The TDMR 2.2 Tutorial: Examples for Tuned Data Mining in R

Wolfgang Konen, Patrick Koch,
TH Köln – University of Applied Sciences

Initial version: June, 2012
Last update: March, 2020

Contents

1 Overview

The TDMR framework is written in R with the aim to facilitate the training, tuning and evaluation of data mining (DM) models. It puts special emphasis on tuning these data mining models as well as simultaneously tuning certain preprocessing options.

This document (TDMR-tutorial.pdf)

- describes the TDMR **installation**
- shows **example usages**: how to use TDMR on new data mining tasks
- provides a **FAQ-section** (frequently asked questions)

This document should be read in conjunction with the companion document TDMR-docu.pdf (Konen and Koch, 2012a), which describes more details and software concepts of TDMR.

Both documents are available online as CIOP Reports (PDF, Konen and Koch (2012a,b)) from http://www.gm.fh-koeln.de/ciopwebpub.[1]

Both documents concentrate more on the software usage aspects of the TDMR package. For a more scientific discussion of the underlying ideas and the results obtained, the reader is referred to Konen et al. (2010, 2011); Konen (2011); Koch et al. (2012); Koch and Konen (2012); Stork et al. (2013); Koch and Konen (2013); Koch et al. (2015).

[1]The precise links are http://www.gm.fh-koeln.de/ciopwebpub/Kone12a.d/Kone12a.pdf and http://www.gm.fh-koeln.de/ciopwebpub/Kone12b.d/Kone12b.pdf. The same files are available as well via the index page of the TDMR package (User guides and package vignettes).

2 Installing TDMR

Once you have R (http://www.r-project.org/), > 2.14, up and running, simply install TDMR with

```
install.packages("TDMR");
```

Then, library TDMR is loaded with

```
library(TDMR);

## Loading required package:  SPOT
## Loading required package:  twiddler
## Loading required package:  tcltk
```

3 Lessons

NOTE: Many, but not all TDMR demos and functions will run under RStudio. That some demos are not running under RStudio is due to some incompatibilities in RStudio's graphic device(s). All demos and functions will however run under `RGui`.

To start a demo, e.g. `demo/demo00-0classif.r`, type

```
demo("demo00-0classif")
```

or

```
demo("demo00-0classif",ask=F)
```

or load the appropriate demo `R` script in RStudio and 'source' it.

3.0 Lesson 0: A simple TDMR program

`demo/demo00-0classif.r`
`demo/demo00-1regress.r`

This demo shows the most simple TDMR program. It does not need any external files.

```
#*# --------- demo/demo00-0classif.r ---------
# set all defaults for data mining process:
opts=tdmOptsDefaultsSet()
opts$TST.SEED <- opts$MOD.SEED <- 5     # reproducible results
gdObj <- tdmGraAndLogInitialize(opts);  # init graphics and log file
```

```
data(iris)
response.vars="Species"                    # names, not data (!)
input.vars=setdiff(names(iris),"Species")

result = tdmClassifyLoop(iris,response.vars,input.vars,opts)

print(result$Err)
```

Here, `tdmOptsDefaultsSet` will construct a default list `opts` with all relevant settings. See TDMR-docu.pdf (Konen and Koch, 2012a), Appendix B, for a complete list of all elements and all defaults for list `opts`. After initializing graphics and log file, the dataset `iris` is loaded and the target (`Species`) as well as the input variables (all other column names from `iris`) are defined.

Now the classification DM task is started with `tdmClassifyLoop`.

Inside `tdmClassifyLoop` the following things happen:

Data partitioning: The dataset will be divided by random sampling in a training set (90%) and validation set (10%), based on `opts$TST.kind="rand", opts$TST.valiFrac=0.1`.

Variable selection: Since you do not specify anything from the `opts$SRF`-block (**s**orted **r**andom **f**orest importance), you use the default SRF variable ranking (`opts$SRF.kind ="xperc"`, `opts$SRF.Xperc=0.95`). This means that the most important columns (containing in sum at least 95% of the overall importance) will be selected.

Modeling and evaluation: Since you do not specify anything else, function `tdmClassifyLoop` builds an RF (`randomForest`) model (`opts$MOD.method="RF"`) using the training data and evaluates it on training and validation data. It returns an object `result`. The object `result` of class `TDMclassifier` is explained in more detail in Table 3 of TDMR-docu.pdf (Konen and Koch, 2012a).

Repeated runs: Since the default setting `opts$NRUN=2` is used, the whole procedure (random partitioning into training and validation set, RF-based selection of the most important variables, model building, and model evaluation) is repeated 2 times in 2 runs with different random seeds (yielding different data partitions & different split decisions in RF). The different runs are aggregated (usually by averaging).

We now take a look at the output generated by `tdmClassifyLoop`. Since we do not change the default `opts$VERBOSE=2`, TDMR will print a lot of diagnostic output:

```
## default.txt : Stratified random training-validation-index with opts$TST.valiFrac =  10 %
##
## default.txt : Importance check ...
## Clipping sampsize to  135
## default.txt : Train RF (importance, sampsize= 135 ) ...
## default.txt : Saving SRF (sorted RF) importance info on opts ...
## Variables sorted by importance (4 ):
## [1] "Petal.Length" "Petal.Width"  "Sepal.Length" "Sepal.Width"
## Dropped columns (0 [=  0.0% of total importance]):
```

```
## Proc time:  0.01
## Clipping sampsize to  135
## Run  1 / 2 :
## default.txt : Train RF with sampsize = 135 ...
## Proc time:  0.02
## default.txt : Apply RF ...
## Proc time:  0.02
## default.txt : Calc confusion matrix + gain ...
##
## Training cases ( 135 ):
##             predicted
## actual       setosa versicolor virginica
##   setosa         45          0         0
##   versicolor      0         42         3
##   virginica       0          3        42
## total gain:   129.0 (is  95.556% of max. gain =   135.0)
##
## Validation cases (15):
##             predicted
## actual       setosa versicolor virginica
##   setosa          5          0         0
##   versicolor      0          5         0
##   virginica       0          1         4
##             setosa versicolor virginica Total
## gain.vector      5          5         4    14
## total gain :    14.0 (is  93.333% of max. gain =    15.0)
##
##   Relative gain (rgain) on   training set    95.55556 %
##   Relative gain (rgain) on validation set    93.33333 %
##
## default.txt : Stratified random training-validation-index with opts$TST.valiFrac =  10 %
##
## default.txt : Importance check ...
## Clipping sampsize to  135
## default.txt : Train RF (importance, sampsize= 135 ) ...
## default.txt : Saving SRF (sorted RF) importance info on opts ...
## Variables sorted by importance (4 ):
## [1] "Petal.Length" "Petal.Width"  "Sepal.Length" "Sepal.Width"
## Dropped columns (1 [=  1.0% of total importance]):
## [1] "Sepal.Width"
## Proc time:  0.00
## Clipping sampsize to  135
## Run  2 / 2 :
## default.txt : Train RF with sampsize = 135 ...
## Proc time:  0.03
## default.txt : Apply RF ...
## Proc time:  0
## default.txt : Calc confusion matrix + gain ...
##
## Training cases ( 135 ):
##             predicted
## actual       setosa versicolor virginica
##   setosa         45          0         0
##   versicolor      0         42         3
##   virginica       0          4        41
## total gain:   128.0 (is  94.815% of max. gain =   135.0)
##
## Validation cases (15):
##             predicted
## actual       setosa versicolor virginica
##   setosa          5          0         0
##   versicolor      0          5         0
##   virginica       0          0         5
##             setosa versicolor virginica Total
## gain.vector      5          5         5    15
## total gain :    15.0 (is 100.000% of max. gain =    15.0)
##
##   Relative gain (rgain) on   training set    94.81481 %
##   Relative gain (rgain) on validation set    100 %
##
##
## Average over all  2  runs:
## cerr$train: (4.81481 +- 0.52378)%
## cerr$vali:  (3.33333 +- 4.71405)%
## gain$train: ( 128.50 +- 0.71)
## gain$vali:  (  14.50 +- 0.71)
## rgain.train:  95.185%
## rgain.vali:   96.667%
##        cerr.trn gain.trn rgain.trn ntrn   cerr.tst gain.tst rgain.tst
## [1,] 0.04444444      129  95.55556  135 0.06666667       14  93.33333
## [2,] 0.05185185      128  94.81481  135 0.00000000       15 100.00000
##       cerr.tst2 gain.tst2 rgain.tst2 ntst
## [1,] 0.06666667        14   6.222222   15
## [2,] 0.00000000        15   6.666667   15
```

The first line tells us that TDMR has set aside 10% of the data (15 records in the case of `iris` with 150 records) for validation, the remaining 135 are for training. A random forest is trained to assess the importance of the input variables. We get with

```
[1] "Petal.Width" "Petal.Length" "Sepal.Length" "Sepal.Width"
```

the variables sorted by decreasing importance. It depends on the importance of the least important variable (here: `Sepal.Width`) whether it will be dropped or not. In the first run it is not dropped, because its importance is above the threshold $1-0.95 = 5\%$. In the second run it is dropped, because due to statistical fluctuations now its importance is with 0.5% below the threshold of 5%.

