Exercise 9

Business Analytics and Data Science WS16/17

Introduction

This exercise is all about approaches to combine simple models into more complex ones, called ensemble models. We will experiment with the random forest, which aggregates a whole forest of randomized decision trees. Another homogeneous ensemble approach is gradient boosting, which trains simple base learners, e.g. very small trees, on the errors of the combination of the previous base learners to refine the model step-by-step.

Pseudo-code of random forest

The random forest is a tree-based classifier that performs competitively in many applications. As explained in the lecture, it is a combination of decision trees (**forest**) grown on random subsets of the observations and the feature space (**random**). Simply speaking, a large number of decison trees are (almost) fully grown but at each split only a (usually small) number of variables is randomly chosen as candidates for the split to make the trees more different to each other.
In this exercise, we will use package **mlr** to train and tune a random forest.

Building a random forest from scratch in R is more than we can do in one exercise. Nevertheless, let's look at the pseudo-code to remind you how the two approaches towards randomization work within the forest. Be careful to understand how each tree is build on a sample (with replacement) of the training observations and how each split is determined only over a sample of the available features.

```
Input: A training set X, features F, and number of trees in forest ntree.
1     For each tree in 1,..., mtry:
2       Take bootstrap sample from X
3       Build tree on bootstrap sample:
4         At each node:
5           Randomly choose a small subset of features
6           Determine best split on each of the sampled features
7           Split node on best split within the feature sample
8     Return the set of trees
```

Tuning a random forest with {mlr}

mlr is a package built to wrap an extensive list of existing model functions/packages into one coherent framework. Check out their homepage for an introduction and extensive tutorials: https://mlr-org.github.io/mlr-tutorial/release/html/index.html
When working with **mlr**, you go through the steps that we have discussed so far to set up the experimental framework and then start model training.

1. Load package **mlr** and create a **task**. Use function **makeClassifTask()** to specify the data and the target variable.
2. Define the prediction model, called *learner*. Use function **makeLearner()** to specify the model (the neural net from before), the prediction type (probability rather than class prediction) and any additional options for the underlying function **randomForest()**.
3. Set up the parameter tuning framework using function **makeResampleDesc()**. You could try (unrepeated) 3-fold cross-validation.
4. Specify the candidate parameter values for the neural network in an object **rf.parms** using function **makeParamSet()**. Use the examples in the R help as a guideline. We want to compare models with

different combinations of *mtry*, *sampsize* and *ntree*. If you have understood how the random forest works, these parameters should be intuitive enough for you to pick useful candidate values. Make sure to have at least several hundred trees. Make sure to choose a sensible range of *discrete* values and set a sensible *resolution* to numeric parameters using function **makeTuneControlGrid()**. How many models are you about to train? How much time to you expect this to take?

