Predicting Beverage pH

S. Kigamba, L. Li, P. Maloney, D. Moscoe, and D. Moste

7/17/2021

## Introduction

pH is a key performance indicator for the beverage manufacturing process. Because beverage products must maintain a pH within a critical range, it’s important to understand how pH relates to other quantifiable aspects of beverage manufacturing. In this report, we seek a model for predicting beverage pH based on data about the beverage itself, along with its manufacturing and bottling process.

In this report, we select the optimal model, and summarize the steps in the model building process. Our criterion for a successful model is low mean absolute percent error (MAPE) when the model is run on test data. We also consider whether the model provides insight into the most important variables affecting pH. In the sections below, we describe the data, sketch our modeling process, and detail the optimal model for predicting pH. We also describe other models that performed nearly as well as the optimal model.

## About the data

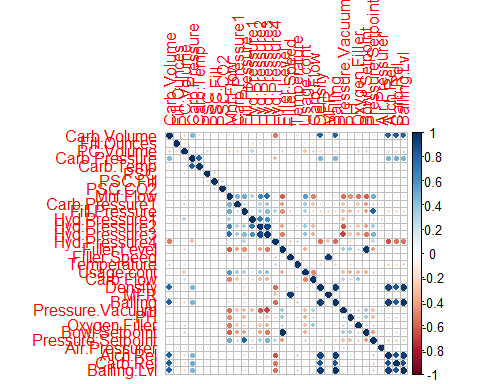
The data set contains information on 2,571 samples of 24-ounce bottled beverages. Most samples comprise information on 33 variables, such as density, temperature, and pH. Overall, less than 1% of values are missing from the data set. We found no pattern in the missing data.

With the exception of Brand Code, every variable is quantitative. Some variables, especially Hyd Pressure1, exhibit low variance. Other variables are highly correlated, which suggests the data set contains some redundant information. We also notice significant skewness in some of the variables.

# Load the datasets  
initial\_import.df <- read.csv("https://raw.githubusercontent.com/dmoste/DATA624/main/Project2/StudentDataTOMODEL.csv")  
to\_predict.df <- read.csv("https://raw.githubusercontent.com/dmoste/DATA624/main/Project2/StudentEvaluationTOPREDICT.csv")

A correlation plot shows the pairwise correlations in the data set:

corr\_matrix <- initial\_import.df %>%  
 keep(is.numeric) %>%  
 drop\_na() %>%  
 cor(method = "pearson")  
  
corrplot::corrplot(corr\_matrix, method = "circle", is.corr = TRUE)

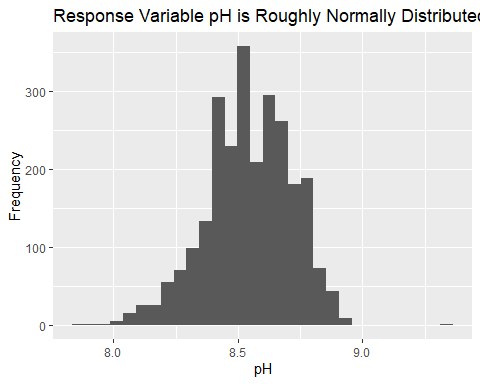


The response variable, pH, is roughly normally distributed, with mean 8.55 and standard deviation 0.173.

ggplot(data = initial\_import.df, aes(x = PH)) +  
 geom\_histogram() +  
 xlab("pH") +  
 ylab("Frequency") +  
 ggtitle("Response Variable pH is Roughly Normally Distributed")

## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 4 rows containing non-finite values (stat\_bin).

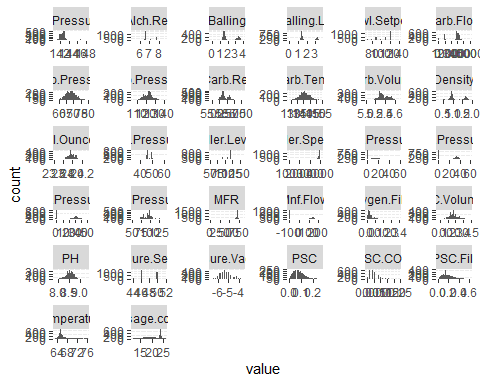


The explanatory variables exhibit a variety of distributions.

initial\_import.df %>%  
 keep(is.numeric) %>%  
 gather() %>%  
 ggplot(aes(value)) +  
 facet\_wrap(~ key, scales = "free") +   
 geom\_histogram()

## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 724 rows containing non-finite values (stat\_bin).



## Our modeling process

In this report we explore a range of linear models, tree-based models, and neural networks to identify a procedure that is highly accurate in predicting the pH of a previously unseen beverage. For each model, we take the following steps:

1. Impute missing data if necessary;
2. Transform data to address skewness, outliers, and low-variance variables;
3. Check that data conform to the assumptions of the model;
4. Fit a model and use cross-validation or another procedure to optimize parameters;
5. Examine residuals;
6. Compute model metrics.

## Summary of models

We built six models in total and used MAPE and RMSE scores to evaluate model performance. The summary table below shows the models and their corresponding MAPE and RMSE scores. We noticed that distance and regression tree models performed the best, followed by linear models. Neural Networks (nonlinear) model had the worst performance overall.

Type <- c("OLS", "PLS", "Elastic Net", "KNN", "Neural Nets", "Random Forest")  
Parameters <- c("None", "Components = 13", "Lambda = 0, Fraction = 1", "y", "Hidden Units", "ntrees")  
MAPE <- c(1.22, 1.19, 1.24, 0.91, 1.34, 0.93)  
RMSE <- c(0.135, 0.134, 0.139, 0.11, 0.14, 0.10)  
  
df <- data.frame(Type, Parameters, MAPE, RMSE)  
df\_sort <- df[with(df, order(MAPE)), ]  
  
kable(df\_sort, caption = "Summary of models")

Summary of models

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Type | Parameters | MAPE | RMSE |
| 4 | KNN | y | 0.91 | 0.110 |
| 6 | Random Forest | ntrees | 0.93 | 0.100 |
| 2 | PLS | Components = 13 | 1.19 | 0.134 |
| 1 | OLS | None | 1.22 | 0.135 |
| 3 | Elastic Net | Lambda = 0, Fraction = 1 | 1.24 | 0.139 |
| 5 | Neural Nets | Hidden Units | 1.34 | 0.140 |

## Optimal model: K-Nearest Neighbors

# Our Goal

Since our goal was to create a model with the most predictive power (based on MAPE), we chose to sacrifice a little bit on interprability for the sake of performance. With this in mind, the model we chose to use to make predictions was the one with the lowest MAPE, which was a KNN model.

