Task 1 SL 2020

Group 3

5 de May, 2020

Contents

1 Problem

We want to predict the solubility of a compound using 72 noisy structural variables for which we have have no other information than values. The solubility of each compound is coded '1' for soluble and '0' for insoluble. It is a binary classification problem. We want to build tree based models: CART, random forest and boosting trees models.

2 Exploratory data analysis

2.1 Variables types and dataset dimension

The data set has **5631** observations on **73** variables, that is one outcome variable and **72** predictors. All the predictors are named as **x1** to **x72** and are continuous variables.

2.2 Variability

Outcome variable:

The variable **y** indicates the solubility. 2138 of the compounds are soluble, and 3493 are not, which represents a 37.97% and a 62.03%, respectively. The classes are not fully balanced but both classes are well represented.

```
##
## insoluble   soluble
##      3493      2138
```

Predictors:

Boxplots of our continuous predictors (Fig.1) show the scale and variability of the predictors vary greatly. The difference in scales is not a problem to build tree based models because those are invariant under strictly monotone transformations of the individual predictors (Hastie, Tibshirani, and Friedman 2009, p352).

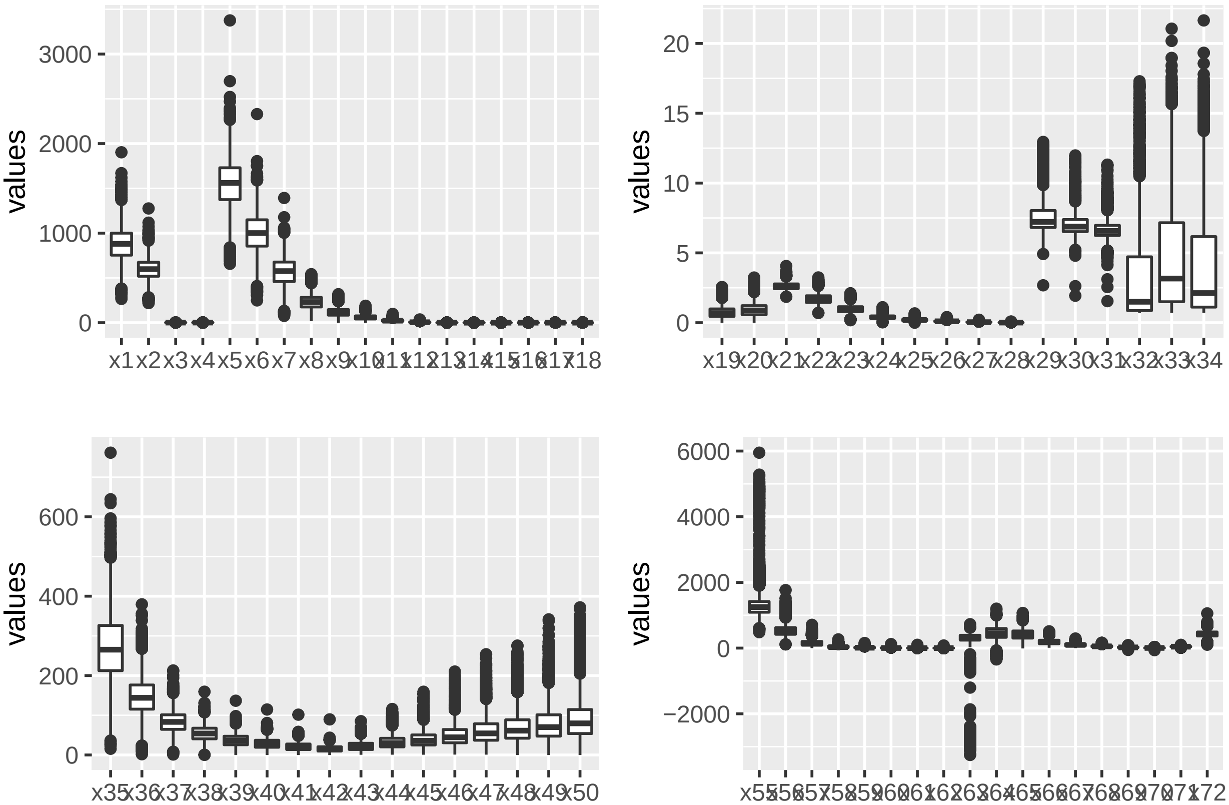

Figure 1: Boxplot of predictors

We also see that some of our predictors seem to have 0 o near 0 variance. The problem with those predictors is that they may become zero-variance predictors when the data are split into sub-samples for cross-validation for example or that a few samples with a different value from the one the majority of the sample takes may have an undue influence on the model. To identify those, two metrics can be used: the frequency of the most prevalent value over the second most frequent value and percentage of unique values. However, tree based models are insensitive to such problematic predictors so we can keep them in the dataset (Kuhn and Johnson 2013, p44).

If near 0 variance predictors are not an issue, the number of unique values in predictors can impact the tree model. Tree based models have a selection bias and favor continuous variables with more granularity as they offer more possible splits (Kuhn and Johnson 2013, p182). There is then the risk of a bias towards noise granular variables while informative but less granular variables might not be given the deserved importance. In Fig.2 below, the histogram of the number of unique values per predictor shows some predictors have less than 100 unique values. We will need to keep this in mind when interpreting the variable importance scores.

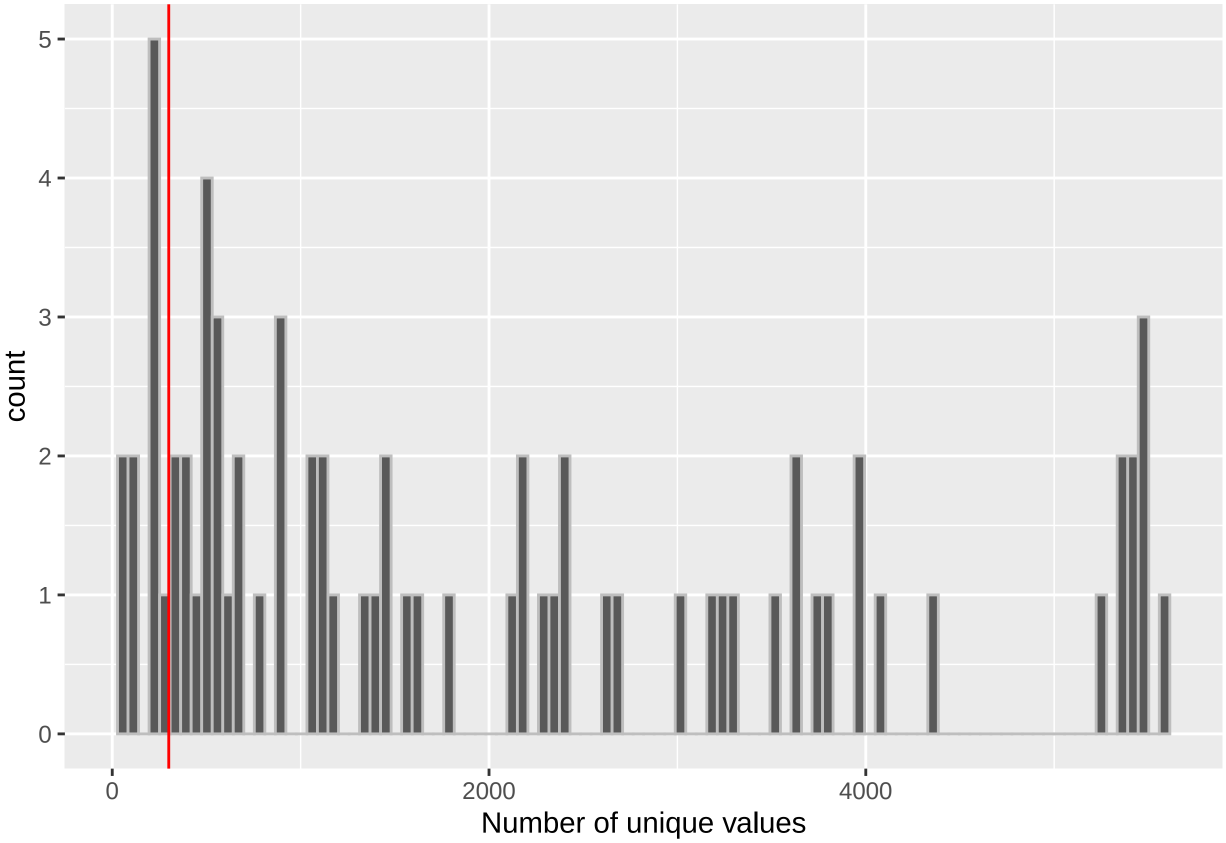

Figure 2: Histogram of number of unique values per predictor

2.3 Correlation

2.3.1 Correlation between predictors

Very high correlation between predictors is often an issue for model selection and it is the case for tree-based models (Kuhn and Johnson 2013, p202). When two predictors are highly correlated, the choice between the two for a split will be driven by small differences between the two predictors that are noise, instead of valuable information. Uninformative predictors with high correlations to informative predictors have abnormally large importance values. Moreover two perfectly correlated predictors represent the same information but both will be chosen at random in the tree growth process, and the importance of the underlying variable they represent will be diluted in the importance scores.

We build a heatmap of the correlation matrix of the predictors and the dendogram for the corresponding hierarchical clustering (Fig.3). We observe some variables correlate highly (red) and form clusters.

