Random Forests tutorial using R

Vanessa McNealis

11/10/2021

Introduction

In this tutorial, you will learn how to build a predictive model using random forests. RStudio is the interface for R which allows you to write your codes and compile them into the console. For this tutorial, you will need to have installed R, RStudio, and various packages

Load dependencies

```r
#install.packages(c("randomForest", "tidyverse", "caret",
#                  "ranger", "pmsampsize", "rms"))
library(randomForest)
library(tidyverse)
library(reshape2)
library(caret)
library(ranger)
library(pmsampsize)
library(rms)
```

Data description

The dataset consists of measurements of fetal heart rate (FHR) and uterine contraction (UC) features on cardiotocograms classified by expert obstetricians.

Attribute Information:

- LB: FHR baseline (beats per minute)
- AC: # of accelerations per second
- FM: # of fetal movements per second
- UC: # of uterine contractions per second
- DL: # of light decelerations per second
- DS: # of severe decelerations per second
- DP: # of prolongued decelerations per second
- ASTV: percentage of time with abnormal short term variability
- MSTV: mean value of short term variability
- ALTV: percentage of time with abnormal long term variability
- MLTV: mean value of long term variability
- Width: width of FHR histogram
- Min: minimum of FHR histogram
- Max: maximum of FHR histogram

- Nmax: # of histogram peaks
- Nzeros: # of histogram zeros
- Mode: histogram mode
- Mean: histogram mean
- Median: histogram median
- Variance: histogram variance
- Tendency: histogram tendency
- NSP: fetal state class code (N=normal; S=suspect; P=pathologic)

Read-in the data

When you read your data, make sure you are using the correct working directory. You can change your working directory using the setwd command. In this tutorial, I am using my working directory to allow the codes to work. Make sure that you replace the directory path that I'm using with your directory path when you perform these tutorials for yourself.

```
setwd("C:/Users/Vanessa/OneDrive - McGill University/McGill - PhD/CMDO AI Workshop")

# Read the data into R.
# The data is saved as a CSV file.
CTG <- read.csv("CTG.csv", sep = ";")

# For simplicity, I am selecting a subset of the variables in the
# data set.
# I am using the function select from the dplyr package,
# hence the dplyr::select syntax.
CTG <- CTG %>% dplyr::select(NSP, LB, AC, FM, UC, DL, DS, DP, ASTV,
                             MSTV, ALTV, MLTV)
head(CTG) # Shows the first six entries
```

```
##   NSP  LB AC FM UC DL DS DP ASTV MSTV ALTV MLTV
## 1   2 120  0  0  0  0  0  0   73  0.5   43  2.4
## 2   1 132  4  0  4  2  0  0   17  2.1    0 10.4
## 3   1 133  2  0  5  2  0  0   16  2.1    0 13.4
## 4   1 134  2  0  6  2  0  0   16  2.4    0 23.0
## 5   1 132  4  0  5  0  0  0   16  2.4    0 19.9
## 6   3 134  1  0 10  9  0  2   26  5.9    0  0.0
```