In this report, we show the complete process for building the KNN model. We also included the other models, however, for reasons of simplicity and readability, we only highlighted the fitted model and model metrics.

# Imputation

To create the model, the data needed to undergo some preprocessing. The first step of this was to impute any missing values, which was done by replacing with the mean of each predictor. This method was chosen since the number of missing values was extremely small and since the large number of predictor variables meant that the distance between observations shouldn’t be impacted too much by a single missing predictor that was set to the mean.

library(plyr)

## ------------------------------------------------------------------------------

## You have loaded plyr after dplyr - this is likely to cause problems.  
## If you need functions from both plyr and dplyr, please load plyr first, then dplyr:  
## library(plyr); library(dplyr)

## ------------------------------------------------------------------------------

##   
## Attaching package: 'plyr'

## The following objects are masked from 'package:dplyr':  
##   
## arrange, count, desc, failwith, id, mutate, rename, summarise,  
## summarize

## The following object is masked from 'package:purrr':  
##   
## compact

data <- read.csv("https://raw.githubusercontent.com/dmoste/DATA624/main/Project2/StudentDataTOMODEL.csv")  
  
data[,1] <- mapvalues(data[,1],  
 from = c("A","B","C","D",""),  
 to = c(1,2,3,4,NA))  
data[,1] <- as.integer(data[,1])  
  
# Removing the response variable since I don't want to impute or transform these values  
drops <- c("PH")  
features <- data[,!(names(data) %in% drops)]  
  
na\_to\_mean <- function(x) replace(x, is.na(x), mean(x, na.rm = TRUE))  
features[] <- lapply(features, na\_to\_mean)

# Transformation

Next, the data had to be transformed. KNN is highly susceptible to data that is on different scales (large values will be much further from each other than small values). With this in mind, we chose to center and scale all of our predictors after running BoxCox transformation where neccessary. The final step of preprocessing was to remove any predictors that had near-zero variance so that there were no overlapping predictors.

library(caret)  
  
trans <- preProcess(features,  
 method = c("BoxCox", "center", "scale"))  
transformed\_feat <- predict(trans, features)  
  
nzv <- nearZeroVar(transformed\_feat, saveMetrics = TRUE)  
nzv[nzv[,"nzv"] == TRUE,]

## freqRatio percentUnique zeroVar nzv  
## Hyd.Pressure1 31.11111 9.568261 FALSE TRUE

# Removing Hy.Pressure1 since it has near zero variance  
drops <- c("Hyd.Pressure1")  
transformed\_feat <- transformed\_feat[,!(names(transformed\_feat) %in% drops)]

# Building Models

At this point, we were ready to build our models. We started by spliting our data into training and testing sets.

processed <- cbind(data[,26], transformed\_feat)  
names(processed)[1] <- ("PH")  
  
processed <- processed[complete.cases(processed),]  
  
set.seed(54321)  
train\_ind <- sample(seq\_len(nrow(processed)),  
 size = floor(0.75\*nrow(processed)))  
  
train <- processed[train\_ind,]  
test <- processed[-train\_ind,]

There are several important factors that need to be considered for any KNN model. These factors are number of nearest neighbors, distance formula, and weighting.

#### Number of Nearest Neighbors - k

KNN stands for k-nearest neighbors where the k is a stand in for how many neighbors are used in determining the prediction. This is a tuneable feature of any KNN model and is best found through creating a variety of models with different values and then reviewing the appropriate prediction metric.

#### Distance Formula

The distance formula is the way in which the distance between two observations is computed. Some options here include Manhattan, Euclidean, Cosine, Jaccard, among many others.

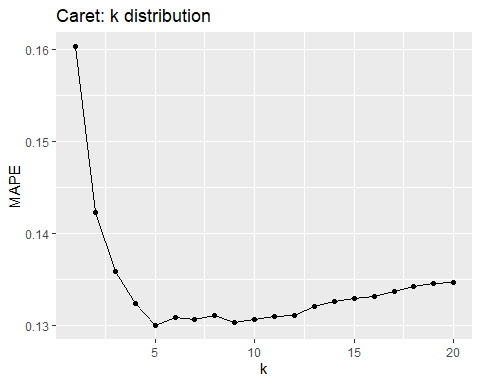
#### Weighting (also known as kernel)

This describes, quite literally, how much weight is given to each observation. A common technique is to give more weight to observations that are closer to the point in question. Like distance, there are many different options/formulas to use to determine weight. These include rectangular, triangular, buweight, triweight, and many many more.

#### Model 1

Our first KNN model was built using the caret library. This model found a minimum MAPE of 1.098% with a k value of 5. The values for distance and weight are not accessible via this package.

library(caret)  
library(ggplot2)  
library(tidyverse)  
  
#### train from caret ####  
knnModel <- train(train[,-1],  
 train[,1],  
 method = "knn",  
 tuneGrid = data.frame(.k = 1:20),  
 trControl = trainControl(method = "cv"))  
  
ggplot(data = knnModel$results, aes(x = k, y = RMSE)) +  
 geom\_line() +  
 geom\_point() +  
 labs(title = "Caret: k distribution",  
 x = "k",  
 y = "MAPE")



# Check best model  
knnPred <- predict(knnModel, newdata = test[,-1])  
  
model1 <- data.frame(cbind(knnPred,test[,1]))  
colnames(model1) <- c("predicted","actual")  
model1 <- model1 %>%  
 mutate(pe = abs(actual - predicted)/actual)  
  
MAPE <- (mean(model1$pe))\*100  
MAPE

## [1] 1.098103

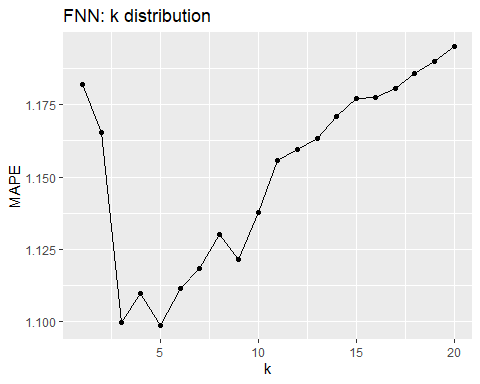
#### Model 2

The second model was built using the fnn library. This model gave a minimum MAPE of 1.098% with a k value of 5.This is the same as the caret model. Again, changing weights and distances was not accessible through this library.

