**Data 624: Project 2**

**Regression Modeling**

**Group 2: Alice Friedman, Kayleah Griffen, Josh Iden, Michael Ippolito**

Table of Contents

[Executive Summary 3](#_Toc140237831)

[Introduction 4](#_Toc140237832)

[Exploratory Data Analysis 5](#_Toc140237833)

[Loading the Data 5](#_Toc140237834)

[First Visualization of the Data 5](#_Toc140237835)

[Correlation between variables 6](#_Toc140237836)

[Near Zero Variance 7](#_Toc140237837)

[Missing Values 7](#_Toc140237838)

[Skewness 8](#_Toc140237839)

[Outliers 8](#_Toc140237840)

[Data Preparation 9](#_Toc140237841)

[Missing Values 9](#_Toc140237842)

[Dummy Variables 10](#_Toc140237843)

[Full Data Set 10](#_Toc140237844)

[Reduced Data Set 10](#_Toc140237845)

[Collinearity 11](#_Toc140237846)

[Near-zero variance 11](#_Toc140237847)

[Skewness 11](#_Toc140237848)

[Outliers 11](#_Toc140237849)

[Data Partitioning 12](#_Toc140237850)

[Regression Modeling 13](#_Toc140237851)

[Modeling Results 13](#_Toc140237852)

[Variable Importance 15](#_Toc140237853)

[Dataset to Predict 16](#_Toc140237854)

[Conclusion 16](#_Toc140237855)

[References 17](#_Toc140237856)

[Appendix 1: R Code 19](#_Toc140237857)

[Appendix 2: Regression Modeling Explanations 101](#_Toc140237858)

[Appendix 3: Textbook Model Summary 105](#_Toc140237859)

# Executive Summary

The purpose of this project is to use the provided data from a beverage manufacturing company to predict potential for hydrogen (pH), which represents acidity/alkalinity, because pH is a key performance indicator and must conform to a specific range. A variety of approaches were used employing various combinations of the data to achieve a prediction model, called a Cubist model, with a mean absolute percentage error of 0.8%. We conclude that key drivers of the pH are “Mnf.Flow” and, therefore, to keep the pH in the critical range we recommend tightly controlling the “Mnf.Flow” range. Upon exploring the “Mnf.Flow” and it’s relationship with pH, we find that values for “Mnf.Flow” greater than zero are more likely to fall within the critical pH range. For this reason, we recommend that “Mnf.Flow” be constrained to greater than zero. Additionally, our modeling also shows that observations that include “Brand.Code” information improves predictive accuracy. Therefore, we recommend “Brand.Code” be a required data input. We note that one caveat is that the predictors in the training data all fell within certain ranges, and therefore our predictions are most accurate for forecasting the pH given predictors that fall between the ranges in the training data. For the dataset we were provided to predict the pH of, our exploratory data analysis showed the predictors used to train and test the model followed similar distributions to the new set of predictors we needed to use to predict the pH. For this reason we believe that our predictions are reliable.

# 

# Introduction

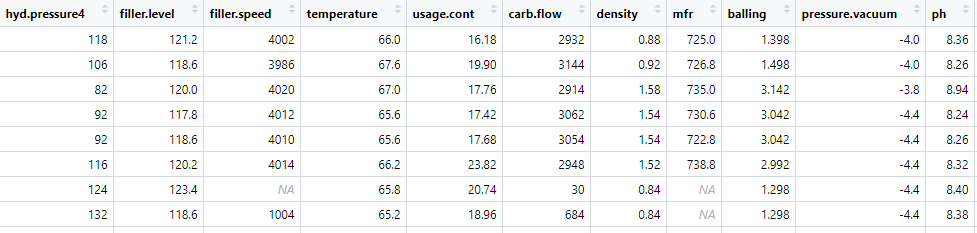
We have been tasked by a beverage manufacturing company to predict pH values of their beverages based on provided product data. pH measures acidity and alkalinity of substances on a scale of 0 to 14, and represents a key performance indicator that affects the flavor profile and quality of beverages as well as their safety for human consumption (Reddy, A. et al, 2016). Beverages with pH values between 2 and 8.5 are generally safe for human consumption (Ashley, R). Values below 7 are acidic, values near 7 are neutral, and values above 7 are alkaline. Certain regions or countries may impose regulations requiring beverages maintain a pH value between certain levels. Achieving desired pH levels may also impact cost and timeline of production. For these reasons, accurate predictions of pH are critical to success for a beverage manufacturing company.

# 

# Exploratory Data Analysis

## Loading the Data

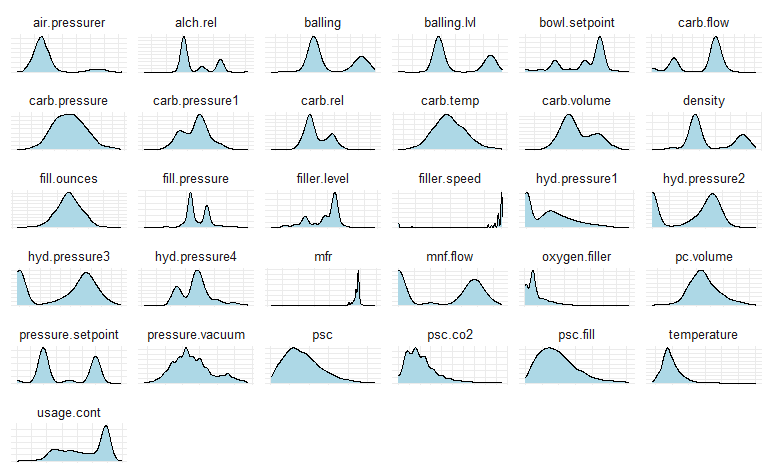
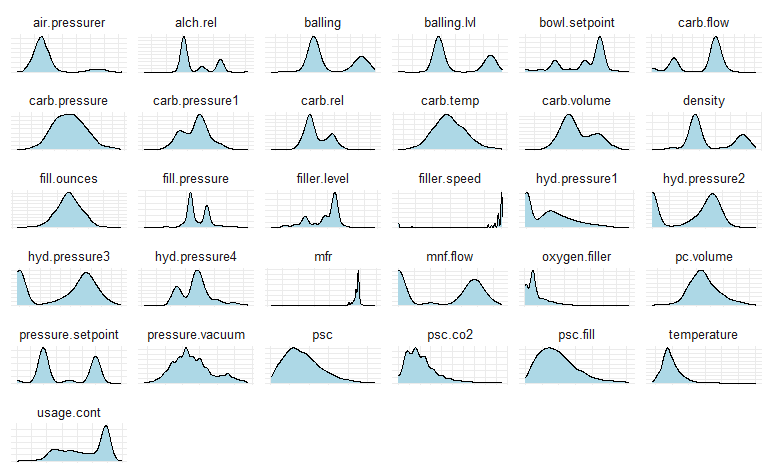
We uploaded the data files provided to us to github, a website that allows for file storage. We then used the open source software, R, to conduct our data analysis. We loaded the data into R, and did a brief inspection of the data. In one of the files provided, we were given 2,571 cases with 33 variables, including pH, to use to train our models. Below is a snippet of the loaded data in tabular format, where you can see some of the variables provided, including pH. It should be noted that we were provided no documentation about the data provided, for example we were not provided descriptions of the columns or units.



We were also given a file with 267 cases of all of the variables, excluding pH, and we were requested to predict the pH.

## First Visualization of the Data

To begin to understand the data, we made two types of graphs. The first type is a density graph and it shows the frequency for each value of each predictor. A subset of the predictors density graphs are shown below. These graphs help us to understand how the values for each predictor are distributed, from these for example, we can tell that most values for “air.pressure” occur in one of two bins, for “alch.rel” occur in one of three bins, and for “pc.volume” most values fall into a single bin.

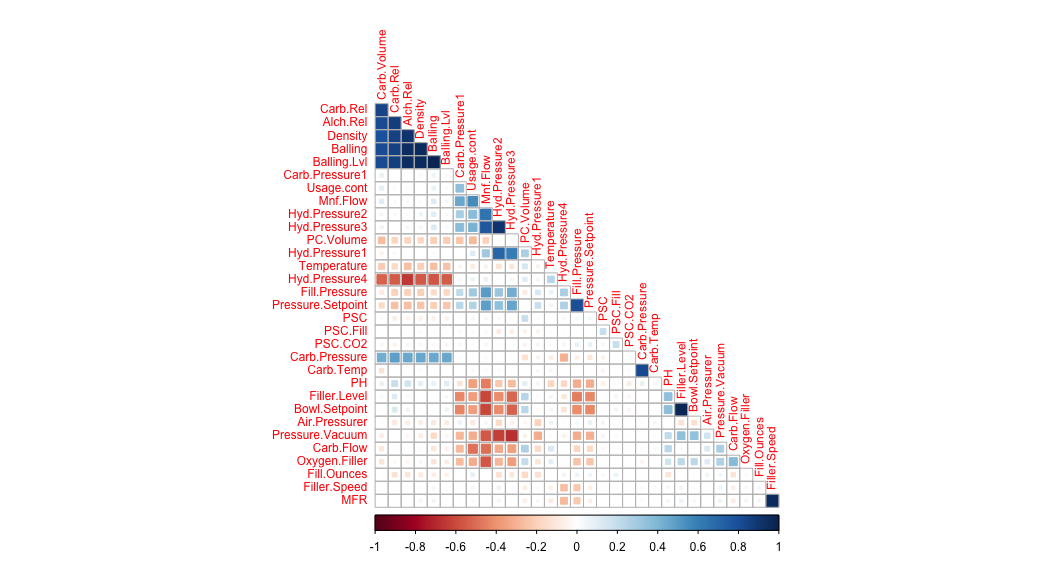


Another helpful type of graph we looked at graphs each predictor against the target, pH value The idea with this type of graph is to see if any strong relationships stand out. We inspected each predictor, a subset of which is below, and couldn’t visually identify any strong relationships.

|  |  |
| --- | --- |
| pH |  |

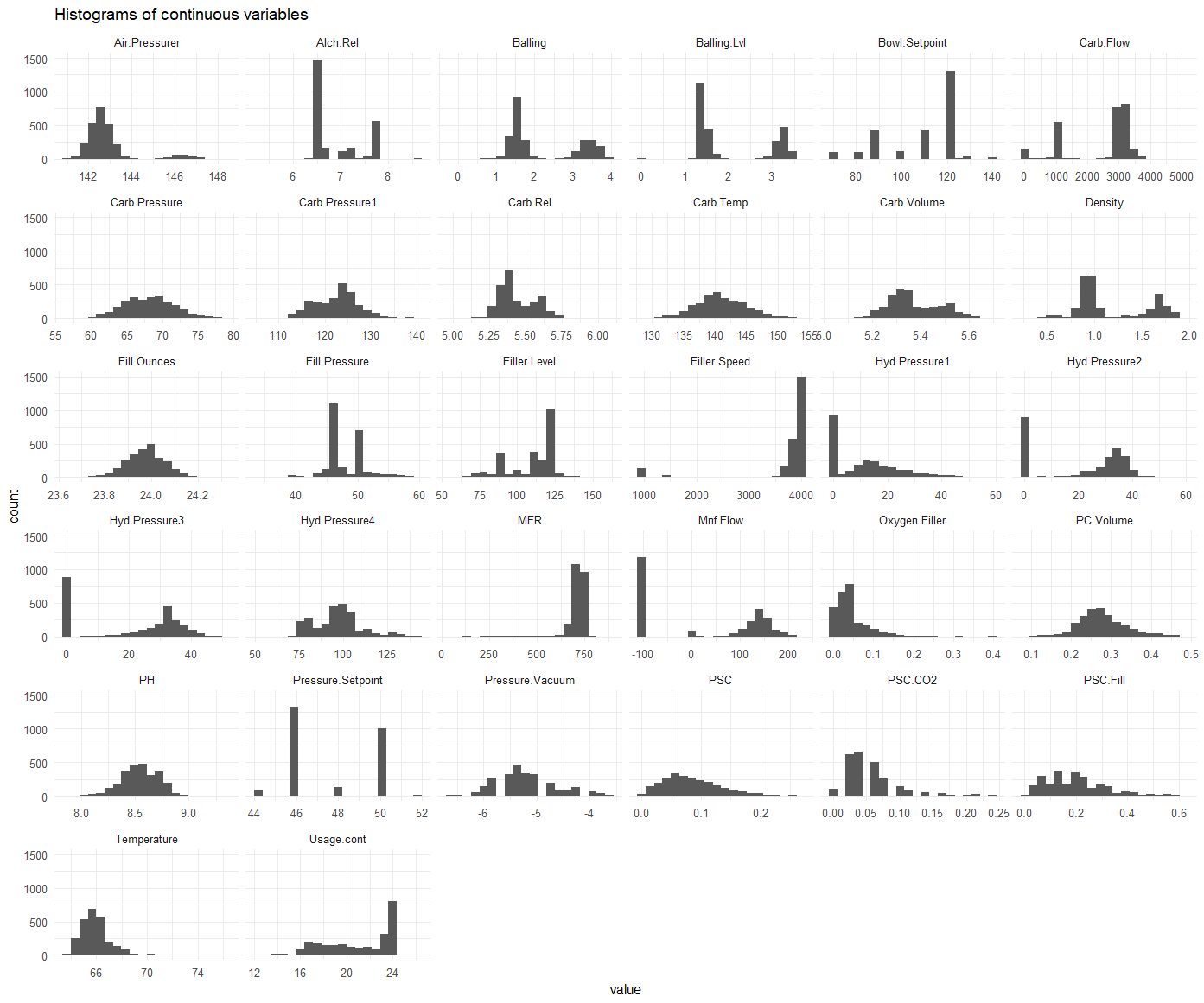
## Correlation between variables

In some cases, when multiple variables are provided as predictors for another, there can be relationships between the variables. This is known as collinearity, or multicollinearity. If there are relationships among the predictor variables themselves, it has a negative effect on some models' predictive power. Darker blues represent a strong, positive correlation between two variables and dark red represents a strong, negative correlation between two variables. Lighter color and white represent little or no correlation. In this case it is important to note that several features which show collinearity, which will be important to handle in data preparation and model selection.



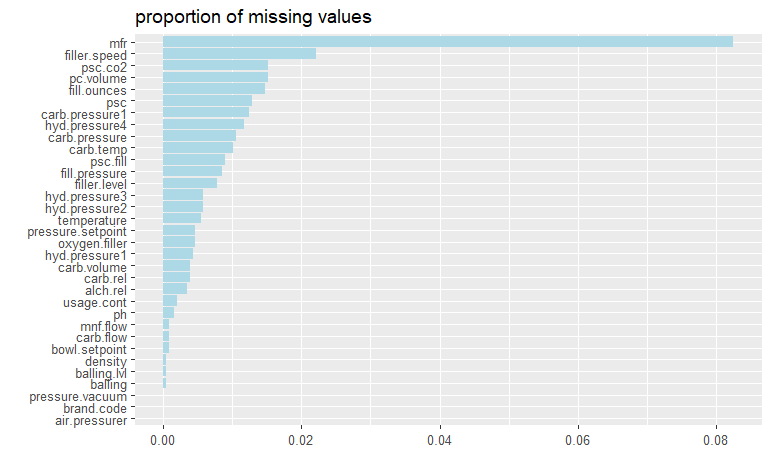
## Near Zero Variance

Near zero variance predictors are predictors for which most values of the predictor are a single value. These types of predictors are not beneficial to the model and can be removed, the variable that we found that had near zero variance was “Hyd.Pressure1”, below it is clear that most values for this variable were zero, followed by a distribution of significantly less frequent other values.



## Missing Values

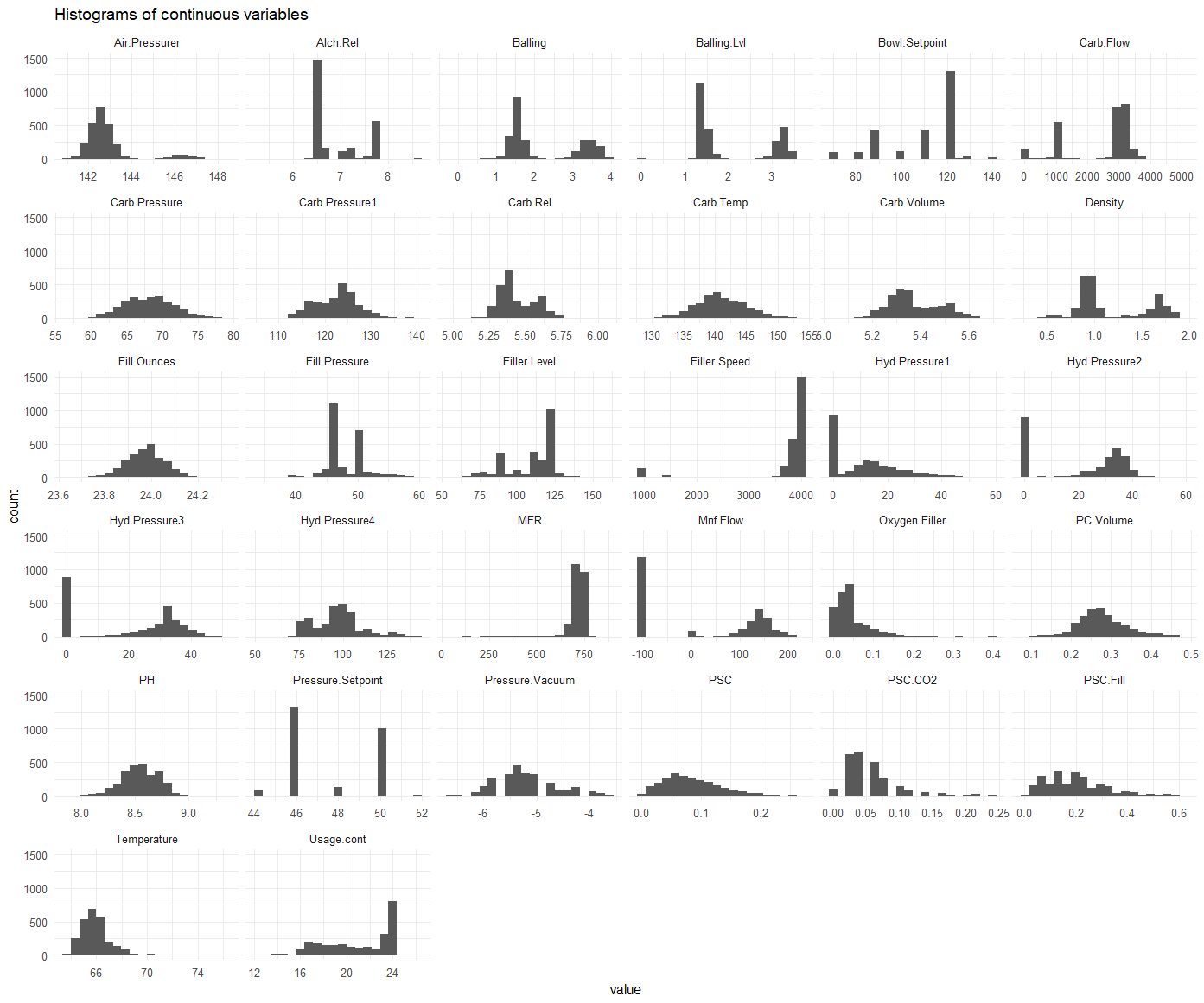
In the provided data there were values that were missing. Below is a chart showing the proportion of missing values. The variable “mfr” had the most missing values (approximately 8%).



## 

## Skewness

Many types of models are sensitive to variables that exhibit skewed distributions, i.e. those that do not exhibit a typical bell-shaped pattern but instead have a disproportionate number of high or low values. Below are some distributions that exhibited skewness.



## Outliers

The presence of outliers in a data set can also negatively impact some models. Outliers are typically defined as those points which lie beyond a certain number of standard deviations from the mean (typically 2, 2.5, or 3 times the standard deviation). We used two types of charts to identify outliers. The chart below on the left highlights the values that would be considered outliers based on standard deviations away from the mean. On the right, is what is called a violin plot. The violin plot is a way of showing the distribution of a variable. We can observe from the violin plot that what appears as outliers in the left hand plot, may not actually be outliers - it may just be a bimodal distribution - meaning that there are two main groups of values.

|  |  |
| --- | --- |
|  |  |

# 

# 

# Data Preparation

Before preparing the data for modeling, we considered several factors based on our exploratory data analysis that influenced our general approach to the problem:

1. There does not appear to be overtly linear relationships between the outcome variable and the vast majority of predictors. This indicates that a linear regression-based approach would likely not yield optimal results.
2. There is significant collinearity between pairs of variables.
3. About half of the variables exhibit heavy multicollinearity.
4. Only one variable can be considered near zero variance, so this wasn’t a strong influence on our modeling approach.
5. Missing values are present but not in significant enough quantities to strongly influence how we chose to model the data.
6. Six variables are heavily skewed and would likely need to be transformed for any models that are sensitive to this condition.
7. There doesn’t appear to be outliers in the dataset, rather there appear to be bimodal distributions.

Considering these factors, we decided to use two data sets for different purposes:

1. **Full data set**: With minimal modification, we used the full data set for those models which are robust to the confounding factors listed above (e.g. collinearity, skewness, and outliers).
2. **Reduced data set**: For models such as those based on linear regression, we prepared another data set that handles the above confounding factors. For example, skewed predictors were transformed, variables exhibiting multicollinearity were addressed, and some outliers were excluded.

Further details of the data preparation are expanded upon below, these sections may not be of interest to those who are not data scientists so if the reader is not a data scientist we recommend skipping to the “Regression Modeling” section for our results.

## Missing Values

There is a single observation that has 12 missing predictor values. We removed this observation as it likely would have contributed little information to the model. In addition, we removed the predictor (MFR) that contains a large number of missing values relative to the total number of observations; there were 212 missing values for this predictor, or 8.2% of the data.

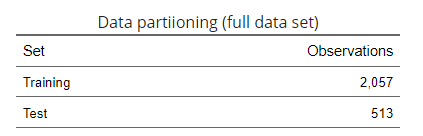
For the remaining missing values, there were multiple imputation methods available to fill in missing values. The number of missing values in our data set is relatively small, so we chose a relatively simple method (KNN imputation). This method uses a well-known statistical algorithm to find the “K” most closely related neighbors to observations having missing values. The value of K is selected by the modeler and for imputation purposes is quite arbitrary. In general, smaller values of K lead to faster but less stable predictions, while larger values of K are more computationally intensive but yield smoother results. We chose a value of 9 for , which is within the typical range for these applications and provides a good balance of performance versus speed.

## Dummy Variables

One additional step we took was to recode the categorical variable, “Brand.Code”, into so-called “dummy” variables. This process creates a series of variables that take either a 0 or 1 as a value; each variable indicates the presence (value of 1) or absence (value of 0) of that particular Brand.Code.

## Full Data Set

After imputation and creating dummy variables, our full set had the following observations:



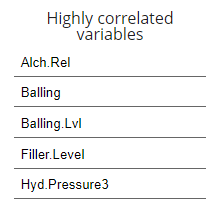
It is important to note that some models are sensitive to the numbers of observations compared to the number of parameters. Models that must invert linear matrices are mathematically unsolvable if the number of parameters exceeds the number of observations. Cross validation must also be taken into account when evaluating the observation-to-parameter count; for example, ten-fold CV only uses 90% of the observations for training. In our data set, we have 34 predictor variables and 513 rows in the smaller of the two data sets. Using ten-fold CV results in 461 rows still available to use for training, so the observation-to-parameter count isn’t something we have to worry about for this situation.

## Reduced Data Set

We generated a second, reduced set to use with models that are sensitive to irregularities. We removed one of the Brand.Code columns, as having all categories of a categorical variable will be problematic for linear-based models. We further prepared the data as described below.

### Collinearity

As we noted earlier, the data exhibits some collinearity and multicollinearity. Almost half the predictors exhibited strong multicollinearity. Since removing that many predictors would likely cause a detrimental loss of information, a different approach was warranted. We opted to use the approach presented by Kuhl and Johnson (2013) in which predictors are iteratively compared pairwise, removing those with the highest correlation value until a certain threshold is attained. While this procedure only addresses collinearity rather than multicollinearity, it can nonetheless yield significant improvements to model performance and stability. The following variables were found to be highly correlated and were removed:



### Near-zero variance

There was only a single predictor with NZV (Hyd.Pressure1). Since this condition can negatively impact linear regression-based models, we removed the variable from the reduced data set.

### Skewness

We first checked for zero or negative values for the variables that were earlier identified as skewed. If there are any such values, the Box-Cox transformation would be infeasible and a different method must be used. Since PSC.CO2 contains zero values, we used the Yeo-Johnson transformation instead. Because the caret package supports Yeo-Johnson natively, we performed the transformations as part of the preprocessing component during modeling. This has the advantage that predictions made based on our models will be automatically back-transformed rather than requiring manual transformation.

### Outliers

Based on the outlier plots from earlier, most of the data points beyond the three-standard-deviation cutoff appeared to be valid data points, though located on the extreme outer edges of the main body of data. One exception is Alch.Red, which includes six suspect points located along the left- and right-hand margins of the plot. The other possible exception is the data point with a pH value of 9.36. It is unclear whether these are truly outliers that should be excluded. But in either case, this variable was already identified as exhibiting high collinearity and has already been removed from the reduced data set.

## Data Partitioning

We split the data sets into two sets: one set to train the models and another to test the results of each model. This is standard practice that allows models to be compared to each other based on their performance with the test set - where the models are confronted with new data. The training data typically further undergoes “cross-validation” (CV), another common technique that trains the model over multiple iterations, at each iteration withholding a different proportion of the data to tune the model. After the final iteration, the model with the optimal tuning parameters is selected, and that model is then used to evaluate the previously withheld test data.