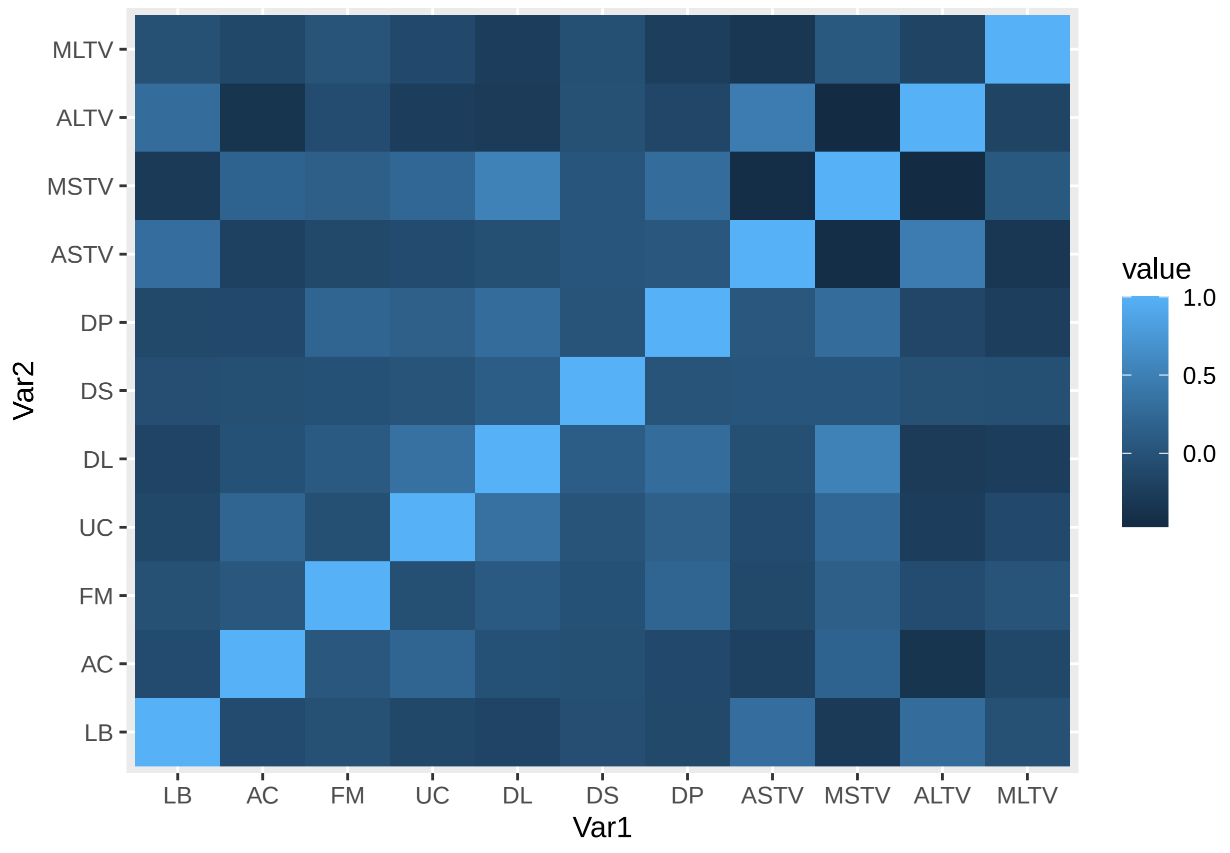
Correlation heatmap for the predictors

In this chunk, we investigate associations between the candidate predictors, while avoiding looking at the associations between the predictors and the outcome, which would lead us to make data-driven decisions and increase the risk of over-fitting.

```
# Create a data set for the correlation heatmap that is
# deprived of the outcome variable
cor.dat <- CTG %>% dplyr::select(-NSP)
cormat <- round(cor(cor.dat),2)
# The function melt(.) (from reshape2) converts the data to
# a long (i.e., tidy) format required for ggplot2
melted_cormat <- melt(cormat)
head(melted_cormat)
```

```
##   Var1 Var2 value
## 1   LB   LB  1.00
## 2   AC   LB -0.08
## 3   FM   LB -0.03
## 4   UC   LB -0.13
## 5   DL   LB -0.16
## 6   DS   LB -0.05
```

```
ggplot(data = melted_cormat, aes(x=Var1, y=Var2, fill=value)) +
  geom_tile()
```



Preliminary steps

```
# Create a binary variable indicating abnormal (suspect or pathological) fetal state
CTG <- CTG %>% mutate(ABNORMAL = case_when(NSP == 1 ~ 0,
                                           NSP == 2 ~ 1,
                                           NSP == 3 ~ 1)) %>%
  select(-c(NSP, MSTV, MLTV))

# Check the prevalence of abnormal CTGs
mean(CTG$ABNORMAL) # prevalence rate of 22.15%
```

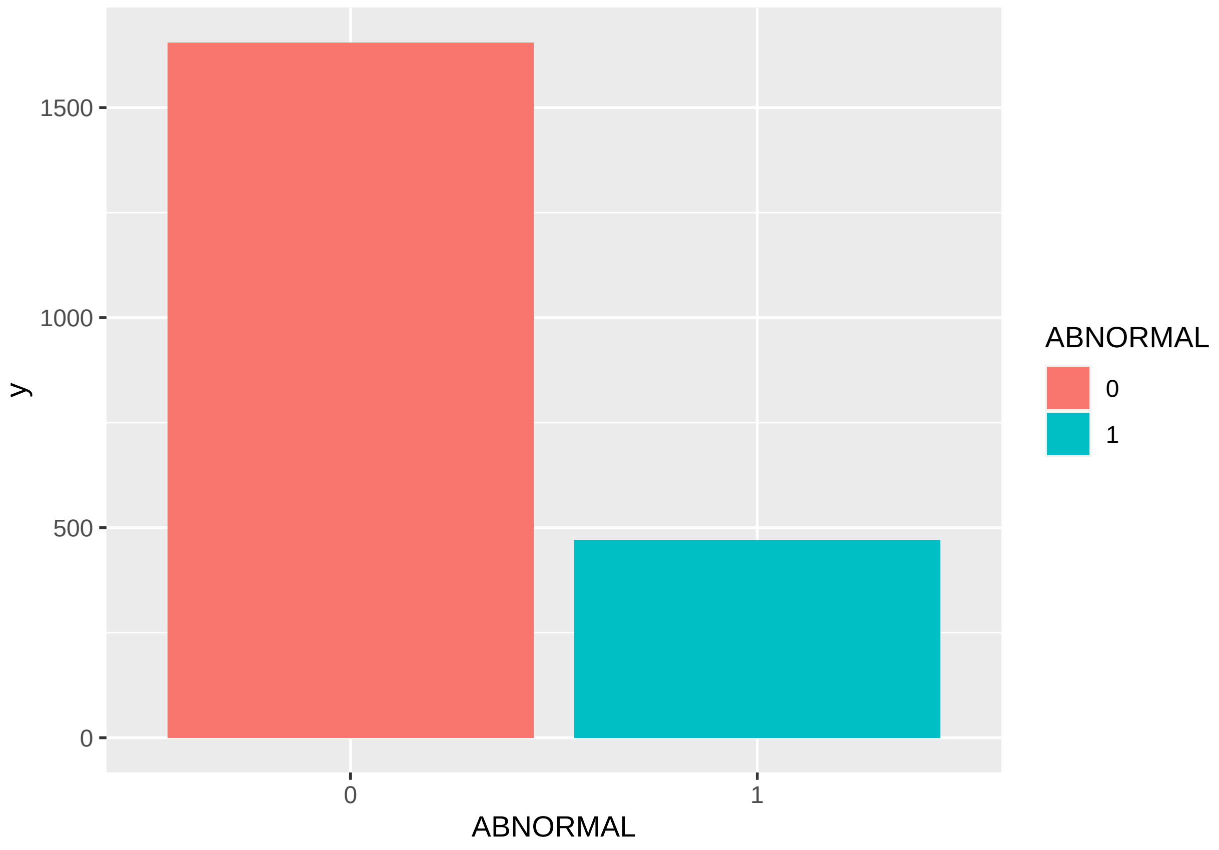
```
## [1] 0.2215428
```

```
table(CTG$ABNORMAL) # 471 events
```

```
##
```

```
##    0    1
## 1655  471
```

```r
# Convert outcome variable to a factor variable
CTG$ABNORMAL <- as.factor(CTG$ABNORMAL)

# Create a bar plot for the outcome
ggplot(data = CTG, aes(x=ABNORMAL, y=1, fill = ABNORMAL)) +
  geom_bar(stat = "identity")
```



Partition the data into development and validation subsets

A popular model validation method consists of forming a test set by setting aside instances in the data set (some may argue that this approach is inefficient compared with cross-validation or bootstrap resampling). This data set will only be used to assess the performance of the final model that was trained on the development set. In this example, we allocate about 75% of the data for model development.

```r
## 75% of the sample size
smp_size <- floor(0.75 * nrow(CTG))

## set the seed to make your partition reproducible
set.seed(123)
train_ind <- sample(seq_len(nrow(CTG)), size = smp_size,
                    replace = FALSE)

train <- CTG[train_ind, ]
```

