Final Project

Matthew Tillmawitz, Heleine Fouda, Marjete Vucinaj, Lewris Mota, Kim Koon

2024-12-15

Contents

1. Introduction

Recent regulatory changes emphasize the importance of having a comprehensive understanding of manufacturing processes and their impact on product quality. At ABC Beverage, pH levels are a critical parameter for ensuring consistency and maintaining product standards. This analysis aims to identify and quantify the factors driving pH variability while developing a predictive model that meets these new regulatory standards. By leveraging advanced analytical methods and predictive modeling techniques, the analysis provides data-driven insights to ensure compliance and enhance understanding of the production process. The scope of this analysis includes data preprocessing, the identification of predictive factors, and the development and validation of a robust predictive model for pH levels. The focus is on exploring relationships within the manufacturing data to uncover insights that can drive better decision-making. By implementing this structured approach, the findings will provide actionable insights into the key drivers of pH variability and a reliable framework for prediction.

Understanding and predicting pH levels is crucial not only for meeting regulatory standards but also for maintaining operational excellence. Accurate prediction will enable more efficient quality control, reducing variability and ensuring that ABC Beverage consistently delivers high-quality products. The results of this analysis will empower the company to address compliance needs while optimizing its manufacturing process, underscoring the importance of data-driven decision-making in the production environment.

2. Dataset Overview

The dataset comprises observations collected from a beverage production line, capturing information about carbonation levels, filling processes, environmental conditions, and quality control metrics. Each row represents a single production instance, and each column corresponds to a specific variable measured or controlled during the process.

```r
# Create a dataframe in R
variables <- data.frame(
  Feature = c(
    "Brand Code", "Carb Volume", "Fill Ounces", "PC Volume", "Carb Pressure",
    "Carb Temp", "PSC", "PSC Fill", "PSC CO2", "Mnf Flow", "Carb Pressure1",
    "Fill Pressure", "Hyd Pressure1", "Hyd Pressure2", "Hyd Pressure3", "Hyd Pressure4",
    "Filler Level", "Filler Speed", "Temperature", "Usage cont", "Carb Flow",
    "Density", "MFR", "Balling", "Pressure Vacuum", "PH", "Oxygen Filler",
    "Bowl Setpoint", "Pressure Setpoint", "Air Pressure", "Alch Rel", "Carb Rel",
    "Balling Lvl"
  ),
  Description = c(
    "Unique identifier for the product's brand.",
    "Volume of carbon dioxide dissolved in the product.",
    "Volume of liquid dispensed into each container.",
    "Process control volume for monitoring liquid levels.",
    "Pressure level during carbonation.",
    "Temperature during carbonation for CO2 solubility.",
    "Process Setpoint Control for maintaining parameter targets.",
    "Filling setpoint under controlled conditions.",
    "Setpoint for CO2 levels during carbonation.",
    "Manufacturing flow rate for liquid or gas.",
    "Secondary carbonation pressure reading.",
    "Pressure applied during filling operations.",
    "Hydraulic pressure reading 1 for machine operation.",
    "Hydraulic pressure reading 2 for machine operation.",
    "Hydraulic pressure reading 3 for machine operation.",
    "Hydraulic pressure reading 4 for machine operation.",
    "Measurement of product level in containers.",
    "Speed of the filling machine or process.",
    "Temperature of the process environment.",
    "Container usage or consumption metrics.",
    "Flow rate of CO2 during carbonation.",
    "Density of the product for consistency monitoring.",
    "Mass flow rate of the material through the system.",
    "Sugar concentration level measured by the Balling scale.",
    "Vacuum pressure in the system.",
    "Acidity level of the product.",
    "Oxygen levels in the filling process.",
    "Target setpoint for intermediate container levels.",
    "Desired pressure level in the process.",
    "Air pressure measurement in the system.",
    "Alcohol release or related parameter.",
    "Carbonation release or related measurement.",
    "Sugar concentration level in the final product."
  ),
  Type = c(
    "Categorical", "Numerical", "Numerical", "Numerical", "Numerical",
    "Numerical", "Numerical", "Numerical", "Numerical", "Numerical",
    "Numerical", "Numerical", "Numerical", "Numerical", "Numerical",
    "Numerical", "Numerical", "Numerical", "Numerical", "Numerical",
    "Numerical", "Numerical", "Numerical", "Numerical", "Numerical",
    "Numerical", "Numerical", "Numerical", "Numerical", "Numerical",
    "Numerical", "Numerical", "Numerical"
  ),
  stringsAsFactors = FALSE
)

variables |> kable(caption = "Beverage Manufacture Process Features") |> kable_styling() |>  kable_clas
```

Table 1: Beverage Manufacture Process Features

Feature	Description	Type

Brand Code	Unique identifier for the product's brand.	Categorical
Carb Volume	Volume of carbon dioxide dissolved in the product.	Numerical
Fill Ounces	Volume of liquid dispensed into each container.	Numerical
PC Volume	Process control volume for monitoring liquid levels.	Numerical
Carb Pressure	Pressure level during carbonation.	Numerical
Carb Temp	Temperature during carbonation for CO2 solubility.	Numerical
PSC	Process Setpoint Control for maintaining parameter targets.	Numerical
PSC Fill	Filling setpoint under controlled conditions.	Numerical
PSC CO2	Setpoint for CO2 levels during carbonation.	Numerical
Mnf Flow	Manufacturing flow rate for liquid or gas.	Numerical
Carb Pressure1	Secondary carbonation pressure reading.	Numerical
Fill Pressure	Pressure applied during filling operations.	Numerical
Hyd Pressure1	Hydraulic pressure reading 1 for machine operation.	Numerical
Hyd Pressure2	Hydraulic pressure reading 2 for machine operation.	Numerical
Hyd Pressure3	Hydraulic pressure reading 3 for machine operation.	Numerical
Hyd Pressure4	Hydraulic pressure reading 4 for machine operation.	Numerical
Filler Level	Measurement of product level in containers.	Numerical
Filler Speed	Speed of the filling machine or process.	Numerical
Temperature	Temperature of the process environment.	Numerical
Usage cont	Container usage or consumption metrics.	Numerical
Carb Flow	Flow rate of CO2 during carbonation.	Numerical
Density	Density of the product for consistency monitoring.	Numerical
MFR	Mass flow rate of the material through the system.	Numerical
Balling	Sugar concentration level measured by the Balling scale.	Numerical
Pressure Vacuum	Vacuum pressure in the system.	Numerical
PH	Acidity level of the product.	Numerical
Oxygen Filler	Oxygen levels in the filling process.	Numerical
Bowl Setpoint	Target setpoint for intermediate container levels.	Numerical
Pressure Setpoint	Desired pressure level in the process.	Numerical
Air Pressure	Air pressure measurement in the system.	Numerical
Alch Rel	Alcohol release or related parameter.	Numerical
Carb Rel	Carbonation release or related measurement.	Numerical
Balling Lvl	Sugar concentration level in the final product.	Numerical

The variables play a crucial role in capturing and monitoring various aspects of the production process, directly impacting product quality and operational efficiency:

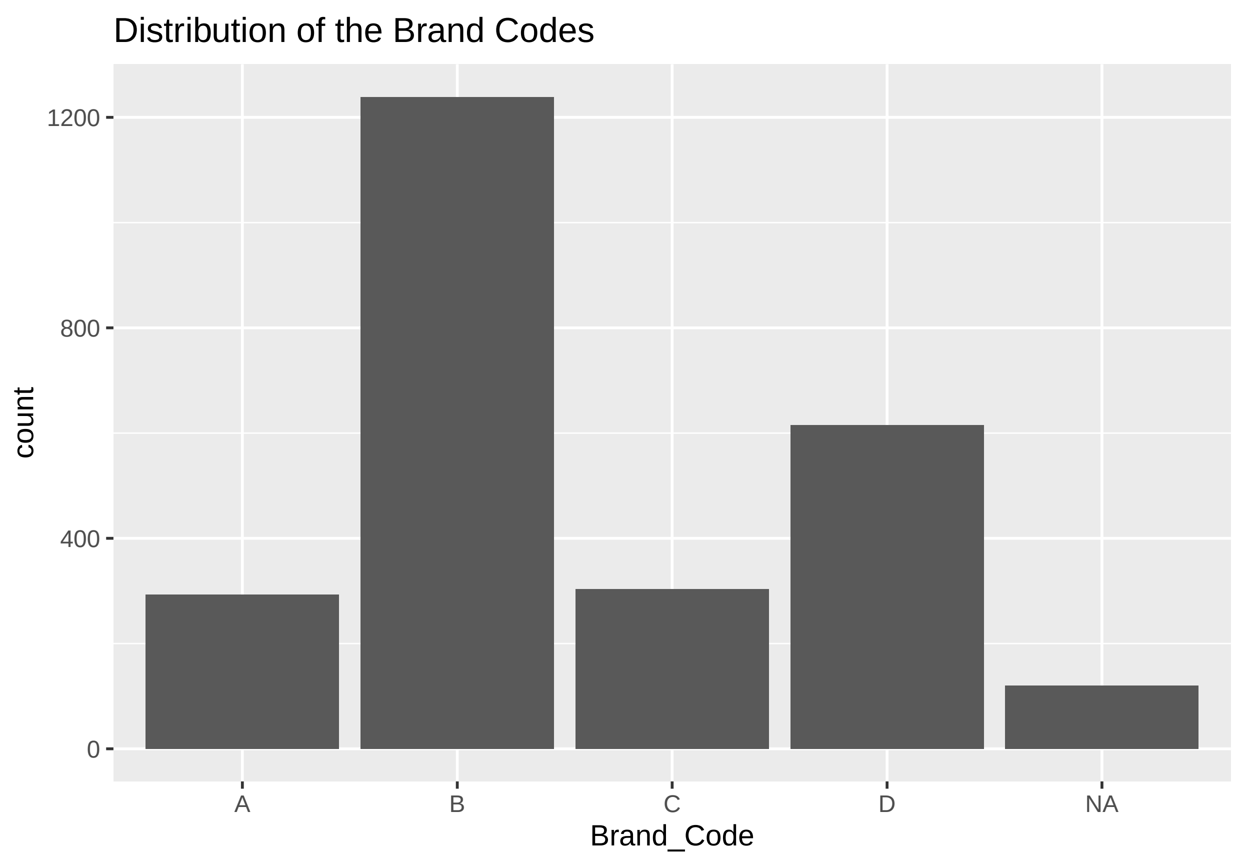
- **Carbonation Variables**: These variables (e.g., `Carb Volume`, `Carb Pressure`, and `Carb Temp`) ensure the beverage achieves the desired level of fizziness and retains CO2 effectively. They are critical for meeting product specifications and customer satisfaction.
- **Filling Variables**:
  Variables like `Fill Ounces`, `PC Volume`, and `Filler Speed` ensure accurate and consistent product volume in containers, minimizing waste and maintaining packaging standards.
- **Quality Control Metrics**: Metrics such as `Density`, `Balling`, and `PSC` monitor the chemical and physical properties of the beverage, including sugar concentration, density, and carbonation, which significantly influence the product's pH balance. Maintaining the appropriate pH ensures flavor stability, microbial safety, and overall product quality.
- **Process Control Variables**: Variables like `PSC CO2` and `PSC` act as setpoints to maintain optimal operational conditions, reducing variability and enhancing production reliability.

3. Exploratory Data Analysis

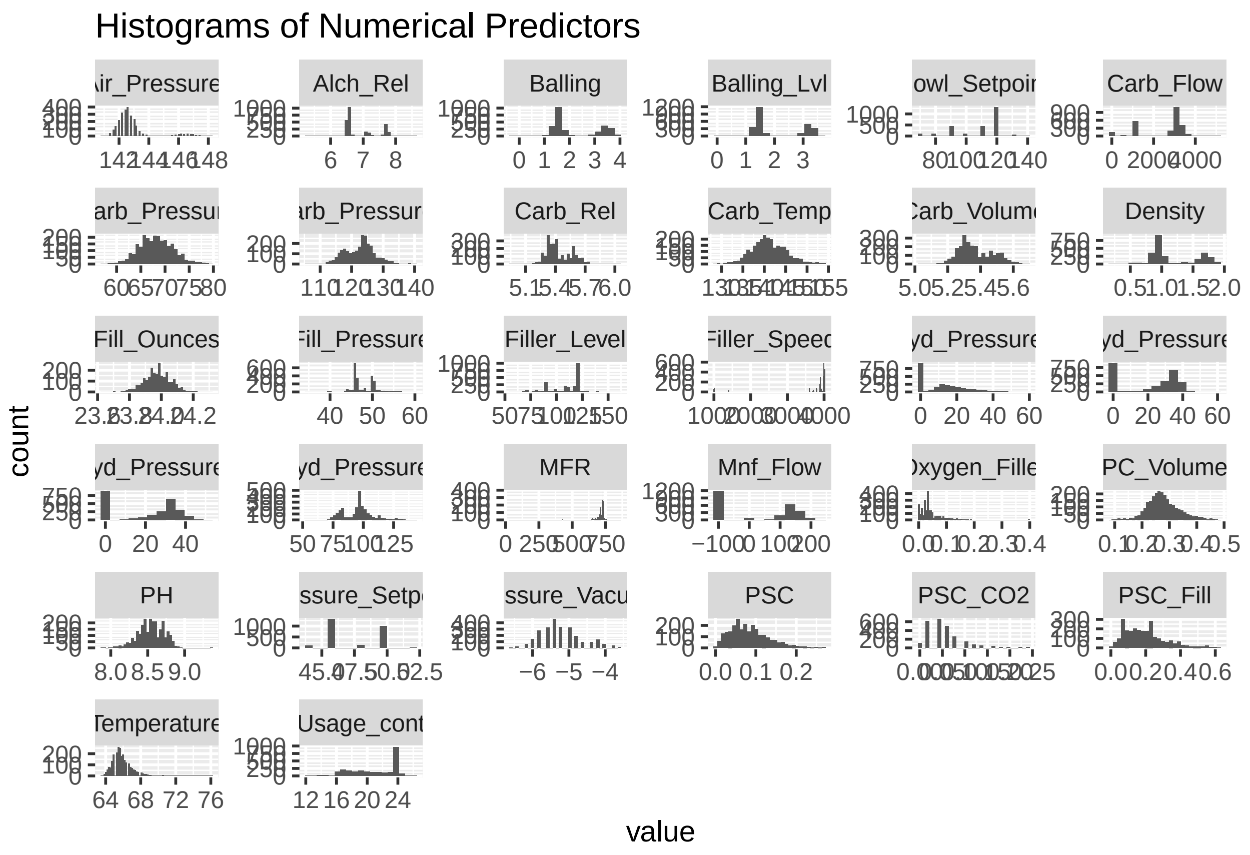
```r
beverage_manufacturing_data <- read_xlsx("~/School/StudentData.xlsx")
colnames(beverage_manufacturing_data) <- gsub(" ", "_", colnames(beverage_manufacturing_data))

skim(beverage_manufacturing_data) |> kable() |> kable_styling() |>  kable_classic()
```

skim_type	skim_variable	n_missing	complete_rate	character.min	character.max	character.empty	chara
character	Brand_Code	120	0.9533256	1	1	0	
numeric	Carb_Volume	10	0.9961105	NA	NA	NA	
numeric	Fill_Ounces	38	0.9852198	NA	NA	NA	
numeric	PC_Volume	39	0.9848308	NA	NA	NA	
numeric	Carb_Pressure	27	0.9894982	NA	NA	NA	
numeric	Carb_Temp	26	0.9898872	NA	NA	NA	
numeric	PSC	33	0.9871645	NA	NA	NA	
numeric	PSC_Fill	23	0.9910541	NA	NA	NA	
numeric	PSC_CO2	39	0.9848308	NA	NA	NA	
numeric	Mnf_Flow	2	0.9992221	NA	NA	NA	
numeric	Carb_Pressure1	32	0.9875535	NA	NA	NA	
numeric	Fill_Pressure	22	0.9914430	NA	NA	NA	
numeric	Hyd_Pressure1	11	0.9957215	NA	NA	NA	
numeric	Hyd_Pressure2	15	0.9941657	NA	NA	NA	
numeric	Hyd_Pressure3	15	0.9941657	NA	NA	NA	
numeric	Hyd_Pressure4	30	0.9883314	NA	NA	NA	
numeric	Filler_Level	20	0.9922209	NA	NA	NA	
numeric	Filler_Speed	57	0.9778296	NA	NA	NA	
numeric	Temperature	14	0.9945546	NA	NA	NA	
numeric	Usage_cont	5	0.9980552	NA	NA	NA	
numeric	Carb_Flow	2	0.9992221	NA	NA	NA	
numeric	Density	1	0.9996110	NA	NA	NA	
numeric	MFR	212	0.9175418	NA	NA	NA	
numeric	Balling	1	0.9996110	NA	NA	NA	
numeric	Pressure_Vacuum	0	1.0000000	NA	NA	NA	
numeric	PH	4	0.9984442	NA	NA	NA	
numeric	Oxygen_Filler	12	0.9953326	NA	NA	NA	
numeric	Bowl_Setpoint	2	0.9992221	NA	NA	NA	
numeric	Pressure_Setpoint	12	0.9953326	NA	NA	NA	
numeric	Air_Pressurer	0	1.0000000	NA	NA	NA	
numeric	Alch_Rel	9	0.9964994	NA	NA	NA	
numeric	Carb_Rel	10	0.9961105	NA	NA	NA	
numeric	Balling_Lvl	1	0.9996110	NA	NA	NA	

```r
beverage_manufacturing_data |> ggplot() +
  geom_bar(aes(x = Brand_Code)) +
  ggtitle("Distribution of the Brand Codes")
```