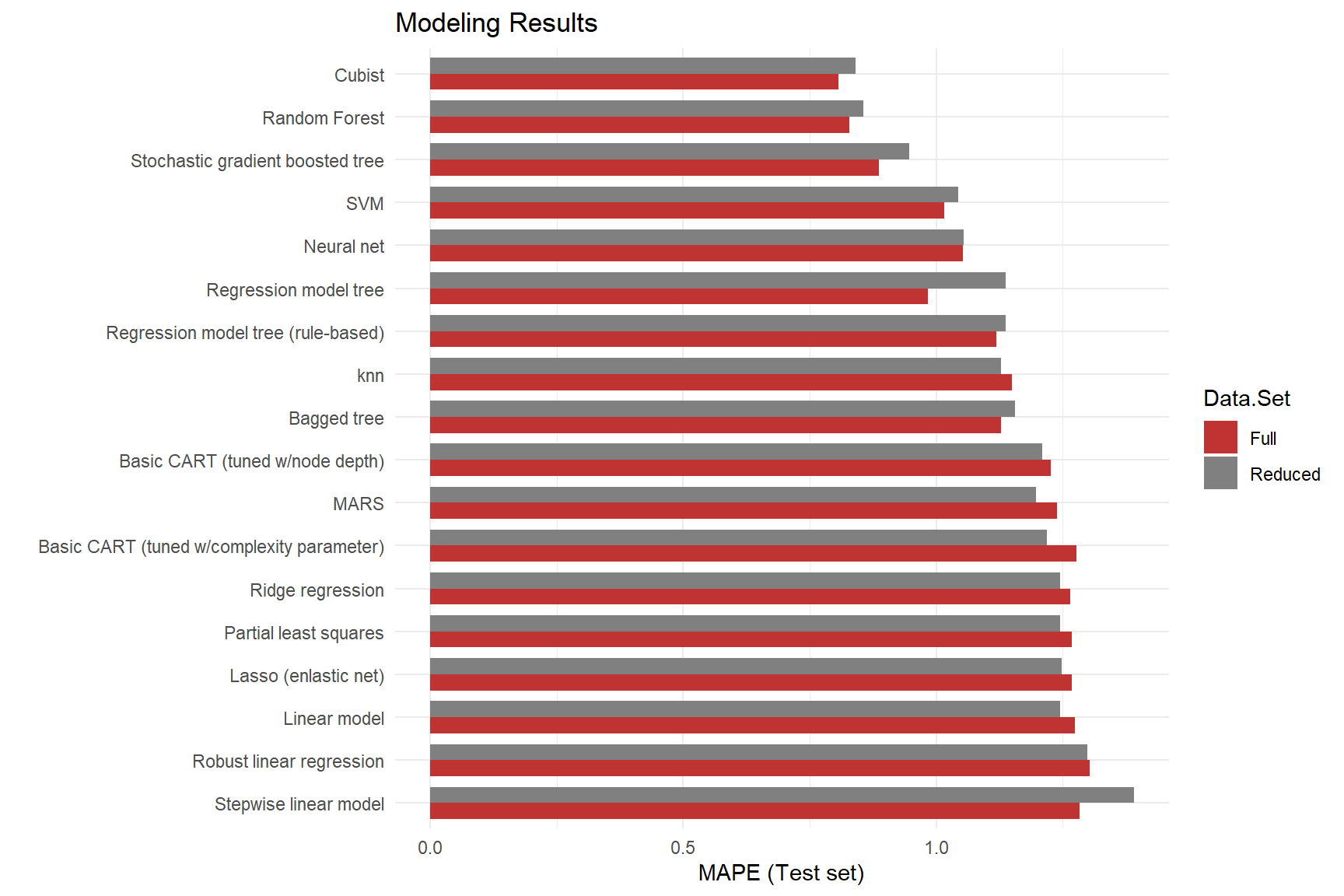
# 

# Regression Modeling

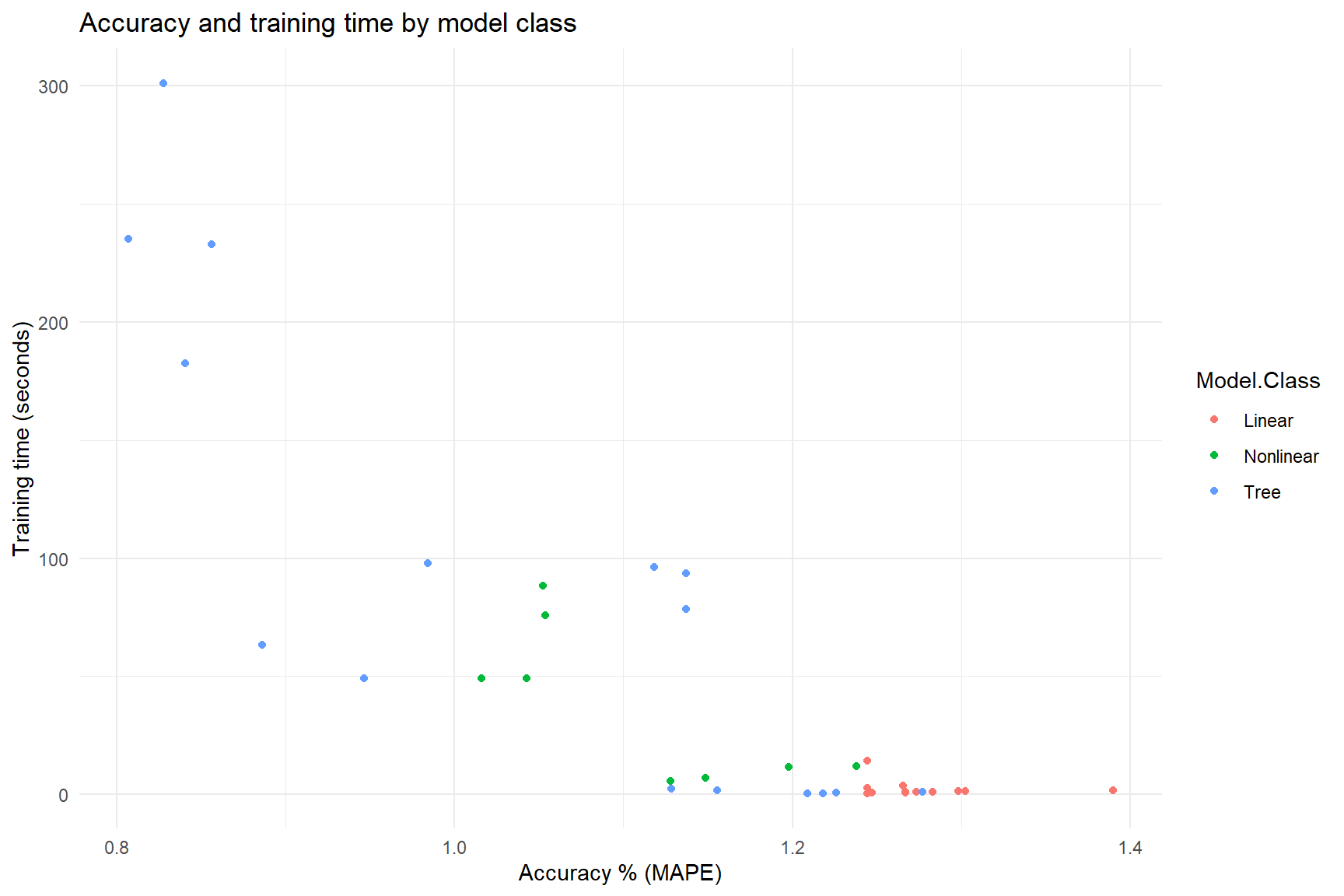
There are several different types of regression modeling approaches. At a high level, these approaches can be broken down into two categories - linear and nonlinear models. Linear models seek to establish linear relationships between the predictor variables and the response. Non-linear models do not rely on linear relationships. The tradeoff between linear and nonlinear models often is interpretability. Appendices [2](#_qlawil8tn7yp) & [3](#_icpfsqueeijc) contain charts outlining each modeling approach taken along with descriptions of each model.

## Modeling Results

Tree-based models were the best performers by a significant margin. Tree-based models are essentially combinations of logical statements mixed with different combinations of other models. Overall, nonlinear regression-based models outperformed linear based models. The Cubist and random forest models were the top performers, followed by the stochastic gradient boosted tree and regression model tree. It is not surprising that the tree-based models performed better when fed the full data set, as there was a richer feature set to inform the outcome. It was, however, surprising that the test data set outperformed the training set; typically models perform slightly worse at data they haven’t encountered before. This was the case with the linear and nonlinear regression-based models. The chart below shows the mean absolute percentage error (MAPE), which is just a measure of model performance, for each model.

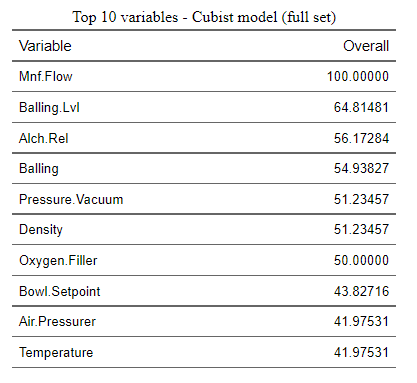


We also gauged accuracy against how long it took to train the various models (see plot below). As expected, the more accurate models generally took much longer to run than those that were less accurate. The model with the best MAPE score was also one of the longest to train.

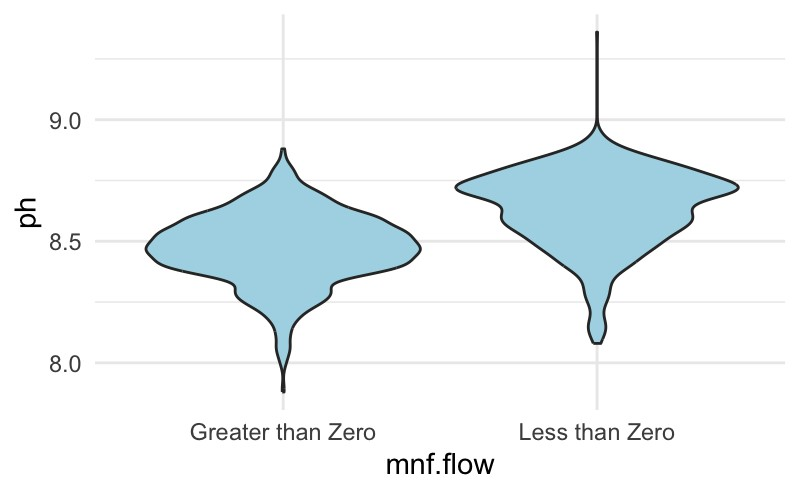


## Variable Importance

To aid with the interpretability of non-linear models, a metric called variable importance was used, below is a table of variable importances for our top performing model, which shows that “Mnf.Flow” is the most important predictor of pH. An interesting point of note is the presence of the highly correlated variables, variables which were removed from the reduced data set: Balling.Lvl, Alch.Rel, and Balling.

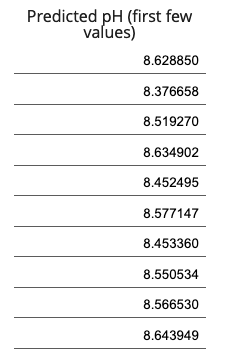


Upon discovering the high variable importance of “Mnf.Flow”, we looked further into this predictors relationship to pH. We binned the variable into two categories, greater than and less than zero. Our finding is that if the “Mnf.Flow” is greater than zero, the pH is more likely to fall within the critical range, or a pH of 8.5 +/- 0.1.



## Dataset to Predict

Finally, the full dataset Cubist model was run against the evaluation data provided to generate pH predictions and saved to .csv – we can see that most of the predictions are just along the threshold range of 8.5 representing the level where beverage consumption becomes bitter and poses potential health risks, highlighting the importance of accurate predictions. It should be noted that we conducted exploratory data analysis of the predictors for pH and found that the distributions were similar to the data we used to train and test the model - which gave us confidence in using our model to make our predictions.



# Conclusion

Different types of regression models are able to predict pH levels within an acceptable error – the Cubist model being the most accurate. Our findings determined that use of a full dataset provided the most accurate predictive results, with the “Mnf.Flow” variable by far the most important in our model. Upon exploring the “Mnf.Flow” and it’s relationship with pH, we found that values for “Mnf.Flow” greater than zero are more likely to fall within the critical pH range. For this reason we recommend that “Mnf.Flow” be constrained to greater than zero. In the future, we recommend imputing missing values with KNN-imputation, and with respect to new inputs, our findings suggest employing a model using the full dataset. We also suggest doing exploratory data analysis with any new datasets that predictions will be made from to ensure that the data used to train the models falls in the same ranges as the new predictor data. Further, we found that we were able to improve on our final model when we included only observations including a brand code, so we recommend to systems engineers requiring the brand code field as a data input.

# 

# References

Reddy A, Norris DF, Momeni SS, Waldo B, Ruby JD. The pH of beverages in the United States. J Am Dent Assoc. 2016 Apr;147(4):255-63. doi: 10.1016/j.adaj.2015.10.019. Epub 2015 Dec 2. PMID: 26653863; PMCID: PMC4808596.

Ashley, R. *Ask the Doctors - Is water with a high pH safe to drink?* <https://www.uclahealth.org/news/ask-the-doctors-is-water-with-a-high-ph-safe-to-drink>

Box, G. and Cox, D. (1964). *An Analysis of Transformations*. Journal of the Royal Statistical Society: Series B (Methodological), Volume 26, Issue 2, June 1964, Pages 211-243. <https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1964.tb00553.x>

Glen, S. (2023). “Variance Inflation Factor” From StatisticsHowTo.com: Elementary Statistics for the rest of us! <https://www.statisticshowto.com/variance-inflation-factor/>

Joanes, D. and Gill, C. (1998). *Comparing measures of sample skewness and kurtosis*. The Statistician, 47, pages 183-189.

Kaplan J. (2020). *fastDummies: Fast Creation of Dummy (Binary) Columns and Rows from Categorical Variables*. R package version 1.6.3, [https://CRAN.R-project.org/package=fastDummies](https://cran.r-project.org/package=fastDummies).

Kuhn M. (2022). *caret: Classification and Regression Training*. R package version 6.0-93, [https://CRAN.R-project.org/package=caret](https://cran.r-project.org/package=caret).

Kuhn, M. and Johnson, K. (2013). *Applied Predictive Modeling*. Springer Science+Business Media.

Meyer D. et al. (2022). *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. R package version 1.7-12, [https://CRAN.R-project.org/package=e1071](https://cran.r-project.org/package=e1071).

Petrie, A. (2020). *regclass: Tools for an Introductory Class in Regression and Modeling*. R package version 1.6, [https://CRAN.R-project.org/package=regclass](https://cran.r-project.org/package=regclass).

Torgo, L. (2016). *Data Mining with R, learning with case studies* (2nd ed.). Chapman and Hall/CRC. <http://ltorgo.github.io/DMwR2>.

van Buren, S. (2018). *Flexible Imputation of Missing Data* (2nd ed.). Chapman & Hall/CRC Interdisciplinary Statistic Series. <https://stefvanbuuren.name/fimd/sec-FCS.html>

van Buren, S. and Groothuis-Oudshoorn, K. (2011). *mice: Multivariate Imputation by Chained Equations in R*. Journal of Statistical Software, 45(3), 1-67. DOI 10.18637/jss.v045.i03.

van Buren, S., et al. (2023). *Multivariate Imputation by Chained Equations*. <https://cran.r-project.org/web/packages/mice/mice.pdf>.

Yeo, I. and Johnson, R. (2000). *A new family of power transformations to improve normality or symmetry*. Biometrika, Volume 87, Issue 4, December 2000, pages 954–959. <https://academic.oup.com/biomet/article-abstract/87/4/954/232908>.

# 

# Appendix 1: R Code

All of the R Code shown below can be found in our github repository as well as on Rpubs.

Github: <https://github.com/klgriffen96/summer23_data624/tree/main/project_2>

Rpubs 1: <https://rpubs.com/josh1den/data624_project2>

Rpubs 2: <https://rpubs.com/mmippolito/ippolito_data624_pj2>

Below is our main code file (Rpubs 2).

Load Data

First, we load the data from Github. We had some trouble reading the Excel files, so we converted them to CSV.

# Load training data (m=modeling)  
dfm\_raw <- read.csv('https://raw.githubusercontent.com/klgriffen96/summer23\_data624/main/project\_2/StudentData%20-%20TO%20MODEL.csv')  
  
# Load evaluation data  
dfe\_raw <- read.csv('https://github.com/klgriffen96/summer23\_data624/raw/main/project\_2/StudentEvaluation-%20TO%20PREDICT.csv')

Preliminary view

A first look at the data is as follows.

# Move outcome variable pH to front for easier access  
dfm <- dfm\_raw %>%  
 dplyr::select(PH, !matches('Brand.Code'), Brand.Code)  
str(dfm)

## 'data.frame': 2571 obs. of 33 variables:  
## $ PH : num 8.36 8.26 8.94 8.24 8.26 8.32 8.4 8.38 8.38 8.5 ...  
## $ Carb.Volume : num 5.34 5.43 5.29 5.44 5.49 ...  
## $ Fill.Ounces : num 24 24 24.1 24 24.3 ...  
## $ PC.Volume : num 0.263 0.239 0.263 0.293 0.111 ...  
## $ Carb.Pressure : num 68.2 68.4 70.8 63 67.2 66.6 64.2 67.6 64.2 72 ...  
## $ Carb.Temp : num 141 140 145 133 137 ...  
## $ PSC : num 0.104 0.124 0.09 NA 0.026 0.09 0.128 0.154 0.132 0.014 ...  
## $ PSC.Fill : num 0.26 0.22 0.34 0.42 0.16 0.24 0.4 0.34 0.12 0.24 ...  
## $ PSC.CO2 : num 0.04 0.04 0.16 0.04 0.12 0.04 0.04 0.04 0.14 0.06 ...  
## $ Mnf.Flow : num -100 -100 -100 -100 -100 -100 -100 -100 -100 -100 ...  
## $ Carb.Pressure1 : num 119 122 120 115 118 ...  
## $ Fill.Pressure : num 46 46 46 46.4 45.8 45.6 51.8 46.8 46 45.2 ...  
## $ Hyd.Pressure1 : num 0 0 0 0 0 0 0 0 0 0 ...  
## $ Hyd.Pressure2 : num NA NA NA 0 0 0 0 0 0 0 ...  
## $ Hyd.Pressure3 : num NA NA NA 0 0 0 0 0 0 0 ...  
## $ Hyd.Pressure4 : int 118 106 82 92 92 116 124 132 90 108 ...  
## $ Filler.Level : num 121 119 120 118 119 ...  
## $ Filler.Speed : int 4002 3986 4020 4012 4010 4014 NA 1004 4014 4028 ...  
## $ Temperature : num 66 67.6 67 65.6 65.6 66.2 65.8 65.2 65.4 66.6 ...  
## $ Usage.cont : num 16.2 19.9 17.8 17.4 17.7 ...  
## $ Carb.Flow : int 2932 3144 2914 3062 3054 2948 30 684 2902 3038 ...  
## $ Density : num 0.88 0.92 1.58 1.54 1.54 1.52 0.84 0.84 0.9 0.9 ...  
## $ MFR : num 725 727 735 731 723 ...  
## $ Balling : num 1.4 1.5 3.14 3.04 3.04 ...  
## $ Pressure.Vacuum : num -4 -4 -3.8 -4.4 -4.4 -4.4 -4.4 -4.4 -4.4 -4.4 ...  
## $ Oxygen.Filler : num 0.022 0.026 0.024 0.03 0.03 0.024 0.066 0.046 0.064 0.022 ...  
## $ Bowl.Setpoint : int 120 120 120 120 120 120 120 120 120 120 ...  
## $ Pressure.Setpoint: num 46.4 46.8 46.6 46 46 46 46 46 46 46 ...  
## $ Air.Pressurer : num 143 143 142 146 146 ...  
## $ Alch.Rel : num 6.58 6.56 7.66 7.14 7.14 7.16 6.54 6.52 6.52 6.54 ...  
## $ Carb.Rel : num 5.32 5.3 5.84 5.42 5.44 5.44 5.38 5.34 5.34 5.34 ...  
## $ Balling.Lvl : num 1.48 1.56 3.28 3.04 3.04 3.02 1.44 1.44 1.44 1.38 ...  
## $ Brand.Code : chr "B" "A" "B" "A" ...

summary(dfm)

## PH Carb.Volume Fill.Ounces PC.Volume   
## Min. :7.880 Min. :5.040 Min. :23.63 Min. :0.07933   
## 1st Qu.:8.440 1st Qu.:5.293 1st Qu.:23.92 1st Qu.:0.23917   
## Median :8.540 Median :5.347 Median :23.97 Median :0.27133   
## Mean :8.546 Mean :5.370 Mean :23.97 Mean :0.27712   
## 3rd Qu.:8.680 3rd Qu.:5.453 3rd Qu.:24.03 3rd Qu.:0.31200   
## Max. :9.360 Max. :5.700 Max. :24.32 Max. :0.47800   
## NA's :4 NA's :10 NA's :38 NA's :39   
## Carb.Pressure Carb.Temp PSC PSC.Fill   
## Min. :57.00 Min. :128.6 Min. :0.00200 Min. :0.0000   
## 1st Qu.:65.60 1st Qu.:138.4 1st Qu.:0.04800 1st Qu.:0.1000   
## Median :68.20 Median :140.8 Median :0.07600 Median :0.1800   
## Mean :68.19 Mean :141.1 Mean :0.08457 Mean :0.1954   
## 3rd Qu.:70.60 3rd Qu.:143.8 3rd Qu.:0.11200 3rd Qu.:0.2600   
## Max. :79.40 Max. :154.0 Max. :0.27000 Max. :0.6200   
## NA's :27 NA's :26 NA's :33 NA's :23   
## PSC.CO2 Mnf.Flow Carb.Pressure1 Fill.Pressure   
## Min. :0.00000 Min. :-100.20 Min. :105.6 Min. :34.60   
## 1st Qu.:0.02000 1st Qu.:-100.00 1st Qu.:119.0 1st Qu.:46.00   
## Median :0.04000 Median : 65.20 Median :123.2 Median :46.40   
## Mean :0.05641 Mean : 24.57 Mean :122.6 Mean :47.92   
## 3rd Qu.:0.08000 3rd Qu.: 140.80 3rd Qu.:125.4 3rd Qu.:50.00   
## Max. :0.24000 Max. : 229.40 Max. :140.2 Max. :60.40   
## NA's :39 NA's :2 NA's :32 NA's :22   
## Hyd.Pressure1 Hyd.Pressure2 Hyd.Pressure3 Hyd.Pressure4   
## Min. :-0.80 Min. : 0.00 Min. :-1.20 Min. : 52.00   
## 1st Qu.: 0.00 1st Qu.: 0.00 1st Qu.: 0.00 1st Qu.: 86.00   
## Median :11.40 Median :28.60 Median :27.60 Median : 96.00   
## Mean :12.44 Mean :20.96 Mean :20.46 Mean : 96.29   
## 3rd Qu.:20.20 3rd Qu.:34.60 3rd Qu.:33.40 3rd Qu.:102.00   
## Max. :58.00 Max. :59.40 Max. :50.00 Max. :142.00   
## NA's :11 NA's :15 NA's :15 NA's :30   
## Filler.Level Filler.Speed Temperature Usage.cont Carb.Flow   
## Min. : 55.8 Min. : 998 Min. :63.60 Min. :12.08 Min. : 26   
## 1st Qu.: 98.3 1st Qu.:3888 1st Qu.:65.20 1st Qu.:18.36 1st Qu.:1144   
## Median :118.4 Median :3982 Median :65.60 Median :21.79 Median :3028   
## Mean :109.3 Mean :3687 Mean :65.97 Mean :20.99 Mean :2468   
## 3rd Qu.:120.0 3rd Qu.:3998 3rd Qu.:66.40 3rd Qu.:23.75 3rd Qu.:3186   
## Max. :161.2 Max. :4030 Max. :76.20 Max. :25.90 Max. :5104   
## NA's :20 NA's :57 NA's :14 NA's :5 NA's :2   
## Density MFR Balling Pressure.Vacuum   
## Min. :0.240 Min. : 31.4 Min. :-0.170 Min. :-6.600   
## 1st Qu.:0.900 1st Qu.:706.3 1st Qu.: 1.496 1st Qu.:-5.600   
## Median :0.980 Median :724.0 Median : 1.648 Median :-5.400   
## Mean :1.174 Mean :704.0 Mean : 2.198 Mean :-5.216   
## 3rd Qu.:1.620 3rd Qu.:731.0 3rd Qu.: 3.292 3rd Qu.:-5.000   
## Max. :1.920 Max. :868.6 Max. : 4.012 Max. :-3.600   
## NA's :1 NA's :212 NA's :1   
## Oxygen.Filler Bowl.Setpoint Pressure.Setpoint Air.Pressurer   
## Min. :0.00240 Min. : 70.0 Min. :44.00 Min. :140.8   
## 1st Qu.:0.02200 1st Qu.:100.0 1st Qu.:46.00 1st Qu.:142.2   
## Median :0.03340 Median :120.0 Median :46.00 Median :142.6   
## Mean :0.04684 Mean :109.3 Mean :47.62 Mean :142.8   
## 3rd Qu.:0.06000 3rd Qu.:120.0 3rd Qu.:50.00 3rd Qu.:143.0   
## Max. :0.40000 Max. :140.0 Max. :52.00 Max. :148.2   
## NA's :12 NA's :2 NA's :12   
## Alch.Rel Carb.Rel Balling.Lvl Brand.Code   
## Min. :5.280 Min. :4.960 Min. :0.00 Length:2571   
## 1st Qu.:6.540 1st Qu.:5.340 1st Qu.:1.38 Class :character   
## Median :6.560 Median :5.400 Median :1.48 Mode :character   
## Mean :6.897 Mean :5.437 Mean :2.05   
## 3rd Qu.:7.240 3rd Qu.:5.540 3rd Qu.:3.14   
## Max. :8.620 Max. :6.060 Max. :3.66   
## NA's :9 NA's :10 NA's :1

kable(table(sapply(dfm\_raw, class)), col.names=c("type","count")) |> kable\_styling()

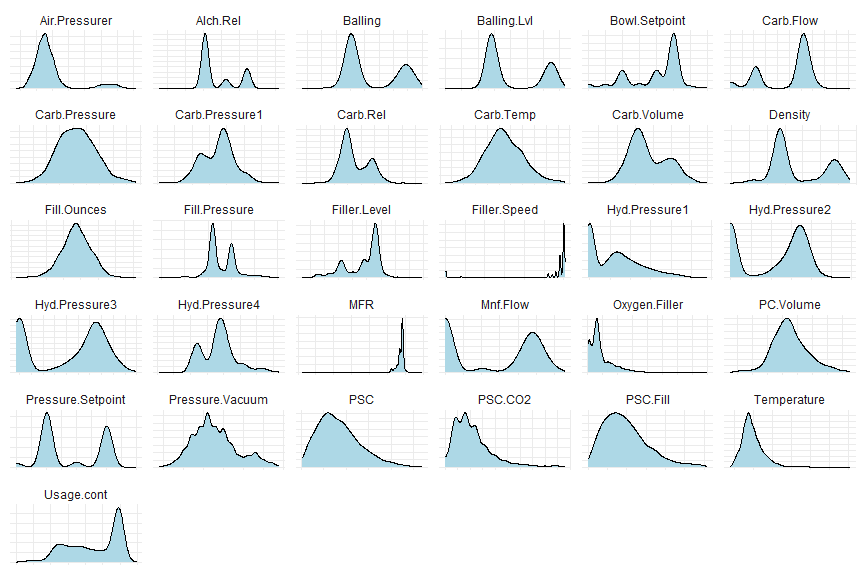
|  |  |
| --- | --- |
| **type** | **count** |
| character | 1 |
| integer | 4 |
| numeric | 28 |

We note several things from a cursory first glance:

1. There are 2571 observations in 33 variables.
2. The outcome variable, pH, is continuous, so this is a regression problem.
3. Of the 32 predictors, 31 are quantitative variables, and there is a single nominal categorical variable (Brand.Code).
4. There are a number of missing values but not at a rate that would be debilitating to fitting a model to the data. Additional investigation will be done to determine how best to handle these data points.
5. Little information was given about the data, so we can’t infer units for the variables (for example, it is unknown whether the temperature and pressure variables are in English or metric units).

Let’s take a look at the distribution of the numerical predictors.

