The TDMR Tutorial:
Examples for Tuned Data Mining in R

Wolfgang Konen, Patrick Koch,
Cologne University of Applied Sciences

Initial version: June, 2012
Last update: May, 2016

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. [2014].

[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
## Warning:  package 'SPOT' was built under R version 3.2.5
## 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)
```

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=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.Width"  "Petal.Length" "Sepal.Length" "Sepal.Width"
## Dropped columns (0 [=  0.0% of total importance]):
```

```
## Proc time:  0.02
## Run  1 / 2 :
## default.txt : Train RF with sampsize = 135 ...
## Proc time:  0.07
## 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 on   training set    95.55556 %
##   Relative gain on validation set    93.33333 %
##
## default.txt : Stratified random training-validation-index with opts$TST.valiFrac =  10 %
##
## default.txt : Importance check ...
## Clipping sampsize to  135
## default.txt : Train RF (importance, sampsize= 135 ) ...
## default.txt : Saving SRF (sorted RF) importance info on opts ...
## Variables sorted by importance (4 ):
## [1] "Petal.Length" "Petal.Width"  "Sepal.Length" "Sepal.Width"
## Dropped columns (1 [=  0.5% of total importance]):
## [1] "Sepal.Width"
## Proc time:  0.02
## Run  2 / 2 :
## default.txt : Train RF with sampsize = 135 ...
## Proc time:  0.06
## default.txt : Apply RF ...
## Proc time:  0.02
## default.txt : Calc confusion matrix + gain ...
##
## Training cases ( 135 ):
##             predicted
## actual       setosa versicolor virginica
##   setosa         45          0         0
##   versicolor      0         42         3
##   virginica       0          3        42
## total gain:   129.0 (is  95.556% of max. gain =   135.0)
##
## Validation cases (15):
##             predicted
## actual       setosa versicolor virginica
##   setosa          5          0         0
##   versicolor      0          5         0
##   virginica       0          0         5
##             setosa versicolor virginica Total
## gain.vector      5          5         5    15
## total gain :    15.0 (is 100.000% of max. gain =    15.0)
##
##   Relative gain on   training set    95.55556 %
##   Relative gain on validation set    100 %
##
##
## Average over all  2  runs:
## cerr$train: (4.44444 +- 0.00000)%
## cerr$vali:  (3.33333 +- 4.71405)%
## gain$train: ( 129.00 +- 0.00)
## gain$vali:  (  14.50 +- 0.71)
## rgain.train:  95.556%
## 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,] 0.04444444      129  95.55556  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 × confusion matrix“ 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 where $C^{(ideal)}$ is the 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,] 0.04444444      129  95.55556  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 (`sonar_00.apd`).

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 several files. This may look complicated at first sight, but it is useful for two reasons:

Separate function file: As a preparation for the tuning process in subsequent lessons: It is very useful if we can package the whole data mining 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.

Separate parameter file: For conducting slightly different variants, runs or experiments, it is useful to package the parameter setting part in one (or several) files as well.

In this lesson we will look at four relevant files:

1. `sonar_00.apd` (the parameter settings)
2. `main_sonar.r` (the DM function `main_sonar`)
3. `start_sonar.r` (starter file)
4. `demo01-1sonar.r` (demo starter - only needed for TDMR-package demo)

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 `myDir/data` relative to the other files.

If you want to build a DM classification model with TDMR, you need to provide two files, `sonar_00.apd` and `main_sonar.r`.[5] The first file, `sonar_00.apd` (`.apd` = algorithmic problem design), is already in preparation for later tuning (see Lesson02 and Lesson03), it defines in list `opts` all relevant settings 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

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

[5]Templates for `sonar_00.apd` and `main_sonar.r` are available from `<inst>/demo02sonar` where `<inst>` refers to the installation directory of package TDMR as returned by `find.package("TDMR")`.

list `result` the evaluation of the DM model. The list `result` is either inspected by the user or by the tuning process.

