059 0.9708597 1.0396945 1.0363064 1.088832 1.154238 0

RMSE

Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

lm 0.1287930 0.1325277 0.1362437 0.1368926 0.1387375 0.1476350 0

pls 0.1265134 0.1353538 0.1433673 0.1408408 0.1459657 0.1548285 0

ridge 0.1271826 0.1303758 0.1374303 0.1364190 0.1407193 0.1489395 0

lasso 0.1254010 0.1297146 0.1351062 0.1365229 0.1426242 0.1523896 0

enet 0.1251163 0.1286107 0.1328380 0.1361211 0.1383498 0.1600881 0

nnet 0.1121481 0.1164474 0.1228666 0.1224904 0.1264715 0.1362914 0

mars 0.1235899 0.1302957 0.1327798 0.1328013 0.1353850 0.1436433 0

svm 0.1055271 0.1088648 0.1198229 0.1182978 0.1240806 0.1337386 0

knn 0.1050759 0.1126564 0.1232082 0.1205954 0.1271983 0.1349977 0

Rsquared

Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

lm 0.2806798 0.3363522 0.3533876 0.3489354 0.3828410 0.3903054 0

pls 0.1882583 0.2824471 0.3266160 0.3128746 0.3442444 0.3758867 0

ridge 0.2948061 0.3225040 0.3567750 0.3548575 0.3803062 0.4233718 0

lasso 0.3057756 0.3191633 0.3584300 0.3535080 0.3736123 0.4327396 0

enet 0.2804868 0.3118119 0.3681973 0.3556181 0.3966171 0.4129961 0

nnet 0.3999156 0.4673620 0.4991341 0.4883450 0.5104003 0.5465416 0

mars 0.3279399 0.3612844 0.3760820 0.3876566 0.4134585 0.4616637 0

svm 0.4245008 0.4719048 0.5365107 0.5209902 0.5694413 0.5915128 0

knn 0.4190152 0.4480026 0.4830352 0.4992955 0.5647440 0.5859971 0

The SVM model has the lowest RMSE, the highest R-squared, and the lowest MAPE, indicating the best performance among the models.

### **Evaluating Models on the Evaluation Set**

We then evaluate all models on the evaluation set:

```{r}

# Function to calculate evaluation metrics (ultimately for eval set)

evaluate\_model = function(model, eval\_data) {

predictions = predict(model, eval\_data)

obs = eval\_data$PH

data.frame(

RMSE = sqrt(mean((predictions - obs)^2)),

R\_squared = cor(predictions, obs)^2,

MAPE = mean(abs((obs - predictions) / obs)) \* 100

)

}

# Evaluating all models on the evaluation set

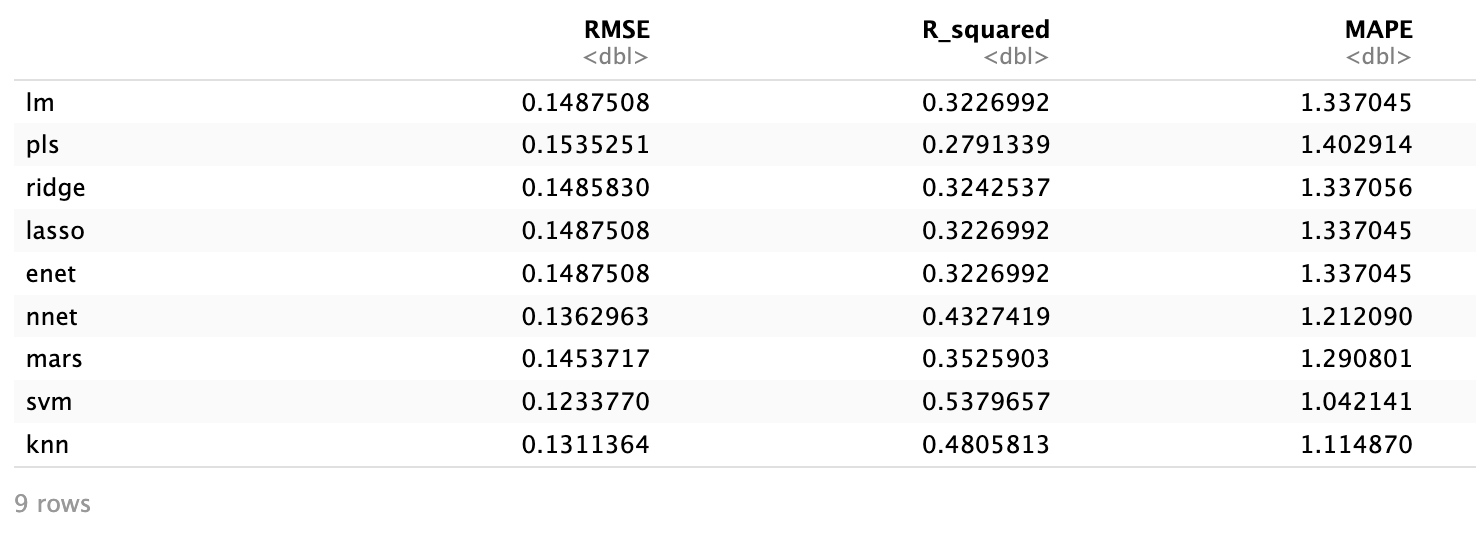
eval\_results = lapply(models, evaluate\_model, eval\_data = eval\_transformed\_encoded)

eval\_results = do.call(rbind, eval\_results)

row.names(eval\_results) = names(models)

print(eval\_results)