library(FNN)

## Warning: package 'FNN' was built under R version 4.0.5

library(ggplot2)  
  
fnn\_func <- function(train\_x, train\_y, test\_x, test\_y){  
 mape\_df <- data.frame(matrix(nrow = 0, ncol = 2))  
   
 for(i in 1:20){  
 knn\_fnn <- knn.reg(train = train\_x,  
 test = test\_x,  
 y = train\_y,  
 k = i,  
 algorithm = "brute")  
   
 mape <- mean(abs(test\_y - knn\_fnn$pred)/test\_y)\*100  
 mape\_df <- rbind(mape\_df,c(i,mape))  
 }  
  
 colnames(mape\_df) <- c("k", "MAPE")  
 mape\_df[,1] <- as.integer(mape\_df[,1])  
 mape\_df[,2] <- as.numeric(mape\_df[,2])  
 return(mape\_df)  
}  
  
fnn\_mape <- fnn\_func(train[,-1], train[,1], test[,-1], test[,1])  
  
ggplot(data = fnn\_mape, aes(x = k, y = MAPE)) +  
 geom\_line() +  
 geom\_point() +  
 labs(title = "FNN: k distribution",  
 x = "k",  
 y = "MAPE")



#### Model 3

The third model we built used the kknn library. With this library we were able to dest different distances as well as different weights (kernels). This model found that a k value of 9 with a distance of 1 (Manhattan) and a weighting function of triweight produced the best model with a MAPE of 0.845%.

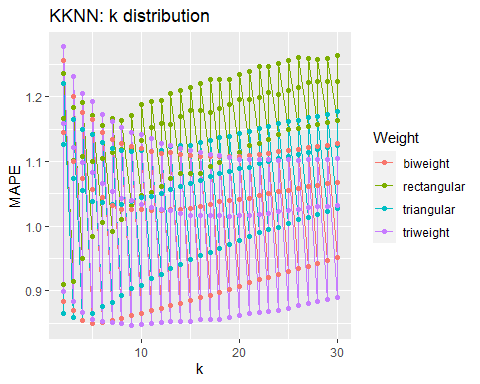
library(kknn)

## Warning: package 'kknn' was built under R version 4.0.5

##   
## Attaching package: 'kknn'

## The following object is masked from 'package:caret':  
##   
## contr.dummy

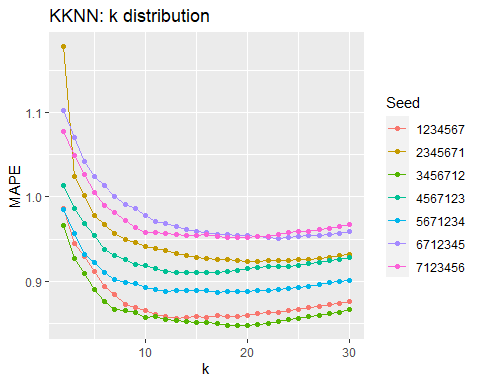
library(ggplot2)  
  
kknn\_func <- function(train\_x, train\_y, test\_x, test\_y){  
 mape\_df <- data.frame(matrix(nrow = 0, ncol = 4))  
   
 weights <- c("rectangular","triangular",  
 "biweight","triweight")  
   
 for(d in 1:3){  
 for(w in weights){  
 for(i in 2:30){  
 kknnModel <- kknn(train\_y ~ .,  
 train\_x,  
 test\_x,  
 k = i,  
 distance = d,  
 kernel = w)  
   
 mape <- mean(abs(test\_y - kknnModel$fitted.values)/test\_y)\*100  
 mape\_df <- rbind(mape\_df,c(i,mape,w,d))  
 }  
 }  
 }  
 colnames(mape\_df) <- c("k", "MAPE","Weight","Distance")  
 mape\_df[,1] <- as.integer(mape\_df[,1])  
 mape\_df[,2] <- as.numeric(mape\_df[,2])  
 mape\_df[,4] <- as.integer(mape\_df[,4])  
 return(mape\_df)  
}  
  
kknn\_mape <- kknn\_func(train[,-1], train[,1], test[,-1], test[,1])  
  
ggplot(data = kknn\_mape, aes(x = k, y = MAPE, color = Weight)) +  
 geom\_line() +  
 geom\_point() +  
 labs(title = "KKNN: k distribution",  
 x = "k",  
 y = "MAPE")



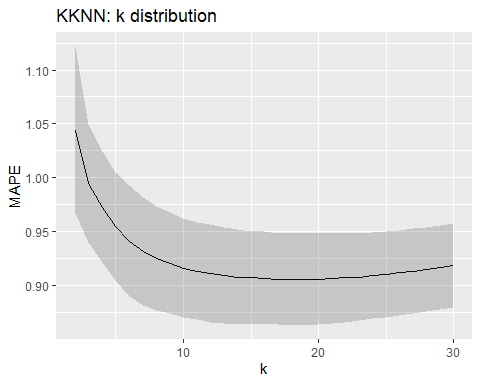
#### Tuning Model 3

Since the third model far outperformed the others, we decided to tune it by scrambling our train/test sets and finding an optimal value of k while using a triweight weighting function and Manhattan distance. We found that the best values for MAPE are all really close. K values between 17 and 20 all produce a MAPE of approximately 0.906%, with k = 18 being the best value by the slimmest of margins.

library(kknn)  
library(ggplot2)  
  
# Changing the kkhn function to accept seed values and only run a triweight model on Manhattan distance  
kknn\_func <- function(train\_x, train\_y, test\_x, test\_y, seed){  
 mape\_df <- data.frame(matrix(nrow = 0, ncol = 4))  
   
 for(i in 2:30){  
 kknnModel <- kknn(train\_y ~ .,  
 train\_x,  
 test\_x,  
 k = i,  
 distance = 1,  
 kernel = "triweight")  
   
 mape <- mean(abs(test\_y - kknnModel$fitted.values)/test\_y)\*100  
 rmse <- sqrt(mean((test\_y - kknnModel$fitted.values)^2))  
 mape\_df <- rbind(mape\_df,c(i,mape,rmse,seed))  
   
 colnames(mape\_df) <- c("k", "MAPE", "RMSE", "Seed")  
 mape\_df[,1] <- as.integer(mape\_df[,1])  
 mape\_df[,2] <- as.numeric(mape\_df[,2])  
 mape\_df[,3] <- as.numeric(mape\_df[,3])  
 mape\_df[,4] <- as.factor(mape\_df[,4])  
 }  
 return(mape\_df)  
}  
  
