R Notebook 4

Prediction modelling: putting it all together

In this notebook, first we will only use higher order functions to generate a prediction modelling pipeline. This has the advantage that less code is required for fitting and testing models and preprocessing data. This prevents making errors in the low-level code. The downside is that technical details remains hidden, so it is important to first fully grasp the concepts underlying optimizing model performance. Another disadvantage is that a new domain specific 'language' has to be learnt.

Model tuning using caret

caret is a library that provides a uniform interface to most regression and classification models that are available in R. It uses a single function for preprocessing the data, tuning and training a model, `train`. The available methods are listed at http://topepo.github.io/caret/available-models.html. Although aimed at supervised learning, its pre-processing functions also contain unsupervised learning techniques, like PCA.

```
rm(list = ls(all = TRUE)) #clean up workspace
repos <- "http://cran-mirror.cs.uu.nl"
to.install <- setdiff(c("FSelector"),installed.packages())
if (length(to.install) != 0){
install.packages(to.install, dependencies = TRUE, repos = repos)}
# path <- "" #type the full path between the quotes and use
          #forward slashes ('/') to separate directories.
#install.packages('mlr', repos = repos, dependecies = T) #critical update (bugfix) for learning curves
# Example:
path <- "c:/users/Tolle002/Dropbox/UU-Data analysis/R-Notebooks"
```

First, we load the data and split it into training and test data.

```
library(dplyr)
load(file.path(path,"notebook 4/ads_prevENGs.RData"))
dat <- na.omit(dat)
dat$REC4 <- factor(dat$REC4)
## recode sparse categories (gives problems in cross validation)
dat$CRIMETYPE[dat$CRIMETYPE %in% c("sexual","property with violence")] <- "violence"
dat$CRIMETYPE <- droplevels(dat$CRIMETYPE) #drop unused factor levels
set.seed(13768) #for reproducability
dat <- dat[1:5000,] # memory conservation
randnum <- runif(nrow(dat)) #uniformly distributed random numbers in the range [0,1]
traindat <- dat[randnum<=0.6, ]
testdat<- dat[randnum>0.6, ]
rm(dat) #clean up for memory
```

In the following code, decision trees are tuned using 10-fold cross-validation. An automatic set of 3 tuning parameter configurations (i.e. `tuneLength`) is tried out.

```
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

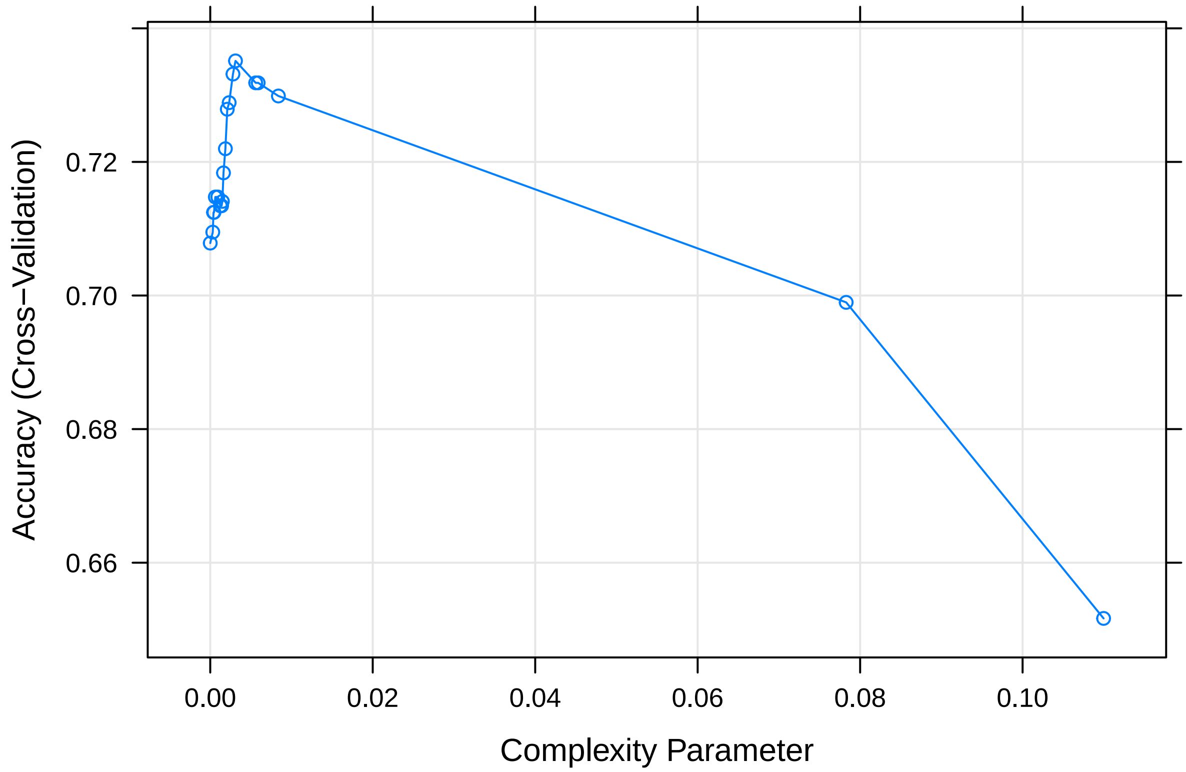
```r
set.seed(47728)
mod <- train(factor(REC4) ~ ., method = "rpart",
             data = traindat,
             # weights = rep(1,nrow(traindat)),
             preProcess = NULL,
             tuneLength = 20,
             trControl = trainControl(method = "cv", number = 10))
```

```
## Loading required package: rpart
```

```r
mod
```

```
## CART
##
## 3043 samples
##    6 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2738, 2739, 2739, 2738, 2738, 2739, ...
## Resampling results across tuning parameters:
##
##   cp            Accuracy   Kappa
##   0.0000000000  0.7078440  0.33684571
##   0.0003106555  0.7094877  0.33871198
##   0.0003994142  0.7124450  0.34446157
##   0.0004659832  0.7124450  0.34506887
##   0.0006213110  0.7147476  0.34771751
##   0.0009319664  0.7147498  0.34775957
##   0.0012426219  0.7134318  0.34283311
##   0.0013979497  0.7134318  0.34250982
##   0.0014911463  0.7140887  0.34404968
##   0.0016309413  0.7183574  0.34977596
##   0.0018639329  0.7219748  0.35627593
##   0.0020969245  0.7278872  0.36926949
##   0.0023299161  0.7288740  0.37001567
##   0.0027958993  0.7331503  0.38101257
##   0.0031065548  0.7351240  0.38577029
##   0.0055917987  0.7318389  0.38017215
##   0.0059024542  0.7318389  0.38126316
##   0.0083876980  0.7298673  0.38087252
##   0.0782851817  0.6989711  0.34212993
##   0.1099720410  0.6516566  0.09363411
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was cp = 0.003106555.
```

```r
plot(mod)
```

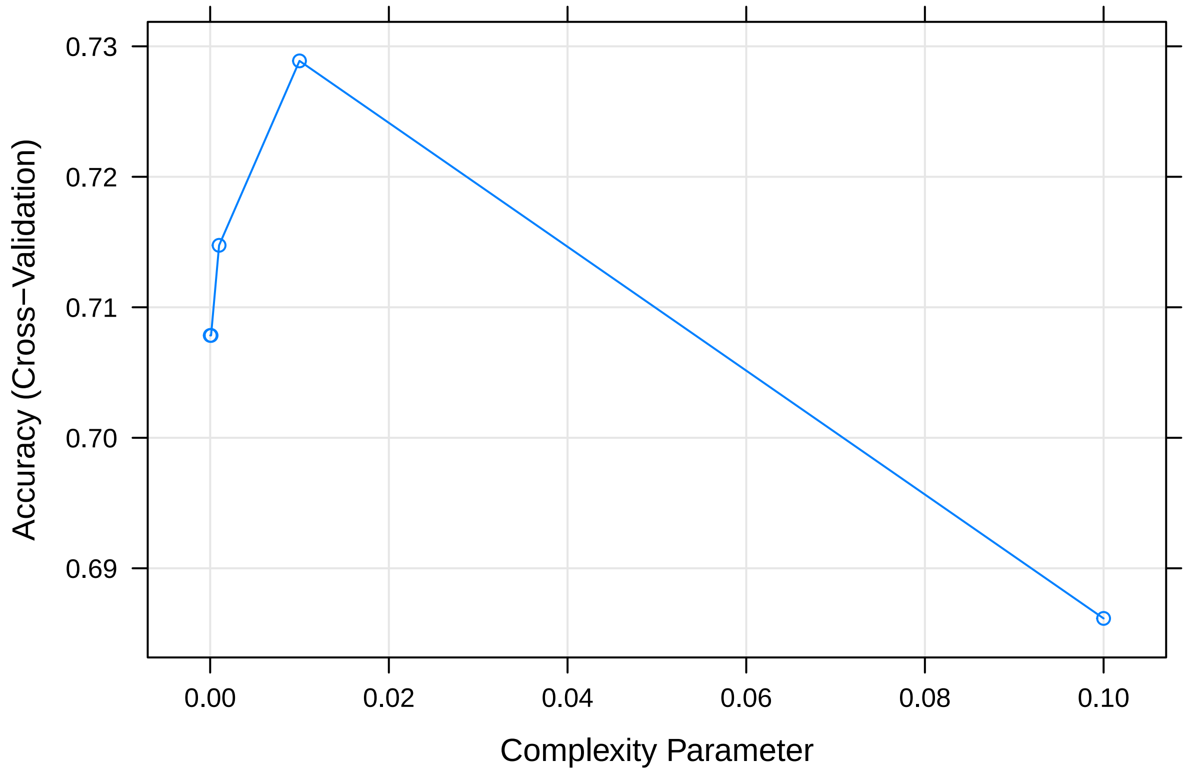


By default, the criterion that is optimised for classification models is accuracy. The output shows us that the value of the complexity at 0.0031 is optimal of the 20 values of `cp` tried.

By specifying `tuneLength`, we let the program itself decide which values for tuning parameters are applied. It is advisable to alseset these yourself. They may only be adequate for data of a typical size.

Setting tuning values yourself is done by supplying your own tuning parameter grid in the `tuneGrid` argument:

```
set.seed(47728) #same seed as above
rpartGrid <- data.frame(cp = c(0.00001, 0.0001, 0.001, 0.01, 0.1))
mod2 <- train(
  factor(REC4) ~ .,
  method = "rpart",
  data = traindat,
  preProcess = NULL,
  tuneGrid = rpartGrid,
  trControl = trainControl(method = "cv", number = 10)
)
plot(mod2)
```




```
mod2
```

```
## CART
##
## 3043 samples
##    6 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2738, 2739, 2739, 2738, 2738, 2739, ...
## Resampling results across tuning parameters:
##
##   cp     Accuracy   Kappa
##   1e-05  0.7078440  0.3368457
##   1e-04  0.7078440  0.3368457
##   1e-03  0.7147498  0.3477596
##   1e-02  0.7288827  0.3806005
##   1e-01  0.6861572  0.3565213
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was cp = 0.01.
```

In this case, the cross-validated accuracy is now much lower at our own tuning parameter `cp` at 0.01. However, the automatic tuning parameter values result in a more complex tree.