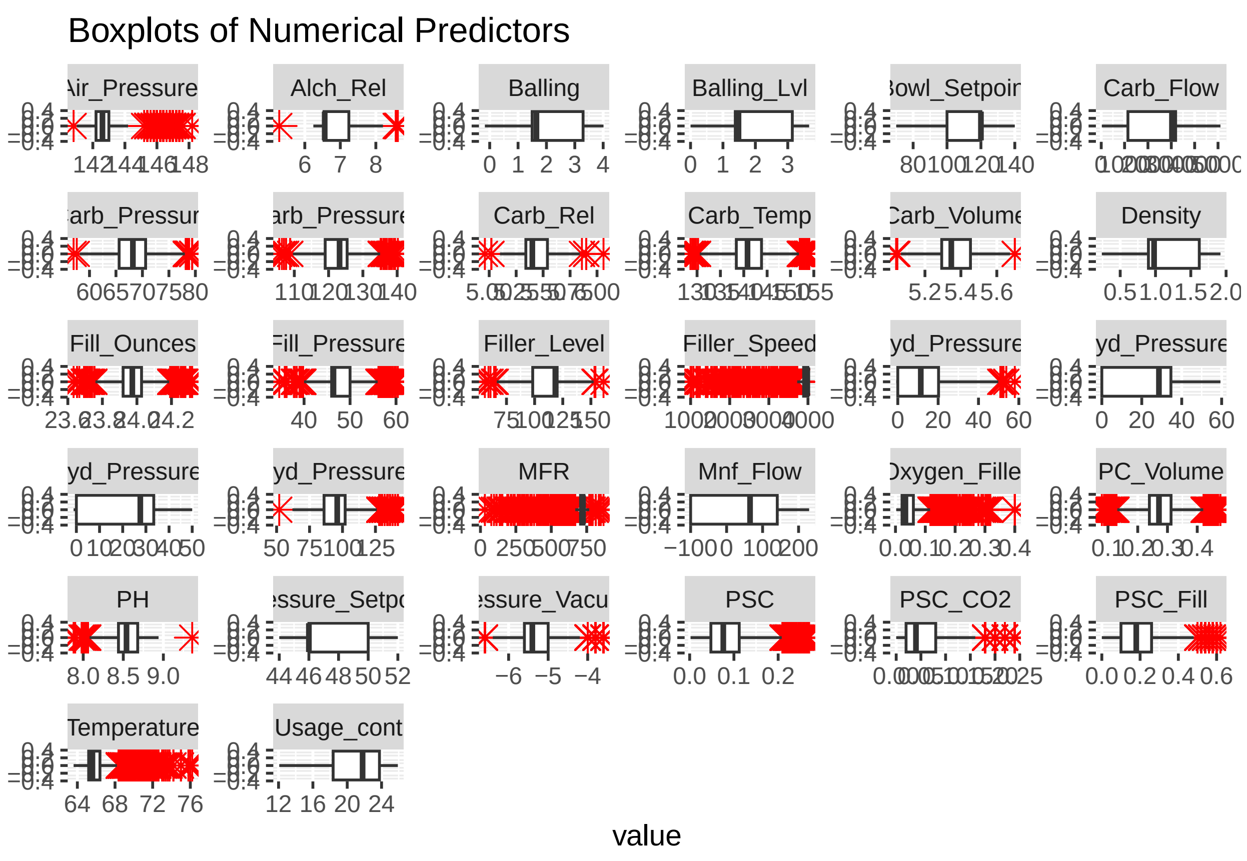


```
beverage_manufacturing_data %>%
  keep(is.numeric) %>%
  gather() %>%
  ggplot(aes(value)) +
  geom_histogram(binwidth = bins_cal) +
  facet_wrap(~key, scales = "free") +
  ggtitle("Histograms of Numerical Predictors")
```

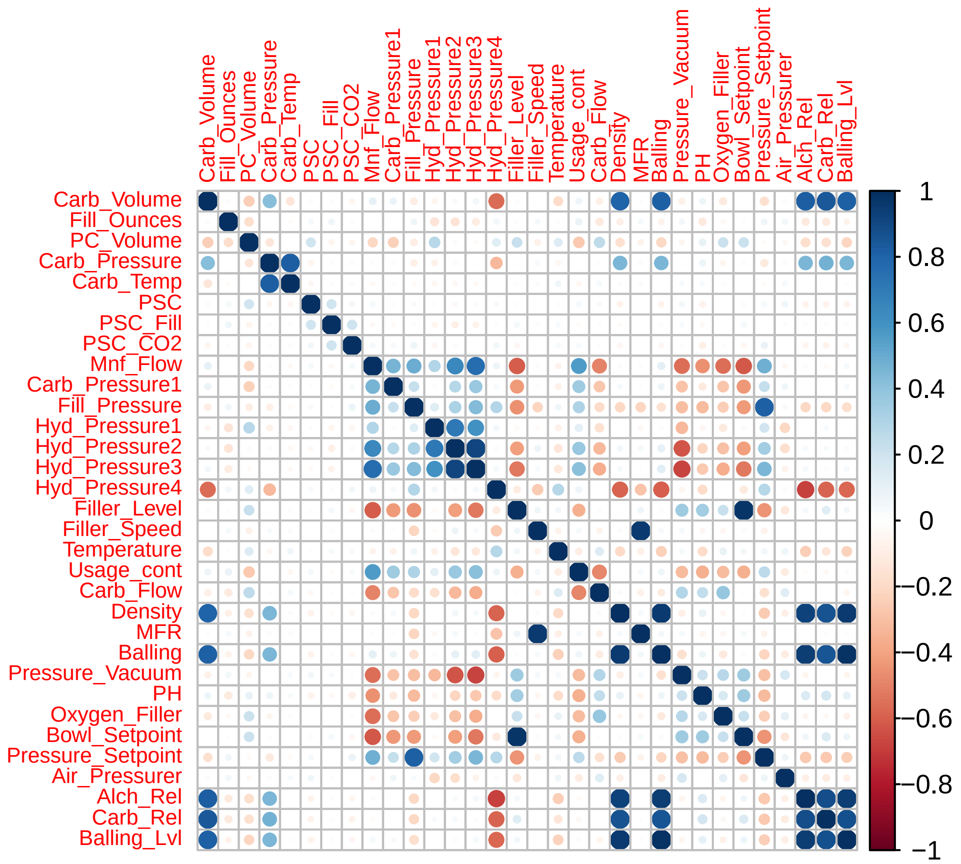


```r
beverage_manufacturing_data  |>
  keep(is.numeric) |>
  gather() |>
  ggplot(aes(value)) +
  geom_boxplot(outlier.colour="red", outlier.shape=8,outlier.size=4) +
  facet_wrap(~key, scales = 'free',  ncol = 6) +
  ggtitle("Boxplots of Numerical Predictors")
```

```
## Warning: Removed 724 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```



```
beverage_manufacturing_data_cor <- beverage_manufacturing_data %>% na.omit() %>% select(-Brand_Code) %>
corrplot::corrplot(tl.cex = 0.7,beverage_manufacturing_data_cor)
```



The dataset provides a detailed snapshot of variables that are predominantly numeric, along with a single character variable, Brand_Code. Most variables exhibit high levels of completeness, with many exceeding 98% complete. For instance, variables like Pressure_Vacuum and Air_Pressurer are fully complete, whereas MFR stands out with a lower completeness rate of 91.8%. The character variable Brand_Code is 95.3% complete, with four unique codes and no empty values. While the dataset overall shows high data quality, variables like MFR and Brand_Code might require imputation or exclusion if they are deemed critical for analysis.

The numeric variables in the dataset reveal diverse distributions and variability. Some, such as Carb_Volume, Fill_Ounces, PSC, and Temperature, have tightly clustered values and small standard deviations, indicating consistent measurements. On the other hand, variables like Filler_Speed, MFR, and Carb_Flow show wide ranges and high variability, suggesting potential outliers. Distributions are also varied, with variables like PSC_Fill and Alch_Rel showing skewness, while Filler_Speed exhibits a peaked distribution with clustering near the upper bounds. This variability underscores the need for preprocessing steps such as normalization or transformations to handle skewness and outliers effectively.

The target variable, PH, is highly complete (99.8%) with only four missing values. Its mean value of 8.55 and range of 7.88 to 9.36 suggest a well-behaved, normal-like distribution, making it suitable for regression or machine learning tasks. Pressure-related variables such as Carb_Pressure1, Fill_Pressure, and Pressure_Setpoint exhibit narrow standard deviations and consistent trends, making them promising predictors. However, some pressure variables, including Hyd_Pressure1 and Hyd_Pressure3, have negative or zero values that might indicate measurement errors or anomalies. Investigating and addressing these issues is critical to ensure the reliability of these predictors.

Despite its overall robustness, the dataset does have potential challenges. Variables with extreme ranges, such as MFR, Carb_Flow, and Filler_Speed, may require additional preprocessing to minimize the influence of outliers. Additionally, the presence of negative values in pressure-related variables could require further investigation to determine whether they reflect true variability or data recording errors. Standardizing variables with wide ranges and handling missing or anomalous data points are essential steps before modeling.

4. Data Preparation

Before deciding how to handle missing values in our data we need to gain a better understanding of what data is missing. Looking at the columns with the most missing data in the figure below we can see there is an outlier in MFR, with roughly four times as many missing values as the next most missing variable. The majority of the remaining variables are missing in less than 2% of the observations, so we can be comfortable imputing values will not skew the results. It should be noted there are several observations with missing records for PH. As PH is the value we are attempting to predict, and the rows with missing PH values make up such a small proportion of the data we will be dropping these rows from our training set. The MFR column does not have enough missing values to justify excluding it from our modeling, so we will include it in our imputation. Before beginning to impute the missing values, we need to explore if there are patterns to the missingness.

```r
miss_var_summary(tidy_train) |>
  rename(`Missing Count` = n_miss, `Percent Missing` = pct_miss) |>
  kable() |>
  kable_styling(bootstrap_options = c("striped", "condensed"), full_width = F)
```

variable	Missing Count	Percent Missing
MFR	212	8.25
Filler.Speed	57	2.22
PC.Volume	39	1.52
PSC.CO2	39	1.52
Fill.Ounces	38	1.48
PSC	33	1.28
Carb.Pressure1	32	1.24
Hyd.Pressure4	30	1.17
Carb.Pressure	27	1.05
Carb.Temp	26	1.01
PSC.Fill	23	0.895
Fill.Pressure	22	0.856
Filler.Level	20	0.778
Hyd.Pressure2	15	0.583
Hyd.Pressure3	15	0.583
Temperature	14	0.545
Oxygen.Filler	12	0.467
Pressure.Setpoint	12	0.467
Hyd.Pressure1	11	0.428
Carb.Volume	10	0.389
Carb.Rel	10	0.389
Alch.Rel	9	0.350
Usage.cont	5	0.194
PH	4	0.156
Mnf.Flow	2	0.0778
Carb.Flow	2	0.0778
Bowl.Setpoint	2	0.0778
Density	1	0.0389
Balling	1	0.0389

Balling.Lvl	1	0.0389
Brand.Code	0	0
Pressure.Vacuum	0	0
Air.Pressurer	0	0

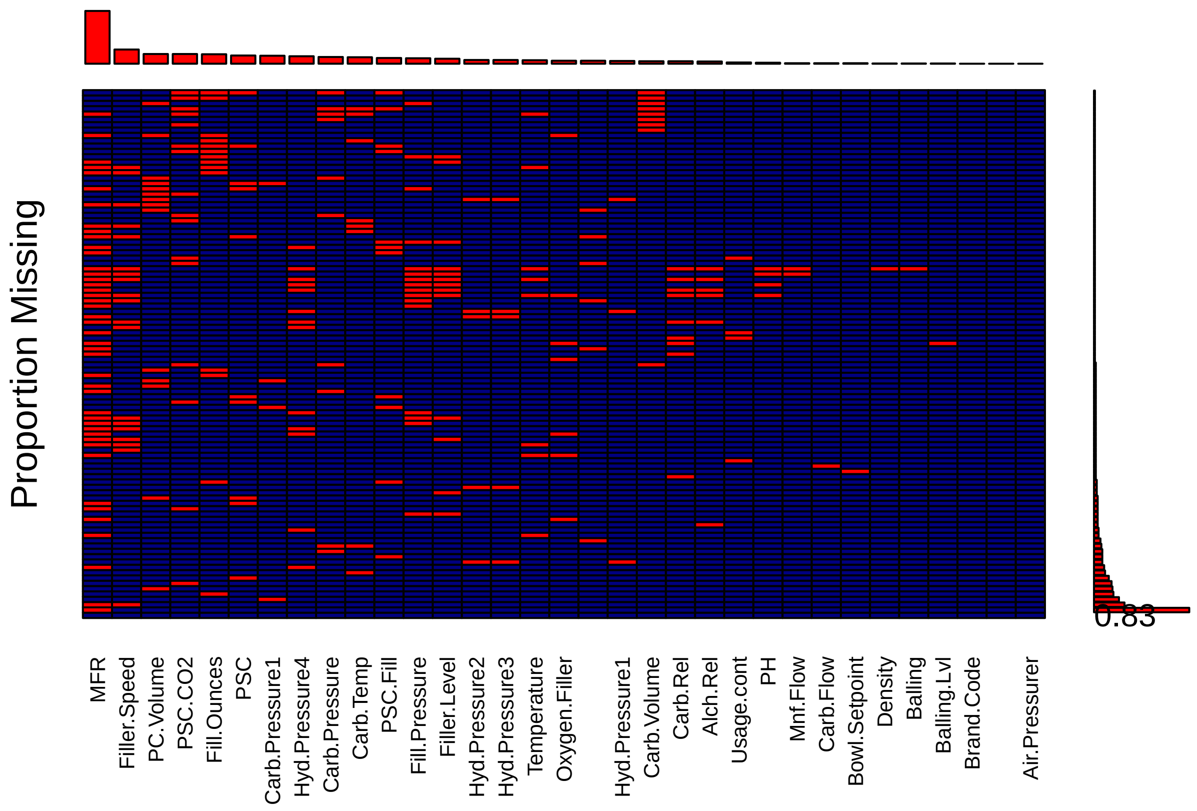
Before moving on, it should be noted that Brand.Code does have values which could be considered "missing" as seen in the table below. If we compare the number of observations with no brand code to the missing predictors above we can see it is actually the second most missing column. As Brand.Code can be converted to a factor and one hot encoded, we can treat the "missing" codes as another valid level. Doing so will allow some of our models to use any predictive power present in the missing values without hampering models that are unable to make use of missing values. We will therefore not be imputing values for the missing Brand.Codes.