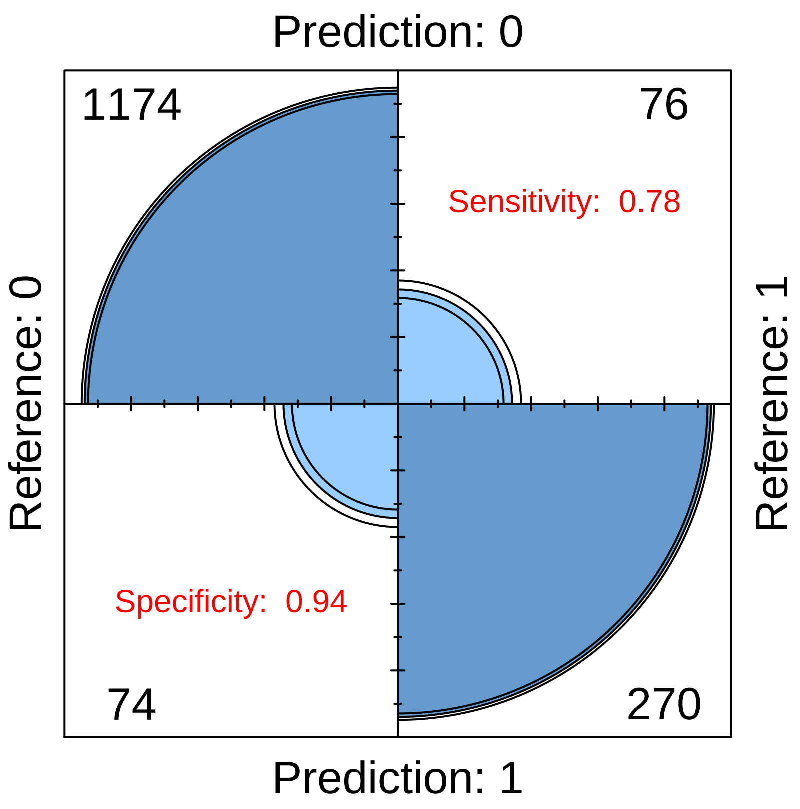
```r
test <- CTG[-train_ind, ]

# It will also be useful to separate the outcome and the predictors
x_train <- train %>% dplyr::select(-ABNORMAL)
y_train <- train %>% dplyr::select(ABNORMAL)

x_test <- test %>% dplyr::select(-ABNORMAL)
y_test <- test %>% dplyr::select(ABNORMAL)
```

Model development via logistic regression

As a means of comparison, we are first going to fit a logistic regression model on our training data set. The performance of the model will be assessed using three metrics:

- **Misclassification error**: Proportion of CTGs that were correctly classified
- **Sensitivity**: Proportion of the abnormal CTGs that were classified as abnormal by the model
- **Specificity**: Proportion of the normal CTGs that were classified as normal by the model

```r
# The syntax ~ followed by a dot means that we are modeling the
# outcome (ABNORMAL) as a function of the rest of the variables/columns
# in the data set.

logit.mod <- glm(ABNORMAL ~ ., data = train, family = "binomial")

# Performance in the training data
# Compute the predictions
predicted.train <- predict(logit.mod, newdata = train, type = "response")

# Create a confusion matrix using a cutoff probability of 0.5
conf.mat.logit.train <- confusionMatrix(as.factor(as.numeric(predicted.train >= 0.5)),
                                  train$ABNORMAL)

# Graphical parameters
par(mfrow=c(1,1), mar = c(5, 5, 1, 2),
    cex.axis=2, cex.lab=2, cex.main=1.2, cex.sub=1.5)

# Neat visualization for a confusion matrix
fourfoldplot(conf.mat.logit.train$table)
sensitivity <- conf.mat.logit.train$byClass[2]
specificity <- conf.mat.logit.train$byClass[1]
text(x = 0.5, y=0.6, paste("Sensitivity: ",round(sensitivity,2)), col = "red", cex = 1)
text(x = -0.5, y=-0.6, paste("Specificity: ",round(specificity,2)), col = "red", cex = 1)
```



Model development via Random Forest

This is the most interesting part of the tutorial, where we finally get to train a random forest model by leveraging the `randomForest` package.

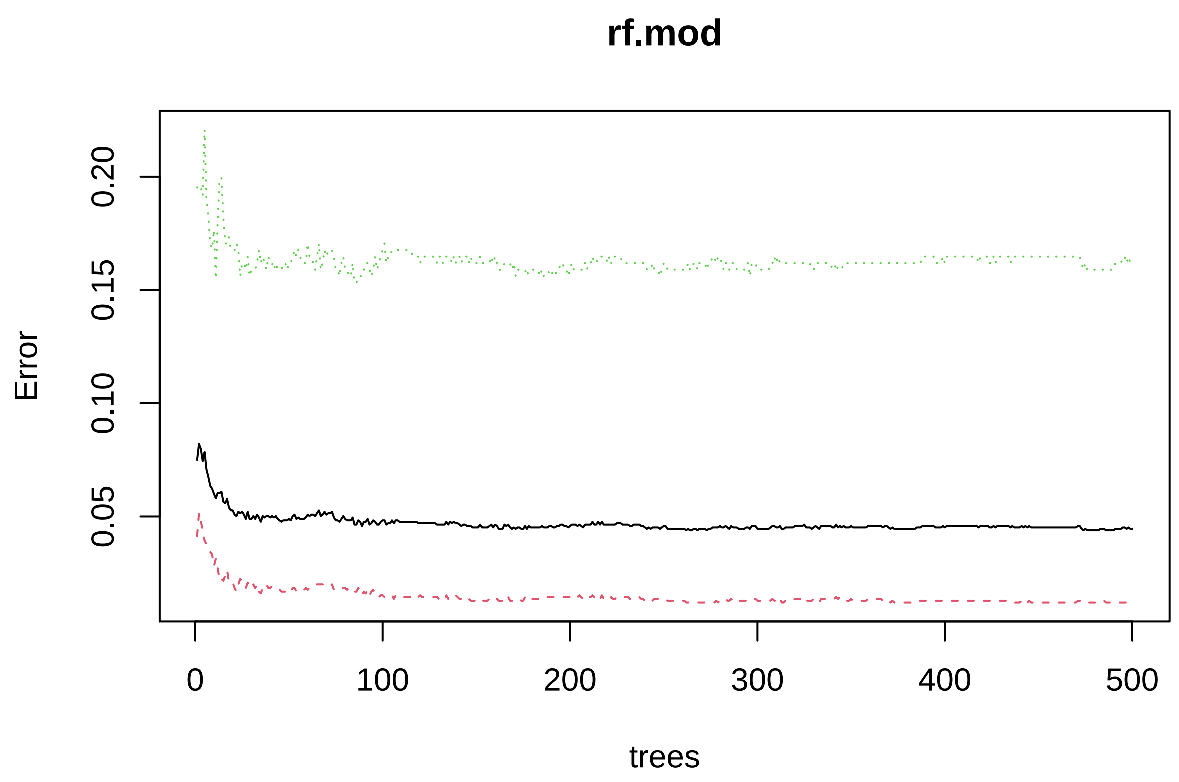
```
help("randomForest")
```

```
## starting httpd help server ... done
```