```r
mod$finalModel
```

```
## n= 3043
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##   1) root 3043 1073 0 (0.6473874 0.3526126)
##     2) PREVCASES< 1.5 1647  316 0 (0.8081360 0.1918640) *
##     3) PREVCASES>=1.5 1396  639 1 (0.4577364 0.5422636)
##       6) PREVCASES< 6.5 882  407 0 (0.5385488 0.4614512)
##        12) AGE>=31.5 434  133 0 (0.6935484 0.3064516) *
##        13) AGE< 31.5 448  174 1 (0.3883929 0.6116071)
##          26) PREVCASES< 3.5 271  125 1 (0.4612546 0.5387454)
##            52) AGE>=20.5 202   98 0 (0.5148515 0.4851485)
##             104) PREVCASES< 2.5 111   46 0 (0.5855856 0.4144144) *
##             105) PREVCASES>=2.5 91   39 1 (0.4285714 0.5714286) *
##            53) AGE< 20.5 69   21 1 (0.3043478 0.6956522) *
##          27) PREVCASES>=3.5 177   49 1 (0.2768362 0.7231638) *
##       7) PREVCASES>=6.5 514  164 1 (0.3190661 0.6809339)
##        14) AGE>=53.5 53   22 0 (0.5849057 0.4150943)
##          28) PREVCASES< 34 45   15 0 (0.6666667 0.3333333) *
##          29) PREVCASES>=34 8    1 1 (0.1250000 0.8750000) *
##        15) AGE< 53.5 461  133 1 (0.2885033 0.7114967) *
```

```r
mod2$finalModel
```

```
## n= 3043
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 3043 1073 0 (0.6473874 0.3526126)
##    2) PREVCASES< 1.5 1647  316 0 (0.8081360 0.1918640) *
##    3) PREVCASES>=1.5 1396  639 1 (0.4577364 0.5422636)
##      6) PREVCASES< 6.5 882  407 0 (0.5385488 0.4614512)
##       12) AGE>=31.5 434  133 0 (0.6935484 0.3064516) *
##       13) AGE< 31.5 448  174 1 (0.3883929 0.6116071) *
##      7) PREVCASES>=6.5 514  164 1 (0.3190661 0.6809339) *
```

Now, we obtain the generalization accuracy on the test data. The caret function `confusionMatrix` reports all statistics available for classification tasks.

```r
confmat <- confusionMatrix(predict(mod, newdata = testdat), testdat$REC4)
confmat
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1090  365
##          1  159  343
##
##                Accuracy : 0.7322
##                  95% CI : (0.712, 0.7518)
##     No Information Rate : 0.6382
```

```
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.3812
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8727
##             Specificity : 0.4845
##          Pos Pred Value : 0.7491
##          Neg Pred Value : 0.6833
##              Prevalence : 0.6382
##          Detection Rate : 0.5570
##    Detection Prevalence : 0.7435
##       Balanced Accuracy : 0.6786
##
##        'Positive' Class : 0
##
```

```
## predict will by default output predicted classes. For probabilities, type = "prob"
## is required.
```

This is a complete list of the performance measures used above

```r
## separate measures can be accessed by using $
cat("overall measures\n\n")
```

```
## overall measures
```

```r
confmat$overall #overall measures
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   7.322432e-01   3.811813e-01   7.120315e-01   7.517567e-01   6.382218e-01
## AccuracyPValue  McnemarPValue
##   5.194857e-19   3.383018e-19
```

```r
cat("\nclass-specific measures\n\n")
```

```
##
## class-specific measures
```

```r
confmat$byClass #class-specific measures
```

```
##          Sensitivity          Specificity       Pos Pred Value
##            0.8726982            0.4844633            0.7491409
##       Neg Pred Value            Precision               Recall
##            0.6832669            0.7491409            0.8726982
##                   F1           Prevalence       Detection Rate
##            0.8062130            0.6382218            0.5569750
## Detection Prevalence    Balanced Accuracy
##            0.7434849            0.6785807
```

now we find the associated ROC-curve

```r
library(pROC)
```

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

```
##
## Attaching package: 'pROC'
```

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

	Reference	
Predicted	Event	No Event
Event	A	B
No Event	C	D

The formulas used here are:

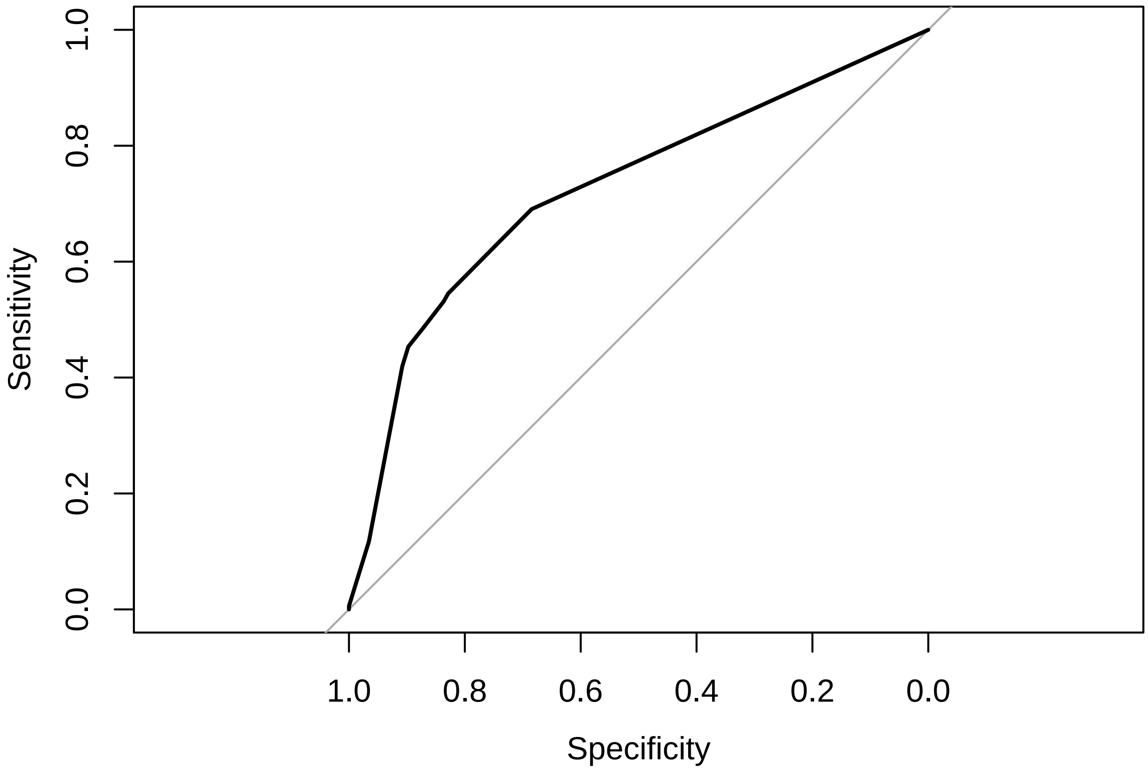
$$Sensitivity = \frac{A}{A+C}$$
$$Specificity = \frac{D}{B+D}$$
$$Prevalence = \frac{A+C}{A+B+C+D}$$
$$PPV = \frac{sensitivity \times prevalence}{((sensitivity \times prevalence) + ((1 - specificity) \times (1 - prevalence))}$$
$$NPV = \frac{specificity \times (1 - prevalence)}{((1 - sensitivity) \times prevalence) + ((specificity) \times (1 - prevalence))}$$
$$Detection\ Rate = \frac{A}{A+B+C+D}$$
$$Detection\ Prevalence = \frac{A+B}{A+B+C+D}$$
$$Balanced\ Accuracy = (sensitivity + specificity)/2$$

$$Precision = \frac{A}{A+B}$$
$$Recall = \frac{A}{A+C}$$
$$F1 = \frac{(1+\beta^2) \times precision \times recall}{(\beta^2 \times precision) + recall}$$

Figure 1: Performance measures for a cross classification table

```
##
##     cov, smooth, var
```

```r
r1 <- roc(testdat$REC4 ~ predict(mod, newdata = testdat, type = "prob")[,2])
plot(r1)
```




```
##
## Call:
## roc.formula(formula = testdat$REC4 ~ predict(mod, newdata = testdat,     type = "prob")[, 2])
##
## Data: predict(mod, newdata = testdat, type = "prob")[, 2] in 1249 controls (testdat$REC4 0) < 708 ca
## Area under the curve: 0.7241
```

A grid of all combinations of values for multiple tuning parameters can be generated using `expand.grid`. Here we expand the grid of the two tuning parameters of a radial basis kernel support vector machine, a nonlinear method for finding the optimal separating hyperplane between two classes (See e.g. James et al. 2013, chapter 9).