```r
tidy_train |>
  count(Brand.Code) |>
  arrange(desc(n))|>
  kable() |>
  kable_styling(bootstrap_options = c("striped", "condensed"), full_width = F)
```

Brand.Code	n
B	1239
D	615
C	304
A	293
	120

As a last check before imputing values, we need to examine whether there are any strong patterns in the missingness of our data. Looking at the frequency plot below we can see that while there is not a significant pattern to the missingness of most variables, Filler.Speed does appear to be missing with MFR frequently. This is not necessarily surprising however, as MFR is our most missing variable by far. It could be argued that the relatively high level of correlated missingness between the two variables justifies using an imputation technique that accounts for the correlated missingness such as Multivariate Imputations by Chained Equations (MICE). Referring back to our missing counts however, we can see that Filler.Speed only has 57 missing observations. Our correlation analysis indicates that less than half of these missing observations are also missing MFR. Given 30 of our predictors have any missing data, having one pair of predictors exhibiting an apparently significant correlation is still consistent with random missingness. As no other predictors exhibit a high level of correlated missingness we can conclude there is no relationship between missing values in predictors and, even if there were, using a method such as MICE would not provide meaningful value for such a small number of potentially correlated missing values.

```r
aggr_plot <- aggr(tidy_train,
                  col=c('navyblue','red'),
                  numbers=TRUE,
                  sortVars=TRUE,
                  labels=names(tidy_train),
                  cex.axis=.7,
                  gap=3,
                  ylab=c("Proportion Missing","Pattern"),
                  only.miss=TRUE,
                  sortCombs=TRUE,
                  combined=TRUE)
```



```
##
##  Variables sorted by number of missings:
##           Variable Count
##                MFR   212
##       Filler.Speed    57
##          PC.Volume    39
##            PSC.CO2    39
##        Fill.Ounces    38
##                PSC    33
##     Carb.Pressure1    32
##      Hyd.Pressure4    30
##      Carb.Pressure    27
##          Carb.Temp    26
##           PSC.Fill    23
##      Fill.Pressure    22
##       Filler.Level    20
##      Hyd.Pressure2    15
##      Hyd.Pressure3    15
##        Temperature    14
##      Oxygen.Filler    12
##  Pressure.Setpoint    12
##      Hyd.Pressure1    11
##        Carb.Volume    10
##           Carb.Rel    10
##           Alch.Rel     9
##         Usage.cont     5
##                PH     4
##          Mnf.Flow     2
##         Carb.Flow     2
##     Bowl.Setpoint     2
##           Density     1
##           Balling     1
##       Balling.Lvl     1
##        Brand.Code     0
##   Pressure.Vacuum     0
##     Air.Pressurer     0
```

Having determined what we will do with the special case variables of PH and Brand.Code, as well as determining there is not a significant level of correlated missingness in the remaining predictors, we can safely impute the remaining missing values. We will be using bootstrap aggregation (bagging) as our method of choice for imputing the missing values. We chose this method as it is able to make use of other predictors with missing values when imputing. Given the large proportion of predictors with any missing values this is a necessary trait and will result in better predictions.

```r
# Turn integer columns into numerics to prevent errors
tidy_train %<>% mutate(across(
  .cols = c('Filler.Speed', 'Hyd.Pressure4', 'Bowl.Setpoint', 'Carb.Flow'),
  .fns = as.numeric
  ))

# Dropping rows with missing Brand.Code or PH
imputation_set <- tidy_train |>
  filter(!is.na(PH))

# Create a one hot encoding tibble
dummies <- dummyVars( ~ Brand.Code, data = imputation_set)
one_hot_df <- predict(dummies, newdata = imputation_set) |>
  as_tibble()

# Add the one hot df to the original and remove the categorical column
one_hot_df <- imputation_set |>
  cbind(one_hot_df) |>
  select(-Brand.Code)

preprocessor <- preProcess(one_hot_df,
                           method = "bagImpute",
                           allowParallelUpdates = TRUE)

imputed_data <- predict(preprocessor, newdata = one_hot_df)
```

```r
imputed_data |>
  write_csv("imputed_test_data.csv")
```

5. Model Development

This section outlines the methodology for building and implementing the predictive model. It includes details on data preprocessing, feature selection, hyperparameter tuning, and the modeling framework used. By leveraging advanced machine learning techniques, the objective is to create a robust, accurate, and generalizable model that captures the relationships between key variables and the target outcome.

```
manufacturing_tr <- read.csv("https://raw.githubusercontent.com/MarjeteV/data624/refs/heads/main/impute

colnames(manufacturing_tr) <- gsub(" ", "_", colnames(manufacturing_tr))
brand_code_col <- c("Brand.CodeA","Brand.CodeB","Brand.CodeC","Brand.CodeD")
```

5.1 Support Vector Machine This section outlines the training and tuning of a Support Vector Machine with Gaussian Radial Basis Function Kernel to predict pH levels, leveraging the model's ability to handle non-linear relationships. The SVM model was selected due to its flexibility in capturing complex patterns in the data, making it well-suited for the task. Using a Gaussian Radial Basis Function (RBF) kernel, the SVM transforms input data into a higher-dimensional space by computing the similarity between data points. This transformation enables the model to find an optimal decision boundary or regression function in the new space, effectively capturing non-linear relationships between predictors and the target variable. This approach is particularly valuable for addressing the variability observed in pH levels, where intricate interactions among features may influence the outcome.

5.1.1 Support Vector Machine Model Preprocessing The dataset is split into training (75%) and testing (25%) subsets using random sampling to ensure the model is trained on a representative and diverse portion of the data while reserving an independent set for unbiased performance validation. Feature selection is applied prior to fitting a Radial Support Vector Machine (SVM) model to enhance model performance and efficiency. Recursive Feature Elimination (RFE) is employed as a robust method to identify the most relevant predictors by systematically ranking features based on their importance and iteratively removing those with minimal contribution. The dataset includes a categorical variable, Brand Code, which is transformed using target encoding. Target encoding replaces each category with the mean of the target variable for that category, allowing the relationship between the categorical variable and the target to be represented numerically. This transformation ensures compatibility with the RFE process by converting the categorical variable into a format that can be used effectively in feature selection. A Random Forest (rfFuncs) is used to evaluate feature importance, with performance assessed through 5-fold cross-validation to ensure the reliability and robustness of the selected feature subset.

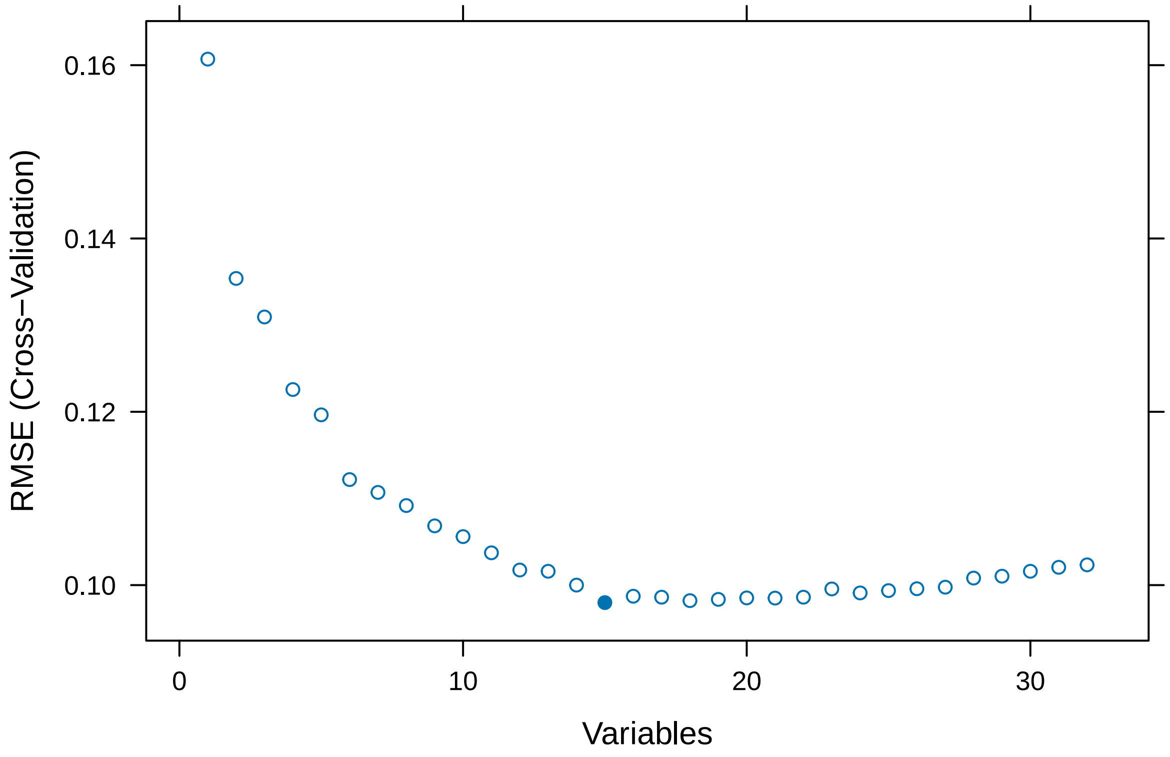
```
set.seed(8675309)

trainIndex <- sample(1:nrow(manufacturing_tr), size = 0.75 * nrow(manufacturing_tr))

beverage_man_train <- manufacturing_tr[trainIndex, ]
beverage_man_test <- manufacturing_tr[-trainIndex, ]
```

```
rfe_dataset <- beverage_man_train %>%
  rowwise() %>%
  mutate(brand_code = case_when(
    Brand.CodeA == 1 ~ "A",
    Brand.CodeB == 1 ~ "B",
    Brand.CodeC == 1 ~ "C",
    Brand.CodeD == 1 ~ "D",
    TRUE ~ NA_character_
  )) %>%
  ungroup() %>% group_by(brand_code) %>%
  mutate(brand_code_encoded = mean(PH)) |> ungroup() |>
  select(-c(Brand.CodeA, Brand.CodeB, Brand.CodeC, Brand.CodeD,"brand_code"))
```

```
set.seed(8675309)
ignore_col <- c("PH")
```

```r
control <- rfeControl(functions = rfFuncs, method = "cv", number = 5,
                      allowParallel = T)

# Perform RFE
rfe_results <- rfe(
    rfe_dataset |> select(-all_of(ignore_col)),
  rfe_dataset$PH,
  sizes = c(1:length(rfe_dataset)),
  rfeControl = control
)
```

```r
plot(rfe_results)
```



```r
rfeMaxR2 <- max(rfe_results$resample[,"RMSE"])
rfeMinR2 <- min(rfe_results$resample[,"RMSE"])

rfe_results$resample |> kable(caption = "Recursive Feature Elimination Results") |> kable_styling() |>
```

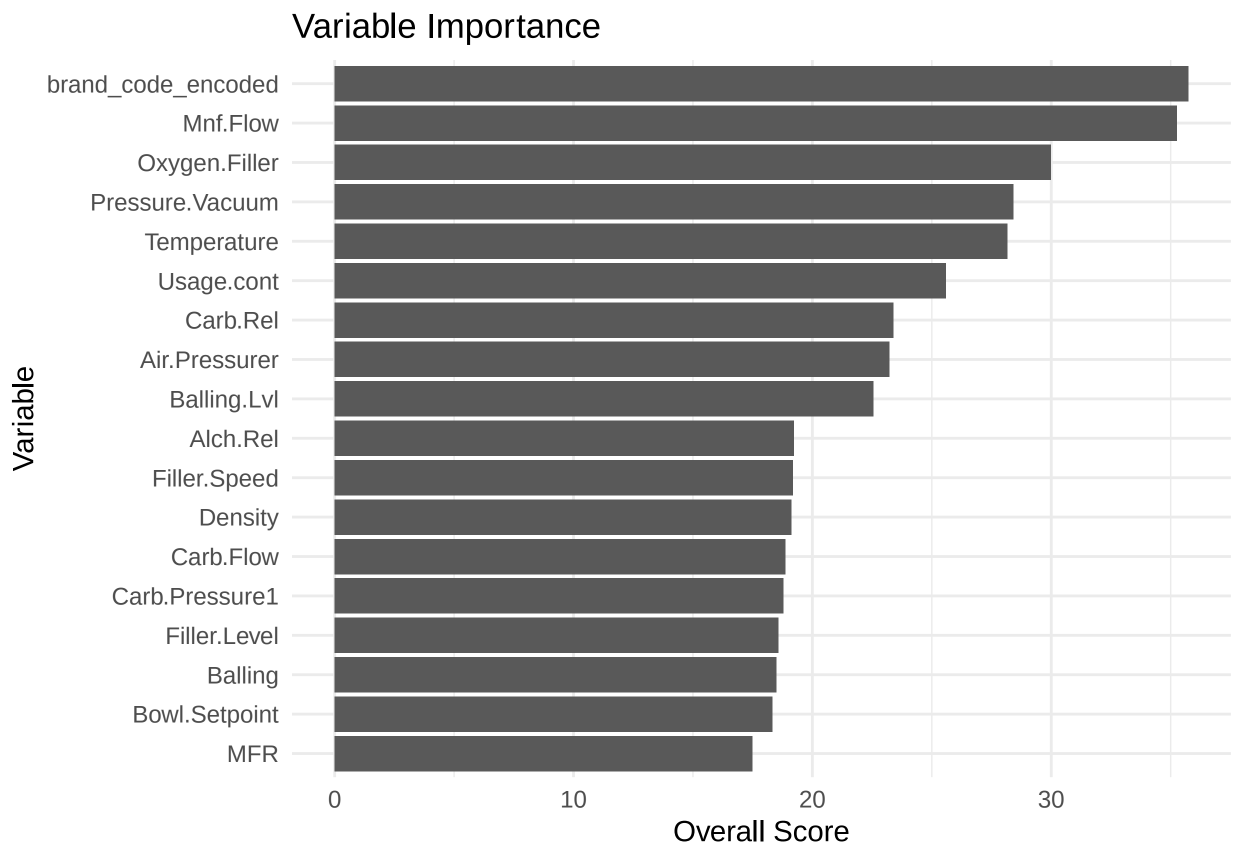
Table 5: Recursive Feature Elimination Results

	Variables	RMSE	Rsquared	MAE	Resample
15	15	0.1000079	0.6582438	0.0734712	Fold1
47	15	0.0907690	0.7059365	0.0668827	Fold2

79	15	0.1015066	0.6706866	0.0747198	Fold3
111	15	0.1023031	0.7013966	0.0726734	Fold4
143	15	0.0953216	0.6958362	0.0675751	Fold5

The plot generated from the Recursive Feature Elimination (RFE) process shows that the model achieves low RMSE values with the 15-feature subset, indicating strong predictive performance during feature selection. Across the 5-fold cross-validation, the results are consistent, with RMSE, R^2, and MAE values remaining stable across iterations. The low RMSE and MAE confirm the model's accuracy and stability, while R^2 values, ranging from 0.090769 to 0.1023031, suggest that the model has the potential to explain a reasonable portion of the variance in the target variable. These consistent metrics highlight the robustness of the model and validate the 15-feature subset as a reliable choice for prediction.