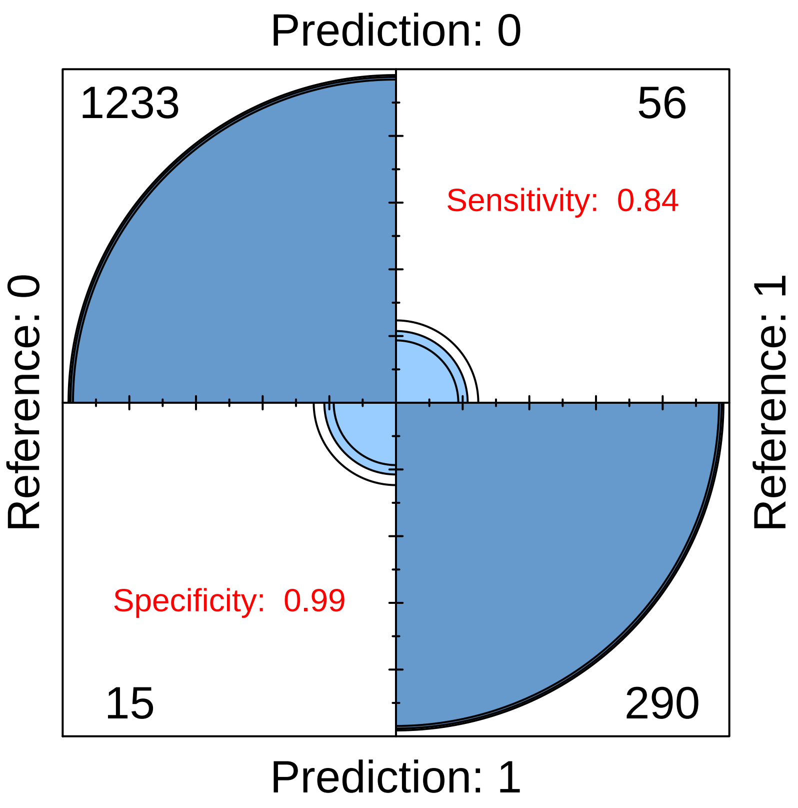
```
# Default values for the hyperparameters in the randomForest function:
# ntree = 500
# mtry = sqrt(p) for a categorical outcome
# nodesize = 1 for a categorical outcome
rf.mod <- randomForest(ABNORMAL ~ ., data = train)

# View the forest results.
print(rf.mod)
```

```
##
## Call:
##  randomForest(formula = ABNORMAL ~ ., data = train)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 4.45%
## Confusion matrix:
##      0   1 class.error
## 0 1233  15  0.01201923
## 1   56 290  0.16184971
```

```
plot(rf.mod)
```



```
# Computation of the confusion matrix
conf.mat.rf.train <- confusionMatrix(rf.mod$predicted, train$ABNORMAL)

# Graphical parameters
par(mfrow=c(1,1), mar = c(5, 5, 1, 2),
    cex.axis=2, cex.lab=2, cex.main=1.2, cex.sub=1.5)

fourfoldplot(conf.mat.rf.train$table)
sensitivity <- conf.mat.rf.train$byClass[2]
specificity <- conf.mat.rf.train$byClass[1]
text(x = 0.5, y=0.6, paste("Sensitivity: ",round(sensitivity,2)), col = "red", cex = 1)
text(x = -0.5, y=-0.6, paste("Specificity: ",round(specificity,2)), col = "red", cex = 1)
```