In the next step the DM model (here: RF) is trained with the selected variables and then the trained model is applied to the training data and to the validation data. In each case the confusion matrix (actual vs. predicted) is shown. The confusion matrices are below the lines `Training cases (135)` and `Validation cases (15)`, resp. In the case of RF, the prediction on the training data is the OOB prediction.

Next, the `total gain` is reported as the sum of the element-wise product „gain matrix $G\times$ confusion matrix C" where the **gain matrix** denotes for every classification outcome „actual vs. predicted" the associated gain.[2] If nothing else is said, the gain matrix is the identity matrix. In this case, relative gain is equivalent to the classification accuracy (percent of correctly classified records). The `relative gain` is defined as

$$\texttt{rgain} = \frac{\sum_{ij} G_{ij} C_{ij}}{\sum_{ij} G_{ij} C_{ij}^{(ideal)}}$$

with G = gain matrix, C = confusion matrix and $C^{(ideal)}$ = perfect confusion matrix (all records appear on the main diagonal).

As the final output from `tdmClassifyLoop`, below the line `Average over all 2 runs`, all runs (2 in this example) are averaged and the average classification error `cerr`, the average `gain`, and the average relative gain `rgain` are reported for training and validation set.

A similar information, but now for each run separately, is provided with the last statement in the demo program

```
print(result$Err)
```

which gives for each run separately classification error (`cerr`), gain (`gain`), and relative gain (`rgain`) on the training set (`.trn`) with `ntrn=135` training records and on the test set (`.tst`) with `nst=15` records.[3]

```
##        cerr.trn gain.trn rgain.trn ntrn   cerr.tst gain.tst rgain.tst
## [1,] 0.04444444      129  95.55556  135 0.06666667       14  93.33333
## [2,] 0.05185185      128  94.81481  135 0.00000000       15 100.00000
##       cerr.tst2 gain.tst2 rgain.tst2 ntst
## [1,] 0.06666667        14   6.222222   15
## [2,] 0.00000000        15   6.666667   15
```

If you add the line

[2]In this toy problem, the gain on the validation set is statistically not very meaningful since the validation set has only 15 records.

[3]The columns with `.tst2` refer to a test set with special postprocessing, see TDMR-docu.pdf and the TDMR manual pages for details.

```
opts$VERBOSE <- opts$SRF.verbose <- 0
```

before calling `tdmClassifyLoop`, then `tdmClassifyLoop` is completely silent. The only output you get is the printout of `result$Err`.

A similar demo program for regression is found in `demo/demo00-1regress.r`.

3.1 Lesson 1: DM on task SONAR

`demo/demo01-1sonar.r`
`demo/demo01-2cpu.r`

Now we want to conduct a data mining process with a pre-defined parameter set different from the defaults.

This lesson demonstrates the usage of TDMR for a somewhat bigger DM task: data are read from file and the information for controlling TDMR is distributed over two files. This may look complicated at first sight, but it is useful for two reasons:

Separate DM function file: As a preparation for the tuning process in subsequent lessons: It is very useful if we can package the whole data mining (DM) process (from training-validation-data generation over model building up to model evaluation) into one function or file. It will be easily callable by the tuner. Additionally it should provide a separate data reading function.

Separate starter file: For conducting slightly different runs or experiments, it is useful to package the parameter setting part together with the starter commands in another file.

In this lesson these files are:

1. `main_sonar.r` (the DM function file containing `main_sonar` + data reading functions)

2. `demo01-1sonar.r` (parameter settings + demo starter)

Suppose that you have a dataset and want to build a DM model for it. To be concrete, we consider the classification dataset SONAR[4]. The data file `sonar.txt` should be in the subdirectory `opts$path/opts$dir.data`.

If you want to build a DM classification model with TDMR, you need to provide two files, `demo01-1sonar.r` and `main_sonar.r`.[5] The first file, `demo01-1sonar.r` defines function `controlDM` the list `opts` with all settings relevant for the DM model building process. The second file, `main_sonar.r`, contains this DM model building process. It gets with list `opts` the settings and returns in list `result` the evaluation of the DM model. The list `result` is either inspected by the user or by the tuning process.

The file `main_sonar.r` contains two functions `main_sonar` and `readTrnSonar`:

[4]see UCI repository or package mlbench for further info on SONAR)

[5]Templates for `main_sonar.r` are available from `<inst>/demo02sonar` where `<inst>` refers to the installation directory of package TDMR as returned by `find.package("TDMR")`.

```
main_sonar <- function(opts, dset=NULL, tset=NULL) {
  opts <- tdmOptsDefaultsSet(opts);    # fill in all opts params not yet set

  gdObj<-tdmGraAndLogInitialize(opts); # init graphics and log file

  ########  PART 1: READ DATA    ##########################
  if (is.null(dset)) {
      cat1(opts,opts$filename,": Read data ...\n")
      dset <- tdmReadDataset(opts);
  }
  names(dset)[61] <- "Class"  # 60 columns V1,...,V60 with input data, one
                              # response column "Class" with levels ["M"|"R"]

  response.vars <- "Class"             # which variable(s) are target

  # which variables are input variables (in this case all others):
  input.vars <- setdiff(names(dset), c(response.vars))

  ########  PART 2: Model building and evaluation #########
  result <- tdmClassifyLoop(dset,response.vars,input.vars,opts,tset);

  # print summary output and attach certain columns
  # (here: y, sd.y, dset) to list result:
  result <- tdmClassifySummary(result,opts,dset);

  tdmGraAndLogFinalize(opts,gdObj);  # close graphics and log file

  result;
}

readTrnSonar <- function(opts) {
  read.csv2(file=paste(opts$path,opts$dir.data, opts$filename, sep="/"),
            dec=".", sep=",", nrow=opts$READ.NROW,header=FALSE);
}
```

This file will be sourced by `demo01-1sonar.r` and **after** this we define with function `controlDM()` the relevant settings for the DM process. [6]

```
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");

source(paste(path,"main_sonar.r",sep="/"));      # needed to define readTrnSonar

controlDM <- function() {
```

[6]Why *after*? - The function `readTrnSonar`, which is defined in `main_sonar.r`, has to be known as a function.

```
  #
  # settings for the DM process (former sonar_00.apd file):
  # (see ?tdmOptsDefaultsSet for a complete list of all default settings
  # and many explanatory comments)
  #
  opts = list(path = path,
              dir.data = "data",             # relative to path
              filename = "sonar.txt",
              READ.TrnFn = readTrnSonar,     # defined in main_sonar.r
              data.title = "Sonar Data",
              NRUN =  1,            # how many runs or CV-runs
              VERBOSE = 2
  );

  opts <- setParams(opts, defaultOpts(), keepNotMatching = TRUE);
}
```

After specifying some specific settings for the list `opts`, all other settings will be filled in via `defaultOpts()`. See TDMR-docu.pdf Konen and Koch (2012a), Appendix B, for a complete list of all elements and all defaults for list `opts`. You need to specify only those things which differ from `defaultOpts()`: in this case most importantly the filename and directory of the SONAR dataset and a function `opts$READ.TrnFn` containing the data-reading command. The parameter `keepNotMatching` ensures that the parameters in `opts` which are not found in `defaultOpts` (because their default value is `NULL`) are nevertheless retained.

Now the whole process is invoked with the remainder of `demo01-1sonar.r`:

```
## --------- demo/demo01-1sonar.r ---------
opts <- controlDM();
result <- main_sonar(opts);
```

See Fig. 1 and following for some output plots produced by this demo lesson.

The above lesson `demo-demo01sonar.r` showed a classification task. For a similar regression task on dataset CPU see `demo-demo01cpu.r`.

3.2 Lesson 2: SPOT tuning on task SONAR

`demo/demo02sonar.r`

In this lesson we not only want to run the data mining process for a fixed parameter set as in Lesson 1 (Sec. 3.1), but we want to *tune the parameters*, i. e. to find good or optimal parameters within a certain range, the *region of interest* `alg.roi`.