```



Again, the SVM model shows the best performance across all metrics. Given this, we decide to try another non-linear model, Random Forest, to see if we can achieve even better results.

### **Training and Evaluating a Random Forest Model**

We start by training a Random Forest model and evaluating it on the evaluation set:

```{r}

library(randomForest)

train\_control = trainControl(method = "cv", number = 10, verboseIter = TRUE)

# Train model

set.seed(7)

rf\_model = train(

PH ~ .,

data = train\_transformed\_encoded,

method = "rf",

trControl = train\_control

)

# Predictions on evaluation set

eval\_predictions\_rf = predict(rf\_model, newdata = eval\_transformed\_encoded)

# Calculating metrics

eval\_rmse\_rf = RMSE(eval\_predictions\_rf, eval\_transformed$PH)

eval\_r2\_rf = R2(eval\_predictions\_rf, eval\_transformed$PH)

eval\_mape\_rf = mean(abs((eval\_transformed$PH - eval\_predictions\_rf) / eval\_transformed$PH)) \* 100

cat("RMSE:", eval\_rmse\_rf, "\n")

cat("R-squared:", eval\_r2\_rf, "\n")

cat("MAPE:", eval\_mape\_rf, "\n")

```

Results:  
RMSE: 0.1156583

R-squared: 0.6037725

MAPE: 0.9959139

The Random Forest model outperforms all other models on all metrics. We then fine-tune the model by exploring different hyperparameters:

```{r}

# Custom RF training function

rf\_custom = function(x, y, ntree = 500, mtry, nodesize = 1, maxnodes = NULL) {

randomForest(x, y, ntree = ntree, mtry = mtry, nodesize = nodesize, maxnodes = maxnodes)

}

# Define the hyperparameter grid

tune\_grid = expand.grid(

mtry = c(15, 17, 20, 25),

nodesize = c(5, 10),

maxnodes = c(50, 100, 150, 500),

ntree = 500

)

# Set up train control with 5-fold cross-validation

train\_control = trainControl(

method = "cv",

number = 5,

verboseIter = TRUE

)

# Custom function for training the Random Forest model

rf\_model\_custom = function(train\_data, train\_labels, param) {

randomForest(

x = train\_data, y = train\_labels,

mtry = param$mtry,

ntree = param$ntree,

nodesize = param$nodesize,

maxnodes = param$maxnodes

)

}

# Creating custom function to train and evaluate model with 5-fold cross-validation (using caret directly ran into some issues)

train\_rf\_custom\_cv = function(data, formula, tuneGrid, trControl) {

results = list()

total\_iterations = nrow(tuneGrid)