```r
svmGrid <- expand.grid(C = c(0.001, 0.01, 0.1, 1), sigma = c(0.001, 0.01, 0.1, 1))
```

And we tune the SVM model. As the SVM-model is extremely sensitive for class imbalance. As the cost parameter C holds for both classes evenly, the model will tend to classify the more prevalent class better. To counter this, we provide case weights so both classes are evenly weighted. Fitting an SVM is quite computer intensive. In order to function optimally, the data needs to be normalized. This can be accomplished by the `preProcess` command. Note that it is also available as a standalone command (see `?preProcess`; this actually only estimates the transformation itself, so it can be reused on other data)

```r
wts <- ifelse(traindat$REC4==1,
nrow(traindat)*0.5 / table(traindat$REC4)[2],
nrow(traindat)*0.5 / table(traindat$REC4)[1]
)
##questionr::wtd.table(traindat$REC4, weights = wts) #check equality of class frequencies

mod3 <- train(factor(REC4) ~ ., method = "svmRadial",
              data = traindat,
              weights = wts,
              preProcess = c("center","scale"),
              tuneGrid = svmGrid,
              trControl = trainControl(method = "cv", number = 5))
mod3
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 3043 samples
##    6 predictor
##    2 classes: '0', '1'
##
## Pre-processing: centered (15), scaled (15)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 2434, 2435, 2434, 2435, 2434
## Resampling results across tuning parameters:
##
##   C      sigma  Accuracy   Kappa
##   0.001  0.001  0.6473879  0.000000000
##   0.001  0.010  0.6473879  0.000000000
##   0.001  0.100  0.6473879  0.000000000
##   0.001  1.000  0.6473879  0.000000000
##   0.010  0.001  0.6473879  0.000000000
##   0.010  0.010  0.6473879  0.000000000
##   0.010  0.100  0.6473879  0.000000000
##   0.010  1.000  0.6473879  0.000000000
##   0.100  0.001  0.6480447  0.002404015
##   0.100  0.010  0.6615180  0.057839018
##   0.100  0.100  0.6726866  0.102921928
##   0.100  1.000  0.6588929  0.048024273
##   1.000  0.001  0.6628322  0.062948902
##   1.000  0.010  0.6927324  0.193136754
##   1.000  0.100  0.7091646  0.277338087
##   1.000  1.000  0.7114662  0.294551817
##
## Accuracy was used to select the optimal model using  the largest value.
## The final values used for the model were sigma = 1 and C = 1.
```

```r
confusionMatrix(predict(mod3, newdata = testdat), testdat$REC4)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1121  431
##          1  128  277
##
```

```
##                Accuracy : 0.7144
##                  95% CI : (0.6938, 0.7343)
##     No Information Rate : 0.6382
##     P-Value [Acc > NIR] : 5.82e-13
##
##                   Kappa : 0.3183
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8975
##             Specificity : 0.3912
##          Pos Pred Value : 0.7223
##          Neg Pred Value : 0.6840
##              Prevalence : 0.6382
##          Detection Rate : 0.5728
##    Detection Prevalence : 0.7931
##       Balanced Accuracy : 0.6444
##
##        'Positive' Class : 0
##
```

That is not impressive. Maybe random hyperparameter tuning will supply better generalization. This way, we may automatically try out values we would not think of ourselves.

```r
mod4 <- train(factor(REC4) ~ ., method = "svmRadial",
              data = traindat,
              weights = wts,
              preProcess = c("center","scale"),
              tuneLength =  4,
              trControl = trainControl(method = "cv", number = 5, search = "random"))
mod4
```

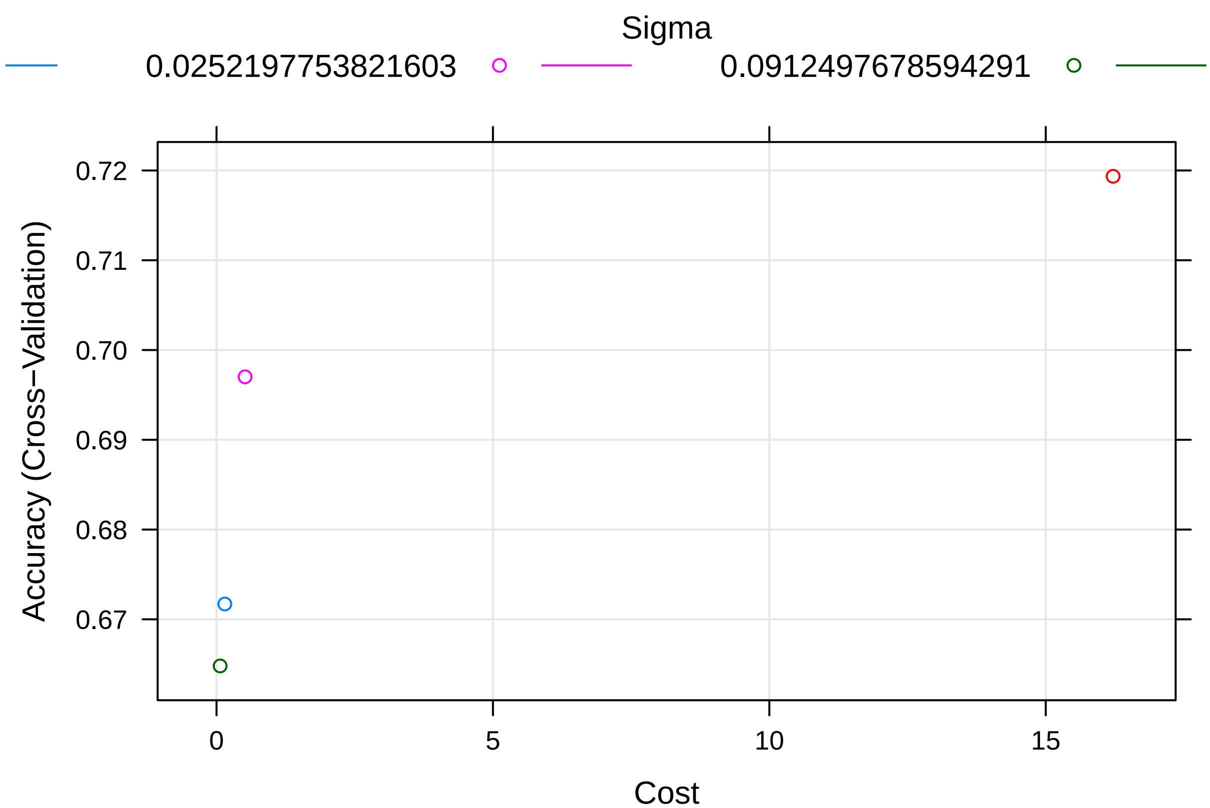
```
## Support Vector Machines with Radial Basis Function Kernel
##
## 3043 samples
##    6 predictor
##    2 classes: '0', '1'
##
## Pre-processing: centered (15), scaled (15)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 2435, 2434, 2434, 2434, 2435
## Resampling results across tuning parameters:
##
##   sigma       C            Accuracy   Kappa
##   0.01639634   0.14965874  0.6717046  0.09914277
##   0.02521978   0.51678367  0.6970135  0.21182234
##   0.09124977   0.06565961  0.6648043  0.07057808
##   0.17669462  16.21917681  0.7193539  0.33681274
##
## Accuracy was used to select the optimal model using  the largest value.
## The final values used for the model were sigma = 0.1766946 and C
##  = 16.21918.
```

```r
confusionMatrix(predict(mod4, newdata = testdat), testdat$REC4)
```

```
## Confusion Matrix and Statistics
##
```

```
##            Reference
## Prediction    0    1
##          0 1083  374
##          1  166  334
##
##                Accuracy : 0.7241
##                  95% CI : (0.7037, 0.7438)
##     No Information Rate : 0.6382
##     P-Value [Acc > NIR] : 4.44e-16
##
##                   Kappa : 0.3619
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8671
##             Specificity : 0.4718
##          Pos Pred Value : 0.7433
##          Neg Pred Value : 0.6680
##              Prevalence : 0.6382
##          Detection Rate : 0.5534
##    Detection Prevalence : 0.7445
##       Balanced Accuracy : 0.6694
##
##        'Positive' Class : 0
##
```

```
plot(mod4) #plot the random choices in the search space
```

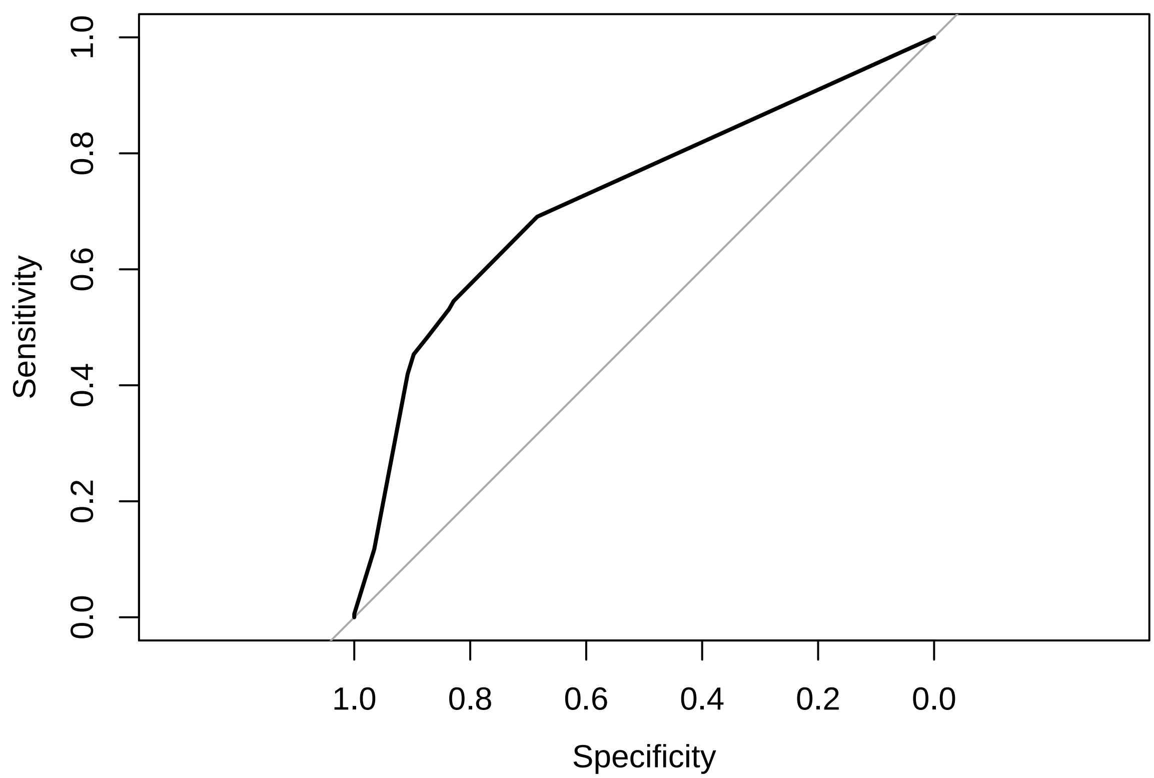


It seems that enlarging the Cost parameter `C` had the most effect.