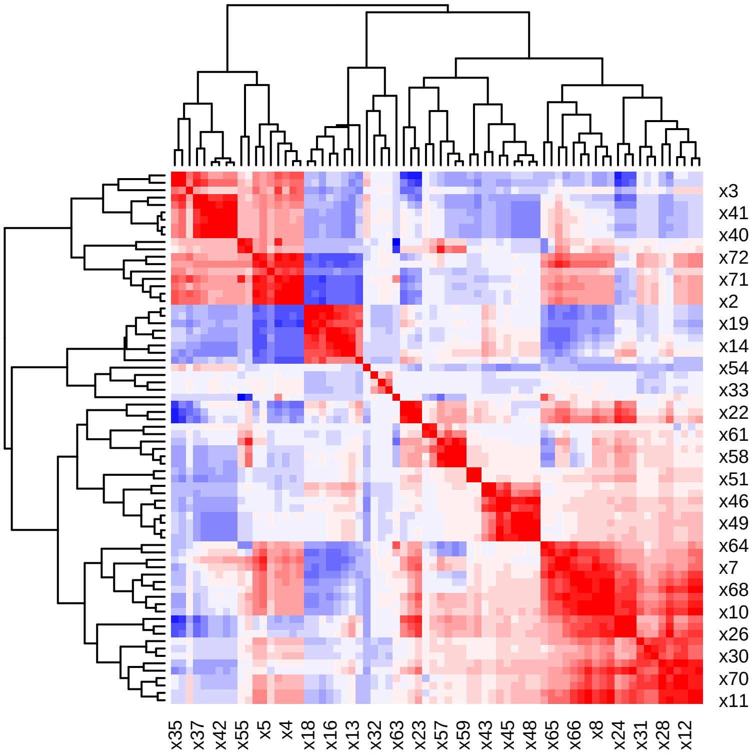

Figure 3: Heatmap of predictors correlation and hierarchical clustering

The `caret` function `findCorrelation` lets us identify quickly which variables are high correlated and can be removed. We used a cutoff for Pearson's correlation of 0.95. We find 26 predictors than can be removed and we are left with 46 predictors.

2.3.2 Correlation between predictors and outcome variable

The analysis of correlation between variables in an exploratory data analysis can also be useful to identify a subset of predictors that correlate with the outcome variable. This has two objectives: guiding the analysis and also finding a subset of predictors for model selection process. This is particularly useful in the case of high dimensional settings where a large number of predictors can severely hinder the model selection and estimation. Dimensionality reduction techniques can also be used to limit the number of predictors. In our case, we have only 47 predictors which is not such a large number and we decide not to reduce the dimension.

In our analysis, the outcome in a binary categorical variable and the predictors are continuous. A way to identify predictors that can be useful in predicting the two classes is through boxplots. A potentially good predictor is a predictor which distribution differs under the two classes. From all the predictors, four present visually different distributions under the two classes: `x35`, `x36`, `x37` and `x41`. and the boxplots are reproduced in Fig. 4.

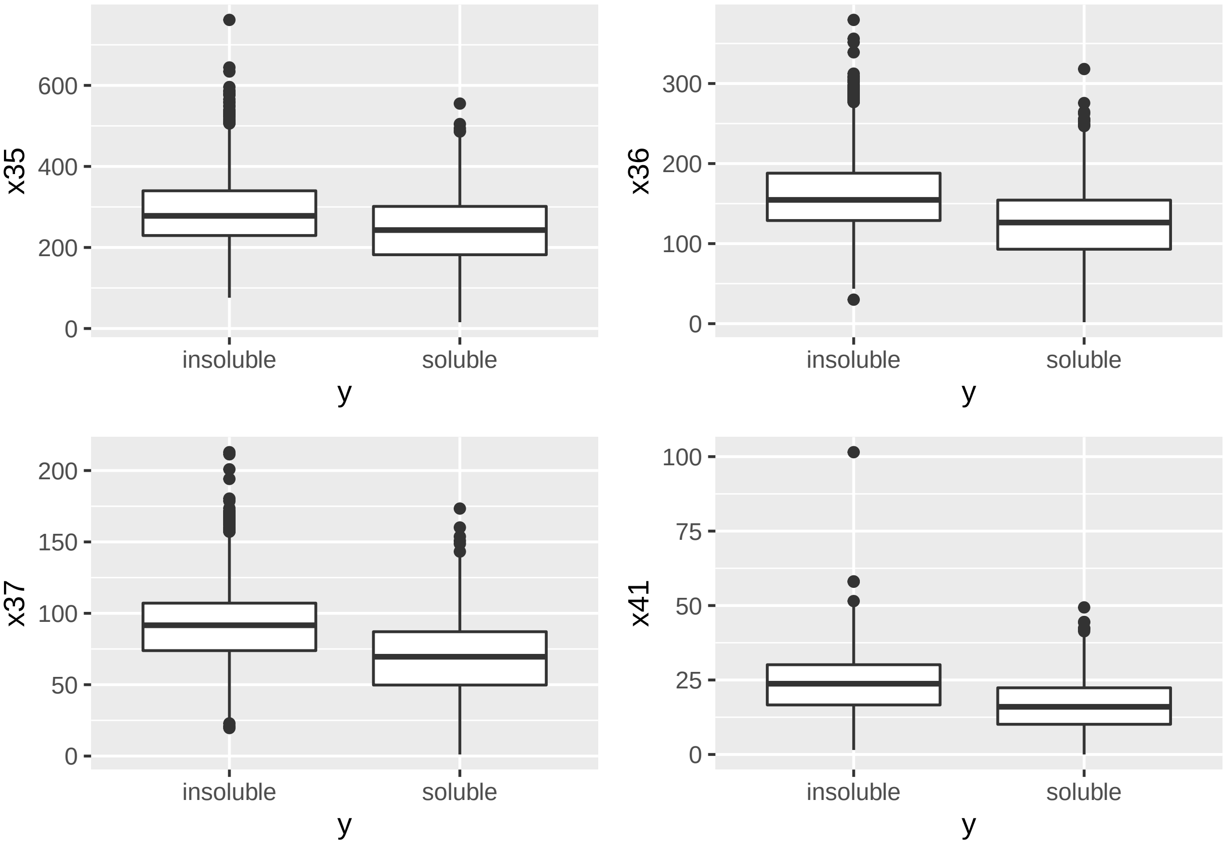

Figure 4: Boxplots of potential good predictors under the two classes

2.4 Missing data

The filtered dataset has no missing data. Before filtering for highly correlated data, the variable `x71` had missings but it has been filtered out. In any case, the presence of missing data is not an issue for tree based models when using package `rpart` (or `rpart` through `caret`) as a surrogate split is used for prediction whenever there is a missing data for a particular variable over which a split is made.

```
## Missing data per predictor
```

```
##  x1  x3  x4  x5  x6 x15 x16 x19 x20 x21 x22 x24 x27 x28 x29 x30
##   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## x31 x32 x33 x34 x35 x36 x37 x41 x43 x44 x45 x47 x49 x51 x53 x54
##   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## x55 x56 x57 x58 x60 x61 x62 x63 x64 x65 x67 x69 x70 x72   y
##   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

2.5 Outliers

A complete exploratory data analysis looks for atypical values in the dataset and, after inspection, decides on keeping or removing them. Although techniques such as dbscan exist, identifying outliers in multivariate datasets is challenging. Moreover we have no information on the nature of the predictors so it is difficult to tell whether statistically atypical values are real outliers, outside the range of possible or expectable values. Last but not least, trees are rather insensitive to outliers in predictors (Hastie, Tibshirani, and Friedman

2009, p352) and we have no outliers in the binary outcome variable. As a consequence, we decide to keep all the observations.

3 Split into training/test dataset

To split the data into balanced training and test sets we can use the caret function createDataPartition.

We check the training and test sets preserve the overall class distribution of the data.

```
## Training set class distribution
```

```
##
## insoluble   soluble
##      0.62      0.38
```

```
## Test set class distribution
```

```
##
## insoluble   soluble
##      0.62      0.38
```

4 Pruned tree

4.1 Fitting

We fit a single classification tree (CART) with rpart1SE method in caret. This implementation uses a Gini impurity function to chose the splits in the tree growth phase. The pruning is done by cost-complexity tuning. We use 5-fold cross-validation with the area under the ROC curve as performance metric to choose the cost-complexity parameter. The choice of a 5-fold cross-validation repeated 3 times as resampling method is driven by the will to pick a method that has little computational cost and that can be used over all the models that will be fitted in this work, such that the training error measures are comparable. Random forest and boosting models are computationally costly by themselves so a high cost resampling can make the fitting of those very lengthy. We chose the area under the ROC as performance metric because it is a more comprehensive performance metric than accuracy or Kappa, that is available for binary classification. Finally, rpart1SE applies the one-standard deviation rule to pick the final tree which results in a smaller tree.

4.2 Performance

4.2.1 Training error

We can compute a series of performance metric for the final model on the training set with the confusionMatrix function of caret. We see the model has good accuracy, significantly different from the non-informative rate. Taking ‘insoluble’ as the positive class, the sensitivity is better than the specificity.

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  insoluble soluble
##   insoluble      1431     362
##   soluble         315     707
##
##                Accuracy : 0.7595
##                  95% CI : (0.7433, 0.7752)
##     No Information Rate : 0.6202
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.4851
##
##  Mcnemar's Test P-Value : 0.07707
##
##             Sensitivity : 0.8196
##             Specificity : 0.6614
##          Pos Pred Value : 0.7981
##          Neg Pred Value : 0.6918
##              Prevalence : 0.6202
##          Detection Rate : 0.5083
##    Detection Prevalence : 0.6369
##       Balanced Accuracy : 0.7405
##
##        'Positive' Class : insoluble
##
```

We can also obtain the area under the ROC curve: 0.747.

4.2.2 Test error

We now compute the same performance metrics on the test set. We see the accuracy is a bit smaller but still significantly different from the non-informative rate. The Mcnemar's test p-value shows the classification is also significantly different from the best guess. Both sensitivity and specificity are smaller than in the training set but by a small amount. Sensitivity is again larger than the specificity.