```
## sonar_00.apd
##
## set the basic elements of list opts for task sonar. See ?tdmOptsDefaultsSet
## for a complete list of all default settings and many explanatory comments
      opts = tdmOptsDefaultsSet();
      opts$dir.data <- "data/";
      opts$filename <- "sonar.txt"
      opts$READ.TrnFn <- readTrnSonar
      opts$NRUN <- 1
      opts$data.title <- "Sonar Data"
```

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`. You need to specify only those things which differ from `tdmOptsDefaultsSet()`: 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 file `main_sonar.r` contains two functions `main_sonar` and `readTrnSonar`:

```
main_sonar <- function(opts=NULL, dset=NULL, tset=NULL) {
  if (is.null(opts)) source("sonar_00.apd", local=TRUE);
  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 <- tdmReadData2(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.variable))

  ########  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$dir.data, opts$filename, sep=""),
            dec=".", sep=",", nrow=opts$READ.NROW,header=FALSE);
}
```

To start the whole procedure, there is a small starter file `start_sonar.r`:

```
source("main_sonar.r");
result <- main_sonar(opts);
```

This file is invoked by `demo01-1sonar.r`:

```
## --------- demo/demo01-1sonar.r ---------
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");
source(paste(path,"main_sonar.r",sep="/"));      # needed to define readTrnSonar
source(paste(path,"sonar_00.apd",sep="/"),local=TRUE);   # set opts, needs readTrnSonar
source(paste(path,"start_sonar.r",sep="/"),chdir=TRUE,print.eval=TRUE);
```

The reason why we have the file chain

$$\texttt{demo01-1sonar.r} \xrightarrow{\texttt{source}} \texttt{start\_sonar.r} \xrightarrow{\texttt{call}} \texttt{main\_sonar.r}$$

is that `main_sonar` needs to perform certain file I/O in the directory `path`. Sourcing `start_sonar.r` with `source(...,chdir=TRUE)` tells R that it changes to the directory `path` prior to sourcing (and automatically returns to the actual working directory at the end of sourcing[6]).

The distinction between `start_sonar.r` and `demo01-1sonar.r` is only needed for the TDMR-package demo. If you write your own application, you can have `main_sonar.r` together with the `.apd` file in the same directory `myDir`. Then you only need one starter script in `myDir` which simply reads like this:

```
source("main_sonar.r");
source("sonar_00.apd");
result <- main_sonar(opts);
```

[6]Even in the case of an error inside `start_sonar.r` R will correctly return to the actual working directory.

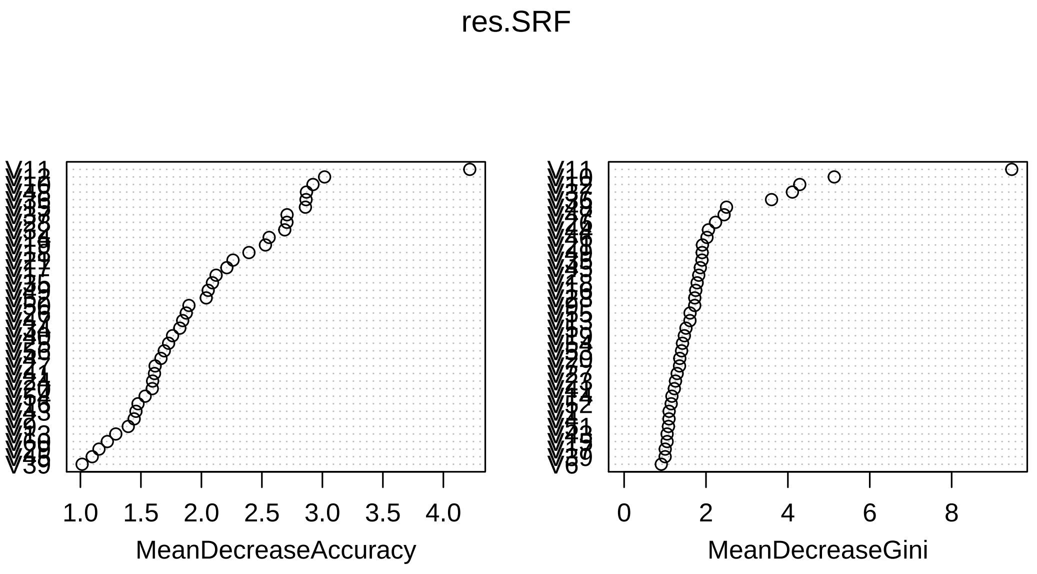


Figure 1: Some plots from demo01-1sonar.r

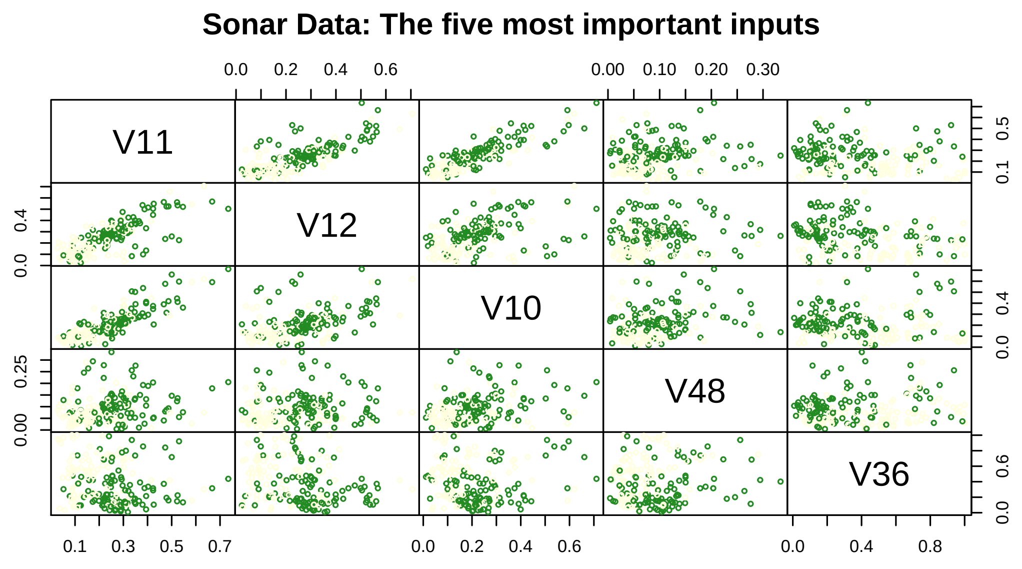


Figure 2: Some plots from demo01-1sonar.r

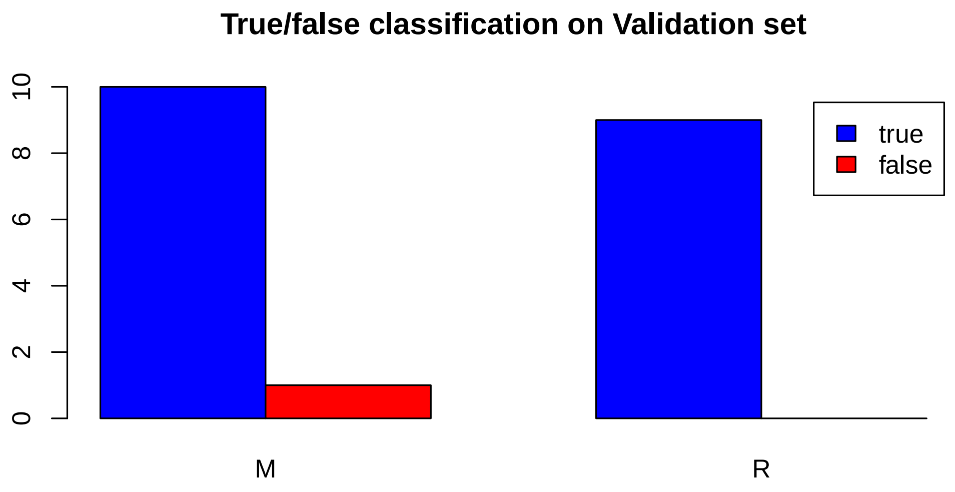


Figure 3: Some plots from demo01-1sonar.r

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 01 (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* (`.roi`-file).

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` from Lesson 01 the three small files `sonar_01.conf`, `sonar_01.apd` and `sonar_01.roi`. The content of these files may look for example like this:

sonar_01.conf