```r
library(pROC)
confusionMatrix(predict(mod, newdata = testdat), testdat$REC4)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1090  365
##          1  159  343
##
##                Accuracy : 0.7322
##                  95% CI : (0.712, 0.7518)
##     No Information Rate : 0.6382
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.3812
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8727
##             Specificity : 0.4845
##          Pos Pred Value : 0.7491
##          Neg Pred Value : 0.6833
##              Prevalence : 0.6382
##          Detection Rate : 0.5570
##    Detection Prevalence : 0.7435
##       Balanced Accuracy : 0.6786
##
##        'Positive' Class : 0
##
```

```r
plot(roc(testdat$REC4 ~ predict(mod, newdata = testdat, type = "prob")[,2]))
```




```
##
## Call:
## roc.formula(formula = testdat$REC4 ~ predict(mod, newdata = testdat,     type = "prob")[, 2])
##
## Data: predict(mod, newdata = testdat, type = "prob")[, 2] in 1249 controls (testdat$REC4 0) < 708 ca
## Area under the curve: 0.7241
```

```r
lines
```

```
## function (x, ...)
## UseMethod("lines")
## <bytecode: 0x03eb63ac>
## <environment: namespace:graphics>
```

Another machine learning pipeline: MLR

Another more object oriented interface for classification and regression (*supervised learning*) is the 'mlr' package. It provides methods for every stage of the model building process and is completely object oriented and thus very extensible. It is also able to do cluster analysis (*unsupervised learning*). It is less limited than `caret` as it allows much more control on possible tuning parameters than caret. It is however somewhat more hard to use. *Note: loading library `mlr` breaks a loaded `caret` library*

A list of available models is obtained by `listLearners()`.

```r
library(mlr)
```

```
## Warning: package 'mlr' was built under R version 3.3.3
```

```
## Loading required package: ParamHelpers

##
## Attaching package: 'mlr'

## The following object is masked from 'package:caret':
##
##     train
```

```
source(file.path(path,"/notebook 4/Rlearner_classif_sknn.R")) # load a custom defined classifier
listLearners("classif", properties = ("prob"))  # here we request a list of available classification mo
```

```
## Warning in listLearners.character("classif", properties = ("prob")): The following learners could no
## classif.evtree,regr.evtree
## Check ?learners to see which packages you need or install mlr with all suggestions.

##                 class                                     name  short.name
## 1         classif.ada                             ada Boosting         ada
## 2 classif.bartMachine      Bayesian Additive Regression Trees bartmachine
## 3         classif.bdk              Bi-Directional Kohonen map         bdk
## 4    classif.binomial                      Binomial Regression    binomial
## 5  classif.blackboost Gradient Boosting With Regression Trees  blackboost
## 6    classif.boosting                          Adabag Boosting      adabag
##        package    type installed numerics factors ordered missings weights
## 1          ada classif      TRUE     TRUE    TRUE   FALSE    FALSE   FALSE
## 2  bartMachine classif      TRUE     TRUE    TRUE   FALSE     TRUE   FALSE
## 3      kohonen classif      TRUE     TRUE   FALSE   FALSE    FALSE   FALSE
## 4        stats classif      TRUE     TRUE    TRUE   FALSE    FALSE    TRUE
## 5 mboost,party classif      TRUE     TRUE    TRUE   FALSE     TRUE    TRUE
## 6 adabag,rpart classif      TRUE     TRUE    TRUE   FALSE     TRUE   FALSE
##   prob oneclass twoclass multiclass class.weights featimp oobpreds    se
## 1 TRUE    FALSE     TRUE      FALSE         FALSE   FALSE    FALSE FALSE
## 2 TRUE    FALSE     TRUE      FALSE         FALSE   FALSE    FALSE FALSE
## 3 TRUE    FALSE     TRUE       TRUE         FALSE   FALSE    FALSE FALSE
## 4 TRUE    FALSE     TRUE      FALSE         FALSE   FALSE    FALSE FALSE
## 5 TRUE    FALSE     TRUE      FALSE         FALSE   FALSE    FALSE FALSE
## 6 TRUE    FALSE     TRUE       TRUE         FALSE    TRUE    FALSE FALSE
##   lcens rcens icens
## 1 FALSE FALSE FALSE
## 2 FALSE FALSE FALSE
## 3 FALSE FALSE FALSE
## 4 FALSE FALSE FALSE
## 5 FALSE FALSE FALSE
## 6 FALSE FALSE FALSE
## ... (67 rows, 22 cols)
```

```
                                                # that are able to generate probability estimates
```

Everything in `mlr` is broken up into objects. As you can see above, each object has properties, that are also checked when running a program. The classification task is created by `makeClassifTask` that defines the data used and the outcome variable (target).

```
crime.task <- makeClassifTask(id = "crime", data = traindat, target = "REC4", positive = 1)
crime.task
```

```
## Supervised task: crime
## Type: classif
## Target: REC4
```

```
## Observations: 3043
## Features:
## numerics  factors  ordered
##        3        3        0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Classes: 2
##    0    1
## 1970 1073
## Positive class: 1
```

Data can be standardized using `normalizeFeatures`.

```
crime.task <- normalizeFeatures(crime.task, method = "standardize")
```

When predicting from a fitted model in a task, it will use the standardization information stored in the task.

Printing the object reveals different characteristics of the data.

Then we define some models we would like to fit (train). Note that default settings for the hyperparameters are in effect.

```
dtree <- makeLearner(cl = "classif.rpart", predict.type = "prob")
sknn <- makeLearner(cl = "classif.sknn", predict.type = "prob") #original k-nn
                  #does not support probabilities
logreg <- makeLearner(cl = "classif.logreg", predict.type = "prob")
print(dtree)
```

```
## Learner classif.rpart from package rpart
## Type: classif
## Name: Decision Tree; Short name: rpart
## Class: classif.rpart
## Properties: twoclass,multiclass,missings,numerics,factors,ordered,prob,weights,featimp
## Predict-Type: prob
## Hyperparameters: xval=0
```

```
print(sknn)
```

```
## Learner classif.sknn from package klaR
## Type: classif
## Name: K-nearest neigbors; Short name: sknn
## Class: classif.sknn
## Properties: twoclass,multiclass,numerics,factors,prob
## Predict-Type: prob
## Hyperparameters:
```

```
print(logreg)
```

```
## Learner classif.logreg from package stats
## Type: classif
## Name: Logistic Regression; Short name: logreg
## Class: classif.logreg
## Properties: twoclass,numerics,factors,prob,weights
## Predict-Type: prob
## Hyperparameters: model=FALSE
```

We can establish ourselves which *measure* to optimize. The complete list of performance criteria can be shown by the following command

```r
listMeasures("classif")
```

```
##  [1] "qsr"               "f1"                "tnr"
##  [4] "ppv"               "mcc"               "timetrain"
##  [7] "lsr"               "gpr"               "tpr"
## [10] "fn"                "fp"                "brier.scaled"
## [13] "fnr"               "kappa"             "wkappa"
## [16] "timeboth"          "fpr"               "multiclass.aunp"
## [19] "multiclass.aunu"   "bac"               "ssr"
## [22] "npv"               "brier"             "gmean"
## [25] "auc"               "ber"               "fdr"
## [28] "featperc"          "timepredict"       "multiclass.brier"
## [31] "acc"               "mmce"              "tn"
## [34] "tp"                "multiclass.au1p"   "multiclass.au1u"
## [37] "logloss"
```

Quite a comprehensive list of measures is available. It is also possible to create your own measures with `makeMeasure`, but we will not cover that here.

Separate models can be fitted (trained) by the function `train`

```r
mod.logreg <- train(logreg, crime.task)
mod.dtree <- train(dtree, crime.task)
mod.sknn <- train(sknn, crime.task)
```

We can estimate the generalization performance by the average out-of-sample metrics, in this case the AUC, accuracy and brier score. By using the `benchmark` function, we can compare several models (learners) using only a single command^(Note that you can also provide multiple tasks by making a list of tasks made of different data sets).

```r
rdesc <- makeResampleDesc(method = "CV", iters = 5) #create resampling strategy object. iters is the nu
bmr <- benchmark(list(dtree,logreg,sknn), task = crime.task, resamplings = rdesc, measures = list(auc,a
```