If you want to do a SPOT tuning (Bartz-Beielstein, 2010) on task SONAR, you should follow the steps described in TDMR-docu.pdf (see Konen and Koch (2012a), Sec. 2.2 *TDMR Workflow, Level 2*) and create in addition to `main_sonar.r` and `opts` from Lesson 1 a SPOT

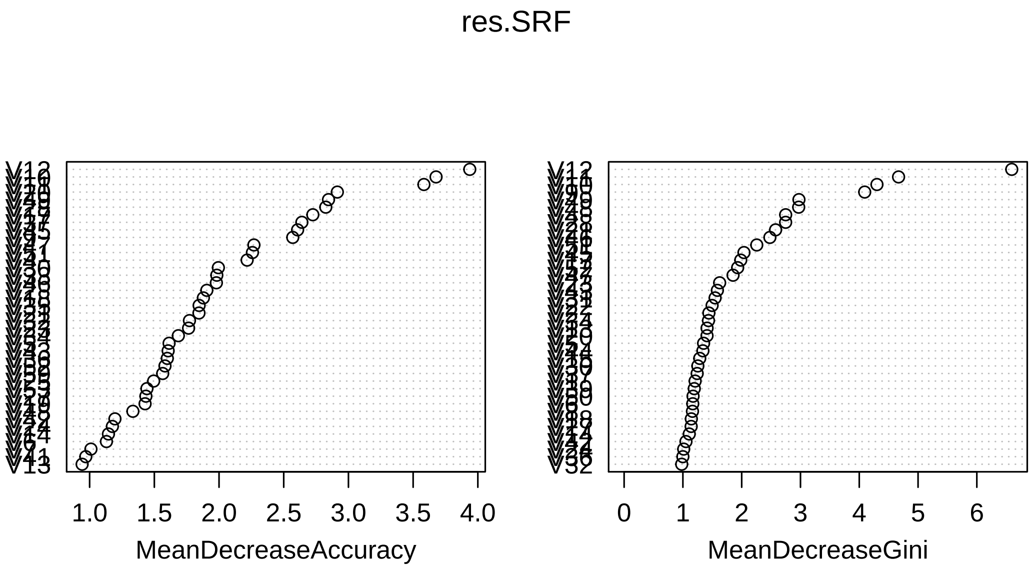


Figure 1: Some plots from demo01-1sonar.r

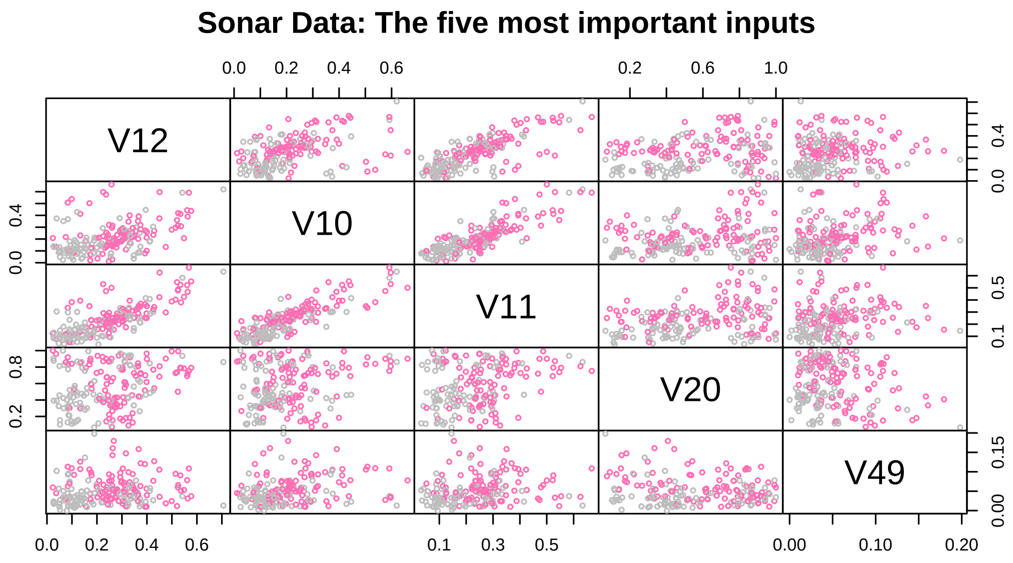


Figure 2: Some plots from demo01-1sonar.r

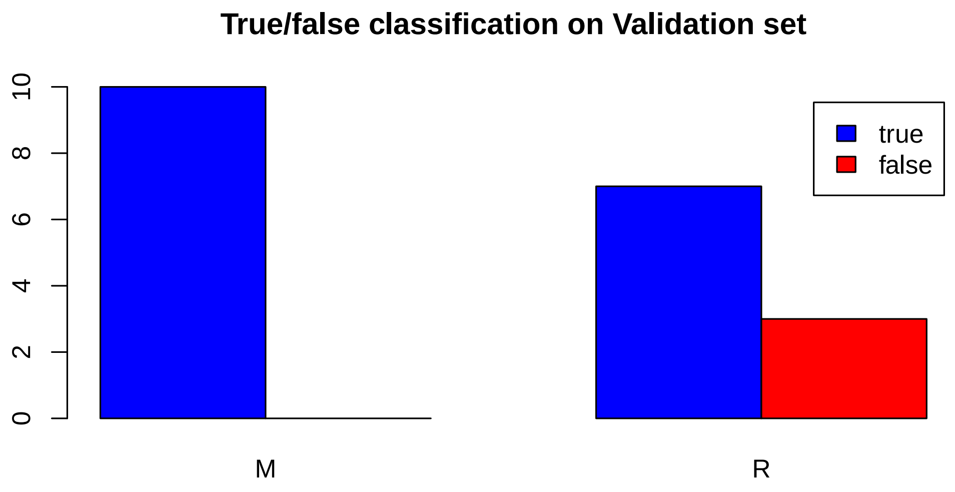


Figure 3: Some plots from demo01-1sonar.r

configuration object `ctrlSC` and a TDMR configuration object `tdm`. This can be all done within `demo02sonar.r`:

First we define a path and minimal `tdm` and `opts` objects similar to Lesson 1 (Sec. 3.1):

```
## path is the dir with data and main_*.r file:
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");

tdm=list(mainFile="main_sonar.r"
         ,path=path
         ,filenameEnvT="envTSonar.RData"  # file to save envT (in dir path)
         ,umode="SP_T"
         ,U.saveModel=FALSE
         ,tuneMethod="spot" # "spot", "cma_j", "lhd"
);
source(paste(path,tdm$mainFile,sep="/"))

controlDM <- function() {
  #
  # settings for the DM process (former .apd file):
  #
  opts = list(path = path,
              dir.data = "data/",
              filename = "sonar.txt",
              READ.TrnFn = readTrnSonar,    # defined in main_sonar.r
              data.title = "Sonar Data",
              TST.SEED = 999,
              MOD.SEED = 999,
              RF.mtry = 4,
              CLS.cutoff = c(0.5,-1),
              CLS.CLASSWT = c(10,10),
              NRUN =  1,        # how many runs with different train & test samples
                                # - or - how many CV-runs, if TST.kind="cv"
              GD.DEVICE="non",  # ["pdf"|"win"|"non"]: all graphics to
                                # [one multi-page PDF | windows (X11) | dev.null]
              GD.RESTART=F,
              VERBOSE = 0,
              SRF.verbose = 0,
              logFile=FALSE     # no logfile (needed for Sweave/.Rnw only)
              );

  opts <- setParams(opts, defaultOpts(), keepNotMatching = TRUE);
}
```

Usually you should set `opts$GRAPHDEV="non"` and `opts$GD.RESTART=F` to avoid any graphic output and any graphics device closing from `main_sonar.r`, so that you get only the graphics made by SPOT.

Next, we define tuner settings (ROI and others):

```
controlSC <- function() {
  ctrlSC = list(alg.roi=data.frame(lower=c(0.1, 5,0.9),
                                   upper=c(0.8,15,1.0),
                                   type=rep("FLOAT",3),
                                   row.names=c("CUTOFF1","CLASSWT2","XPERC"))
              ,funEvals = 20
              ,designControl.size = 4
              ,seq.merge.func = mean   # mean or min
              ,replicates = 2
              ,noise=TRUE
              ,sCName="sonar_02.conf"
  );

  ctrlSC <- setParams(ctrlSC,defaultSC());
  # defaultSC() fills in sensible defaults for all other controls
  ctrlSC;
}
```

The three parameters `CUTOFF1`, `CLASSWT2` and `XPERC` are tuned within the borders specified by `ctrlSC$alg.roi`. According to Appendix A in TDMR-docu.pdf (Konen and Koch, 2012a) these values are mapped to `opts$CLS.cutoff[1]`, `opts$CLS.CLASSWT[2]` and `opts$SRF.XPerc`, resp. All other parameters are set in `controlDM`: `opts$CLS.cutoff[2]` is set to -1 (meaning: „remainder to 1.0") and `opts$CLS.CLASSWT[1]` is set to 10.

Note that for an n-class problem you should tune at most $n-1$ `CUTOFF` parameters and $n-1$ `CLASSWT` parameters. See Sec. 7.2 „cutoff" and 7.3 „classwt" in TDMR-docu.pdf (Konen and Koch, 2012a) for more details.