Hyperparameter tuning for `mtry` and `ntree`

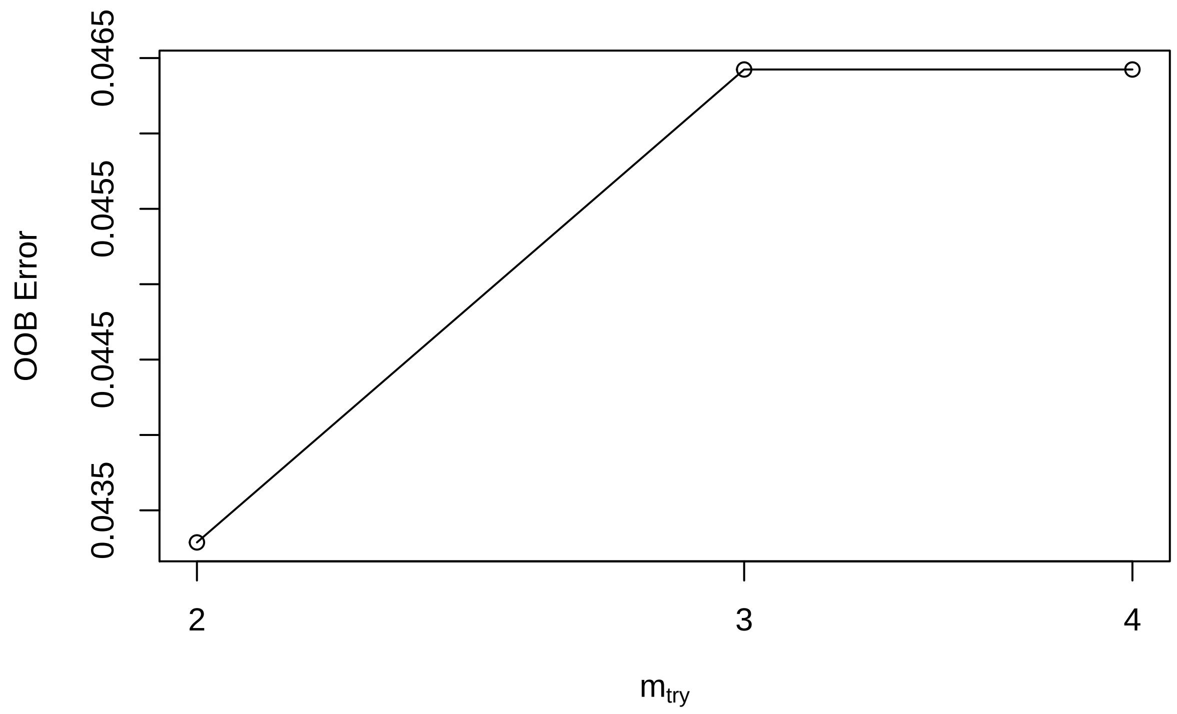
The second stage of model development is model refinement. Using an estimation of the generalization error (here, the out-of-bag error), we will perform a grid search to find values of `mtry` and `nodesize` that could yield a better predictive performance.

First method: Function `tuneRF` from the `randomForest` package

The function `tuneRF`() searches for value of `mtry` that minimizes the out-of-bag error. Aside from `ntree`, it is the hyperparameter that is most likely to affect final accuracy of the model.

```
set.seed(445)
# As opposed to randomForest() from the same package, tuneRF()
# does not take a formula object as an argument.
# It can only take a vector of the outcome 'y' and a matrix or
# data frame 'x' containing the predictors as arguments.

bestmtry <- tuneRF(x_train, y_train$ABNORMAL, stepFactor=1.5, improve=1e-5, ntree=500)
```

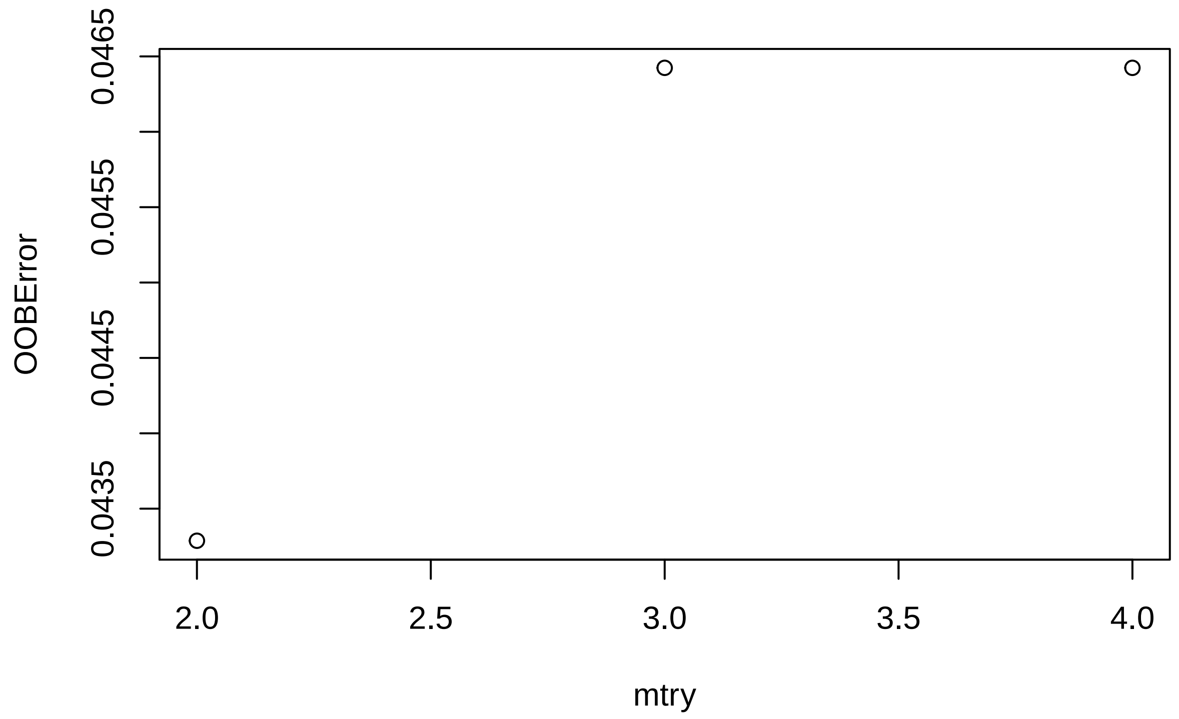
```
## mtry = 3  OOB error = 4.64%
## Searching left ...
## mtry = 2     OOB error = 4.33%
## 0.06756757 1e-05
## Searching right ...
## mtry = 4     OOB error = 4.64%
## -0.07246377 1e-05
```



```
print(bestmtry)
```

```
##       mtry   OOBError
## 2.OOB    2 0.04328733
## 3.OOB    3 0.04642409
## 4.OOB    4 0.04642409
```

```
plot(bestmtry, lty = 1)
```



The search stopped after three values, and we conclude that two features selected at random at each node split (`mtry`=2) is the optimal value.