```
## Task: crime, Learner: classif.rpart

## [Resample] cross-validation iter 1:

## auc.test.mean=0.737,acc.test.mean=0.721,brier.test.mean=0.194

## [Resample] cross-validation iter 2:

## auc.test.mean=0.71,acc.test.mean=0.711,brier.test.mean=0.196

## [Resample] cross-validation iter 3:

## auc.test.mean=0.74,acc.test.mean=0.755,brier.test.mean=0.177

## [Resample] cross-validation iter 4:

## auc.test.mean=0.663,acc.test.mean=0.711,brier.test.mean=0.204

## [Resample] cross-validation iter 5:

## auc.test.mean=0.736,acc.test.mean=0.738,brier.test.mean=0.186

## [Resample] Aggr. Result: auc.test.mean=0.717,acc.test.mean=0.727,brier.test.mean=0.192

## Task: crime, Learner: classif.logreg

## [Resample] cross-validation iter 1:

## auc.test.mean=0.772,acc.test.mean=0.718,brier.test.mean=0.19

## [Resample] cross-validation iter 2:
```

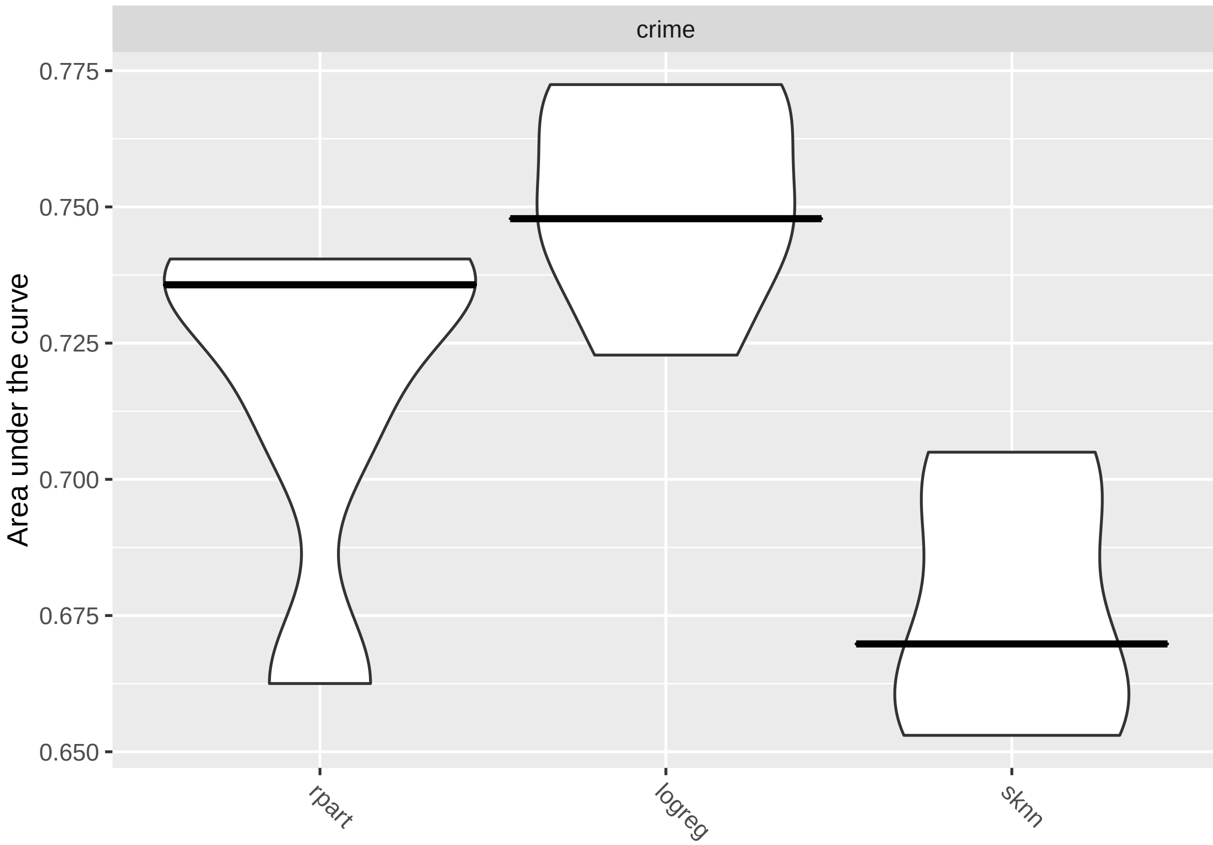
```
## auc.test.mean=0.744,acc.test.mean=0.706,brier.test.mean=0.185
## [Resample] cross-validation iter 3:
## auc.test.mean=0.748,acc.test.mean=0.734,brier.test.mean=0.183
## [Resample] cross-validation iter 4:
## auc.test.mean=0.723,acc.test.mean=0.696,brier.test.mean=0.198
## [Resample] cross-validation iter 5:
## auc.test.mean=0.77,acc.test.mean=0.73,brier.test.mean=0.185
## [Resample] Aggr. Result: auc.test.mean=0.751,acc.test.mean=0.717,brier.test.mean=0.188
## Task: crime, Learner: classif.sknn
## [Resample] cross-validation iter 1:
## auc.test.mean=0.697,acc.test.mean=0.704,brier.test.mean=0.23
## [Resample] cross-validation iter 2:
## auc.test.mean=0.67,acc.test.mean=0.674,brier.test.mean=0.237
## [Resample] cross-validation iter 3:
## auc.test.mean=0.657,acc.test.mean=0.675,brier.test.mean=0.243
## [Resample] cross-validation iter 4:
## auc.test.mean=0.653,acc.test.mean=0.681,brier.test.mean=0.244
## [Resample] cross-validation iter 5:
## auc.test.mean=0.705,acc.test.mean=0.702,brier.test.mean=0.219
## [Resample] Aggr. Result: auc.test.mean=0.676,acc.test.mean=0.687,brier.test.mean=0.235
```

```
bmr
```

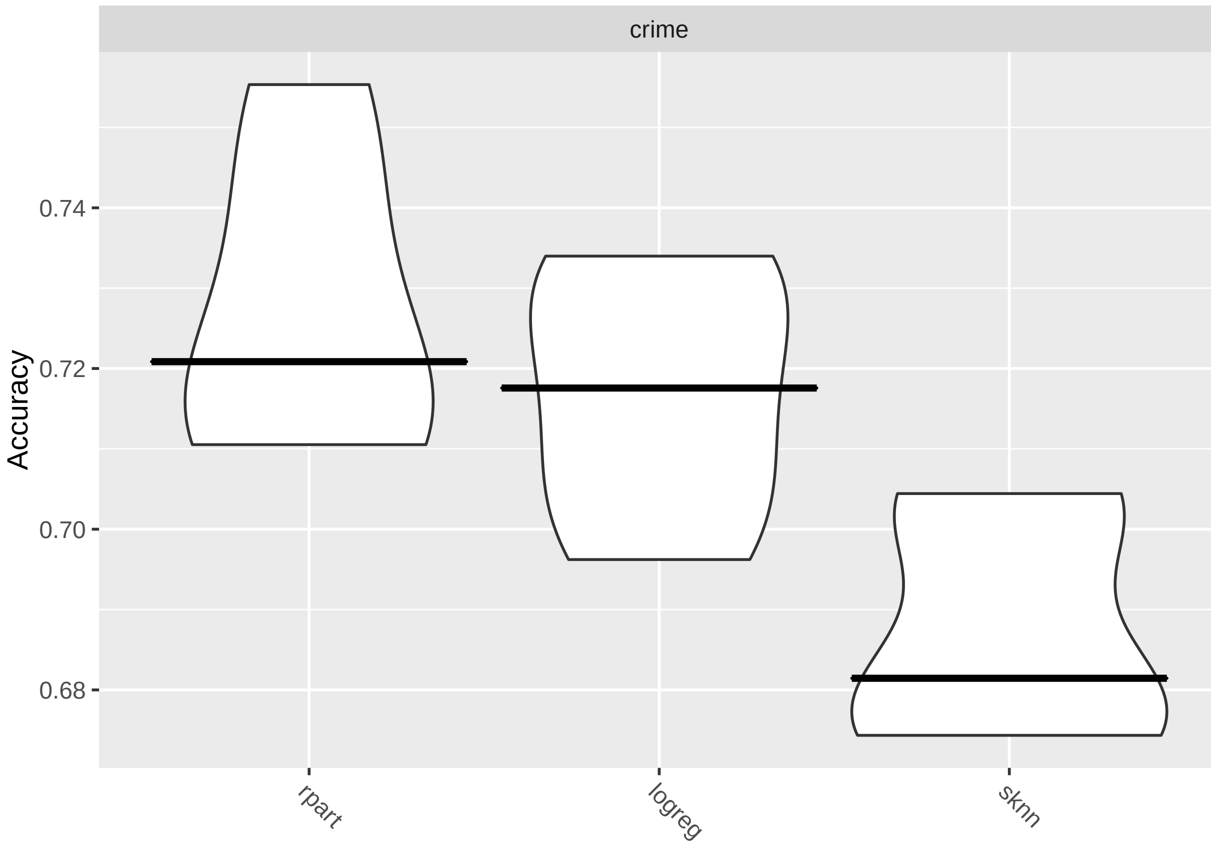
```
##   task.id      learner.id auc.test.mean acc.test.mean brier.test.mean
## 1   crime   classif.rpart     0.7171707     0.7272411       0.1915713
## 2   crime classif.logreg     0.7514901     0.7167277       0.1881437
## 3   crime    classif.sknn     0.6763212     0.6874800       0.2346327
```

and these come with a handy plot function, which also shows the standard error of the cross validation estimate.

```
plotBMRBoxplots(bmr, style = "violin") #default: first measure
```




```
plotBMRBoxplots(bmr, style = "violin", measure = acc)
```




```
plotBMRBoxplots(bmr, style = "violin", measure = brier)
```

Performance results can also be beautifully visualized in a web page

```
link  <-  plotViperCharts(bmr, chart = "lift", browse = TRUE)
```

Tuning

Like in caret, models can be 'tuned' over sets of hyperparameter values (i.e. hyperparameters are parameters that influence over/underfitting of the data, that can be set) The tunable parameters can be request by the `getParamSet` function

```
getParamSet(dtree)
```

```
##                       Type len  Def   Constr Req Tunable Trafo
## minsplit           integer   -   20 1 to Inf   -    TRUE     -
## minbucket          integer   -    - 1 to Inf   -    TRUE     -
## cp                 numeric   - 0.01   0 to 1   -    TRUE     -
## maxcompete         integer   -    4 0 to Inf   -    TRUE     -
## maxsurrogate       integer   -    5 0 to Inf   -    TRUE     -
## usesurrogate      discrete   -    2    0,1,2   -    TRUE     -
## surrogatestyle discrete   -    0      0,1   -    TRUE     -
## maxdepth           integer   -   30  1 to 30   -    TRUE     -
## xval               integer   -   10 0 to Inf   -   FALSE     -
## parms              untyped   -    -        -   -    TRUE     -
```

Default tuning parameter values are shown in column `Def`. The admissable range for the parameters is depicted in colum `Constr`.

We will try out different values of the complexity parameter and minbucket. The `makeDiscreteParam` function creates the grid over the (in this case two)objects supplied by `makeDiscreteParam`. The first controls the amount of splitting in a tree by setting the minimal improvement in the Gini impurity. The second controls the minimum number of observations that should be in a terminal node of the tree.