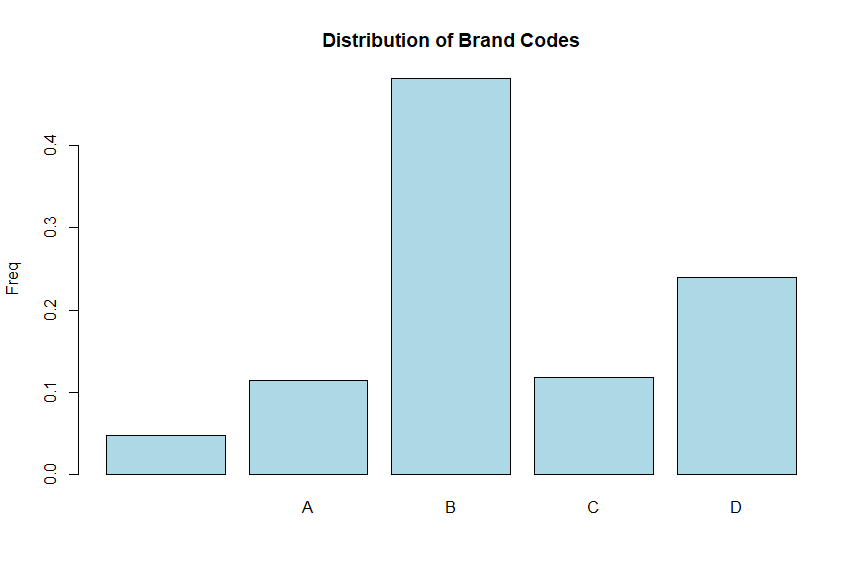
dfm |>  
 select(-c(PH,Brand.Code)) |>  
 gather(key = "predictor", value = "value") |>  
 ggplot(aes(x = value)) +  
 geom\_density(fill="lightblue") +  
 facet\_wrap(~ predictor, scales = "free") +  
 theme\_minimal() +   
 theme(axis.text.x = element\_blank(),  
 axis.text.y = element\_blank(),  
 axis.title.x = element\_blank(),  
 axis.title.y = element\_blank())



We can observe that there is either bimodality or skew in most of the variables, suggesting a nonlinear model is appropriate. Because of the bimodality, violin plots are preferable than boxplots to observe any outliers,

We’ll need to do a small amount of data preparation first: The categorical variable Brand.Code will need to be factored into numerical levels, as some functions that we’ll use later requires it (for example near-zero variance calculations, seen later). This will only be useful for initial data exploration; for modeling purposes, we’ll need to handle this variable differently (see Data Preparation below).

dfm |>  
 select(where(is.character)) |>  
 table() |>  
 prop.table() |>  
 barplot(main="Distribution of Brand Codes", col="lightblue", ylab="Freq")

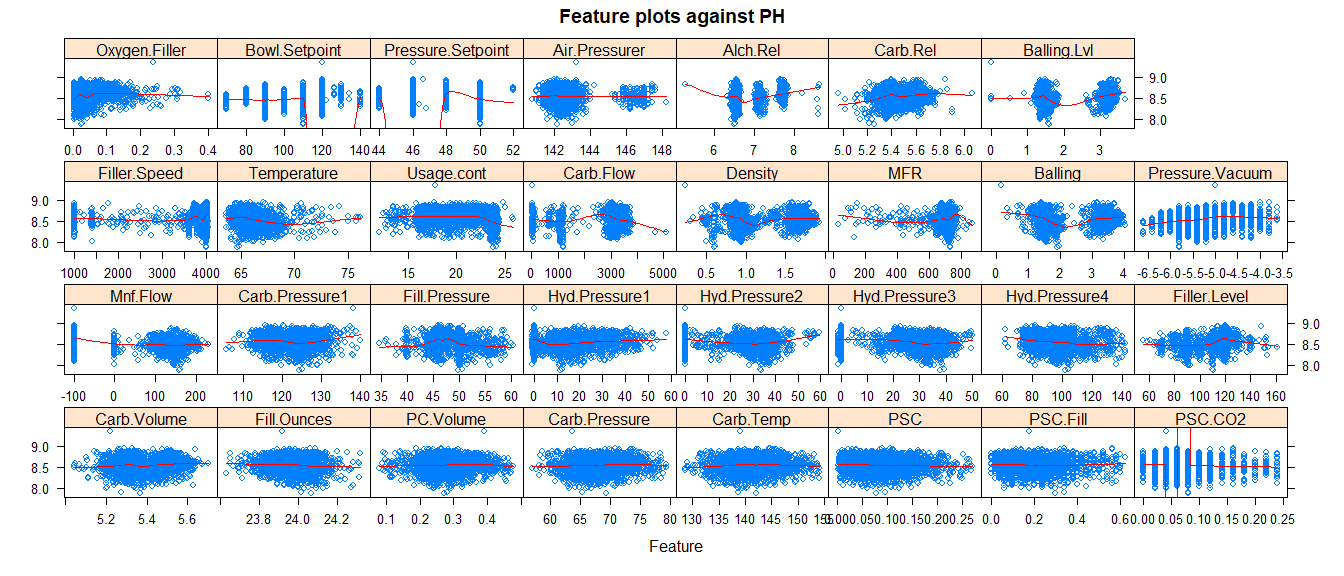


# Factor categorical variable Brand.Code  
dfm$Brand.Code <- factor(dfm$Brand.Code)

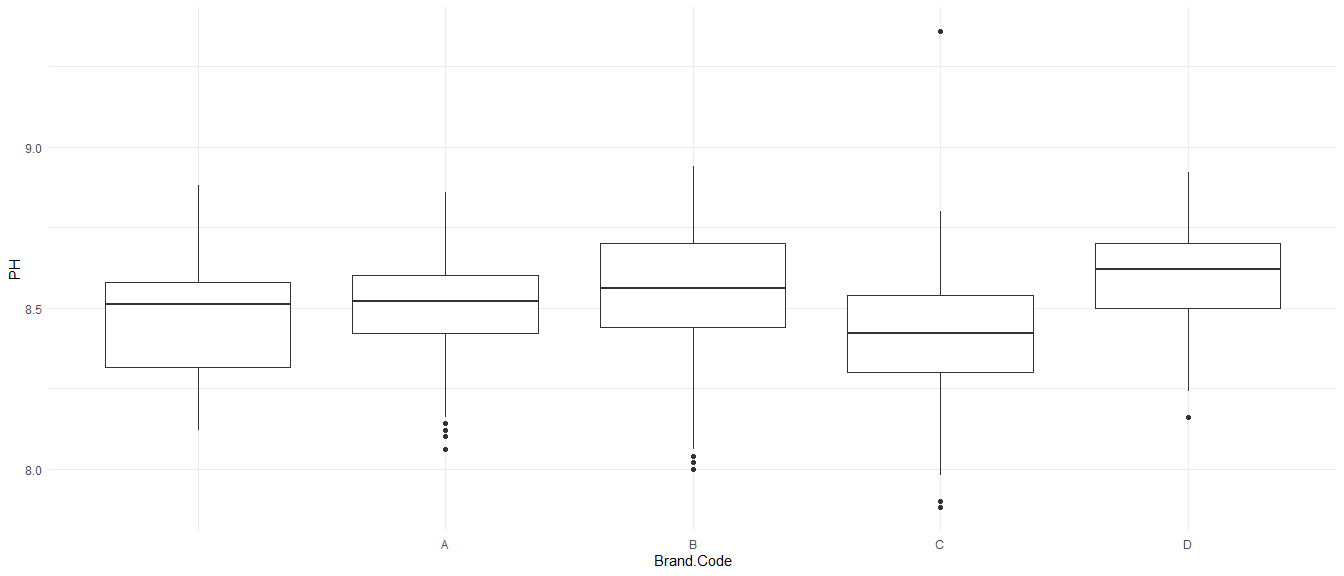
Feature plots

To visually evaluate our data set, we’ll generate feature plots of the continuous variables and a boxplot of the single categorical variable (Brand.Code). These plot show the relationship of each continuous variable against the outcome variable (pH) and can be used to detect how individual predictors influence the outcome.

# Feature plots - exclude the last column, which is Brand.Code (a categorical variable)  
featurePlot(x=dfm[,2:(ncol(dfm)-1)], y=dfm$PH, plot='scatter', main='Feature plots against PH', type=c('p', 'smooth'), span=0.5, layout=c(8, 4), col.line='red')



# Boxplot of Brand.Code against PH  
dfm %>%  
 filter(!is.na(Brand.Code) & !is.na(PH)) %>%  
 ggplot(aes(x=Brand.Code, y=PH)) +  
 geom\_boxplot()



At first glance there doesn’t appear to be any strong relationship between pH and any of the predictors. There is a slightly positive relationship between pH and some variables like pressure vacuum, but in general no strong trends are evident.

We also note that some of the observations have a blank Brand.Code, which we’ll have to handle during data preparation.

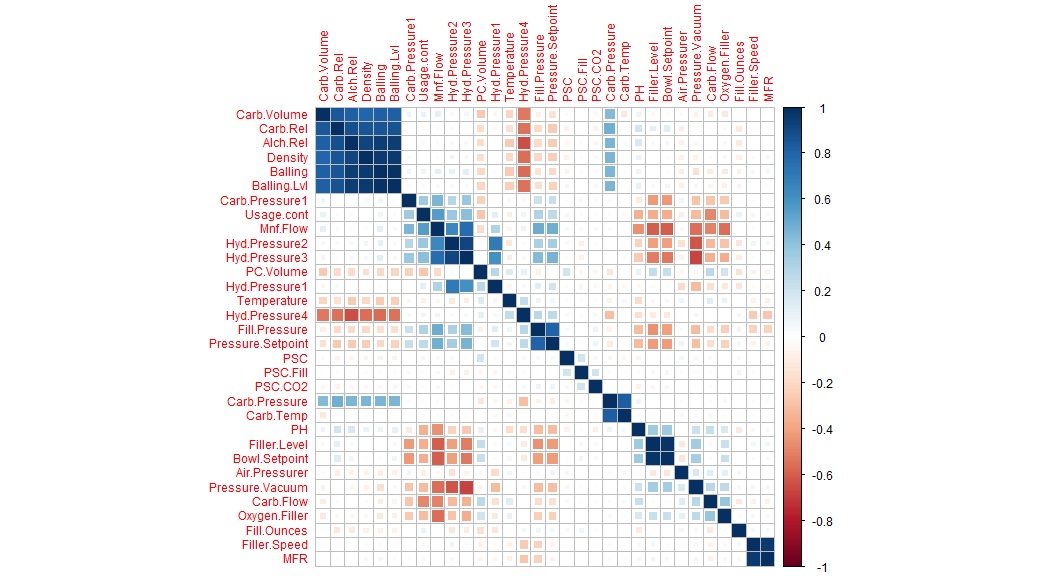
Collinearity

We’ll also look for collinearity among variables. Collinearity is a problem for linear regression-based models, as these models can generate severely inflated coefficients for predictors that are strongly collinear. In these cases, we have several options:

1. Retain only a single predictor among those that are collinear relative to each other.
2. Transform the predictors such that collinearity is minimized, for example using principle component analysis (PCA), partial least squares (PLS), or linear discriminant analysis (LDA).
3. Employ a model that is insensitive (or less sensitive) to collinear predictors, for example ridge, lasso, PLS, or multivariate adaptive regression splines (MARS).

To examine collinearity in our data set, we’ll use the corrplot() function from the package of the same name to generate a correlation plot. This shows strongly correlated variables in darker colors and with larger squares. Positive correlations are shown in blue, while negative correlations are in red. Correlation values of 1.0 indicate perfect positive correlation between variables, while values of -1.0 indicate a perfect inverse correlation. The plot is clustered by variables that exhibit strong correlations with each other, which can provide some indication of multicollinearity among groups of variables.

# Generate corr plot for pairwise correlations  
corr1 <- cor(dfm[,1:(ncol(dfm)-1)], use='complete')  
corrplot::corrplot(corr1, method='square', order='hclust', type='full', diag=T, tl.cex=0.75, cl.cex=0.75)



As shown in the correlation plot, there are some strong correlations among the following six variables:

* Carb.Volume
* Carb.Rel
* Alch.Rel
* Density
* Bailing
* Bailing.Lvl

It may be advantageous to remove one or more of these variables, transform them to remove collinearity, or choose a model that is robust to collinearity. Interestingly, these same six variables exhibit a relatively strong inverse relationship with Hyd.Pressure4.

Multicollinearity

In addition to collinearity, which evaluates relationships between pairs of predictors, we’ll look at multicollinearity, which gives a general indication of how much more each predictor is related to the other predictors than to the response variable. To evaluate multicollinearity, we’ll fit a simple linear model to the data using the full set of predictors. Then, we’ll use the vif() function in the cars package to estimate the variance inflation factor (VIF), which is a statistic purpose-built to measure multicollinearity among variables. Values less than or equal to 1 indicate no multicollinearity, values between 1 and 5 indicate moderate multicollinearity, and values greater than 5 indicate strong multicollinearity (Glen, 2023).

# Examine multicollinearity by generating a simple linear model of the data,  
# then looking at the variance inflation factor (VIF) of each variable.  
# VIF <= 1: low multicollinearity  
# 1 < VIF <= 5: moderate multicollinearity  
# VIF > 5: high multicollinearity  
lmod <- lm(PH ~ ., data=dfm)  
dfvif <- data.frame(vif(lmod))  
dfvif <- dfvif %>%  
 mutate(Predictor=row.names(dfvif)) %>%  
 rename(VIF=GVIF) %>%  
 dplyr::select(Predictor, VIF) %>%  
 arrange(desc(VIF)) %>%  
 mutate(Rank=rank(-VIF))  
dfvif %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Variance Inflation Factor')

Variance Inflation Factor

| Predictor | VIF | Rank |
| --- | --- | --- |
| Balling.Lvl | 196.699873 | 1 |
| Balling | 168.721067 | 2 |
| Brand.Code | 161.462006 | 3 |
| Carb.Pressure | 43.629727 | 4 |
| Carb.Temp | 35.744235 | 5 |
| Alch.Rel | 33.466584 | 6 |
| Bowl.Setpoint | 26.051364 | 7 |
| Filler.Level | 25.385854 | 8 |
| Carb.Volume | 17.895899 | 9 |
| Density | 16.438808 | 10 |
| Hyd.Pressure3 | 12.964503 | 11 |
| MFR | 11.435430 | 12 |
| Filler.Speed | 11.224377 | 13 |
| Hyd.Pressure2 | 10.635457 | 14 |
| Carb.Rel | 7.483881 | 15 |
| Mnf.Flow | 4.979607 | 16 |
| Pressure.Vacuum | 4.289511 | 17 |
| Fill.Pressure | 3.741073 | 18 |
| Pressure.Setpoint | 3.486682 | 19 |
| Hyd.Pressure1 | 3.000519 | 20 |
| Hyd.Pressure4 | 2.686600 | 21 |
| Carb.Flow | 2.256335 | 22 |
| Usage.cont | 1.773788 | 23 |
| PC.Volume | 1.712473 | 24 |
| Oxygen.Filler | 1.579977 | 25 |
| Carb.Pressure1 | 1.467316 | 26 |
| Temperature | 1.390864 | 27 |
| Air.Pressurer | 1.234829 | 28 |
| Fill.Ounces | 1.181213 | 29 |
| PSC | 1.163257 | 30 |
| PSC.Fill | 1.112048 | 31 |
| PSC.CO2 | 1.067962 | 32 |

As shown in the table almost half (15) of the variables have a VIF greater than 5, which indicate that they exhibit strong multicollinearity. This will inform our decision on the type of model to employ.

It should be noted that, while collinearity and multicollinearity adversely affect a linear regression-based model’s capacity to generate useful information about the contribution of each predictor to the outcome, these conditions don’t impede the model’s performance. Therefore, the presence of collinear or multicollinear predictors don’t automatically preclude us from using these types of models. **[need citation]**

Near-zero variance

We’ll look for variables having near-zero variance (NZV) since some models are sensitive to this condition. A variable exhibiting NZV is characterized as having only a single value (or a very small number of values) across the entire range of observations. These types of “degenerate” distributions can be problematic for models such as those based on linear regression. If NZV variables are observed, either they should be removed or a model that is immune to NZV variables should be used (e.g. tree-based models).

To calculate NZV, we’ll use the NearZeroVar() function from the caret package. This calculates the ratio of the frequency of the most common value in each variable to that of the next most common value. If the ratio is high, this indicates NZV conditions, and the variable may need special handling if a model that is sensitive to NZV variables are being used (Kuhn, 2022).

The following table shows the frequency ratios of variables in the data set in descending order.

# Look for NZV variables  
nzv <- nearZeroVar(dfm, saveMetrics=T)  
nzv$Variable <- row.names(nzv)  
nzv %>% dplyr::select(Variable, everything()) %>%  
 arrange(desc(freqRatio)) %>%  
 flextable() %>%  
 width(width = 1) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Frequency ratios of variables')

Frequency ratios of variables

| Variable | freqRatio | percentUnique | zeroVar | nzv |
| --- | --- | --- | --- | --- |
| Hyd.Pressure1 | 31.111111 | 9.5293660 | FALSE | TRUE |
| Hyd.Pressure3 | 11.450704 | 7.4679113 | FALSE | FALSE |
| Hyd.Pressure2 | 7.271028 | 8.0513419 | FALSE | FALSE |
| Bowl.Setpoint | 2.990847 | 0.4278491 | FALSE | FALSE |
| Brand.Code | 2.014634 | 0.1944769 | FALSE | FALSE |
| Fill.Pressure | 1.762931 | 4.2007001 | FALSE | FALSE |
| Carb.Flow | 1.586207 | 20.7312330 | FALSE | FALSE |
| Pressure.Vacuum | 1.389728 | 0.6223259 | FALSE | FALSE |
| Pressure.Setpoint | 1.319361 | 0.3111630 | FALSE | FALSE |
| Balling.Lvl | 1.294872 | 3.1894205 | FALSE | FALSE |
| Oxygen.Filler | 1.266667 | 13.1466356 | FALSE | FALSE |
| MFR | 1.222222 | 22.8315830 | FALSE | FALSE |
| PSC | 1.211538 | 5.0175029 | FALSE | FALSE |
| Balling | 1.193182 | 8.4402956 | FALSE | FALSE |
| Fill.Ounces | 1.163043 | 3.5783742 | FALSE | FALSE |
| Density | 1.108374 | 3.0338390 | FALSE | FALSE |
| Usage.cont | 1.105263 | 18.7086737 | FALSE | FALSE |
| PC.Volume | 1.100000 | 17.6584986 | FALSE | FALSE |
| Alch.Rel | 1.098315 | 2.0614547 | FALSE | FALSE |
| Air.Pressurer | 1.093407 | 1.2446519 | FALSE | FALSE |
| PSC.Fill | 1.091892 | 1.2446519 | FALSE | FALSE |
| Filler.Level | 1.086957 | 11.2018670 | FALSE | FALSE |
| PSC.CO2 | 1.078303 | 0.5056398 | FALSE | FALSE |
| PH | 1.078125 | 2.0225593 | FALSE | FALSE |
| Carb.Volume | 1.055556 | 3.9284325 | FALSE | FALSE |
| Mnf.Flow | 1.048443 | 18.9420459 | FALSE | FALSE |
| Filler.Speed | 1.045161 | 9.4904706 | FALSE | FALSE |
| Temperature | 1.040000 | 2.1781408 | FALSE | FALSE |
| Carb.Pressure1 | 1.031746 | 5.4453520 | FALSE | FALSE |
| Carb.Temp | 1.016667 | 4.7841307 | FALSE | FALSE |
| Carb.Pressure | 1.016393 | 4.1229094 | FALSE | FALSE |
| Carb.Rel | 1.011583 | 1.6336056 | FALSE | FALSE |
| Hyd.Pressure4 | 1.008264 | 1.5558149 | FALSE | FALSE |

As shown, only a single variable (Hyd.Pressure1) exhibited a high enough frequency ratio to be classified as an NZV variable. We’ll examine this variable in a bit more detail.

# Generate table of top unique values of Hyd.Pressure1  
hpfreq <- dfm %>%  
 group\_by(Hyd.Pressure1) %>%  
 summarize(Unique.count=n()) %>%  
 arrange(desc(Unique.count))  
  
# Tally up values other than the top 2  
others <- hpfreq %>%  
 filter(Unique.count < hpfreq[[2,'Unique.count']]) %>%  
 summarize(Unique.count=sum(Unique.count))  
  
# Add the "other values" row to the table  
hpfreq <- hpfreq %>%  
 head(2) %>%  
 rbind(data.frame(Hyd.Pressure1='other values', Unique.count=as.numeric(others))) %>%  
 rename(Value=Hyd.Pressure1) %>%  
 mutate(Percent=round(100 \* Unique.count / nrow(dfm), 2))  
  
# Display the table  
hpfreq %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Hyd.Pressure1')

Hyd.Pressure1

| Value | Unique.count | Percent |
| --- | --- | --- |
| 0 | 840 | 32.67 |
| 12.4 | 27 | 1.05 |
| other values | 1,704 | 66.28 |

As indicated in the table above, a large proportion of the observations have zero values (32.7%). Without knowing anything about how the data was collected or what the variable indicates, it is difficult to know the best way to treat the variable in terms of modeling The best we can do is make an assumption that zero values are correctly entered, structurally sound, and have meaning in the context of the data Based on these assumptions, this variable is a candidate to be dropped if a linear regression-based model is used.

Missing values

Now we’ll examine missing values in the data set. As noted earlier, there aren’t that many, but some models are sensitive to missing values. As such, the following options are available:

1. Discard observations that have missing values for many of the predictors.
2. Discard entire predictors that have many values missing across a great number of observations.
3. Impute (fill in) the missing values using any of a number of statistical methods available (e.g. mean, median, or K nearest neighbor modeling).

We’ll take a first look at missing values on a column-by-column basis.

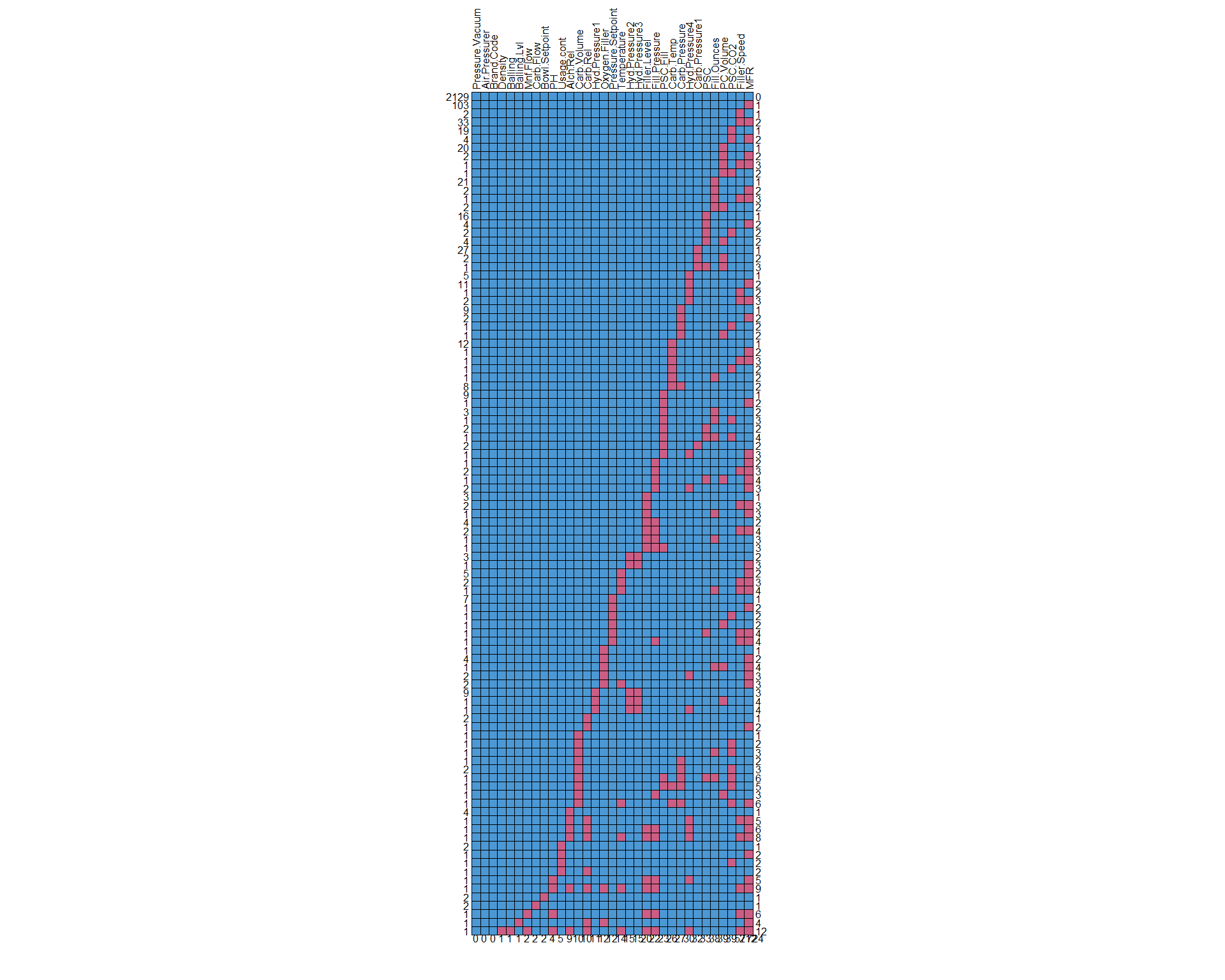
# Count NAs per column  
missing\_values <- data.frame(colSums(is.na(dfm)))  
missing\_values$Variable <- row.names(missing\_values)  
colnames(missing\_values) <- c('Missing.values', 'Variable')  
missing\_values %>%  
 dplyr::select(Variable, Missing.values) %>%  
 arrange(desc(Missing.values)) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Missing values')

Missing values

| Variable | Missing.values |
| --- | --- |
| 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 |
| Pressure.Vacuum | 0 |
| Air.Pressurer | 0 |
| Brand.Code | 0 |

A good way to visualize missing values is to use the md.pattern() function from the mice package (van Buren, 2011). This function produces a plot that shows patterns of missing data in order of increasing missing information. The numbers along the left indicate the number of observations that correspond to the pattern indicated by that row of squares. Blue squares indicate no missing values, while red squares indicate missing values. The numbers on the far right indicate the number of variables with missing data for that set of observations.