To start the whole procedure, we use the remainder of `demo02sonar.r`:

```
ctrlSC <- controlSC();
ctrlSC$opts <- controlDM();

# construct envT from settings given in tdm & sCList:
envT <- tdmEnvTMakeNew(tdm,sCList=list(ctrlSC));
dataObj <- tdmReadTaskData(envT,envT$tdm);
envT <- tdmBigLoop(envT,dataObj=dataObj);      # start the big tuning loop
```

What happens in this demo is the following:

- The first command sets `ctrlSC`, the controls for the tuner (here: SPOT).
- The second command adds the control for the data mining (DM) process.
- With the command `tdmEnvTMakeNew(tdm)` we construct an environment `envT` with all necessary TDMR data and functions for this lesson. Inside `tdmEnvTMakeNew` the minimal `tdm` is filled with all defaults (see `tdmDefaultsFill`).
- Function `tdmReadTaskData` reads the data from disk, so that no further file access is needed after this point. It splits them in a train/vali and a test part.
- The command `tdmBigLoop` is the main workhorse. It calls the desired tuner(s) (in this case only SPOT, `tdm$tuneMethod="spot"`, but `tdm$tuneMethod` could be a vector of tuners as well). Each tuner performs multiple DM runs in order to find the best values for the tunable parameters defined in the `ctrlSC$alg.roi`. The results of the whole tuning process are returned in `envT`. More details on `envT` are in the manual / help section for `tdmBigLoop` and in Table 5 and Sec. 5.4 of TDMR-docu.pdf (Konen and Koch, 2012a).

The resulting output after tuning is

```
Best solution:
      CUTOFF1 CLASSWT2     XPERC STEP CONFIG         Y
102 0.4808984 6.836568 0.9437066    6      7 -85.33333
Best Config:     7
```

which tells us that the best solution was found for design point CONFIG=7 with RGain=85.333 (the negative of column Y).

`tdmBigLoop` takes after tuning the parameters of the best solution and performs with it `NRUN=5` independent (unbiased) runs. The results are summarized in `envT$theFinals` which reads in this case:

```
  CONF TUNER NEXP   CUTOFF1 CLASSWT2     XPERC NEVAL RGain.bst RGain.avg
1  C01  spot    1 0.4808984 6.836568 0.9437066    20  85.33333  80.96667
  Time.TRN NRUN  RGain.TRN  sdR.TRN RGain.SP\_T sdR.SP\_T Time.TST
1     5.19    5       84.8 1.725624    78.04878  2.439024     1.09
```

The data frame `envT$theFinals` has in its columns 4-6 the best solution, found after a tuning budget of `NEVAL=20` model evaluations. `RGain.bst=85.333` is the best result from tuning and `RGain.avg=80.966` is the average tuning quality. The columns after `NRUN=5` show the results from the `NRUN` unbiased runs: RGain.TRN=84.8 is the mean relative gain on the training set and RGain.SP_T=78.048 is the mean RGain on the test set. It shows that `RGain.bst` was a bit too optimistic, the realistic RGain on independent test data is only 78.048 in this case. `sdR.*` are the standard deviations over the `NRUN=5` runs.

More details on `envT$theFinals` are in Table 2 of TDMR-docu.pdf (Konen and Koch, 2012a).

3.3 Lesson 3: „The Big Loop“ on task SONAR

`demo/demo03sonar.r`
`demo/demo03sonar_B.r`
`demo/demo03newdata.r`

3.3.1 Multiple Configurations

To start „The Big Loop“, you configure a file similar to `demo/demo03sonar.r`:

```
#*# --------- demo/demo03sonar.r, Part 1 ---------
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");

tdm <- list( mainFile="main_sonar.r"
           , umode="CV"              # { "CV" | "RSUB" | "TST" | "SP_T" }
           , tuneMethod="spot"       # "spot", "cma_j", "lhd", ...
           , path=path
           , filenameEnvT="demo03.RData"   # file to save envT (in dir path)
           , nrun=3, nfold=2         # repeats and CV-folds for unbiased runs
           , nExperim=1
           , parallelCPUs=1
           , parallelFuncs=c("readTrnSonar")
#            , U.saveModel=F
           , optsVerbosity = 3       # the verbosity for the unbiased runs
           );
```

```
source(paste(path,tdm$mainFile,sep="/"));
```

This is very much the same as in Lesson 2, Sec. 3.2. The only difference is that now **multiple** tuning runs can be performed with respect to the following three dimensions:

- configurations (see `controlSC..` below)
- tuners (elements of `tdm$tuneMethod`)
- repeated experiments with different random seeds (number `tdm$nExperim`).

The function `tdmBigLoop` realizes a triple `for`-loop over these dimensions. With k =`length(sCList)` (see below, parameter to `tdmEnvTMakeNew`), m =`length(tuneMethod)`, and n =`nExperim` we have in total kmn tuning runs.

Here, the script `demo03sonar.r` will trigger the following sequence of experiments:

- Configuration `sonar_04` (see `controlSC04`) with tuner `spot`
- Configuration `sonar_06` (see `controlSC06`) with tuner `spot`.

This sequence of 2 tuning experiments is repeated `nExperim=1` time. The corresponding 2 result lines are written to data frame `envT$theFinals`.

Next, we configure the controls, similarly to Lesson 2, but now for two slightly different configurations:

```
#*# --------- demo/demo03sonar.r, Part 2 ---------
controlDM <- function() {
  #
  # settings for the DM process (former sonar_04.apd & sonar_06.apd file):
  #
  opts = list(path = path,
              dir.data = "data/",
              filename = "sonar.txt",
              READ.TrnFn = readTrnSonar,    # defined in main_sonar.r
              READ.NROW=100,
              data.title = "Sonar Data",
              TST.SEED = 124,
              MOD.SEED = 124,
              CLS.cutoff = c(0.9,-1),
              SRF.cutoff = c(0.9,-1),
              NRUN =  1,        # how many runs with different train & test samples
                                # - or - how many CV-runs, if TST.kind="cv"
              GD.DEVICE="non",  # ["pdf"|"win"|"non"]: all graphics to
                                # [one multi-page PDF | windows (X11) | dev.null]
              GD.RESTART=F,
              VERBOSE = 0,
            SRF.verbose = 0,
            logFile=FALSE     # no logfile (needed for Sweave/.Rnw only)
  );
  opts <- setParams(opts, defaultOpts(), keepNotMatching = TRUE);
}

controlSC04 <- function() {
  #
  # settings for the tuning process (former sonar_04.roi and .conf file):
  #
  ctrlSC = list(alg.roi=data.frame(lower=c(0.0, 5,0.5),
                                   upper=c(0.1,15,1.0),
                                   type=rep("FLOAT",3),
                                   row.names=c("CUTOFF1","CLASSWT2","XPERC"))
            ,funEvals = 20
            ,designControl.size = 4
            ,optimizerControl.retries = 2   # optimLHD retries
                                            # (former seq.design.retries)
            ,replicates = 2
            ,noise=TRUE
            ,sCName="sonar_04.conf"
  );

  ctrlSC <- setParams(ctrlSC,defaultSC());
  # defaultSC() fills in sensible defaults for all other controls
  ctrlSC;
}

controlSC06 <- function() {
  #
  # settings for the tuning process (former sonar_06.roi and .conf file):
  #
  ctrlSC = list(alg.roi=data.frame(lower=c(0.0, 5,0.5),
                                   upper=c(0.5,15,1.0),
                                   type=rep("FLOAT",3),
                                   row.names=c("CUTOFF1","CLASSWT2","XPERC"))
            ,funEvals = 20
            ,designControl.size = 4
            ,replicates = 2
            ,noise=TRUE
            ,sCName="sonar_06.conf"
  );

  ctrlSC <- setParams(ctrlSC,defaultSC());
  # defaultSC() fills in sensible defaults for all other controls
  ctrlSC;
}
```

The difference between `controlSC04` and `controlSC06` is the different `alg.roi$upper[1]` and an additional `optimizerControl.retries` in `controlSC04`. The names given in `sCName` will be the names printed in column `CONF` of `envT$theFinals`.[7]

Finally, we start the whole process with these commands:

```
#*# --------- demo/demo03sonar.r, Part 3 ---------
ctrlSC04 <- controlSC04();
ctrlSC04$opts <- controlDM();
ctrlSC06 <- controlSC06();
ctrlSC06$opts <- controlDM();

# construct envT from settings given in tdm & sCList
envT <- tdmEnvTMakeNew(tdm,sCList=list(ctrlSC04,ctrlSC06));
dataObj <- tdmReadTaskData(envT,envT$tdm);
envT <- tdmBigLoop(envT,dataObj=dataObj);     # start the big tuning loop
```

The results are reported in data frame `envT$theFinals`:

```
print(envT$theFinals);

##        CONF TUNER NEXP    CUTOFF1 CLASSWT2     XPERC NEVAL RGain.bst RGain.avg
## 1 sonar_04  spot    1 0.07872958 9.126950 0.6928838    20  92.77778  89.66667
## 2 sonar_06  spot    1 0.29708521 8.135792 0.8448112    20  97.77778  96.27778
##   Time.TRN NRUN RGain.TRN   sdR.TRN RGain.CV   sdR.CV Time.TST
## 1     1.75    3  86.89796 3.611265 84.66667 4.041452     0.31
## 2     1.95    3  97.64626 2.101281 95.33333 1.527525     0.30
```

Here `CUTOFF1`, `CLASSWT2`, and `XPERC` are the tuning parameters, the other columns of the data frame are defined in Table 2 of TDMR-docu.pdf (Konen and Koch, 2012a). Note that for an n-class problem you will specify at most $n-1$ `CUTOFF` parameters and $n-1$ `CLASSWT` parameters. See Sec. 7.2 „cutoff“ and 7.3 „classwt“ in TDMR-docu.pdf (Konen and Koch, 2012a).