# Re-sample the data with 7 different test/train sets  
kknn\_mape <- data.frame(matrix(nrow = 0, ncol = 4))  
seeds <- c(1234567,2345671,3456712,4567123,5671234,6712345,7123456)  
  
for(i in seeds){  
 set.seed(i)  
 train\_ind3 <- sample(seq\_len(nrow(processed)),  
 size = floor(0.75\*nrow(processed)))  
   
 train3 <- processed[train\_ind3,]  
 test3 <- processed[-train\_ind3,]  
   
 current\_mape <- kknn\_func(train3[,-1],  
 train3[,1],  
 test3[,-1],  
 test3[,1],  
 i)  
 kknn\_mape <- rbind(kknn\_mape, current\_mape)  
}  
  
colnames(kknn\_mape) <- c("k", "MAPE", "RMSE", "Seed")  
kknn\_mape[,1] <- as.integer(kknn\_mape[,1])  
kknn\_mape[,2] <- as.numeric(kknn\_mape[,2])  
kknn\_mape[,3] <- as.numeric(kknn\_mape[,3])  
kknn\_mape[,4] <- as.factor(kknn\_mape[,4])  
  
ggplot(data = kknn\_mape, aes(x = k, y = MAPE, color = Seed)) +  
 geom\_line() +  
 geom\_point() +  
 labs(title = "KKNN: k distribution",  
 x = "k",  
 y = "MAPE")



# Check which value of k performs the best on average  
mape\_mean <- aggregate(kknn\_mape[,2], list(kknn\_mape$k), mean)  
mape\_sd <- aggregate(kknn\_mape[,2], list(kknn\_mape$k), sd)  
mape\_data <- cbind(mape\_mean, mape\_sd[,2]) %>%  
 mutate(LB = x - mape\_sd[,2], UB = x + mape\_sd[,2])  
colnames(mape\_data) <- c("k", "MAPE", "SD", "LB", "UB")  
  
# Visualize the aggregate data  
ggplot(data = mape\_data, aes(x = k, y = MAPE)) +  
 geom\_line() +  
 geom\_ribbon(aes(ymin = LB, ymax = UB), alpha = 0.2) +  
 labs(title = "KKNN: k distribution",  
 x = "k",  
 y = "MAPE")



# Make Predictions

Now that we have our model, we can go ahead and make predictions! We need to apply all the same methods to the prediction data as we did to our modeling data.

library(plyr)  
library(caret)  
library(kknn)  
  
# Read in the data  
predict\_df <- read.csv("https://raw.githubusercontent.com/dmoste/DATA624/main/Project2/StudentEvaluationTOPREDICT.csv")  
  
# Check for missing data  
missing\_data <- sapply(predict\_df, function(x) sum(is.na(x)))  
missing\_data <- data.frame(missing\_data)  
  
# Remove PH and Hyd.Pressure1 from the features data  
drops <- c("PH", "Hyd.Pressure1")  
predict\_features <- predict\_df[,!(names(predict\_df) %in% drops)]  
  
# Map Brand.Code values to numerical options  
predict\_features[,1] <- mapvalues(predict\_features[,1],  
 from = c("A","B","C","D",""),  
 to = c(1,2,3,4,NA))  
predict\_features[,1] <- as.integer(predict\_features[,1])  
  
# Replace missing values with the mean of the predictor  
na\_to\_mean <- function(x) replace(x, is.na(x), mean(x, na.rm = TRUE))  
predict\_features[] <- lapply(predict\_features, na\_to\_mean)  
  
# Apply BoxCox transformations, center the data, and scale it  
trans <- preProcess(predict\_features,  
 method = c("BoxCox", "center", "scale"))  
transformed\_feat <- predict(trans, predict\_features)  
  
# Recombine the PH response with the transformed features  
predict\_df <- cbind(predict\_df[,26], transformed\_feat)  
names(predict\_df)[1] <- ("PH")  
  
# Train and predict using the model decided from the modeling data  
kknn\_fit <- kknn(train[,1] ~ .,  
 train[,-1],  
 predict\_df[,-1],  
 k = 18,  
 distance = 1,  
 kernel = "triweight")  
  
predictions <- data.frame(kknn\_fit$fitted.values)

## 

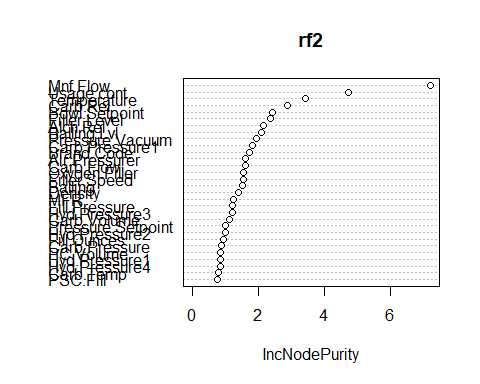
## Other models

The following section includes other models. Note that only the fitted model and model metrics code are shown.

### Random Forest

Random Forest model is an ensemble tree-based learning algorithm that averages the prediction over many individual trees. The algorithm uses bootstrap aggregation, or bagging to reduce over fitting and improve accuracy.

# Random Forest model  
rf2 <- randomForest(PH ~ ., data = train2, ntrees = 500)  
varImpPlot(rf2)



rf2

##   
## Call:  
## randomForest(formula = PH ~ ., data = train2, ntrees = 500)   
## Type of random forest: regression  
## Number of trees: 500  
## No. of variables tried at each split: 10  
##   
## Mean of squared residuals: 0.01039709  
## % Var explained: 64.85

test\_rf2 <- predict(rf2, test2)  
  
caret\_test\_rf2 <- data.frame(cbind(test\_rf2,test2[,1]))  
colnames(caret\_test\_rf2) <- c("caret","actual")  
caret\_test\_rf2 <- caret\_test\_rf2 %>%  
 mutate(pe = abs(actual - caret)/actual)  
  
MAPE2 <- (mean(caret\_test\_rf2$pe))\*100  
MAPE2

## [1] 0.9259718

The model with the transformed predictor variables and median imputation produced the same MAPE value as the baseline model. This makes some sense since the random forest algorithm is based on partitioning of the data by certain variable values.