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  insoluble soluble
##   insoluble      1371     435
##   soluble         376     634
##
##                Accuracy : 0.712
##                  95% CI : (0.6949, 0.7287)
##     No Information Rate : 0.6204
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.3819
##
##  Mcnemar's Test P-Value : 0.04168
##
##             Sensitivity : 0.7848
##             Specificity : 0.5931
##          Pos Pred Value : 0.7591
##          Neg Pred Value : 0.6277
##              Prevalence : 0.6204
##          Detection Rate : 0.4869
##    Detection Prevalence : 0.6413
##       Balanced Accuracy : 0.6889
##
##        'Positive' Class : insoluble
```

```
##
```

The area under the ROC curve is smaller than on the training set at 0.73.

5 Random forest

5.1 Fitting

We use `rf` method in `caret` that sources the implementation in package `randomForest`. We need to choose two tuning parameters, the number of trees and m_{try}, the number of randomly selected predictors to choose from at each split. In the `rf` method in `caret`, m_{try} is the only parameter marked as tuning parameter. So to implement the grid search, we used the inbuilt function `tuneGrid` for m_{try} and a loop over values of the number of trees. The computational cost in high so we ran the code in a separate Google colab using parallel computing (see 'Parallel Random Forest' attached). The code is reproduced in this report in the Appendix. We first tried equally spaced values of m_{try} between 2 and 45 and then reduced the range to 2 to 12 because the best m_{try} found was always among small values. We try values of 1000, 1500, 2000 and 2500 for the number of trees. Such values were chosen following the recommendations of Kuhn and Johnson (2013 p387). The grid for m_{try} includes the often recommended value of the square root of the number of predictors, approximately 7. The resampling applied for the parameters tuning is a 5-fold cross-validation repeated 3 times, consistent with the one used for the CART model. The performance metric is the area under the ROC.

The values in the grid for number of trees yielded very similar results (Fig.5). For all of them, the maximum AUC is attained for small values of m_{try}, 2 or 4. We noted than for initial runs of the random forest with the unfiltered set of predictors, the best m_{try} varied more across the number of trees values. This might be due to the confusion introduced in the best split selection by highly correlated predictors as explained in the exploratory data analysis. We see little to no improvement in the area under the ROC curve for number of trees larger than 1500.

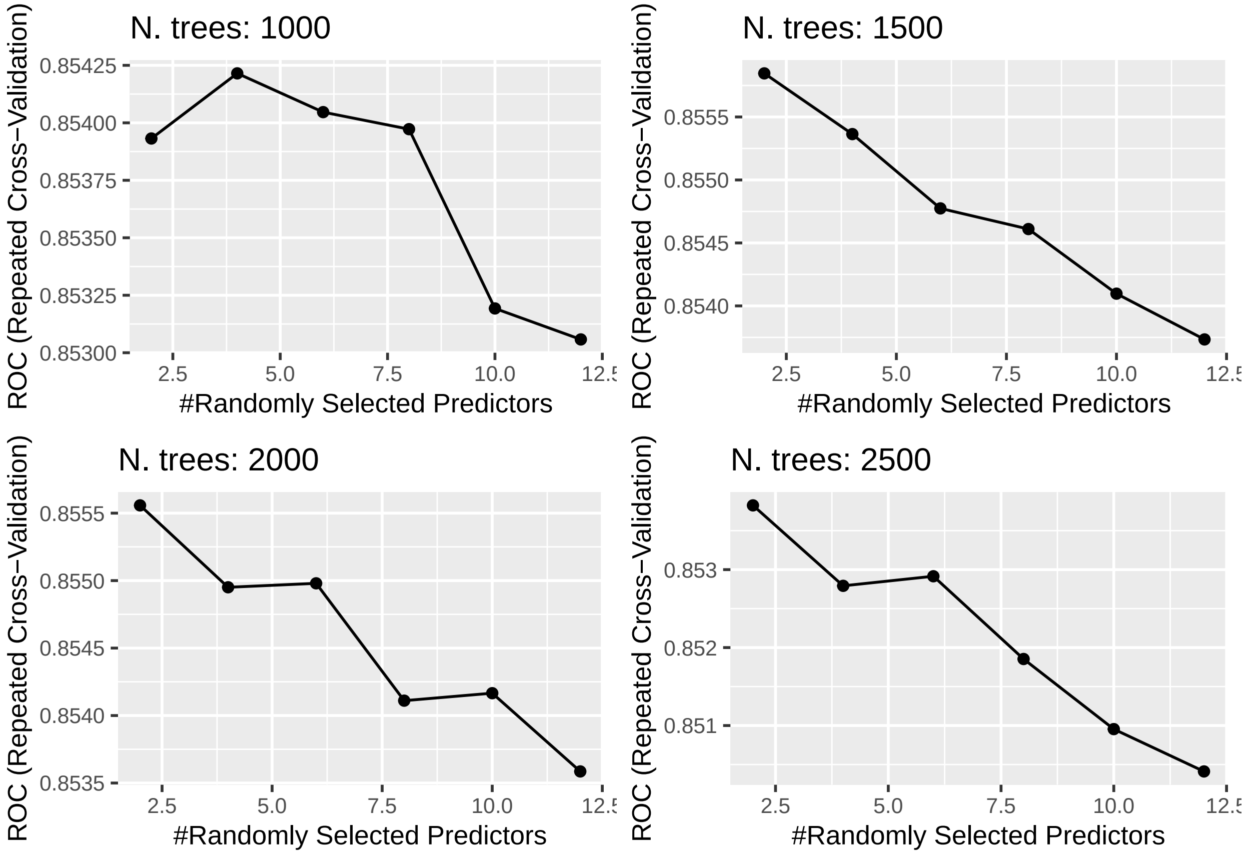

Figure 5: ROC in function of mtry and number of trees

We order the pairs of m_{try} and number of trees by ROC and present the top 10 ones in Table 1. The highest ROC is obtained for 1500 trees with m_{try} equal to 2. That is our final random forest model.

Table 1: Top 10 pairs of tuning parameters

mtry	Number of trees	ROC
2	1500	0.8558
2	2000	0.8556
4	1500	0.8554
6	2000	0.8550
4	2000	0.8550
6	1500	0.8548
8	1500	0.8546
4	1000	0.8542
10	2000	0.8542
8	2000	0.8541

5.2 Performance

5.2.1 Training error

We can compute a series of performance metric for the final model on the training set. We see the model has excellent accuracy, significantly different from the non-informative rate. Taking ‘insoluble’ as the positive class, both the sensitivity and specificity are very high.

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  insoluble soluble
##   insoluble      1745       0
##   soluble           1    1069
##
##                Accuracy : 0.9996
##                  95% CI : (0.998, 1)
##     No Information Rate : 0.6202
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9992
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9994
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9991
##              Prevalence : 0.6202
##          Detection Rate : 0.6199
##    Detection Prevalence : 0.6199
##       Balanced Accuracy : 0.9997
##
##        'Positive' Class : insoluble
##
```

We can also obtain the area under the ROC curve: 0.856

5.2.2 Test error

We now compute the same performance metrics on the test set. We see the accuracy is smaller but still significantly different from the non-informative rate. The Mcnemar’s test p-value shows the classification is also significantly different from the best guess. Both sensitivity and specificity are smaller than in the training set. Sensitivity is larger than the specificity.

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  insoluble soluble
##   insoluble      1581     438
##   soluble         166     631
##
##                Accuracy : 0.7855
##                  95% CI : (0.7699, 0.8005)
##     No Information Rate : 0.6204
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.521
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9050
##             Specificity : 0.5903
##          Pos Pred Value : 0.7831
##          Neg Pred Value : 0.7917
##              Prevalence : 0.6204
##          Detection Rate : 0.5614
##    Detection Prevalence : 0.7170
##       Balanced Accuracy : 0.7476
##
##        'Positive' Class : insoluble
##
```

The area under the ROC curve is larger than on the training set at 0.865

5.3 Variable importance

We can extract the variable importance using the `varImp` function of `caret`. For random forest models, the variable importance is computed as the difference in out-of-bag accuracy when permuting each predictor, standardized over all the trees. The values are scaled to have a maximum value of 100. We find that the three most important variables are three out of the 4 variables we had flagged as potentially good predictors in the exploratory data analysis: `x36`, `x37` and `x41`. We note that those variables have a large number of unique values: between 389 and 1609. Their observed importance is then not misleading.

