# SurvMeth 895 Final Project

**Weining Xu**

**11/21/2021**

## Abstract

The water quality was always a problem in worldwide. Michigan state was suffered from water crisis over years in the city, Flint. With inadequate treatment and testing of the water resulted in a series of major water quality and health issues for Flint residents—issues that were chronically ignored, overlooked, and discounted by government officials even as complaints mounted that the foul-smelling, discolored, and off-tasting water piped into Flint homes for 18 months was causing skin rashes, hair loss, and itchy skin. Water filter seems like a must-have appliance at home. Access to safe drinking-water is essential to health, a basic human right and a component of effective policy for health protection. This is important as a health and development issue at a national, regional and local level. In some regions, it has been shown that investments in water supply and sanitation can yield a net economic benefit, since the reductions in adverse health effects and health care costs outweigh the costs of undertaking the interventions. This inspired me to study on the water quality issues, and use machine learning to train different classification models and predict the water is safe to drink or not based on the water quality dataset from Kaggle. Although this is a set of data created from imaginary data of water quality in an urban environment, we can still apply the models to the real-life situation with adjustments and make predictions on the water quality afterwards.

## Set Up

```
library(dplyr)
```

```
##
## 载入程辑包：'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(caret)
```

```
## 载入需要的程辑包：lattice
```

```
## 载入需要的程辑包：ggplot2
```

```
library(e1071)
library(rpart)
library(fastAdaboost)
library(gbm)
```

```
## Loaded gbm 2.1.8
```

```
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## 载入程辑包：'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

# Data Preparation

Import data waterQuality1.csv

```
library(readr)
data <- read_csv("~/Downloads/UMICH/Survmeth895/Final Project/waterQuality1.csv")
```

```
## Rows: 7999 Columns: 21
```

```
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl (21): aluminium, ammonia, arsenic, barium, cadmium, chloramine, chromium...
```

```
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
head(data)
```

| aluminium <dbl> | amm… <dbl> | arsenic <dbl> | bari… <dbl> | cadm… <dbl> | chloramine <dbl> | chromi… <dbl> | cop… <dbl> | flouride <dbl> | bacteria <dbl> |
|---|---|---|---|---|---|---|---|---|---|
| 1.65 | 9.08 | 0.04 | 2.85 | 0.007 | 0.35 | 0.83 | 0.17 | 0.05 | 0.20 |
| 2.32 | 21.16 | 0.01 | 3.31 | 0.002 | 5.28 | 0.68 | 0.66 | 0.90 | 0.65 |
| 1.01 | 14.02 | 0.04 | 0.58 | 0.008 | 4.24 | 0.53 | 0.02 | 0.99 | 0.05 |
| 1.36 | 11.33 | 0.04 | 2.96 | 0.001 | 7.23 | 0.03 | 1.66 | 1.08 | 0.71 |
| 0.92 | 24.33 | 0.03 | 0.20 | 0.006 | 2.67 | 0.69 | 0.57 | 0.61 | 0.13 |

| aluminium <dbl> | amm… <dbl> | arsenic <dbl> | bari… <dbl> | cadm… <dbl> | chloramine <dbl> | chromi… <dbl> | cop… <dbl> | flouride <dbl> | bacteria <dbl> |
|---|---|---|---|---|---|---|---|---|---|
| 0.94 | 14.47 | 0.03 | 2.88 | 0.003 | 0.80 | 0.43 | 1.38 | 0.11 | 0.67 |

6 rows | 1-10 of 21 columns

check data dimension, type and missing values

```
dim(data)
```

```
## [1] 7999   21
```

```
glimpse(data)
```

```
## Rows: 7,999
## Columns: 21
## $ aluminium   <dbl> 1.65, 2.32, 1.01, 1.36, 0.92, 0.94, 2.36, 3.93, 0.60, 0.22…
## $ ammonia     <dbl> 9.08, 21.16, 14.02, 11.33, 24.33, 14.47, 5.60, 19.87, 24.5…
## $ arsenic     <dbl> 0.040, 0.010, 0.040, 0.040, 0.030, 0.030, 0.010, 0.040, 0.…
## $ barium      <dbl> 2.85, 3.31, 0.58, 2.96, 0.20, 2.88, 1.35, 0.66, 0.71, 1.37…
## $ cadmium     <dbl> 0.007, 0.002, 0.008, 0.001, 0.006, 0.003, 0.004, 0.001, 0.…
## $ chloramine  <dbl> 0.35, 5.28, 4.24, 7.23, 2.67, 0.80, 1.28, 6.22, 3.14, 6.40…
## $ chromium    <dbl> 0.83, 0.68, 0.53, 0.03, 0.69, 0.43, 0.62, 0.10, 0.77, 0.49…
## $ copper      <dbl> 0.17, 0.66, 0.02, 1.66, 0.57, 1.38, 1.88, 1.86, 1.45, 0.82…
## $ flouride    <dbl> 0.05, 0.90, 0.99, 1.08, 0.61, 0.11, 0.33, 0.86, 0.98, 1.24…
## $ bacteria    <dbl> 0.20, 0.65, 0.05, 0.71, 0.13, 0.67, 0.13, 0.16, 0.35, 0.83…
## $ viruses     <dbl> 0.000, 0.650, 0.003, 0.710, 0.001, 0.670, 0.007, 0.005, 0.…
## $ lead        <dbl> 0.054, 0.100, 0.078, 0.016, 0.117, 0.135, 0.021, 0.197, 0.…
## $ nitrates    <dbl> 16.08, 2.01, 14.16, 1.41, 6.74, 9.75, 18.60, 13.65, 14.66,…
## $ nitrites    <dbl> 1.13, 1.93, 1.11, 1.29, 1.11, 1.89, 1.78, 1.81, 1.84, 1.46…
## $ mercury     <dbl> 0.007, 0.003, 0.006, 0.004, 0.003, 0.006, 0.007, 0.001, 0.…
## $ perchlorate <dbl> 37.75, 32.26, 50.28, 9.12, 16.90, 27.17, 45.34, 53.35, 23.…
## $ radium      <dbl> 6.78, 3.21, 7.07, 1.72, 2.41, 5.42, 2.84, 7.24, 4.99, 0.08…
## $ selenium    <dbl> 0.08, 0.08, 0.07, 0.02, 0.02, 0.08, 0.10, 0.08, 0.08, 0.03…
## $ silver      <dbl> 0.34, 0.27, 0.44, 0.45, 0.06, 0.19, 0.24, 0.08, 0.25, 0.31…
## $ uranium     <dbl> 0.02, 0.05, 0.01, 0.05, 0.02, 0.02, 0.08, 0.07, 0.08, 0.01…
## $ is_safe     <dbl> 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1…
```

```
# 6 missing element, drop NAs
sum(is.na(data))
```

```
## Warning: One or more parsing issues, see `problems()` for details
```

```
## [1] 6
```

```
clean <- na.omit(data)
dim(clean)
```

```
## [1] 7996   21
```

```
table(clean$is_safe)
```

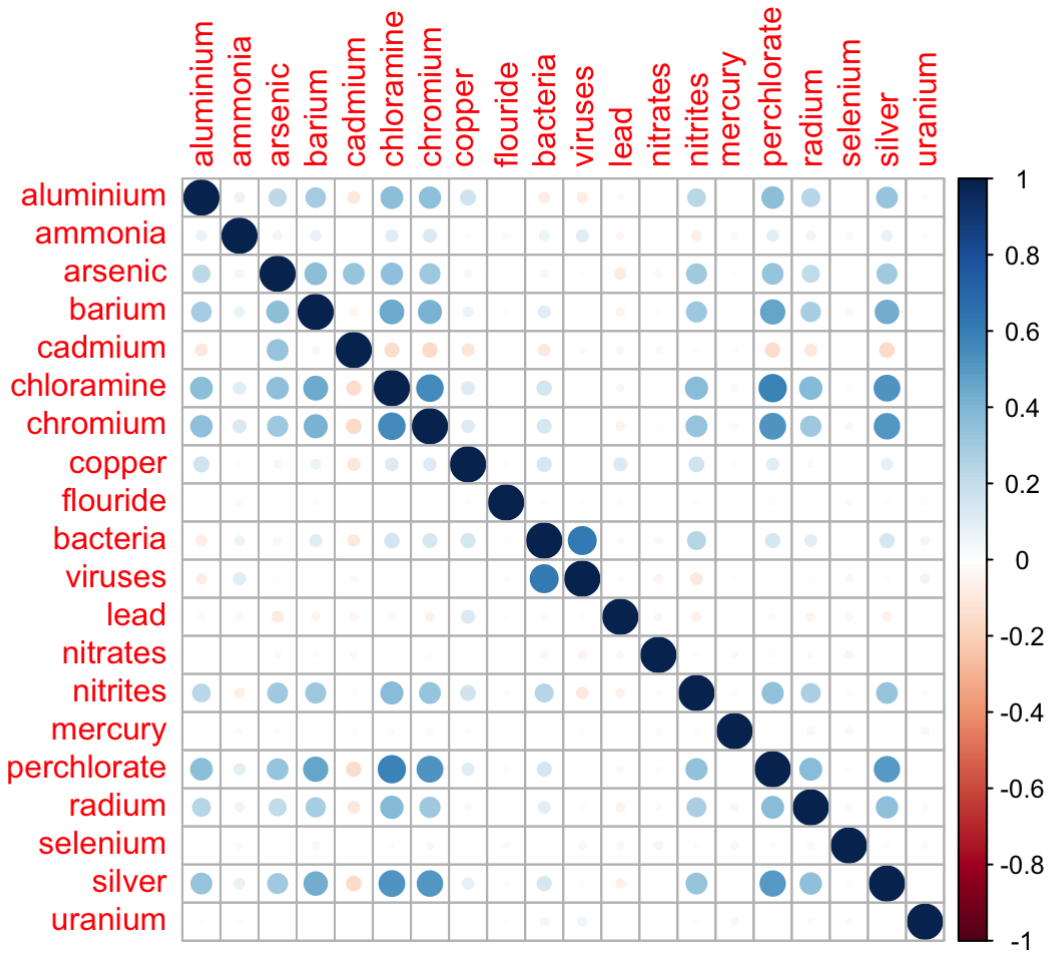
```
##
##    0    1
## 7084  912
```

# Data Glimpse

Let get a brief overview of each attributes and outcome:

Aluminium - dangerous if greater than 2.8 Ammonia - dangerous if greater than 32.5 Arsenic - dangerous if greater than 0.01 Barium - dangerous if greater than 2 Cadmium - dangerous if greater than 0.005 Chloramine - dangerous if greater than 4 Chromium - dangerous if greater than 0.1 Copper - dangerous if greater than 1.3 Flouride - dangerous if greater than 1.5 Bacteria - dangerous if greater than 0 Viruses - dangerous if greater than 0 Lead - dangerous if greater than 0.015 Nitrates - dangerous if greater than 10 Nitrites - dangerous if greater than 1 Mercury - dangerous if greater than 0.002 Perchlorate - dangerous if greater than 56 Radium - dangerous if greater than 5 Selenium - dangerous if greater than 0.5 Silver - dangerous if greater than 0.1 Uranium - dangerous if greater than 0.3 is_safe - class attribute {0 - not safe, 1 - safe} The numbers are all in unit of level in water per liter.