```
rdesc <- makeResampleDesc(method = "CV", iters = 5 ) #create resampling strategy object
ps <-  makeParamSet(  makeDiscreteParam("cp", values = c(0.0001,0.001,0.01,0.1)),
                      makeDiscreteParam("minbucket", values = c(5,10,15,20)))
ctrl <- makeTuneControlGrid() #needed for actually making a grid of all combinations out of the above.
res <- tuneParams("classif.rpart", task = crime.task, resampling = rdesc, par.set = ps,
  control = ctrl, measures = list(acc))
```

```
## [Tune] Started tuning learner classif.rpart for parameter set:

##                Type len Def               Constr Req Tunable Trafo
## cp        discrete   -   - 1e-04,0.001,0.01,0.1   -    TRUE     -
## minbucket discrete   -   -           5,10,15,20   -    TRUE     -

## With control class: TuneControlGrid

## Imputation value: -0

## [Tune-x] 1: cp=1e-04; minbucket=5

## [Tune-y] 1: acc.test.mean=0.697; time: 0.0 min

## [Tune-x] 2: cp=0.001; minbucket=5

## [Tune-y] 2: acc.test.mean=0.709; time: 0.0 min

## [Tune-x] 3: cp=0.01; minbucket=5

## [Tune-y] 3: acc.test.mean=0.734; time: 0.0 min

## [Tune-x] 4: cp=0.1; minbucket=5

## [Tune-y] 4: acc.test.mean=0.671; time: 0.0 min

## [Tune-x] 5: cp=1e-04; minbucket=10

## [Tune-y] 5: acc.test.mean=0.716; time: 0.0 min

## [Tune-x] 6: cp=0.001; minbucket=10

## [Tune-y] 6: acc.test.mean=0.718; time: 0.0 min

## [Tune-x] 7: cp=0.01; minbucket=10

## [Tune-y] 7: acc.test.mean=0.734; time: 0.0 min

## [Tune-x] 8: cp=0.1; minbucket=10

## [Tune-y] 8: acc.test.mean=0.671; time: 0.0 min

## [Tune-x] 9: cp=1e-04; minbucket=15

## [Tune-y] 9: acc.test.mean=0.72; time: 0.0 min

## [Tune-x] 10: cp=0.001; minbucket=15

## [Tune-y] 10: acc.test.mean=0.721; time: 0.0 min

## [Tune-x] 11: cp=0.01; minbucket=15

## [Tune-y] 11: acc.test.mean=0.734; time: 0.0 min

## [Tune-x] 12: cp=0.1; minbucket=15
```

```
## [Tune-y] 12: acc.test.mean=0.671; time: 0.0 min
## [Tune-x] 13: cp=1e-04; minbucket=20
## [Tune-y] 13: acc.test.mean=0.724; time: 0.0 min
## [Tune-x] 14: cp=0.001; minbucket=20
## [Tune-y] 14: acc.test.mean=0.725; time: 0.0 min
## [Tune-x] 15: cp=0.01; minbucket=20
## [Tune-y] 15: acc.test.mean=0.734; time: 0.0 min
## [Tune-x] 16: cp=0.1; minbucket=20
## [Tune-y] 16: acc.test.mean=0.671; time: 0.0 min
## [Tune] Result: cp=0.01; minbucket=5 : acc.test.mean=0.734
```

```
res_df <- as.data.frame(res$opt.path)
print(res_df) #optimal tuning result
```

```
##        cp minbucket acc.test.mean dob eol error.message exec.time
## 1  1e-04         5     0.6966787   1  NA          <NA>      0.11
## 2  0.001         5     0.7094936   2  NA          <NA>      0.14
## 3   0.01         5     0.7341419   3  NA          <NA>      0.10
## 4    0.1         5     0.6707372   4  NA          <NA>      0.09
## 5  1e-04        10     0.7160649   5  NA          <NA>      0.11
## 6  0.001        10     0.7180386   6  NA          <NA>      0.11
## 7   0.01        10     0.7341419   7  NA          <NA>      0.09
## 8    0.1        10     0.6707372   8  NA          <NA>      0.09
## 9  1e-04        15     0.7200112   9  NA          <NA>      0.11
## 10 0.001        15     0.7209992  10  NA          <NA>      0.11
## 11  0.01        15     0.7341419  11  NA          <NA>      0.11
## 12   0.1        15     0.6707372  12  NA          <NA>      0.10
## 13 1e-04        20     0.7236275  13  NA          <NA>      0.11
## 14 0.001        20     0.7252722  14  NA          <NA>      0.09
## 15  0.01        20     0.7341419  15  NA          <NA>      0.09
## 16   0.1        20     0.6707372  16  NA          <NA>      0.09
```

```
print(res$opt.path) #some technical information on the optimization
```

```
## Optimization path
##   Dimensions: x = 2/2, y = 1
##   Length: 16
##   Add x values transformed: FALSE
##   Error messages: TRUE. Errors: 0 / 16.
##   Exec times: TRUE. Range: 0.09 - 0.14. 0 NAs.
```

An improvement over the default value of `minbucket` was achieved.

Unlike `caret`, the final model is not stored in the object.

Therefore, we must now update the base learner with the new value for this hyperparameter and refit the model.

```
dtree <- setHyperPars(dtree, par.vals = res$x)
mod.dtree <- train(dtree, crime.task)
```

Of course, in all of the above we calculated generalized performance using data that was used to train the model. That is not representative of the actual generalization performance. For that, we must establish it on the actual test set.

```
pred.dtree <- predict(mod.dtree, newdata = testdat)
pred.logreg <- predict(mod.logreg, newdata = testdat)
pred.sknn <- predict(mod.sknn, newdata = testdat)

perf.dtree <- performance(pred.dtree, measures = list(auc, acc, brier))
perf.logreg <- performance(pred.logreg, measures = list(auc, acc, brier))
perf.sknn <- performance(pred.sknn, measures = list(auc, acc, brier))

rbind(perf.dtree[], perf.logreg[], perf.sknn[])
```

```
##             auc       acc     brier
## [1,] 0.6601824 0.6177823 0.2387327
## [2,] 0.7539975 0.6821666 0.3111404
## [3,] 0.5594854 0.6796117 0.3199341
```

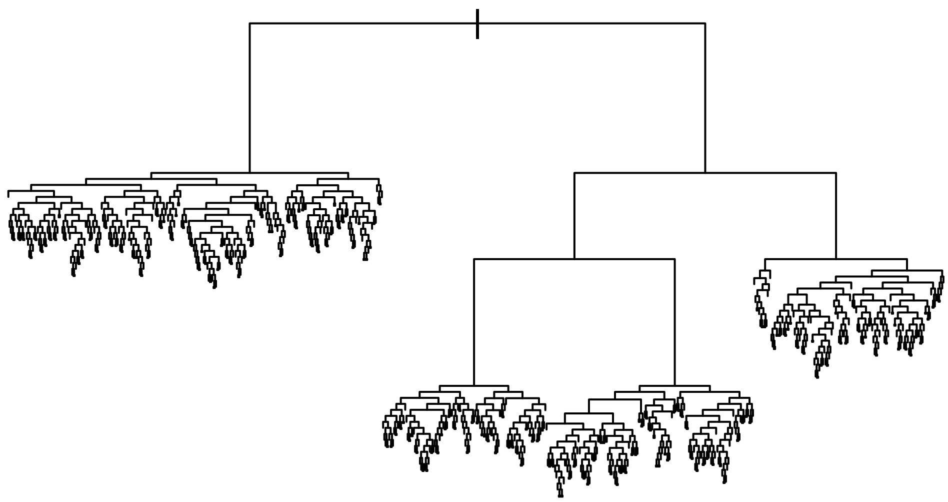
Actually, the tree outperforms logistic regression on accuracy and brier score, whereas logistic regression outperforms the tree on the AUC.

Learning curves

An important concept in het field of supervised learning is the learning curve. The learning curve shows the performance of a model while increasing the size of its training data. It can provide a quick assessment of whether the models are either low or high on bias and low or high on variance. When a model has high bias, it needs to be more complex. This can be attained by adding more variables or transformations of the original variables. Increasing the training data size further will not enhance model performance. When a model has high variance, it needs to reduce complexity or it needs more data to estimate its parameters more reliably. Complexity can be reduced either by regularization (L_1 or L_2), by performing variable selection, or by reducing the effective *df* of the parameters for a variable.

The learning curve provides strong hints on whether you need a more complex model or more training data when the insample and out-of-sample learning curves are plotted simultaneously.

```
library(mlr)
dtree <-  setHyperPars(dtree, par.vals = list(cp = 0.00001, minsplit = 2, minbucket = 1)) #make an over
mod.dtree <- train(dtree, crime.task)
plot(mod.dtree$learner.model)
```



```r
res <-  makeResampleDesc(method = "CV",  predict = 'both', stratify = TRUE)
set.seed(5173)
lcd <- generateLearningCurveData(learners = list(dtree, logreg),
                                 crime.task,resampling = res,
                                  measures = list(setAggregation(mmce,train.mean),
                                                  setAggregation(mmce, test.mean)))
```

```
## Task: crime, Learner: classif.rpart.1
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0256,mmce.test.mean=0.431
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.00733,mmce.test.mean=0.365
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.00733,mmce.test.mean=0.357
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0147,mmce.test.mean=0.395
## [Resample] cross-validation iter 5:
## mmce.train.mean=   0,mmce.test.mean=0.368
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.011,mmce.test.mean=0.357
```

```
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.00733,mmce.test.mean=0.372
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0147,mmce.test.mean=0.345
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.00366,mmce.test.mean=0.336
## [Resample] cross-validation iter 10:
## mmce.train.mean=   0,mmce.test.mean=0.325
## [Resample] Aggr. Result: mmce.train.mean=0.00916,mmce.test.mean=0.365
## Task: crime, Learner: classif.rpart.2
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0219,mmce.test.mean=0.332
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0128,mmce.test.mean=0.385
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0165,mmce.test.mean=0.325
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0238,mmce.test.mean=0.345
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0183,mmce.test.mean=0.339
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0274,mmce.test.mean=0.357
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0128,mmce.test.mean=0.349
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0219,mmce.test.mean=0.326
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0146,mmce.test.mean=0.355
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.0183,mmce.test.mean=0.374
## [Resample] Aggr. Result: mmce.train.mean=0.0188,mmce.test.mean=0.349
## Task: crime, Learner: classif.rpart.3
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0317,mmce.test.mean=0.382
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0219,mmce.test.mean=0.28
```