iter = 1

for (i in 1:nrow(tuneGrid)) {

fold\_metrics = list()

folds = createFolds(data$PH, k = trControl$number, list = TRUE, returnTrain = TRUE)

for (fold in folds) {

train\_data = data[fold, ]

test\_data = data[-fold, ]

# Tracking progress

print(paste("Training with parameters:", paste(tuneGrid[i,], collapse = " ")))

print(paste("Progress:", iter, "/", total\_iterations \* trControl$number))

# Training the model with custom params

model = rf\_model\_custom(

train\_data = model.matrix(formula, train\_data)[, -1],

train\_labels = train\_data$PH,

param = tuneGrid[i,]

)

# Predicting on the "test set"

predictions = predict(model, newdata = model.matrix(formula, test\_data)[, -1])

# Calculating evaluation metrics

rmse = RMSE(predictions, test\_data$PH)

r2 = R2(predictions, test\_data$PH)

mape = mean(abs((test\_data$PH - predictions) / test\_data$PH)) \* 100

# Storing fold metrics

fold\_metrics[[length(fold\_metrics) + 1]] = c(RMSE = rmse, R2 = r2, MAPE = mape)

iter = iter + 1

}

# Average fold metrics

avg\_rmse = mean(sapply(fold\_metrics, function(x) x["RMSE"]))

avg\_r2 = mean(sapply(fold\_metrics, function(x) x["R2"]))

avg\_mape = mean(sapply(fold\_metrics, function(x) x["MAPE"]))

# Store average metrics for current parameters

results[[i]] = c(tuneGrid[i,], RMSE = avg\_rmse, R2 = avg\_r2, MAPE = avg\_mape)

}

do.call(rbind, results)

}

# Set seed for reproducibility

set.seed(7)

# Run RF training with 5-fold cross-validation

results\_cv = train\_rf\_custom\_cv(

data = train\_transformed\_encoded,

formula = PH ~ .,

tuneGrid = tune\_grid,

trControl = train\_control

)

```

After training and cross-validation, we summarize the results and find the best hyperparameters:

```{r}

results\_df = as.data.frame(results\_cv, stringsAsFactors = FALSE)

results\_df$mtry = as.numeric(results\_df$mtry)

results\_df$nodesize = as.numeric(results\_df$nodesize)

results\_df$maxnodes = as.numeric(results\_df$maxnodes)

results\_df$ntree = as.numeric(results\_df$ntree)

results\_df$RMSE = as.numeric(results\_df$RMSE)

results\_df$R2 = as.numeric(results\_df$R2)

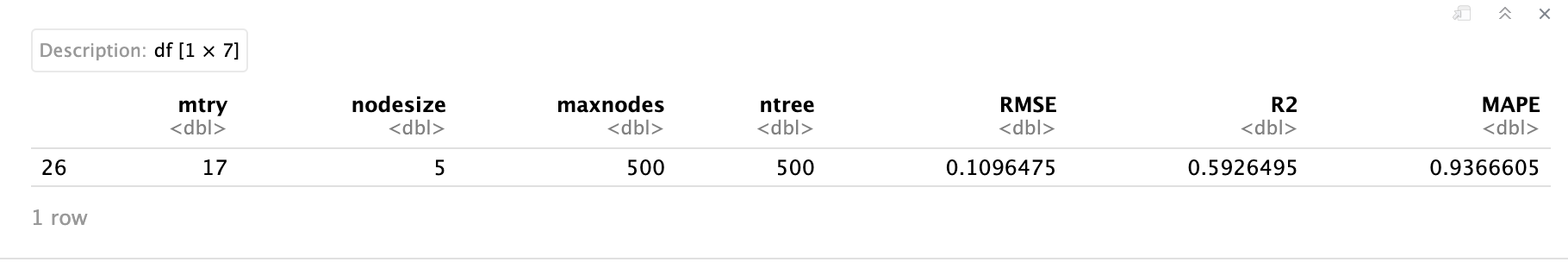
results\_df$MAPE = as.numeric(results\_df$MAPE)

# Find best params (based on MAPE)

best\_params = results\_df[order(results\_df$MAPE), ][1, ]

print(best\_params)