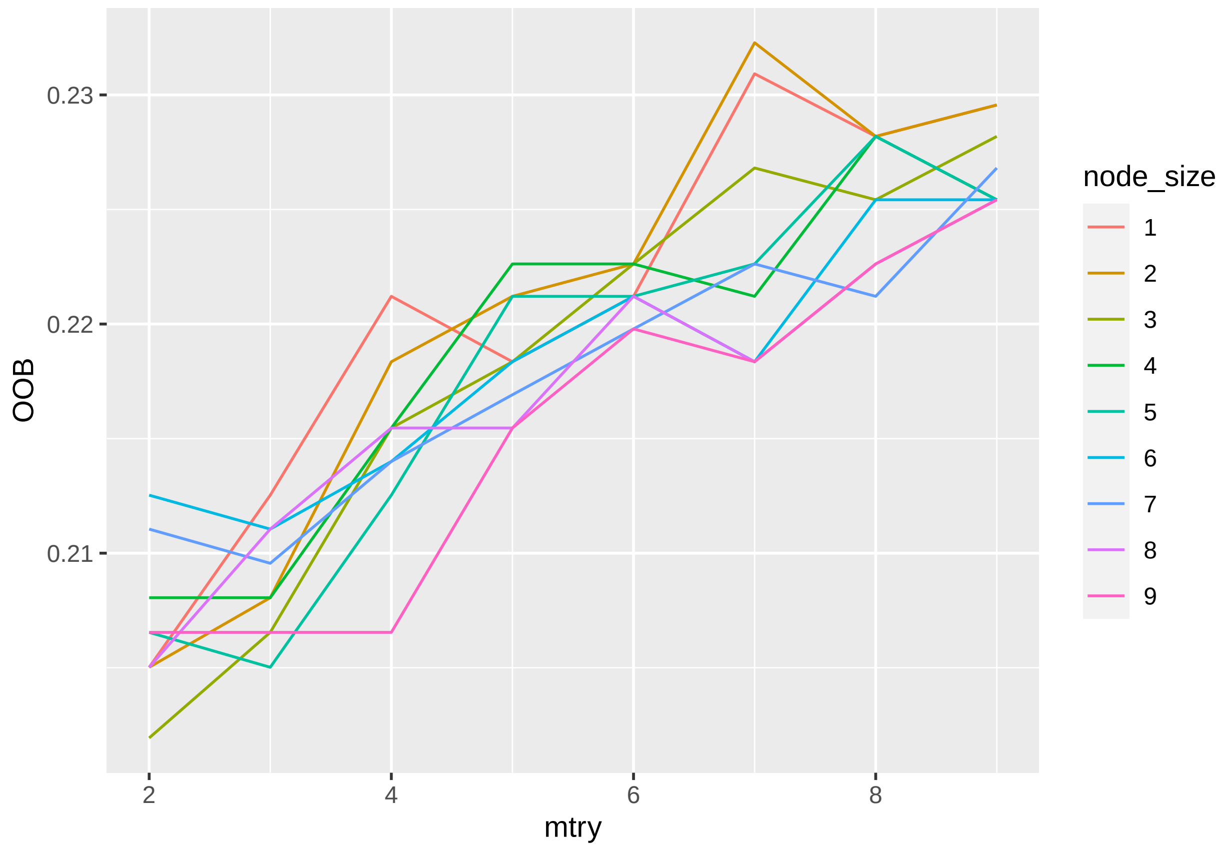
Second method: Manual search using the `ranger` package

If we wish to minimize the OOB error over more than one hyperparameter, we will need to program the search ourselves. In this chunk, we are optimizing the OOB error as a function of `mtry` and `nodesize`. For every combination of values, we fit a random forest model on the training data and record the value of the OOB error. For computational efficiency, we use the `ranger()` function in the `ranger` package, which is simply a different implementation of the random forest algorithm that runs 6 times as fast as `randomForest`.

```
hyper_grid <- expand.grid(
  mtry       = seq(2, 9, by = 1),
  node_size  = seq(1, 9, by = 1),
  OOB   = 0
)

# total number of combinations
nrow(hyper_grid)
```

```
## [1] 72
```

```
## [1] 96

for(i in 1:nrow(hyper_grid)) {

  # train model
  model <- ranger(
    formula         = ABNORMAL ~ .,
    data            = train,
    num.trees       = 500,
    mtry            = hyper_grid$mtry[i],
    min.node.size   = hyper_grid$node_size[i],
    seed            = 123
  )

  # add OOB error to grid
  hyper_grid$OOB[i] <- sqrt(model$prediction.error)
}

search <- hyper_grid %>%
  dplyr::arrange(OOB)

search$node_size <- as.factor(search$node_size)
ggplot(data = search, aes(x = mtry, y = OOB, color = node_size)) +
  geom_line()
```



We obtain that the optimal combination is (`mtry`=2, `nodesize`=3). Recall that `nodesize` controls the minimum size of terminal nodes.

Retrain the model with selected hyperparameters

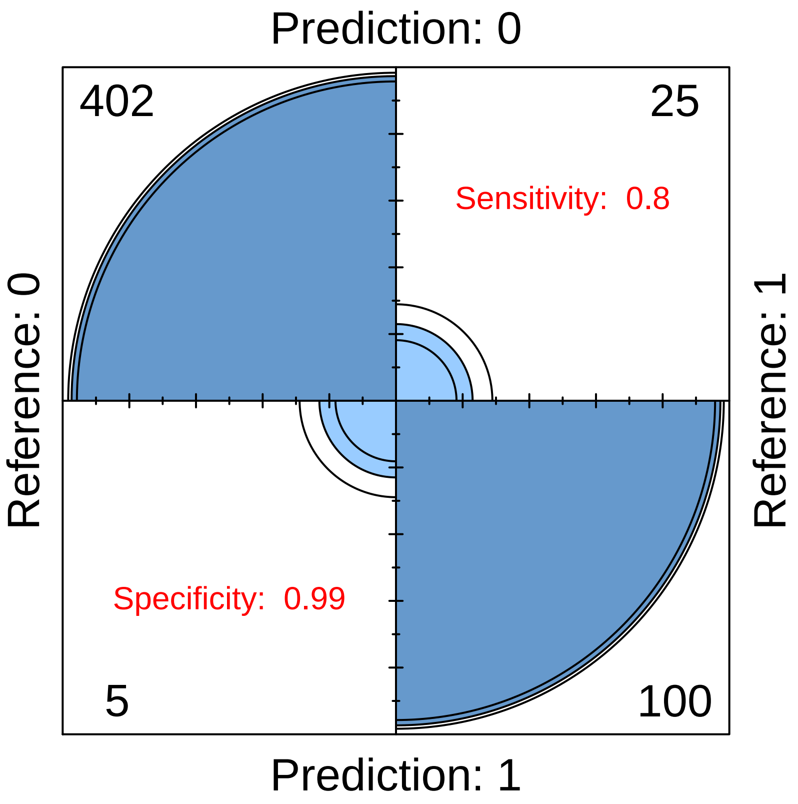
```
set.seed(123)
rf.mod2 <- randomForest(ABNORMAL ~ ., data = train,
                        mtry = 2, nodesize = 3, ntree = 500)

# View the forest results.
print(rf.mod2)
```

```
##
## Call:
##  randomForest(formula = ABNORMAL ~ ., data = train, mtry = 2,      nodesize = 3, ntree = 500)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##         OOB estimate of  error rate: 4.27%
## Confusion matrix:
##      0   1 class.error
## 0 1239   9 0.007211538
## 1   59 287 0.170520231
```