### Ordinary Least Squares

#### Check model assumptions

One important assumption for ordinary least squares models is that variables are uncorrelated with each other. To check this assumption, we search for highly correlated variables. While removing these variables doesn’t guarantee an absence of multicollinearity, it is a useful first step, and can reduce the total number of variables in the model.

Groups of variables that are highly correlated:  
Alch Rel Density, Balling, Carb Rel, Balling Lvl;  
Air Pressurer Carb Rel, Balling Lvl;  
Bowl Setpoint, Filler Level;  
Filler Speed, MFR;  
Hyd Pressure2, Hyd Pressure3;  
Carb Temp, Carb Pressure.

To avoid collinearity in a linear model, we eliminate some of the most highly correlated variables:  
Keep Balling and drop Balling Lvl and Density;  
Keep Alch Rel and drop Carb Rel;  
Keep Bowl Setpoint and drop Filler Level;  
Keep MFR and drop Filler Speed;  
Keep Hyd Pressure2 and drop Hyd Pressure3;  
Keep Carb Pressure and drop Carb Temp.

There are still some large correlations remaining, for example, between Balling and Alch Rel. But because I know Balling is a measure of sugar content, and I expect that sugar content is related to pH, I’m going to keep it for now. Mnf Flow is also correlated to many other variables. It may drop out of a linear model later on.

Another assumption for fitting the OLS linear model are that predictors are normally distributed. By applying a Box-Cox transformation to the data in the **Transform Data** section above, we make sure the data roughly conforms to this assumption.

After fitting a model, we’ll check the final assumption– that residuals have mean zero with approximately constant variance.

#### Fit model

#OLS  
data\_ctrl <- trainControl(method = 'cv', number = 10)  
train1.lm <- train(train\_preds2.df, train\_resp.df,  
 method = "lm")  
summary(train1.lm)

##   
## Call:  
## lm(formula = .outcome ~ ., data = dat)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -0.53016 -0.08102 0.00998 0.08665 0.54546   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 8.537855 0.014183 601.985 < 2e-16 \*\*\*  
## Carb.Volume -0.008618 0.005278 -1.633 0.10266   
## Fill.Ounces -0.005558 0.003215 -1.729 0.08395 .   
## PC.Volume -0.009437 0.003594 -2.626 0.00870 \*\*   
## Carb.Pressure 0.003561 0.003362 1.059 0.28960   
## PSC -0.001866 0.003166 -0.589 0.55560   
## PSC.Fill -0.003453 0.003097 -1.115 0.26496   
## PSC.CO2 -0.001965 0.003084 -0.637 0.52402   
## Mnf.Flow -0.074255 0.005727 -12.966 < 2e-16 \*\*\*  
## Carb.Pressure1 0.033806 0.003639 9.290 < 2e-16 \*\*\*  
## Fill.Pressure 0.014237 0.004481 3.177 0.00151 \*\*   
## Hyd.Pressure2 0.019701 0.004519 4.360 1.37e-05 \*\*\*  
## Hyd.Pressure4 -0.003333 0.004386 -0.760 0.44739   
## Temperature -0.018775 0.003405 -5.514 3.96e-08 \*\*\*  
## Usage.cont -0.023108 0.003872 -5.968 2.84e-09 \*\*\*  
## Carb.Flow 0.008902 0.003882 2.293 0.02194 \*   
## MFR -0.001151 0.003321 -0.346 0.72903   
## Balling -0.055721 0.010163 -5.483 4.72e-08 \*\*\*  
## Pressure.Vacuum -0.011571 0.004176 -2.771 0.00564 \*\*   
## Oxygen.Filler -0.017482 0.003619 -4.830 1.47e-06 \*\*\*  
## Bowl.Setpoint 0.039836 0.004363 9.131 < 2e-16 \*\*\*  
## Pressure.Setpoint -0.017455 0.004449 -3.924 9.01e-05 \*\*\*  
## Air.Pressurer 0.001208 0.003281 0.368 0.71285   
## Alch.Rel 0.051913 0.011109 4.673 3.16e-06 \*\*\*  
## Brand.CodeB 0.027754 0.021427 1.295 0.19538   
## Brand.CodeC -0.086064 0.021563 -3.991 6.80e-05 \*\*\*  
## Brand.CodeD 0.045551 0.016202 2.811 0.00498 \*\*   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 0.1345 on 2026 degrees of freedom  
## Multiple R-squared: 0.4082, Adjusted R-squared: 0.4006   
## F-statistic: 53.75 on 26 and 2026 DF, p-value: < 2.2e-16

train1\_MAPE <- 100 \* (sum(abs(train1.lm$finalModel$residuals) / train\_resp.df)) / length(train\_resp.df)

This initial model has MAPE = 1.217 and RMSE 0.1345. We refine the model by removing variables that have low explanatory power (p > 0.05).

train2.lm <- train(train\_preds2.df[,c(3, 8:11, 13:15, 17:21, 23, 24)], train\_resp.df, method = "lm")  
summary(train2.lm)