```



With the best hyperparameters, we train a new Random Forest model on the full data and evaluate it on the evaluation set:

```{r}

final\_model = randomForest(

x = model.matrix(PH ~ ., train\_transformed\_encoded)[, -1],

y = train\_transformed\_encoded$PH,

mtry = 17,

ntree = 500,

nodesize = 5,

maxnodes = 500

)

# Evaluation eval set

eval\_predictions = predict(final\_model, newdata = model.matrix(PH ~ ., eval\_transformed\_encoded)[, -1])

# Metrics

eval\_rmse = RMSE(eval\_predictions, eval\_transformed\_encoded$PH)

eval\_r2 = R2(eval\_predictions, eval\_transformed\_encoded$PH)

eval\_mape = mean(abs((eval\_transformed\_encoded$PH - eval\_predictions) / eval\_transformed\_encoded$PH)) \* 100

cat("Evaluation RMSE:", eval\_rmse, "\n")

cat("Evaluation R2:", eval\_r2, "\n")

cat("Evaluation MAPE:", eval\_mape, "\n")

```

Evaluation RMSE: 0.1159889

Evaluation R2: 0.6007391

Evaluation MAPE: 1.00507  
  
Disappointingly, the fine-tuned Random Forest model did not outperform the previously trained Random Forest model (rf\_model). The performance metrics of rf\_model are as follows:

* RMSE: 0.1156583
* R-squared: 0.6037725
* MAPE: 0.9959139

This discrepancy is likely due to differences in hyperparameters other than mtry. Given this, we'll examine feature importance using rf\_model and use it to generate predictions for the test set.

First, we extract the Random Forest model and calculate feature importance:

```{r}

# Extracting RF model

final\_rf\_model = rf\_model$finalModel

# Calculating feature importance, converting to df

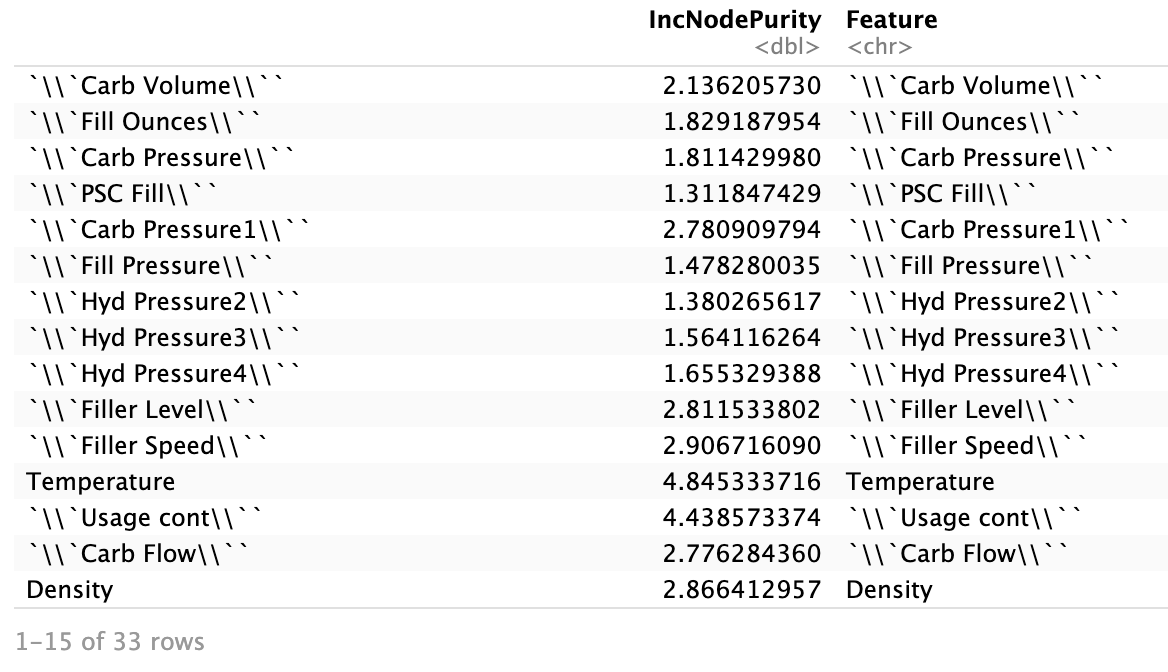
feature\_importance = importance(final\_rf\_model)

feature\_importance\_df = as.data.frame(feature\_importance)

feature\_importance\_df$Feature = rownames(feature\_importance\_df)

print(feature\_importance\_df)