5. Start parameter tuning with function **tuneParams()** based on the area-under-curve. Report the optimal parameters for the neural net given the data.
6. Train the model at the best metaparameter values on the full training data using function **train()**. Be careful to avoid a conflict with the caret function of the same name. You can use mlr::train() or caret::train() to specify which package the function is from.
7. Predict along the lines of previous exercises (**yhat**[[**"rf"**]]) and evaluate the model performance on the test set using the AUC.

```
## Supervised task: tr
## Type: classif
## Target: BAD
## Observations: 920
## Features:
## numerics  factors  ordered
##       29        0        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Classes: 2
## good  bad
##  677  243
## Positive class: bad
```

```
## Learner classif.randomForest from package randomForest
## Type: classif
## Name: Random Forest; Short name: rf
## Class: classif.randomForest
## Properties: twoclass,multiclass,numerics,factors,ordered,prob,class.weights,oobpreds,featimp
## Predict-Type: prob
## Hyperparameters: replace=TRUE,importance=FALSE
```

```
## $mtry
## [1] 4
##
## $sampsize
## [1] 200
##
## $ntree
## [1] 1000
```

```
##    mtry sampsize ntree auc.test.mean iteration exec.time
## 1     2      300   200     0.6230571         1     1.056
## 2     4      300   200     0.6283433         2     0.531
## 3     6      300   200     0.6198527         3     0.750
## 4     2      200   200     0.6330073         4     0.339
## 5     4      200   200     0.6361253         5     0.514
## 6     6      200   200     0.6375824         6     0.554
## 7     2      300   600     0.6334758         7     1.516
## 8     4      300   600     0.6309336         8     2.006
## 9     6      300   600     0.6294411         9     1.729
## 10    2      200   600     0.6344156        10     0.966
```

```
## 11    4      200   600     0.6345584        11     1.422
## 12    6      200   600     0.6333226        12     1.232
## 13    2      300  1000     0.6358433        13     2.998
## 14    4      300  1000     0.6356299        14     2.903
## 15    6      300  1000     0.6270414        15     2.934
## 16    2      200  1000     0.6346946        16     2.011
## 17    4      200  1000     0.6418655        17     2.150
## 18    6      200  1000     0.6331497        18     1.810
```

```
##         2         4         6
## 0.6324156 0.6345760 0.6300650
```

```
## Learner classif.randomForest from package randomForest
## Type: classif
## Name: Random Forest; Short name: rf
## Class: classif.randomForest
## Properties: twoclass,multiclass,numerics,factors,ordered,prob,class.weights,oobpreds,featimp
## Predict-Type: prob
## Hyperparameters: replace=TRUE,importance=FALSE
```

```
## Learner classif.randomForest from package randomForest
## Type: classif
## Name: Random Forest; Short name: rf
## Class: classif.randomForest
## Properties: twoclass,multiclass,numerics,factors,ordered,prob,class.weights,oobpreds,featimp
## Predict-Type: prob
## Hyperparameters: replace=TRUE,importance=FALSE,mtry=4,sampsize=200,ntree=1000
```

```
## List of 7
##  $ predict.type: chr "prob"
##  $ data        :'data.frame':    305 obs. of  4 variables:
##   ..$ truth    : Factor w/ 2 levels "good","bad": 1 1 1 1 1 1 1 2 1 1 ...
##   ..$ prob.good: num [1:305] 0.703 0.744 0.69 0.857 0.803 0.783 0.864 0.665 0.574 0.762 ...
##   ..$ prob.bad : num [1:305] 0.297 0.256 0.31 0.143 0.197 0.217 0.136 0.335 0.426 0.238 ...
##   ..$ response : Factor w/ 2 levels "good","bad": 1 1 1 1 1 1 1 1 1 1 ...
##  $ threshold   : Named num [1:2] 0.5 0.5
##   ..- attr(*, "names")= chr [1:2] "good" "bad"
##  $ task.desc   :List of 11
##   ..$ id          : chr "tr"
##   ..$ type        : chr "classif"
##   ..$ target      : chr "BAD"
##   ..$ size        : int 920
##   ..$ n.feat      : Named int [1:3] 29 0 0
##   .. ..- attr(*, "names")= chr [1:3] "numerics" "factors" "ordered"
##   ..$ has.missings: logi FALSE
##   ..$ has.weights : logi FALSE
##   ..$ has.blocking: logi FALSE
##   ..$ class.levels: chr [1:2] "good" "bad"
##   ..$ positive    : chr "bad"
##   ..$ negative    : chr "good"
##   ..- attr(*, "class")= chr [1:3] "ClassifTaskDesc" "SupervisedTaskDesc" "TaskDesc"
##  $ time        : num 0.037
##  $ error       : chr NA
##  $ dump        : NULL
##  - attr(*, "class")= chr [1:2] "PredictionClassif" "Prediction"
```

(*Class only*) Parallel random forests

When talking about parallel computing, it is useful to think of the computer as a factory that produces our code output. Just like a factory, our system houses many processes at the same time. Usually when we run an R script, a single worker is working its way through the script and producing output at it goes along. And just like in a factory, several workers to work at separate parts of the project at the same time. Naturally, this requires some organization and the consolidation of the output of all workers overhead. In the case of paramter tuning, we face a large project that is "embarrassingly parallel", where the output of each worker is completely independent of the other workers' results. It is therefore especially convenient to split this work up between several workers and that's what we will do.