##   
## Call:  
## lm(formula = .outcome ~ ., data = dat)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -0.54117 -0.07958 0.01146 0.08834 0.53450   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 8.534004 0.013464 633.842 < 2e-16 \*\*\*  
## PC.Volume -0.008986 0.003451 -2.604 0.00928 \*\*   
## Mnf.Flow -0.074395 0.005674 -13.113 < 2e-16 \*\*\*  
## Carb.Pressure1 0.034093 0.003618 9.423 < 2e-16 \*\*\*  
## Fill.Pressure 0.013764 0.004317 3.188 0.00145 \*\*   
## Hyd.Pressure2 0.021949 0.004381 5.010 5.91e-07 \*\*\*  
## Temperature -0.018806 0.003368 -5.585 2.66e-08 \*\*\*  
## Usage.cont -0.024078 0.003834 -6.280 4.13e-10 \*\*\*  
## Carb.Flow 0.010432 0.003724 2.802 0.00513 \*\*   
## Balling -0.058074 0.009799 -5.927 3.63e-09 \*\*\*  
## Pressure.Vacuum -0.011302 0.004048 -2.792 0.00528 \*\*   
## Oxygen.Filler -0.017254 0.003595 -4.800 1.70e-06 \*\*\*  
## Bowl.Setpoint 0.039960 0.004286 9.323 < 2e-16 \*\*\*  
## Pressure.Setpoint -0.017864 0.004389 -4.070 4.88e-05 \*\*\*  
## Alch.Rel 0.054416 0.010992 4.951 8.00e-07 \*\*\*  
## Brand.CodeB 0.033793 0.020510 1.648 0.09959 .   
## Brand.CodeC -0.079357 0.020291 -3.911 9.49e-05 \*\*\*  
## Brand.CodeD 0.044924 0.014818 3.032 0.00246 \*\*   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 0.1345 on 2035 degrees of freedom  
## Multiple R-squared: 0.405, Adjusted R-squared: 0.4001   
## F-statistic: 81.49 on 17 and 2035 DF, p-value: < 2.2e-16

train2\_MAPE <- 100 \* (sum(abs(train2.lm$finalModel$residuals) / train\_resp.df)) / length(train\_resp.df)

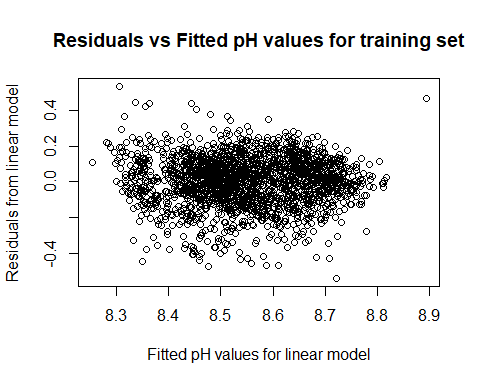
The refined model has MAPE = 1.219 and RMSE = 0.1345.

After dropping variables with high p-values, the simpler model retains almost all its explanatory power.

The plot of fitted vs actual values for the training data shows a clear positive linear relationship, although variability is large. The large variability corresponds to the relatively low value of .

#### Examine residuals

#Predicted vs residual  
plot(x = train2.lm$finalModel$fitted.values, y = train2.lm$finalModel$residuals,  
 xlab = "Fitted pH values for linear model",  
 ylab = "Residuals from linear model",  
 main = "Residuals vs Fitted pH values for training set")



The residuals appear to be randomly distributed with roughly constant variability about a mean of zero.

#### Compute model metrics

For the refined model, MAPE = 1.217, and RMSE = 0.1345.

### Partial Least Squares

Because there is large possibility of multicollinearity in linear models involving this data, feature selection is an important part of the modeling process. Partial least squares performs a kind of feature selection, because it generates new uncorrelated predictors based on "underlying… relationships among the predictors which are highly correlated with the response (KJ 114).

The optimal number of components for the PLS model is 13.

print("RMSE:")

## [1] "RMSE:"

print(train.pls$results$RMSE[13])

## [1] 0.133904

print("R^2:")

## [1] "R^2:"

print(train.pls$results$Rsquared[13])

## [1] 0.4069861

PLS\_resid <- train.pls$finalModel$residuals[, 1, 13]  
train\_PLS\_MAPE <- (100 / length(train\_resp.df)) \* sum(abs(PLS\_resid /train\_resp.df))  
  
print("MAPE:")

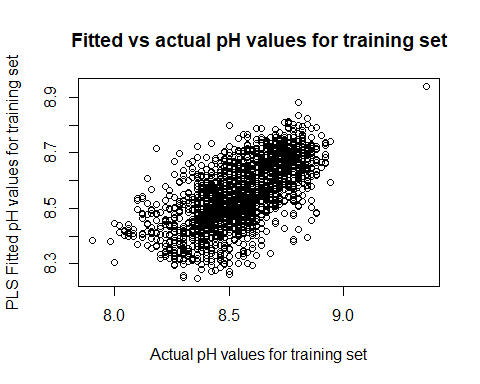
## [1] "MAPE:"

print(train\_PLS\_MAPE)

## [1] 1.191362

Examining the actual and predicted values of the response variable:

train.pls\_predicted <- predict(train.pls, train\_preds.df)  
  
plot(x = train\_resp.df, y = train.pls\_predicted,  
 xlab = "Actual pH values for training set",  
 ylab = "PLS Fitted pH values for training set",  
 main = "Fitted vs actual pH values for training set")



The residuals appears to be randomly distributed with roughly constant variability about a mean of zero.

### Model: Elastic Net

An elastic net model is another linear modeling method that performs feature selection and is robust to multicollinearity. The elastic net model combines a ridge penalty with a lasso penalty on model coefficients to improve the overall stability of the model. Elastic net models tolerate some increase in coefficient bias in order to reduce variance. Here, we search a range of lasso and ridge parameters to determine an optimal model.

The optimal model occurs with and fraction = 1. This is equivalent to a pure lasso model. Including a ridge penalty did not improve model performance.

Minimum RMSE is 0.1387272.

For the optimal elastic net model, .

The relationship between actual and fitted values is linear with positive slope. It demonstrates constant but large variability, consistent with a relatively low value of .

#MAPE for Elastic Net:  
enet\_MAPE <- (100 / length(train\_resp.df)) \* sum(abs((train\_resp.df - enet\_predicted) / train\_resp.df))  
  
print("MAPE for elastic net:")

## [1] "MAPE for elastic net:"

print(enet\_MAPE)

## [1] 1.235704

The residuals appear to be randomly distributed about a mean of zero. Variability appears to be largest near fitted values around 8.6.

#### Test best model, PLS

The model with the best performance among those examined here is PLS. Here we test the performance of the PLS model on the holdout data.

testset\_predicted <- predict(train.pls, test\_preds.df)  
PLS\_test <- data.frame(cbind(test\_resp.df, testset\_predicted))  
PLS\_test <- PLS\_test %>%  
 mutate("diff" = testset\_predicted - test\_resp.df)  
  
PLS\_test <- PLS\_test %>%  
 mutate("sq\_diff" = diff^2)  
  
PLS\_test\_RMSE <- sqrt(sum(PLS\_test$sq\_diff) / nrow(PLS\_test))  
  
print("RMSE:")

## [1] "RMSE:"

print(PLS\_test\_RMSE)

## [1] 0.1326695

The PLS model performs similarly on the holdout data as it did on the test set. This provides evidence that the model has not been over-fitted to the training data.