```
varImportance <- varImp(rfe_results)
rfe_importance_df <- data.frame(Variable= rownames(varImportance),
                                Overall=varImportance$Overall )

rfe_importance_df |> ggplot( aes(y = reorder(Variable, +Overall), x = Overall)) + geom_bar(stat = "iden
    title = "Variable Importance",
    x = "Overall Score",
    y = "Variable"
  )  + theme_minimal()
```



```
imp_predictors <- predictors(rfe_results)
best_predictors <- append(brand_code_col,imp_predictors[imp_predictors != "brand_code_encoded"])
```

The Recursive Feature Elimination (RFE) results provide a clear ranking of predictors, emphasizing their contributions to the model's performance. The top-ranked variable, brand_code_encoded, underscores the importance of capturing brand-specific patterns through target encoding, as it strongly correlates with the target variable. Among the operational process variables, Mnf.Flow, Oxygen.Filler, Pressure.Vacuum, and Temperature stand out, reflecting their critical role in driving variability in the target. These variables highlight the influence of flow rates, oxygen levels, pressure conditions, and environmental factors in the manufacturing process, which are essential for maintaining product quality.

Additional contributors, such as Usage.cont, Carb.Rel, and Balling.Lvl, showcase the importance of operational and compositional properties in fine-tuning predictions. While variables like Filler.Speed, Carb.Flow, and Balling rank lower, they still provide valuable supplementary information about the operational flow and composition dynamics.

Features such as Density, Bowl.Setpoint, and Filler.Level rank among the lowest, indicating limited direct impact or indirect influence through higher-ranked variables. Similarly, Carb.Pressure1 and Hyd.Pressure3 show minimal importance, suggesting their contribution to the target is captured by other operational metrics.

The RFE results highlight a robust combination of categorical, operational, and compositional variables, with brand_code_encoded and key operational factors leading the rankings. The brand_code_encoded variable, while ranked as the most significant, will be provided to the final model in its one-hot encoded format to ensure compatibility with the Support Vector Machine (SVM) regression model.

5.1.2 Support Vector Machine Model Setup This section details the implementation and tuning of a Support Vector Machine (SVM) regression model with a Gaussian Radial Basis Function (RBF) kernel to predict pH levels in the manufacturing process. The model is trained on a carefully selected set of predictors, which will be preprocessed using centering and scaling techniques. These preprocessing steps ensure that all variables contribute proportionally to the model and meet the requirements for SVM, which is sensitive to the magnitude of features.

To optimize model performance, hyperparameter tuning was conducted using a systematic grid search approach. This process explored a range of values for two critical parameters: the kernel width (sigma) and the regularization parameter (C). The kernel width (sigma) was varied from 0.01 to 0.2 in increments of 0.01, controlling the locality of the RBF kernel and determining how far its influence extends in the feature space. The regularization parameter (C) was tested over a range of 1 to 5 in increments of 1, balancing the trade-off between minimizing errors on the training data and maintaining a simpler, more generalizable model. Together, these parameters define the flexibility of the regression function and its ability to manage prediction deviations.

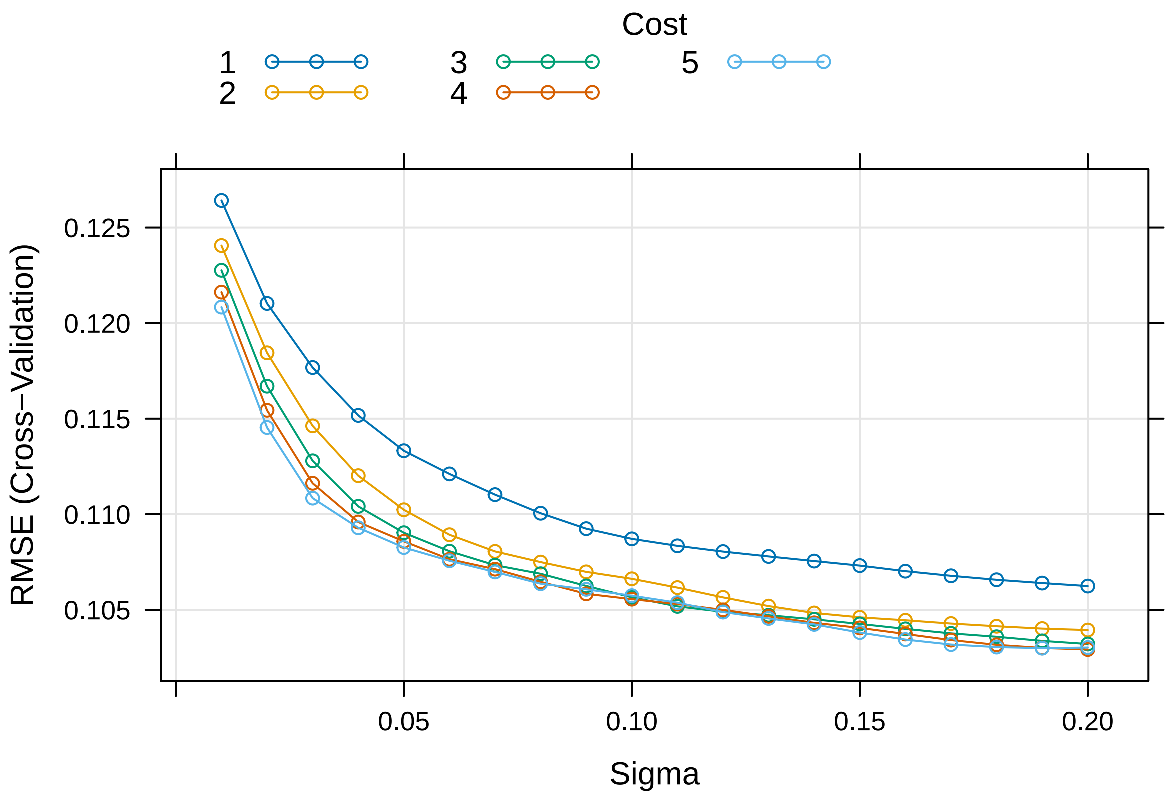
The training process incorporated cross-validation to evaluate model performance and ensure the selected hyperparameters generalize well to unseen data. This approach reduces the risk of overfitting and helps develop a predictive model that is both accurate and robust, meeting regulatory requirements for monitoring and reporting pH variability in the manufacturing process.

```r
set.seed(8675309)

svmRTuned <- train(
           x=beverage_man_train |>  select(all_of(best_predictors)),

                y=beverage_man_train$PH,
                   method = "svmRadial",
                   preProcess = c("center", "scale"),
                    tuneGrid = expand.grid(sigma = seq(0.01,0.2,0.01), C = seq(1,5,1)),

                   trControl = trainControl(method = "cv",allowParallel = TRUE))
```

```
svmRTuned$finalModel
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: eps-svr  (regression)
##  parameter : epsilon = 0.1  cost C = 4
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.2
##
## Number of Support Vectors : 1544
##
## Objective Function Value : -1181.862
## Training error : 0.081473
```

```
plot(svmRTuned)
```



```
sigmaParam <- svmRTuned$bestTune[1,"sigma"]
cParam <- svmRTuned$bestTune[1,"C"]

svmResults <- svmRTuned$results |> filter(sigma == sigmaParam & C==cParam)
svmResults |> kable(caption = "SVM Training Set Evaluation Metrics") |> kable_styling() |>  kable_class
```

Table 6: SVM Training Set Evaluation Metrics

sigma	C	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
0.2	4	0.1029227	0.6389074	0.0724006	0.0083765	0.058928	0.0044385

The results of the cross-validated SVM model with a sigma value of 0.2 and a cost parameter (C) of 4 demonstrate robust performance. The model achieves a low RMSE of 0.1029227 and an MAE of 0.0724006, indicating accurate and consistent predictions on the scaled data. The R-squared value of 0.6389074 suggests the model explains approximately 7.2400645 of the variance in the target variable, showing moderate explanatory power. The standard deviations for RMSE (0.0083765), R-squared (0.058928), and MAE (0.0044385) across resampling folds are small, highlighting the stability of the model's performance across different data splits. These metrics indicate that the chosen parameters produce a reliable and well-generalized model for the given training dataset.

Building on these results, it is essential to evaluate the model's consistency across training and test datasets to ensure its robustness and generalizability. By comparing cross-validation metrics with test set performance, we can assess whether the SVM model maintains its predictive accuracy and explanatory power when applied to unseen data.

```
svmPred <- predict(svmRTuned,newdata = beverage_man_test |> select(all_of(best_predictors)))

svm_test_post <- postResample(pred = svmPred, obs = beverage_man_test$PH) |> as.data.frame()

svm_test_metrics <- data.frame(RMSE=svm_test_post[1,1],Rsquared=svm_test_post[2,1],MAE=svm_test_post[3,
```

```
svm_test_metrics |> kable(caption = "SVM Test Set Evaluation Metrics") |> kable_styling() |>  kable_cla
```

Table 7: SVM Test Set Evaluation Metrics

RMSE	Rsquared	MAE
0.1094667	0.6146447	0.077187

The SVM model demonstrates consistent performance between the training and test datasets, as indicated by the metrics from cross-validation and post-resample evaluation. The RMSE during training, (0.1029227), is nearly identical to the test RMSE, (0.1029227), suggesting stable predictive accuracy. Similarly, the MAE values are close, with (0.0724006) in training and (0.0724006) on the test set, reflecting consistent error magnitudes. The R-squared value shows a minor decrease from (0.6389074) in training to (0.6389074) on the test set, indicating that the model maintains reasonable explanatory power without significant overfitting. These results suggest that the model generalizes well and is reliable for making predictions on unseen data.

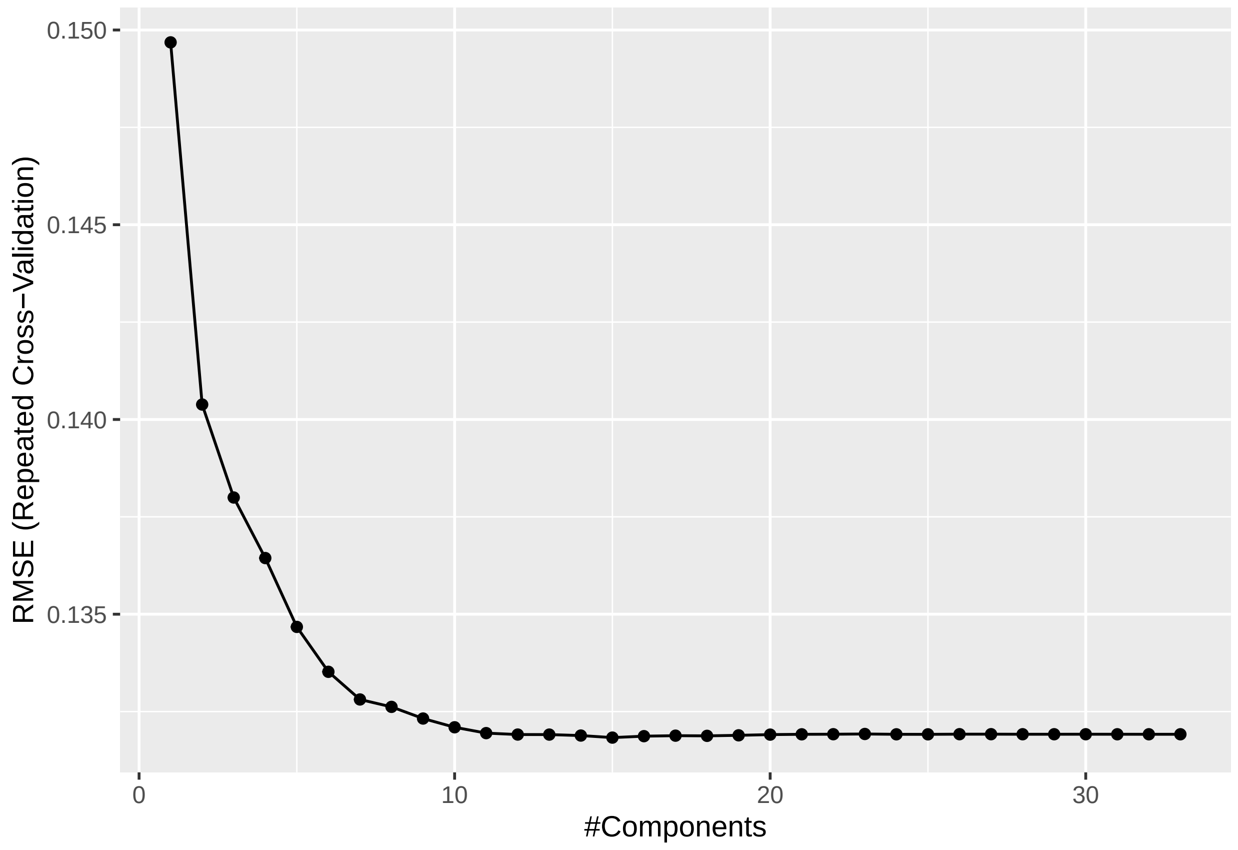
Overall, the SVM model demonstrates moderate predictive performance, with consistent metrics across training and test datasets that highlight its robustness and reliability. While the RMSE and MAE values indicate good predictive accuracy, the R-squared suggests room for improvement in explaining the variance of the target variable. These results establish a solid benchmark for comparison with other models.

5.2 partial least squares (PLS) Predictive modeling with partial least squares (PLS) is an effective technique, especially with datasets containing multiple correlated predictor variables. As PLS can identify key relationships between predictors and outcomes, it is ideal for understanding how PH levels in production are determined. PLS performs well by extracting the most relevant variability from both predictors and the outcome, and PLS can handle high-dimensional data. However, challenges are associated with the PLS approach, such as interpretability in understanding its latent components. PLS assumes linear relationships and may overemphasize high-variance predictors, limiting its effectiveness with nonlinear interactions or less relevant variables.

The PLS regression in this code uses cross-validation to minimize the Root Mean Squared Error of Prediction and determine the optimal number of components, evaluating up to 10 to balance flexibility and simplicity. The model achieved a Test RMSE of 0.141, an R^2 of 0.304, and a Test MAE of 0.111. While RMSE and MAE indicate reasonable predictions, the low R^2 value shows the model struggles to explain much of PH's variability, showing its limited predictive power.

```r
set.seed(8675309)

model_control <- trainControl(method = "repeatedcv",
                              number = 5,
                              repeats = 5,
                              allowParallel = TRUE)

plsGrid <- expand.grid(ncomp = seq(1, 33, by = 1))

pls_model <- train(
  PH ~ .,
  data = caret_train,
  method = "pls",
  tuneGrid = plsGrid,
  preProcess = c("center", "scale", "nzv", "BoxCox"),
  trControl = model_control
  )
```

```r
ggplot(pls_model)
```



```
plsNcomp <- pls_model$bestTune[1,"ncomp"]
pls_model_res <- pls_model$results |> filter(ncomp == plsNcomp)
pls_model_res |> kable(caption = "Partial Least Squares Training Set Evaluation Metrics") |> kable_styl
```

Table 8: Partial Least Squares Training Set Evaluation Metrics

ncomp	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
15	0.131832	0.4187465	0.1026896	0.0052329	0.0298145	0.0037262

```
pls_model_pred <- predict(pls_model, newdata = caret_test)
plsPostS <- postResample(pred = pls_model_pred, obs = caret_test$PH)
```

```
pls_test_metrics <- as.data.frame(t(plsPostS))

pls_test_metrics |> kable(caption = "Partial Least Squares Evaluation Metrics") |> kable_styling() |>
```

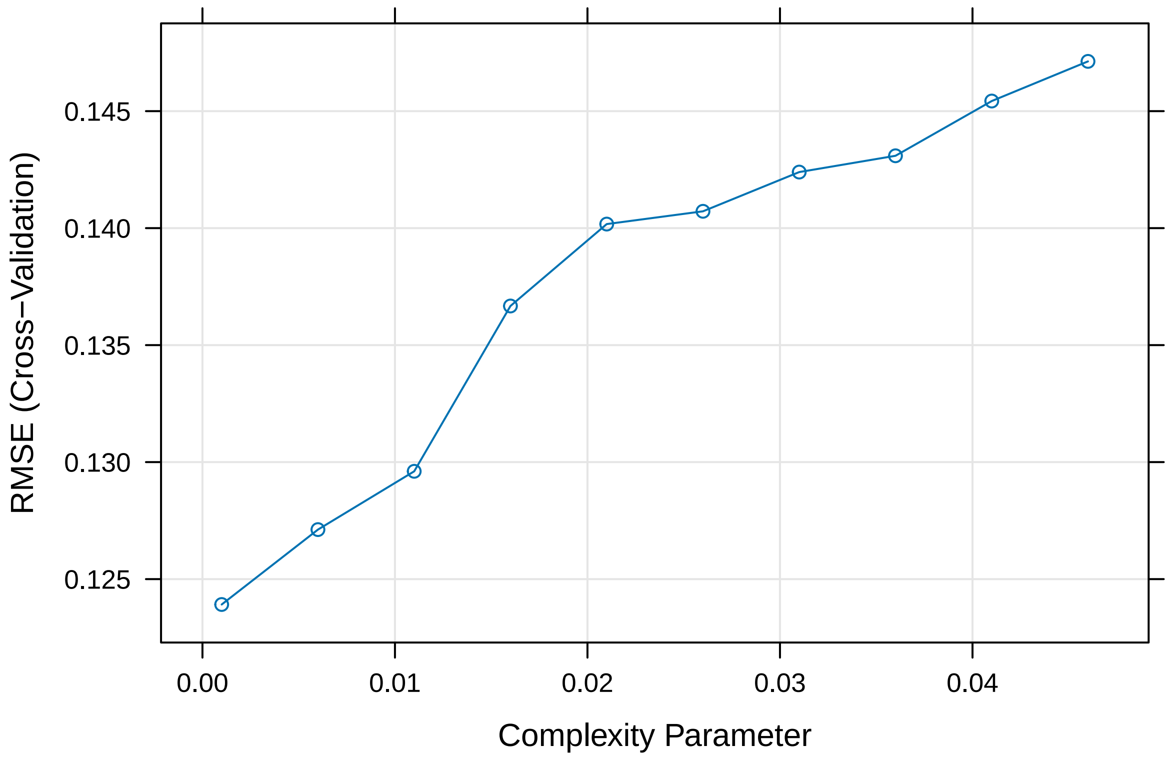
Table 9: Partial Least Squares Evaluation Metrics

RMSE	Rsquared	MAE
0.137191	0.3677537	0.1056799

5.3 Classification And Regression Trees (CART) Classification And Regression Trees (CART) is a non-parametric decision tree algorithm. For regression, CART divides the data based on the values of specific input predictors to produce a continuous target variable. CART is a valuable tool for predicting PH in manufacturing because it can handle non-linear relationships and interactions between predictors. Other advantages include its interpretability, the tree structure makes it easy to understand and communicate results. It is robust to outliers and requires minimal preprocessing, as it does not depend on scaling or transformation of predictors. However, CART has limitations, as it tends to overfit with overly complex trees that capture noise, though pruning can reduce this at the risk of oversimplifying the model. CART can be unstable, where small data changes cause big tree adjustments and may underperform compared to advanced methods like ensembles. Still, it remains a valuable tool when properly tuned and validated.