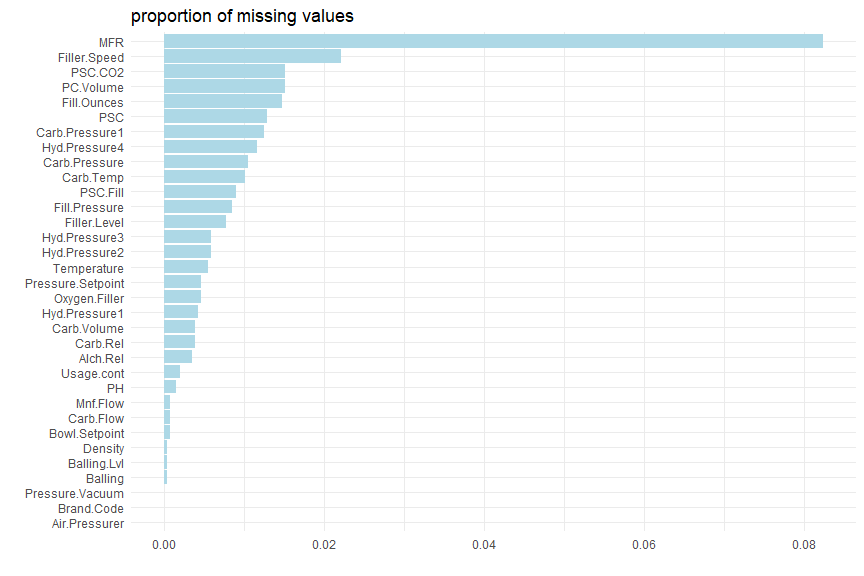
# Missing value patterns  
invisible(md.pattern(dfm, rotate.names=T, plot=T))



As shown, there are 2129 observations for which no values are missing, or 82.8% of the data. We note that there are only a few rows with many red squares, indicating that there aren’t many observations missing a significant number of variables. There is only a single observation that had missing values numbering in the double digits of predictors (12); this row is likely a good candidate for removal. In addition, one of the predictors (MFR) contained a significant number of missing values across all observations (212 values, or 8.2% of the data) and should likely be removed. The remaining data can be imputed (filled in) using one of the statistical methods listed above.

Another way to see this clearly is with a column chart.

nas <- colMeans(is.na(dfm))  
  
data.frame(variable = names(nas), missing = nas, row.names = NULL) |>  
 ggplot(aes(y=reorder(variable,missing), x=missing)) +  
 geom\_col(fill="lightblue") +  
 ggtitle('proportion of missing values') + xlab('') + ylab('')



An additional option is to use multivariate imputation by chained equations (MICE) (van Buren et al., 2023). MICE uses the fully conditional specification (FCS), which imputes each variable using a separate method. The method used can either be specified by the modeler or chosen by a set of defaults, depending on the kind of variable. For example, there are default methods for continuous variables and those for categorical variables, with additional methods for categorical variables that depend on the number of factors.

Skewness

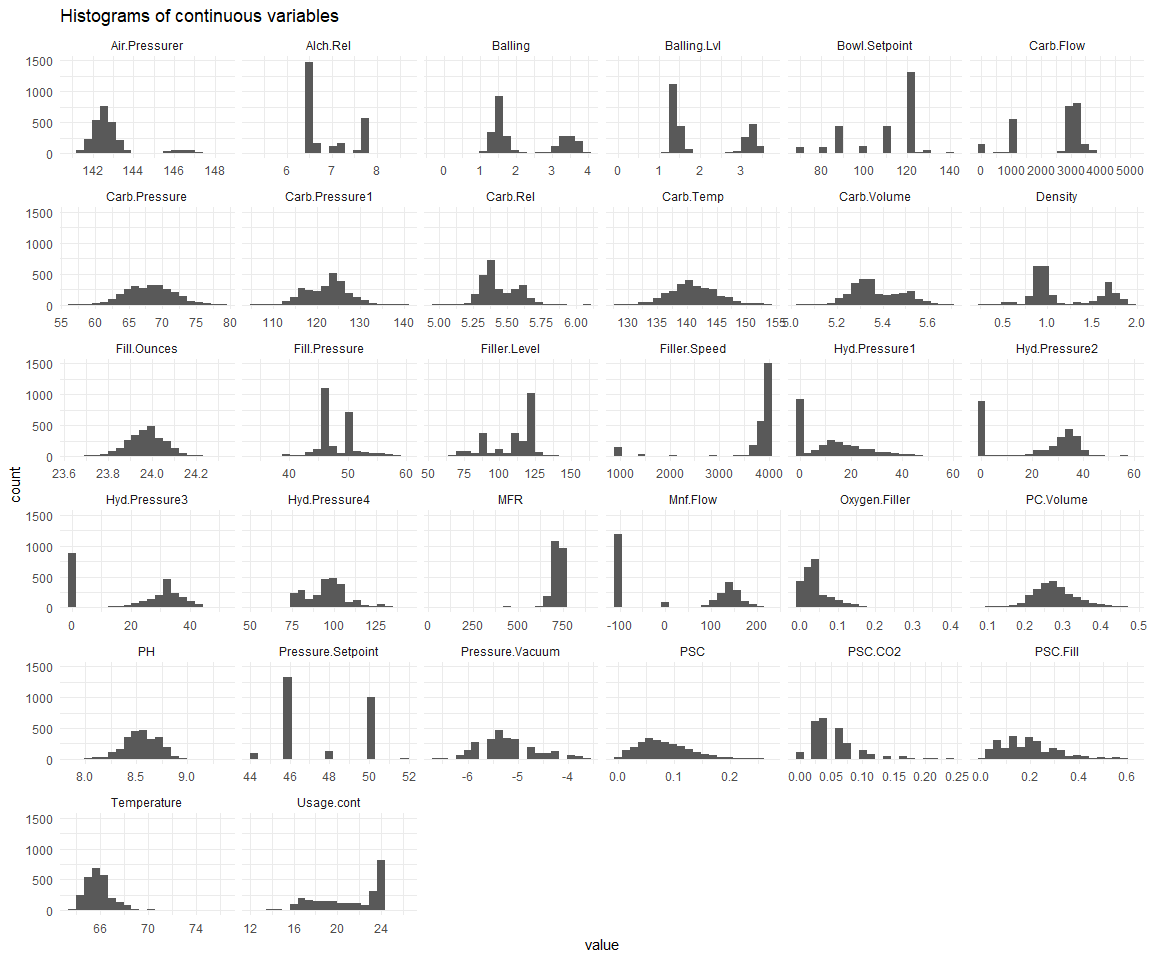
Many types of models are sensitive to variables that exhibit skewed distributions, i.e. those that do not exhibit a typical bell-shaped pattern but instead have a disproportionate number of high or low values. Linear regression-based models are particularly susceptible to skewness.

To reduce skewness, skewed variables can often be mathematically transformed, for example by taking the natural logarithm or using a method such as the Box-Cox transformation. This method uses an empirical means to find a transformation of the data (Box and Cox, 1964).

Box-Cox transformations only work on positive values, so if zero or negative values are encountered a different method must be used. If only zero values are observed, a constant (typically 1) can be added to the value before applying the Box-Cox transformation; however, there is debate on what constant to use in these cases. Alternatively, if both zero and negative values are present, a method such as one proposed by Yeo and Johnson (Yeo & Johnson, 2000) can be employed to transform the variable. The Yeo-Johnson method can accept positive, negative, or zero values and, thus, is more robust when values aren’t guaranteed to be positive.

To visually examine skewness, we’ll generate histograms of the continuous variables.

# Generate histograms of continuous variables  
dfm %>%  
 dplyr::select(c(1:(ncol(dfm)-1))) %>%  
 gather() %>%  
 ggplot(aes(x=value)) +  
 geom\_histogram(bins=20) +  
 facet\_wrap(~key, scales = 'free\_x') +  
 ggtitle('Histograms of continuous variables')



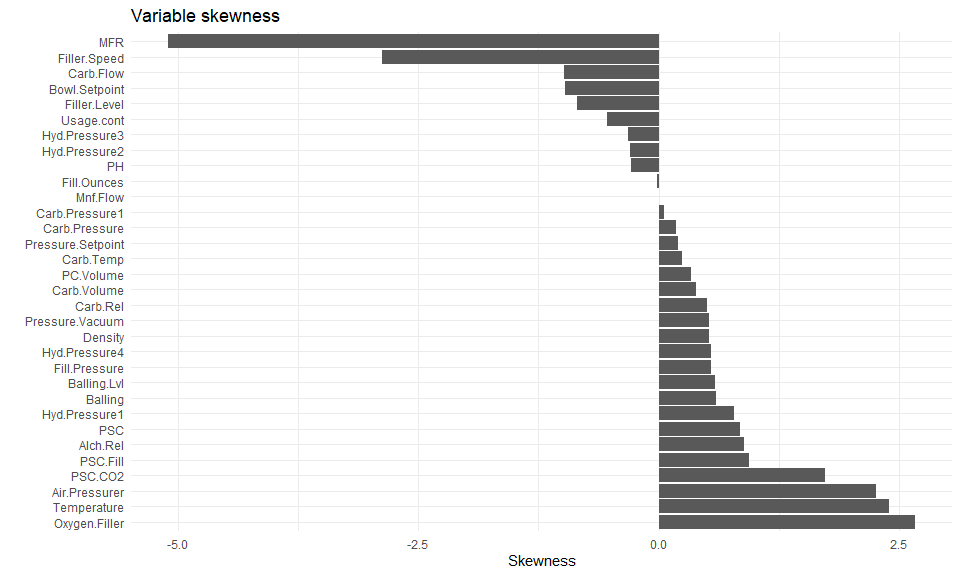
Some variables exhibit obviously skewed distributions (e.g. Filler.Speed, PSC, and Usage.cont), while others appear to be normally distributed (i.e., approximately symmetric), for example PH and PSC.Fill. Others like Bailing and Bailing.Lvl appear to be bimodal.

To gain a better understanding of which variables are skewed and to what degree they are skewed, we can calculate skewness quantitatively using the e1071 package (Meyer, 2022). Skewness with absolute values that are greater than 1 are considered highly skewed, while those between 0.5 and 1 are moderately skewed. Absolute values less than 0.5 can be considered approximately symmetric (Joanes and Gill, 1998).

The sign of the skewness value indicates in which direction the distribution is skewed: Left-skewed distributions will exhibit negative skewness values, while the sign of right-skewed distributions will be positive.

The following plot quantitatively enumerates the skewness of each variable, ordered by how heavily skewed the variable is.

# Look at skewness quantitatively  
# As a general rule of thumb:  
# highly skewed: |skewness| > 1  
# moderately skewed: 0.5 < |skewness| < 1  
# approximately symmetric: 0 < |skewness| < 0.5  
# Negative values indicate left-skewed distributions; positive values indicate right-skewed distributions.  
  
# Create function to calculate skewness (needed because it handles columns with missing values inconsistently)  
calcSkewness <- function(x, type) {  
 return(e1071::skewness(x[!is.na(x)], type=type))  
}  
  
# Calculate skewness on columns  
colskewness <- data.frame(apply(dfm[,1:(ncol(dfm)-1)], 2, calcSkewness, type=1))  
colskewness$Variable <- row.names(colskewness)  
colnames(colskewness) <- c('Skewness', 'Variable')  
  
# Graph skewness values  
colskewness %>%  
 dplyr::select(Variable, Skewness) %>%  
 arrange(desc(abs(Skewness))) %>%  
 ggplot(aes(x=reorder(Variable, desc(Skewness)), y=Skewness)) +  
 geom\_bar(stat='identity') +  
 coord\_flip() +  
 ggtitle('Variable skewness') +  
 xlab('')



As seen in the table, six variables are heavily skewed, with the first two being skewed left and the remaining skewed right:

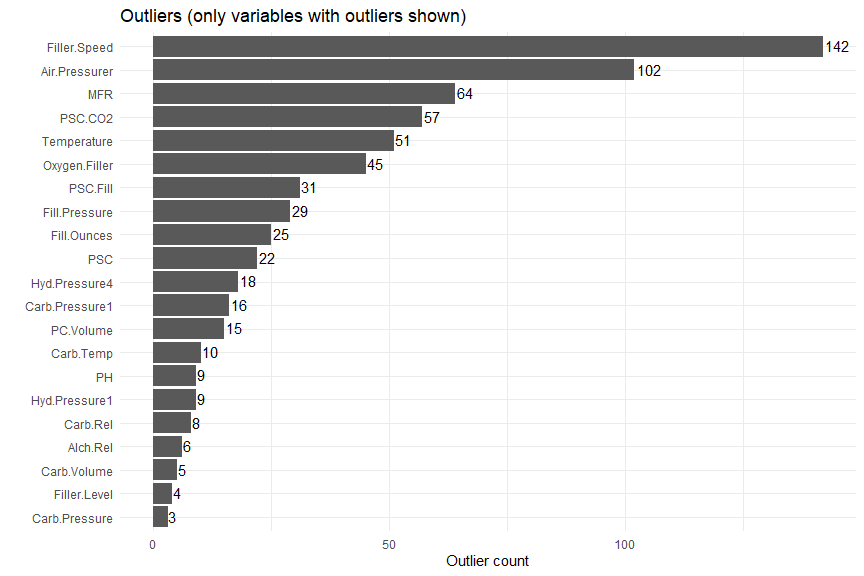
* MFR
* Filler.Speed
* Oxygen.Filler
* Temperature
* Air.Pressurer
* PSC.CO2

This reinforces the idea that some type of transformation will be needed if a model is used that is sensitive to skewness (e.g. linear regression-based models). Alternatively, models that aren’t sensitive to skewness can be employed (e.g. tree-based models).

Outliers

Like collinearity and skewness, the presence of outliers in a data set can also negatively impact some models. Outliers are typically defined as those points which lie beyond a certain number of standard deviations from the mean (typically 2, 2.5, or 3 times the standard deviation). We’ll examine our data set for outlying values that fall under this definition, using a cutoff of 3 standard deviations.

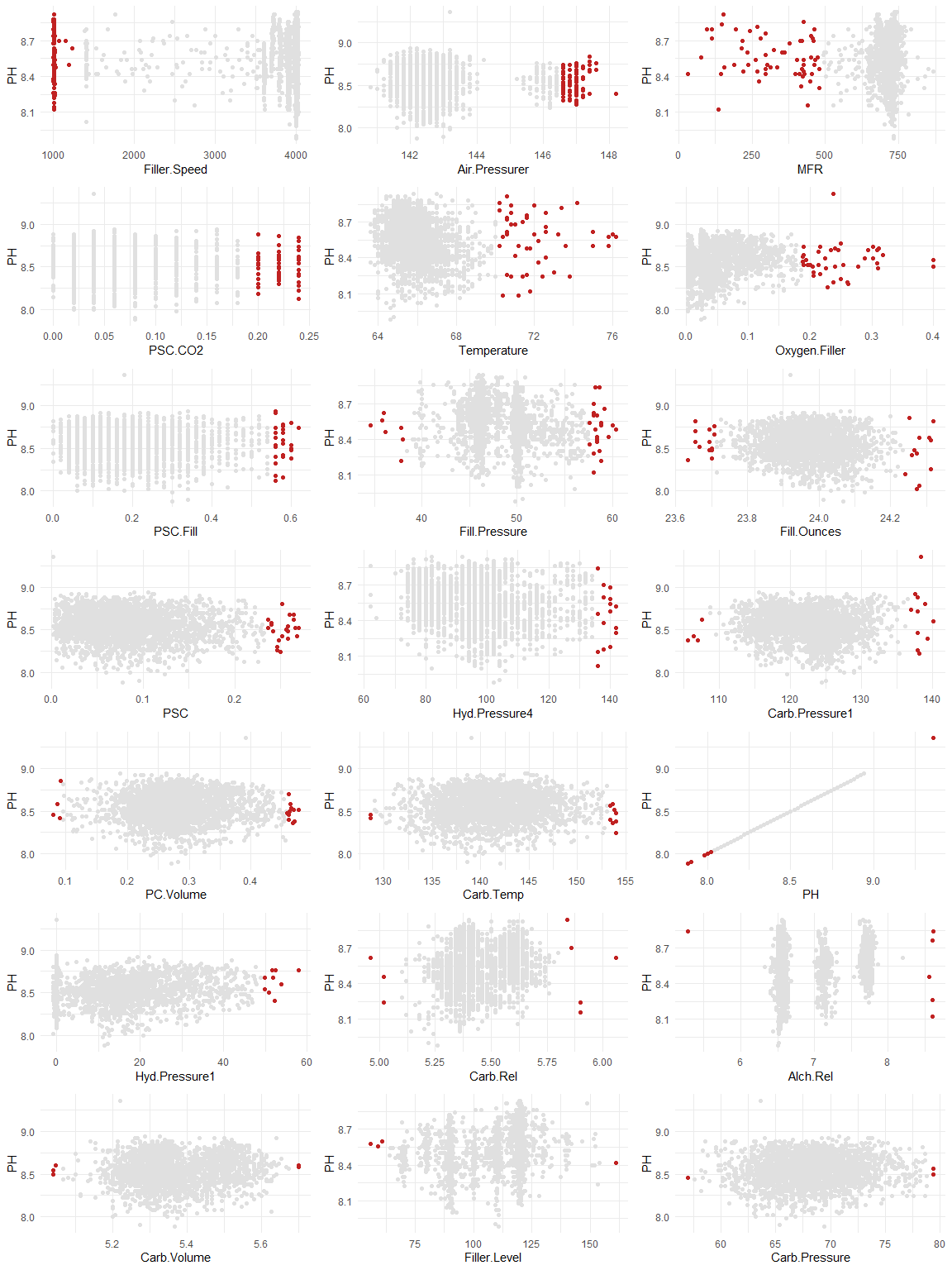
# Create function to count the number of outliers outside of (mult) standard deviations from the mean  
getOutliers <- function(x, mult) {  
 mean\_val <- mean(x[!is.na(x)])  
 loval <- mean\_val - (sd(x[!is.na(x)]) \* mult)  
 hival <- mean\_val + (sd(x[!is.na(x)]) \* mult)  
 #return(vals[vals < loval | vals > hival])  
 return(!is.na(x) & (x < loval | x > hival))  
}  
  
# Set multiplier; outliers are considered as such when they lie outside of (multiplier) standard deviations from the mean  
mult <- 3  
outliers <- apply(dfm[,1:(ncol(dfm)-1)], 2, getOutliers, mult=mult)  
dfout <- data.frame(outliers)  
dfout <- data.frame(colSums(outliers))  
dfout$Variable <- row.names(dfout)  
colnames(dfout) <- c('Outlier.count', 'Variable')  
  
# Filter just those variables with outliers and sort by outlier count  
dfout <- dfout %>%  
 dplyr::select(Variable, Outlier.count) %>%  
 arrange(desc(Outlier.count)) %>%  
 filter(Outlier.count > 0)  
  
# Generate bar graph of outlier counts  
dfout %>%  
 ggplot(aes(x=reorder(Variable, Outlier.count), y=Outlier.count, label=Outlier.count)) +  
 geom\_bar(stat='identity') +  
 coord\_flip() +  
 geom\_text(check\_overlap=T, hjust=-0.1, nudge\_x=0.05) +  
 ggtitle('Outliers (only variables with outliers shown)') +  
 xlab('') + ylab('Outlier count')



The graph above illustrates that Filler.Speed has the most number of outliers (142, or 5.5% of the data). If models that are sensitive to outliers are used (e.g. linear regression-based models), the outliers must be handled in some way, either by removal or by performing an appropriate transformation.

To better understand the outliers, we plotted variables having outliers against the outcome variable, pH. In the plots below, outliers beyond three times the standard deviation appear in red.

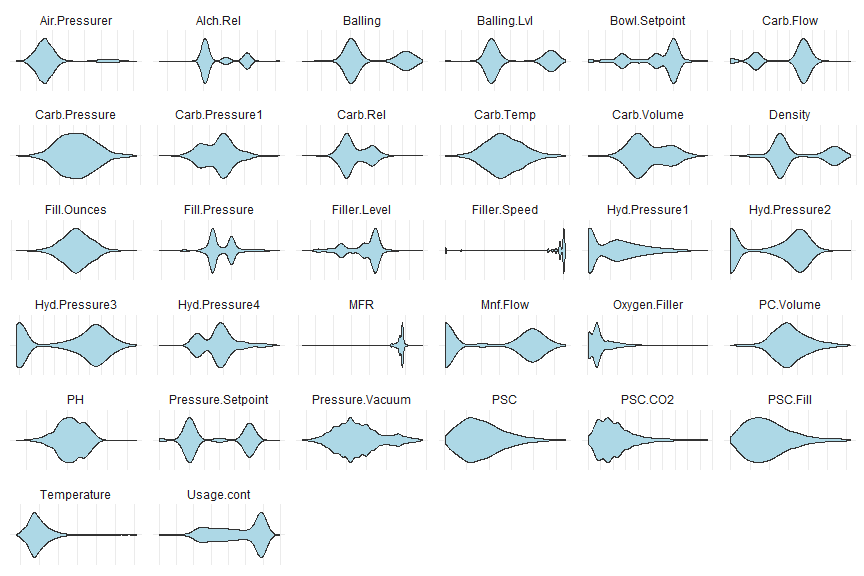
# Create a function to filter and graph the outliers for a specific column  
graphOutliers <- function(x, mult) {  
  
 # Get an array of T/F values indicating whether each observation is an outlier  
 outlier\_array <- getOutliers(dfm[,x], 3)  
   
 # Create data set consisting only of outliers  
 df\_outliers <- dfm[outlier\_array,] %>%  
 dplyr::select(PH, any\_of(x)) %>%  
 mutate(outlier=T)  
   
 # Create data set consisting only of non-outliers  
 df\_non\_outliers <- dfm[!outlier\_array,] %>%  
 dplyr::select(PH, any\_of(x)) %>%  
 mutate(outlier=F)  
   
 # Bind the outlier data set to the non-outlier data set  
 dftmp <- df\_non\_outliers %>%  
 rbind(df\_outliers)  
   
 # Graph the outliers  
 plt <- dftmp %>%  
 filter(!is.na(PH) & !is.na(!!sym(x))) %>%  
 ggplot(aes(x=!!sym(x), y=PH, color=outlier)) +  
 geom\_point() +  
 scale\_colour\_manual(values=c('#e0e0e0', '#c02222')) +  
 theme(legend.position='none')  
 return(plt)  
  
}  
  
# Initialize list of plots  
plts <- list()  
  
# Iterate through all variables with outliers  
for (i in 1:nrow(dfout)) {  
 plts[[i]] <- graphOutliers(dfout[i, 'Variable'], 3)  
}  
do.call("grid.arrange", c(plts, ncol=3))



Based on the plot above, there don’t appear to be any outliers that are structurally unsound (i.e., those with impossible values, such as negative pH values). There are some values that are somewhat suspect (e.g. the six values on the far left and right sides of the Alch.Rel plot), but without a description of the data set or knowing how the data was collected, one can only hazard a guess as to which are valid and which are true outliers. Besides the suspect Alch.Rel data points, the remaining points appear to be valid (though extreme) values when compared to the main body of data points.

As mentioned prior, because of the bimodality, violin plots are preferable than boxplots to observe any outliers,

dfm |>  
 select(-c(Brand.Code)) |>  
 gather(key = "predictor", value = "value") |>  
 ggplot(aes(x = predictor, y = value)) +  
 geom\_violin(fill="lightblue") +  
 facet\_wrap(~ predictor, scales = "free") +  
 coord\_flip() +   
 theme\_minimal() +   
 theme(axis.text.x = element\_blank(),  
 axis.text.y = element\_blank(),  
 axis.title.x = element\_blank(),  
 axis.title.y = element\_blank())



Modeling Approach

At this point we’ll stop to consider several significant factors that will influence our approach to modeling even before we begin cleaning and transforming the data. These factors are as follows:

1. There does not appear to be overtly linear relationships between the outcome variable and the vast majority of predictors, as evidenced by the feature plots. This indicates that a linear regression-based approach may not yield optimal results.
2. As shown in the correlation plot, there is significant collinearity between pairs of variables.
3. About half of the variables exhibit heavy multicollinearity (see the table of VIFs above).
4. Only one variable can be considered NZV, so this won’t be a strong influence on our modeling approach.
5. Likewise, missing values are present but not in significant enough quantities to strongly influence how we’ll choose to model the data.
6. Six variables are heavily skewed (see the histograms and skewness plot above) and would likely need to be transformed for any models that are sensitive to this condition.
7. There are a significant number of outliers, but we can’t be certain if these are valid data points.

Considering these factors, we decided to use an approach to modeling similar to that described in Chapter 10 of our text (Kuhn and Johnson, 2013). Namely, we will create two data sets to be used for two purposes, as follows:

1. **Full data set**: With minimal modification, we’ll use the full data set for those models which are robust to the confounding factors listed above (e.g. collinearity, skewness, and outliers).
2. **Reduced data set**: For models such as those based on linear regression, we’ll prepare another data set that handles the above confounding factors. For example, skewed predictors will be transformed, variables exhibiting multicollinearity will be addressed, and, if feasible, outliers will be excluded.

Using the above approach will give us insights into the best way to model the data using all available models, while generally adhering to the recommended best practices for each model type.

Although we will create two data sets (full and reduced), there is nothing preventing us from running both data sets through all models. We expect linear regression-based models to be unstable and to perform poorly on the full set, but there is no inherent disadvantage to doing so.

Since the outcome variable (pH) is continuous, this is a regression problem, so we’ll naturally choose regression-based models. We’ll start by performing linear regression-based modeling, then move to non-linear regression models, followed by tree-based regression models.

Data Preparation

As stated, we’ll prepare two data sets: the full data set to use with more robust models, and the reduced data set to use with more sensitive models. But while some models are robust to many types of irregularities such as skewed variables and outliers, missing values can pose a problem even to these models and for many of the functions involved in prepare these models. Therefore, we’ll need to handle missing values for both the full set and the reduced set.

Imputation

As previously noted, there is a single observation that has 12 missing predictor values. We’ll remove this observation as it likely would contribute little information to the model. In addition, we’ll remove the predictor (MFR) that contains a large number of missing values relative to the total number of observations; there were 212 missing values for this predictor, or 8.2% of the data.

# Init the full data set  
dfm\_full <- dfm  
  
# Remove observation with 12 missing values  
dfm\_full <- dfm\_full[(rowSums(is.na(dfm\_full)) < 12),]  
  
# Remove the MFR predictor because it has 212 missing values (8.2% of the data)  
dfm\_full <- dfm\_full %>% select(-MFR)

For the remaining missing values, there are multiple imputation methods that can be used to fill in missing values. The MICE method is robust but is computationally intensive. The number of missing values in our data set is relatively small, so we’ll choose a simpler method from the DMwR2 package, the knnImputation() function (Torgo, 2016). knnImputation() uses a well-known statistical algorithm to find the “k” most closely related neighbors to observations having missing values. The value of k is selected by the modeler and for imputation purposes is quite arbitrary. In general, smaller values of k lead to faster but less stable predictions, while larger values of k are more computationally intensive but yield smoother results. We chose a value of 9 for k, which is within the typical range for these applications and provides a good balance of performance versus speed.

# Change blank brand codes to NA so they will be imputed  
dfm\_full <- dfm\_full %>%  
 mutate(Brand.Code=ifelse(Brand.Code=='', NA, as.character(Brand.Code)))  
  
# Factor Brand.Code (needed for imputation)  
dfm\_full$Brand.Code <- factor(dfm\_full$Brand.Code)  
  
# MICE - not used due to computational intensiveness  
#imp <- mice(dfm\_full, maxit=5, m=5, seed=77)  
#complete(imp)  
  
# Use knnImputation to impute values  
dfm\_full <- knnImputation(dfm\_full, k=9)

Dummy variables

One additional step we’ll need to take is to recode the categorical variable Brand.Code into so-called “dummy” variables. This creates a series of variables that take either a 0 or 1 as a value; each variable indicates the presence (value of 1) or absence (value of 0) of that particular Brand.Code. We’ll use the fastDummies package (Kaplan, 2020), which automatically creates dummy variables for factor- or character-type variables.