```
alg.func = "tdmStartSpot"
alg.resultColumn = "Y"
alg.seed = 1235

io.apdFileName = "sonar_01.apd"
io.roiFileName = "sonar_01.roi"
spot.seed = 120 # 125
io.verbosity = 3;
auto.loop.steps = 50;    # number of spot metamodels to be generated
auto.loop.nevals = 50;  # concurrently, max number of algo evaluations

init.design.func = "spotCreateDesignLhd";
init.design.size = 10;    # number of initial design points
init.design.repeats = 1;  # number of initial repeats

seq.merge.func <- mean;
seq.design.size = 100;
seq.design.retries = 15;
seq.design.maxRepeats = 2;
seq.design.oldBest.size <- 1;
seq.design.new.size <- 3;

seq.predictionModel.func = "spotPredictRandomForest";

report.func = "spotReportSens"
```

sonar_01.apd

```
opts = tdmOptsDefaultsSet();
opts$dir.data <- "data/";
opts$filename = "sonar.txt"
opts$READ.TrnFn <- readTrnSonar    # defined in main_sonar.r
opts$data.title <- "Sonar Data"

opts$RF.mtry = 4
opts$NRUN =  1          # how many runs with different train & vali samples
                        # - or - how many CV-runs, if TST.kind="cv"
opts$GD.DEVICE="non"    # e.g. ["pdf"|"win"|"non"]
opts$GD.RESTART=F;
opts$VERBOSE = opts$SRF.verbose = 0;
opts$logFile=FALSE      # no logfile (needed for Sweave/.Rnw only)
```

sonar_01.roi

```
name low high type
CUTOFF1 0.1 0.80 FLOAT
CLASSWT2 5 15 FLOAT
XPERC 0.90 1.00 FLOAT
%@
```

The three parameters `CUTOFF1`, `CLASSWT2` and `XPERC` are tuned within the borders specified by `sonar_01.roi`. 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.

To start the whole procedure, there is a small starter file `start_bigLoop.r`:

```
## --------- start_bigLoop.r ---------
envT <- tdmEnvTMakeNew(tdm);
opts <- tdmEnvTGetOpts(envT,1);
dataObj <- tdmSplitTestData(opts,tdm);
envT <- tdmBigLoop(envT,spotStep,dataObj);
```

This file is invoked by `demo02sonar.r`:

```
## --------- demo/demo02sonar.r ---------
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");
tdm=list(mainFile="main_sonar.r",runList="sonar_01.conf");
spotStep = "auto";
source(paste(path,tdm$mainFile,sep="/"));
source(paste(path,"start_bigLoop.r",sep="/"),chdir=TRUE);
```

The reason why we have the file chain

$$\texttt{demo02sonar.r} \xrightarrow{\texttt{source}} \texttt{start\_bigLoop.r} \xrightarrow{\texttt{call}} \texttt{tdmEnvTMakeNew}$$

is the same as in Lesson 1: `tdmEnvTMakeNew` may need to perform certain file I/O in the directory `path`. Sourcing `start_bigLoop.r` with `source(...,chdir=TRUE)` tells R that it changes to the directory `path` prior to sourcing (and automatically returns to the actual working directory at the end of sourcing[7]).

Again, as in Lesson 1, the distinction between `start_bigLoop.r` and `demo02sonar.r` is only needed for the TDMR-package demo. If you write your own application, you can have `main_sonar.r` together with the `.apd`, `.roi` and `.conf` files in the same directory `myDir`. The data file `sonar.txt` should be in the subdirectory `myDir/data`. Then you only need one starter script in `myDir` which simply reads like this:

```
tdm=list(mainFile="main_sonar.r",runList="sonar_01.conf");
source(paste(path,tdm$mainFile,sep="/"));
envT <- tdmEnvTMakeNew(tdm);
opts <- tdmEnvTGetOpts(envT,1);
dataObj <- tdmSplitTestData(opts,tdm);
envT <- tdmBigLoop(envT,"auto",dataObj);
```

In any case, what happens in this demo is the following:

- The first command sets the minimal `tdm`.
- The second command sources the main file.
- With the command `tdmEnvTMakeNew(tdm)` we construct an environment with all necessary TDMR data and functions for this lesson. Inside `tdmEnvTMakeNew` the minimal `tdm` is filled with all defaults (see `tdmDefaultsFill`).

[7] Even in the case of an error inside `start_bigLoop.r` R will correctly return to the actual working directory.

- The command `tdmBigLoop` is the main workhorse. It reads in the DM data, splits them in a train/vali and a test part. Then it calls the desired tuner(s) (in this case only SPOT, since the default for `tdm$tuneMethod` is `"spot"`, but `tdm$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 `.roi` file. The results of the whole tuning process are returned in `envT`, more details on `envT` are in the manual / help section for `tdmBigLoop`.

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

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

3.3.1 Multiple `.conf` Files

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

```
##* --------- demo/demo03sonar.r ---------
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");
tdm <- list( mainFile="main_sonar.r"
           , runList = c("sonar_04.conf","sonar_06.conf")
           , umode="CV"               # { "CV" | "RSUB" | "TST" | "SP_T" }
           , tuneMethod = c("lhd")
           , filenameEnvT="demo03.RData"   # file to save envT (in dir "path")
           , nrun=3, nfold=2          # repeats and CV-folds for the unbiased runs
           , nExperim=1
           , parallelCPUs=1
           , parallelFuncs=c("readTrnSonar")
           , optsVerbosity = 3        # the verbosity for the unbiased runs
           );
spotStep = "auto";
source(paste(path,tdm$mainFile,sep="/"));
source(paste(path,"start_bigLoop.r",sep="/"),chdir=TRUE,local=TRUE);
# change dir to "path" while sourcing
```

This is very much the same as in Lesson 2, we reuse the small starter file `start_bigLoop.r` from there. The only difference is that now **multiple** tuning runs can be performed with respect to the following three dimensions:

- configuration files (elements of `tdm$runList`)
- tuners (elements of `tdm$tuneMethod`)
- repeated experiments with different random seeds (number `tdm$nExperim`).

The function `tdmBigLoop` realizes a triple `for`-loop over these dimensions. With k =`length(runList)`, 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:

- `sonar_04.conf` is started with tuner `lhd`
- `sonar_06.conf` is started with tuner `lhd`.

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

```
print(envT$theFinals);

##       CONF TUNER NEXP    CUTOFF1  CLASSWT2     XPERC NEVAL RGain.bst
## 1 sonar_04   lhd    1 0.06907716 13.637625 0.8581948    10  91.66667
## 2 sonar_06   lhd    1 0.45476980  5.536813 0.6545237    10  97.22222
##   RGain.avg Time.TRN NRUN RGain.TRN    sdR.TRN RGain.CV    sdR.CV Time.TST
## 1  77.66667     0.96    3  82.47619 5.1428706 82.66667 1.5275252     0.35
## 2  88.55556     0.99    3  98.66667 0.5773503 96.66667 0.5773503     0.36
```

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]. 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`. 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 `.conf` File

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 3.9 (**Tuning with fewer data**) for a fully worked-out example with `tdmTuneIt`.

Alternatively you may look at the demo `demo03newdata` for another example:

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

```
##* --------- demo/demo03newdata.r ---------
path  <- paste(find.package("TDMR"), "demo02sonar",sep="/");
#path <- paste("../inst", "demo02sonar",sep="/");
oldwd <- getwd(); setwd(path);
envT <- tdmEnvTLoad("demo03.RData");
source(envT$tdm$mainFile);
source("sonar_06.apd")     # opts
opts$READ.NROW=-1;
dataObj <- tdmSplitTestData(opts,envT$tdm);
envT <- tdmBigLoop(envT,"rep",dataObj);
setwd(oldwd);
```

Here we use `tdmBigLoop` (and not `tdmTuneIt`), since we have two CONF files in `envT` of `demo03.RData`. `tdmBigLoop` will retrieve the best tuning parameters from each tuning run and perform with it a re-training and re-evaluation on the new data.[9]

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
## 1 sonar_04   lhd    1 0.06907716 13.637625 0.8581948    10  91.66667
## 2 sonar_06   lhd    1 0.45476980  5.536813 0.6545237    10  97.22222
##   RGain.avg Time.TRN NRUN RGain.TRN  sdR.TRN RGain.CV    sdR.CV Time.TST
## 1  77.66667     0.02    3  57.53205 1.468774 57.05128 0.2775722     1.14
## 2  88.55556     0.01    3  80.12821 2.001602 76.12179 4.8634586     0.99
```