```

  
  
The feature importance values (IncNodePurity) indicate the contribution of each feature to the model  
  
We observe that Carb Volume, Fill Ounces, and Carb Pressure1 are among the most important features. This indicates these features have the highest contribution to reducing node impurity in the Random Forest model.

Plotting Feature Importance

```{r}

feature\_importance\_df$IncNodePurity = as.numeric(feature\_importance\_df$IncNodePurity)

ggplot(feature\_importance\_df, aes(x = reorder(Feature, IncNodePurity), y = IncNodePurity)) +

geom\_bar(stat = "identity", fill = "blue") +

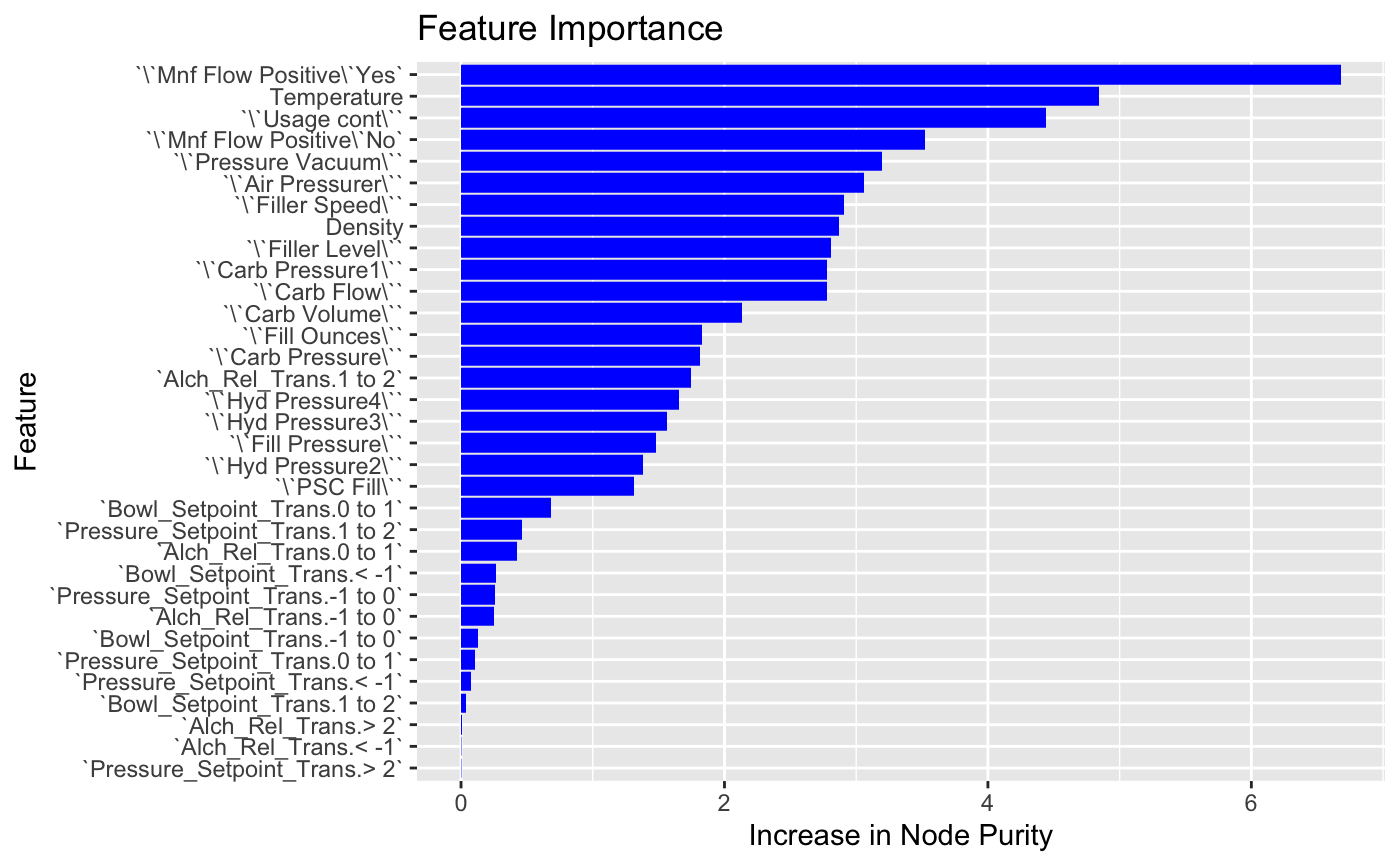
coord\_flip() +

xlab("Feature") +

ylab("Increase in Node Purity") +

ggtitle("Feature Importance")