```
## rf variable importance
##
##   only 20 most important variables shown (out of 46)
##
##     Overall
## x41  100.00
## x37   97.65
## x36   97.62
## x51   78.10
## x72   70.44
## x54   69.93
## x35   65.93
## x5    64.34
## x58   63.53
## x63   61.31
## x64   60.69
## x57   57.79
## x60   56.99
## x55   56.23
## x65   56.22
## x21   55.86
## x1    55.04
## x24   54.57
## x4    54.07
## x6    53.06
```

6 Comparison pruned single tree and random forest

6.1 Training error

The function `resamples` of `caret` lets us compare easily the performance of the two models on the training set. Resamples provide an estimate of the standard error of the performance metric for each model on the training set. First, we can visualize the distributions of the three performance metrics ROC, specificity and sensitivity over the cross-validation folds for each model (Fig. 6). We see the boxplots of the ROC are well separated with the median, the third and first quartiles of random forest ROC being superior to the single tree ones. The same occurs for the sensitivity. The boxplots of the specificity of the single tree and the random forest models overlap but the median of the random forest specificities is higher that the single tree one.

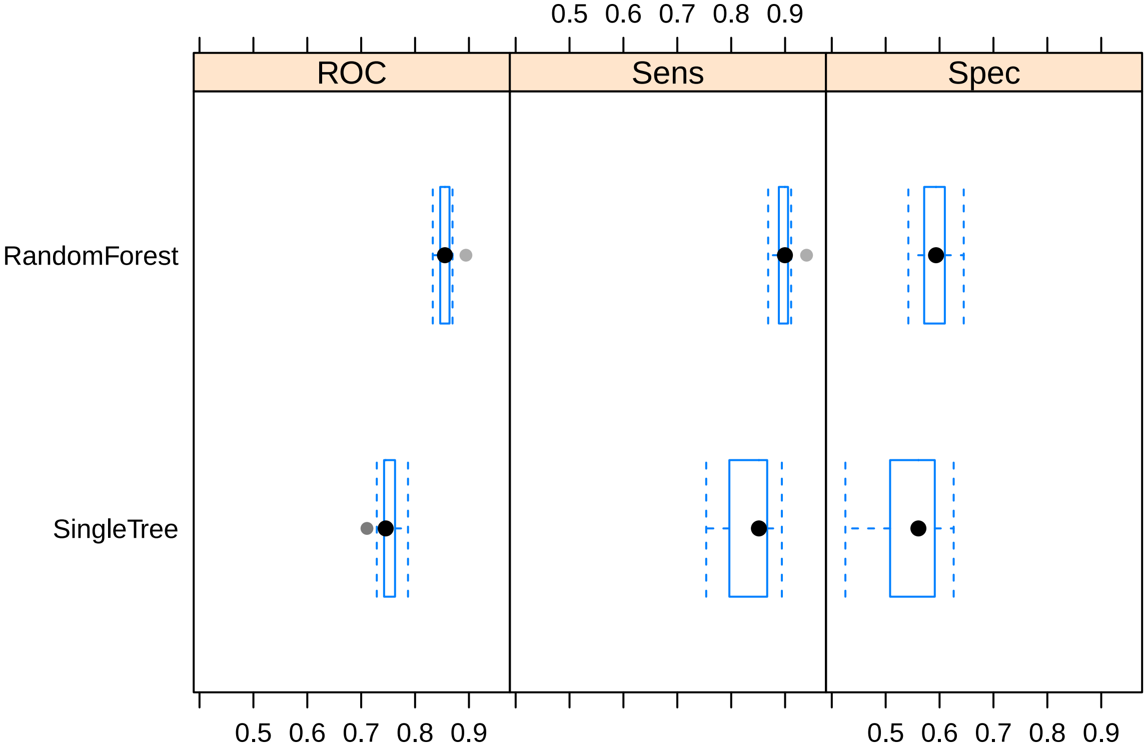

Figure 6: Boxplot of performance metric distributions

The average three metrics on the training set are always higher with random forest than for the single tree model. The 95% confidence interval of the ROC for each model do not overlap (Fig. 7). The same occurs with sensitivity. However, they do overlap in the case of specificity.

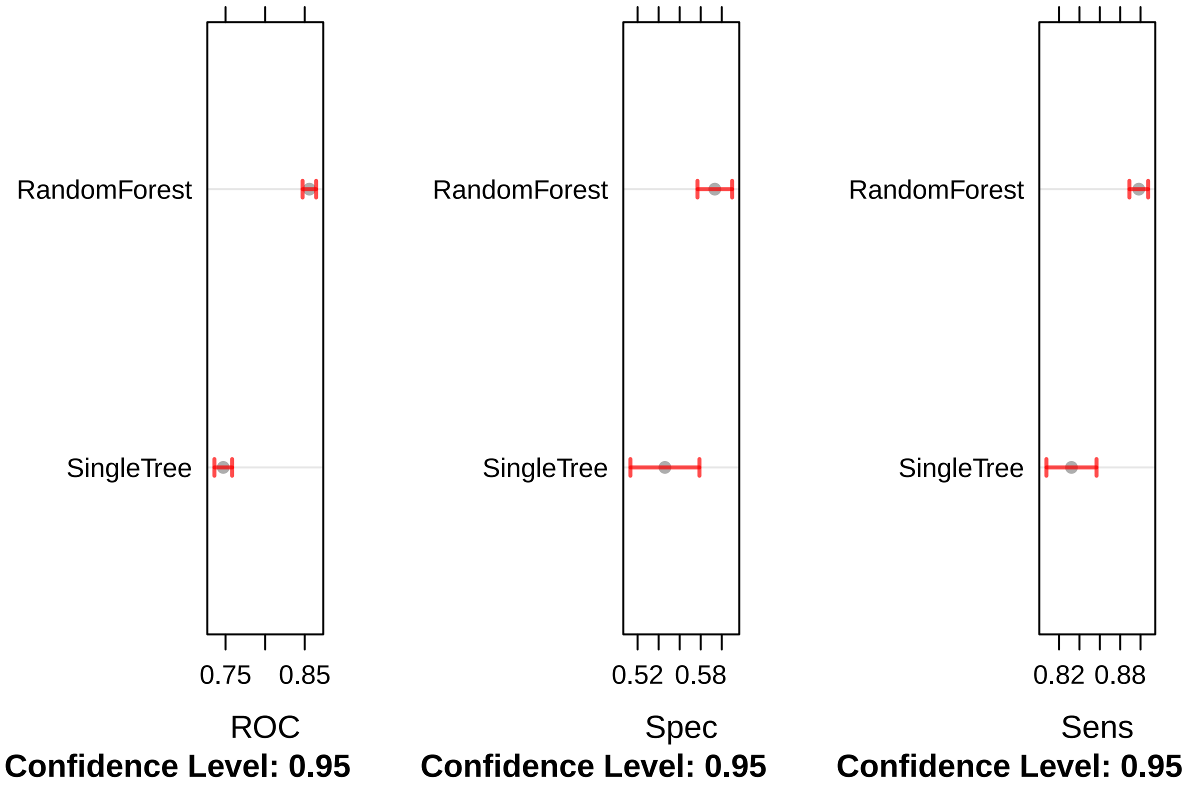

Figure 7: Confidence intervals of performance metrics

Finally, caret provides a p-value for the difference in each performance metric between the two models, assuming a Student's T distribution and performing a Bonferroni adjustment for multi-testing. On the training set, the differences are significant at 5% for the ROC, the sensitivity and the specificity.

