Random forests

STAT30270 – Statistical Machine Learning

Contents

1 Random forests

Random forests are implemented in R in the package `randomForest` via the function with the same name. Take some time to read the documentation page `?randomForest`.

To showcase the method we consider the `Ionosphere` data available in the package `mlbench`. The data consist of radar signal measurements, classified into one of two classes: `bad` if the signal does not show any evidence of ionosphere structure, `good` if the signal has evidence of a structure. The class labels are in the variable `Class`. The data include a number of features, in particular 32 numerical features recording the electromagnetic signal.

We partition the data into training and test and train and evaluate a random forest classifier. The function `randomForest` allows to give in input also the test data through the arguments `xtest` and `ytest`, enabling training and testing in one call. Argument `keep.forest = TRUE` to retain the collection of trees.

```
library(randomForest)
library(caret)
data("Ionosphere", package = "mlbench")

ionosphere <- Ionosphere[,-c(1,2)]  # keep only numerical features + class

# split
set.seed(779900)
train <- createDataPartition(ionosphere$Class, p = 0.8, list = FALSE)
test <- setdiff(1:nrow(ionosphere), train)

# train
rf <- randomForest(Class ~ ., data = ionosphere, subset = train,
                   mtry = 15, ntree = 1000,
                   xtest = ionosphere[test,-33], ytest = ionosphere$Class[test],
                   keep.forest = TRUE)
rf

# the output on the test set is equivalent to:
yhat <- predict(rf, newdata = ionosphere[test,])
table(ionosphere$Class[test], yhat)
```

1.1 Task

- Take some time to explore the output and read the documentation of function `randomForest`. What is the OOB estimate of the generalized predictive performance?
- The `randomForest` object is a list with a lot of useful information. Explore the different slots, in particular `err.rate`, `oob.times`, and `votes`. What are these quantities?

2 Tuning random forests

We now implement a proper cross-validation procedure to tune a random forest classifier in application to these data. The set `train` set aside at earlier will be used to implement the cross-validation. The sample size available for the cross-validation is not too large, not too small, so we consider a 5-fold cross-validation, replicated 10 times. In this way, the size of the out-of-sample set is also in the same ballpark of the test data.

Upon inspection, the distribution of the classes in the data is mildly imbalanced (35% `bad` vs 65% `good`), so for an appropriate assessment we consider the metrics: accuracy, sensitivity, and specificity, and AU-ROC. The AU-ROC will be the metric used for model selection. Use of these metrics can be implemented by calling `twoClassSummary` in the `trainControl` function. See `?twoClassSummary` and `?trainControl`.

The main hyperparameter that can be tuned in `caret` is the number of variables considered at each split, `mtry`. However, we also wish to tune the size of the forest, `ntree`. Tuning of this hyperparameter is not automatically implemented in `caret`, so it is required to tune it manually. We do so with (simple) ad-hoc coding.

```
# enable parallel computing
library(doParallel)
cl <- makeCluster(detectCores() - 2)  # keep 2 cores free
registerDoParallel(cl)

# set training parameters
train_ctrl <- trainControl(method = "repeatedcv", number = 10, repeats = 5,
                           summaryFunction = twoClassSummary, classProbs = TRUE)

# set grids of hyperparameters
ntree_set <- c(100, 200, 500, 1000)
grid <- expand.grid( mtry = 2:(ncol(ionosphere)-1) )   # note: sqrt(32) = 5.67

# run tuning procedure
# NOTE: this will take time to run!
out <- vector("list", length(ntree_set))
for ( j in 1:length(ntree_set) ) {

  # remember that we need to make sure to train and
  # validate on the same splits!
  set.seed(3344)

  out[[j]] <- train(Class ~ .,
                    trControl = train_ctrl,
                    data = ionosphere,
                    method = "rf",
                    metric = "ROC",
                    tuneGrid = grid,
                    ntree = ntree_set[j])
}

stopCluster(cl)
```

Object `out` contains a list of objects `train`. In each slot, we can access the k-fold AU-ROC average, that we can then use to compare and select the best configuration (`mtry`, `ntree`). In each slot, these are in `"results"` –> `"ROC"`. We extract them and produce a plot of AU-ROC as a function of `mtry` and `ntree`.

```
# check for example
out[[3]]
out[[3]]$results

# extract selected `mtry`
lapply(out, "[[", "bestTune")

# extract and tidy up AU-ROC values
roc <- sapply(out, "[[", c("results", "ROC"))
colnames(roc) <- ntree_set
roc <- cbind(grid, roc)
head(roc)

# plot curves
colors <- c("purple2", "forestgreen", "darkorange3", "deepskyblue3")
matplot(roc$mtry, roc[,-1], type = "l", lwd = 2, lty = 1,
        col = colors)
grid()
legend("topright", legend = ntree_set, fill = colors, bty = "n")
```

Function `resamples` can be applied directly to the list `out` to extract summaries and comparing the performance over the `ntree` hyperparameters, with `mtry` fixed to the selected value by `caret`.

```
names(out) <- paste0("ntree_", ntree_set)
res <- resamples(out)
summary(res)
```

2.1 Task

- What is the best configuration (`mtry`, `ntree`)?
- What happens when `mtry` is equal to 32? What approach are we implementing? Why its performance is (generally) the worst?
- Use the selected model to predict the classification of the observations in the test data and assess its generalized predictive performance.
- Reproduce a similar example to compare and tune SVMs (with a kernel of your choice) and random forests on these data.