# Create dummies for factor and character variables  
dfm\_full <- dummy\_cols(dfm\_full, select\_columns='Brand.Code', remove\_first\_dummy=F, remove\_selected\_columns=T)

It is noted that after the dummy columns are created, the data set now has 34 predictors instead of 31.

Full data set

Now that we’ve filled in the missing values, we have a full data set to use with models that are robust to the confounding problems previously discussed. Besides preparing the full and reduced data sets, we’ll further split each set into two additional sets: one set to train the model and another with which to test the results of the model. This is a standard practice that aims to generate a model that is more accurate when it is confronted with new data it hasn’t yet seen. This is accomplished by training the model with only a portion of the data (typically 75 or 80%) while withholding the rest to gauge the results. The training data typically further undergoes “cross-validation” (CV), another common technique that trains the model over multiple iterations, at each iteration withholding a different proportion of the data to tune the model. After the final iteration, the model with the optimal tuning parameters is selected, and that model is then used to evaluate the previously withheld test data.

We have chosen to use 80% of the data for training, withholding the remaining 20% for testing. And we’ll use 10-fold CV, a commonly accepted practice in modeling. We’ll use the createDataPartition() function from the caret package (Kuhn, 2022) which partitions the data evenly based on the outcome variable. This avoids a common problem when modeling data having a strong imbalance in values of the outcome variable.

# Create vector of training rows  
set.seed(77)  
train\_rows <- createDataPartition(dfm\_full$PH, p=0.8, list=F)  
  
# Separate training from test  
dfm\_full\_train <- dfm\_full[train\_rows,]  
dfm\_full\_test <- dfm\_full[-train\_rows,]  
  
# Separate outcome from predictors  
trainx\_full <- dfm\_full\_train[,2:ncol(dfm\_full\_train)]  
trainy\_full <- dfm\_full\_train[,1]  
testx\_full <- dfm\_full\_test[,2:ncol(dfm\_full\_test)]  
testy\_full <- dfm\_full\_test[,1]  
  
# Generate a pretty table for the report  
data.frame(Set=c('Training', 'Test'), Observations=c(nrow(dfm\_full\_train), nrow(dfm\_full\_test))) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Data partiioning (full data set)')

Data partiioning (full data set)

| Set | Observations |
| --- | --- |
| Training | 2,057 |
| Test | 513 |

It is important to note that some models are sensitive to the numbers of observations compared to the number of parameters. Models that must invert linear matrices are mathematically unsolvable if the number of parameters exceeds the number of observations. Cross validation must also be taken into account when evaluating the observation-to-parameter count; for example, ten-fold CV only uses 90% of the observations for training. In our data set, we have 34 predictor variables and 513 observations in the smaller of the two data sets. Using ten-fold CV results in 461 observations still available to use for training, so the observation-to-parameter count isn’t something we have to worry about for this situation.

Reduced data set

Now that we have a full data set will missing values handled, we can generate a second, reduced set to use with models that are sensitive to irregularities. We’ll first make a copy of the full set. Then we’ll need to remove one of the Brand.Code columns, as having all categories of a categorical variable will be problematic for linear-based models.

# First, make a copy the full set to the reduced set  
dfm\_red <- dfm\_full  
  
# Remove the first dummy Brand.Code from the reduced set, since this will be problematic for linear models which use a y-intercept  
dfm\_red <- dfm\_red %>%  
 select(-Brand.Code\_A)

Collinearity

As we noted earlier, there the data set exhibits some collinearity and multicollinearity. Based on VIFs calculated previously, almost half the predictors exhibited strong multicollinearity. Since removing that many predictors would likely cause a detrimental loss of information, a different approach is warranted. We opted to use the approach presented by Kuhl and Johnson (2013) in which predictors are iteratively compared pairwise, removing those with the highest correlation value until a certain threshold is attained. While this procedure only addresses collinearity rather than multicollinearity, it can nonetheless yield significant improvements to model performance and stability.

We’ll use the findCorrelation() function to find candidate variables for removal due to collinearity.

# Get correlation table  
corr2 <- cor(dfm\_red[,2:ncol(dfm\_red)], use='complete')  
corr\_vars <- findCorrelation(corr2, names=T, cutoff=0.9)  
data.frame(Variable=corr\_vars) %>%  
 arrange(Variable) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Highly correlated variables') %>%  
 delete\_part( part = "header")

Highly correlated variables

|  |
| --- |
| Alch.Rel |
| Balling |
| Balling.Lvl |
| Filler.Level |
| Hyd.Pressure3 |

This is a manageable number of variables, so we’ll proceed with removing them from our data set.

# Remove highly correlated variables from the reduced data set  
dfm\_red <- dfm\_red[,-match(corr\_vars, colnames(dfm\_red))]

Near-zero variance

There is only a single predictor with NZV (Hyd.Pressure1). Since this condition can negatively impact linear regression-based models, we’ll remove that from the reduced data set.

# Remove NZV variables from the reduced data set  
dfm\_red <- dfm\_red[,-match('Hyd.Pressure1', colnames(dfm\_red))]

Skewness

We’ll first check for zero or negative values for the variables that were earlier identified as skewed. If there are any such values, the Box-Cox transformation will be infeasible and we’ll have to use a different method.

# Define variables id'ed as skewed from earlier  
skewed\_vars <- c('Filler.Speed', 'Oxygen.Filler', 'Temperature', 'Air.Pressurer', 'PSC.CO2')  
  
# Generate summary to look at min/max values - we're looking for zero or negative values  
summary(dfm\_red[,skewed\_vars])

## Filler.Speed Oxygen.Filler Temperature Air.Pressurer   
## Min. : 998 Min. :0.0024 Min. :63.60 Min. :140.8   
## 1st Qu.:3824 1st Qu.:0.0220 1st Qu.:65.20 1st Qu.:142.2   
## Median :3980 Median :0.0334 Median :65.60 Median :142.6   
## Mean :3649 Mean :0.0469 Mean :65.97 Mean :142.8   
## 3rd Qu.:3996 3rd Qu.:0.0600 3rd Qu.:66.40 3rd Qu.:143.0   
## Max. :4030 Max. :0.4000 Max. :76.20 Max. :148.2   
## PSC.CO2   
## Min. :0.00000   
## 1st Qu.:0.02000   
## Median :0.04000   
## Mean :0.05649   
## 3rd Qu.:0.08000   
## Max. :0.24000

Since PSC.CO2 contains zero values, we’ll use the Yeo-Johnson transformation instead. Because the caret package supports Yeo-Johnson natively, we’ll perform the transformations as part of the preprocessing component during modeling. This has the advantage that predictions made based on our models will be automatically back-transformed rather than requiring manual transformation.

Outliers

Based on the outlier plots from earlier, most of the data points beyond the three-standard-deviation cutoff appear to be valid data points, though located on the extreme outer edges of the main body of data. One exception is Alch.Red, which includes six somewhat suspect points located along the left- and right-hand margins of the plot. The other possible exception is data point exhibiting a PH value of 9.36. It is unclear whether these are truly outliers that should be excluded. But in either case, this variable was already identified as exhibiting high collinearity and has already been removed from the reduced data set.

# Remove sus PH data point  
dfm\_red <- dfm\_red[dfm\_red$PH < 9.36,]  
  
# Alch.Red has already been removed, so no need to do anything there with outliers  
  
# Find cutoff value of Alch.Red (three sd's)  
#loval <- mean(dfm\_red$Alch.Rel) - (3 \* sd(dfm\_red$Alch.Rel))  
#hival <- mean(dfm\_red$Alch.Rel) + (3 \* sd(dfm\_red$Alch.Rel))  
  
# Remove sus Alch.Red data points  
#dfm\_red <- dfm\_red[dfm\_red$Alch.Rel >+ loval & dfm\_red$Alch.Rel <= hival,]

After all modifications to the reduced data set have been made, our final version of the reduced set contains 2569 observations of 28 predictors, reduced from the full set of 2570 observations of 34 predictors. As with the full data set, we’ll split the reduced set into training and test sets and verify that the number of predictors doesn’t exceed the number of observations.

# Create vector of training rows  
set.seed(77)  
train\_rows <- createDataPartition(dfm\_red$PH, p=0.8, list=F)  
  
# Separate training from test  
dfm\_red\_train <- dfm\_red[train\_rows,]  
dfm\_red\_test <- dfm\_red[-train\_rows,]  
  
# Separate outcome from predictors  
trainx\_red <- dfm\_red\_train[,2:ncol(dfm\_red\_train)]  
trainy\_red <- dfm\_red\_train[,1]  
  
# Separate outcome from predictors  
testx\_red <- dfm\_red\_test[,2:ncol(dfm\_red\_test)]  
testy\_red <- dfm\_red\_test[,1]  
  
# Generate a pretty table for the report  
data.frame(Set=c('Training', 'Test'), Observations=c(nrow(dfm\_red\_train), nrow(dfm\_red\_test))) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Data partiioning (reduced data set)')

Data partiioning (reduced data set)

| Set | Observations |
| --- | --- |
| Training | 2,057 |
| Test | 512 |

The table above shows that the test set contains 512 observations, far exceeding the number of predictors (27), so we won’t need to worry about models that perform poorly when there are more predictors than observations.

Modeling

Now that we have a full and reduced set of data–and training and test sets within those two sets–we can begin modeling. As we mentioned, we’ll run both the full and reduced data sets through all models, as there is no inherent disadvantage to do so, while recognizing that some models will be unstable and perform poorly.

We’ll start by performing linear regression-based modeling, then move to non-linear regression models, followed by tree-based regression models.

# Create a data frame to store the results  
dfr <- data.frame(matrix(nrow=0, ncol=8))  
colnames(dfr) <- c('Data.Set', 'Model', 'Model.Class', 'Tuning.Parameters', 'RMSE.Train', 'RMSE.Test', 'MAPE.Test', 'Train.Time')  
  
# specify 10x cross-validation  
ctrl <- trainControl(method='cv', number=10)  
  
# Create function to calculate MAPE  
mape <- function(predicted, actual) {  
 mape <- mean(abs((actual - predicted) / actual)) \* 100  
 return (mape)  
}  
  
# Parallel processing; detect number of cores and start a parallel socket cluster;  
# all subsequent calls to caret::train() will use this cluster  
num\_cores <- detectCores()  
cl <- makePSOCKcluster(num\_cores)  
registerDoParallel(cl)

Linear regression

# Reduced model  
  
# Linear model  
tstart <- Sys.time()  
set.seed(77)  
fitlm1 <- train(x=trainx\_red, y=trainy\_red, method='lm', trControl=ctrl, preProcess=c('center', 'scale', 'YeoJohnson'))  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitlm1  
predlm1 <- predict(fitlm1, testx\_red)  
dfr[1,] <- data.frame(  
 Data.Set='Reduced',   
 Model='Linear model',   
 Model.Class='Linear',  
 Tuning.Parameters='',   
 RMSE.Train=min(fitlm1$results[['RMSE']]),  
 RMSE.Test=postResample(predlm1, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predlm1, testy\_red),  
 Train.Time=round(telapsed, 3)  
)  
  
  
# Stepwise linear model using MASS package/caret  
stepGrid <- data.frame(.nvmax=seq(1, round(ncol(trainx\_red) / 2, 0))) # Max number of parameters to use  
tstart <- Sys.time()  
set.seed(77)  
fitlm2 <- train(x=trainx\_red, y=trainy\_red, method='leapSeq', trControl=ctrl, tuneGrid=stepGrid, preProcess=c('center', 'scale', 'YeoJohnson'))  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitlm2  
predlm2 <- predict(fitlm2, testx\_red)  
dfr[2,] <- data.frame(  
 Data.Set='Reduced',   
 Model='Stepwise linear model',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('nvmax=', fitlm2$bestTune[['nvmax']]),   
 RMSE.Train=min(fitlm2$results[['RMSE']]),  
 RMSE.Test=postResample(predlm2, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predlm2, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Linear model  
tstart <- Sys.time()  
set.seed(77)  
fitlm1\_full <- train(x=trainx\_full, y=trainy\_full, method='lm', trControl=ctrl, preProcess=c('center', 'scale', 'YeoJohnson'))  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitlm1\_full  
predlm1\_full <- predict(fitlm1\_full, testx\_full)

## Warning in predict.lm(modelFit, newdata): prediction from a rank-deficient fit  
## may be misleading

dfr[19,] <- data.frame(  
 Data.Set='Full',   
 Model='Linear model',   
 Model.Class='Linear',  
 Tuning.Parameters='',   
 RMSE.Train=min(fitlm1\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predlm1\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predlm1\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)  
  
# Stepwise linear model using MASS package/caret  
stepGrid <- data.frame(.nvmax=seq(1, round(ncol(trainx\_red) / 2, 0))) # Max number of parameters to use  
tstart <- Sys.time()  
set.seed(77)  
fitlm2\_full <- train(x=trainx\_full, y=trainy\_full, method='leapSeq', trControl=ctrl, tuneGrid=stepGrid, preProcess=c('center', 'scale', 'YeoJohnson'))

## Warning in leaps.setup(x, y, wt = weights, nbest = nbest, nvmax = nvmax, : 1  
## linear dependencies found

telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitlm2\_full  
predlm2\_full <- predict(fitlm2\_full, testx\_full)  
dfr[20,] <- data.frame(  
 Data.Set='Full',   
 Model='Stepwise linear model',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('nvmax=', fitlm2\_full$bestTune[['nvmax']]),   
 RMSE.Train=min(fitlm2\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predlm2\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predlm2\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Robust linear regression

# Reduced model  
  
# Robust linear regression  
tstart <- Sys.time()  
set.seed(77)  
fitrlm <- train(x=trainx\_red, y=trainy\_red, method='rlm', preProcess=c('center', 'scale', 'pca'), trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitrlm  
predrlm <- predict(fitrlm, testx\_red)  
dfr[3,] <- data.frame(  
 Data.Set='Reduced',   
 Model='Robust linear regression',   
 Model.Class='Linear',  
 Tuning.Parameters='',   
 RMSE.Train=min(fitrlm$results[['RMSE']]),  
 RMSE.Test=postResample(predrlm, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predrlm, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Robust linear regression  
tstart <- Sys.time()  
set.seed(77)  
fitrlm\_full <- train(x=trainx\_full, y=trainy\_full, method='rlm', preProcess=c('center', 'scale', 'pca'), trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitrlm\_full  
predrlm\_full <- predict(fitrlm\_full, testx\_full)  
dfr[21,] <- data.frame(  
 Data.Set='Full',   
 Model='Robust linear regression',   
 Model.Class='Linear',  
 Tuning.Parameters='',   
 RMSE.Train=min(fitrlm\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predrlm\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predrlm\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Partial least squares

# Reduced model  
  
# PLS  
tstart <- Sys.time()  
set.seed(77)  
fitpls <- train(x=trainx\_red, y=trainy\_red, method='pls', preProcess=c('center', 'scale'), trControl=ctrl, tuneLength=nrow(trainx\_red) / 2)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitpls  
predpls <- predict(fitpls, testx\_red)  
dfr[4,] <- data.frame(  
 Data.Set='Reduced',   
 Model='Partial least squares',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('ncomp=', fitpls$bestTune),   
 RMSE.Train=min(fitpls$results[['RMSE']]),  
 RMSE.Test=postResample(predpls, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predpls, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# PLS  
tstart <- Sys.time()  
set.seed(77)  
fitpls\_full <- train(x=trainx\_full, y=trainy\_full, method='pls', preProcess=c('center', 'scale'), trControl=ctrl, tuneLength=nrow(trainx\_full) / 2)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitpls\_full  
predpls\_full <- predict(fitpls\_full, testx\_full)  
dfr[22,] <- data.frame(  
 Data.Set='Full',   
 Model='Partial least squares',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('ncomp=', fitpls\_full$bestTune),   
 RMSE.Train=min(fitpls\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predpls\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predpls\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Ridge regression

# Reduced model  
  
# Ridge regression  
tstart <- Sys.time()  
set.seed(77)  
ridgeGrid <- data.frame(.lambda=seq(0, 0.1, length=15))  
fitridge <- train(x=trainx\_red, y=trainy\_red, method='ridge', preProcess=c('center', 'scale'), trControl=ctrl, tuneGrid=ridgeGrid)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitridge  
predridge <- predict(fitridge, testx\_red)  
dfr[5,] <- data.frame(  
 Data.Set='Reduced',   
 Model='Ridge regression',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('labmda=', round(fitridge$bestTune[['lambda']], 4)),   
 RMSE.Train=min(fitridge$results[['RMSE']]),  
 RMSE.Test=postResample(predridge, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predridge, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Ridge regression  
tstart <- Sys.time()  
set.seed(77)  
ridgeGrid <- data.frame(.lambda=seq(0, 0.1, length=15))  
fitridge\_full <- train(x=trainx\_full, y=trainy\_full, method='ridge', preProcess=c('center', 'scale'), trControl=ctrl, tuneGrid=ridgeGrid)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitridge\_full  
predridge\_full <- predict(fitridge\_full, testx\_full)  
dfr[23,] <- data.frame(  
 Data.Set='Full',   
 Model='Ridge regression',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('labmda=', round(fitridge\_full$bestTune[['lambda']], 4)),   
 RMSE.Train=min(fitridge\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predridge\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predridge\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Lasso regression

# Reduced model  
  
# Lasso regression  
enetGrid <- expand.grid(.lambda=c(0, 0.01, 0.1), .fraction=seq(0.05, 1, length=20))  
tstart <- Sys.time()  
set.seed(77)  
fitlasso <- train(x=trainx\_red, y=trainy\_red, method='enet', preProcess=c('center', 'scale'), trControl=ctrl, tuneGrid=enetGrid)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitlasso  
predlasso <- predict(fitlasso, testx\_red)  
dfr[6,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Lasso (enlastic net)',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('lambda=', fitlasso$bestTune[['lambda']], ', fraction=', fitlasso$bestTune[['fraction']]),   
 RMSE.Train=min(fitlasso$results[['RMSE']]),  
 RMSE.Test=postResample(predlasso, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predlasso, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Lasso regression  
enetGrid <- expand.grid(.lambda=c(0, 0.01, 0.1), .fraction=seq(0.05, 1, length=20))  
tstart <- Sys.time()  
set.seed(77)  
fitlasso\_full <- train(x=trainx\_full, y=trainy\_full, method='enet', preProcess=c('center', 'scale'), trControl=ctrl, tuneGrid=enetGrid)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitlasso\_full  
predlasso\_full <- predict(fitlasso\_full, testx\_full)  
dfr[24,] <- data.frame(  
 Data.Set='Full',  
 Model='Lasso (enlastic net)',   
 Model.Class='Linear',  
 Tuning.Parameters=paste0('lambda=', fitlasso\_full$bestTune[['lambda']], ', fraction=', fitlasso\_full$bestTune[['fraction']]),   
 RMSE.Train=min(fitlasso\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predlasso\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predlasso\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

K nearest neighbors

# Reduced model  
  
# KNN model  
tstart <- Sys.time()  
set.seed(77)  
fitknn <- train(x=trainx\_red, y=trainy\_red, method='knn', preProc=c("center", "scale"), tuneLength=10)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitknn  
predknn <- predict(fitknn, testx\_red)  
dfr[7,] <- data.frame(  
 Data.Set='Reduced',  
 Model='knn',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('k=', fitknn$bestTune[['k']]),   
 RMSE.Train=min(fitknn$results[['RMSE']]),  
 RMSE.Test=postResample(predknn, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predknn, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# KNN model  
tstart <- Sys.time()  
set.seed(77)  
fitknn\_full <- train(x=trainx\_full, y=trainy\_full, method='knn', preProc=c("center", "scale"), tuneLength=10)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitknn\_full  
predknn\_full <- predict(fitknn\_full, testx\_full)  
dfr[25,] <- data.frame(  
 Data.Set='Full',  
 Model='knn',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('k=', fitknn\_full$bestTune[['k']]),   
 RMSE.Train=min(fitknn\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predknn\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predknn\_full, testy\_full),  
 Train.Time=round(telapsed, 3),  
 Train.Time=round(telapsed, 3)  
)

## Warning in `[<-.data.frame`(`\*tmp\*`, 25, , value = structure(list(Data.Set =  
## "Full", : provided 9 variables to replace 8 variables

Multivariate adaptive regression splines

# Reduced model  
  
# MARS model  
marsGrid <- expand.grid(.degree=1:2, .nprune=2:10) # set tuning parameters  
tstart <- Sys.time()  
set.seed(77)  
fitmars <- train(x=trainx\_red, y=trainy\_red, method='earth', tuneGrid=marsGrid, trControl=ctrl)

## Loading required package: earth

## Warning: package 'earth' was built under R version 4.2.3

## Loading required package: Formula

## Loading required package: plotmo

## Warning: package 'plotmo' was built under R version 4.2.3

## Loading required package: plotrix

## Loading required package: TeachingDemos

## Warning: package 'TeachingDemos' was built under R version 4.2.3

##   
## Attaching package: 'TeachingDemos'

## The following object is masked \_by\_ '.GlobalEnv':  
##   
## outliers

telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitmars  
predmars <- predict(fitmars, testx\_red)  
dfr[8,] <- data.frame(  
 Data.Set='Reduced',  
 Model='MARS',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('degree=', fitmars$bestTune[['degree']], ', nprune=', fitmars$bestTune[['nprune']]),   
 RMSE.Train=min(fitmars$results[['RMSE']]),  
 RMSE.Test=postResample(predmars, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predmars, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# MARS model  
marsGrid <- expand.grid(.degree=1:2, .nprune=2:10) # set tuning parameters  
tstart <- Sys.time()  
set.seed(77)  
fitmars\_full <- train(x=trainx\_full, y=trainy\_full, method='earth', tuneGrid=marsGrid, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitmars\_full  
predmars\_full <- predict(fitmars\_full, testx\_full)  
dfr[26,] <- data.frame(  
 Data.Set='Full',  
 Model='MARS',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('degree=', fitmars\_full$bestTune[['degree']], ', nprune=', fitmars\_full$bestTune[['nprune']]),   
 RMSE.Train=min(fitmars\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predmars\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predmars\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Support vector machine

# Reduced model  
  
# SVM model  
tstart <- Sys.time()  
set.seed(77)  
fitsvm <- train(x=trainx\_red, y=trainy\_red, method='svmRadial', preProc=c('center', 'scale'), tuneLength=14, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitsvm  
predsvm <- predict(fitsvm, testx\_red)  
dfr[9,] <- data.frame(  
 Data.Set='Reduced',  
 Model='SVM',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('C=', fitsvm$bestTune[['C']], ', sigma=', round(fitsvm$bestTune[['sigma']], 3)),   
 RMSE.Train=min(fitsvm$results[['RMSE']]),  
 RMSE.Test=postResample(predsvm, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predsvm, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# SVM model  
tstart <- Sys.time()  
set.seed(77)  
fitsvm\_full <- train(x=trainx\_full, y=trainy\_full, method='svmRadial', preProc=c('center', 'scale'), tuneLength=14, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitsvm\_full  
predsvm\_full <- predict(fitsvm\_full, testx\_full)  
dfr[27,] <- data.frame(  
 Data.Set='Full',  
 Model='SVM',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('C=', fitsvm\_full$bestTune[['C']], ', sigma=', round(fitsvm\_full$bestTune[['sigma']], 3)),   
 RMSE.Train=min(fitsvm\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predsvm\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predsvm\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Neural net

# Reduced model  
  
# nnet model using model averaging  
nnetGrid <- expand.grid(.decay=c(0, 0.01, 0.1), .size=c(1:10), .bag=F) # set tuning parameters  
tstart <- Sys.time()  
set.seed(77)  
fitnnet <- train(x=trainx\_red, y=trainy\_red, method='avNNet', preProc=c('center', 'scale'), tunGrid=nnetGrid, trControl=ctrl,  
 linout=T, trace=F, MaxNWts=10 \* (ncol(trainx\_red) + 1) + 10 + 1, maxit=500)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitnnet  
prednnet <- predict(fitnnet, testx\_red)  
dfr[10,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Neural net',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('decay=', fitnnet$bestTune[['decay']], ', size=', fitnnet$bestTune[['size']], ', bag=False'),   
 RMSE.Train=min(fitnnet$results[['RMSE']]),  
 RMSE.Test=postResample(prednnet, testy\_red)[['RMSE']],  
 MAPE.Test=mape(prednnet, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# nnet model using model averaging  
nnetGrid <- expand.grid(.decay=c(0, 0.01, 0.1), .size=c(1:10), .bag=F) # set tuning parameters  
tstart <- Sys.time()  
set.seed(77)  
fitnnet\_full <- train(x=trainx\_full, y=trainy\_full, method='avNNet', preProc=c('center', 'scale'), tunGrid=nnetGrid, trControl=ctrl,  
 linout=T, trace=F, MaxNWts=10 \* (ncol(trainx\_red) + 1) + 10 + 1, maxit=500)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitnnet\_full  
prednnet\_full <- predict(fitnnet\_full, testx\_full)  
dfr[28,] <- data.frame(  
 Data.Set='Full',  
 Model='Neural net',   
 Model.Class='Nonlinear',  
 Tuning.Parameters=paste0('decay=', fitnnet\_full$bestTune[['decay']], ', size=', fitnnet\_full$bestTune[['size']], ', bag=False'),   
 RMSE.Train=min(fitnnet\_full$results[['RMSE']]),  
 RMSE.Test=postResample(prednnet\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(prednnet\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Basic CART (tuned with complexity parameter)

# Reduced model  
  
# Basic regression tree  
tstart <- Sys.time()  
set.seed(77)  
fitcart1 <- train(trainx\_red, trainy\_red, method='rpart', tuneLength=10, trControl=ctrl)

## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo,  
## : There were missing values in resampled performance measures.

telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcart1  
predcart1 <- predict(fitcart1, testx\_red)  
dfr[11,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Basic CART (tuned w/complexity parameter)',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('cp=', round(fitcart1$bestTune[['cp']], 4)),   
 Train.RMSE=min(fitcart1$results[['RMSE']]),  
 RMSE.Test=postResample(predcart1, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predcart1, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Basic regression tree  
tstart <- Sys.time()  
set.seed(77)  
fitcart1\_full <- train(trainx\_full, trainy\_full, method='rpart', tuneLength=10, trControl=ctrl)

## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo,  
## : There were missing values in resampled performance measures.

telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcart1\_full  
predcart1\_full <- predict(fitcart1\_full, testx\_full)  
dfr[29,] <- data.frame(  
 Data.Set='Full',  
 Model='Basic CART (tuned w/complexity parameter)',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('cp=', round(fitcart1\_full$bestTune[['cp']], 4)),   
 Train.RMSE=min(fitcart1\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predcart1\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predcart1\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Basic CART (tuned with node depth)