```
##
## Call:
## summary.diff.resamples(object = diff_rf)
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## ROC
##             SingleTree RandomForest
## SingleTree             -0.1087
## RandomForest 2.214e-13
##
## Sens
##             SingleTree RandomForest
## SingleTree             -0.06608
## RandomForest 0.0001992
##
## Spec
##             SingleTree RandomForest
## SingleTree             -0.04743
## RandomForest 0.009222
```

6.2 Test error

From the performance metrics on test set reported above, we see the accuracy, ROC and sensitivity are higher for the random forest model than for the single tree. However, the specificity is almost the same between the two models on the test set.

6.3 Conclusion

Considering all the performance metrics, the random forest model performs better than the single tree model in general. However, if the main concern of the study is maximizing the number of true soluble component, that is the specificity (positive class is insoluble), both the random forest and the single tree models are valid choices as the difference in performance is almost null on the test set. Moreover, the random forest model is less interpretable and much more costly computationally than a single tree so for specificity the single tree model can be a good choice.

7 Gradient boosting - adaboost

7.1 Adaboost with stumps

7.1.1 Fitting

We use the `gbm` method in `caret` with `distribution` argument equal to "adaboost" to fit a adaboost model. We set the `interaction.depth` parameter at 1 to obtain an adaboost with stumps. We perform a grid search over 11 values of the number of trees between 1 and 2000 and 5 values of the shrinkage parameter between 0.0001 and 1. We leave the fourth tuning parameter the minimum number of observation in a terminal node at 10, the default value. We use 5-fold cross-validation repeated 3 times resampling for consistency in this work and computational cost. We used accuracy as performance metric. We do not evaluate the code within this markdown for its computation time. We ran the code in an independent Google colab (see 'Parallel Boosting Trees' attached). The code is reproduced in this report in the Appendix.

From the results of the grid search over the number of trees and shrinkage (8), the best shrinkage value is 0.1 as it gives the best accuracy almost uniformly across the number of trees values. It yields the best or second best for all number of trees. Furthermore, the accuracy with shrinkage at 0.1 reaches the top accuracy faster than with other shrinkage values. We retain 0.1 as value of shrinkage.

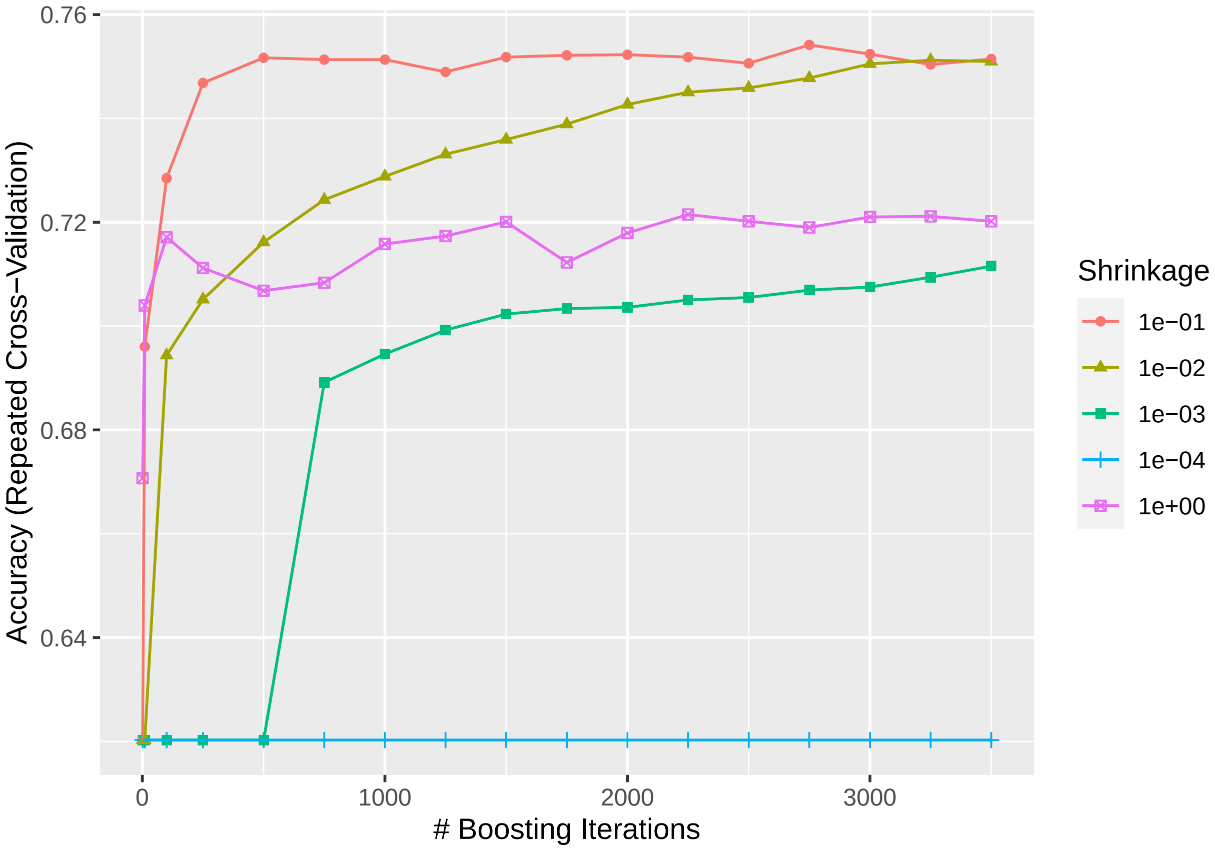

Figure 8: Accuracy in function of the number of trees and shrinkage on training set

7.1.2 Performance - Training error

We can compute the missclassification rate from the training result by computing 1-accuracy. We obtain a curve symmetric to the one we observed above for the accuracy at shrinkage = 0.1 (Fig. 9). The missclassification decreases sharply from 1 to 250 boosting iterations (number of trees) and then stagnates. The lowest values is obtained for 1250 boosting iterations.

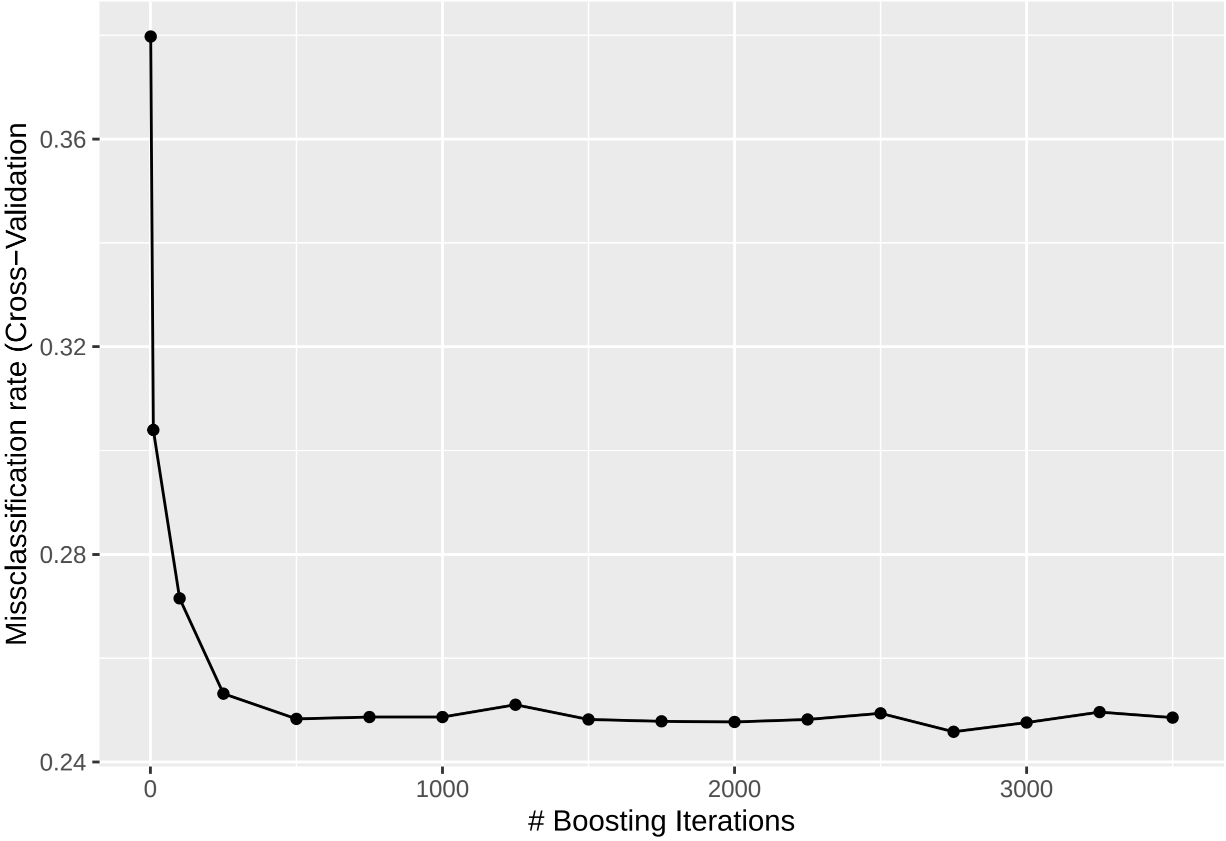

Figure 9: Missclassifaction rate in function of boosting iterations on training set

7.1.3 Performance - Test error