In the case of the example above, the tuning process had a budget of `NEVAL=10` model trainings, resulting in a best solution with class accuracy `RGain.bst` (in %). The average class accuracy (mean w.r.t. all design points) during tuning is `RGain.avg`. When the tuning is finished, the best solution is taken and `NRUN=3` unbiased evaluation runs are done with the parameters of the best solution. The mean classification accuracy `RGain.TRN` from the 3 training runs is returned.[8] Additionally, `NRUN=3` trainings are done with cross validation (CV) with new randomly created folds in each run, resulting in an average class accuracy `RGain.CV`.

[7] If `sCName` is missing, TDMR will generate generic defaults like `C01.conf`, `C02.conf`, ...

[8] Since the classification model in this example is RF (Random Forest), `RGain.TRN` refers to the OOB-error.

For each measure `RGain.*` there is also an accompanying column `sdr.*` giving the standard deviation with respect to the `NRUN` runs.

Tuning runs are rather short, to make this example run quickly. Do not expect good numeric results. See `demo/demo03sonar_B.r` for a somewhat longer tuning run, with two tuners SPOT and LHD.

3.3.2 Single Configuration

If you have only a single CONF file it is recommended that you use `tdmTuneIt` instead of `tdmBigLoop`. `tdmTuneIt` has `dataObj` as a mandatory calling parameter. This makes it easier to build up your task since it has a clearer data flow concept behind.

See Lesson 8 (Sec. 3.8: **Tuning with fewer data**) for a fully worked-out example with `tdmTuneIt`.

Above we used `tdmBigLoop` (and not `tdmTuneIt`), since we have two configurations `sonar_04` and `sonar_06`. `tdmBigLoop` will find the best tuning parameters for each tuning run and perform with it unbiased (training and evaluation) runs.[9]

3.3.3 New Data and Sensitivity Plot

Another possibility to predict on new data is shown in the following example from demo `demo03newdata`:

```
#*# --------- demo/demo03newdata.r ---------
require(randomForest);
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");
envT <- tdmEnvTLoad("demoSonar.RData",path);
source(paste(path,"main_sonar.r",sep="/"));
opts <- tdmEnvTGetOpts(envT);
opts$READ.NROW=-1;
opts$path <- path;
envT <- tdmEnvTSetOpts(envT,opts);
dataObj <- tdmReadAndSplit(opts,envT$tdm); # read data, needs readTrnSonar
envT$tdm$nrun=1;     # =0: no unbiasedRun,
                     # >0: perform unbiasedRun with opts£NRUN=envT£tdm£nrun
envT <-  tdmEnvTReport(envT,1);            # requires randomForest
```

We first read the tuning results stored in `demoSonar.RData`. Then we read data (potentially other than those used during tuning) on `dataObj`. The call

```
envT <-  tdmEnvTReport(envT,1);
```

[9]Note that the dataset `dataObj`, when specified in `tdmBigLoop`, is used for **every** run (every CONF file) in the big loop. If `dataObj` were not specified in the call to `tdmBigLoop`, each configuration would construct its own `dataObj` inside `tdmBigLoop`.

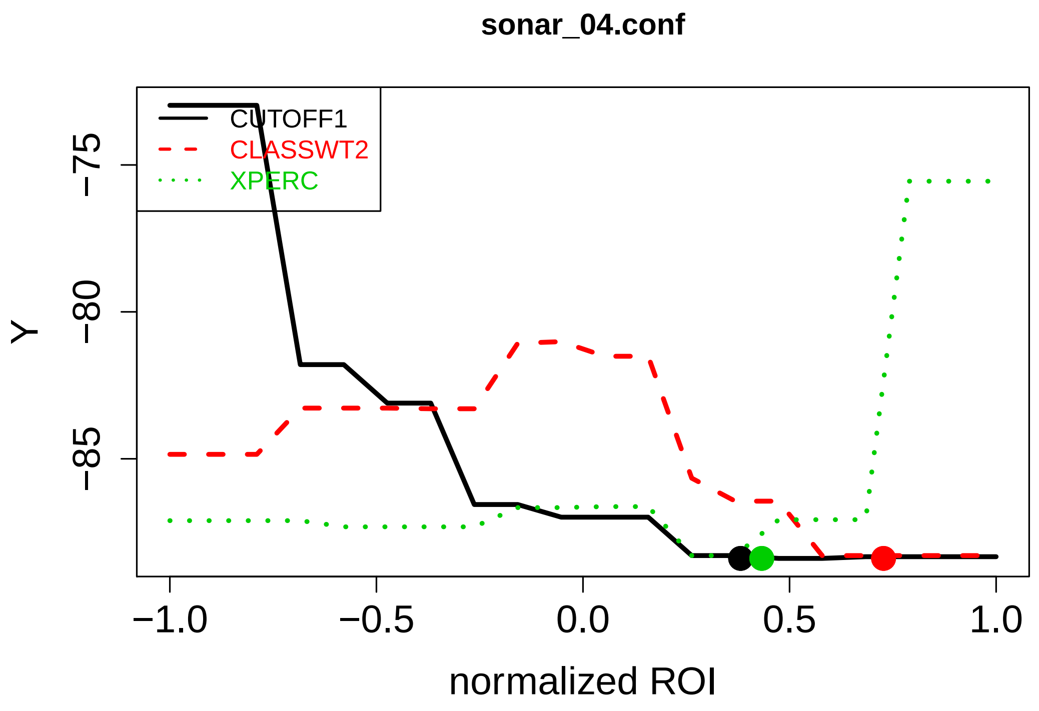


Figure 4: Sensitivity plot for the parameters tuned on dataset Sonar: The x-axis has the ROI of each parameter mapped to the interval [-1.0, 1.0]. Each curve shows Y (usually RGain) as a function of the parameter varied in its ROI while the other parameters are at their values for the best design points (marked by the circles).

```
## Warning in file(file, "rt"):  cannot open file 'C:/Users/wolfgang/Documents/R/win-library/3.4/TD
No such file or directory
## Error in file(file, "rt"):  cannot open the connection
```

does two things:

1. It makes a sensitivity plot (Fig. 4) for the tuning run 1 (here: first run with CONF `sonar_04` and tuner `spot`).
2. It makes `envT$tdm$nrun=1` unbiased evaluation runs on the new data, using the best parameters from tuning run 1.

The results of the new unbiased evaluation runs are recorded in `envT$theFinals`:

```
print(envT$theFinals);

##       CONF TUNER NEXP    CUTOFF1 CLASSWT2     XPERC NEVAL RGain.bst RGain.avg
## 1 sonar_04  spot    1 0.06907716 13.63763 0.8581948    10  91.66667  77.66667
##   Time.TRN NRUN RGain.TRN sdR.TRN RGain.CV sdR.CV Time.TST
## 1     0.96    1  54.32692      NA 55.76923     NA     0.42
```

3.4 Lesson 4: Regression Big Loop

`demo/demo04cpu.r`

The same as Lesson 3, but applied to a regression task (dataset CPU).

3.5 Lesson 5: Performance Measure Plots

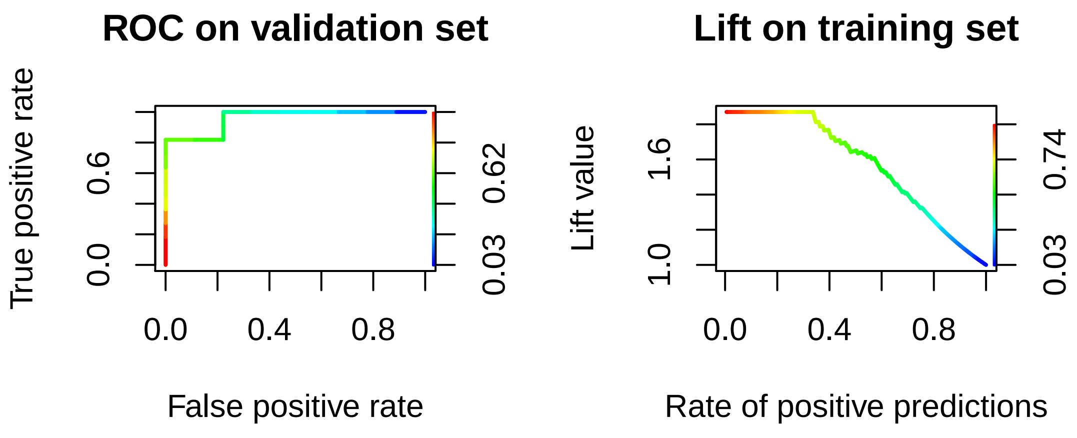
`demo/demo05ROCR.r`

With the help of package ROCR (Sing et al., 2005), several area performance measures can be used for binary classification. The file `demo/demo05ROCR.r` shows an example:

```
    path <- paste(find.package("TDMR"), "demo02sonar",sep="/");
    source(paste(path,"main_sonar.r",sep="/"));    # defines readTrnSonar

    controlDM <- function() {
      #
      # settings for the DM process (former sonar_00.apd file):
      # (see ?tdmOptsDefaultsSet for a complete list of all default settings
      # and many explanatory comments)
      #
      opts = list(path = path,
                   dir.data = "data",              # relative to path
                   filename = "sonar.txt",
                   READ.TrnFn = readTrnSonar,      # defined in main_sonar.r
                   data.title = "Sonar Data",
                   NRUN =  1,
                   rgain.type = "arROC",
                   VERBOSE = 2
      );
      opts <- setParams(opts, defaultOpts(), keepNotMatching = TRUE);
    }

    opts <- controlDM();
    result <- main_sonar(opts);
```