# Reduced model  
  
# Basic CART model - tuned using node depth  
tstart <- Sys.time()  
set.seed(77)  
fitcart2 <- train(trainx\_red, trainy\_red, method='rpart2', tuneLength=10, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcart2  
predcart2 <- predict(fitcart2, testx\_red)  
dfr[12,] = data.frame(  
 Data.Set='Reduced',  
 Model='Basic CART (tuned w/node depth)',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('maxdepth=', fitcart2$bestTune[['maxdepth']]),   
 Train.RMSE=min(fitcart2$results[['RMSE']]),  
 RMSE.Test=postResample(predcart2, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predcart2, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Basic CART model - tuned using node depth  
tstart <- Sys.time()  
set.seed(77)  
fitcart2\_full <- train(trainx\_full, trainy\_full, method='rpart2', tuneLength=10, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcart2\_full  
predcart2\_full <- predict(fitcart2\_full, testx\_full)  
dfr[30,] = data.frame(  
 Data.Set='Full',  
 Model='Basic CART (tuned w/node depth)',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('maxdepth=', fitcart2\_full$bestTune[['maxdepth']]),   
 Train.RMSE=min(fitcart2\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predcart2\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predcart2\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Regression model tree

# Reduced model  
  
# Model tree  
mtreeGrid1 <- expand.grid(.pruned=c('Yes', 'No'), .smoothed=c('Yes', 'No'), .rules=c('Yes', 'No'))  
tstart <- Sys.time()  
set.seed(77)  
fitmtree1 <- train(trainx\_red, trainy\_red, method='M5', trControl=ctrl, control=Weka\_control(M=10), tuneGrid=mtreeGrid1)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitmtree1  
predmtree1 <- predict(fitmtree1, testx\_red)  
dfr[13,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Regression model tree',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('pruned=', fitmtree1$bestTune[['pruned']],  
 ', smoothed=', fitmtree1$bestTune[['smoothed']],  
 ', rules=', fitmtree1$bestTune[['rules']]),   
 RMSE.Train=min(fitmtree1$results[['RMSE']]),  
 RMSE.Test=postResample(predmtree1, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predmtree1, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Model tree  
mtreeGrid1 <- expand.grid(.pruned=c('Yes', 'No'), .smoothed=c('Yes', 'No'), .rules=c('Yes', 'No'))  
tstart <- Sys.time()  
set.seed(77)  
fitmtree1\_full <- train(trainx\_full, trainy\_full, method='M5', trControl=ctrl, control=Weka\_control(M=10), tuneGrid=mtreeGrid1)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitmtree1\_full  
predmtree1\_full <- predict(fitmtree1\_full, testx\_full)  
dfr[31,] <- data.frame(  
 Data.Set='Full',  
 Model='Regression model tree',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('pruned=', fitmtree1\_full$bestTune[['pruned']],  
 ', smoothed=', fitmtree1\_full$bestTune[['smoothed']],  
 ', rules=', fitmtree1\_full$bestTune[['rules']]),   
 RMSE.Train=min(fitmtree1\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predmtree1\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predmtree1\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Regression model tree (rule-based)

# Reduced model  
  
# Model tree (rule-based)  
mtreeGrid2 <- expand.grid(.pruned=c('Yes', 'No'), .smoothed=c('Yes', 'No'))  
tstart <- Sys.time()  
set.seed(77)  
fitmtree2 <- train(trainx\_red, trainy\_red, method='M5Rules', trControl=ctrl, control=Weka\_control(M=10), tuneGrid=mtreeGrid2)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitmtree2  
predmtree2 <- predict(fitmtree2, testx\_red)  
dfr[14,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Regression model tree (rule-based)',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('pruned=', fitmtree2$bestTune[['pruned']],  
 ', smoothed=', fitmtree2$bestTune[['smoothed']]),   
 RMSE.Train=min(fitmtree2$results[['RMSE']]),  
 RMSE.Test=postResample(predmtree2, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predmtree2, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Model tree (rule-based)  
mtreeGrid2 <- expand.grid(.pruned=c('Yes', 'No'), .smoothed=c('Yes', 'No'))  
tstart <- Sys.time()  
set.seed(77)  
fitmtree2\_full <- train(trainx\_full, trainy\_full, method='M5Rules', trControl=ctrl, control=Weka\_control(M=10), tuneGrid=mtreeGrid2)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitmtree2\_full  
predmtree2\_full <- predict(fitmtree2\_full, testx\_full)  
dfr[32,] <- data.frame(  
 Data.Set='Full',  
 Model='Regression model tree (rule-based)',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('pruned=', fitmtree2\_full$bestTune[['pruned']],  
 ', smoothed=', fitmtree2\_full$bestTune[['smoothed']]),   
 RMSE.Train=min(fitmtree2\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predmtree2\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predmtree2\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Bagged tree

# Reduced model  
  
# Bagged tree  
tstart <- Sys.time()  
set.seed(77)  
fitbag <- train(trainx\_red, trainy\_red, method='treebag', trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitbag  
predbag <- predict(fitbag, testx\_red)  
dfr[15,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Bagged tree',   
 Model.Class='Tree',  
 Tuning.Parameters='',   
 RMSE.Train=min(fitbag$results[['RMSE']]),  
 RMSE.Test=postResample(predbag, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predbag, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Bagged tree  
tstart <- Sys.time()  
set.seed(77)  
fitbag\_full <- train(trainx\_full, trainy\_full, method='treebag', trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitbag\_full  
predbag\_full <- predict(fitbag\_full, testx\_full)  
dfr[33,] <- data.frame(  
 Data.Set='Full',  
 Model='Bagged tree',   
 Model.Class='Tree',  
 Tuning.Parameters='',   
 RMSE.Train=min(fitbag\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predbag\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predbag\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Random Forest

# Reduced model  
  
# Random Forest  
tstart <- Sys.time()  
set.seed(77)  
fitrf <- train(trainx\_red, trainy\_red, method='rf', tuneLength=10, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitrf  
predrf <- predict(fitrf, testx\_red)  
dfr[16,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Random Forest',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('mtry=', fitrf$bestTune[['mtry']]),   
 RMSE.Train=min(fitrf$results[['RMSE']]),  
 RMSE.Test=postResample(predrf, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predrf, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Random Forest  
tstart <- Sys.time()  
set.seed(77)  
fitrf\_full <- train(trainx\_full, trainy\_full, method='rf', tuneLength=10, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitrf\_full  
predrf\_full <- predict(fitrf\_full, testx\_full)  
dfr[34,] <- data.frame(  
 Data.Set='Full',  
 Model='Random Forest',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('mtry=', fitrf\_full$bestTune[['mtry']]),   
 RMSE.Train=min(fitrf\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predrf\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predrf\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Stochastic gradient boosted tree

# Had to set message=FALSE to prevent thousands of trace messages  
  
# Reduced model  
  
# Stochastic gradient boosting  
gbmGrid <- expand.grid(.interaction.depth=seq(1, 7, by=2),  
 .n.trees=seq(100, 1000, by=50),  
 .shrinkage=c(0.01, 0.1),  
 .n.minobsinnode=10)  
tstart <- Sys.time()  
set.seed(77)  
fitgbm <- train(trainx\_red, trainy\_red, method='gbm', tuneGrid=gbmGrid, trControl=ctrl)

telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))

# Reduced model, cont'd  
# fitgbm  
predgbm <- predict(fitgbm, testx\_red)  
dfr[17,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Stochastic gradient boosted tree',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('interaction.depth=', fitgbm$bestTune[['interaction.depth']],   
 ', n.trees=', fitgbm$bestTune[['n.trees']],   
 ', shrinkage=', fitgbm$bestTune[['shrinkage']],  
 ', n.minobsinnode=10'),   
 Train.RMSE=min(fitgbm$results[['RMSE']]),  
 RMSE.Test=postResample(predgbm, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predgbm, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Had to set message=FALSE to prevent thousands of trace messages  
  
# Full model  
  
# Stochastic gradient boosting  
gbmGrid <- expand.grid(.interaction.depth=seq(1, 7, by=2),  
 .n.trees=seq(100, 1000, by=50),  
 .shrinkage=c(0.01, 0.1),  
 .n.minobsinnode=10)  
tstart <- Sys.time()  
set.seed(77)  
fitgbm\_full <- train(trainx\_full, trainy\_full, method='gbm', tuneGrid=gbmGrid, trControl=ctrl)

telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))

# Full model, cont'd  
# fitgbm\_full  
predgbm\_full <- predict(fitgbm\_full, testx\_full)  
dfr[35,] <- data.frame(  
 Data.Set='Full',  
 Model='Stochastic gradient boosted tree',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('interaction.depth=', fitgbm\_full$bestTune[['interaction.depth']],   
 ', n.trees=', fitgbm\_full$bestTune[['n.trees']],   
 ', shrinkage=', fitgbm\_full$bestTune[['shrinkage']],  
 ', n.minobsinnode=10'),   
 Train.RMSE=min(fitgbm\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predgbm\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predgbm\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

Cubist

# Reduced model  
  
# Cubist  
cubGrid <- expand.grid(.committees=c(seq(1, 10), seq(20, 100, by=10)), .neighbors=c(0, 1, 5, 9))  
tstart <- Sys.time()  
set.seed(77)  
fitcub <- train(trainx\_red, trainy\_red, method='cubist', tuneGrid=cubGrid, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcub  
predcub <- predict(fitcub, testx\_red)  
dfr[18,] <- data.frame(  
 Data.Set='Reduced',  
 Model='Cubist',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('committees=', fitcub$bestTune[['committees']],   
 ', neighbors=', fitcub$bestTune[['neighbors']]),   
 Train.RMSE=min(fitcub$results[['RMSE']]),  
 RMSE.Test=postResample(predcub, testy\_red)[['RMSE']],  
 MAPE.Test=mape(predcub, testy\_red),  
 Train.Time=round(telapsed, 3)  
)

# Full model  
  
# Cubist  
cubGrid <- expand.grid(.committees=c(seq(1, 10), seq(20, 100, by=10)), .neighbors=c(0, 1, 5, 9))  
tstart <- Sys.time()  
set.seed(77)  
fitcub\_full <- train(trainx\_full, trainy\_full, method='cubist', tuneGrid=cubGrid, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcub\_full  
predcub\_full <- predict(fitcub\_full, testx\_full)  
dfr[36,] <- data.frame(  
 Data.Set='Full',  
 Model='Cubist',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('committees=', fitcub\_full$bestTune[['committees']],   
 ', neighbors=', fitcub\_full$bestTune[['neighbors']]),   
 Train.RMSE=min(fitcub\_full$results[['RMSE']]),  
 RMSE.Test=postResample(predcub\_full, testy\_full)[['RMSE']],  
 MAPE.Test=mape(predcub\_full, testy\_full),  
 Train.Time=round(telapsed, 3)  
)

# Stop the parallel processing cluster  
stopCluster(cl)

Modeling Results

Model performance can be compared using a variety of metrics. One metric commonly used is root mean square error (RMSE), which measures the difference between actual outcome values (pH in this case) and those predicted by the model. RMSE units are in the same units as the outcome variable, pH, making it easy to compare to the original variable.

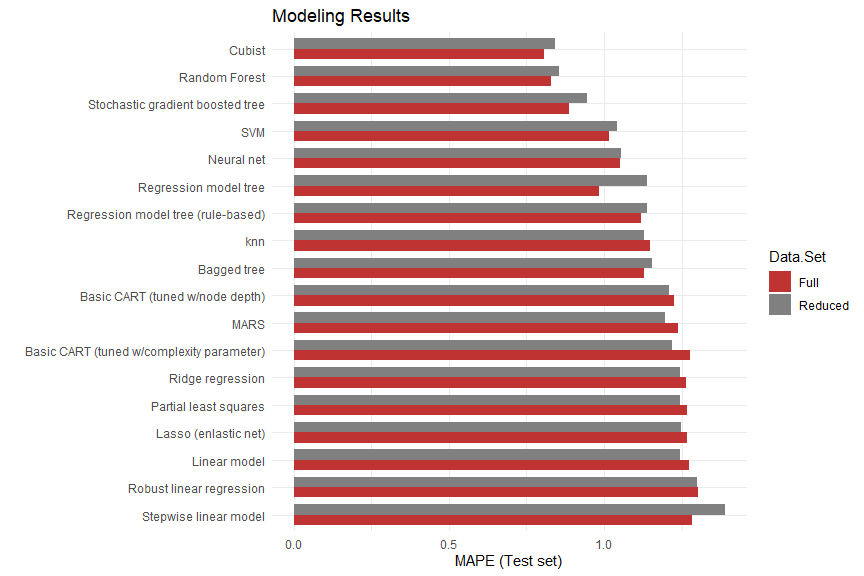
Another useful metric is the mean absolute percentage error (MAPE), which measures the percentage difference between predicted and actual values of the outcome. Below is a summary of RMSE and MAPE values, along with the tuning parameters of the best-performing models:

# Results table  
dfr %>%  
 arrange(MAPE.Test) %>%  
 select(Model, Data.Set, Tuning.Parameters, Train.Time, RMSE.Train, RMSE.Test, MAPE.Test) %>%  
 flextable() %>%  
 width(width = 1) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Modeling Results')

Modeling Results

| Model | Data.Set | Tuning.Parameters | Train.Time | RMSE.Train | RMSE.Test | MAPE.Test |
| --- | --- | --- | --- | --- | --- | --- |
| Cubist | Full | committees=100, neighbors=5 | 387.711 | 0.09489731 | 0.09467403 | 0.8069488 |
| Random Forest | Full | mtry=26 | 397.041 | 0.09573711 | 0.09518857 | 0.8278027 |
| Cubist | Reduced | committees=30, neighbors=9 | 278.724 | 0.09590937 | 0.09602074 | 0.8406549 |
| Random Forest | Reduced | mtry=18 | 355.858 | 0.09756142 | 0.10008600 | 0.8564273 |
| Stochastic gradient boosted tree | Full | interaction.depth=7, n.trees=950, shrinkage=0.1, n.minobsinnode=10 | 86.171 | 0.10471437 | 0.10147753 | 0.8862444 |
| Stochastic gradient boosted tree | Reduced | interaction.depth=5, n.trees=650, shrinkage=0.1, n.minobsinnode=10 | 60.957 | 0.10722935 | 0.10653321 | 0.9463547 |
| Regression model tree | Full | pruned=Yes, smoothed=Yes, rules=No | 149.254 | 0.12150091 | 0.11406403 | 0.9841826 |
| SVM | Full | C=16, sigma=0.019 | 89.682 | 0.11328172 | 0.11362331 | 1.0158661 |
| SVM | Reduced | C=8, sigma=0.024 | 100.252 | 0.11517041 | 0.11670584 | 1.0427894 |
| Neural net | Full | decay=0.1, size=5, bag=False | 142.491 | 0.11565164 | 0.11789163 | 1.0522013 |
| Neural net | Reduced | decay=0.1, size=5, bag=False | 120.425 | 0.11713042 | 0.11587517 | 1.0535517 |
| Regression model tree (rule-based) | Full | pruned=Yes, smoothed=No | 158.470 | 0.12672150 | 0.13331952 | 1.1181864 |
| knn | Reduced | k=11 | 11.323 | 0.12570259 | 0.12448787 | 1.1280221 |
| Bagged tree | Full |  | 2.728 | 0.11960417 | 0.12225904 | 1.1282624 |
| Regression model tree | Reduced | pruned=Yes, smoothed=No, rules=Yes | 157.684 | 0.12342576 | 0.13725888 | 1.1370836 |
| Regression model tree (rule-based) | Reduced | pruned=Yes, smoothed=No | 120.657 | 0.12342576 | 0.13725888 | 1.1370836 |
| knn | Full | k=13 | 13.973 | 0.12546626 | 0.12635171 | 1.1488225 |
| Bagged tree | Reduced |  | 2.360 | 0.12080801 | 0.12566115 | 1.1556535 |
| MARS | Reduced | degree=2, nprune=10 | 19.501 | 0.13075924 | 0.13523319 | 1.1976969 |
| Basic CART (tuned w/node depth) | Reduced | maxdepth=10 | 1.140 | 0.12801291 | 0.13305594 | 1.2089476 |
| Basic CART (tuned w/complexity parameter) | Reduced | cp=0.0136 | 1.229 | 0.12948167 | 0.13469959 | 1.2183462 |
| Basic CART (tuned w/node depth) | Full | maxdepth=11 | 1.170 | 0.12911119 | 0.13328713 | 1.2259351 |
| MARS | Full | degree=1, nprune=10 | 22.742 | 0.13164275 | 0.13438371 | 1.2379820 |
| Partial least squares | Reduced | ncomp=11 | 1.157 | 0.13440134 | 0.13419731 | 1.2442610 |
| Linear model | Reduced |  | 18.542 | 0.13418504 | 0.13426328 | 1.2442876 |
| Ridge regression | Reduced | labmda=0.0214 | 5.882 | 0.13440744 | 0.13416671 | 1.2445097 |
| Lasso (enlastic net) | Reduced | lambda=0, fraction=0.9 | 2.373 | 0.13435146 | 0.13448619 | 1.2471906 |
| Ridge regression | Full | labmda=0.0071 | 7.448 | 0.13214741 | 0.13687962 | 1.2653748 |
| Lasso (enlastic net) | Full | lambda=0, fraction=0.9 | 2.170 | 0.13204044 | 0.13763948 | 1.2671424 |
| Partial least squares | Full | ncomp=29 | 1.509 | 0.13214800 | 0.13763492 | 1.2671462 |
| Linear model | Full |  | 2.031 | 0.13305920 | 0.13768953 | 1.2735829 |
| Basic CART (tuned w/complexity parameter) | Full | cp=0.0127 | 1.222 | 0.13031164 | 0.13833394 | 1.2770764 |
| Stepwise linear model | Full | nvmax=12 | 2.025 | 0.13465862 | 0.13991498 | 1.2830543 |
| Robust linear regression | Reduced |  | 2.570 | 0.13929776 | 0.13920669 | 1.2980118 |
| Robust linear regression | Full |  | 2.949 | 0.13748209 | 0.13930736 | 1.3025277 |
| Stepwise linear model | Reduced | nvmax=14 | 2.009 | 0.13537079 | 0.15091506 | 1.3898956 |

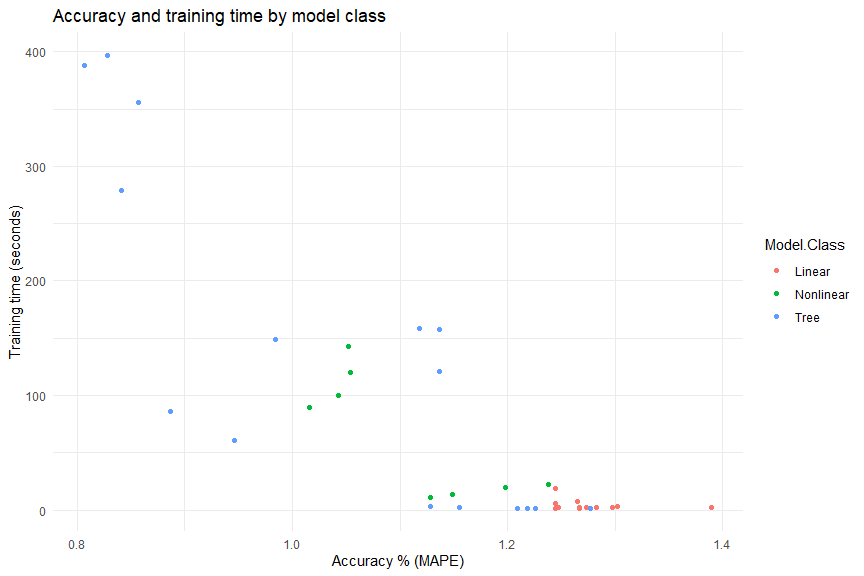
dfr %>%  
 ggplot(aes(x=reorder(Model, desc(MAPE.Test)), y=MAPE.Test, fill=Data.Set)) +  
 geom\_bar(stat='identity', position='dodge', width=0.75) +  
 coord\_flip() +  
 xlab('') +  
 ylab('MAPE (Test set)') +  
 scale\_fill\_manual(values=c('#c03333', '#808080')) +  
 ggtitle('Modeling Results')



Our preliminary runs show that tree-based models were the best performers by a significant margin. Nonlinear regression-based models outperformed those that are linear based, but were still outclassed by the tree-based models. The Cubist and random forest models were the top performers, followed by the stochastic gradient boosted tree and regression model tree. It is not surprising that the tree-based models performed better when fed the full data set, as there was a richer feature set to inform the outcome. It was, however, surprising that the test data set outperformed the training set; typically models perform slightly worse against data they haven’t encountered before. This was the case with the linear and nonlinear regression-based models.

We also gauged accuracy against how long it took to train the various models (see plot below). As expected, the more accurate models generally took much longer to run than those that were less accurate.The model with the best MAPE score was also one of the longest to train.

# Performance vs accuracy  
dfr %>%   
 ggplot(aes(x=MAPE.Test, y=Train.Time, color=Model.Class)) +  
 geom\_point() +  
 xlab('Accuracy % (MAPE)') +  
 ylab('Training time (seconds)') +  
 ggtitle('Accuracy and training time by model class')



It should be noted that if this model is to be used in a real-time, production environment, a model with faster runtimes would be preferred over those that perform better but are slower to train.

Model selection

While our modeling yielded very good results by the top performers, we’ll take a closer look at the top two models to see if there are ways to improve it before selecting a final model. As indicated above, the top two performers were the Cubist model run against the full data set with 100 committees and five neighbors, and the random forest model with 26 predictors, also run against the full data set.

Because it was unclear whether the missing values in the categorical variable (Brand.Code) were intentionally uncoded or indeed missing, we decided to try rerunning the top two best-performing models after adjusting this variable. On the first run, we dropped the variable altogether. On the second run, we recoded missing Brand.Code values as “U” to represent unbranded products.

# Rerun top 2 models - full set, but drop Brand.Code  
dfm\_nobrand <- dfm\_full %>%  
 select(-starts\_with('Brand.Code'))  
  
# Create vector of training rows  
set.seed(77)  
train\_rows <- createDataPartition(dfm\_nobrand$PH, p=0.8, list=F)  
  
# Separate training from test  
dfm\_nobrand\_train <- dfm\_nobrand[train\_rows,]  
dfm\_nobrand\_test <- dfm\_nobrand[-train\_rows,]  
  
# Separate outcome from predictors  
trainx\_nobrand <- dfm\_nobrand\_train[,2:ncol(dfm\_nobrand\_train)]  
trainy\_nobrand <- dfm\_nobrand\_train[,1]  
testx\_nobrand <- dfm\_nobrand\_test[,2:ncol(dfm\_nobrand\_test)]  
testy\_nobrand <- dfm\_nobrand\_test[,1]  
  
# Parallel processing  
cl <- makePSOCKcluster(num\_cores)  
registerDoParallel(cl)  
  
# Cubist  
cubGrid <- expand.grid(.committees=c(seq(1, 10), seq(20, 100, by=10)), .neighbors=c(0, 1, 5, 9))  
tstart <- Sys.time()  
set.seed(77)  
fitcub\_nobrand <- train(trainx\_nobrand, trainy\_nobrand, method='cubist', tuneGrid=cubGrid, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcub\_nobrand  
predcub\_nobrand <- predict(fitcub\_nobrand, testx\_nobrand)  
dfr[37,] <- data.frame(  
 Data.Set='Rerun (no Brand.Code)',  
 Model='Cubist',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('committees=', fitcub\_nobrand$bestTune[['committees']],   
 ', neighbors=', fitcub\_nobrand$bestTune[['neighbors']]),   
 Train.RMSE=min(fitcub\_nobrand$results[['RMSE']]),  
 RMSE.Test=postResample(predcub\_nobrand, testy\_nobrand)[['RMSE']],  
 MAPE.Test=mape(predcub\_nobrand, testy\_nobrand),  
 Train.Time=round(telapsed, 3)  
)  
  
# Random Forest  
tstart <- Sys.time()  
set.seed(77)  
fitrf\_nobrand <- train(trainx\_nobrand, trainy\_nobrand, method='rf', tuneLength=10, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitrf\_nobrand  
predrf\_nobrand <- predict(fitrf\_nobrand, testx\_nobrand)  
dfr[39,] <- data.frame(  
 Data.Set='Rerun (no Brand.Code)',  
 Model='Random forest',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('mtry=', fitrf\_nobrand$bestTune[['mtry']]),   
 Train.RMSE=min(fitrf\_nobrand$results[['RMSE']]),  
 RMSE.Test=postResample(predrf\_nobrand, testy\_nobrand)[['RMSE']],  
 MAPE.Test=mape(predrf\_nobrand, testy\_nobrand),  
 Train.Time=round(telapsed, 3)  
)  
  
# Stop the parallel processing cluster  
stopCluster(cl)

# Rerun top 2 models - full set, but add uncoded Brand.Code values as U  
  
# Init the full data set  
dfm\_unbrand <- dfm  
  
# Remove observation with 12 missing values  
dfm\_unbrand <- dfm\_unbrand[(rowSums(is.na(dfm\_unbrand)) < 12),]  
  
# Remove the MFR predictor because it has 212 missing values (8.2% of the data)  
dfm\_unbrand <- dfm\_unbrand %>% select(-MFR)  
  
# Recode blank brand codes as "U"  
dfm\_unbrand <- dfm\_unbrand %>%  
 mutate(Brand.Code=ifelse(Brand.Code=='', 'U', as.character(Brand.Code)))  
  
# Factor Brand.Code (needed for imputation)  
dfm\_unbrand$Brand.Code <- factor(dfm\_unbrand$Brand.Code)  
  
# Use knnImputation to impute values  
dfm\_unbrand <- knnImputation(dfm\_unbrand, k=9)  
  
# Create dummies for factor and character variables  
dfm\_unbrand <- dummy\_cols(dfm\_unbrand, select\_columns='Brand.Code', remove\_first\_dummy=F, remove\_selected\_columns=T)  
  
# Create vector of training rows  
set.seed(77)  
train\_rows <- createDataPartition(dfm\_unbrand$PH, p=0.8, list=F)  
  
# Separate training from test  
dfm\_unbrand\_train <- dfm\_unbrand[train\_rows,]  
dfm\_unbrand\_test <- dfm\_unbrand[-train\_rows,]  
  
# Separate outcome from predictors  
trainx\_unbrand <- dfm\_unbrand\_train[,2:ncol(dfm\_unbrand\_train)]  
trainy\_unbrand <- dfm\_unbrand\_train[,1]  
testx\_unbrand <- dfm\_unbrand\_test[,2:ncol(dfm\_unbrand\_test)]  
testy\_unbrand <- dfm\_unbrand\_test[,1]  
  