```
corrplot::corrplot(cor(clean[, 1:ncol(clean)-1]))
```




```
clean$is_safe <- as.factor(clean$is_safe)
```

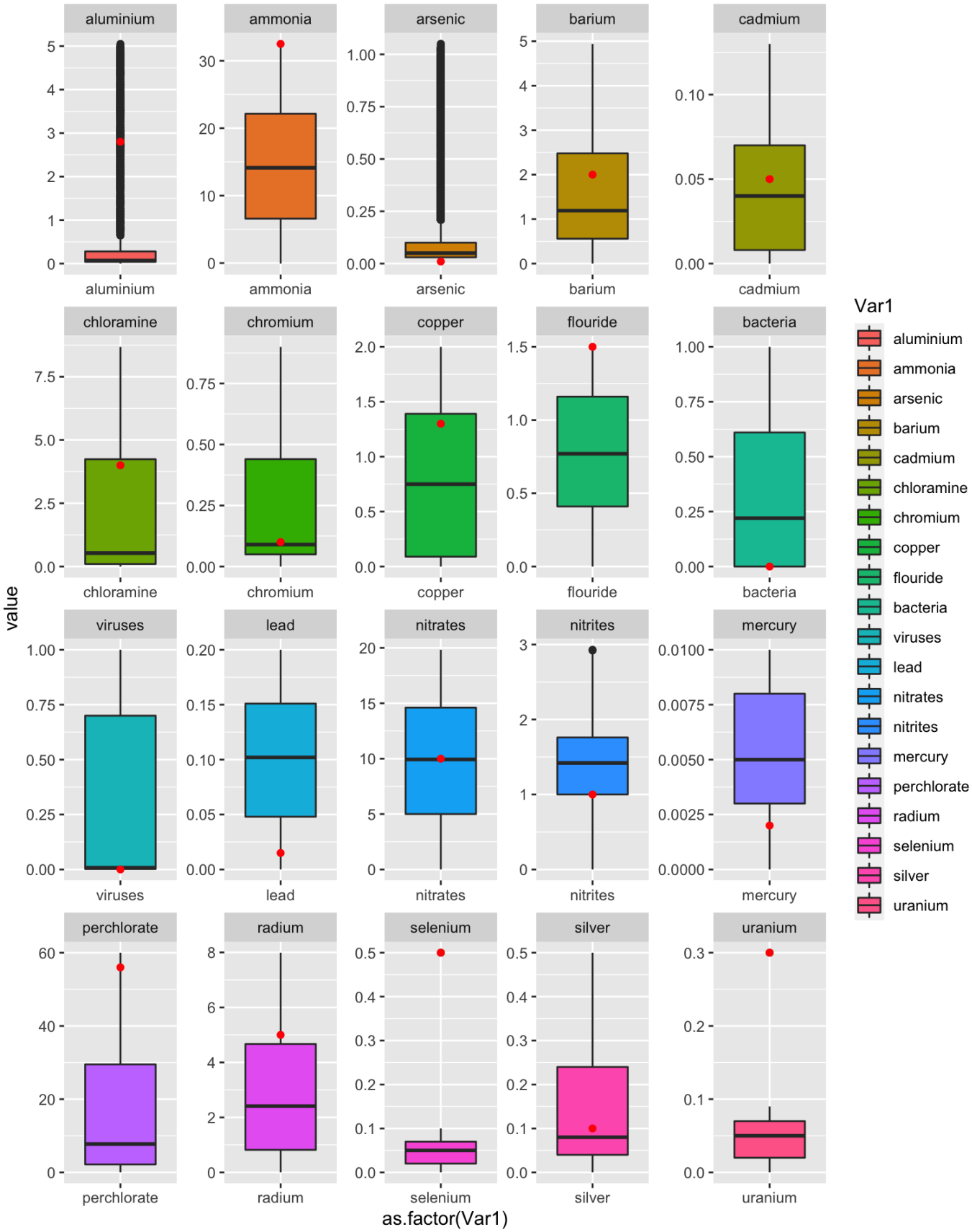
From the correlation plot, we can see how each attributes are related among others. Only chloramine - chromium - silver - perchlorate and bacteria - viruses are highly correlated with each other, other attributes are more likely to be independent from others.

Then we plot a boxplot with all the observations, and have a red point marked as the dangerous margin to see the distribution of the data in each attributes.

```
# put standard scale in the first row
standard <- rbind(c(2.8, 32.5, 0.01, 2, 0.05, 4, 0.1, 1.3, 1.5, 0,0,0.015, 10,1,0.002
,56,5,0.5,0.1,0.3),
                  clean[, 1:ncol(clean)-1])

library(reshape2)

ggplot(data = melt(t(standard)), aes(x=as.factor(Var1), y=value)) +
  geom_boxplot(aes(fill=Var1)) +
  geom_point(data = melt(t(standard[1,])), aes(x=as.factor(Var1), y=value), color =
"red")+
  facet_wrap(~Var1, scales = "free")
```




From observing the boxplot, we can see that some attributes (arsenic, bacteria, viruses) data points are above the standard scale, which would affect the evaluation for water quality. Even though some attributes remain dangerous beyond the standard line, there are still 912 observations are labeled safe. Therefore, in the following models, we want to apply Decision Trees, Bagging, Random Forest, AdaBoost, Gradient Boost, and Logistic Regression to train and predict the data, and see how accurate each model performs.

# Train and test split

We first split the data into train and test set, using seed=123. Factorize and set levels for “is_safe” attributes to not safe and safe.

```
set.seed(123)

clean$is_safe <- as.factor(clean$is_safe)
train <- sample(1:nrow(clean), 0.8*nrow(clean))
c_train <- clean[train,]
c_test <- clean[-train,]
levels(c_train$is_safe) <-c("not safe", "safe")
```

# Decision Tree

First, we apply Decision Trees to train the model.

```
tree <- rpart(make.names(is_safe) ~ ., data = c_train, method = "class")
tree
```

```
## n= 6396
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 6396 729 not.safe (0.88602251 0.11397749)
##    2) cadmium>=0.0095 4601 163 not.safe (0.96457292 0.03542708)
##      4) silver>=0.105 1257   0 not.safe (1.00000000 0.00000000) *
##      5) silver< 0.105 3344 163 not.safe (0.95125598 0.04874402)
##       10) chloramine< 0.695 3025  92 not.safe (0.96958678 0.03041322) *
##       11) chloramine>=0.695 319  71 not.safe (0.77742947 0.22257053)
##         22) uranium>=0.025 222  10 not.safe (0.95495495 0.04504505) *
##         23) uranium< 0.025 97  36 safe (0.37113402 0.62886598)
##           46) radium>=4.905 20   0 not.safe (1.00000000 0.00000000) *
##           47) radium< 4.905 77  16 safe (0.20779221 0.79220779)
##             94) selenium>=0.085 12   0 not.safe (1.00000000 0.00000000) *
##             95) selenium< 0.085 65   4 safe (0.06153846 0.93846154) *
##    3) cadmium< 0.0095 1795 566 not.safe (0.68467967 0.31532033)
##      6) aluminium< 0.405 1058  94 not.safe (0.91115312 0.08884688) *
##      7) aluminium>=0.405 737 265 safe (0.35956581 0.64043419)
##       14) perchlorate>=35.06 287  73 not.safe (0.74564460 0.25435540)
##         28) ammonia>=10.855 190  10 not.safe (0.94736842 0.05263158) *
##         29) ammonia< 10.855 97  34 safe (0.35051546 0.64948454)
##           58) perchlorate>=48.4 36  10 not.safe (0.72222222 0.27777778) *
##           59) perchlorate< 48.4 61   8 safe (0.13114754 0.86885246) *
##       15) perchlorate< 35.06 450  51 safe (0.11333333 0.88666667) *
```

```
#summary(tree)
```

# Bagging via caret

Then we implement Bagging. The `train()` function of the `caret` package can be used to call a variety of supervised learning methods and also offers a number of evaluation approaches. For this, we first specify our evaluation method.

```
ctrl  <- trainControl(method = "cv",
                      number = 5)
```

Now we can call `train()`, along with the specification of the model and the evaluation method. Return the cross-validation results.

```
cbag <- train(make.names(is_safe) ~ .,
              data = c_train,
              method = "treebag",
              trControl = ctrl)

cbag
```

```
## Bagged CART
##
## 6396 samples
##   20 predictor
##    2 classes: 'not.safe', 'safe'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 5117, 5117, 5116, 5117, 5117
## Resampling results:
##
##   Accuracy   Kappa
##   0.9620072  0.7942875
```

# Random Forests

In order to also use random forests for our prediction task, we first specify a set of try-out values for model tuning. For random forest, we primarily have to care about `mtry`, i.e. the number of features to sample at each split point.

```
ncols <- ncol(c_train)
mtrys <- expand.grid(mtry = c(sqrt(ncols)-1,sqrt(ncols),sqrt(ncols)+1))
```

This object can be passed on to `train()`, along with the specification of the model, and the tuning and prediction method. Calling the random forest object lists the results of the tuning process.