#### Predict

Finally, we use the PLS model to predict pH values for new samples.

Because many of the variables in this data set exhibit high correlation, using this data with linear models risks violating the assumption of no multicollinearity. One way to deal with the risk of multicollinearity is to employ a model that performs feature selection, such as partial least squares. Partial least squares performs a kind of feature selection, because it generates new uncorrelated predictors based on "underlying… relationships among the predictors which are highly correlated with the response (Kuhn and Johnson 114). The other modeling assumptions are addressed by transforming the data as described in the corresponding section for OLS.

#### Fit model

summary(train.pls)

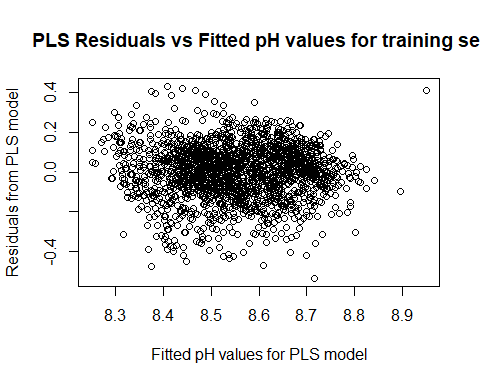
## Data: X dimension: 2053 33   
## Y dimension: 2053 1  
## Fit method: oscorespls  
## Number of components considered: 15  
## TRAINING: % variance explained  
## 1 comps 2 comps 3 comps 4 comps 5 comps 6 comps 7 comps  
## X 17.35 24.34 40.17 45.84 49.45 52.74 55.67  
## .outcome 22.65 32.93 35.97 38.49 40.38 41.33 41.81  
## 8 comps 9 comps 10 comps 11 comps 12 comps 13 comps 14 comps  
## X 58.76 61.16 63.76 67.22 70.06 72.26 74.57  
## .outcome 42.10 42.32 42.45 42.53 42.61 42.69 42.73  
## 15 comps  
## X 77.88  
## .outcome 42.75

Examining the actual and predicted values of the response variable:

There is a positive linear relationship among the actual and fitted values for the PLS model. This suggests a linear model like PLS is appropriate for this data set.

#### Examine residuals

#Residuals  
plot(x = train.pls\_predicted, y = train\_resp.df - train.pls\_predicted,  
 xlab = "Fitted pH values for PLS model",  
 ylab = "Residuals from PLS model",  
 main = "PLS Residuals vs Fitted pH values for training set")



The residuals appears to be randomly distributed with roughly constant variability about a mean of zero.

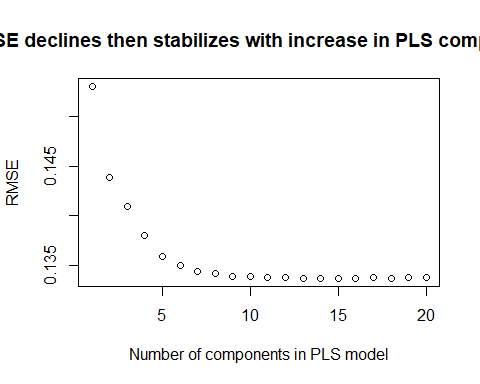
#### Compute model metrics

What is the optimal number of components for the PLS model?

summary(train.pls)

## Data: X dimension: 2053 33   
## Y dimension: 2053 1  
## Fit method: oscorespls  
## Number of components considered: 15  
## TRAINING: % variance explained  
## 1 comps 2 comps 3 comps 4 comps 5 comps 6 comps 7 comps  
## X 17.35 24.34 40.17 45.84 49.45 52.74 55.67  
## .outcome 22.65 32.93 35.97 38.49 40.38 41.33 41.81  
## 8 comps 9 comps 10 comps 11 comps 12 comps 13 comps 14 comps  
## X 58.76 61.16 63.76 67.22 70.06 72.26 74.57  
## .outcome 42.10 42.32 42.45 42.53 42.61 42.69 42.73  
## 15 comps  
## X 77.88  
## .outcome 42.75

plot(x = train.pls$results$ncomp, y = train.pls$results$RMSE,  
 xlab = "Number of components in PLS model",  
 ylab = "RMSE",  
 main = "RMSE declines then stabilizes with increase in PLS components")



The optimal number of components for the PLS model is 13.

MAPE for the PLS model is 1.19, and RMSE = 0.134.

### SVM

If our target, PH is particularly skewed, it could lead to biased predictions.

PH is normally distributed with possible outliers on the low and high ends. Given the normal shape, a regression or possible ensemble with regression and classification seems more appropriate.

### Missing Data

Here we review the patterns of missingness across predictor features.

Notice that approximately 8.25 percent of the rows are missing a value for MFR. This will be dropped to avoid the potential negative consequences of imputation. Additionally, the categorical feature Brand.Code is missing approximately 4.67 percent of its values and will create a new feature category ‘Unknown’ consisting of missing values.

We visualize the distributions of each of the predictor features. This will help us assess relationships between features and with PH, and identify outliers as well as transformations that might improve model resolution.

The distribution profiles show the prevalence of kurtosis, specifically right skew in differenct variables.

### Boxplots

Lets use boxplots to understand the spread of each feature.

The boxplots reveal outliers, though none of them seem extreme enough to warrant imputing or removal.

### Feature-Target Correlations

We next quantify the relationships visualized above. In general, our model should focus on features showing stronger positive or negative correlations with PH. Features with correlations closer to zero will probably not provide any meaningful information on pH levels.

It appears that Bowl.Setpoint, Filler.Level, Carb.Flow, Pressure.Vacuum, Carb.Rel, Alch.Rel and Oxygen.Filler have the highest correlations (positive) with PH, while Mnf.Flow, Usage.cont, Fill.Pressure, Pressure.Setpoint, Hyd.Pressure3, and Hyd.Pressure2 have the strongest negative correlations with PH. All others have a weak or slightly negative correlation, which implies they have less predictive power.

#### Multicollinearity

Lets check for correlation between predictive features, or multicollinearity.

We can see that some variables are highly correlated with one another with a correlation between 0.75 and 1. During our modeling is possible to avoid including pairs with strong correlations.