Output final predictive performance

Now that we have settled on a model, we can deploy the object `rf.mod2` to perform predictions on "external" data sets. The test set that we created at the start of the tutorial now comes in handy.

```
par(mfrow=c(1,1), mar = c(5, 5, 1, 2),
    cex.axis=2, cex.lab=2, cex.main=1.2, cex.sub=1.5)

# Compute predictions in the test set
test.pred <- predict(rf.mod2, x_test)

conf.mat.rf2.test <- confusionMatrix(test.pred, test$ABNORMAL)

fourfoldplot(conf.mat.rf2.test$table)
sensitivity <- conf.mat.rf2.test$byClass[2]
specificity <- conf.mat.rf2.test$byClass[1]
text(x = 0.5, y=0.6, paste("Sensitivity: ",round(sensitivity,2)), col = "red", cex = 1)
text(x = -0.5, y=-0.6, paste("Specificity: ",round(specificity,2)), col = "red", cex = 1)
```

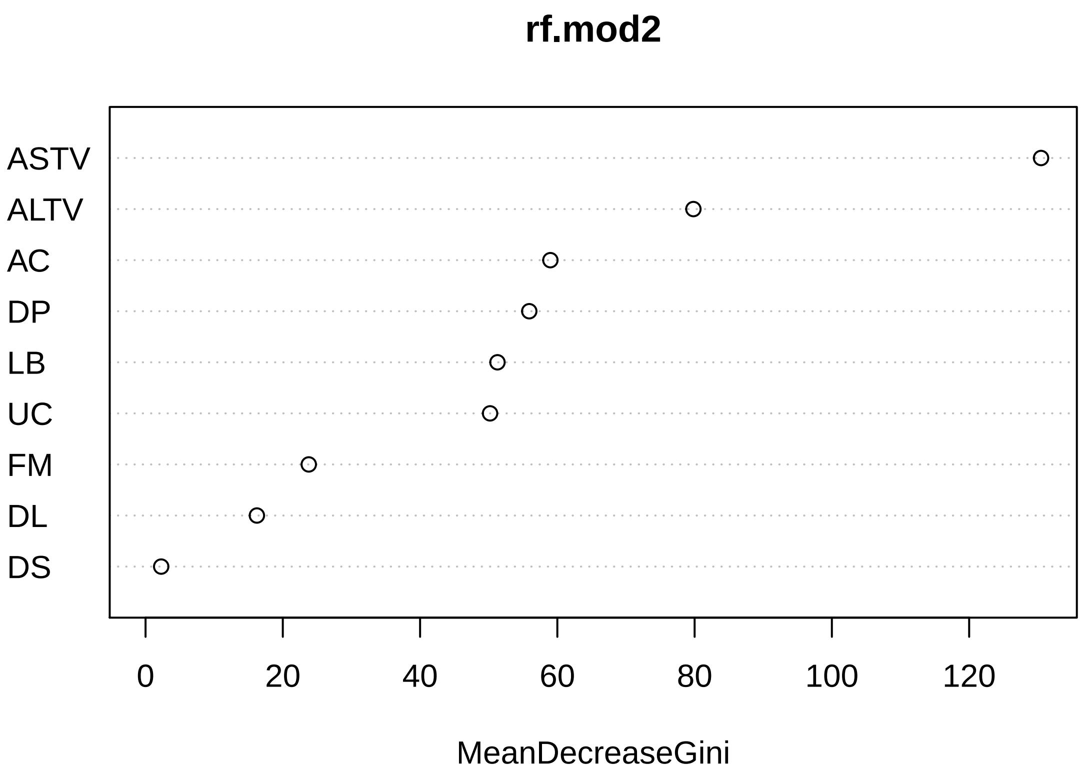


We obtain a specificity of 0.99 and a sensitivity of 0.8 on *unseen* data. Not bad!

Variance importance plot

With any `randomForest` object is associated a variable importance plot, which indicates the average improvement in purity (measured by the Gini index) brought by each predictor.

```
# Importance of each predictor.
varImpPlot(rf.mod2)
```