There are several levels here on which you could parallelize. The growing of each tree in a forest, the training of different forests/parameter setting, and the cross-validation are all embarrassingly parallel tasks. The most efficient approach depends on the size of each task and the available number of cores. A few big tasks might not make use of all available workers, while many small tasks introduce more overhead and often overuse memory.

```
## $simple
##    user  system elapsed
##  25.393   0.657  27.681
##
## $parallel_param
##    user  system elapsed
##   0.077   0.015  13.927
##
## $parallel_resample
##    user  system elapsed
##   0.334   0.055  16.327
```

Gradient booting

In gradient boosting, you add models in a forward-stagewise manner, i.e. one model at a time, based on the pseudo-residuals from the models that are summed up so far. This approach is implemented in R in the packages **gbm** (generalized boosted regression) and **xgboost** (eXtreme gradient boosting). We will see how to boost decision trees with **xgboost**.

Use the {mlr} framework to train a gradient boosting model using {xgboost}. Tune a gradient boosting model on the training data called **classif.xgboost** (or by using function **xgboost()**). Gradient Boosting has a several parameters that can be tuned. The parameters are: - Number of iterations: Governs the number of boosting steps, that is the number of trees that are added to the ensemble. Too many steps/trees might lead to overfitting, while too few trees might limit the expressive power of the model (this also depends on the complexity of an individual tree; see max. tree depth below) - Maximum depth of the boosted trees: Deeper trees fit more complex data, but could overfit. - Shrinkage/ Learning rate (eta): Impact of each tree on the final prediction. Think about it as the step-size when optimizing the loss function - Minimum loss reduction (gamma): Minimum loss reduction required to make a further partition on a leaf node of the tree. Controls the size of the tree to prevent overfitting - Subsample ratio of columns when constructing a tree (like the random subspace of variables in RF) - Min. sum of instance weight: Minimum number/weight of observations required to make a further partition on a leaf node of the tree

Select an optimal *learning rate* [0;1] (try between 0.01 and 0.2), *maximum tree depth* (try between 4 and 8), and *number of iterations* (try between 20 and 200), while keeping the other parameters at default. Same as for random forest, gradient boosting can be made stochastic by subsampling the training data at each iteration. Besides imposing a form of regularization, this speeds up training. You can try to further optimize the other parameters once the more important once are set or see how long it takes to optimize over a grid of all parameters.

```
## Learner classif.xgboost from package xgboost
## Type: classif
## Name: eXtreme Gradient Boosting; Short name: xgboost
## Class: classif.xgboost
## Properties: twoclass,multiclass,numerics,prob,weights,missings,featimp
## Predict-Type: prob
## Hyperparameters: nrounds=1,verbose=1
```

```
## $nrounds
## [1] 100
##
## $max_depth
## [1] 2
##
## $eta
## [1] 0.1
##
## $gamma
## [1] 0
##
## $colsample_bytree
## [1] 0.8
##
## $min_child_weight
## [1] 1
##
## $subsample
## [1] 0.8
```

```
## Learner classif.xgboost from package xgboost
## Type: classif
## Name: eXtreme Gradient Boosting; Short name: xgboost
## Class: classif.xgboost
## Properties: twoclass,multiclass,numerics,prob,weights,missings,featimp
## Predict-Type: prob
## Hyperparameters: nrounds=100,verbose=0,max_depth=2,eta=0.1,gamma=0,colsample_bytree=0.8,min_child_we
```

```
## $rf
##       auc
## 0.6297778
##
## $xgb
##     auc
## 0.65125
```