```
rf <- train(make.names(is_safe) ~ .,
            data = c_train,
            method = "rf",
            trControl = ctrl,
            tuneGrid = mtrys)

rf
```

```
## Random Forest
##
## 6396 samples
##   20 predictor
##    2 classes: 'not.safe', 'safe'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 5116, 5117, 5117, 5117, 5117
## Resampling results across tuning parameters:
##
##   mtry      Accuracy   Kappa
##   3.582576  0.9534087  0.7339729
##   4.582576  0.9579422  0.7636122
##   5.582576  0.9588807  0.7712390
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 5.582576.
```

# AdaBoost

In order to build a set of prediction models it is helpful to follow the `caret` workflow and first decide how to conduct model tuning. Here we use 5-Fold Cross-Validation, mainly to keep computation time to a minimum. `caret` offers many performance metrics, however, they are stored in different functions that need to be combined first.

Now we can specifiy the `trainControl` object.

```
evalStats <- function(...) c(twoClassSummary(...),
                             defaultSummary(...),
                             mnLogLoss(...))
```

```
ctrl  <- trainControl(method = "cv",
                      number = 5,
                      summaryFunction = evalStats,
                      #verboseIter = TRUE,
                      classProbs = TRUE)
```

As a first method we try out AdaBoost as implemented in the `fastAdaboost` package. Specifically, Adaboost.M1 will be used with three try-out values for the number of iterations.

```
grid <- expand.grid(nIter = c(50, 100, 150),
                    method = "Adaboost.M1")
```

Now we can pass these two objects on to `train` , along with the specification of the model and the method, i.e. `adaboost` . List the results of the tuning process.

```
#set.seed(744)
levels(c_train$is_safe) <-c("not safe", "safe")
ada <- train(make.names(is_safe) ~.,
             data = c_train,
             method = "adaboost",
             trControl = ctrl,
             tuneGrid = grid,
             metric = "ROC")

ada
```

```
## AdaBoost Classification Trees
##
## 6396 samples
##   20 predictor
##    2 classes: 'not.safe', 'safe'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 5117, 5117, 5117, 5116, 5117
## Resampling results across tuning parameters:
##
##   nIter  ROC        Sens       Spec       Accuracy   Kappa      logLoss
##    50    0.9831076  0.9910003  0.7572508  0.9643533  0.8083607  0.1498508
##   100    0.9820817  0.9911770  0.7490033  0.9635715  0.8035215  0.1622645
##   150    0.9818702  0.9918826  0.7256778  0.9615389  0.7898520  0.1687610
##
## Tuning parameter 'method' was held constant at a value of Adaboost.M1
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were nIter = 50 and method = Adaboost.M1.
```

# GBM

For Gradient Boosting as implemented by the `gbm` package, we have to take care of a number of tuning parameters. Now the `expand.grid` is helpful as it creates an object with all possible combinations of our try-out values.

```
grid <- expand.grid(interaction.depth = 1:3,
                    n.trees = c(500, 750, 1000),
                    shrinkage = c(0.05, 0.01),
                    n.minobsinnode = 10)
```

List the tuning grid…

```
grid
```

| interaction.depth <int> | n.trees <dbl> | shrinkage <dbl> | n.minobsinnode <dbl> |
|---|---|---|---|
| 1 | 500 | 0.05 | 10 |
| 2 | 500 | 0.05 | 10 |

| interaction.depth <int> | n.trees <dbl> | shrinkage <dbl> | n.minobsinnode <dbl> |
|---|---|---|---|
| 3 | 500 | 0.05 | 10 |
| 1 | 750 | 0.05 | 10 |
| 2 | 750 | 0.05 | 10 |
| 3 | 750 | 0.05 | 10 |
| 1 | 1000 | 0.05 | 10 |
| 2 | 1000 | 0.05 | 10 |
| 3 | 1000 | 0.05 | 10 |
| 1 | 500 | 0.01 | 10 |

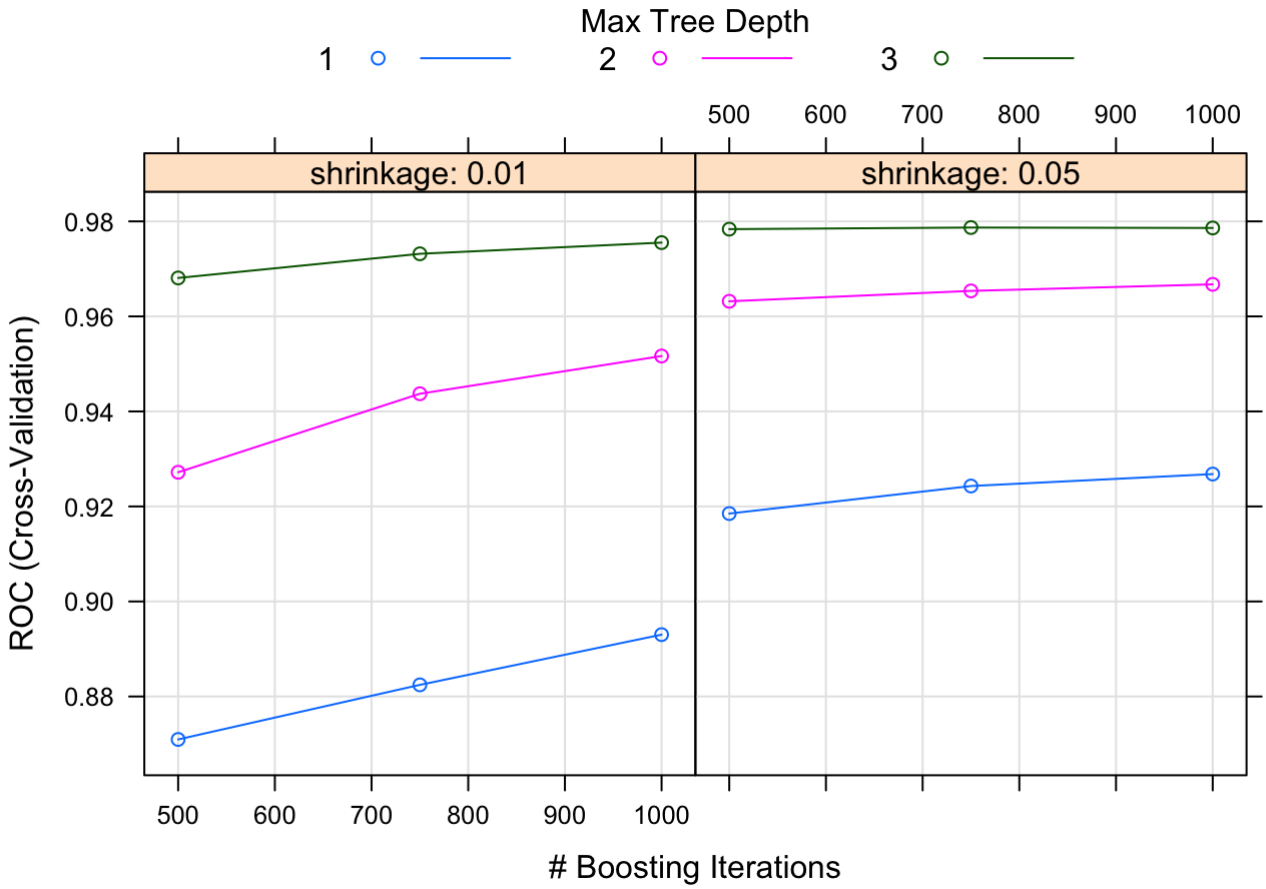
1-10 of 18 rows Previous **1** 2 Next

…and begin the tuning process.

```
gbm <- train(make.names(is_safe) ~.,
             data = c_train,
             method = "gbm",
             trControl = ctrl,
             tuneGrid = grid,
             metric = "ROC",
             distribution = "bernoulli",
             verbose = FALSE)
```

Instead of just printing the results from the tuning process, we can also plot them.

```
plot(gbm)
```




### Logistic regression

Finally we also add a logistic regression model. Obviously we have no tuning parameter here.

```
set.seed(744)
logit <- train(make.names(is_safe) ~.,
               data = c_train,
               method = "glm",
               trControl = ctrl)
```

We may want to take a glimpse at the regression results.

```
summary(logit)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1428  -0.4063  -0.2507  -0.1342   3.7621
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.855010   0.262315   3.259 0.001116 **
## aluminium     0.728479   0.037220  19.572  < 2e-16 ***
## ammonia      -0.026987   0.005484  -4.921 8.60e-07 ***
## arsenic      -3.561512   0.376653  -9.456  < 2e-16 ***
## barium        0.121875   0.044544   2.736 0.006218 **
## cadmium     -19.971121   2.015911  -9.907  < 2e-16 ***
## chloramine    0.204413   0.022822   8.957  < 2e-16 ***
## chromium      1.116680   0.201234   5.549 2.87e-08 ***
## copper       -0.401702   0.079209  -5.071 3.95e-07 ***
## flouride      0.075853   0.109451   0.693 0.488291
## bacteria      0.852641   0.239968   3.553 0.000381 ***
## viruses      -1.236774   0.205998  -6.004 1.93e-09 ***
## lead         -1.533819   0.841818  -1.822 0.068450 .
## nitrates     -0.058207   0.008593  -6.774 1.26e-11 ***
## nitrites     -0.374788   0.109884  -3.411 0.000648 ***
## mercury     -48.633292  15.626291  -3.112 0.001857 **
## perchlorate  -0.027899   0.003496  -7.980 1.47e-15 ***
## radium       -0.035367   0.022140  -1.597 0.110176
## selenium     -4.749325   1.658280  -2.864 0.004183 **
## silver       -1.391791   0.390303  -3.566 0.000363 ***
## uranium     -12.024070   1.810252  -6.642 3.09e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4538.0  on 6395  degrees of freedom
## Residual deviance: 3131.7  on 6375  degrees of freedom
## AIC: 3173.7
##
## Number of Fisher Scoring iterations: 6
```

# Prediction and Performance

Finally, we predict the outcome in the test set.