3.4 Lesson 4: Regression Big Loop

`demo/demo04cpu.r`

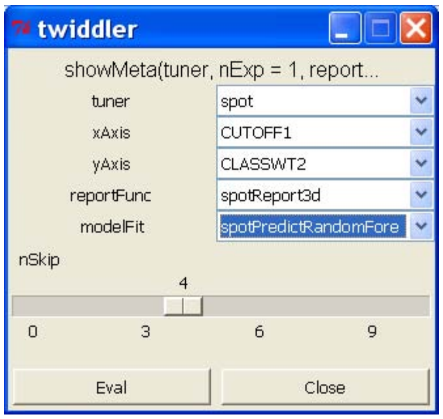
The same as Lesson 3, but applied to a regression task (dataset CPU).

3.5 Lesson 5: Interactive Visualization

`demo/demo05visMeta.r`

Once a Lesson-3 experiment is completed, the return value `envT` from `tdmBigLoop()` contains the result of such an experiment and may be visually inspected. Alternatively, `envT` may be loaded from an appropriate `.RData` file. The call

[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 CONF file would construct its own `dataObj` inside `tdmBigLoop`.



showMeta(tuner, nExp = 1, report...
tuner: spot
xAxis: CUTOFF1
yAxis: CLASSWT2
reportFunc: spotReport3d
modelFit: spotPredictRandomFore
nSkip: 4

Figure 4: The user interface in `tdmPlotResMeta`. The user may select the tuner, the design variables to show on x- and y-axis, the display function (`spotReport3d` or `spotReportContour`) and the metamodel function (`modelFit`). Two optional sliders are `nExper` and `nSkip` (see text).

```
tdmPlotResMeta(envT);
```

allows to visually inspect all RES data frames contained in `envT`.

The user interface is shown and explained in Fig. 4. An additional combo box `confFile` appears only, if `envT$runList` has more than one element. An additional slider `nExper` appears only, if `envT$tdm$nExperim>1`.

The user selects with `tuner`, `confFile` and `nExper` a certain RES data frame from `envT`. This data frame contains a collection of function evaluations for certain design points selected by the tuner. With one of the metamodel construction functions (see package SPOT for further details)

- spotPredictGausspr
- spotPredictRandomForest
- spotPredictMlegp

a metamodel is fitted to the RES data frame and the result is shown as shaded surface in the plot. The RES data points are shown as black points in Fig. 5. Since certain "bad" RES point may dominate the plot as outliers and hinder the user to inspect the region near the optimum, there are two options to suppress "bad" points:

1. If the slider `nSkip` has a value > 0, then the `nSkip` RES data points with the worst y-value are discarded.
2. If the checkbox "Skip incomplete CONFIGs" is activated, then design points belonging