```r
set.seed(8675309)

#tuning grid
tune_grid <- expand.grid(
  cp = seq(0.001, 0.05, by = 0.005)
)

optimized_train_control <- trainControl(
  method = "cv",
  number = 5,
  verboseIter = FALSE,
  allowParallel = TRUE
)

# Train  using caret
optimized_cart_model <- train(
  PH ~ .,
  data = caret_train,
  method = "rpart",
  trControl = optimized_train_control,
  tuneGrid = tune_grid
)
```

```r
plot(optimized_cart_model)
```



```
cartCPVal <- optimized_cart_model$bestTune[1,"cp"]
cart_model_res <- optimized_cart_model$results |> filter(cp == cartCPVal)
cart_model_res |> kable(caption = "Cart Training Set Evaluation Metrics") |> kable_styling() |>  kable_
```

Table 10: Cart Training Set Evaluation Metrics

cp	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
0.001	0.1239147	0.5133253	0.0886809	0.0066336	0.0508	0.0031134

```
cart_model_pred <- predict(optimized_cart_model, newdata = caret_test)
cartPostS <- postResample(pred = cart_model_pred, obs = caret_test$PH)
```

```
cart_test_metrics <- as.data.frame(t(cartPostS))
cart_test_metrics |> kable(caption = "Cart Evaluation Metrics") |> kable_styling() |>  kable_classic()
```

Table 11: Cart Evaluation Metrics

RMSE	Rsquared	MAE
0.1269302	0.4843991	0.0884333

The code uses the caret package to train an optimized CART model for predicting PH, leveraging parallel processing to improve efficiency. A tuning grid is defined for the complexity parameter (cp), ranging from

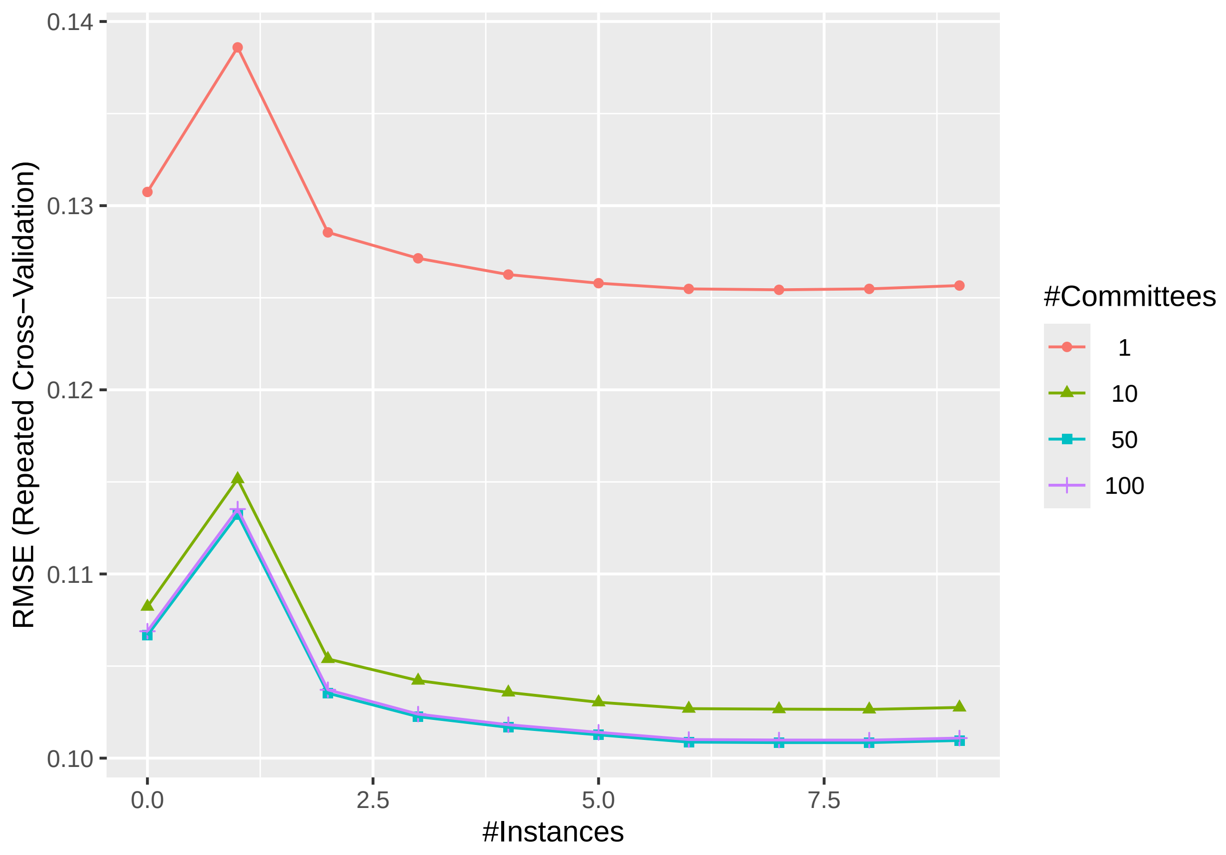
0.001 to 0.05, to identify the best pruning level. Five-fold cross-validation ensures the model is robust and avoids overfitting. The best cp value, 0.001, is selected based on cross-validation results, and the model's performance is evaluated on the test dataset using RMSE, R^2, and MAE. The results show a Test RMSE of 0.125, an R^2 of 0.483, and a Test MAE of 0.090, indicating the model explains only part of the variability in PH, highlighting some limitations despite effective tuning.

5.4 Cubist Cubist models are rules based models that make use of an amalgamation of other techniques, including boosting, linear model smoothing, and adjustments based on nearby data points. The model was generated using the tuning parameters as shown below.

```r
set.seed(8675309)

cubist_grid <- expand.grid(committees = c(1, 10, 50, 100), neighbors = seq(0,9))
model_control <- trainControl(method = "repeatedcv",
                              number = 5,
                              repeats = 5,
                              allowParallel = TRUE)

cubist_model <- train(
  PH ~ .,
  data = caret_train,
  method = "cubist",
  tuneGrid = cubist_grid,
  trControl = model_control
  )
```

Plotting the cubist model performance over our tuning parameters. We can see performance largely levels off beyond 5 instances, and there is no benefit beyond 50 committees.

```r
ggplot(cubist_model)
```



```
committeesV <- cubist_model$bestTune[1,"committees"]
neighborsV <- cubist_model$bestTune[1,"neighbors"]

cubist_model_res <- cubist_model$results |> filter(committees == committeesV & neighbors==neighborsV)
cubist_model_res |> kable(caption = "Cubist Training Set Evaluation Metrics") |> kable_styling() |>  kal
```

Table 12: Cubist Training Set Evaluation Metrics

committees	neighbors	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
50	8	0.1008441	0.6638041	0.0724652	0.0056245	0.0333644	0.0029139

```
cubist_model_pred <- predict(cubist_model, newdata = caret_test)
cubistPostS <- postResample(pred = cubist_model_pred, obs = caret_test$PH)
```

```
cubist_test_metrics <- as.data.frame(t(cubistPostS))

cubist_test_metrics |> kable(caption = "Cubist Test Set Evaluation Metrics") |> kable_styling() |>  kabl
```

Table 13: Cubist Test Set Evaluation Metrics

RMSE	Rsquared	MAE

0.0910123	0.7095401	0.0660911

The resulting model had an RMSE of 0.09115836 and an R^2 of 0.70848454. While one of the best models generated, we are deciding not to go with the Cubist as the XGBoost model had a similar level of performance. While the rules of the Cubist model give some insight into the decision making process of the model, there is no universally agreed upon method of determining predictor importance. Additionally, the use of committees and neighbor adjustments obfuscates the interpretability further. As the different methods of determining predictor importance can provide significantly different interpretations of the model, and the number of rules is extremely large, XGBoost stands as the more interpretable model. As they have very similar performance, interpretability is the deciding factor.

5.5 Ridge Ridge regression models are an improvement upon PLS and deal with highly dimensional data as well as highly correlated data. They are also highly resilient to overfitting which can plague other models. As a linear model without feature selection it is highly interpretable, and generally provides a good baseline of performance. The model was tuned using the code below.

```r
ridgeGrid <- expand.grid(lambda = seq(0, 0.03, by = 0.005))

ridge_model <- train(
  PH ~ .,
  data = caret_train,
  method = "ridge",
  preProcess = c("center", "scale", "corr"),
  tuneGrid = ridgeGrid,
  trControl = model_control
  )
```

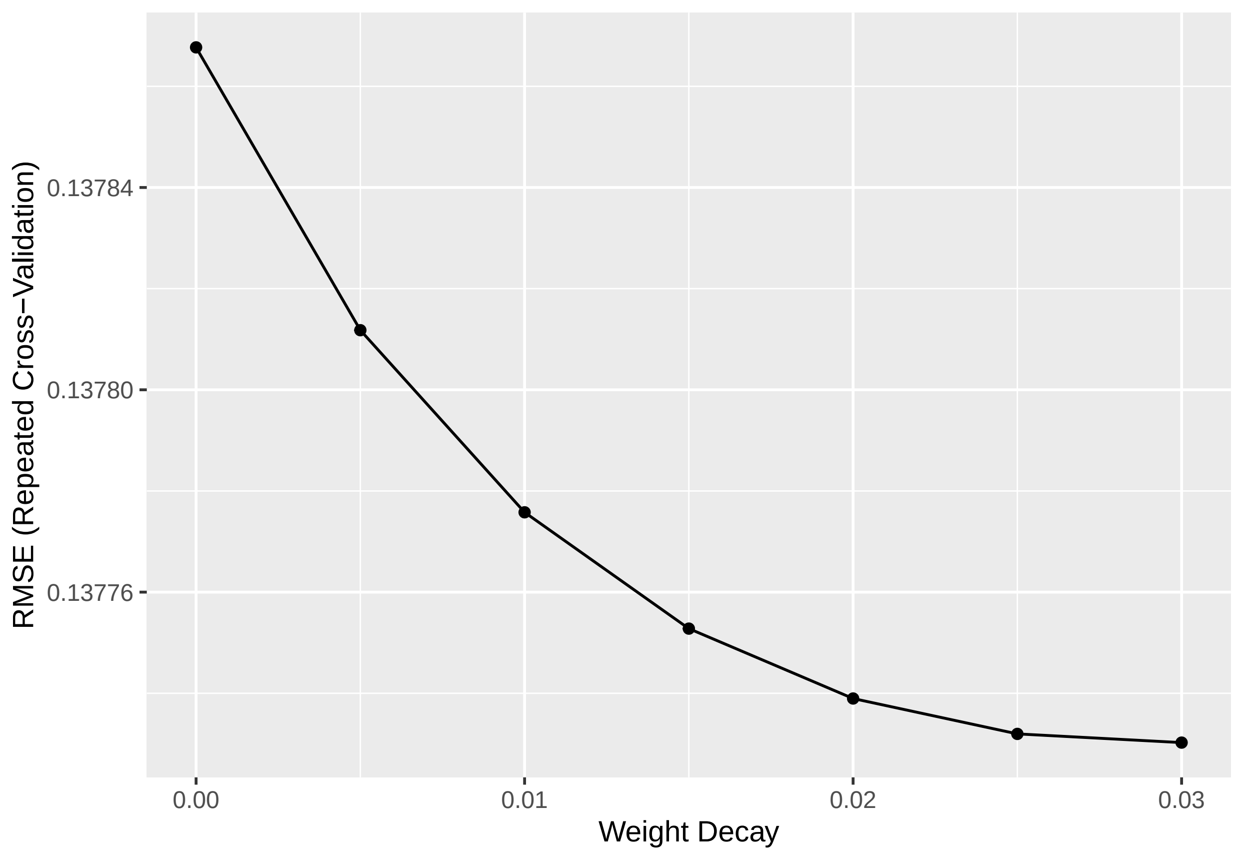
The resulting model performed best with a decay of 0.2 as seen in the plot below. The model had an RMSE of 0.1293958 and R^2 of 0.4102869 which, while respectable for a linear model, does not compete with the other models under consideration.

```r
riddeLambdaV <- ridge_model$bestTune[1,"lambda"]
ridge_model_res <- ridge_model$results |> filter(lambda == riddeLambdaV)
ridge_model_res |> kable(caption = "Ridge Training Set Evaluation Metrics") |> kable_styling() |>  kabl
```

Table 14: Ridge Training Set Evaluation Metrics

lambda	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
0.03	0.1377302	0.3730064	0.1070736	0.0056827	0.0438851	0.0040439

```r
ggplot(ridge_model)
```



```
ridge_model_pred <- predict(ridge_model, newdata = caret_test)
ridgePost <- postResample(pred = ridge_model_pred, obs = caret_test$PH)
```

```
ridge_test_metrics <- as.data.frame(t(ridgePost))

ridge_test_metrics |> kable(caption = "Ridge Test Set Evaluation Metrics") |> kable_styling() |>  kable_
```

Table 15: Ridge Test Set Evaluation Metrics

RMSE	Rsquared	MAE
0.129377	0.4104624	0.1028956

5.6 LASSO Regression LASSO regression is an extension of Ridge Regression which, in addition to regularizing, conducts feature selection. The model was generated using the code below.