Combining the `update.train` and the `predict` functions of `caret` we can get the predictions on the test set from our `train` object for all the number of boosting trees values in the grid (Fig.10). We see the missclassification rate decreases as fast as the training error until 1000 iterations, then stagnates. The missclassification starts increasing from 2750 trees while the training error remains at its minimum level, indicating overfitting.

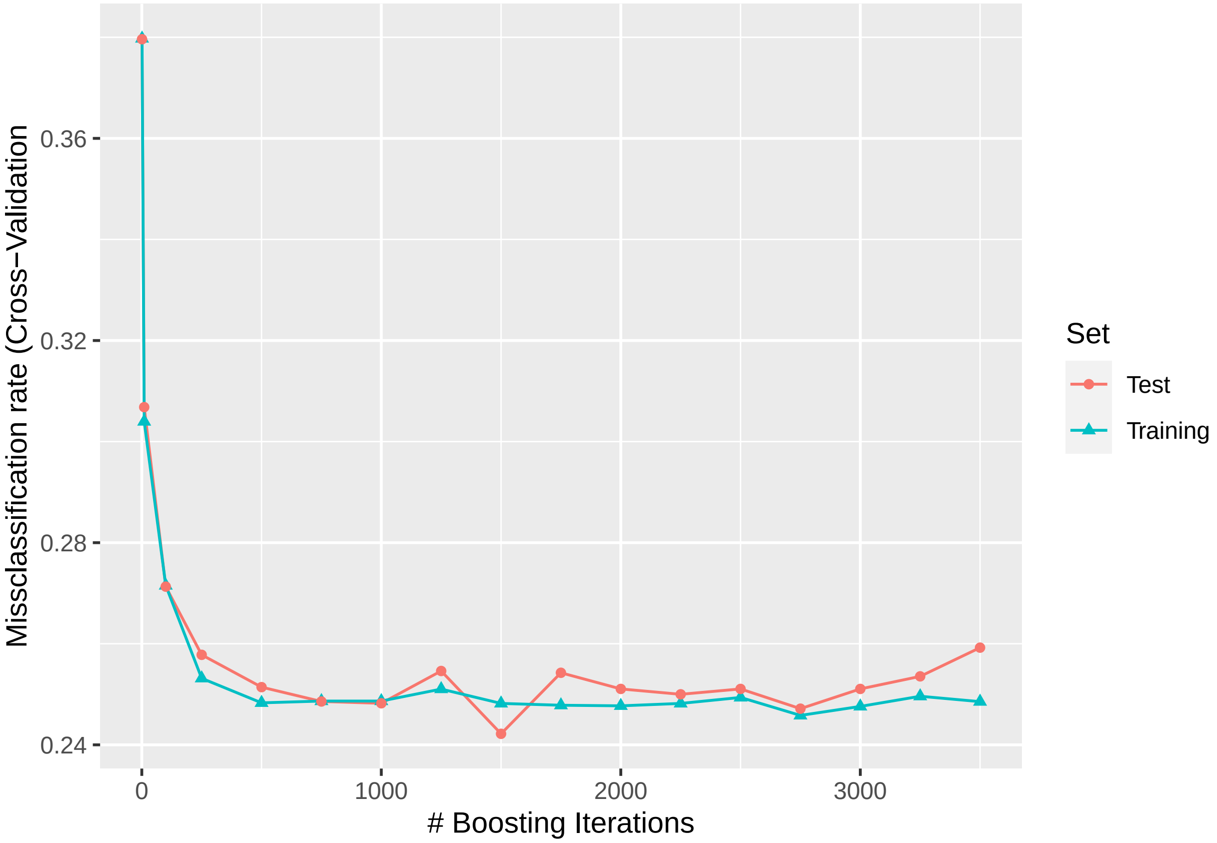

Figure 10: Missclassifaction rate in function of boosting iterations on test set

7.1.4 Increasing the number of boosting iterations in a single adaboost model

The study of the evolution of missclassification rate along the number of boosting iterations above has been carried out with `caret` which fits different instances of adaboost for the different values of the tuning parameter n.tree. With `gbm.more` from the `gbm` package, we can increase incrementally the number of tree of a single adaboost model to study the evolution of the missclassification rate. Fig 11 more clearly shows the overfitting as the number of iterations increases: the test missclassification rate stagnates while the training one keeps decreasing.

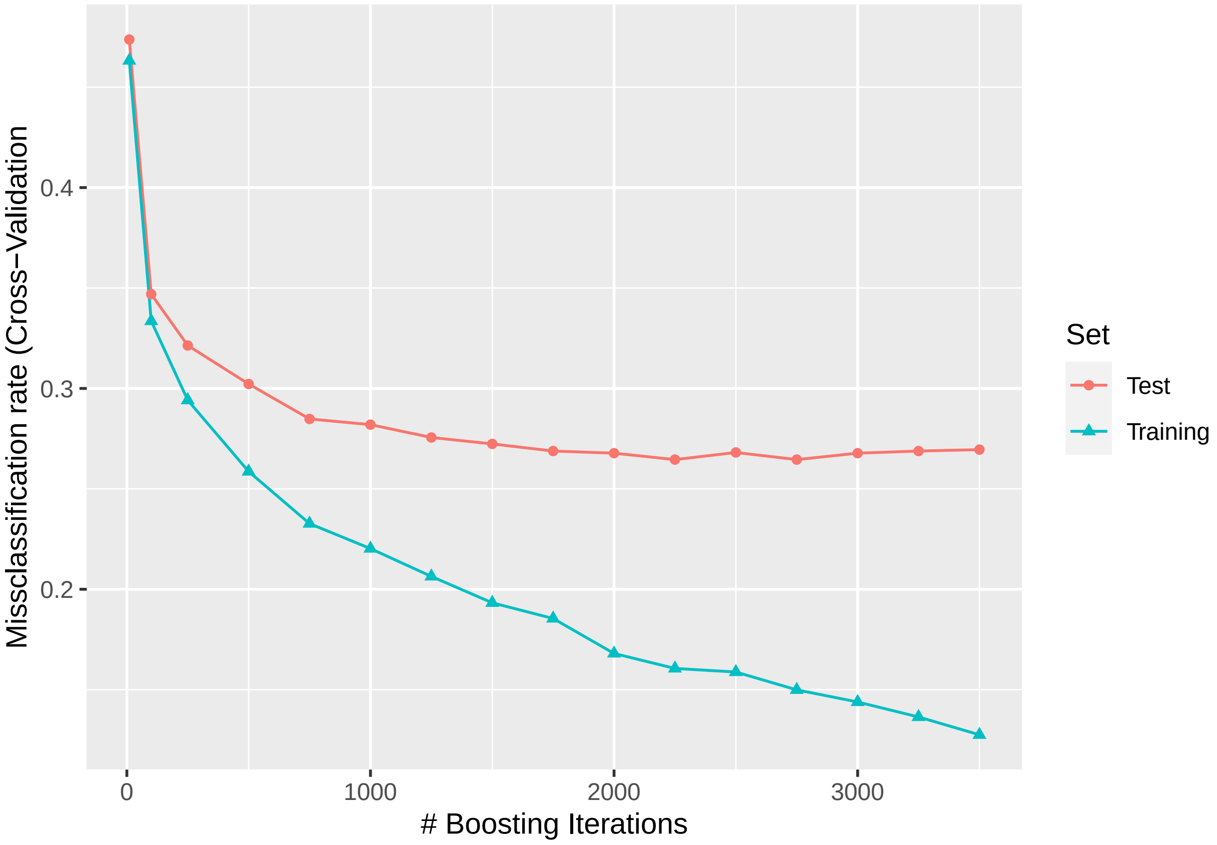

Figure 11: Missclassifaction rate with incremental boosting iterations on test set

7.2 Adaboost with larger interaction depth

7.2.1 Fitting

We use the same `gbm` method in `caret` with `distribution` argument equal to "adaboost" to fit an adaboost model. Instead of having the `interaction.depth` parameter set at 1, we include it in the grid search and iterate over the values of 1, 4, 8 and 16. We leave the fourth tuning parameter, the minimum number of observations in a terminal node, at the default value of 10. Again we uses 5-fold cross-validation repeated 3 times as resampling method. We ran the code in an independent Google colab (see 'Parallel Boosting Trees' attached). The code is reproduced in this report in the Appendix.

We see from the grid search output (Fig. 12) that the value of 0.1 for the shrinkage in the best across all interaction depths. In all the cases it is the best or the second best across all the values of number of boosting iterations and interaction depth. Accuracy with shrinkage at 0.1 reaches its top value faster than with other shrinkage values. However for large numbers of boosting iterations and large interaction depths, there is little to no difference in the accuracy between shrinkage values of 0.1 and 0.01. We choose a shrinkage value of 0.1 for all the the interaction depths and number of boosting iterations.

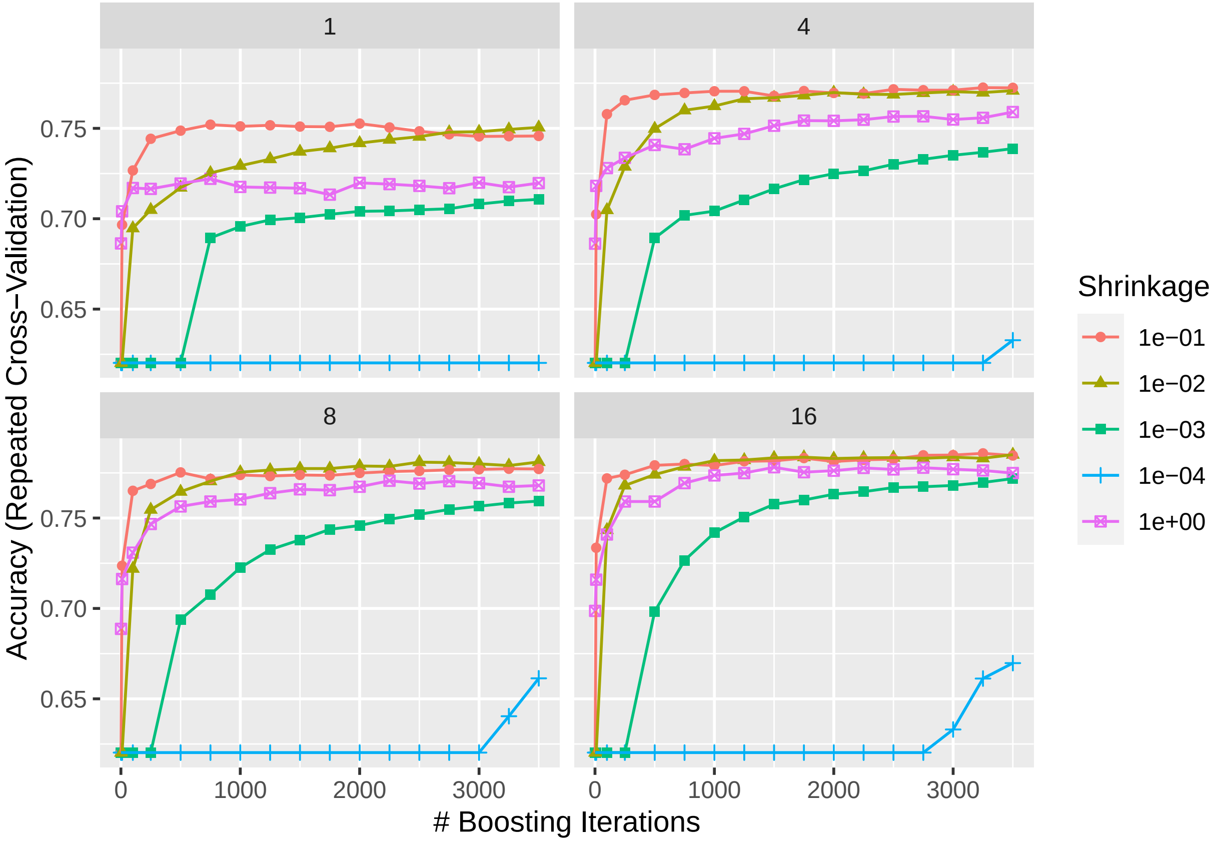

Figure 12: Accuracy in function of boosting iterations, shrinkage and interaction depth on training set

7.2.2 Test error

Making use of the `update.train` and the `predict` functions of `caret` again, we get the predictions on the test set from our `train` object for all the number of boosting trees and interaction depth values in the grid (Fig.13). The test error is lower with maximum depth of 16 across all values of the number of boosting iterations.

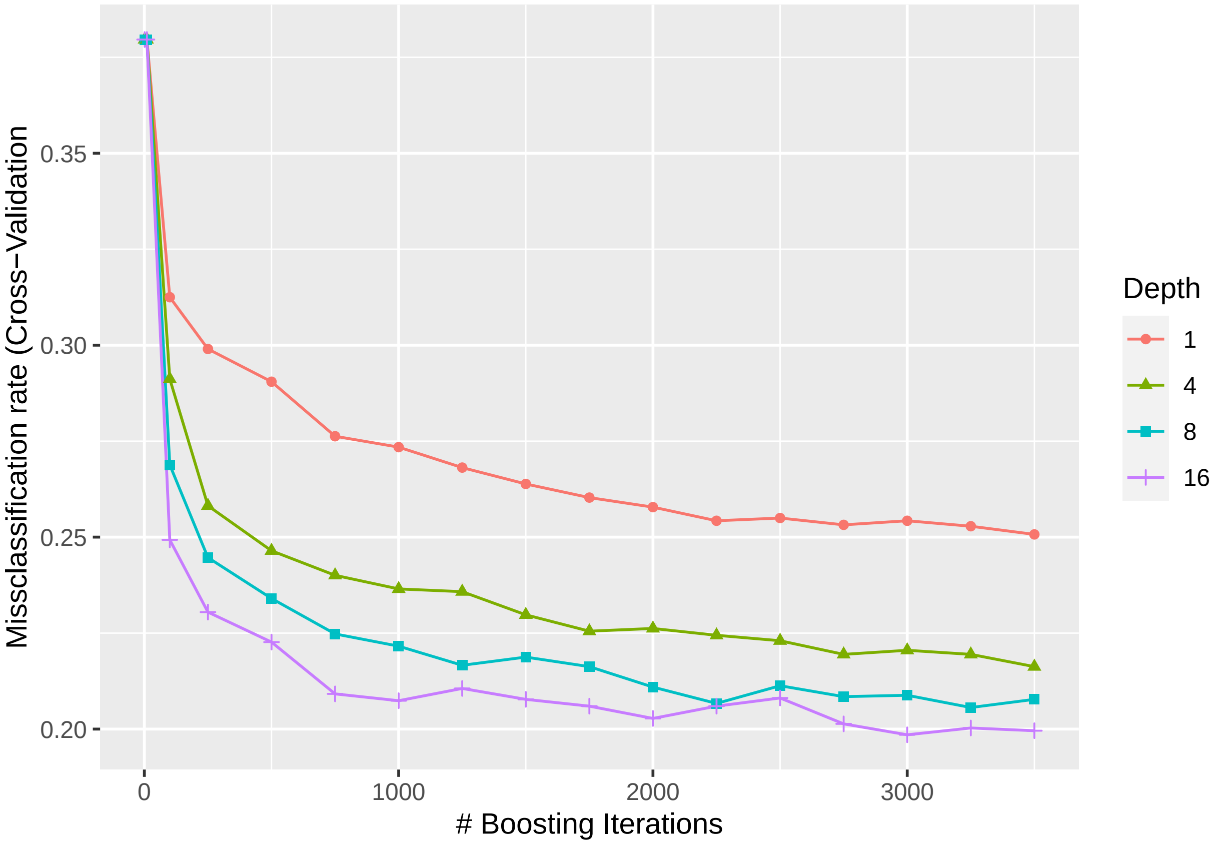

Figure 13: Missclassifaction rate in function of boosting iterations and depth on test set

8 Appendix