```
c_tree <- predict(tree, newdata = c_test, type = "class")
c_tree <- as.factor(ifelse(c_tree=="not.safe", 0, 1))
c_cbag <- predict(cbag, newdata = c_test)
c_cbag <- as.factor(ifelse(c_cbag=="not.safe", 0, 1))
c_rf <- predict(rf, newdata = c_test)
c_rf <- as.factor(ifelse(c_rf=="not.safe", 0, 1))
c_ada <- predict(ada, newdata = c_test)
c_ada <- as.factor(ifelse(c_ada=="not.safe", 0, 1))
c_gbm <- predict(gbm, newdata = c_test)
c_gbm <- as.factor(ifelse(c_gbm=="not.safe", 0, 1))
c_logit <- predict(logit, newdata = c_test)
c_logit <-as.factor(ifelse(c_logit=="not.safe", 0, 1))

p_tree <- predict(tree, newdata = c_test, type = "prob")
p_cbag <- predict(cbag, newdata = c_test, type = "prob")
p_rf <- predict(rf, newdata = c_test, type = "prob")
p_ada <- predict(ada, newdata = c_test, type = "prob")
p_gbm <- predict(gbm, newdata = c_test, type = "prob")
p_logit <- predict(logit, newdata = c_test, type = "prob")
```

Given predicted class membership, we can use the function `postResample` in order to get a short summary of each models' performance in the test set.

```
paste0("Accuracy predicted by tree: ", postResample(c_tree, c_test$is_safe)[1])
```

```
## [1] "Accuracy predicted by tree: 0.956875"
```

```
paste0("Accuracy predicted by bagging tree: ",postResample(c_cbag, c_test$is_safe)[1
])
```

```
## [1] "Accuracy predicted by bagging tree: 0.97"
```

```
paste0("Accuracy predicted by random forest: ",postResample(c_rf, c_test$is_safe)[1])
```

```
## [1] "Accuracy predicted by random forest: 0.963125"
```

```
paste0("Accuracy predicted by Adaboosting: ", postResample(pred = c_ada, obs = c_test
$is_safe)[1])
```

```
## [1] "Accuracy predicted by Adaboosting: 0.969375"
```

```
paste0("Accuracy predicted by XGboosting: ", postResample(pred = c_gbm, obs = c_test
$is_safe)[1])
```

```
## [1] "Accuracy predicted by XGboosting: 0.9575"
```

```
paste0("Accuracy predicted by Logistic Regreesion: ", postResample(pred = c_logit, ob
s = c_test$is_safe)[1])
```

```
## [1] "Accuracy predicted by Logistic Regreesion: 0.91125"
```

# ROC Curve

Creating `ROC` objects based on predicted probabilities…

```
tree_roc <- roc(c_test$is_safe, p_tree[,2])
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

```
cbag_roc <- roc(c_test$is_safe, p_cbag[,2])
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
rf_roc <- roc(c_test$is_safe, p_rf[,2])
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
ada_roc <- roc(c_test$is_safe, p_ada$safe)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
gbm_roc <- roc(c_test$is_safe, p_gbm$safe)
```

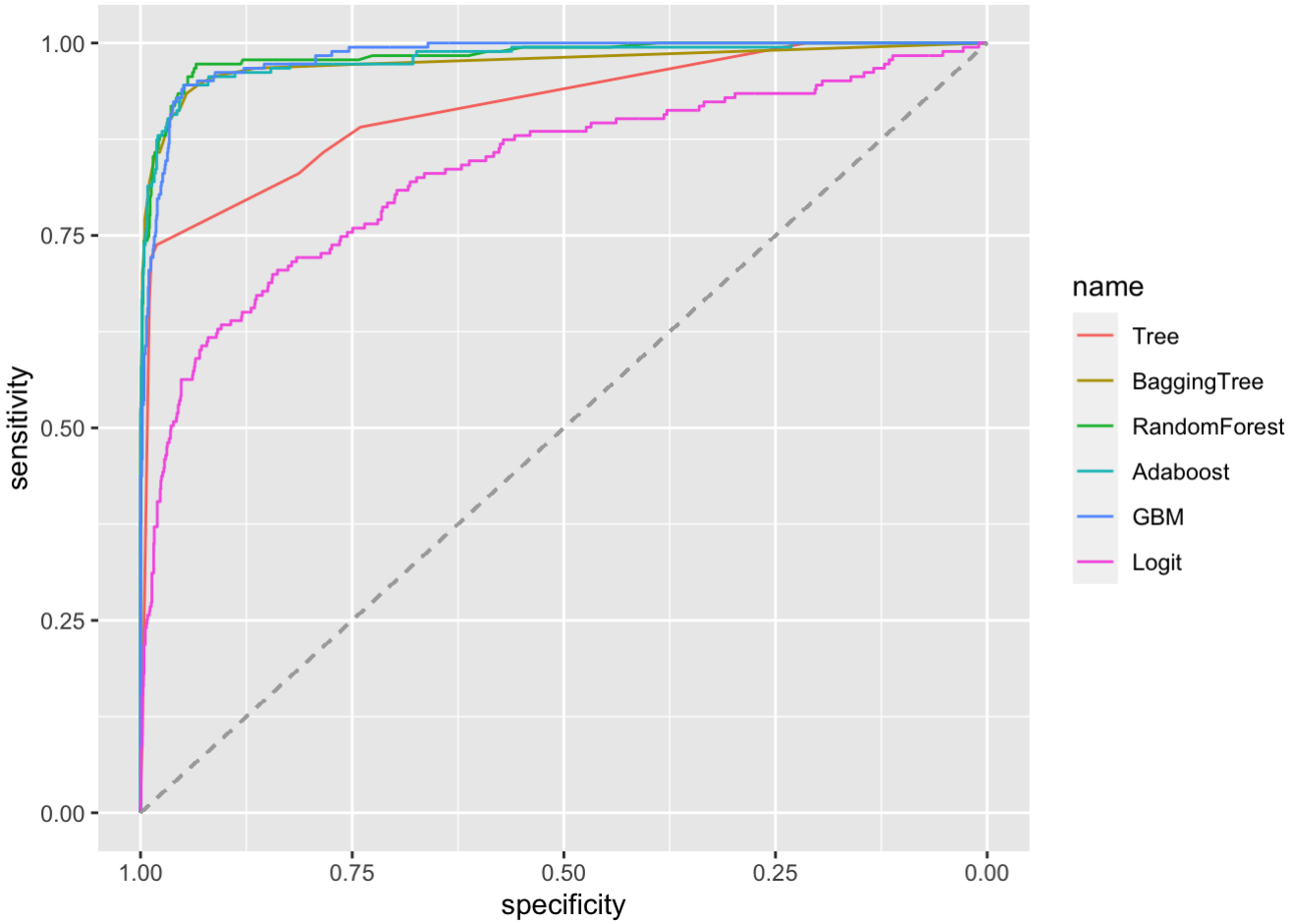
```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
logit_roc <- roc(c_test$is_safe, p_logit$safe)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

…and plotting the ROC curves.