```
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0158,mmce.test.mean=0.344
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0231,mmce.test.mean=0.336
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0329,mmce.test.mean=0.342
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0207,mmce.test.mean=0.328
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0231,mmce.test.mean=0.342
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0183,mmce.test.mean=0.296
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0183,mmce.test.mean=0.336
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.0305,mmce.test.mean=0.37
## [Resample] Aggr. Result: mmce.train.mean=0.0236,mmce.test.mean=0.336
## Task: crime, Learner: classif.rpart.4
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0247,mmce.test.mean=0.365
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0274,mmce.test.mean=0.355
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0274,mmce.test.mean=0.311
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0265,mmce.test.mean=0.349
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0292,mmce.test.mean=0.395
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0228,mmce.test.mean=0.357
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0265,mmce.test.mean=0.365
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0374,mmce.test.mean=0.345
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0311,mmce.test.mean=0.336
```

```
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.0301,mmce.test.mean=0.393
## [Resample] Aggr. Result: mmce.train.mean=0.0283,mmce.test.mean=0.357
## Task: crime, Learner: classif.rpart.5
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0336,mmce.test.mean=0.345
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0329,mmce.test.mean=0.382
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0299,mmce.test.mean=0.354
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0299,mmce.test.mean=0.355
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0248,mmce.test.mean=0.336
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0343,mmce.test.mean=0.364
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0321,mmce.test.mean=0.382
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.027,mmce.test.mean=0.319
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0263,mmce.test.mean=0.316
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.0329,mmce.test.mean=0.357
## [Resample] Aggr. Result: mmce.train.mean=0.0304,mmce.test.mean=0.351
## Task: crime, Learner: classif.rpart.6
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0304,mmce.test.mean=0.372
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0329,mmce.test.mean=0.378
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0378,mmce.test.mean=0.344
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0347,mmce.test.mean=0.365
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0341,mmce.test.mean=0.352
```

```
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0335,mmce.test.mean=0.367
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0316,mmce.test.mean=0.339
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0335,mmce.test.mean=0.303
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0329,mmce.test.mean=0.352
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.0359,mmce.test.mean=0.318
## [Resample] Aggr. Result: mmce.train.mean=0.0337,mmce.test.mean=0.349
## Task: crime, Learner: classif.rpart.7
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0428,mmce.test.mean=0.319
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.035,mmce.test.mean=0.372
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0355,mmce.test.mean=0.318
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0376,mmce.test.mean=0.345
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.035,mmce.test.mean=0.309
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0303,mmce.test.mean=0.397
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.036,mmce.test.mean=0.405
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0391,mmce.test.mean=0.352
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0329,mmce.test.mean=0.332
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.0407,mmce.test.mean=0.361
## [Resample] Aggr. Result: mmce.train.mean=0.0365,mmce.test.mean=0.351
## Task: crime, Learner: classif.rpart.8
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0393,mmce.test.mean=0.342
```

```
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0406,mmce.test.mean=0.372
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0384,mmce.test.mean=0.315
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0388,mmce.test.mean=0.349
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0365,mmce.test.mean=0.362
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.042,mmce.test.mean=0.344
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0393,mmce.test.mean=0.359
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0393,mmce.test.mean=0.349
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0365,mmce.test.mean=0.332
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.0406,mmce.test.mean=0.328
## [Resample] Aggr. Result: mmce.train.mean=0.0391,mmce.test.mean=0.345
## Task: crime, Learner: classif.rpart.9
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0438,mmce.test.mean=0.326
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0402,mmce.test.mean=0.345
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0381,mmce.test.mean=0.318
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0422,mmce.test.mean=0.336
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0422,mmce.test.mean=0.322
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0369,mmce.test.mean=0.393
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0406,mmce.test.mean=0.352
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0402,mmce.test.mean=0.355
```

```
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.0406,mmce.test.mean=0.362
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.039,mmce.test.mean=0.351
## [Resample] Aggr. Result: mmce.train.mean=0.0404,mmce.test.mean=0.346
## Task: crime, Learner: classif.rpart.10
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.0456,mmce.test.mean=0.345
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.0424,mmce.test.mean=0.368
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.0398,mmce.test.mean=0.361
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.0438,mmce.test.mean=0.332
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.0431,mmce.test.mean=0.336
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.0409,mmce.test.mean=0.384
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.0416,mmce.test.mean=0.319
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.0427,mmce.test.mean=0.28
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.042,mmce.test.mean=0.339
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.042,mmce.test.mean=0.348
## [Resample] Aggr. Result: mmce.train.mean=0.0424,mmce.test.mean=0.341
## Task: crime, Learner: classif.logreg.1
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.271,mmce.test.mean=0.296
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.227,mmce.test.mean=0.306
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.286,mmce.test.mean=0.292
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.26,mmce.test.mean=0.276
```

```
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.267,mmce.test.mean=0.326
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.253,mmce.test.mean=0.305
## [Resample] cross-validation iter 7:
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
## mmce.train.mean=0.209,mmce.test.mean=0.286
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.289,mmce.test.mean=0.23
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.242,mmce.test.mean=0.25
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.205,mmce.test.mean=0.272
## [Resample] Aggr. Result: mmce.train.mean=0.251,mmce.test.mean=0.284
## Task: crime, Learner: classif.logreg.2
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.289,mmce.test.mean=0.289
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.247,mmce.test.mean=0.293
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.271,mmce.test.mean=0.305
## [Resample] cross-validation iter 4:
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
## mmce.train.mean=0.25,mmce.test.mean=0.257
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.305,mmce.test.mean=0.332
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.25,mmce.test.mean=0.279
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.263,mmce.test.mean=0.276
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.278,mmce.test.mean=0.23
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.291,mmce.test.mean=0.28
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.293,mmce.test.mean=0.318
```

```
## [Resample] Aggr. Result: mmce.train.mean=0.274,mmce.test.mean=0.286
## Task: crime, Learner: classif.logreg.3
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.262,mmce.test.mean=0.266
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.284,mmce.test.mean=0.293
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.259,mmce.test.mean=0.311
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.251,mmce.test.mean=0.26
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.255,mmce.test.mean=0.342
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.266,mmce.test.mean=0.295
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.266,mmce.test.mean=0.276
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.275,mmce.test.mean=0.227
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.263,mmce.test.mean=0.26
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.274,mmce.test.mean=0.275
## [Resample] Aggr. Result: mmce.train.mean=0.265,mmce.test.mean=0.281
## Task: crime, Learner: classif.logreg.4
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.298,mmce.test.mean=0.273
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.263,mmce.test.mean=0.293
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.273,mmce.test.mean=0.279
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.274,mmce.test.mean=0.263
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.267,mmce.test.mean=0.316
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.267,mmce.test.mean=0.315
```

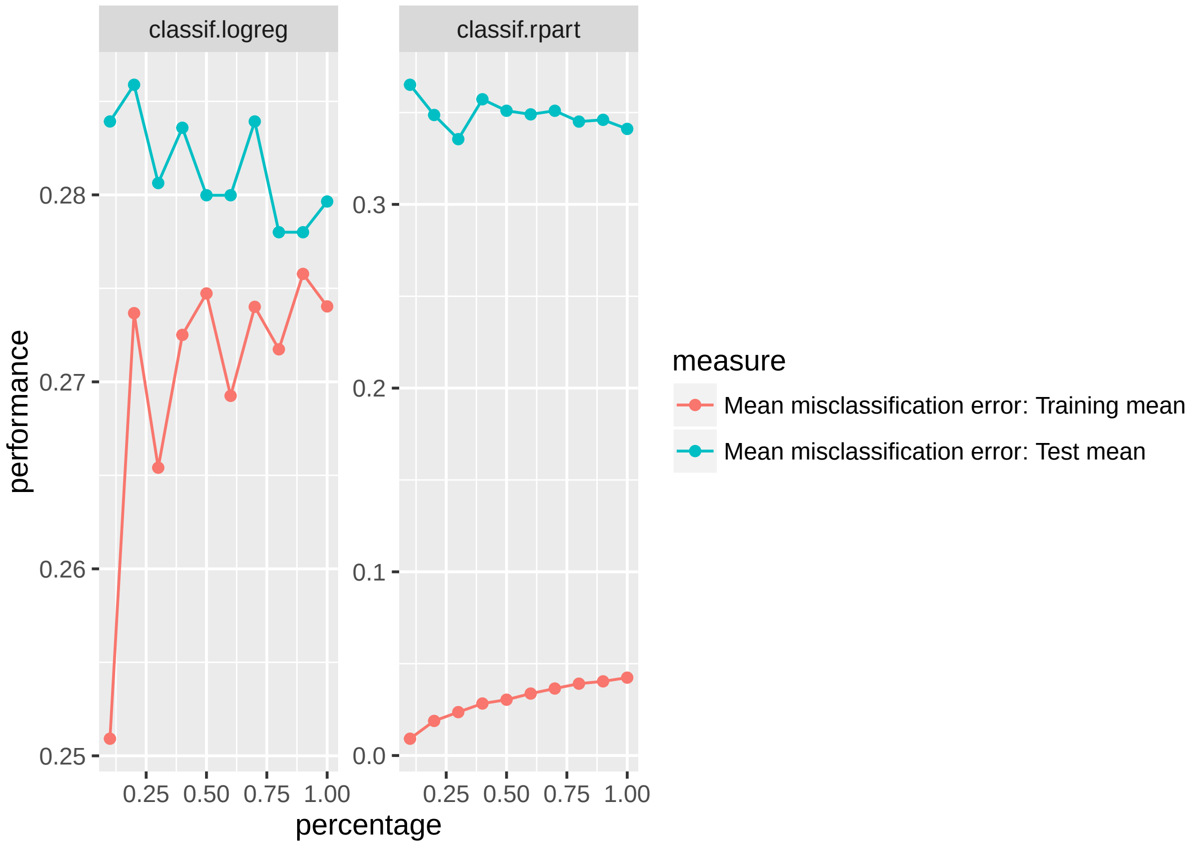
```
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.268,mmce.test.mean=0.25
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.281,mmce.test.mean=0.25
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.274,mmce.test.mean=0.299
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.261,mmce.test.mean=0.298
## [Resample] Aggr. Result: mmce.train.mean=0.273,mmce.test.mean=0.284
## Task: crime, Learner: classif.logreg.5
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.278,mmce.test.mean=0.253
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.274,mmce.test.mean=0.276
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.275,mmce.test.mean=0.289
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.277,mmce.test.mean=0.266
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.269,mmce.test.mean=0.326
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.264,mmce.test.mean=0.298
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.283,mmce.test.mean=0.303
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.28,mmce.test.mean=0.23
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.271,mmce.test.mean=0.28
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.278,mmce.test.mean=0.279
## [Resample] Aggr. Result: mmce.train.mean=0.275,mmce.test.mean=0.28
## Task: crime, Learner: classif.logreg.6
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.274,mmce.test.mean=0.263
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.254,mmce.test.mean=0.289
```