(a) ROC curve (b) Lift chart

Figure 5: (a) ROC curve on validation set with `tdmROCRbase(result)`; (b) Lift chart on training set with `tdmROCRbase(...,typ="lift")`. The bar on the right side shows a color coding of the cutoff parameter.

As explained in Lesson 1 in more detail, the file `start_sonar.r` contains the line

```
    result <- main_sonar(opts);
```

Once the variable `result` contains an object of class `TDMclassifier`, we can infer from it with `tdmROCRbase` the area under the ROC curve and – as a side effect – plot the ROC curve (Fig. 5(a)). The **ROC curve** is a plot 'false positive rate' vs. 'true positive rate', which is obtained by varying the cutoff. Each record is rated by the model and if the model output is above cutoff, then this record is marked 'positive'. The bigger the area between ROC curve and main diagonal, the better the model.

```
  cat("Area under ROC-curve for validation data set: ",
      tdmROCRbase(result),"\n");     # side effect: plot ROC-curve

## Area under ROC-curve for validation data set:  0.959596
```

Equally well we can infer with `typ="lift"` the area under the lift curve and plot a lift chart (Fig. 5(b)). A **lift chart** is constructed in the following way: The records are sorted according to model output. If a high cutoff is choosen only a small portion of the data is marked 'positive' (we have a low rate of positive predictions), but within this portion the rate of true positives is much higher than the overall 'true' rate. The ratio 'true rate in portion'/'overall true rate' is the lift. If we move to lower cutoff values, the 'positive' portion becomes bigger, it is eventually the whole dataset, but at the same time the lift reduces to 1.0. The bigger the area between the lift curve and the horizontal line at 1.0, the better the model.

```
  cat("Area under lift curve for  training data set: ",
      # side effect: plot lift chart:
      tdmROCRbase(result,dataset="training",typ="lift"),"\n");

## Area under lift curve for  training data set:  0.5732612
```

The curves in Fig. 5(a) and 5(b) are colorized according to the cutoff, whose range is shown in the colorbar to the right. That is, if the color is blue, the cutoff is 0.02 in the left plot. This is a very low value, leading to the acceptance of every record. The true positive rate will be 1.0, but of course the false positive rate will be 1.0 as well.

Once the variable `result` contains an object of class `TDMclassifier`, it is also possible to inspect such an object interactively with the following command:

```
  tdmROCR(result);
```

A `twiddler` interface for object `result` shows up (Fig. 6) and allows to select between

- different performance measure plots (ROC-, lift- or precision-recall-chart)
- different data sets (training set or validation set)
- different runs stored in object `result`.

NOTE: The twiddler interface of `tdmROCR(result)` does sometimes not launch successfully when issued from RStudio. If started a second or third time, it will normally launch, but even then the interaction between RStudio's graphics device and `twiddler` may have the problem, that the next lift chart only shows after a second hit on the `Eval` button. If you observe such problems, then start `tdmROCR(result)` from the normal R console (`RGui` under Windows), this works always correctly.

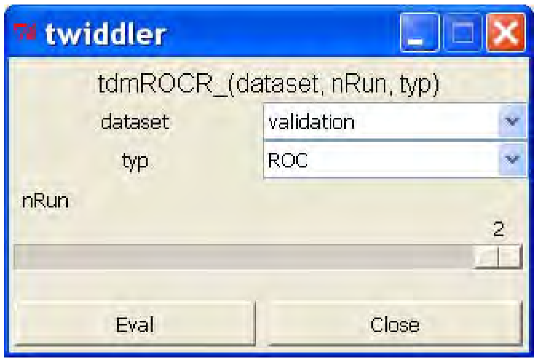


Figure 6: Twiddler interface for `tdmROCR(result)`. The user may select the dataset (`training` or `validation`), the type of plot (ROC, lift, or precision-recall) and the number of the run (only if `Opts(result)$NRUN>1`).

3.6 Lesson 6: Tuner CMA-ES (rCMA)

`demo/demo06cma_j.r`

This demo conducts for tuner `cma_j` (Java version of CMA-ES (Hansen, 2006) interfaced to R via package rCMA) a complete tuned data mining process (TDMR, level 3). Other settings are the same as in `demo03sonar.r`, except that we use `sonar_03.conf` as configuration file. `rCMA` uses `rJava` for the R-to-Java-interface.

3.6.1 Fixing problems with the `rJava` installation

On some operating systems, especially Windows 7, it may happen that the command `require(rJava)` in `demo06cma_j.r` issues an error of the form

```
Error : .onLoad failed in loadNamespace() for rJava, details: ...
```

This means that `rJava` was not installed properly on your computer. Try then the following:

1. Define the environment variable `JAVA_HOME`: Explorer - RightMouse on "Computer" - Properties - Environment Variables, and add there

   ```
   JAVA_HOME = C:\Program Files\Java\jdk1.7.0_11\jre7
   ```

   and **restart R**. (The path is the correct one on my computer, on others it might be slightly different.)

2. Package `rJava` needs to find the Java DLL `jvm.dll`. To enable this, expand the environment variable `Path`: Explorer - RightMouse on "Computer" - Properties - Environment Variables - Path - Edit, and add at the end of the `Path` string

```
C:\Program Files\Java\jdk1.7.0_11\jre\bin\server
```

and **restart R**. (The path is the correct one on my computer, on others it might be slightly different. It must be the directory of the current Java installation containing `jvm.dll`.)

Note that the above remarks are for 64-bit-Java and 64-bit-R. If you use 32-bit-Java, the locations might be slightly different as well.

On some Linux/UNIX systems there might be also problems with the installation of `rJava` because R cannot locate the Java installation. In that case, fix it permanently by issuing the command

```
sudo R CMD javareconf -e
```

at the UNIX prompt (needs admin rights). If you do not have admin rights, you may invoke

```
R CMD javareconf -e
```

in each session where you need `rJava`.

3.7 Lesson 7: Parallel TDMR

`demo/demo07parallel.r`

This demo does the same as `demo03sonar.r`, but it runs 4 experiments on 4 parallel cores (if your environment supports parallel clusters with the R core-package parallel).

3.8 Lesson 8: Tuning with fewer data: The *winequality* dataset

```
examples/ex-winequality/start-wine.r
examples/ex-winequality/final-wine.r
```

This demo lesson operates on a larger data set *winequality* (Cortez et al., 2009), and it answers the following question: Is it possible to do quick tuning runs with a small portion of the data (of course at the price of a somewhat reduced tuning quality) and later, with the best tuning parameters, do a full, high-quality training and testing on all data?

The answer is 'Yes', and we show in this lesson how it can be accomplished in two parts:

1. `start-wine.r`: Performs quick tuning runs and stores the best parameters.
2. `final-wine.r`: Re-loads best parameters from `start-wine.r` and does high-quality unbiased runs (training and test) on all data.

3.8.1 Tuning

To reduce the tuning time, you may specify the parameter `opts$READ.NROW` to a value smaller than the size of the dataset.[10] Then only this number of records is read and used for training and validation during tuning.

Tuning is started in `start_wine.r` with the function `tdmTuneIt`:

```
#*# ---------ex-winequality/start_wine.r, Part 1 ---------
path <- paste(find.package("TDMR"), "examples/ex-winequality",sep="/");

tdm=list(mainFile="main_wine.r"
         ,filenameEnvT="wine_01.RData"   # file to save environment envT
         ,path=path
         ,umode="SP_T"
         , U.saveModel=T
         ,optsVerbosity=1
         ,nrun=2
);

source(paste(path,tdm$mainFile,sep="/")) ;
```

As usual, the file `main_wine.r` containts the function `main_wine` with the template for the data mining process and function `readTrnFn` to read the data. This tuning experiment is repeated `nExperim=1` time.