```
ggroc(list(Tree = tree_roc,
           BaggingTree = cbag_roc,
           RandomForest = rf_roc,
           Adaboost = ada_roc,
           GBM = gbm_roc,
           #CART = cart_roc,
           Logit = logit_roc)) +
  geom_segment(aes(x = 1, xend = 0, y = 0, yend = 1),
               color="darkgrey", linetype="dashed")
```

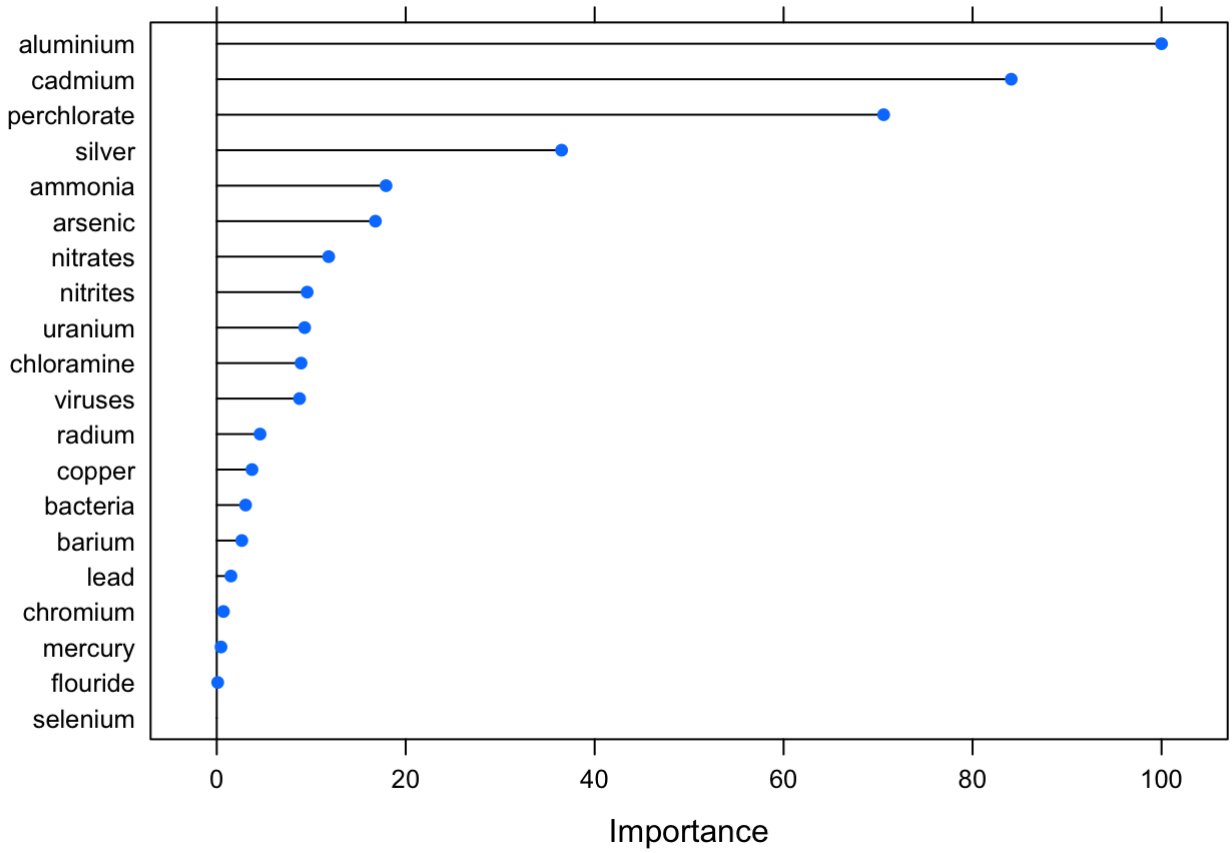



From the ROC curve, we can see that models other than Decision Trees and Logistic Regression performed pretty good, since the curve are very close to the top left corner.

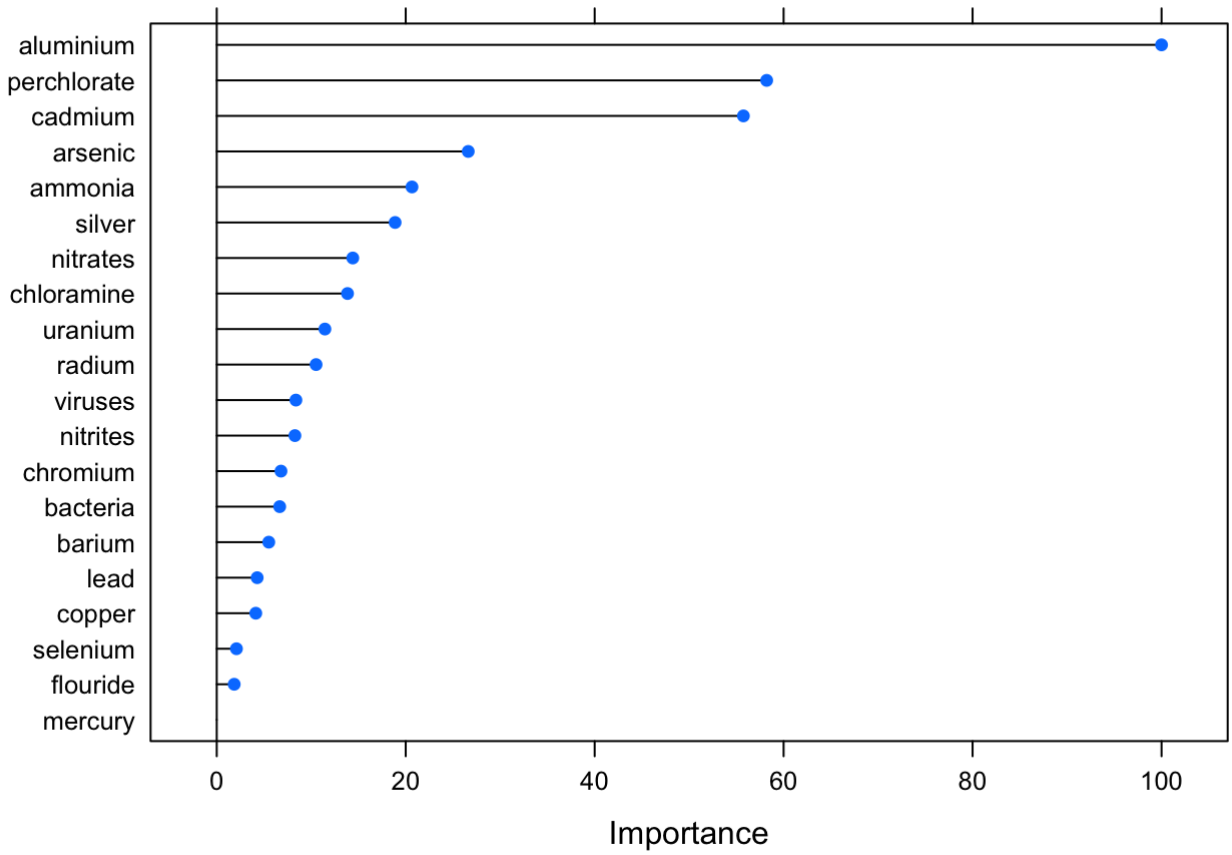
# Variable Importance (Method I)

A quick look at feature importances in different models.

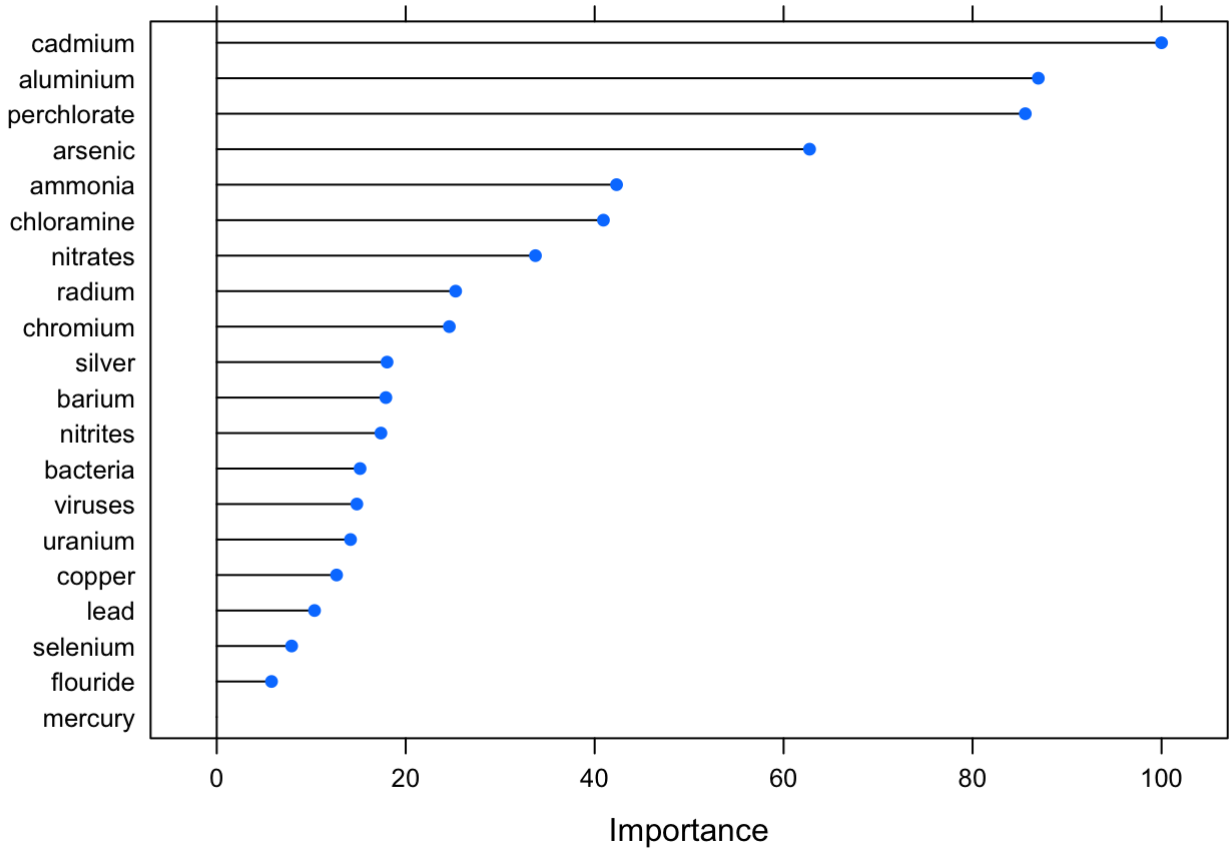
```
plot(varImp(gbm))
```




```
plot(varImp(rf))
```



```
plot(varImp(cbag))
```




```
#plot(varImp(ada))
#plot(varImp(logit))
#plot(varImp(tree))
```

# Variable Importance (Method II)

Another approach for selecting features is (simple) feature screening via filtering. In this context it is important to be cautious when estimating predicting performance and correctly combine filtering and CV. The `caret` can be used to take care of that. First, we have to set up our inner (trainControl) and outer (sbfControl) evaluation techniques in Lasso variable selection.

```
sbf_ctrl <- sbfControl(functions = caretSBF,
                       method = "cv",
                       number = 10)

ctrl <- trainControl(method = "none")
```

Now we can run CV with feature selection by filter via `sbf` .

```
m3 <- sbf(is_safe ~.,
          data = c_train,
          method = 'glmnet',
          sbfControl = sbf_ctrl,
          trControl = ctrl)
```

The corresponding results object gives us an estimate of prediction performance and information on the selected features.

```
m3
```

```
##
## Selection By Filter
##
## Outer resampling method: Cross-Validated (10 fold)
##
## Resampling performance:
##
##  Accuracy  Kappa AccuracySD KappaSD
##    0.9057 0.3853   0.009575  0.0657
##
## Using the training set, 16 variables were selected:
##    aluminium, ammonia, arsenic, barium, cadmium...
##
## During resampling, the top 5 selected variables (out of a possible 17):
##    aluminium (100%), ammonia (100%), arsenic (100%), barium (100%), cadmium (100%)
##
## On average, 16.2 variables were selected (min = 16, max = 17)
```

# After choosing variable

We compared two ways of variables importance check, and found that aluminium, cadmium, arsenic, ammonia, barium, chloramine, and perchlorate are the most important variables among 20 features.

Partial dependence plots can be useful in order to see how the features are related to the outcome according to the fitted model. With the `pdp` package, we start by running the `partial()` function with the variables of interest.