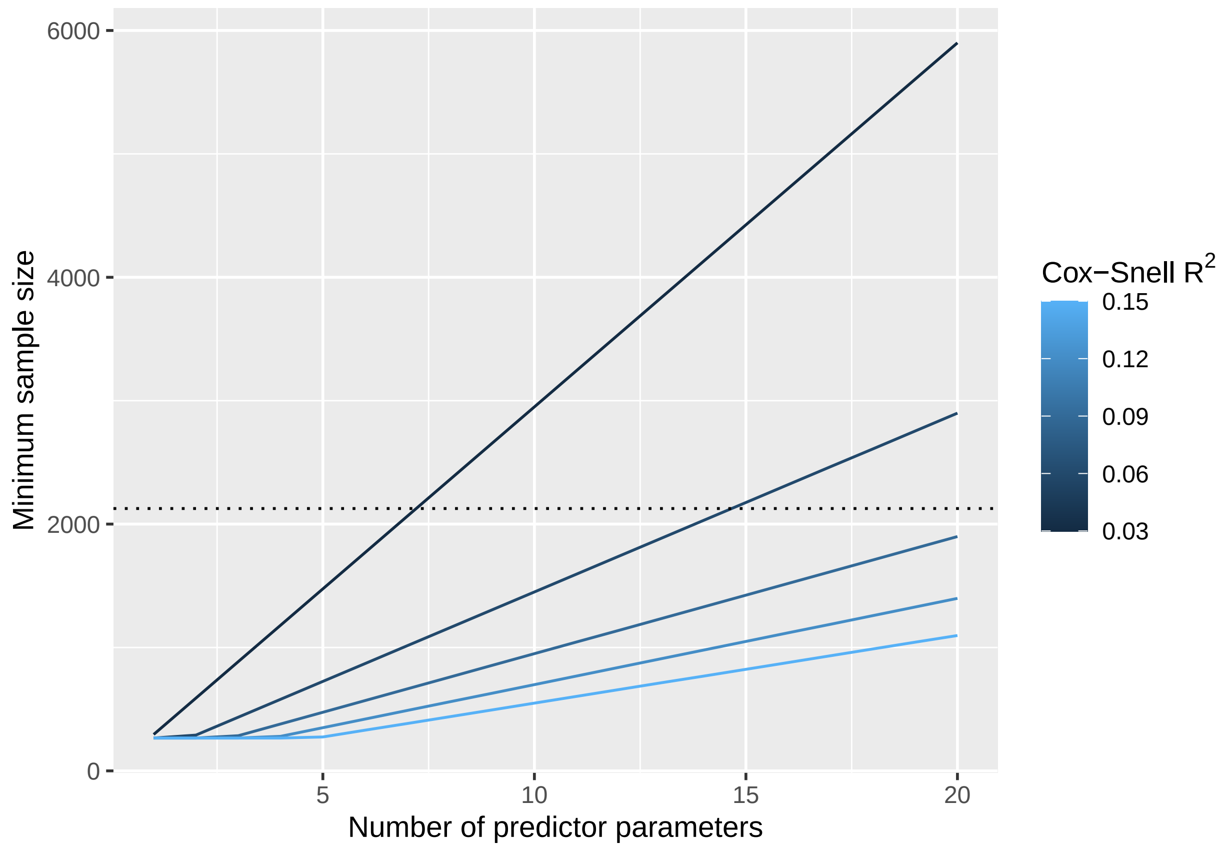


Best practices

Determine budget of predictor parameters for logistic regression using the package pmsampsize

I suggest determining how many predictors a logistic regression model can afford before thinking about using out-of-the-box prediction algorithms, which typically involve more predictor *parameters* (a lot of interactions in the case of random forest) for the same set of predictors. This approach gives an upper bound for the number of predictors that you can include in a prediction model. The **pmsampsize** package was written by Riley et al. (2020)

```
# Sample size calculation for prediction

rsq.vec <- seq(0.03, 0.15, by = 0.03)
pp.vec <- seq(1:20)


sample.size <- data.frame(expand.grid(rsq.vec, pp.vec))
colnames(sample.size) <- c("rsq", "pp")
sample.size$size <- NA

for (i in 1:nrow(sample.size)){
    temp <- pmsampsize(type = "b",
                                 # Prevalence in the sample
                                 prevalence = mean(CTG$ABNORMAL == 1),
                                 # Number of predictor parameters
                                 parameters = sample.size$pp[i],
                                 # Anticipated Cox-Snell R squared
                                 rsquared = sample.size$rsq[i])
    sample.size$size[i] <- temp$sample_size
}

ggplot(data = sample.size, aes(x=pp, y = size, color = rsq)) +
  geom_line(aes(group = rsq)) +
  geom_hline(yintercept = nrow(CTG), lty = "dotted") +
  labs(x = "Number of predictor parameters",
       y = "Minimum sample size",
       color = expression(paste("Cox-Snell ", R^2)))
```



Prediction using the rms package

Finally, if you choose to adopt a more conventional predictive modeling approach for a data analysis project, I suggest having a look at the **rms** package. It allows you to fit models for continuous, categorical or time-to-event outcomes and to validate the fitted models using bootstrap resampling through the simple call of `rms::validate`. Since it uses bootstrap resampling for model validation, you don't need to split the data set into training and validation sets. You will then have an idea of the apparent performance in your sample, whether there is evidence of over-fitting and optimism-corrected performance metrics.

```r
# Now using the whole CTG dataset

# lrm fits a logistic regression model for the outcome
# ABNORMAL as a function of all other predictors in the data set
logit.mod2 <- lrm(data = CTG, formula = ABNORMAL ~ .,
                  x= TRUE, y = TRUE)

set.seed(244)

# Internal validation using bootstrap resampling
validation <- rms::validate(logit.mod2, B = 1000)
```

```
## singular information matrix in lrm.fit (rank= 9 ).  Offending variable(s):
## DS
##
## Divergence or singularity in 1 samples
```

```r
validation
```

```
##           index.orig training    test optimism index.corrected   n
## Dxy           0.9156   0.9177  0.9136   0.0042          0.9115 999
## R2            0.7185   0.7228  0.7131   0.0098          0.7087 999
## Intercept     0.0000   0.0000 -0.0123   0.0123         -0.0123 999
## Slope         1.0000   1.0000  0.9652   0.0348          0.9652 999
## Emax          0.0000   0.0000  0.0099   0.0099          0.0099 999
## D             0.6325   0.6376  0.6259   0.0117          0.6208 999
## U            -0.0009  -0.0009  0.0003  -0.0012          0.0003 999
## Q             0.6335   0.6386  0.6256   0.0130          0.6205 999
## B             0.0661   0.0651  0.0668  -0.0017          0.0678 999
## g             4.6109   4.7157  4.5463   0.1694          4.4415 999
## gp            0.3163   0.3166  0.3153   0.0013          0.3150 999
```