```
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.281,mmce.test.mean=0.266
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.261,mmce.test.mean=0.25
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.27,mmce.test.mean=0.319
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.253,mmce.test.mean=0.308
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.274,mmce.test.mean=0.276
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.283,mmce.test.mean=0.24
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.274,mmce.test.mean=0.296
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.268,mmce.test.mean=0.292
## [Resample] Aggr. Result: mmce.train.mean=0.269,mmce.test.mean=0.28
## Task: crime, Learner: classif.logreg.7
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.271,mmce.test.mean=0.273
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.26,mmce.test.mean=0.299
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.268,mmce.test.mean=0.275
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.286,mmce.test.mean=0.253
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.268,mmce.test.mean=0.322
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.27,mmce.test.mean=0.295
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.281,mmce.test.mean=0.276
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.292,mmce.test.mean=0.237
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.277,mmce.test.mean=0.309
```

```
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.266,mmce.test.mean=0.298
## [Resample] Aggr. Result: mmce.train.mean=0.274,mmce.test.mean=0.284
## Task: crime, Learner: classif.logreg.8
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.271,mmce.test.mean=0.266
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.265,mmce.test.mean=0.286
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.275,mmce.test.mean=0.289
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.266,mmce.test.mean=0.27
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.265,mmce.test.mean=0.319
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.264,mmce.test.mean=0.305
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.281,mmce.test.mean=0.26
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.283,mmce.test.mean=0.224
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.281,mmce.test.mean=0.28
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.267,mmce.test.mean=0.282
## [Resample] Aggr. Result: mmce.train.mean=0.272,mmce.test.mean=0.278
## Task: crime, Learner: classif.logreg.9
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.278,mmce.test.mean=0.276
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.27,mmce.test.mean=0.286
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.274,mmce.test.mean=0.285
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.274,mmce.test.mean=0.266
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.266,mmce.test.mean=0.303
```

```
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.267,mmce.test.mean=0.298
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.282,mmce.test.mean=0.26
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.287,mmce.test.mean=0.23
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.281,mmce.test.mean=0.283
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.278,mmce.test.mean=0.292
## [Resample] Aggr. Result: mmce.train.mean=0.276,mmce.test.mean=0.278
## Task: crime, Learner: classif.logreg.10
## [Resample] cross-validation iter 1:
## mmce.train.mean=0.277,mmce.test.mean=0.27
## [Resample] cross-validation iter 2:
## mmce.train.mean=0.271,mmce.test.mean=0.28
## [Resample] cross-validation iter 3:
## mmce.train.mean=0.271,mmce.test.mean=0.289
## [Resample] cross-validation iter 4:
## mmce.train.mean=0.274,mmce.test.mean=0.27
## [Resample] cross-validation iter 5:
## mmce.train.mean=0.268,mmce.test.mean=0.322
## [Resample] cross-validation iter 6:
## mmce.train.mean=0.27,mmce.test.mean=0.298
## [Resample] cross-validation iter 7:
## mmce.train.mean=0.279,mmce.test.mean=0.26
## [Resample] cross-validation iter 8:
## mmce.train.mean=0.284,mmce.test.mean=0.234
## [Resample] cross-validation iter 9:
## mmce.train.mean=0.275,mmce.test.mean=0.283
## [Resample] cross-validation iter 10:
## mmce.train.mean=0.272,mmce.test.mean=0.292
## [Resample] Aggr. Result: mmce.train.mean=0.274,mmce.test.mean=0.28
```

```r
plotLearningCurve(lcd, facet = "learner")
```



The train performance is measured on the actual fraction of the training set, whereas the performance of the testing set is measured on the total testing set. The large difference between the training and testing error of the large decision tree shows us that ridiculous amounts of extra training data would be needed to close the gap. The overcomplicated model overfits the training set badly.

The low variance logistic regression model shows performance values of the mean error rate that are comparable in the training and validation set from early on (from about 20% of the training data). We clearly do not need more training data, but may benefit from making a more flexible model.

Because the decision tree is grown too large, the discrepancy between the training and validation error becomes exaggerated over time, indicating a large degree of overfitting.

Filters

Filter values for filtering features can be generated using `generateFilterValuesData`. Use `listFilterMethods()` to get a complete list of all implemented methods.

Here, we use `information.gain` and `chi-squared`. Information gain H is defined as $H(class) + H(attribute) - H(class, attribute)$, where $H(X) = -\sum_x p(x)log(p(x))$. Chi-squared calculated the chi-squared statistic for a a test of independence between the variable (attribute) and the outcome. For both measures, continuous variables are by default automatically categorized into 5 (equally distributed) classes.

```
## Important note: this requires a working JAVA runtime environment (JRE)!!
## The java architecture (x64, x32) must match the version of R (64-bit or 32-bit)
fv <- generateFilterValuesData(crime.task, method = c("information.gain","chi.squared"))
fv$data
```

```
##            name    type information.gain chi.squared
```

```
## 1         SEX  factor     2.639649e-06 0.002298535
## 2 COUNTRYBIRTH  factor     8.626015e-03 0.130620466
## 3          AGE numeric    1.397613e-02 0.161003608
## 4   AGE1STCASE numeric    5.757623e-02 0.335832589
## 5    CRIMETYPE  factor     9.733595e-03 0.138861442
## 6    PREVCASES numeric    8.377661e-02 0.404297737
```

Wrappers

Finally, we merge a learner with a variable selection procedure (confusingly also called a wrapper), and tune the hyperparameters of the model.

```
dtree.filt <- makeFilterWrapper(learner = dtree, fw.method = "information.gain", fw.threshold = 1e-3)
rdesc <- makeResampleDesc(method = "CV", iters = 5 ) #create resampling strategy object
ps <-  makeParamSet(  makeDiscreteParam("cp", values = c(0.0001,0.001,0.01,0.1)),
                      makeDiscreteParam("minbucket", values = c(5,10,15,20)))
ctrl <- makeTuneControlGrid() #needed for actually making a grid of all combinations out of the above.
res <- tuneParams(learner = dtree.filt, task = crime.task, resampling = rdesc, par.set = ps,
  control = ctrl, measures = list(acc))
```

```
## [Tune] Started tuning learner classif.rpart.filtered for parameter set:

##               Type len Def               Constr Req Tunable Trafo
## cp        discrete   -   - 1e-04,0.001,0.01,0.1   -    TRUE     -
## minbucket discrete   -   -          5,10,15,20   -    TRUE     -

## With control class: TuneControlGrid

## Imputation value: -0

## [Tune-x] 1: cp=1e-04; minbucket=5

## [Tune-y] 1: acc.test.mean=0.688; time: 0.0 min

## [Tune-x] 2: cp=0.001; minbucket=5

## [Tune-y] 2: acc.test.mean=0.711; time: 0.0 min

## [Tune-x] 3: cp=0.01; minbucket=5

## [Tune-y] 3: acc.test.mean=0.726; time: 0.0 min

## [Tune-x] 4: cp=0.1; minbucket=5

## [Tune-y] 4: acc.test.mean=0.689; time: 0.0 min

## [Tune-x] 5: cp=1e-04; minbucket=10

## [Tune-y] 5: acc.test.mean=0.702; time: 0.0 min

## [Tune-x] 6: cp=0.001; minbucket=10

## [Tune-y] 6: acc.test.mean=0.714; time: 0.0 min

## [Tune-x] 7: cp=0.01; minbucket=10

## [Tune-y] 7: acc.test.mean=0.726; time: 0.0 min

## [Tune-x] 8: cp=0.1; minbucket=10

## [Tune-y] 8: acc.test.mean=0.689; time: 0.0 min

## [Tune-x] 9: cp=1e-04; minbucket=15
```

```
## [Tune-y] 9: acc.test.mean=0.717; time: 0.0 min
## [Tune-x] 10: cp=0.001; minbucket=15
## [Tune-y] 10: acc.test.mean=0.719; time: 0.0 min
## [Tune-x] 11: cp=0.01; minbucket=15
## [Tune-y] 11: acc.test.mean=0.726; time: 0.0 min
## [Tune-x] 12: cp=0.1; minbucket=15
## [Tune-y] 12: acc.test.mean=0.689; time: 0.0 min
## [Tune-x] 13: cp=1e-04; minbucket=20
## [Tune-y] 13: acc.test.mean=0.718; time: 0.0 min
## [Tune-x] 14: cp=0.001; minbucket=20
## [Tune-y] 14: acc.test.mean=0.723; time: 0.0 min
## [Tune-x] 15: cp=0.01; minbucket=20
## [Tune-y] 15: acc.test.mean=0.726; time: 0.0 min
## [Tune-x] 16: cp=0.1; minbucket=20
## [Tune-y] 16: acc.test.mean=0.689; time: 0.0 min
## [Tune] Result: cp=0.01; minbucket=15 : acc.test.mean=0.726
```

```r
##fit the final model
dtree <- setHyperPars(dtree, par.vals = res$x)
dtree.filt.final <- makeFilterWrapper(dtree, fw.method = "information.gain", fw.threshold = 1e-3)
dtree.filt.final <- train(dtree.filt.final, task = crime.task)
getFilteredFeatures(dtree.filt.final) #obtain the finally chosen variables.
```

```
## [1] "COUNTRYBIRTH" "AGE"          "AGE1STCASE"   "CRIMETYPE"
## [5] "PREVCASES"
```

For all clarity, the filter will be applied in each cv-iteration separately. In the final model, we can see SEX has been dropped. We can also tune for the optimal threshold of the filter by including `fw.perc"` as a parameter in `makeParamSet`.