```
library(iml)
library(DALEX)
```

```
## Welcome to DALEX (version: 2.3.0).
## Find examples and detailed introduction at: http://ema.drwhy.ai/
```

```
##
## 载入程辑包：'DALEX'
```

```
## The following object is masked from 'package:dplyr':
##
##     explain
```

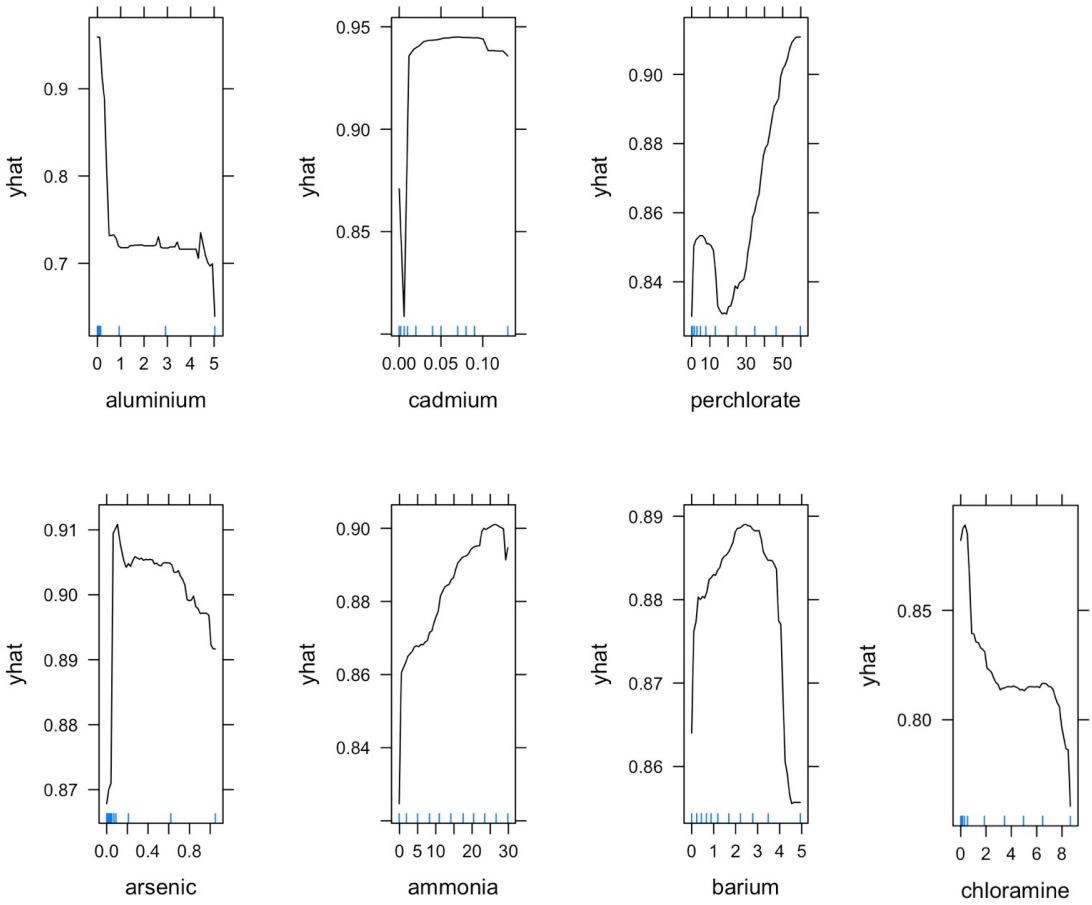
```
library(pdp)
pdp1 <- partial(gbm, pred.var = "aluminium",
                type = "classification",
                which.class = 1, prob = T)
pdp2 <- partial(rf, pred.var = "cadmium",
                type = "classification",
                which.class = 1, prob = T)
pdp3 <- partial(rf, pred.var = "perchlorate",
                type = "classification",
                which.class = 1, prob = T)
pdp4 <- partial(rf, pred.var = "arsenic",
                type = "classification",
                which.class = 1, prob = T)
pdp5 <- partial(rf, pred.var = "ammonia",
                type = "classification",
                which.class = 1, prob = T)
pdp6 <- partial(rf, pred.var = "barium",
                type = "classification",
                which.class = 1, prob = T)
pdp7 <- partial(rf, pred.var = "chloramine",
                type = "classification",
                which.class = 1, prob = T)
```

# PDP Plot

The actual plots can be created with `plotPartial()`.

```
p1 <- plotPartial(pdp1, rug = T, train = c_train)
p2 <- plotPartial(pdp2, rug = T, train = c_train)
p3 <- plotPartial(pdp3, rug = T, train = c_train)
p4 <- plotPartial(pdp4, rug = T, train = c_train)
p5 <- plotPartial(pdp5, rug = T, train = c_train)
p6 <- plotPartial(pdp6, rug = T, train = c_train)
p7 <- plotPartial(pdp7, rug = T, train = c_train)


grid.arrange(p1, p2, p3,p4,p5,p6,p7, ncol = 2)
```




From PDP plots, we can see that the predicted y was wider spread in aluminium and chlormine, especially when attributes' level per liter increase, the predicted results drop, which elicit the probabilities increase for the water tested not safe. Others remain small differences in predicted results, either increases when level of attributes' in water per liter also increases, or drop a litter compared with aluminium and chlormine.

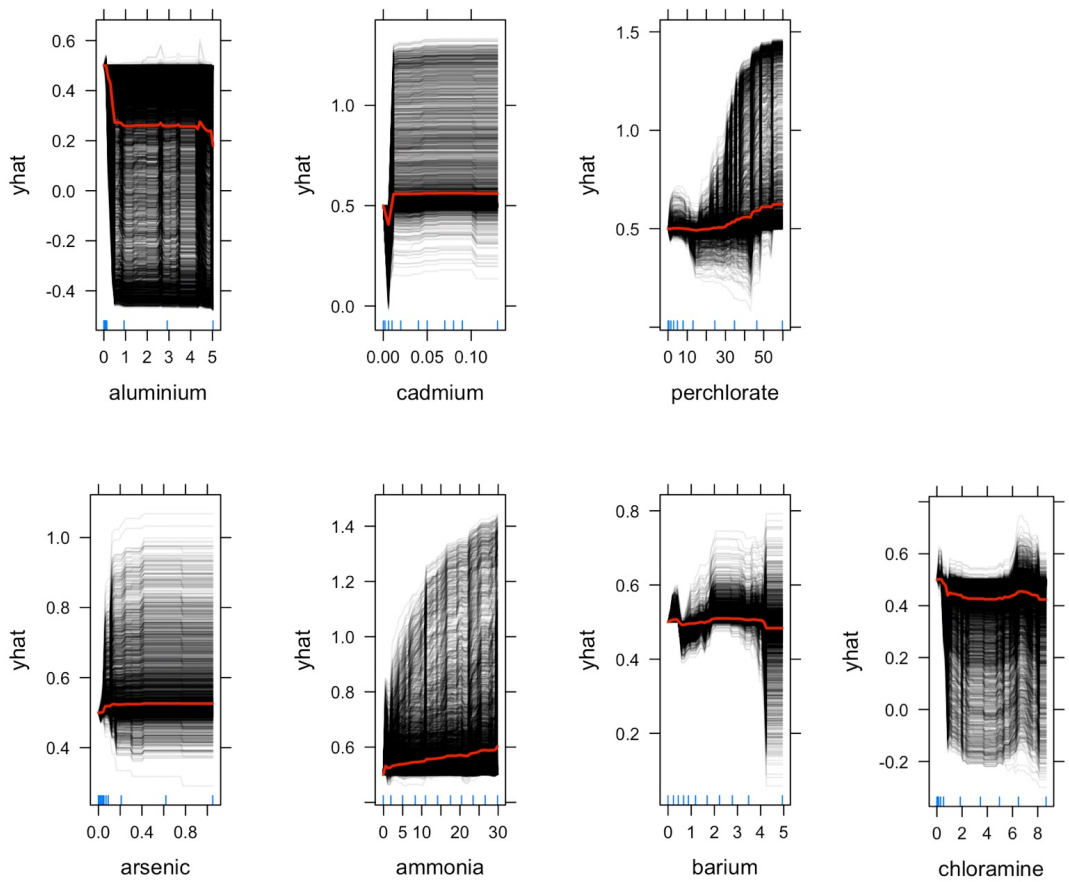
# ICE Plot

Then we plot ICE plots to show how the instance's prediction changes when a feature changes.

```
pdp1 <- partial(gbm, pred.var = "aluminium",
                type = "classification",
                which.class = 1, prob = T,
                ice = TRUE, center = T)
pdp2 <- partial(gbm, pred.var = "cadmium",
                type = "classification",
                which.class = 1, prob = T,
                ice = TRUE, center = T)
pdp3 <- partial(gbm, pred.var = "perchlorate",
                type = "classification",
                which.class = 1, prob = T,
                ice = TRUE, center = T)
pdp4 <- partial(gbm, pred.var = "arsenic",
                type = "classification",
                which.class = 1, prob = T,
                ice = TRUE, center = T)
pdp5 <- partial(gbm, pred.var = "ammonia",
                type = "classification",
                which.class = 1, prob = T,
                ice = TRUE, center = T)
pdp6 <- partial(gbm, pred.var = "barium",
                type = "classification",
                which.class = 1, prob = T,
                ice = TRUE, center = T)
pdp7 <- partial(gbm, pred.var = "chloramine",
                type = "classification",
                which.class = 1, prob = T,
                ice = TRUE, center = T)
```

```
p1 <- plotPartial(pdp1, rug = T, train = c_train, alpha = 0.1)
p2 <- plotPartial(pdp2, rug = T, train = c_train, alpha = 0.1)
p3 <- plotPartial(pdp3, rug = T, train = c_train, alpha = 0.1)
p4 <- plotPartial(pdp4, rug = T, train = c_train, alpha = 0.1)
p5 <- plotPartial(pdp5, rug = T, train = c_train, alpha = 0.1)
p6 <- plotPartial(pdp6, rug = T, train = c_train, alpha = 0.1)
p7 <- plotPartial(pdp7, rug = T, train = c_train, alpha = 0.1)


grid.arrange(p1, p2, p3,p4,p5,p6,p7, ncol = 2)
```




From ICE plots, we can see that except aluminium and cadmium, the prediction remain unchanged throughout the changes in levels of other attributes in water per liter. For aluminium, the prediction decreases when level of aluminium reached 0.5. For cadmium, the prediction increases when level of aluminium reached 0.005.

Therefore, we can assume that using these attributes to predict the water quality would be enough, since knowing the levels of attributes would not affect the prediction quite a lot. So we use these attributes to support our assumption.

# Decision Trees

```
tree1 <- rpart(make.names(is_safe) ~ aluminium +chloramine+barium + perchlorate +
                ammonia+arsenic + cadmium, data = c_train, method = "class")
tree1
```

```
## n= 6396
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 6396 729 not.safe (0.88602251 0.11397749)
##    2) cadmium>=0.0095 4601 163 not.safe (0.96457292 0.03542708) *
##    3) cadmium< 0.0095 1795 566 not.safe (0.68467967 0.31532033)
##      6) aluminium< 0.405 1058  94 not.safe (0.91115312 0.08884688) *
##      7) aluminium>=0.405 737 265 safe (0.35956581 0.64043419)
##       14) perchlorate>=35.06 287  73 not.safe (0.74564460 0.25435540)
##         28) ammonia>=10.855 190  10 not.safe (0.94736842 0.05263158) *
##         29) ammonia< 10.855 97  34 safe (0.35051546 0.64948454)
##           58) perchlorate>=48.4 36  10 not.safe (0.72222222 0.27777778) *
##           59) perchlorate< 48.4 61   8 safe (0.13114754 0.86885246) *
##       15) perchlorate< 35.06 450  51 safe (0.11333333 0.88666667) *
```

```
#summary(tree)
```

# Bagging via caret

Although useful for demonstration purposes, we don't need to program our own loop each time to implement Bagging. The `train()` function of the `caret` package can be used to call a variety of supervised learning methods and also offers a number of evaluation approaches. For this, we first specify our evaluation method.