```
knitr::opts_chunk$set(
  echo = FALSE, warning = FALSE, error = FALSE, message = FALSE,
  tidy.opts = list(width.cutoff = 55)
)

# libraries

# If the package is not installed then it will be installed
if (!require("rpart")) install.packages("rpart")
if (!require("gbm")) install.packages("gbm")
if (!require("readr")) install.packages("readr")
if (!require("knitr")) install.packages("knitr")
if (!require("caret")) install.packages("caret")
if (!require("gridExtra")) install.packages("gridExtra")
if (!require("ggplot2")) install.packages("ggplot2")

require("readr")
require("tidyverse")
require("caret")
require("gridExtra")
```

```r
require("ggplot2")
require("gbm")

# data parsing
soldat <- read_csv("soldat.csv")
soldat$y <- factor(soldat$y, levels = c(-1, 1), labels = c("insoluble", "soluble"))
n <- nrow(soldat)
p_start <- ncol(soldat)

##################################
#  Exploratory data analysis
#################################

# class frequency
table(soldat$y)


# predictors variance
bxp1 <- soldat %>%
  select(-y) %>%
  select(num_range("x", c(1:18))) %>%
  stack() %>%
  ggplot(aes(x = ind, y = values)) +
  geom_boxplot() +
  xlab("")

bxp2 <- soldat %>%
  select(-y) %>%
  select(num_range("x", c(19:34))) %>%
  stack() %>%
  ggplot(aes(x = ind, y = values)) +
  geom_boxplot() +
  xlab("")

bxp3 <- soldat %>%
  select(-y) %>%
  select(num_range("x", c(35:50))) %>%
  stack() %>%
  ggplot(aes(x = ind, y = values)) +
  geom_boxplot() +
  xlab("")

bxp4 <- soldat %>%
  select(-y) %>%
  select(num_range("x", c(55:72))) %>%
  stack() %>%
  ggplot(aes(x = ind, y = values)) +
  geom_boxplot() +
  xlab("")

grid.arrange(bxp1, bxp2, bxp3, bxp4, ncol = 2)
```

```r
# predictors granularity / number of unique values
nb_unique_val <- function(x) {
  return(length(unique(x)))
}

nb_unique_val_pred <- tibble(nb = sapply(soldat %>% select(-y), nb_unique_val), predictor = colnames(sol

ggplot(nb_unique_val_pred) +
  geom_histogram(aes(x = nb), col = "grey", bins = 100) +
  geom_vline(xintercept = 300, col = "red") +
  xlab("Number of unique values")


# predictors correlation heatmap
v <- cor(soldat %>% select(-y), use = "pairwise.complete.obs")
col <- colorRampPalette(c("blue", "white", "red"))(20)
heatmap(v, col = col, symm = TRUE, Colv = "Rowv")

# filtering of highly correlated predictors
high_corr <- colnames(soldat)[findCorrelation(v, cutoff = 0.95)]
soldat <- soldat %>% select(-all_of(high_corr))

# covariation of outcome and selected predictors
p <- list()
i <- 1
for (v in c("x35", "x36", "x37", "x41")) {
  p[[i]] <- ggplot(data = soldat) +
    geom_boxplot(aes_string(x = "y", y = v))
  i <- i + 1
}
do.call(grid.arrange, list(grobs = p, ncol = 2))

# number of missing data
cat("Missing data per predictor\n")
apply(is.na(soldat), 2, sum)

##################################
#  data partitioning
#################################

set.seed(1234)
inTest <- createDataPartition(soldat$y, p = 0.5, list = FALSE)[, 1]
test <- soldat[inTest, ]
training <- soldat[-inTest, ]

# check for balance training and test sets
cat("Training set class distribution\n")
round(table(training[, "y"]) / nrow(training), 2)
cat("Test set class distribution\n")
round(table(test[, "y"]) / nrow(test), 2)

##################################
#  CART model
```

```
#################################

# fitting
ctrl <- trainControl(
  method = "repeatedcv",
  number = 5,
  repeats = 3,
  classProbs = TRUE,
  summaryFunction = twoClassSummary
)
CART <- train(y ~ ., data = training, method = "rpart1SE", trControl = ctrl, metric = "ROC")

# training error
pred_train_CART <- predict.train(CART, type = "raw")
confusionMatrix(data = pred_train_CART, reference = training$y)

# test error
pred_CART <- predict(CART, newdata = test)

confusionMatrix(data = pred_CART, reference = test$y)

df <- data.frame(obs = test$y, pred = pred_CART, predict(CART, newdata = test, type = "prob"))

## ##################################
## #  Random forest
## #################################
##
## # fitting (not evaluated inside R markdown but in seperate colab 'Parallel Random Forest')
## control <- trainControl(
##   method = "repeatedcv",
##   number = 5,
##   repeats = 3,
##   search = "grid",
##   classProbs = TRUE,
##   allowParallel = TRUE,
##   summaryFunction = twoClassSummary
## )
##
## metric <- "ROC"
## tunegrid <- expand.grid(mtry = seq(2, 45, 5)) # then seq(2,12,2)
## rf_models <- list()
##
##
## i <- 1
## for (ntree in c(1000, 1500, 2000, 2500)) {
##   cat(paste0("Training with ", ntree, " trees ..."))
##   rf_models[[i]] <- train(y ~ .,
##     data = training,
##     method = "rf",
##     metric = metric,
##     tuneGrid = tunegrid,
##     trControl = control,
##     ntree = ntree
##   )
##   i <- i + 1
## }
## save(rf_models, file = "rf_models.Rdata")

# loading 'Parallel random forest' output
load("rf_models.Rdata")

# parameters turning
n_trees <- c(1000, 1500, 2000, 2500)
p <- list()
for (i in 1:4) {
  p[[i]] <- ggplot(rf_models[[i]]) +
    ggtitle(paste("N. trees:", n_trees[i])) +
    theme(text = element_text(size = 10))
}
do.call(grid.arrange, list(grobs = p, ncol = 2))

df <- rbind(
  rf_models[[1]]$results[, 1:2],
  rf_models[[2]]$results[, 1:2],
  rf_models[[3]]$results[, 1:2],
  rf_models[[4]]$results[, 1:2]
)
df$n_trees <- c(rep(1000, 6), rep(1500, 6), rep(2000, 6), rep(2500, 6))

df <- df[, c(1, 3, 2)] %>%
  arrange(-ROC) %>%
  slice(1:10)
kable(df,
  digits = c(0, 0, 4),
  col.names = c("mtry", "Number of trees", "ROC"),
  align = "c",
  caption = "Top 10 pairs of tuning parameters"
)

RF <- rf_models[[2]]

# training error
pred_train_RF <- predict.train(RF, type = "raw")
confusionMatrix(data = pred_train_RF, reference = training$y)

# test error
pred_RF <- predict(RF, newdata = test)

confusionMatrix(data = pred_RF, reference = test$y)

df <- data.frame(obs = test$y, pred = pred_RF, predict(RF, newdata = test, type = "prob"))