```
lassoGrid <- expand.grid(.fraction = seq(.05, 1, length = 20))

lasso_model <- train(
  PH ~ .,
  data = caret_train,
  method = "lasso",
  preProcess = c("center", "scale"),
  tuneGrid = lassoGrid,
  trControl = model_control
  )
```

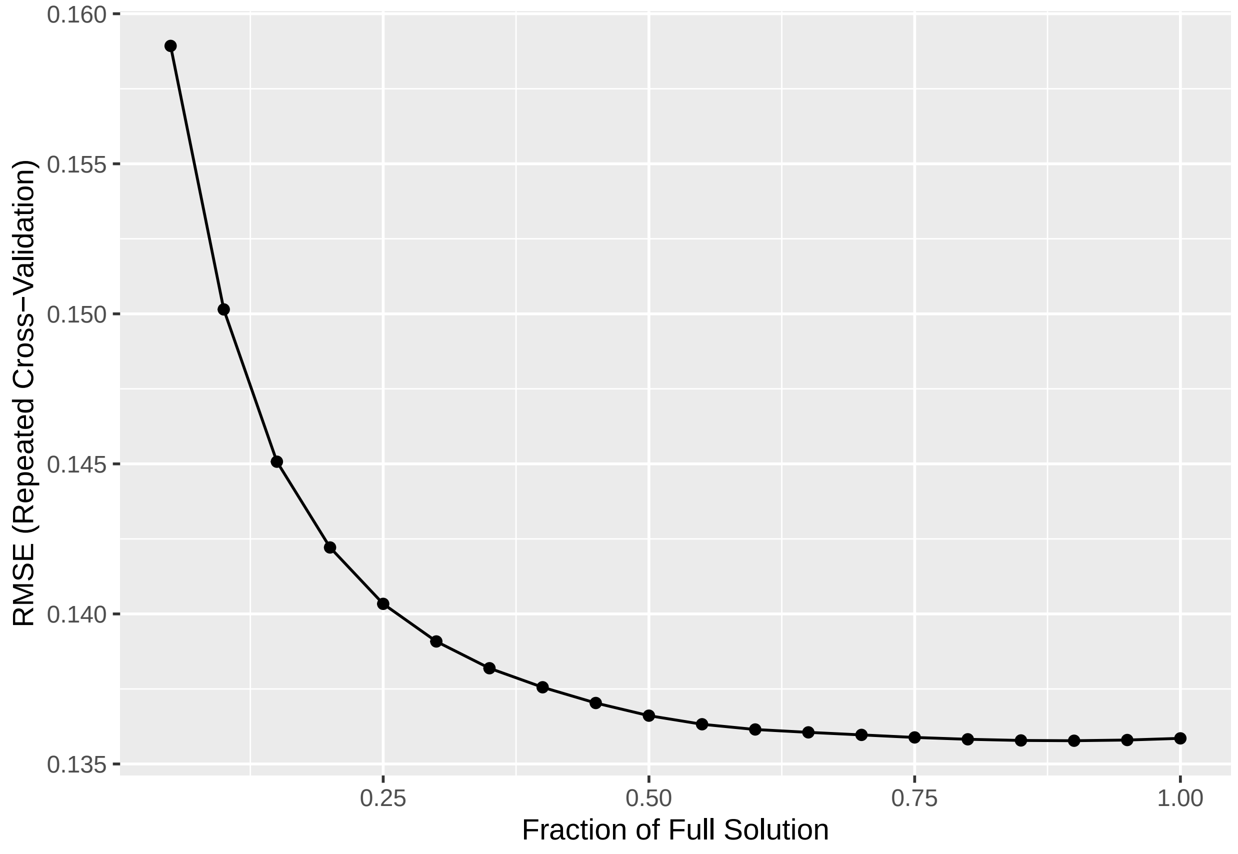
```
fractionLassoV <- lasso_model$bestTune[1,"fraction"]
lasso_model_res <- lasso_model$results |> filter(fraction == fractionLassoV)
lasso_model_res |> kable(caption = "Lasso Regression Training Set Evaluation Metrics") |> kable_styling
```

Table 16: Lasso Regression Training Set Evaluation Metrics

fraction	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
0.9	0.1357746	0.3914469	0.1047628	0.0077368	0.0395431	0.0050844

The resulting model had performance largely level off with a regression coefficient of 0.75. The resulting model had an RMSE of 0.1266923 and an R^2 of 0.4345533 on the test set which is practically equivalent to the Ridge model

```
ggplot(lasso_model)
```



```
lasso_model_pred <- predict(lasso_model, newdata = caret_test)
lassoPostS <- postResample(pred = lasso_model_pred, obs = caret_test$PH)
```

```
lasso_test_metrics <- as.data.frame(t(lassoPostS))
lasso_test_metrics |> kable(caption = "Lasso Regression Test Set Evaluation Metrics") |> kable_styling(
```

Table 17: Lasso Regression Test Set Evaluation Metrics

RMSE	Rsquared	MAE
0.1266923	0.4345533	0.1001281

5.7 Elastic Net (ENET) Elastic Net models are a generalization of LASSO which makes use of multiple tuning parameters to get the benefits of the regularization of Ridge models with the feature selection of LASSO models. The model was tuned using the code below.

```
set.seed(8675309)

enetGrid <- expand.grid(.lambda = c(0, 0.01, .1), .fraction = seq(.05, 1, length = 20))

enet_model <- train(
  PH ~ .,
  data = caret_train,
  method = "enet",
  preProcess = c("center", "scale"),
  tuneGrid = enetGrid,
  trControl = model_control
  )
```

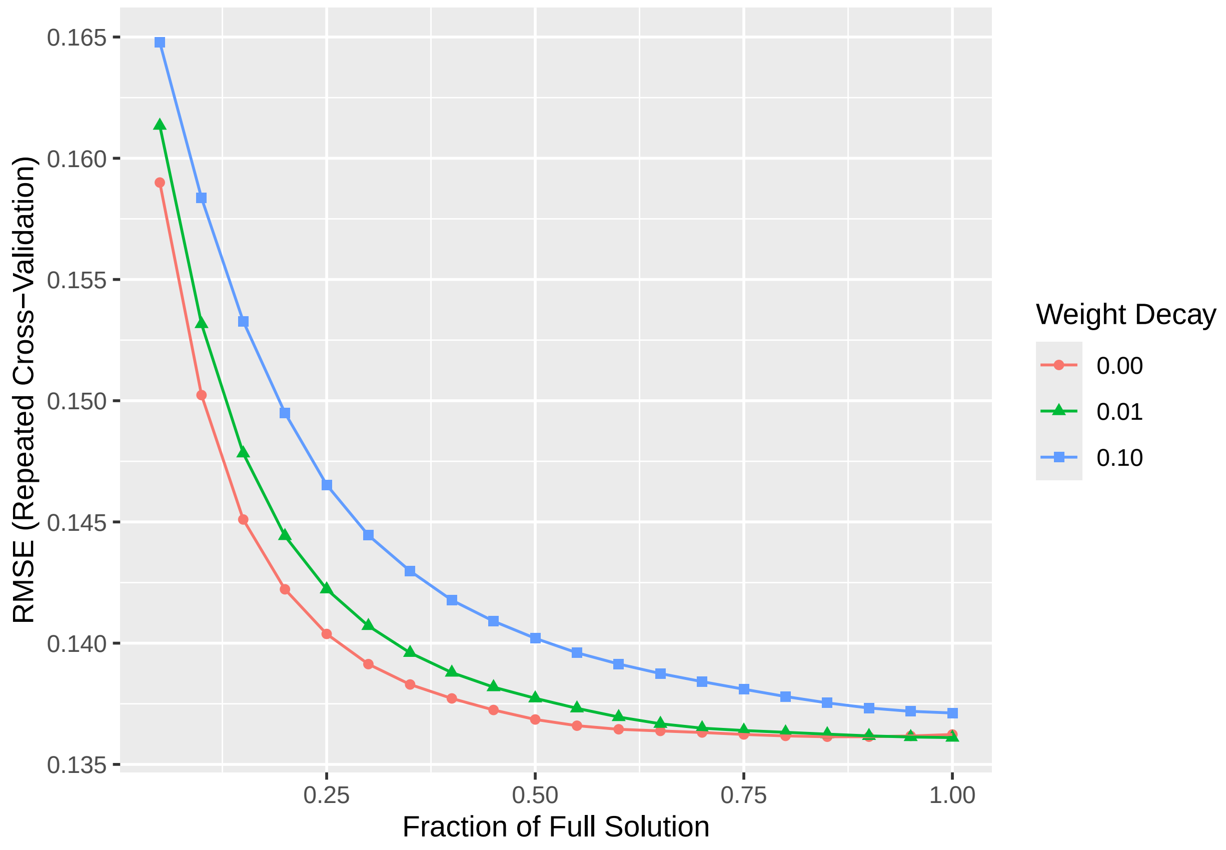
```
fractionEnentV <- enet_model$bestTune[1,"fraction"]
enetLambdaV <- enet_model$bestTune[1,"lambda"]

enet_model_res <- enet_model$results |> filter(fraction == fractionEnentV & lambda == enetLambdaV)
enet_model_res |> kable(caption = "Elastic Net Training Set Evaluation Metrics") |> kable_styling() |>
```

Table 18: Elastic Net Training Set Evaluation Metrics

lambda	fraction	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
0.01	1	0.1361057	0.3879753	0.1049803	0.0049833	0.027471	0.0032481

The resulting model demonstrates that linear models are likely not the best solution to predicting this data, as model performance improved as more of the solution was added. The final model had an RMSE of 0.1267145 and 0.4343588 on the test set, performing equivalently to the Ridge and LASSO models.

```
ggplot(enet_model)
```



```
enet_model_pred <- predict(enet_model, newdata = caret_test)
enetPostS <- postResample(pred = enet_model_pred, obs = caret_test$PH)
```

```
enet_test_metrics <- as.data.frame(t(enetPostS))
enet_test_metrics |> kable(caption = "Elastic Net Test Set Evaluation Metrics") |> kable_styling() |>  
```

Table 19: Elastic Net Test Set Evaluation Metrics

RMSE	Rsquared	MAE
0.1267145	0.4343588	0.1003512

5.8 Random Forest The optimal Random Forest model ended up having a slightly disappoint R^2=0.6639744 on the training set, however when evaluated on the test set we saw an R^2=0.7273597 and a decreased RMSE from 0.1018283 to 0.0891283. This is a particularly encouraging result as one of the dangers of Random Forest models is a propensity to overfit. Better performance on the test set than the training set indicates there is a low likelihood of overfitting and this model will likely generalize well.

```
set.seed(8675309)

data_split <- createDataPartition(imputed_data$PH, p = 0.75, list = FALSE)
training_data <- imputed_data[data_split, ]
testing_data <- imputed_data[-data_split, ]
```

```r
# Separate predictors and response variable for both training and testing sets
train_x <- training_data %>% select(-PH)
train_y <- training_data$PH
test_x <- testing_data %>% select(-PH)
test_y <- testing_data$PH
```

```r
set.seed(8675309)

train_control <- trainControl(method = "cv", number = 5)
# Define the grid for hyperparameters to tune (only mtry here)
tune_grid <- expand.grid(
  mtry = 34)  # mtry values to test


# Train the Random Forest model using cross-validation for mtry
rf_cv_model <- train(
  x = train_x,
  y = train_y,
  method = "rf",
  trControl = train_control,
  tuneGrid = tune_grid,
  ntree =  1000,  # Number of trees
  importance = TRUE
)
```

```r
rfMtryV <- rf_cv_model$bestTune[1,"mtry"]

rf_model_res <- rf_cv_model$results |> filter(mtry == rfMtryV)
rf_model_res |> kable(caption = "Random Forest Training Set Evaluation Metrics") |> kable_styling() |>
```

Table 20: Random Forest Training Set Evaluation Metrics

mtry	RMSE	Rsquared	MAE	RMSESD	RsquaredSD	MAESD
34	0.1018283	0.6639744	0.0734955	0.0050027	0.0456675	0.0016966

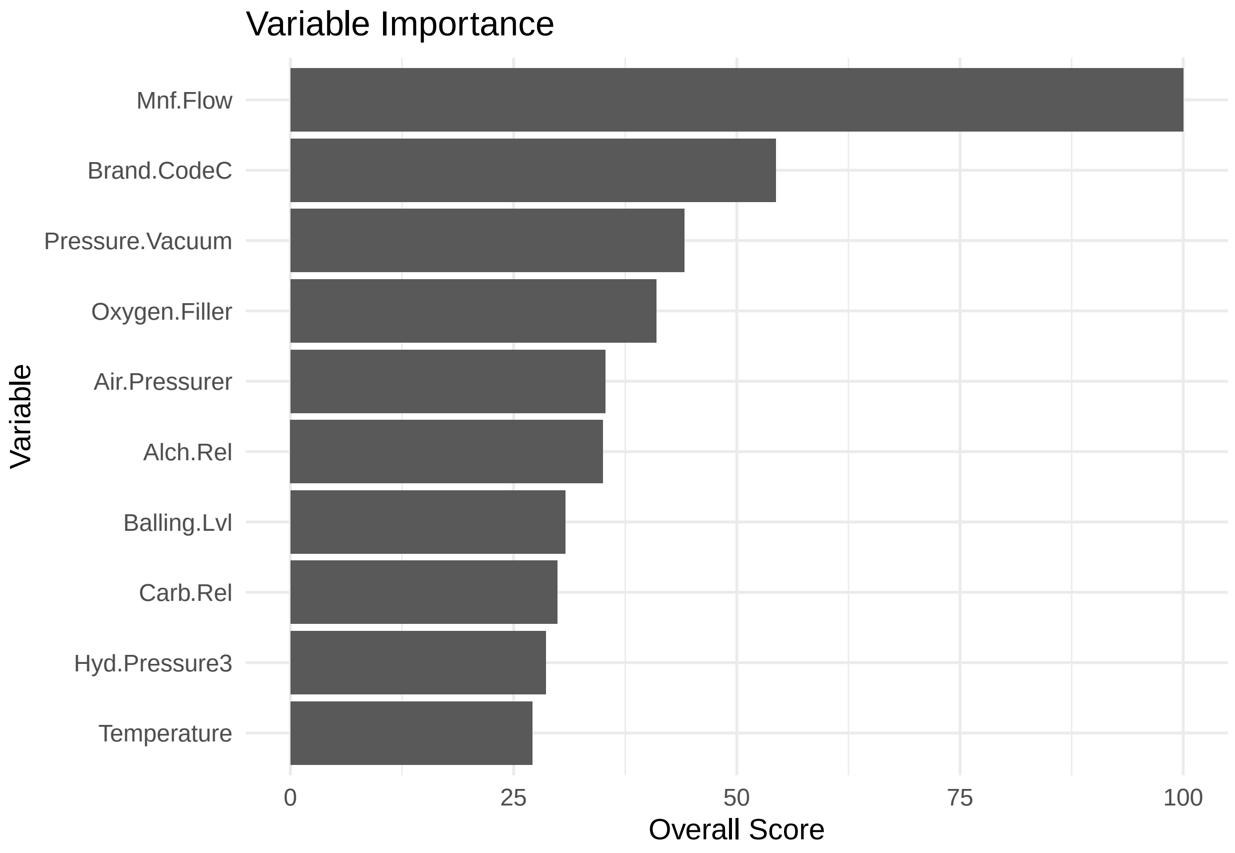
```r
rf_cv_model_pred <- predict(rf_cv_model, newdata = test_x)
randomForestPost <- postResample(pred = rf_cv_model_pred, obs = test_y)
```

```r
rf_test_metrics <- as.data.frame(t(randomForestPost))
rf_test_metrics |> kable(caption = "Random Forest Test Set Evaluation Metrics") |> kable_styling() |>
```

Table 21: Random Forest Test Set Evaluation Metrics

RMSE	Rsquared	MAE
0.0891283	0.7273597	0.0646925

```
rf_final_varImportance <- varImp(rf_cv_model)
rf_final_importance_df <- data.frame(Variable= rownames(rf_final_varImportance$importance),
                                Overall=rf_final_varImportance$importance$Overall)

rf_final_importance_df |> arrange(-Overall) %>% head(10) %>%
  ggplot( aes(y = reorder(Variable, +Overall), x = Overall)) + geom_bar(stat = "identity") + labs(
    title = "Variable Importance",
    x = "Overall Score",
    y = "Variable"
  )  + theme_minimal()
```



5.9 XGBoost XGBoost (Extreme Gradient Boosting) is a robust machine learning algorithm built on gradient boosting techniques, which combine gradient descent for numerical optimization with boosting, an ensemble method that iteratively improves weak learners to create strong models. The term "gradient" reflects the algorithm's use of derivatives from the loss function to optimize predictions. XGBoost operates through three main components: an additive model that sequentially builds improvements, a customizable differentiable loss function to measure prediction errors, and weak learners, typically decision trees, refined iteratively by addressing their shortcomings.XGBoost enhances flexibility and predictive performance by framing boosting as a numerical optimization problem, making it a favored tool in machine learning.