```
ctrl  <- trainControl(method = "cv",
                      number = 5)
```

Now we can call `train()`, along with the specification of the model and the evaluation method. Return the cross-validation results.

```
cbag1 <- train(make.names(is_safe) ~ aluminium +chloramine+barium + perchlorate + amm
onia+arsenic + cadmium,
               data = c_train,
               method = "treebag",
               trControl = ctrl)

cbag1
```

```
## Bagged CART
##
## 6396 samples
##    7 predictor
##    2 classes: 'not.safe', 'safe'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 5117, 5117, 5116, 5117, 5117
## Resampling results:
##
##   Accuracy   Kappa
##   0.9468414  0.6947227
```

# Random Forests

In order to also use random forests for our prediction task, we first specify a set of try-out values for model tuning. For random forest, we primarily have to care about `mtry`, i.e. the number of features to sample at each split point.

```
ncols <- ncol(c_train)
mtrys <- expand.grid(mtry = c(sqrt(ncols)-1,sqrt(ncols),sqrt(ncols)+1))
```

This object can be passed on to `train()`, along with the specification of the model, and the tuning and prediction method. For random forests, we use `rf`. Calling the random forest object lists the results of the tuning process.

```
rf1 <- train(make.names(is_safe) ~ aluminium +chloramine+barium + perchlorate + ammon
ia+arsenic + cadmium,
             data = c_train,
             method = "rf",
             trControl = ctrl,
             tuneGrid = mtrys)

rf1
```

```
## Random Forest
##
## 6396 samples
##    7 predictor
##    2 classes: 'not.safe', 'safe'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 5116, 5117, 5116, 5118, 5117
## Resampling results across tuning parameters:
##
##   mtry      Accuracy   Kappa
##   3.582576  0.9484056  0.7005305
##   4.582576  0.9485622  0.7013267
##   5.582576  0.9479367  0.6982482
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 4.582576.
```

# AdaBoost

In order to build a set of prediction models it is helpful to follow the `caret` workflow and first decide how to conduct model tuning. Here we use 5-Fold Cross-Validation, mainly to keep computation time to a minimum. `caret` offers many performance metrics, however, they are stored in different functions that need to be combined first.

Now we can specifiy the `trainControl` object.

```
evalStats <- function(...) c(twoClassSummary(...),
                             defaultSummary(...),
                             mnLogLoss(...))
```

```
ctrl  <- trainControl(method = "cv",
                      number = 5,
                      summaryFunction = evalStats,
                      #verboseIter = TRUE,
                      classProbs = TRUE)
```

As a first method we try out AdaBoost as implemented in the `fastAdaboost` package. Specifically, Adaboost.M1 will be used with three try-out values for the number of iterations.

```
grid <- expand.grid(nIter = c(50, 100, 150),
                    method = "Adaboost.M1")
```

Now we can pass these two objects on to `train`, along with the specification of the model and the method, i.e. `adaboost`. List the results of the tuning process.

```
#set.seed(744)
ada1 <- train(make.names(is_safe) ~aluminium +chloramine+barium + perchlorate + ammon
ia+arsenic + cadmium,
              data = c_train,
              method = "adaboost",
              trControl = ctrl,
              tuneGrid = grid,
              metric = "ROC")

ada1
```

```
## AdaBoost Classification Trees
##
## 6396 samples
##    7 predictor
##    2 classes: 'not.safe', 'safe'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 5116, 5117, 5117, 5116, 5118
## Resampling results across tuning parameters:
##
##   nIter  ROC        Sens       Spec       Accuracy   Kappa      logLoss
##    50    0.8639130  0.9862365  0.5994332  0.9421529  0.6713462  0.2591924
##   100    0.8633594  0.9878246  0.6049315  0.9441854  0.6818016  0.2581924
##   150    0.8599476  0.9890601  0.6008125  0.9448110  0.6830466  0.2620907
##
## Tuning parameter 'method' was held constant at a value of Adaboost.M1
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were nIter = 50 and method = Adaboost.M1.
```

# GBM

For Gradient Boosting as implemented by the `gbm` package, we have to take care of a number of tuning parameters. Now the `expand.grid` is helpful as it creates an object with all possible combinations of our try-out values.

```
grid <- expand.grid(interaction.depth = 1:3,
                    n.trees = c(500, 750, 1000),
                    shrinkage = c(0.05, 0.01),
                    n.minobsinnode = 10)
```

List the tuning grid…

```
grid
```

| interaction.depth <int> | n.trees <dbl> | shrinkage <dbl> | n.minobsinnode <dbl> |
|---|---|---|---|
| 1 | 500 | 0.05 | 10 |
| 2 | 500 | 0.05 | 10 |
| 3 | 500 | 0.05 | 10 |
| 1 | 750 | 0.05 | 10 |
| 2 | 750 | 0.05 | 10 |
| 3 | 750 | 0.05 | 10 |
| 1 | 1000 | 0.05 | 10 |
| 2 | 1000 | 0.05 | 10 |
| 3 | 1000 | 0.05 | 10 |
| 1 | 500 | 0.01 | 10 |

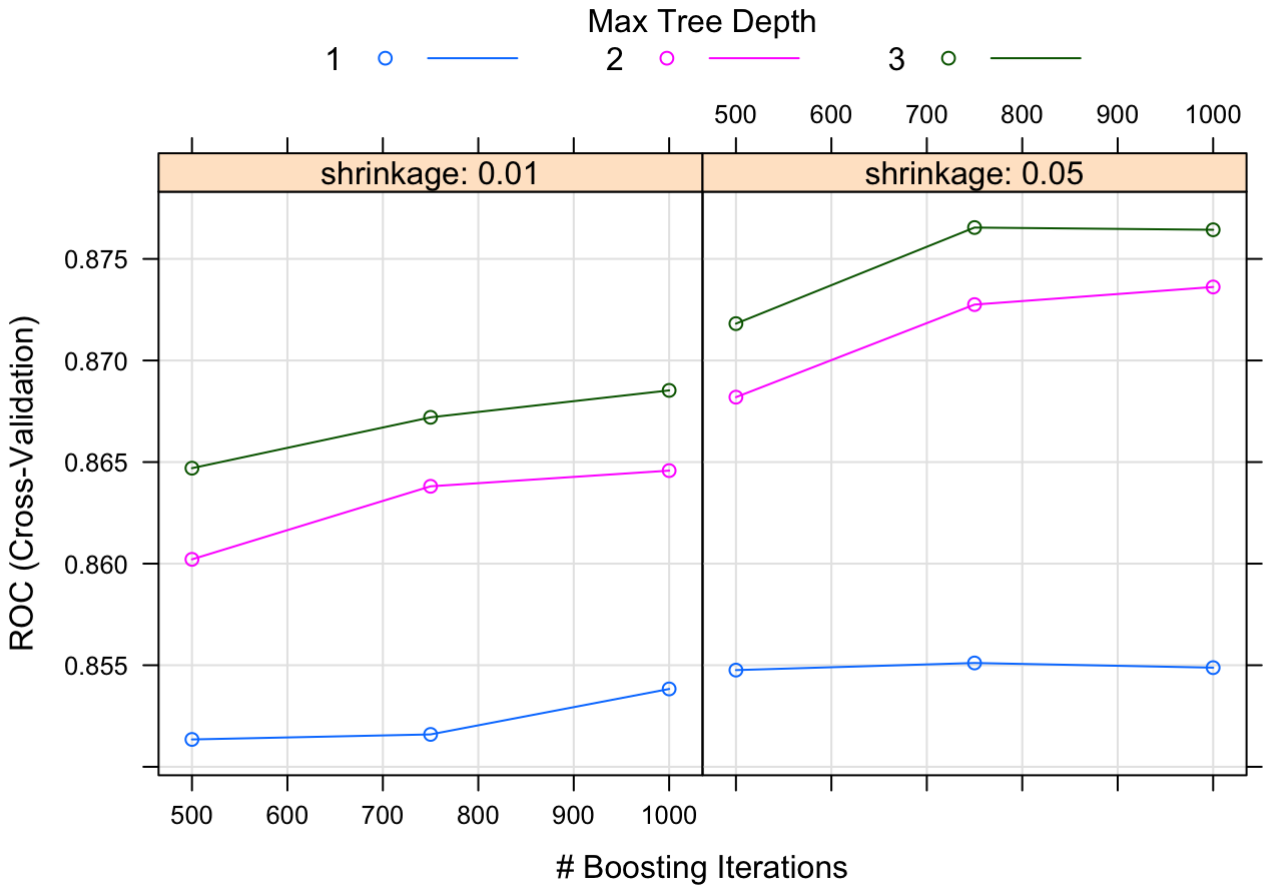
1-10 of 18 rows Previous **1** 2 Next

…and begin the tuning process.

```
#set.seed(744)
gbm1 <- train(make.names(is_safe) ~aluminium +chloramine+barium + perchlorate + ammon
ia+arsenic + cadmium,
              data = c_train,
              method = "gbm",
              trControl = ctrl,
              tuneGrid = grid,
              metric = "ROC",
              distribution = "bernoulli",
              verbose = FALSE)
```

Instead of just printing the results from the tuning process, we can also plot them.