For brevity, we do not show the control functions `controlDM` and `controlSC`, they are similar to Lesson 2 and 3. See `start_wine.r` for the full listing. The important difference are the parameters `opts$READ.NROW = 600`, which limits the data records during tuning to 600, and `opts$RF.samp = 500`, which limits the Random Forest sample size to 500.

Now we start the whole tuning process with:

```
#*# ---------ex-winequality/start_wine.r, Part 2 ---------
ctrlSC <-   controlSC();
ctrlSC$opts <- controlDM() ;

# construct envT from settings given in tdm & sCList
envT <-  tdmEnvTMakeNew(tdm,sCList=list(ctrlSC));
dataObj <- tdmReadTaskData(envT,envT$tdm);# read the task data from <path>/<data>
envT <- tdmTuneIt(envT,dataObj=dataObj);  # start the tuning loop
```

The corresponding 1 result line is written to data frame `envT$theFinals`:

[10] Additionally, you may reduce `opts$RF.samp`, the `sampsize` parameter in case of Random Forest, to a small value. These settings are all done in function `controlDM()` in file `start-wine.r`.

```
print(envT$theFinals);

##      CONF TUNER NEXP     XPERC NEVAL RGain.bst RGain.avg Time.TRN NRUN
## 1 wine_01  spot    1 0.9997911     8  61.43187  60.62356     3.48    2
##   RGain.TRN  sdR.TRN RGain.SP_T sdR.SP_T Time.TST
## 1  61.77829 1.143129         60 2.357023     0.76
```

Tuning runs are rather short, to make this example run quickly. Do not expect good numeric results. We note that we get in the unbiased runs a quality of `RGain.TRN=60.7` on the training set and `RGain.SP_T=64.5` on the test set.

3.8.2 Retrain on bigger data, test on new data

The tuning results are saved in `wine_01.RData`. The following code from `final_wine.r` shows how to retrain with these tuning results and how to test this model on new data.

We load `wine_01.RData`, then set `opts$READ.NROW=-1`, `opts$READ.TstFn=readTstWine` and `tdm$umode="TST"`. This means that we now read **all** data with `tdmReadAndSplit`, we take all 3919 records in white-wine-train.csv as training data and all 979 records in white-wine-test.csv as test data.

Note the bracketing lines with
`opts <- tdmEnvTGetOpts(envT,1)`
and
`tdmEnvTSetOpts(envT,opts)`.
Between these lines we first retrieve all `opts`-settings from `envT` and then modify specific `opts`-values for the retraining. Here we set for example `opts$RF.samp` to a larger value.

Now we enter `tdmEnvTReport` with the new dataset `dataObj`, the new `opts` indicating that we shall grab the best tuning result and perform training and evaluation on the new dataset:

```
#*# ---------ex-winequality/final_wine.r ---------
path <- paste(find.package("TDMR"), "examples/ex-winequality",sep="/");

tdm=list(mainFile="main_wine.r"
         ,path=path
         ,filenameEnvT="wine_01.RData"   # reload envT from <path>/wine_01.RData"
         ,umode="TST"
         ,TST.valiFrac=0
         ,U.saveModel=T
         ,optsVerbosity=1
         ,nrun=2
);
source(paste(path,tdm$mainFile,sep="/"));

#
```

```
#
# re-use prior tuning result from envT: do only tdmEnvTReport and unbiased eval on
# best tuning result. But do so by training a model on all training data
# (80% of 4898 =3919 records: white-wine-train.csv) and testing it on all test data
# (20% of 4898 = 979 records: white-wine-tst.csv).
#
tdm <- tdmDefaultsFill(tdm);
print(path);
print(tdm$filenameEnvT);
envT<- tdmEnvTLoad(tdm$filenameEnvT,path);    # load envT

envT<- tdmEnvTUpdate(envT,tdm);
                    # update envT£tdm with new elements given in tdm

opts <- tdmEnvTGetOpts(envT);
opts$READ.NROW=-1;  # read *all* records in opts£filename, opts£filetest
opts$RF.samp=5000;
opts$READ.TstFn = readTstWine
opts$VERBOSE=1;
                    # read 'new' data (from opts£filename and opts£filetest):
dataObj <- tdmReadAndSplit(opts,envT$tdm);
envT <- tdmEnvTSetOpts(envT,opts);
envT$tdm$nrun=1;    # =0: no unbiasedRun,
                    # >0: perform unbiasedRun with opts£NRUN=envT£tdm£nrun
envT <-  tdmEnvTReport(envT,1);
if (!is.null(envT$theFinals)) print(envT$theFinals);
```

The results of the new unbiased training + evaluation runs are again recorded in `envT$theFinals`:

```
print(envT$theFinals);

##      CONF TUNER NEXP     XPERC NEVAL RGain.bst RGain.avg Time.TRN NRUN
## 1 wine_01  spot    1 0.9893518     8  62.26852  61.89236     3.26    1
##   RGain.TRN sdR.TRN RGain.TST sdR.TST Time.TST
## 1  69.12478      NA  67.62002      NA     4.02
```

Note that we get a higher gain (higher accuracy) on `RGain.TRN` and `RGain.SP_T` than we had after tuning in Sec. 3.8.1. This is due to the increased number of data used during the unbiased training and evaluation runs conducted by `final_wine.r`.

Note that this lesson writes `envT` to `wine_01.RData` in the directory given by `path`. Depending on the configuration on your machine, this directory might not be writeable for you. In that case, copy directory `examples/ex-winequality` to a writable location and change `path` accordingly in `start_wine.r` and `final_wine.r`.

The warning that sometimes appears „The response has five or fewer unique values. Are you sure you want to do regression?“ is not serious, it only appears because a sensitivity plot (see Fig. 4) is made as a side effect of `tdmEnvTReport`. This plot is based on relatively few points and then Random Forest is used for making the regression over the whole normalized ROI range. The text just warns that the regression is based on not more than five points.

A Appendix A: Frequently Asked Questions (FAQ)

A.1 I have already obtained a best tuning solution for some data set. How can I rerun (re-train) and test it on the same / other data?

A fully worked-out example with larger train data and completely new test data is in Lesson 8 (Sec. 3.8) with the files `start_wine.r` and `final_wine.r`.

As another example, we assume that `demo03sonar.r` has been run, so that `demo03.RData` is available. You may look at the demo `demo03newdata` already presented in Lesson 3.3:

```
##  --------- demo/demo03newdata.r ---------
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");
envT <- tdmEnvTLoad("demoSonar.RData",path);
source(paste(path,"main_sonar.r",sep="/"));
opts <- tdmEnvTGetOpts(envT);
opts$READ.NROW=-1;
opts$path <- path;
envT <- tdmEnvTSetOpts(envT,opts);
dataObj <- tdmReadAndSplit(opts,envT$tdm); # read data, needs readTrnSonar
envT$tdm$nrun=1;     # =0: no unbiasedRun,
                     # >0: perform unbiasedRun with opts£NRUN=envT£tdm£nrun
envT <-  tdmEnvTReport(envT,1);
if (!is.null(envT$theFinals)) print(envT$theFinals);
```

This will reload the tuning results from `demoSonar.RData`. Then all data will be read with `tdmReadAndSplit` into `dataObj` (Since `envT$tdm$umode="CV"`, we will have all 208 data records in the train-validation set. The split into 2 cross-validation folds with 104 records each is done later in `tdmClassifyLoop`, for each seed differently.)

`tdmEnvTReport` will use the best tuning parameters previously found in tuning run 1, train it on the CV-train data and test it on the CV-validate data. The results are reported, as usually, in `envT$theFinals`.

A.2 How can I make with a trained model new predictions?

Run your Lesson-3 script or Lesson-4 script to produce an environment `envT`, which is an object of class `TDMenvir`. There is an element `lastModel` defined in `envT` which contains the model trained on the best tuning solution during the last unbiased run.[11] TDMR defines a function `predict.TDMenvir` , which makes it easy to do new predictions:

```
  newdata=read.csv2(file="cpu.csv", sep="", dec=".")[1:15,];
  z=predict(envT,newdata);
  print(z);
```

[11]The last model `lastModel` is available, if `tdm$U.saveModel=TRUE`, which is the default.

Remarks:

- If the new data contain factor variables (e.g. `vendor` in case of CPU data), it is necessary that `levels(newdata$vendor)` is the same as during training. Therefore we read in the above code snippet first all CPU-data to get the levels right. Only then we shorten them with `[1:15,]` to the first 15 records.
- `lastModel` will be saved to `.RData` file only if `tdm$U.saveModel=TRUE`. This is however the default.
- See also the examples in `demo/demo04cpu.r` and in `predict.TDMenvir`.

A.3 My `.RData` files for saving `envT` are pretty big. Is there a way to make them smaller?

It is the default, that the last DM model is saved in `envT$result$lastRes$lastModel`. Such a DM model can be pretty big. If you do not want this, set `tdm$U.saveModel=FALSE`.