# variable importance
varImp(RF)

##################################
```

```
#  Comparison CART and random forest
#################################

# boxplot performance on training set
resamps <- resamples(list(
  SingleTree = CART,
  RandomForest = RF
))

theme1 <- trellis.par.get()
theme1$plot.symbol$col <- rgb(.2, .2, .2, .4)
theme1$plot.symbol$pch <- 16
theme1$plot.line$col <- rgb(1, 0, 0, .7)
theme1$plot.line$lwd <- 2
trellis.par.set(theme1)
bwplot(resamps, layout = c(3, 1))

# confidence intervals for performance metrics on training set
metric <- c("ROC", "Spec", "Sens")
p <- list()
for (i in 1:3) {
  p[[i]] <- dotplot(resamps, metric = metric[i])
}
do.call(grid.arrange, list(grobs = p, ncol = 3))

# pvalue for difference in perf. metrics on training set

diff_rf <- diff(resamps)
summary(diff_rf)

## ##################################
## #  Adaboost with stumps
## #################################
##
## # fitting (not evaluated inside R markdown but in seperate colab 'Parallel Boosting Trees')
## control <- trainControl(
##   method = "repeatedcv",
##   number = 5,
##   repeats = 3,
##   returnResamp = "all",
##   classProbs = TRUE
## )
##
## metric <- "Accuracy"
##
## grid <- expand.grid(
##   .interaction.depth = 1,
##   .n.trees = c(
##     1, 10, 100, 250, 500, 750, 1000, 1250, 1500,
##     1750, 2000, 2250, 2500, 2750, 3000, 3250, 3500
##   ),
##   .shrinkage = c(.0001, .001, .01, .1, 1),
##   .n.minobsinnode = 10
## )
##
## stumps_2000_model <- train(y ~ .,
##   data = training,
##   method = "gbm",
##   bag.fraction = 0.5,
##   distribution = "adaboost",
##   trControl = control,
##   tuneGrid = grid,
##   verbose = FALSE,
##   metric = metric
## )
##
## save(stumps_2000_model, file = "stumps_2000_model.Rdata")

# loading output of 'Parallel Boosting Trees'
load("stumps_2000_model.Rdata")

# tuning parameter grid search plot
ggplot(stumps_2000_model)

# training missclassification rate as a function of n.trees
df_res_ada_train <- stumps_2000_model$results %>% filter(shrinkage == 0.1)
df_res_ada_train$missclassification <- 1 - df_res_ada_train$Accuracy

ggplot(df_res_ada_train) +
  geom_line(aes(x = n.trees, y = missclassification)) +
  geom_point(aes(x = n.trees, y = missclassification)) +
  xlab("# Boosting Iterations") +
  ylab("Missclassification rate (Cross-Validation")

## # test missclassification rate as a function of n.trees
## # not evaluated in the Rmarkdown
## boosting_iter <- c(1, 10, 100, 250, 500, 750, 1000, 1250, 1500, 1750, 2000, 2250, 2500, 2750, 3000,
## miss_rate <- c()
## for (i in 1:length(boosting_iter)) {
##   model <- update(stumps_2000_model, list(
##     interaction.depth = 1,
##     n.trees = boosting_iter[i],
##     shrinkage = .1,
##     n.minobsinnode = 10
##   ))
##   pred_adaboost <- predict(model, newdata = test)
##   miss_rate <- c(miss_rate, mean(pred_adaboost != test$y))
## }
##
## df_res_ada_test <- data.frame(n.trees = boosting_iter, missclassification = miss_rate)
##
## df_res_ada <- rbind(
##   df_res_ada_train[, c("n.trees", "missclassification")],
##   df_res_ada_test
## )
## df_res_ada$Set <- c(
##   rep("Training", nrow(df_res_ada_train), ),
##   rep("Test", nrow(df_res_ada_test))
## )
## save(df_res_ada, file = "df_res_ada.Rdata")

load("df_res_ada.Rdata")
ggplot(df_res_ada) +
  geom_line(aes(x = n.trees, y = missclassification, col = Set)) +
  geom_point(aes(x = n.trees, y = missclassification, col = Set, pch = Set)) +
  xlab("# Boosting Iterations") +
  ylab("Missclassification rate (Cross-Validation")

## #### Train and test missclassification rates as a function of n.trees with gbm.more
##
## # not evaluated in Rmarkdown
##
## # recodifying y for gbm adaboost
## train_gbm <- training
## train_gbm$y <- as.numeric(training$y)
## train_gbm <- train_gbm %>% mutate(y = case_when(y == 2 ~ 0, TRUE ~ 1))
##
## test_gbm <- test
## test_gbm$y <- as.numeric(test$y)
## test_gbm <- test_gbm %>% mutate(y = case_when(y == 2 ~ 0, TRUE ~ 1))
##
## # first gbm with 10 iterations
## b_start <- 10
## ada <- gbm(y ~ .,
##   distribution = "adaboost", data = train_gbm,
##   n.trees = 10, interaction.depth = 1, n.minobsinnode = 10,
##   shrinkage = 0.1, cv.folds = 15
## )
## pred_ada_test <- as.numeric(predict.gbm(ada, newdata = test_gbm, n.trees = ada$n.trees) > 0.5)
## miss_rate_test <- mean(pred_ada_test != test_gbm$y)
##
## pred_ada_train <- as.numeric(predict.gbm(ada, newdata = train_gbm, n.trees = ada$n.trees) > 0.5)
## miss_rate_train <- mean(pred_ada_train != train_gbm$y)
##
## # updating gbm with more iterations
## for (diff_b in diff(boosting_iter)[-1]) {
##   ada <- gbm.more(ada, n.new.trees = diff_b, data = train_gbm)
##   pred_ada_test <- as.numeric(predict.gbm(ada, newdata = test_gbm, n.trees = ada$n.trees) > 0.5)
##   miss_rate_test <- c(miss_rate_test, mean(pred_ada_test != test_gbm$y))
##
##   pred_ada_train <- as.numeric(predict.gbm(ada, newdata = train_gbm, n.trees = ada$n.trees) > 0.5)
##   miss_rate_train <- c(miss_rate_train, mean(pred_ada_train != train_gbm$y))
## }
##
## # data frame of missclassification rate
## df <- data.frame(
##   n.trees = rep(boosting_iter[-1], 2),
##   Set = c(rep("Training", length(boosting_iter[-1])), rep("Test", length(boosting_iter[-1]))),
##   missclassification = c(miss_rate_train, miss_rate_test)
```

```r
## )
## save(df, file = "df_res_ada_incre.Rdata")

load("df_res_ada_incre.Rdata")
# plot
ggplot(df) +
  geom_line(aes(x = n.trees, y = missclassification, col = Set)) +
  geom_point(aes(x = n.trees, y = missclassification, col = Set, pch = Set)) +
  xlab("# Boosting Iterations") +
  ylab("Missclassification rate (Cross-Validation")

## ##################################
## #  Adaboost with larger interaction depth
## #################################
##
## # fitting (not evaluated inside R markdown but in seperate colab 'Parallel Boosting Trees')
## grid <- expand.grid(
##   .interaction.depth = c(1, 4, 8, 16),
##   .n.trees = c(
##     1, 10, 100, 250, 500, 750, 1000, 1250, 1500,
##     1750, 2000, 2250, 2500, 2750, 3000, 3250, 3500
##   ),
##   .shrinkage = c(.0001, .001, .01, .1, 1),
##   .n.minobsinnode = 10
## )
##
## multi_lvl_model <- train(y ~ .,
##   data = training,
##   method = "gbm",
##   bag.fraction = 0.5,
##   distribution = "adaboost",
##   trControl = control,
##   tuneGrid = grid,
##   verbose = FALSE,
##   metric = metric
## )
##
## multi_lvl_model
##
## save(multi_lvl_model, file = "multi_lvl_model.Rdata")

# loading output from 'Parallel Boosting Trees'
load("multi_lvl_model.Rdata")

# tuning parameters grid search plot
ggplot(multi_lvl_model)

## # test missclassification rate as a function of n.trees
## # not evaluated in the Rmarkdown
## boosting_iter <- c(
##   1, 10, 100, 250, 500, 750, 1000, 1250, 1500,
##   1750, 2000, 2250, 2500, 2750, 3000, 3250, 3500
## )
## depth <- c(1, 4, 8, 16)
## miss_rate <- c()
## depth_ord <- c()
## boosting_iter_ord <- c()
##
## for (i in 1:length(boosting_iter)) {
##   for (j in 1:length(depth)) {
##     model <- update(multi_lvl_model, list(
##       interaction.depth = depth[j],
##       n.trees = boosting_iter[i],
##       shrinkage = .01,
##       n.minobsinnode = 10
##     ))
##     pred_adaboost <- predict(model, newdata = test)
##     miss_rate <- c(miss_rate, mean(pred_adaboost != test$y))
##     boosting_iter_ord <- c(boosting_iter_ord, boosting_iter[i])
##     depth_ord <- c(depth_ord, depth[j])
##   }
## }
##
## df_res_ada_test_depth <- data.frame(
##   n.trees = boosting_iter_ord,
##   missclassification = miss_rate,
##   depth = depth_ord
## )
##
## df_res_ada_test_depth$depth <- as.factor(df_res_ada_test_depth$depth)
## save(df_res_ada_test_depth, file = "df_res_ada_test_depth.Rdata")

load("df_res_ada_test_depth.Rdata")

ggplot(df_res_ada_test_depth) +
  geom_line(aes(x = n.trees, y = missclassification, col = depth)) +
  geom_point(aes(x = n.trees, y = missclassification, col = depth, pch = depth)) +
  xlab("# Boosting Iterations") +
  ylab("Missclassification rate (Cross-Validation") +
  labs(col = "Depth", pch = "Depth")

## NA
```

9 Bibliography

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *Elements of Statistical Learning*. Springer.

Kuhn, Max, and Kjell Johnson. 2013. *Applied Predictive Modeling*. Springer.