```
plot(gbm1)
```




### Logistic regression

Finally we also add a logistic regression model. Obviously we have no tuning parameter here. We may want to take a glimpse at the regression results.

```
set.seed(744)
logit1 <- train(make.names(is_safe) ~aluminium +chloramine+barium + perchlorate + amm
onia+arsenic + cadmium,
              data = c_train,
              method = "glm",
              trControl = ctrl)

summary(logit1)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.0731  -0.4291  -0.2912  -0.1815   3.8124
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -1.559591   0.121503 -12.836  < 2e-16 ***
## aluminium     0.640487   0.033102  19.349  < 2e-16 ***
## chloramine    0.189343   0.020181   9.382  < 2e-16 ***
## barium        0.092667   0.042049   2.204   0.0275 *
## perchlorate  -0.025792   0.003259  -7.915 2.48e-15 ***
## ammonia      -0.026007   0.005211  -4.991 6.01e-07 ***
## arsenic      -3.374574   0.362144  -9.318  < 2e-16 ***
## cadmium     -20.102047   1.936697 -10.380  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4538.0  on 6395  degrees of freedom
## Residual deviance: 3362.1  on 6388  degrees of freedom
## AIC: 3378.1
##
## Number of Fisher Scoring iterations: 6
```

# Prediction and Performance

Finally, we predict the outcome in the test set.

```
c_tree1 <- predict(tree1, newdata = c_test, type = "class")
c_tree1 <- as.factor(ifelse(c_tree1=="not.safe", 0, 1))
c_cbag1 <- predict(cbag1, newdata = c_test)
c_cbag1 <- as.factor(ifelse(c_cbag1=="not.safe", 0, 1))
c_rf1 <- predict(rf1, newdata = c_test)
c_rf1 <- as.factor(ifelse(c_rf1=="not.safe", 0, 1))
c_ada1 <- predict(ada1, newdata = c_test)
c_ada1 <- as.factor(ifelse(c_ada1=="not.safe", 0, 1))
c_gbm1 <- predict(gbm1, newdata = c_test)
c_gbm1 <- as.factor(ifelse(c_gbm1=="not.safe", 0, 1))
c_logit1 <- predict(logit1, newdata = c_test)
c_logit1 <-as.factor(ifelse(c_logit1=="not.safe", 0, 1))

p_tree1 <- predict(tree1, newdata = c_test, type = "prob")
p_cbag1 <- predict(cbag1, newdata = c_test, type = "prob")
p_rf1 <- predict(rf1, newdata = c_test, type = "prob")
p_ada1 <- predict(ada1, newdata = c_test, type = "prob")
p_gbm1 <- predict(gbm1, newdata = c_test, type = "prob")
p_logit1 <- predict(logit1, newdata = c_test, type = "prob")
```

Given predicted class membership, we can use the function `postResample` in order to get a short summary of each models' performance in the test set.

```
paste0("Accuracy predicted by tree: ", postResample(c_tree1, c_test$is_safe)[1])
```

```
## [1] "Accuracy predicted by tree: 0.946875"
```

```
paste0("Accuracy predicted by bagging tree: ",postResample(c_cbag1, c_test$is_safe)[1
])
```

```
## [1] "Accuracy predicted by bagging tree: 0.945625"
```

```
paste0("Accuracy predicted by random forest: ",postResample(c_rf1, c_test$is_safe)[1
])
```

```
## [1] "Accuracy predicted by random forest: 0.945"
```

```
paste0("Accuracy predicted by Adaboosting: ", postResample(pred = c_ada1, obs = c_tes
t$is_safe)[1])
```

```
## [1] "Accuracy predicted by Adaboosting: 0.943125"
```

```
paste0("Accuracy predicted by XGboosting: ", postResample(pred = c_gbm1, obs = c_test
$is_safe)[1])
```

```
## [1] "Accuracy predicted by XGboosting: 0.9425"
```

```
paste0("Accuracy predicted by Logistic Regreesion: ", postResample(pred = c_logit1, o
bs = c_test$is_safe)[1])
```

```
## [1] "Accuracy predicted by Logistic Regreesion: 0.90875"
```

# ROC Curve

Creating `ROC` objects based on predicted probabilities…

```
tree_roc1 <- roc(c_test$is_safe, p_tree1[,2])
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

```
cbag_roc1 <- roc(c_test$is_safe, p_cbag1$safe)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
rf_roc1 <- roc(c_test$is_safe, p_rf1$safe)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
ada_roc1 <- roc(c_test$is_safe, p_ada1$safe)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
gbm_roc1 <- roc(c_test$is_safe, p_gbm1$safe)
```

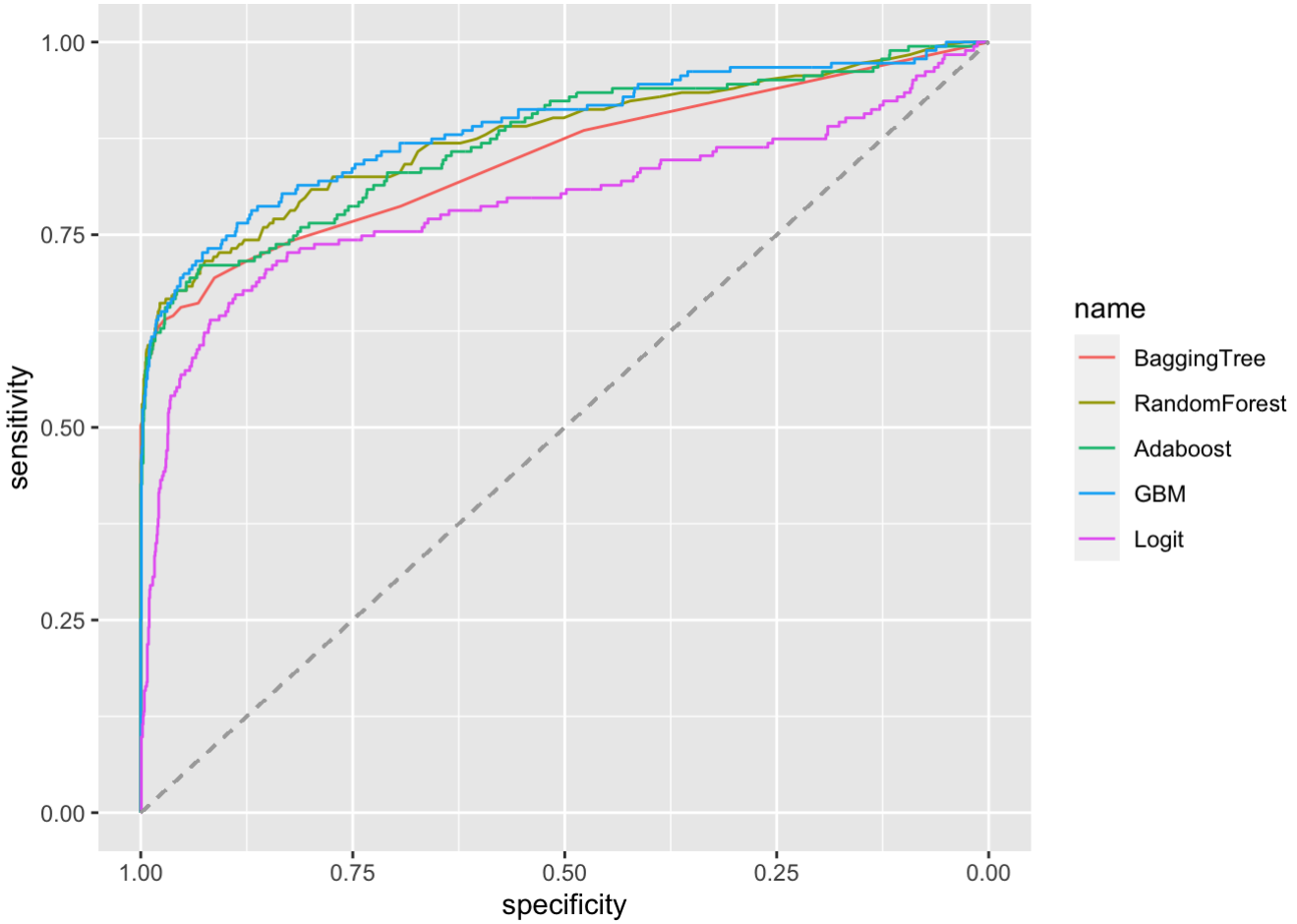
```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
logit_roc1 <- roc(c_test$is_safe, p_logit1$safe)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

…and plotting the ROC curves. We skip Tree for its lowest accuracy score.

```
ggroc(list(
            BaggingTree = cbag_roc1,
            RandomForest = rf_roc1,
            Adaboost = ada_roc1,
            GBM = gbm_roc1,
            #CART = cart_roc,
            Logit = logit_roc1)) +
  geom_segment(aes(x = 1, xend = 0, y = 0, yend = 1),
               color="darkgrey", linetype="dashed")
```




From the results above using only seven out of twenty variables, we can see that the accuracy scores do not vary much compared with previous models. Also the Bagging, Random Forest, AdaBoost and GBM performed evenly in predicting water quality in both full attributes model and selected attributes model.

# Conclusion

In conclusion, we can conclude that we can predict the water quality in all models with high accuracy scores using all attributes. We can also use a few attributes to predict the water quality with relatively high accuracy scores, which would be a faster method with similar accuracy score. When we apply the models into real-life scenarios, we are sure that even we can make it faster to get the results than going over all the ingredients in the water with maintaining similar accuracy score. Although there would be some limitations in the process, for example, the data is not generated in the real world, the result could be a little bit idealistic. We can still use the structure of data processing, and model training into real-life problem. The other limitation is the data is not representative for all the water resources around the world. However, with more real data collection, we could sample a dataset that is representitive enough for prediction in the future.

# Reference

Data Source: Water Quality from Kaggle (https://www.kaggle.com/mssmartypants/water-quality (https://www.kaggle.com/mssmartypants/water-quality))