```
set.seed(8675309)

data_split <- createDataPartition(manufacturing_tr$PH, p = 0.75, list = FALSE)
training_data <- imputed_data[data_split, ]
testing_data <- imputed_data[-data_split, ]
```

```r
set.seed(8675309)

# Define trainControl for cross-validation
train_control <- trainControl(
  method = "cv",                # Cross-validation
  number = 5,                   # Number of folds
  verboseIter = F            # Show training progress
)

# Define a grid for hyperparameter tuning
xgb_grid <- expand.grid(
  nrounds = 1000,               # Number of boosting rounds
  max_depth = 6,                # Maximum tree depth
  eta = 0.05,                   # Learning rate
  gamma = 0,                    # Minimum loss reduction
  colsample_bytree = 0.8,       # Fraction of features used for each tree
  min_child_weight = 6,         # Minimum number of observations for a split
  subsample = 0.9               # Fraction of data used for each tree
)

# Train the XGBoost model using caret
xgb_model <- train(
  PH ~ .,                       # Formula (PH is the target variable)
  data = training_data,         # Training dataset
  method = "xgbTree",           # Use XGBoost
  trControl = train_control,    # Cross-validation control
  tuneGrid = xgb_grid,          # Hyperparameter grid
  metric = "RMSE"               # Optimize for RMSE
)
```

```r
xgb_model$results |> kable(caption = " XGBoost Set Evaluation Metrics") |> kable_styling() |>  kable_cla
```

Table 22: XGBoost Set Evaluation Metrics

nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample	RMSE	Rsquared	
1000	6	0.05	0	0.8	6	0.9	0.1040341	0.643032	0

```r
# Make predictions using the trained model and iteration_range
model_pred <- predict(xgb_model, newdata = testing_data)
xgBoostPostS <- postResample(pred = model_pred, obs = testing_data$PH)
```

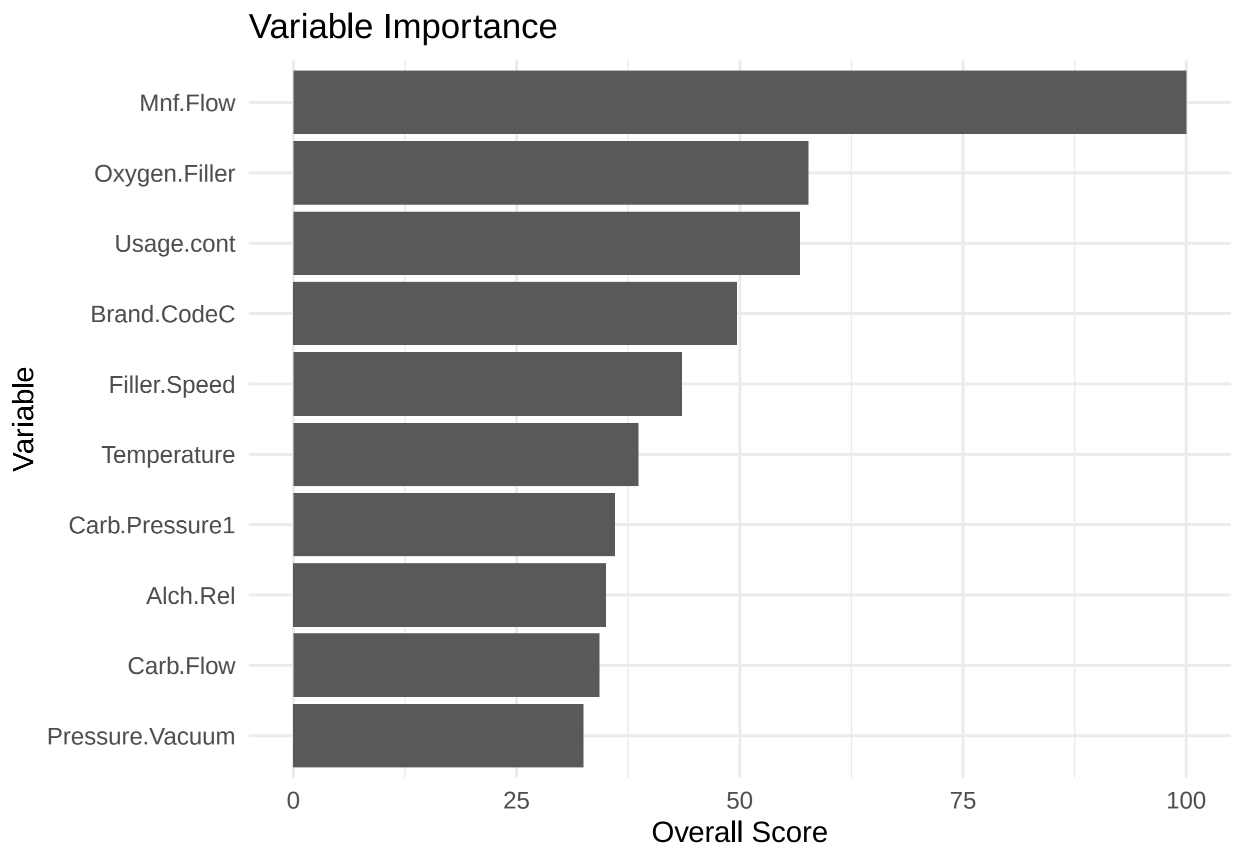
```r
xgBoos_test_metrics <- as.data.frame(t(xgBoostPostS))
xgBoos_test_metrics |> kable(caption = "XGBoost Test Set Evaluation Metrics") |> kable_styling() |>  kal
```

Table 23: XGBoost Test Set Evaluation Metrics

RMSE	Rsquared	MAE

0.0890684	0.7218483	0.0664595

```
xgBoost_varImportance <- varImp(xgb_model)
xgBoost_importance_df <- data.frame(Variable= rownames(xgBoost_varImportance$importance),
                                    Overall=xgBoost_varImportance$importance$Overall)

xgBoost_importance_df |> arrange(-Overall) %>% head(10) %>%
  ggplot( aes(y = reorder(Variable, +Overall), x = Overall)) + geom_bar(stat = "identity") + labs(
    title = "Variable Importance",
    x = "Overall Score",
    y = "Variable"
  )  + theme_minimal()
```



5.9 Neural Networks The concept of Neural Networks is a node-layer system trained on an existing dataset. An artificial neural network (ANN) model contains an input layer, hidden layers, and an output layer, with each of these layers containing nodes. At the input layer, each node contains a dimension of the dataset. At the outer layer, each node represents the final output or prediction. The hidden layers are the core of the model, where the computation and learning take place. Each node of a neural network in a hidden layer takes in a set of the input, performs a computation based on the activation function, and then creates an output. These outputs are assigned weights which determine its level of consideration in the final output and are adjusted throughout the training process. The model is trained over many cycles, called epochs. At the end of each batch, a loss function is evaluated based on the model's performance. Batches are subsets of the training dataset, and in the case of the nnet package in R, batch size is equal to the size of the dataset as a whole. With supervised learning, the dataset will have various input parameters as well as a known output. The loss function compares the model output with the known output. Weights for each node are adjusted based on this loss function, and the model is said to converge when the loss function no longer decreases.

ANN models can utilize parallel processing, allowing faster model training. As the weight of each node is adjusted through the process mentioned above, ANN models can distinguish important input parameters; as our dataset contains over 30 input variables, we tested various ANN models as a potential choice for predicting pH. Since some of the input variables are highly correlated, we can account for this in the ANN model by testing various weight decay values to prevent overfitting.

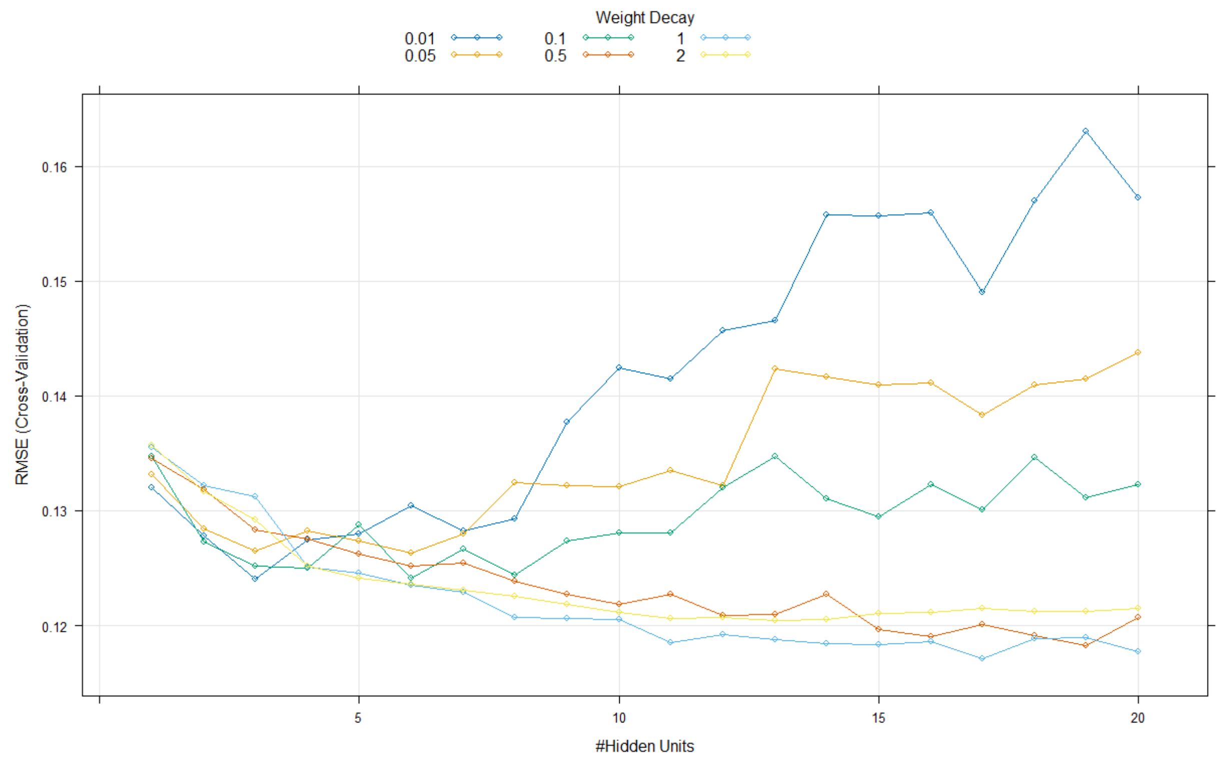
The nnet method (nnet) in the nnet package was used with caret to tune an ANN model to the data.

```
StudentData <- read.csv("https://raw.githubusercontent.com/MarjeteV/data624/refs/heads/main/imputed_tes

# Parallel processing


set.seed(8675309)
trainIndex <- createDataPartition(StudentData$PH, p = 0.75, list = FALSE)
trainData <- StudentData[trainIndex, ]
testData <- StudentData[-trainIndex, ]
# Separate x, y, train
x_train <- trainData[, names(trainData) != "PH"]
y_train <- trainData$PH
x_test <- testData[, names(testData) != "PH"]
y_test <- testData$PH
```

Data was preprocessed with centering and scaling, and a linear activation function was used. The model utilized cross validation, and 500 maximum iterations were specified in the tuning phase. Since the nnet package defaults to a batch size equal to the training set size, the number of epochs is equal to the max number of iterations specified given that the model does not converge before the max iterations is reached. For the tuning grid, we tested 1:35 nodes and weight decay regularization values of 0.01, 0.05, 0.1, 0.5, 1, and 2. Using these parameters, we found a best tune model with 17 nodes and a decay value of 1. This resulted in a test set RMSE of 0.113 and an R^2 of 0.555.

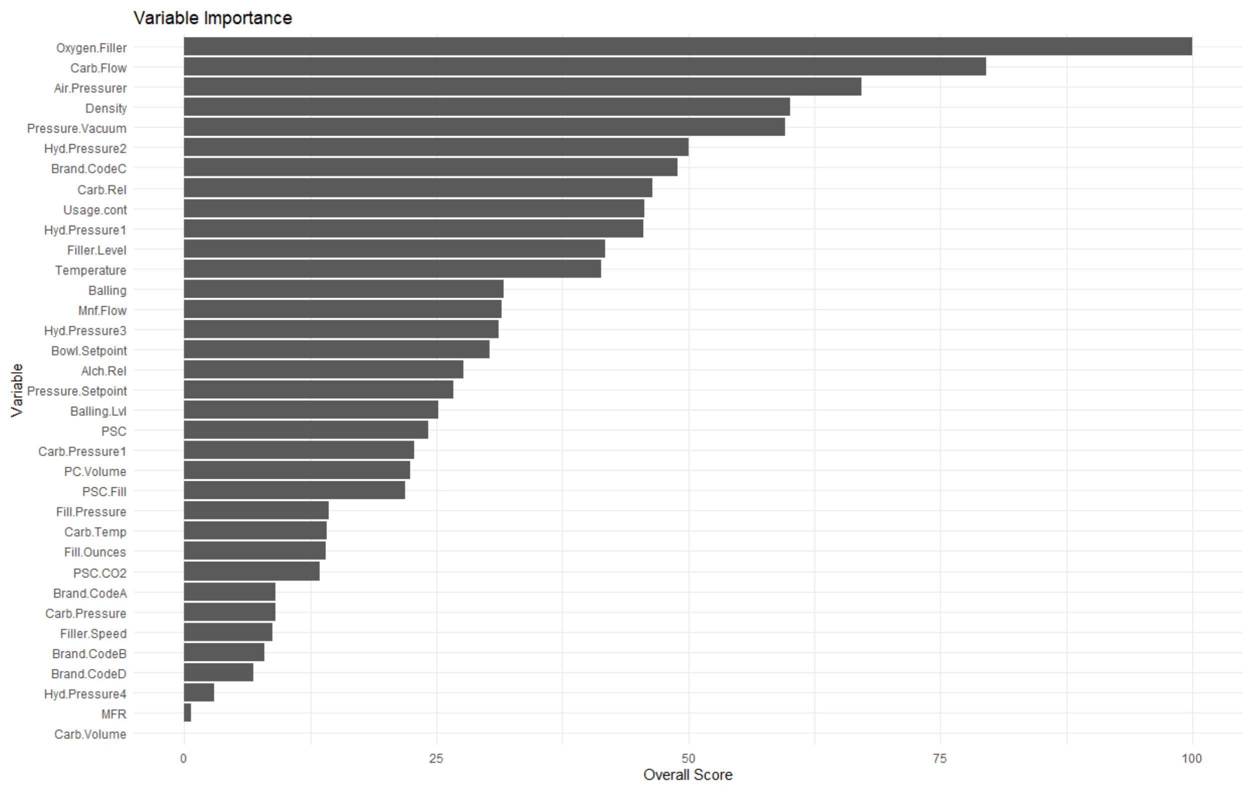
```
# tuning grid
nnetGrid <- expand.grid(
 size = 1:35,
 decay = c(0.01,0.05, 0.1, 0.5, 1, 2)
 )

# model
nnetTuned <- train(
 x = x_train,
 y = y_train,
 method = "nnet",
 tuneGrid = nnetGrid,
 preProc = c("center", "scale"),
 trControl = trainControl(method = "cv"),
 linout = TRUE,
 trace = FALSE,
 maxit = 500,
 MaxNWts = 35 * (ncol(x_train) + 1) + 35 + 1)