# Parallel processing  
cl <- makePSOCKcluster(num\_cores)  
registerDoParallel(cl)  
  
# Cubist - full set - wider range of neighbors and committees  
cubGrid <- expand.grid(.committees=c(seq(1, 10), seq(20, 100, by=10)), .neighbors=c(0, 1, 5, 9))  
tstart <- Sys.time()  
set.seed(77)  
fitcub\_unbrand <- train(trainx\_unbrand, trainy\_unbrand, method='cubist', tuneGrid=cubGrid, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitcub\_unbrand  
predcub\_unbrand <- predict(fitcub\_unbrand, testx\_unbrand)  
dfr[38,] <- data.frame(  
 Data.Set='Rerun (keep unbranded)',  
 Model='Cubist',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('committees=', fitcub\_unbrand$bestTune[['committees']],   
 ', neighbors=', fitcub\_unbrand$bestTune[['neighbors']]),   
 Train.RMSE=min(fitcub\_unbrand$results[['RMSE']]),  
 RMSE.Test=postResample(predcub\_unbrand, testy\_unbrand)[['RMSE']],  
 MAPE.Test=mape(predcub\_unbrand, testy\_unbrand),  
 Train.Time=round(telapsed, 3)  
)  
  
# Random Forest  
tstart <- Sys.time()  
set.seed(77)  
fitrf\_unbrand <- train(trainx\_unbrand, trainy\_unbrand, method='rf', tuneLength=10, trControl=ctrl)  
telapsed <- as.numeric(difftime(Sys.time(), tstart ,units='secs'))  
# fitrf\_unbrand  
predrf\_unbrand <- predict(fitrf\_unbrand, testx\_unbrand)  
dfr[40,] <- data.frame(  
 Data.Set='Rerun (keep unbranded)',  
 Model='Random forest',   
 Model.Class='Tree',  
 Tuning.Parameters=paste0('mtry=', fitrf\_unbrand$bestTune[['mtry']]),   
 Train.RMSE=min(fitrf\_unbrand$results[['RMSE']]),  
 RMSE.Test=postResample(predrf\_unbrand, testy\_unbrand)[['RMSE']],  
 MAPE.Test=mape(predrf\_unbrand, testy\_unbrand),  
 Train.Time=round(telapsed, 3)  
)  
  
# Stop the parallel processing cluster  
stopCluster(cl)

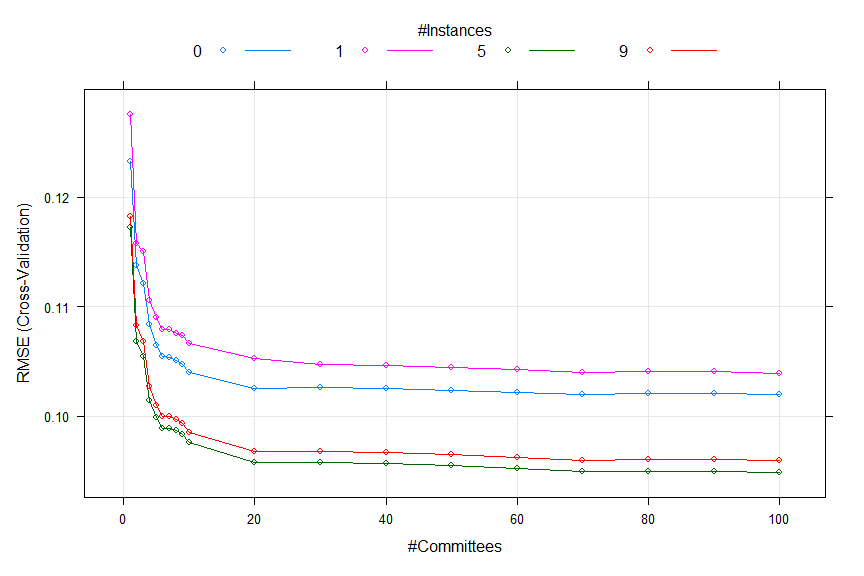
# Reprint top results  
dfr[c(16,18,34,36,37:40),] %>%  
 arrange(MAPE.Test) %>%  
 select(Model, Data.Set, Tuning.Parameters, Train.Time, RMSE.Train, RMSE.Test, MAPE.Test) %>%  
 flextable() %>%  
 width(width = 1) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Cubist/Random forest results (including reruns)')

Cubist/Random forest results (including reruns)

| Model | Data.Set | Tuning.Parameters | Train.Time | RMSE.Train | RMSE.Test | MAPE.Test |
| --- | --- | --- | --- | --- | --- | --- |
| Cubist | Full | committees=100, neighbors=5 | 387.711 | 0.09489731 | 0.09467403 | 0.8069488 |
| Cubist | Rerun (keep unbranded) | committees=50, neighbors=5 | 345.853 | 0.09558272 | 0.09502358 | 0.8090312 |
| Random Forest | Full | mtry=26 | 397.041 | 0.09573711 | 0.09518857 | 0.8278027 |
| Random forest | Rerun (keep unbranded) | mtry=31 | 420.192 | 0.09582768 | 0.09605555 | 0.8292539 |
| Cubist | Reduced | committees=30, neighbors=9 | 278.724 | 0.09590937 | 0.09602074 | 0.8406549 |
| Cubist | Rerun (no Brand.Code) | committees=70, neighbors=5 | 396.078 | 0.09882926 | 0.10185535 | 0.8444153 |
| Random Forest | Reduced | mtry=18 | 355.858 | 0.09756142 | 0.10008600 | 0.8564273 |
| Random forest | Rerun (no Brand.Code) | mtry=23 | 395.925 | 0.10338676 | 0.10653838 | 0.9037640 |

As shown, rerunning the models using different Brand.Code combinations did not improve the MAPE of the models run with the full data set. As such, the top-performing model remains the Cubist model run against the full data set (imputing missing Brand.Code values), having 100 committees and five neighbors. As shown on the plot below, performance drops sharply from 0 to around 5 committees, followed by a more gradual decline thereafter. Assuming that more committees means longer runtimes, it would be feasible to decrease the number of committees from 100 to 20 and still obtain almost equivalent performance. But if runtime isn’t a concern, we recommend 100 committees.

# Plot  
plot(fitcub\_full)



Variable importance

Having selected the optimal model, we’ll examine the importance of individual predictors within the model.

# Variable importance - cubist full set  
tmpdf <- varImp(fitcub\_full)$importance  
tmpdf$Variable <- row.names(tmpdf)  
tmpdf %>%  
 select(Variable, Overall) %>%  
 arrange(desc(Overall)) %>%  
 head(10) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Top 10 variables - Cubist model (full set)')

Top 10 variables - Cubist model (full set)

| Variable | Overall |
| --- | --- |
| Mnf.Flow | 100.00000 |
| Balling.Lvl | 64.81481 |
| Alch.Rel | 56.17284 |
| Balling | 54.93827 |
| Pressure.Vacuum | 51.23457 |
| Density | 51.23457 |
| Oxygen.Filler | 50.00000 |
| Bowl.Setpoint | 43.82716 |
| Air.Pressurer | 41.97531 |
| Temperature | 41.97531 |

As shown, Mnf.Flow was the most informative predictor in the model, followed by Bailing.Lvl and Alch.Rel.

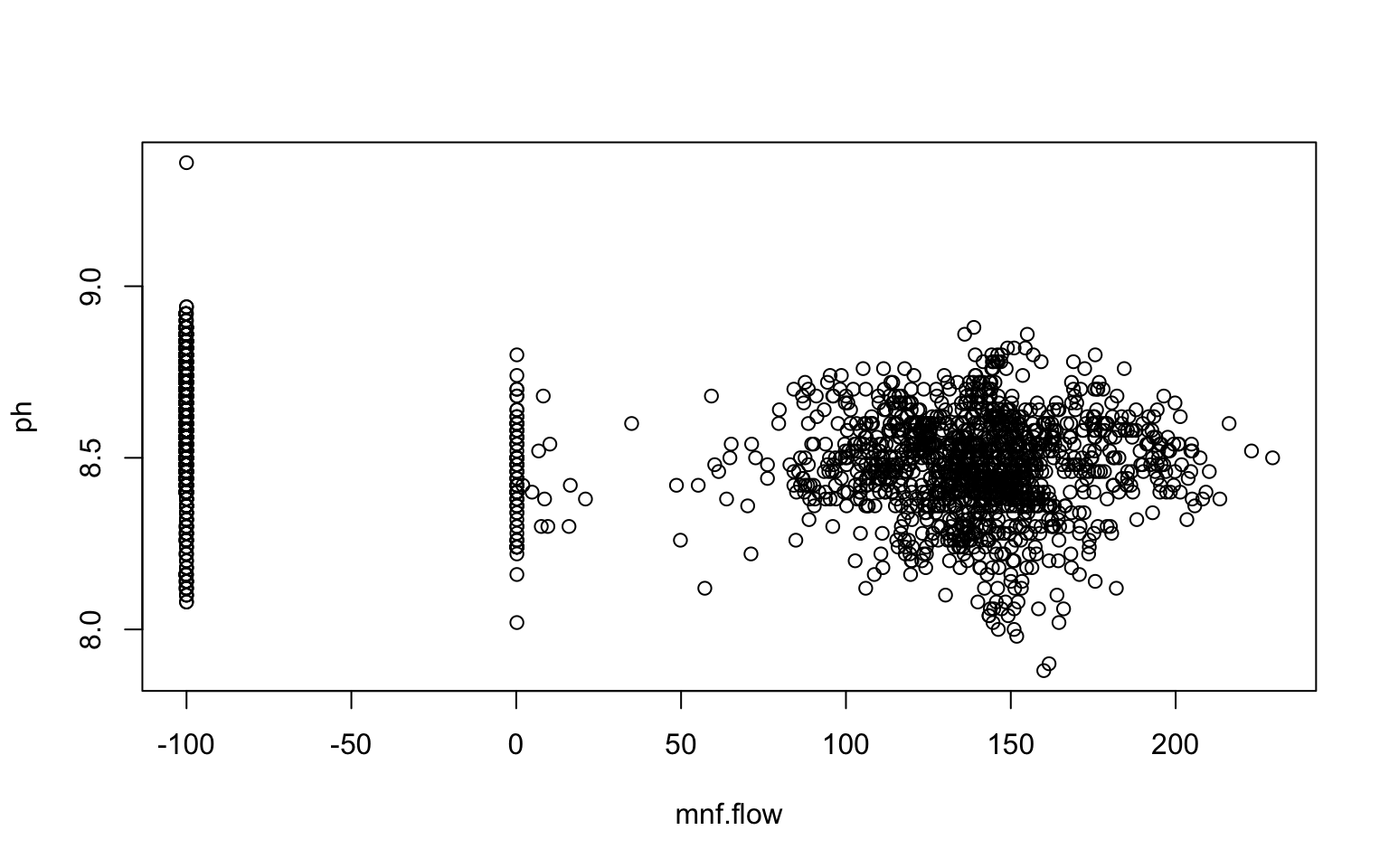
Cubist: The Cubist output contains variable usage statistics. It gives the percentage of times where each variable was used in a condition and/or a linear model. Note that this output will probably be inconsistent with the rules shown in the output from summary.cubist. At each split of the tree, Cubist saves a linear model (after feature selection) that is allowed to have terms for each variable used in the current split or any split above it. Quinlan (1992) discusses a smoothing algorithm where each model prediction is a linear combination of the parent and child model along the tree. As such, the final prediction is a function of all the linear models from the initial node to the terminal node. The percentages shown in the Cubist output reflects all the models involved in prediction (as opposed to the terminal models shown in the output). The variable importance used here is a linear combination of the usage in the rule conditions and the model.

We observe the mnf.flow variable is the most important in our Cubist model, let’s take a look it’s relationship with pH,

*# mnf.flow scatter*

dfm |>

select(Mnf.Flow, PH) |>

plot()

We can see there appears to be three groups of data, values around the -100 range, values around the 0 range, and values greater than zero. However, upon further inspection, we discover there are no zero values in the mnf.flow column, only values of 0.2. We discretize the variable and plot violin diagrams of the distributions,

*# mnf.flow violin*

*dfm |>*

*select(PH, Mnf.Flow) |>*

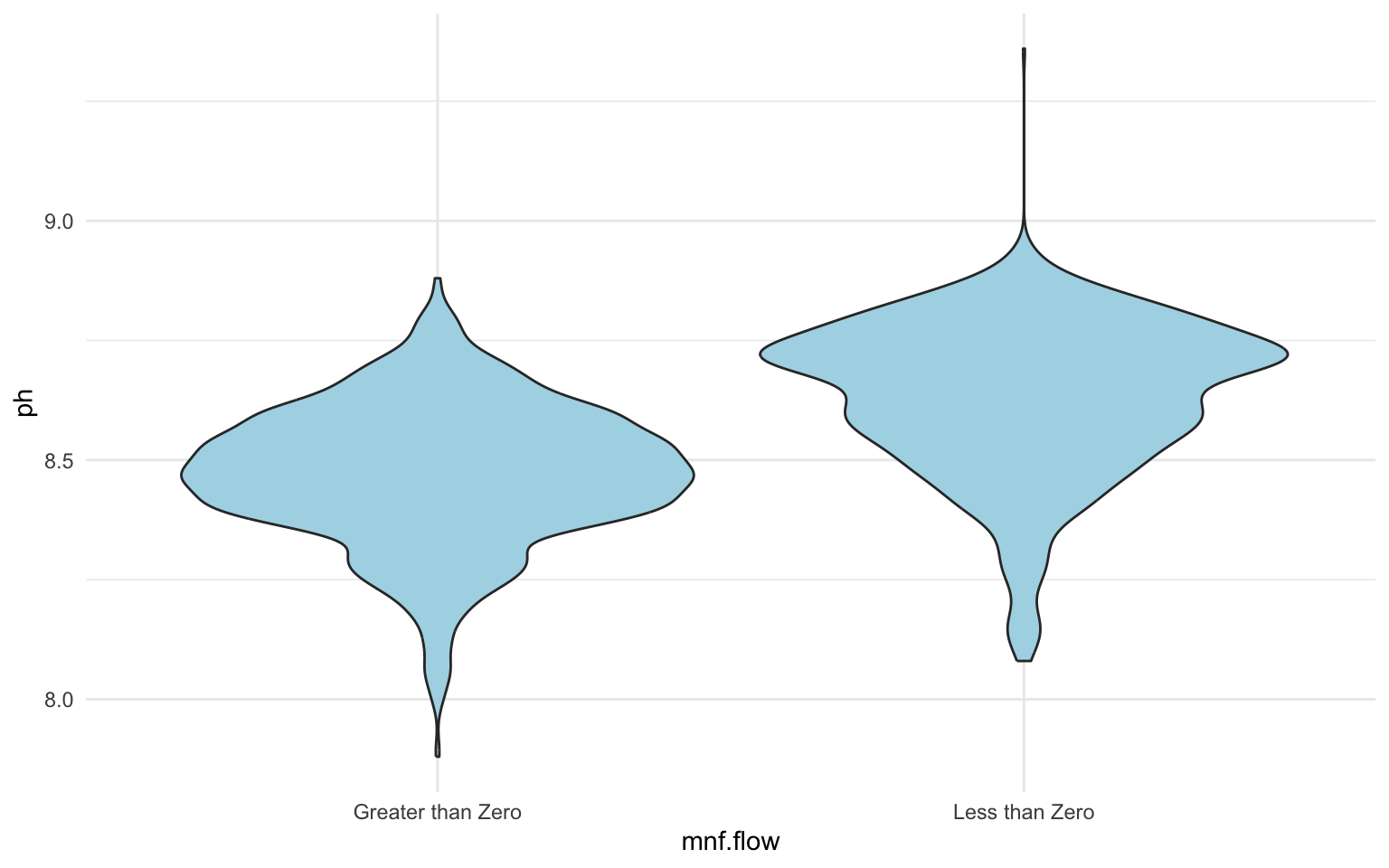
*na.omit() |>*

*mutate(Mnf.Flow = ifelse(Mnf.Flow < 0, "Less than Zero",*

*"Greater than Zero")) |>*

*ggplot(aes(x=Mnf.Flow, y=PH)) +*

*geom\_violin(fill="lightblue") +*

*theme\_minimal()*

We observe that the mnf.flow variable values which are greater than zero are likelier to fall within the critical pH range, as the observations with values less than zero contain the outliers in the range above 9.0 pH.

select(PH, Mnf.Flow) |>

na.omit() |>

mutate(Mnf.Flow = ifelse(Mnf.Flow <= 0, "Less than Zero",

"Greater than Zero")) |>

group\_by(Mnf.Flow) |>

summarize(mean.ph = mean(PH),

median.ph = median(PH),

min.ph = min(PH),

max.ph = max(PH),

sd.ph = sd(PH)) |>

as.data.frame() |>

t() |>

kable() |>

kable\_styling()

|  |  |  |
| --- | --- | --- |
| mnf.flow | Greater than Zero | Less than Zero |
| mean.ph | 8.470983 | 8.633001 |
| median.ph | 8.48 | 8.66 |
| min.ph | 7.88 | 8.08 |
| max.ph | 8.88 | 9.36 |
| sd.ph | 0.1449628 | 0.1608029 |

We can see that the mean, median, standard deviation, minimum and maximum pH are all higher for mnf.flow values below zero, providing evidence that observations with mnf.flow values greater than zero are likelier to fall within the critical pH range.

Evaluation data

Before we generate predictions on the evaluation data set, we’ll take do some cursory exploratory data analysis to verify that the evaluation data set isn’t vastly different than the training set.

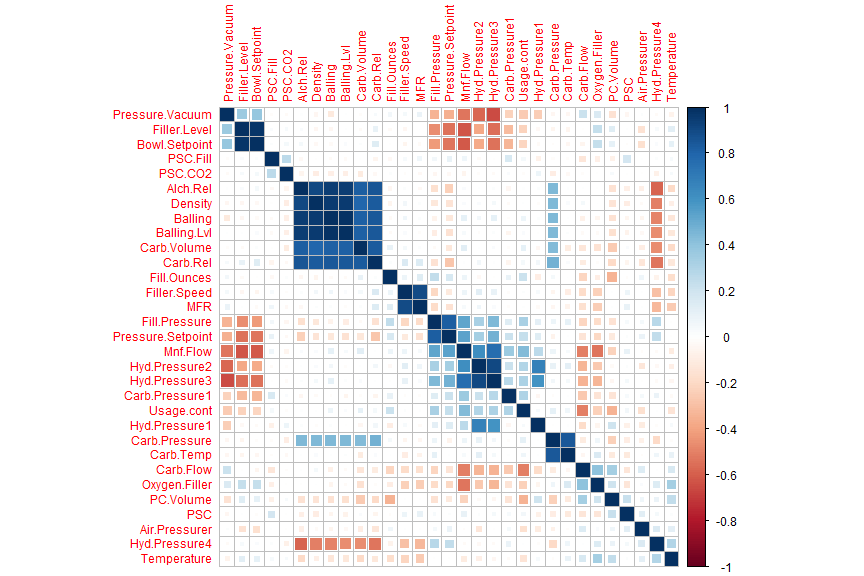
Exploratory data analysis

# Move outcome variable pH to front for easier access  
dfe <- dfe\_raw %>%  
 dplyr::select(PH, !matches('Brand.Code'), Brand.Code)  
summary(dfe)

## PH Carb.Volume Fill.Ounces PC.Volume   
## Mode:logical Min. :5.147 Min. :23.75 Min. :0.09867   
## NA's:267 1st Qu.:5.287 1st Qu.:23.92 1st Qu.:0.23333   
## Median :5.340 Median :23.97 Median :0.27533   
## Mean :5.369 Mean :23.97 Mean :0.27769   
## 3rd Qu.:5.465 3rd Qu.:24.01 3rd Qu.:0.32200   
## Max. :5.667 Max. :24.20 Max. :0.46400   
## NA's :1 NA's :6 NA's :4   
## Carb.Pressure Carb.Temp PSC PSC.Fill   
## Min. :60.20 Min. :130.0 Min. :0.00400 Min. :0.0200   
## 1st Qu.:65.30 1st Qu.:138.4 1st Qu.:0.04450 1st Qu.:0.1000   
## Median :68.00 Median :140.8 Median :0.07600 Median :0.1800   
## Mean :68.25 Mean :141.2 Mean :0.08545 Mean :0.1903   
## 3rd Qu.:70.60 3rd Qu.:143.8 3rd Qu.:0.11200 3rd Qu.:0.2600   
## Max. :77.60 Max. :154.0 Max. :0.24600 Max. :0.6200   
## NA's :1 NA's :5 NA's :3   
## PSC.CO2 Mnf.Flow Carb.Pressure1 Fill.Pressure   
## Min. :0.00000 Min. :-100.20 Min. :113.0 Min. :37.80   
## 1st Qu.:0.02000 1st Qu.:-100.00 1st Qu.:120.2 1st Qu.:46.00   
## Median :0.04000 Median : 0.20 Median :123.4 Median :47.80   
## Mean :0.05107 Mean : 21.03 Mean :123.0 Mean :48.14   
## 3rd Qu.:0.06000 3rd Qu.: 141.30 3rd Qu.:125.5 3rd Qu.:50.20   
## Max. :0.24000 Max. : 220.40 Max. :136.0 Max. :60.20   
## NA's :5 NA's :4 NA's :2   
## Hyd.Pressure1 Hyd.Pressure2 Hyd.Pressure3 Hyd.Pressure4   
## Min. :-50.00 Min. :-50.00 Min. :-50.00 Min. : 68.00   
## 1st Qu.: 0.00 1st Qu.: 0.00 1st Qu.: 0.00 1st Qu.: 90.00   
## Median : 10.40 Median : 26.80 Median : 27.70 Median : 98.00   
## Mean : 12.01 Mean : 20.11 Mean : 19.61 Mean : 97.84   
## 3rd Qu.: 20.40 3rd Qu.: 34.80 3rd Qu.: 33.00 3rd Qu.:104.00   
## Max. : 50.00 Max. : 61.40 Max. : 49.20 Max. :140.00   
## NA's :1 NA's :1 NA's :4   
## Filler.Level Filler.Speed Temperature Usage.cont Carb.Flow   
## Min. : 69.2 Min. :1006 Min. :63.80 Min. :12.90 Min. : 0   
## 1st Qu.:100.6 1st Qu.:3812 1st Qu.:65.40 1st Qu.:18.12 1st Qu.:1083   
## Median :118.6 Median :3978 Median :65.80 Median :21.44 Median :3038   
## Mean :110.3 Mean :3581 Mean :66.23 Mean :20.90 Mean :2409   
## 3rd Qu.:120.2 3rd Qu.:3996 3rd Qu.:66.60 3rd Qu.:23.74 3rd Qu.:3215   
## Max. :153.2 Max. :4020 Max. :75.40 Max. :24.60 Max. :3858   
## NA's :2 NA's :10 NA's :2 NA's :2   
## Density MFR Balling Pressure.Vacuum   
## Min. :0.060 Min. : 15.6 Min. :0.902 Min. :-6.400   
## 1st Qu.:0.920 1st Qu.:707.0 1st Qu.:1.498 1st Qu.:-5.600   
## Median :0.980 Median :724.6 Median :1.648 Median :-5.200   
## Mean :1.177 Mean :697.8 Mean :2.203 Mean :-5.174   
## 3rd Qu.:1.600 3rd Qu.:731.5 3rd Qu.:3.242 3rd Qu.:-4.800   
## Max. :1.840 Max. :784.8 Max. :3.788 Max. :-3.600   
## NA's :1 NA's :31 NA's :1 NA's :1   
## Oxygen.Filler Bowl.Setpoint Pressure.Setpoint Air.Pressurer   
## Min. :0.00240 Min. : 70.0 Min. :44.00 Min. :141.2   
## 1st Qu.:0.01960 1st Qu.:100.0 1st Qu.:46.00 1st Qu.:142.2   
## Median :0.03370 Median :120.0 Median :46.00 Median :142.6   
## Mean :0.04666 Mean :109.6 Mean :47.73 Mean :142.8   
## 3rd Qu.:0.05440 3rd Qu.:120.0 3rd Qu.:50.00 3rd Qu.:142.8   
## Max. :0.39800 Max. :130.0 Max. :52.00 Max. :147.2   
## NA's :3 NA's :1 NA's :2 NA's :1   
## Alch.Rel Carb.Rel Balling.Lvl Brand.Code   
## Min. :6.400 Min. :5.18 Min. :0.000 Length:267   
## 1st Qu.:6.540 1st Qu.:5.34 1st Qu.:1.380 Class :character   
## Median :6.580 Median :5.40 Median :1.480 Mode :character   
## Mean :6.907 Mean :5.44 Mean :2.051   
## 3rd Qu.:7.180 3rd Qu.:5.56 3rd Qu.:3.080   
## Max. :7.820 Max. :5.74 Max. :3.420   
## NA's :3 NA's :2

At first glance the evaluation set appears to be roughly on par with the training set. A number of missing values exist, but not in vast proportions.