#### Near-Zero Variance

Before we move to data preparation, lets check for any features that show near zero-variance. This will be eliminated since they add little predictive information.

Hyd.Pressure1 will be dropped.

### Data Preparation

#### Eliminated Fields/Variables

MFR has lots of missing values that exceed 8%. Hyd.Pressure1 shows near zero variance.

#### Drop Missing Values

We need to drop 4 PHrows with missing values Replace missing values for Brand.Code with “Unknown” Impute remaining missing values using kNN()

#### Convert Categorical to Dummy

Brand.Code is a categorical variable with values A, B, C, D and Unknown. We convert it to a set of dummy columns for modeling.

#### Transform features with skewed distributions

Lets apply the Box-Cox transformation to the skewed features using preProcess from caret to ensure we are using distributions that better approximate normal.

#### Build Models

First, we split our cleaned dataset into training and testing sets (80% training, 20% testing). This split is necessary because the provided evaluation data set does not provide PH values.

#### Model 1 - Support Vector Machine (SVM)

Support Vector Machine (SVM) is a supervised machine learning algorithm which is mainly used to classify data into different classes. Unlike most algorithms, SVM makes use of a hyperplane which acts like a decision boundary between the various classes. SVM can be used to generate multiple separating hyperplanes such that the data is divided into segments and each segment contains only one kind of data.

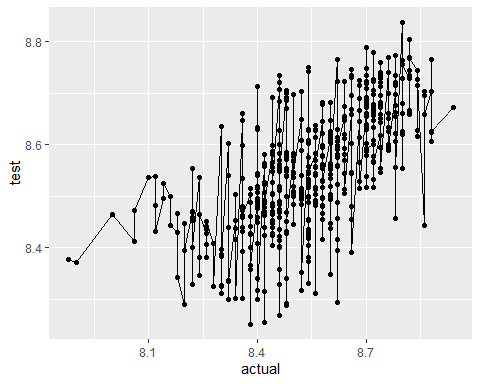
cl <- makePSOCKcluster(5)  
registerDoParallel(cl)  
set.seed(100)  
  
trctrl <- trainControl(method = "repeatedcv", number = 10, repeats = 3)  
  
svm\_Linear <- train(x = X.train, y = y.train, method = "svmLinear",  
 trControl=trctrl,  
 preProcess = c("center", "scale"),  
 tuneLength = 10)  
  
stopCluster(cl)  
  
svm\_Linear$results #summary(svm\_Linear)

## C RMSE Rsquared MAE RMSESD RsquaredSD MAESD  
## 1 1 0.1349054 0.3979139 0.1030911 0.006811931 0.04551836 0.00474587

# Applying Model 1 against our Test Data:  
svm\_pred <- predict(svm\_Linear, newdata = X.test)  
test <- data.frame(cbind(svm\_pred,y.test))  
colnames(test) <- c("test","actual")  
test <- test %>%  
 mutate(pe = abs(actual - test)/actual)  
  
MAPE <- (mean(test$pe))\*100  
MAPE

## [1] 1.172121

ggplot(test, aes(x = actual, y = test)) +  
 geom\_line() +  
 geom\_point()



# Bind results to a table to compare performance of our two models  
results <- data.frame()  
results <- data.frame(t(postResample(pred = svm\_pred, obs = y.test))) %>% mutate(Model = "Support Vector Machine (SVM)") %>% rbind(results)

#### Model 2 - Multivariate Adaptive Regression Splines (MARS)

The approach used for the second model, Multivariate Adaptive Regression Splines (MARS), creates contrasting versions of each predictor to enter the model. These versions, features known as hinge functions, each represent an exclusive portion of the data. Such features are created iteratively for all model predictors, a process that is followed by “pruning” of individual features that do not contribute to the model.

### Model Summary

We evaluate our two models using three criteria: root mean squared error (RMSE), R-squared, and mean absolute error. The table below lists these criteria for each model.

results %>% dplyr::select(Model, RMSE, Rsquared, MAE)

## Model RMSE Rsquared  
## 1 Multivariate Adaptive Regression Splines (MARS) 0.1275186 0.4523130  
## 2 Support Vector Machine (SVM) 0.1328184 0.4077656  
## MAE  
## 1 0.09672048  
## 2 0.09949331

## Model Selection

Based on evaluating both RMSE and , MARS slightly outperformed SVM. MARS also has a better MAPE at 1.14.

## Predictions

We apply **Model #2 (MARS)** to the holdout evaluation set to predict the targets. We have saved these predictions as csv in the file eval\_predicted.csv.

### Neural Network Analysis

Neural networks are non-linear regression techniques inspired by the brain. A neural network takes input based on existing data and uses backpropagation to optimize the weights of input variables to improve the model. The output predictions are based on the data from the input and hidden layers.

Neural networks are powerful and work well as deep learning algorithms. They work well with large datasets and produce results with high accuracy. However, they are hard to interpret, require large amount of data and can be time consuming to build as they need a lot of customizations.

#### Data Processing

The data is processed by replacing missing values with mean and converting categorical variable to numeric. Visualization of distributions of predictor variables is generated.

### Visualize distribution of predictive variables

The data exhibit a mixture of distribution patterns. Fill. Ounces, PC.Volume, and Carb.Temp are normally distributed. PSC and PSC.Fill are skewed; and other variables have bimodal distributions.

#### Transformation

It’s important to normalize the data before training for Neural Network analysis. A standard approach is to scale the inputs to have mean 0 and a variance. Also linear decorrelation/whitening/pca helps a lot.

#### Building a Neural Networks model

library(mlbench)  
  
# Rank variables by importance  
importance <- varImp(nnetFit, scale = FALSE)  
print(importance)  
plot(importance)

varImp() shows that Brand.Code, Hyd.Pressure2, Carb.Flow, Bowl.Setpoint and Temperature are the top five attributes and PSC is the least important attribute.

# Model evaluation  
RMSE(test$PH, nnetPred)  
mean(abs((test$PH-nnetPred)/test$PH)) \* 10

library(MLmetrics)  
MAPE(nnetPred, test$PH)

varImp() shows that Brand.Code, Hyd.Pressure2, Carb.Flow, Bowl.Setpoint and Temperature are the top five attributes and PSC is the least important attribute.

#### Transform and process predict data