# Tuning plot
```

```
plot(nnetTuned)
# Model results
nnetTuned$bestTune
```



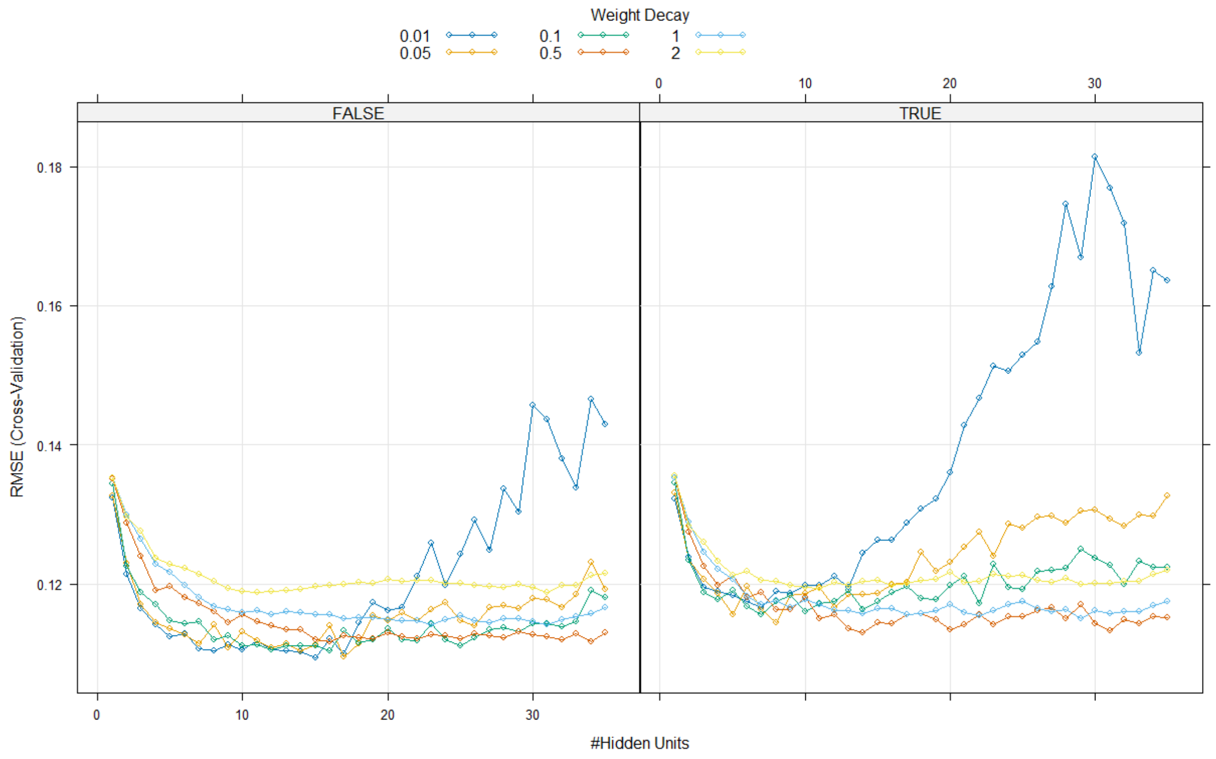
Taking a look at the variable importance of this nnet model, we can see that variables of high importance are Density, Mnf Flow, Bowl Setpoint, and Temperature.

```
# Variable importance
nnetTuned_final_varImportance <- varImp(nnetTuned)

nnetTuned_final_importance_df <- data.frame(Variable= rownames(nnetTuned_final_varImportance$importance
                                Overall=nnetTuned_final_varImportance$importance$Overall)

nnetTuned_final_importance_df |> ggplot( aes(y = reorder(Variable, +Overall), x = Overall)) + geom_bar(
    title = "Variable Importance",
    x = "Overall Score",
    y = "Variable"
  )  + theme_minimal()
```



In addition to exploring various regularization values, an ensemble neural network method, average neural net (avnnet), was also tuned to account for the highly correlated variables. For the tuning grid, we again tested 1:35 nodes and weight decay regularization values of 0.01, 0.05, 0.1, 0.5, 1, and 2. We also specified both bootstrapped aggregation and no bootstrapped aggregation in the tuning grid. The best tuned model had 15 nodes and 0.01 decay, with no bootstrapped aggregation.

```
# tuning grid
avNNetGrid <- expand.grid(
 size = 1:35,
 decay = c(0.01,0.05, 0.1, 0.5, 1, 2),
 bag = c(TRUE,FALSE))

# model
avNNetTuned <- train(
 x = x_train,
 y = y_train,
 method = "avNNet",
 tuneGrid = avNNetGrid,
 preProc = c("center", "scale"),
 trControl = trainControl(method = "cv", allowParallel = TRUE),
 linout = TRUE,
 trace = FALSE,
 maxit = 500,
 MaxNWts = 35 * (ncol(x_train) + 1) + 35 + 1)
# Model results
avNNetTuned$bestTune
# Tuning plot
plot(avNNetTuned)
```



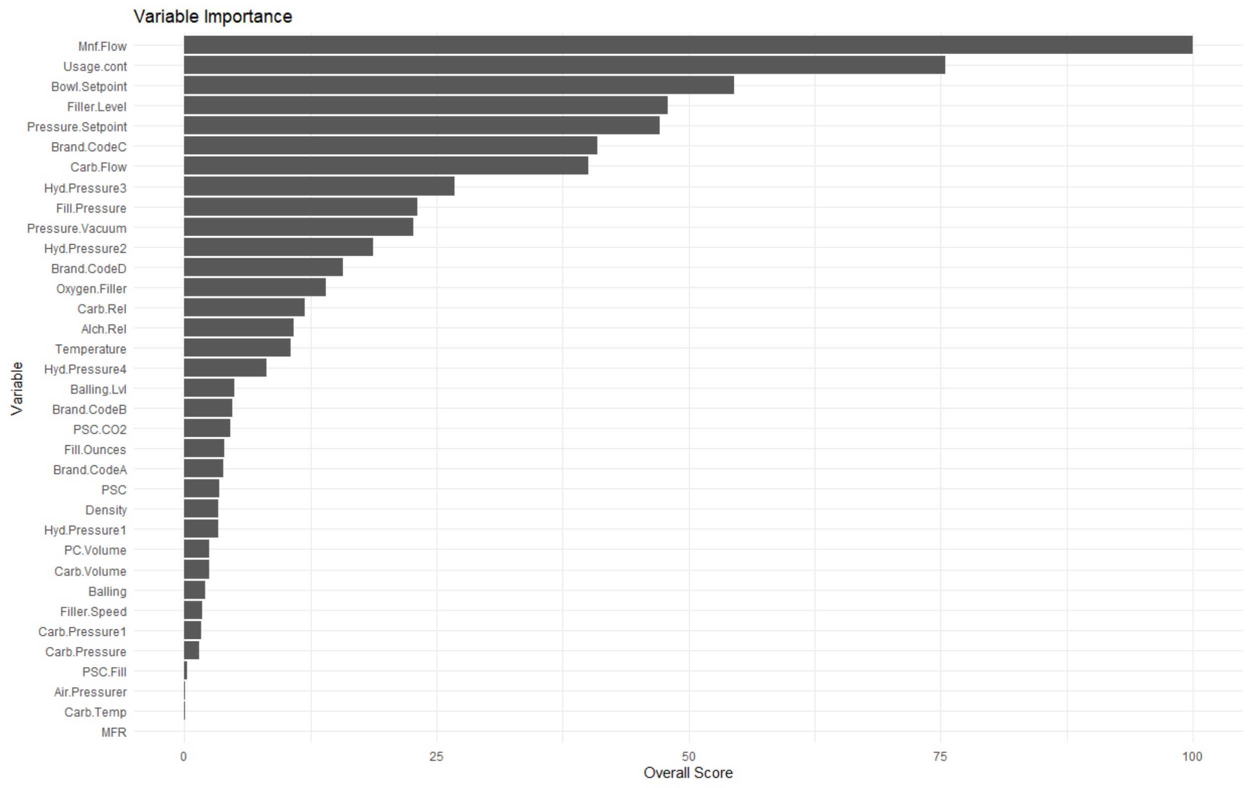
```
# Test set predictions & metrics
avNNetPred <- predict(avNNetTuned, newdata = x_test)
postResample(pred = avNNetPred, obs = y_test)
```

However, after a best tune was determined, these best tuned parameters were used for a final model training that included a higher and higher numbers of maximum iterations until RMSE and R^2 stabilized. This final model used a maximum iteration of 5000 and had an RMSE of 0.109 and R^2 of 0.603.

```
avNNetGrid_final <- expand.grid(
 size = 15,
 decay = 0.01,
 bag = FALSE)

avNNetTuned_final <- train(
 x = x_train,
 y = y_train,
 method = "avNNet",
 tuneGrid = avNNetGrid_final,
 preProc = c("center", "scale"),
 trControl = trainControl(method = "cv", allowParallel = TRUE),
 linout = TRUE,
 trace = FALSE,
 maxit = 1000,
 MaxNWts = 15 * (ncol(x_train) + 1) + 15 + 1)

avNNetPred_final <- predict(avNNetTuned_final, newdata = x_test)
postResample(pred = avNNetPred_final, obs = y_test)
```

When examining the variable importance, we can see that Mnf flow ranks as the highest importance variable by a large margin, followed by Usage cont and Bowl Setpoint.

```
# Variable importance
avvNNetTuned_final_varImportance <- varImp(avNNetTuned_final)

avvNNetTuned_final_importance_df <- data.frame(Variable= rownames(avvNNetTuned_final_varImportance$impo
                                Overall=avvNNetTuned_final_varImportance$importance$Overall)

avvNNetTuned_final_importance_df %>% ggplot( aes(y = reorder(Variable, +Overall), x = Overall)) + geom_
    title = "Variable Importance",
    x = "Overall Score",
    y = "Variable"
  )  + theme_minimal()
```



6. Model Selection

From the model summary table above, the best performing models were the XGBoost model and the Cross-Validated Random Forest model. Both models had an R^2 of approximately 0.72, and both models had an RMSE of 0.89 on the test set.

Overall, the performance metrics for these two models were very similar, especially when looking at the test set evaluation. As such, model selection took into account other selection criteria. The Random Forest model has better interpretability, but XGBoost excels in other areas such as in its generalization - specifically, XGBoost has built in regularization techniques that can prevent overfitting to data. When looking at the training R^2 and RMSE, the RF model had a higher R^2 at 0.66 and a lower RMSE at 0.10 compared to an R^2 of 0.64 and an RMSE of 0.10 for the XGBoost model. As such, it does not appear overfitting was an issue for either model given the higher R^2 for the out-of-sample data.

When looking at variable importance, XGBoost model and Random Forest model show similar variables of importance, with 6 of the top 10 variables overlapping. Both models also appear to have a similar distribution of importance among its top ten variables. The importance of the Mnf Flow variable is greatest for both models, having an overall score of 100. After the top variable, both models have a steep drop off in importance score for the remaining top variables.

The Random Forest model and the XGBoost model exhibited very similar performance metrics and variable importance distributions. While XGBoost models are less prone to overfitting due to its built in regularization, comparing the in-sample R^2 and RMSE to the out-of-sample R^2 and RMSE show that neither models are overfitted. As a result, the Random Forest model was chosen for its better interpretability.

```r
# Retraining selected model on full training set and making final predictions
set.seed(8675309)

train_control <- trainControl(method = "cv", number = 5)
# Define the grid for hyperparameters to tune (only mtry here)
tune_grid <- expand.grid(
  mtry = 34)  # mtry values to test


# Train the Random Forest model using cross-validation for mtry
rf_final_model <- train(
  PH ~ .,
  data = imputed_data,
  method = "rf",
  trControl = train_control,
  tuneGrid = tune_grid,
  ntree =  1000,  # Number of trees
  importance = TRUE
)

test %<>% mutate(across(
  .cols = c('Filler.Speed', 'Hyd.Pressure4', 'Bowl.Setpoint', 'Carb.Flow'),
  .fns = as.numeric
  ))

# Create a one hot encoding tibble
dummies_final <- dummyVars( ~ Brand.Code, data = test)

one_hot_df_final <- predict(dummies_final, newdata = test) |>
  as_tibble()

# Add the one hot df to the original and remove the categorical column
one_hot_df_final <- test |>
  cbind(one_hot_df_final) |>
  select(-Brand.Code)

final_pred <- predict(rf_final_model, newdata = one_hot_df_final)
```

```r
names(final_pred) <- "PH"
write.csv(final_pred, "~/School/data624/final_predictions.csv")
```

7. Conclusion

This project explored multiple predictive models to address the new regulatory requirement of understanding and predicting PH in the manufacturing process at ABC Beverage. The analysis began with exploratory data analysis to understand variable distributions and relationships, detect outliers, and address missing data. Bagging imputation technique was applied to fill in missing values. Additionally, rows with missing PH values were dropped, and we applied one-hot encoding to the Brand.Code variable to retain its categorical information without introducing imputation risks.

Several models were tested, covering a range of categories: linear models (PLS, Ridge Regression, LASSO, Elastic Net), non-linear models (SVM, Neural Networks), tree-based models (CART, Random Forest, XGBoost), and rule-based hybrid models (Cubist). Each model's performance was evaluated based on metrics such as RMSE, R^2, and MAE, focusing on balancing predictive accuracy and interpretability. Among the models, Cubist, Random Forest, and XGBoost demonstrated a strong ability to capture variability. However, the Cubist model, despite achieving an R^2 of 0.66, and XGBoost, with a strong test R^2 of 0.71, were both not selected due to their limited interpretability and weaker performance during training evaluation, making them less suitable for identifying the key drivers of pH variability.

The Random Forest model demonstrated strong performance and was selected as the best model for predicting pH. On the training set, it achieved an R^2 of 0.664 with an RMSE of 0.1018 and an MAE of 0.0735, indicating effective learning of relationships within the data. On the test set, the model continued to generalize well, achieving an R^2 of 0.727, RMSE of 0.0891, and MAE of 0.0647. These results highlight the Random Forest model's ability to explain variability and produce accurate predictions on unseen data. While other models, such as XGBoost, were considered, Random Forest's robust performance across both the training and test sets, along with its ability to handle complex, non-linear relationships, made it the preferred choice for this analysis. The balanced test metrics indicate minimal overfitting and strong predictive accuracy, making Random Forest the most reliable model for understanding and predicting pH variability in the manufacturing process. We also used Random Forest to identify the variables with the highest importance: Manufacturing Flow Rate (Mnf.Flow), Continuous Usage (Usage.cont) and Oxygen in Filler (Oxygen.Filler) are key contributors. These variables are critical in regulating pH levels during production and ensuring product consistency.

Future improvements could focus on adding more predictors, enhancing feature engineering, and increasing the dataset size with additional observations to help us better understand our manufacturing process, predictive factors, and accurately report our model of PH. Exploring other interpretable models or improving current models through advanced tuning and validation techniques could also enhance performance while maintaining transparency. Accurate pH predictions are essential for meeting regulatory standards, improving quality control, and reducing variability, ensuring ABC Beverage consistently delivers high-quality products.

8. Annotated References

1. **Anton Paar Wiki. (n.d.).** *Carbon Dioxide in Beverages.*
   Carbon Dioxide in Beverages
   This resource provides an in-depth understanding of the role of carbon dioxide in beverages, including its effect on carbonation, fizziness, and product quality. It directly supports our analysis of variables like **Carb Flow** and **Carb Pressure**, which influence carbonation and pH levels.

2. **Omega. (n.d.).** *What is pH?*
   What is pH?
   This article explains the concept of pH, its measurement, and its relevance to various industries. It is useful for understanding the chemical principles behind pH variability in beverages and helps contextualize our target variable within the manufacturing process.

3. **Emerson. (n.d.).** *Training Beverage Process Solutions Guide on De-Aeration.*
   Training Beverage Process Solutions Guide on De-Aeration
   This document focuses on de-aeration processes in beverage production, particularly the removal of

dissolved oxygen and its impact on carbonation and quality. It directly relates to variables like **Oxygen.Filler** and **Pressure.Vacuum**, providing insights into their operational importance.

4. **Jochamp. (n.d.).** *Carbonated Beverages Manufacturing Process - A Step by Step Guide.*
   Carbonated Beverages Manufacturing Process - A Step by Step Guide
   This guide offers a comprehensive overview of the carbonation process in beverage production, including the role of temperature, pressure, and filling speed. It supports our understanding of variables like **Temperature**, **Filler.Speed**, and **Carb Pressure**, helping us interpret their influence on product quality.