Note however, that then it is not possible to do directly new predictions (see FAQ A.2) on a reloaded `.RData` file. After reloading `envT`, prior to making predictions, you would have first to re-train a model with appropriate training data (see FAQ A.1).

A.4 Why do I get sometimes "Warning in randomForest.default(x, y): The response has five or fewer unique values. Are you sure you want to do regression?"

This comes from the the sensitivity plot made as a side effect of `tdmEnvTReport`. If we have the setting

```
envT <-  tdmEnvTReport(envT,1);
```

then the metamodel used for generating the sensitivity curve(s) is a Random Forest. If this model gets only a few data points for training, it issues this warning, because the data could come also from a classification task.

You can safely ignore this warning, we want to do regression at this point. (This is also true if *your* data mining task is a classification.)

A.5 Why are there two similar functions `tdmTuneIt` and `tdmBigLoop`? Which function should I use when?

If you only have a single configuration in parameter `sCList` of `tdmEnvTMakeNew`, then `tdmTuneIt` is the recommended choice. It has a clearer syntax, `dataObj` is a mandatory calling parameter and this makes the data flow more easy to understand.[12]

[12]If you pass to `tdmTuneIt` a list of configurations, only the first one will be taken.

If you want to process multiple configurations in one TDMR experiment, then `tdmBigLoop` is the recommended choice. If each configuration works with the same `dataObj`, it is recommended to pass `dataObj` as argument to `tdmBigLoop`. The argument `dataObj` is however optional in a call to `tdmBigLoop`. This is for the cases where each configuration needs different data: If `dataObj` is not an argument to `tdmBigLoop` then `dataObj` is constructed anew on each loop-pass inside `tdmBigLoop`.

A.6 How can I add a new tuning parameter to TDMR?

- As a user: Add a new line to `userMapDesign.csv` in directory `tdm$path`. If such a file does not exist yet, the user has to create it with a first line

  ```
      roiValue;   optsValue;      isInt
  ```

  Suppose you want to tune the variable `opts$SRF.samp`: add to file `userMapDesign.csv` a line

  ```
      SRF.SAMP;   opts$SRF.samp;  0
  ```

  This specifies that whenever `SRF.SAMP` appears in a ROI specification `controlSC`, the tuner will tune this variable. TDMR maps `SRF.SAMP` to `opts$SRF.SAMP`. The last `0` means that `SRF.SAMP` is not an integer but a continuous variable.

- As a developer: Add similarly a new line to `tdmMapDesign.csv`. This means that the mapping is available for all tasks, not only for those in the current `tdm$path`.

- Optional, as a developer: For a new variable `opts$Z`, add to `tdmOptsDefaultsSet()` a line specifying a default value for `opts$Z`. Then all existing and further tasks will have this default for `opts$Z`.

A.7 How can I add a new tuning algorithm to TDMR?

See Sec. 10.1.2 „How to integrate new tuners“ in TDMR-docu.pdf (Konen and Koch, 2012a).

A.8 How can I add a new machine learning algorithm to TDMR?

See Sec. 10.2 „How to integrate new machine learning algorithms“ in TDMR-docu.pdf (Konen and Koch, 2012a).

A.9 How can it happen that some variables have an importance that is exactly zero?

Well, the importance for variables with low importance can be zero or even slightly negative (as a consequence of some statistical fluctuations). All those zero or negative importance values will be clipped to zero, therefore a variable with apparently exactly zero importance can happen more frequently than expected.

A.10 What is the difference between `opts$path` and `tdm$path`?

- `opts$path` is the directory where `main_*.r` and the DM task data in `opts$dir.data` are searched.
- `tdm$path` is the directory where `envT` is saved/loaded (filename `envT$tdm$filenameEnvT`). This can be the same as `opts$path`, or it can be different.

If `tdm$path` is not set prior to calling `tdmDefaultsFill`, it will be set to the current working directory.

B Appendix B: Overview TDMR Demos

`demo/00Index`

`demo00-0classif`	Simple, self-contained classification with `tdmClassifyLoop` on task `iris`
`demo00-1regress`	Simple, self-contained regression with `tdmRegressLoop` on task `cpu`
`demo01-1sonar`	TDMR, level 1: Simple TDMR classification on task `sonar` (one run / no tuning)
`demo01-2cpu`	TDMR, level 1: Simple TDMR regression on task `cpu` (one run / no tuning)
`demo02sonar`	TDMR, level 2: SPOT tuning for task `sonar`
`demo03sonar`	TDMR, level 3: Tuning for TDMR classification task `sonar` (multiple runs / short tuning)
`demo03sonar_B`	TDMR, level 3: Same as `demo03sonar`, but with parameters for a longer tuning run
`demo03newdata`	TDMR, level 3: Apply the result of `demo03sonar` to new data (redo training on new data with best-tuned parameters)
`demo03newpredict`	TDMR, level 3: Apply the result of `demo03sonar` to new data (use last trained model in `envT`)
`demo04cpu`	TDMR, level 3: Tuning for TDMR regression task `cpu` (multiple runs / short tuning)
`demo05ROCR`	Visualization of classification results using package `ROCR`
`demo06cma_j`	Tuning demo for tuner `cma_j` (CMAES, Java version through `rCMA`, runs on all platforms)
`demo07parallel`	Demo for parallel execution (8 experiments of type `demo03sonar` on 4 cores)

In addition, there is the directory `examples/ex-winequality` with the demos `start_wine.r` and `final_wine.r` (see Lesson 8, Sec. 3.8). And the example `ex_demo04par.r` which does the same as `demo04cpu`, but on two parallel cluster nodes.

References

Bartz-Beielstein, T. (2010). SPOT: An R package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization. arXiv.org e-Print archive, `http://arxiv.org/abs/1006.4645`.

Cortez, P., Cerdeira, A., Almeida, F., Matos, T., and Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553.

Hansen, N. (2006). The CMA evolution strategy: a comparing review. In Lozano, J., Larranaga, P., Inza, I., and Bengoetxea, E., editors, *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, pages 75–102. Springer.

Koch, P., Bischl, B., Flasch, O., Bartz-Beielstein, T., Weihs, C., and Konen, W. (2012). Tuning and evolution of support vector kernels. *Evolutionary Intelligence*, 5:153–170.

Koch, P. and Konen, W. (2012). Efficient sampling and handling of variance in tuning data mining models. In Coello Coello, C., Cutello, V., et al., editors, *PPSN'2012: 12th International Conference on Parallel Problem Solving From Nature, Taormina*, pages 195–205, Heidelberg. Springer.

Koch, P. and Konen, W. (2013). Subsampling strategies in SVM ensembles. In Hoffmann, F. and Hüllermeier, E., editors, *Proceedings 23. Workshop Computational Intelligence*, pages 119–134. Universitätsverlag Karlsruhe.

Koch, P., Wagner, T., Emmerich, M. T. M., Bäck, T., and Konen, W. (2015). Efficient multi-criteria optimization on noisy machine learning problems. *Applied Soft Computing*, 29:357–370.

Konen, W. (2011). Self-configuration from a machine-learning perspective. CIOP Technical Report 05/11; arXiv: 1105.1951, Research Center CIOP (Computational Intelligence, Optimization and Data Mining), Cologne University of Applied Science, Faculty of Computer Science and Engineering Science. e-print published at http://arxiv.org/abs/1105.1951 and Dagstuhl Preprint Archive, Workshop 11181 "Organic Computing – Design of Self-Organizing Systems".

Konen, W. and Koch, P. (2012a). The TDMR Package: Tuned Data Mining in R. Technical Report 02/2012, Research Center CIOP (Computational Intelligence, Optimization and Data Mining), Cologne University of Applied Science, Faculty of Computer Science and Engineering Science. Last update: June, 2017.

Konen, W. and Koch, P. (2012b). The TDMR Tutorial: Examples for Tuned Data Mining in R. Technical Report 03/2012, Research Center CIOP (Computational Intelligence, Optimization and Data Mining), Cologne University of Applied Science, Faculty of Computer Science and Engineering Science. Last update: May, 2016.

Konen, W., Koch, P., Flasch, O., and Bartz-Beielstein, T. (2010). Parameter-Tuned Data Mining: A General Framework . In *Proc. 20th Workshop Computational Intelligence*, pages 136–150. KIT Scientific Publishing, `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000020316`.

Konen, W., Koch, P., Flasch, O., Bartz-Beielstein, T., Friese, M., and Naujoks, B. (2011). Tuned data mining: A benchmark study on different tuners. In Krasnogor, N., editor, *GECCO '11: Proceedings of the 13th Annual Conference on Genetic andEvolutionary Computation*, volume 11, pages 1995–2002.

Sing, T., Sander, O., Beerenwinkel, N., and Lengauer, T. (2005). ROCR: visualizing classifier performance in R. *Bioinformatics*, 21(20):3940–3941.

Stork, J., Ramos, R., Koch, P., and Konen, W. (2013). SVM ensembles are better when different kernel types are combined. In Lausen, B., editor, *European Conference on Data Analysis (ECDA13)*. GfKl.