# Factor categorical variable Brand.Code  
dfe$Brand.Code <- factor(dfe$Brand.Code)  
  
# Generate corr plot for pairwise correlations  
corr1 <- cor(dfe[,2:(ncol(dfe)-1)], use='complete')  
corrplot::corrplot(corr1, method='square', order='hclust', type='full', diag=T, tl.cex=0.75, cl.cex=0.75)



Like the training set data, some collinearity exists between pairs of variables. The same variables appear to be correlated as in the training set:

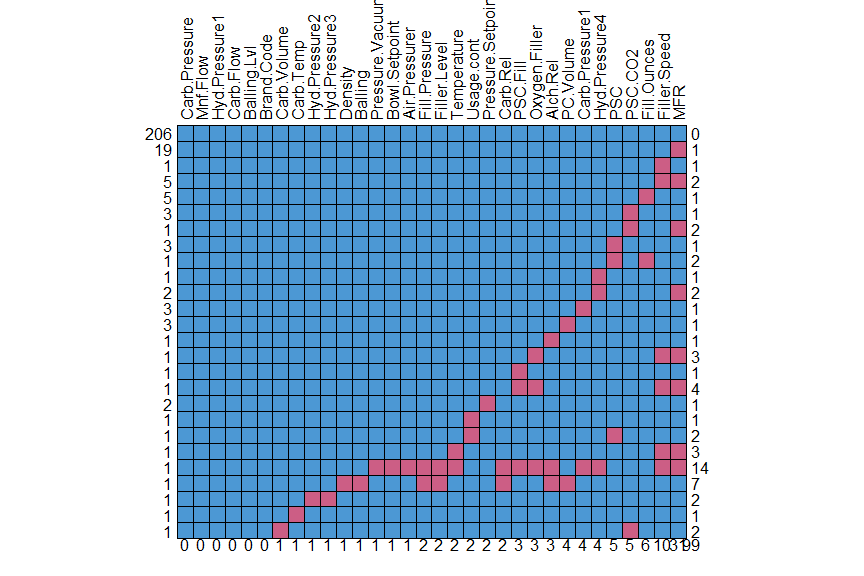
* Carb.Volume
* Carb.Rel
* Alch.Rel
* Density
* Bailing
* Bailing.Lvl

# Count NAs per column  
missing\_values <- data.frame(colSums(is.na(dfe[,-1])))  
missing\_values$Variable <- row.names(missing\_values)  
colnames(missing\_values) <- c('Missing.values', 'Variable')  
missing\_values %>%  
 dplyr::select(Variable, Missing.values) %>%  
 arrange(desc(Missing.values)) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Missing values')

Missing values

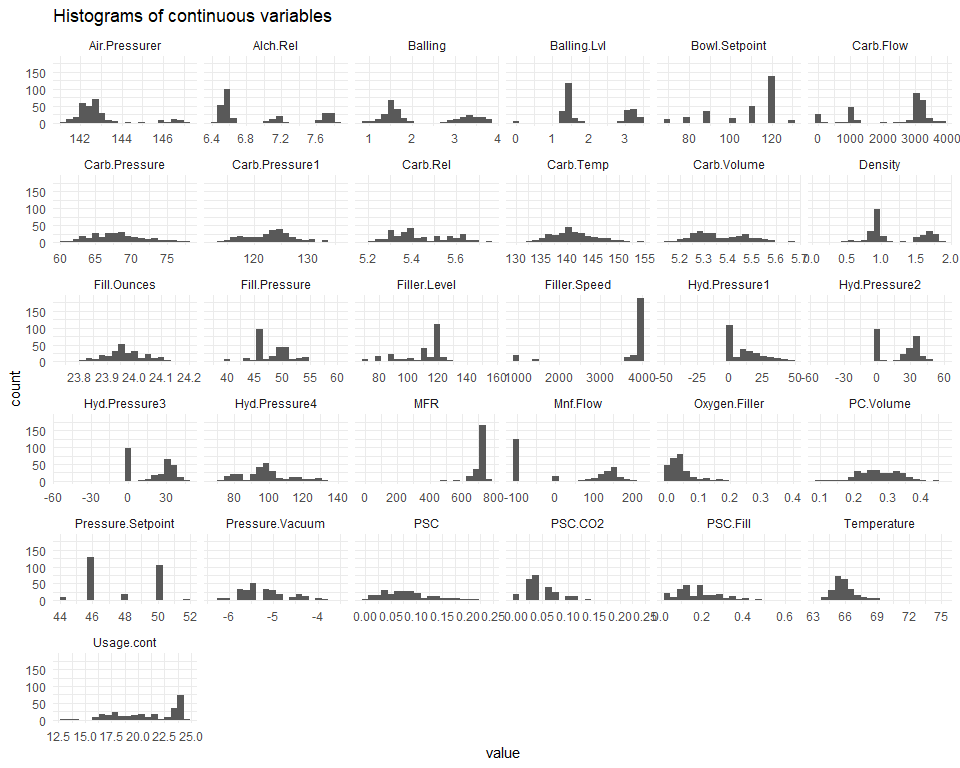
| Variable | Missing.values |
| --- | --- |
| MFR | 31 |
| Filler.Speed | 10 |
| Fill.Ounces | 6 |
| PSC | 5 |
| PSC.CO2 | 5 |
| PC.Volume | 4 |
| Carb.Pressure1 | 4 |
| Hyd.Pressure4 | 4 |
| PSC.Fill | 3 |
| Oxygen.Filler | 3 |
| Alch.Rel | 3 |
| Fill.Pressure | 2 |
| Filler.Level | 2 |
| Temperature | 2 |
| Usage.cont | 2 |
| Pressure.Setpoint | 2 |
| Carb.Rel | 2 |
| Carb.Volume | 1 |
| Carb.Temp | 1 |
| Hyd.Pressure2 | 1 |
| Hyd.Pressure3 | 1 |
| Density | 1 |
| Balling | 1 |
| Pressure.Vacuum | 1 |
| Bowl.Setpoint | 1 |
| Air.Pressurer | 1 |
| Carb.Pressure | 0 |
| Mnf.Flow | 0 |
| Hyd.Pressure1 | 0 |
| Carb.Flow | 0 |
| Balling.Lvl | 0 |
| Brand.Code | 0 |

# Missing value patterns  
invisible(md.pattern(dfe[,-1], rotate.names=T, plot=T))

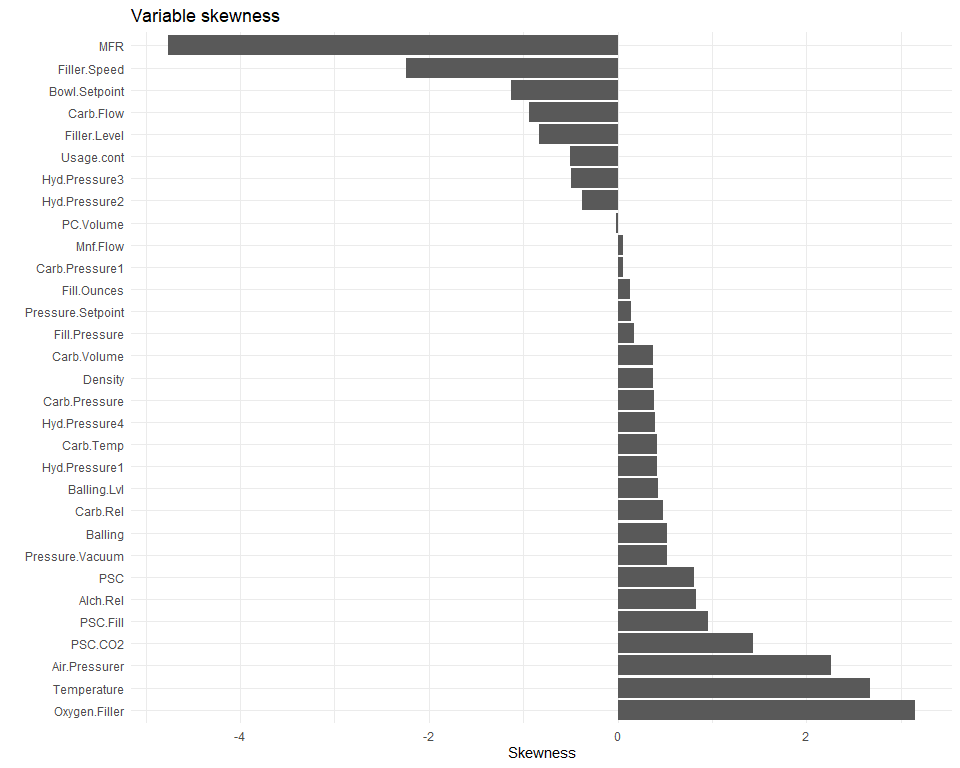


Missing values also appear to be on par with the training set, with MFR having the most missing values. There is one observation that contains 14 missing values, but since we’ll need to predict against every row in the evaluation set, we won’t remove it.

# Generate histograms of continuous variables  
dfe %>%  
 dplyr::select(c(2:(ncol(dfe)-1))) %>%  
 gather() %>%  
 ggplot(aes(x=value)) +  
 geom\_histogram(bins=20) +  
 facet\_wrap(~key, scales = 'free\_x') +  
 ggtitle('Histograms of continuous variables')

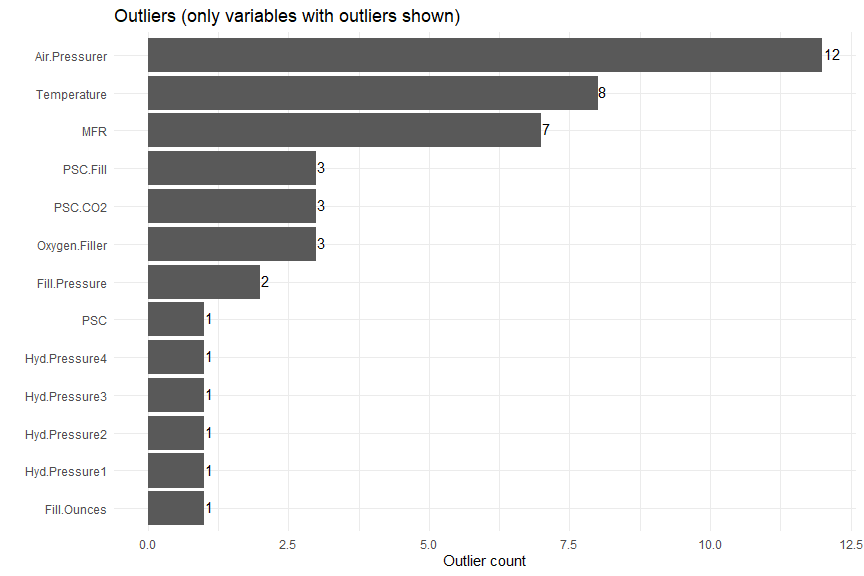


# Calculate skewness on columns  
colskewness <- data.frame(apply(dfe[,2:(ncol(dfe)-1)], 2, calcSkewness, type=1))  
colskewness$Variable <- row.names(colskewness)  
colnames(colskewness) <- c('Skewness', 'Variable')  
  
# Graph skewness values  
colskewness %>%  
 dplyr::select(Variable, Skewness) %>%  
 arrange(desc(abs(Skewness))) %>%  
 ggplot(aes(x=reorder(Variable, desc(Skewness)), y=Skewness)) +  
 geom\_bar(stat='identity') +  
 coord\_flip() +  
 ggtitle('Variable skewness') +  
 xlab('')



Roughly the same variables appear to be skewed.

# Set multiplier; outliers are considered as such when they lie outside of (multiplier) standard deviations from the mean  
mult <- 3  
outliers <- apply(dfe[,2:(ncol(dfe)-1)], 2, getOutliers, mult=mult)  
dfout <- data.frame(outliers)  
dfout <- data.frame(colSums(outliers))  
dfout$Variable <- row.names(dfout)  
colnames(dfout) <- c('Outlier.count', 'Variable')  
  
# Filter just those variables with outliers and sort by outlier count  
dfout <- dfout %>%  
 dplyr::select(Variable, Outlier.count) %>%  
 arrange(desc(Outlier.count)) %>%  
 filter(Outlier.count > 0)  
  
# Generate bar graph of outlier counts  
dfout %>%  
 ggplot(aes(x=reorder(Variable, Outlier.count), y=Outlier.count, label=Outlier.count)) +  
 geom\_bar(stat='identity') +  
 coord\_flip() +  
 geom\_text(check\_overlap=T, hjust=-0.1, nudge\_x=0.05) +  
 ggtitle('Outliers (only variables with outliers shown)') +  
 xlab('') + ylab('Outlier count')



The proportion of outliers also seems to be similar to that of the training set.

Data preparation

Now that we’ve verified the evaluation data is on par with the training data, we’ll proceed with data preparation. Since our best-performing model used the full data set, we’ll perform the same steps on the evaluation set that we used earlier on the full training set, namely the following:

1. Remove the MFR predictor because it has a high proportion of missing values (31 values, or 11.6% of the data).
2. Change the blank brand codes to NA so that they will be imputed.
3. Impute the missing values using KNN imputation.
4. Create dummy variables for the Brand.Code predictor.

# Init data frame, removing outcome variable, PH  
dfe\_eval <- dfe[,-1]  
  
# Remove the MFR predictor because it has a high proportion of missing data  
dfe\_eval <- dfe\_eval %>% select(-MFR)  
  
# Change blank brand codes to NA so they will be imputed  
dfe\_eval <- dfe\_eval %>%  
 mutate(Brand.Code=ifelse(Brand.Code=='', NA, as.character(Brand.Code)))  
  
# Factor Brand.Code (needed for imputation)  
dfe\_eval$Brand.Code <- factor(dfe\_eval$Brand.Code)  
  
# Use knnImputation to impute values  
dfe\_eval <- knnImputation(dfe\_eval, k=9)  
  
# Create dummies for factor and character variables  
dfe\_eval <- dummy\_cols(dfe\_eval, select\_columns='Brand.Code', remove\_first\_dummy=F, remove\_selected\_columns=T)

After preparing the data, we have 266 observations in 34 variables. The data is now ready for prediction.

Prediction

To predict pH in the evaluation data set, we’ll use the modeling results from the top performing model (Cubist run against the full data set). The first few predicted values are included below.

# Predict using the Cubist full model  
pred\_eval <- predict(fitcub\_full, dfe\_eval)  
df\_ph <- data.frame(pred\_eval)  
colnames(df\_ph) <- c('PH')  
  
# Show first few values  
data.frame(pred\_eval) %>%  
 head(10) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Predicted pH (first few values)') %>%  
 delete\_part( part = "header")

Predicted pH (first few values)

|  |
| --- |
| 8.628850 |
| 8.376658 |
| 8.519270 |
| 8.634902 |
| 8.452495 |
| 8.577147 |
| 8.453360 |
| 8.550534 |
| 8.566530 |
| 8.643949 |

# Save to csv  
write.csv(df\_ph, 'predicted\_ph.csv', row.names=F)

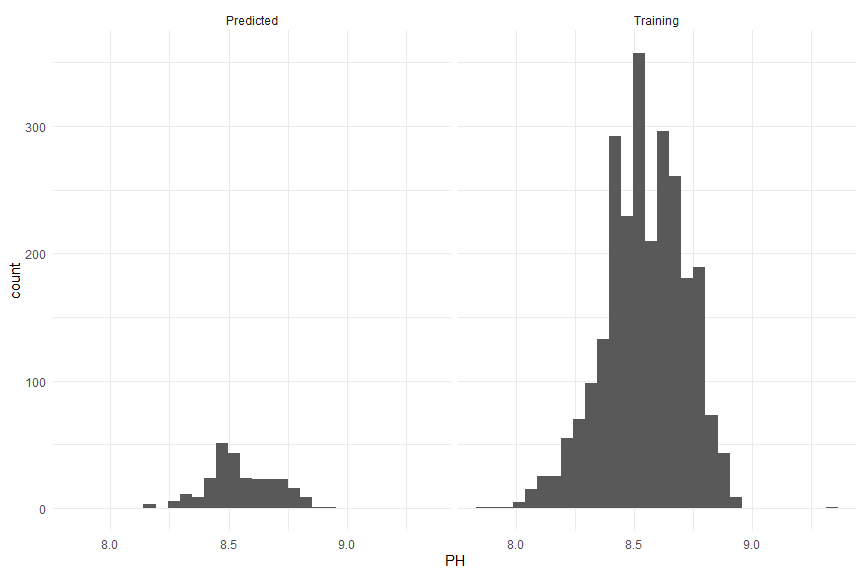
We’ll also generate some summary statistics for comparison purposes.

# Generate summary stats for predicted pH values  
tmpdf <- data.frame(as.matrix(summary(pred\_eval)))  
tmpdf$Statistic <- row.names(tmpdf)  
colnames(tmpdf) <- c('Evaluation', 'Statistic')  
tmpdf <- tmpdf %>%  
 rbind(data.frame(Evaluation=sd(pred\_eval), Statistic='Std dev'))  
  
# Generate summary stats for training pH values  
tmpdfm <- data.frame(as.matrix(summary(dfm\_full$PH)))  
tmpdfm$Statistic <- row.names(tmpdfm)  
colnames(tmpdfm) <- c('Training', 'Statistic')  
tmpdfm <- tmpdfm %>%  
 rbind(data.frame(Training=sd(dfm\_full$PH), Statistic='Std dev'))  
  
# Merge the summary stats to display in a table  
tmpdf2 <- tmpdfm %>%  
 merge(tmpdf, by=c('Statistic'))  
  
# Display summary  
tmpdf2 %>%  
 select(Statistic, Training, Evaluation) %>%  
 flextable() %>%  
 width(width = 2) %>%  
 fontsize(size = 10) %>%  
 line\_spacing(space = 1) %>%  
 hline(part = "all") %>%  
 set\_caption('Comparison of pH statistics')

Comparison of pH statistics

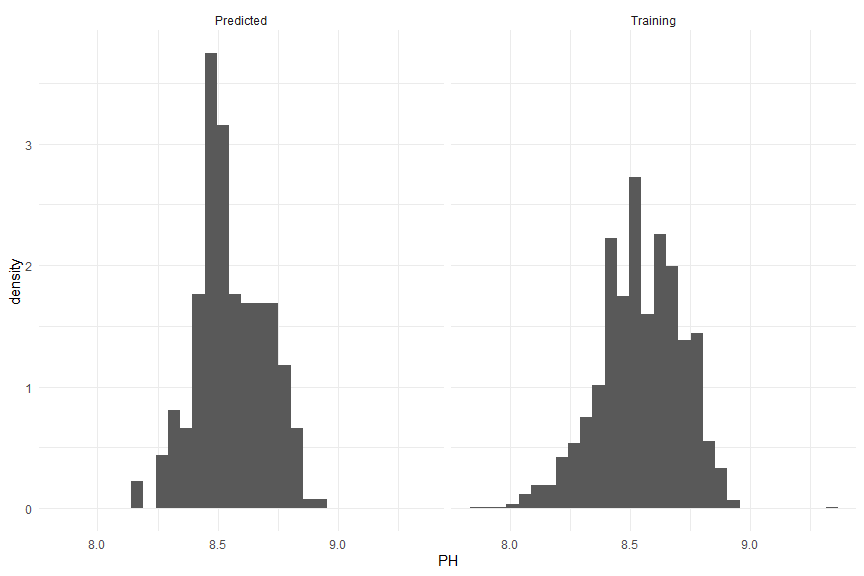
| Statistic | Training | Evaluation |
| --- | --- | --- |
| 1st Qu. | 8.4400000 | 8.4597397 |
| 3rd Qu. | 8.6800000 | 8.6659360 |
| Max. | 9.3600000 | 8.9317131 |
| Mean | 8.5457724 | 8.5525592 |
| Median | 8.5400000 | 8.5318241 |
| Min. | 7.8800000 | 8.1460953 |
| Std dev | 0.1724898 | 0.1459234 |

# Generate histogram of training vs predicted PH  
tmpdf3 <- df\_ph %>%  
 mutate(Set='Predicted') %>%  
 rbind(data.frame(PH=dfm\_full$PH) %>% mutate(Set='Training'))  
tmpdf3 %>%   
 ggplot(aes(x=PH)) +  
 geom\_histogram(bins=30) +  
 facet\_wrap('Set')



As shown in the table above, the mean, median, and first and third quartiles are all roughly even between the training and evaluation data. The minimum, maximum, and standard deviation of the training data show a wider range than that of the evaluation data, but this is natural given the greater number of samples in the training set.

# Generate histogram of training vs predicted PH  
tmpdf3 <- df\_ph %>%  
 mutate(Set='Predicted') %>%  
 rbind(data.frame(PH=dfm\_full$PH) %>% mutate(Set='Training'))  
tmpdf3 %>%   
 ggplot(aes(x=PH, y=after\_stat(density))) +  
 geom\_histogram(bins=30) +  
 facet\_wrap('Set')



The distributions look to be similar.

# Appendix 2: Regression Modeling Explanations

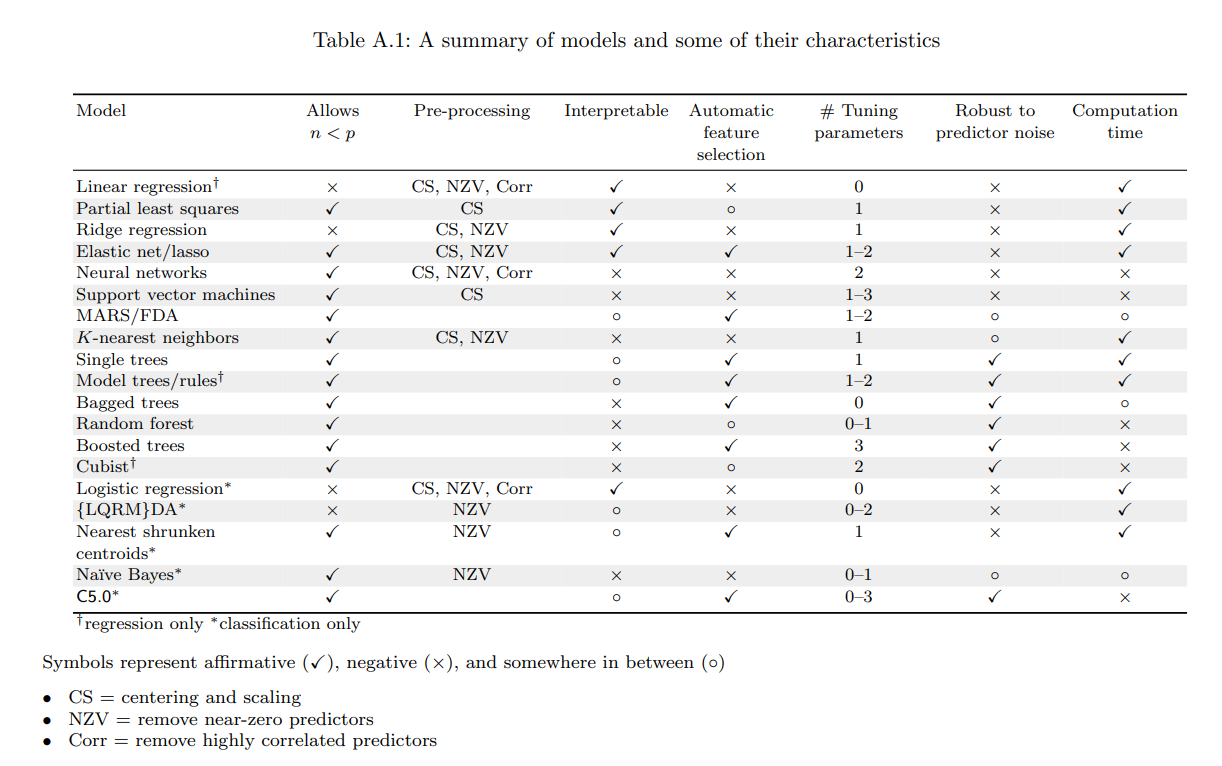
This table represents a summary of the material covered in *Applied Predictive Modeling* by Max Kuhn and Kjell Johnson - Chapters 6,7 and 8.

|  |  |  |
| --- | --- | --- |
| **Category** | **Model** | **Description** |
| Linear Regression | Ordinary Linear Regression (OLS) | Minimizes the sum-of-squared errors (SSE) by minimizing bias of all parameter estimates. For OLS, the predictors should not be collinear or multicollinearity and the number of samples should be greater than the number of predictors. If there is collinearity/multicollinearity, predictors should be removed. If there are more predictors than samples principle component analysis can be undertaken to decrease the number of predictors. If the predictor vs residual plot is not linear then that indicates relationships that are not linear - more terms can be added but this may overcomplicate the solution. One drawback of using SSE is that outliers can affect the regression. There are some remedies by using alternatives to SSE. |
| Partial Least Squares (PLS) | PLS is capable of handling collinearity as well as there being more predictors than samples. PLS is a supervised algorithm in that it finds relationships between the predictors that are highly correlated with the response. The relationships between the predictors are defined as latent variables and it can be difficult to see the relative contribution of each predictor - for this reason the “variable importance in the projection” function was created to show the relative importance of predictors. |
| Penalized Models | Penalized models involve controlling parameter estimates to reduce the SSE by adding a penalty to the SSE. There are three types of penalized models, the ridge, the lasso, and the elastic net regression. Features selection is when some parameters have a coefficient of 0 - meaning they are removed from the model. The ridge regression does not conduct feature selection, where the lasso does. The elastic net is a combination of the penalties of ridge as well as lasso. These penalties trade model bias and model variance, making the variance lower can result in a lower mean squared error (MSE) than an unbiased model. |
| Non-linear Regression | Neural Networks | Similar to PLS, neural networks outputs are modeled by “unobserved” or “hidden” variables which are linear combinations of the original predictors but there are no constraints that define the linear combinations. Usually parameters start at random values and the SSE of the residuals is minimized - however solutions may not be global solutions which means that it is not clear if any one set of parameters is better than another. Neural networks can overfit relationships because they have so many coefficients. This is a single-layer-feed-forward network, there are others with more layers of hidden units and loops between layers. To handle the multiple local solutions - model averaging can be done by starting the models at different initial points and averaging the resulting models. Note that high correlation between predictor variables negatively affects neural networks. |
| Multivariate Adaptive Regression Splines (MARS) | MARS splits the predictors into two groups and models the linear relationships between the predictor and response using a “hockey stick” function creating a “piecewise linear model” where each feature represents just a part of the original data. Each potential cut off point for the hockey stick is evaluated. Once the initial MARS model is created - it then goes through a “pruning” phase where it cuts out features that had a small impact of the error. MARS automatically conducts feature selection and is interpretable because of the linear combinations of predictors. Additionally, MARS does not require a lot of preprocessing of the data. Near-zero-variance predictors will be removed. Correlated predictors do not affect performance but can affect interpretation because of redundant information. |
| Support Vector Machines (SVM) | SVMs define a threshold where data with residuals within the threshold do not contribute to the regression fit and outside of it only contribute a linear amount rather than squared. This means that outliers do not have an outsized effect. The linear SVM is written as a function of unknown parameters (as many unknown parameters as data points) and the individual training set data points are required for new predictions. The training set data points used to create regression lines are called “support vectors”. |
| K-nearest neighbors (KNN) | KNN predicts new samples using a number of the closest samples from the training set. The prediction is the mean of the nearest neighbors predictions. The distance can vary between Euclidean (linear), or other forms depending on the context. Because KNN includes distances - scale becomes especially important and additionally it is important that missing values are imputed. Additionally the model performs poorly if the nearest neighbors are not predictive or if there is noise - removing noise is a pre-processing step. |
| Nonlinear Regression - Trees and Rule-Based Models | Basic Regression Trees | Tree based models are a series of if-then statements that use thresholds of the predictor to make splits that ultimately lead to a terminal node or leave (which is the predicted value). Tree models are very interpretable and can handle different predictor types without preprocessing. Shortcomings of trees are instability and predictor performance. Basic regression trees have terminal nodes that are constants (not equations). One method for building this tree is called classification and regression tree (CART) which finds predictors to split on which minimize SSE after building the initial tree it is pruned back that still minimizes a penalized SSE to avoid overfitting. Trees can handle missing data by ignoring them. Also trees can conduct feature selection. If predictors are correlated - the choice of predictor can be random. Disadvantages of basic regression trees is they can perform worse than other models, they may not capture intricacies of the data, they can be unstable, predictors with more distinct values are favored and if there are too many missing values selection becomes biased. Another option is “conditional inference trees' ' which use statistical hypothesis tests to search all predictors for all split points. P-values put the predictors on the same scale. This type of tree formation does not prune. |
| Regression Model Trees | In comparison to basic regression trees, regression model trees use different split criteria, the terminal nodes are linear models, and predictions are a combination of predictions from different models along a tree path. |
| Rule-Based Models | Trees can be simplified by removing rules or removing conditions for a rule. |
| Bagged Trees | Bagging stands for “bootstrap aggregation” and uses bootstrapping plus regression to make an “ensemble”. The advantages of this method are that variance is reduced through the aggregation process. Trees for different samples are built and then averaged lowering the overall variance. |
| Random Forests | Correlation between trees limits the variance reduction capacity. Random forests improve on bagging in that they add randomly selected predictors at each split. |
| Boosting | Boosting is where weak classifiers are combined into an ensemble that has improved predictive performance. “Gradient boosting machines” include classification and regression. |
| Cubist | In the cubist model, models are combined using a linear combination of two models. Cubist combines linear models at each node into one smoothed equation. |

# 

# Appendix 3: Textbook Model Summary

For reference, we have attached Appendix A, “Table A.1: A summary of models and some of their characteristics” from the textbook *Applied Predictive Modeling* by Max Kuhn and Kjell Johnson because it contains helpful information about each of the different types of models that we used for this project (pg. 550).

****