```



The feature importance plot doesn't indicate the direction of the relationship between the variables and PH. To address this, we can use the correlation matrix for the numeric variables to get a sense of the direction of their impact on PH:

```{r}

numeric\_columns = sapply(train, is.numeric)

correlations = cor(train[, numeric\_columns], use = "complete.obs")['PH',] # correlations with ph

correlations\_df = data.frame(Feature = names(correlations), Correlation = correlations)

#take out correlation of PH with itself

correlations\_df = correlations\_df[rownames(correlations\_df) != 'PH',]

```

## Predictions

Finally, let's generate predictions for the test set.

```{r}

# We can't use the object itself (errors) so we can use the non-formula method with train and then make predictions

train\_data = train\_transformed\_encoded[, !names(train\_transformed\_encoded) %in% "PH"]

train\_labels = train\_transformed\_encoded$PH

# Train again using mtry = 17

set.seed(7)

final\_rf\_model = randomForest(

x = train\_data,

y = train\_labels,

mtry = 17

)

# Generate predictions on the test set

test\_predictions = predict(final\_rf\_model, newdata = test\_transformed\_encoded)

```

```{r}

# loading test set again so that it has the original values:

library(tidyverse)

test = read\_csv("https://raw.githubusercontent.com/gsteinmetzsilber/DATA-624-Project-2/main/Data/StudentEvaluation-%20TO%20PREDICT.xlsx%20-%20Subset%20(2).csv")

```

### Evaluate Predictions

Now, let's evaluate the predictions:

```{r}

# Adding predictions to the test df

test$PH = test\_predictions

test$PH = as.numeric(as.character(test$PH))

```

```{r}

# Writing to an xlsx file

library(writexl)

write\_xlsx(test, "test\_with\_predictions.xlsx")

```

## Conclusion

In this project, we aimed to predict pH levels in beverage production using a variety of machine learning models. Our process included comprehensive data preprocessing, such as handling missing values, removing highly correlated variables, scaling data, and encoding categorical variables.

We trained and evaluated several models, including Linear Regression, Partial Least Squares, Ridge Regression, Lasso Regression, Elastic Net, Neural Networks, Multivariate Adaptive Regression Splines, Support Vector Machines, and K-Nearest Neighbors. The Support Vector Machine (SVM) initially showed the best performance.

To further enhance our predictions, we implemented and fine-tuned a Random Forest model. The Random Forest model outperformed all other models, achieving the lowest RMSE, highest R-squared, and lowest MAPE values. Despite additional hyperparameter tuning, the initial Random Forest model (rf\_model) demonstrated superior performance.

Feature importance analysis provided insights into the key variables influencing pH levels. To understand the direction of these impacts, we used a correlation matrix for numeric variables.

Finally, we generated predictions for the test set using the Random Forest model and saved these predictions to an Excel file.

Overall, this project highlights the importance of thorough data preprocessing, careful model selection, and effective hyperparameter tuning in developing accurate predictive models. The Random Forest model, with its ability to handle a large number of features and interactions, proved to be the best choice for our prediction task.