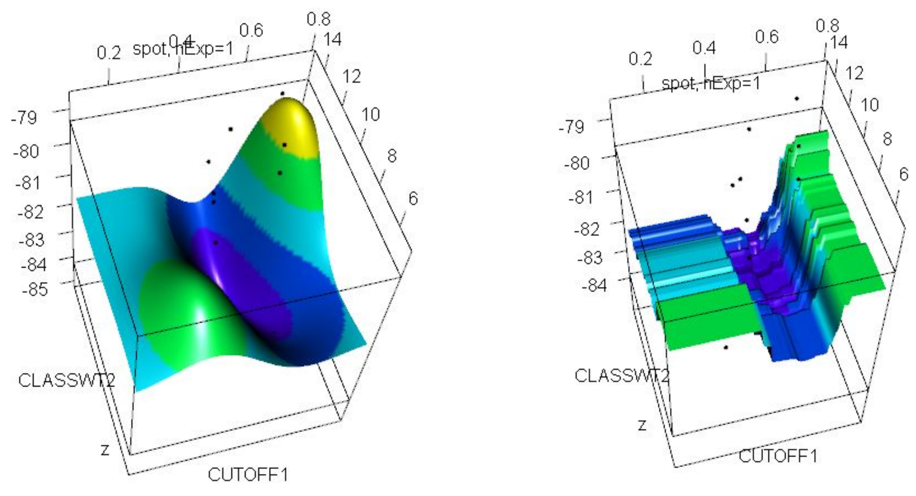


Figure 5: Two example outputs from `tdmPlotResMeta` with `reportFunc=spotReport3d`. Left: `modelFit = spotPredictGausspr`, right: `= spotPredictRandomForest`.

to a configuration which was not evaluated `maxRepeats` times are discarded (relevant for SPOT only).

Note that both options will reduce the number of RES data points. This will also affect the metamodel fit, so use both options with care, if the number of RES data points is small.

The plots created with `spotReport3d` make use of the rgl-package. They can be interactively manipulated with the mouse. They can be selected and saved as PNG images with commands like

```
rgl.set(7);
rgl.snapshot("myFile.png");
```

A complete demo example is invoked with:

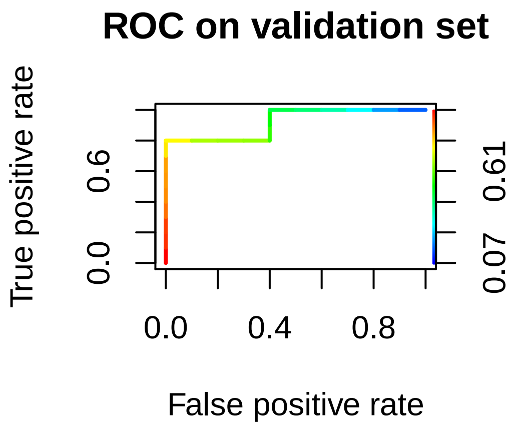
```
demo(demo05visMeta);
```

3.6 Lesson 6: Performance Measure Plots

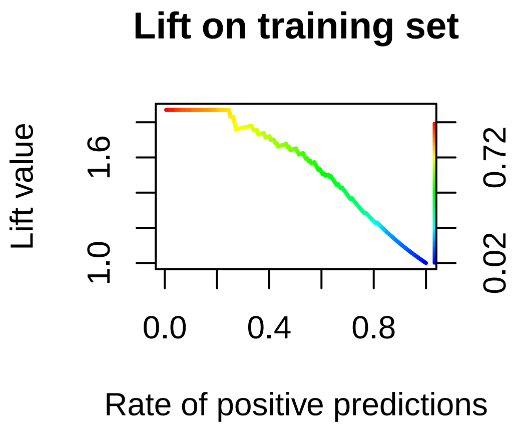
`demo/demo06ROCR.r`

With the help of package ROCR Sing et al. [2005], several area performance measures can be used for binary classification. The file `demo/demo06ROCR.r` shows an example:

```
path <- paste(find.package("TDMR"), "demo02sonar",sep="/");
source(paste(path,"main_sonar.r",sep="/"),chdir=TRUE);
opts = tdmOptsDefaultsSet();
opts$filename = "sonar.txt"
opts$READ.TrnFn = readTrnSonar     # defined in main_sonar.r
opts$data.title <- "Sonar Data";
opts$rgain.type <- "arROC";
source(paste(path,"start_sonar.r",sep="/"),chdir=TRUE);
```



(a) ROC curve



(b) Lift chart

Figure 6: (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. 6(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:  0.92
```

Equally well we can infer with `typ="lift"` the area under the lift curve and plot a lift chart (Fig. 6(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:  0.5552925
```

The curves in Fig. 6(a) and 6(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 0.1 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. 7) 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.

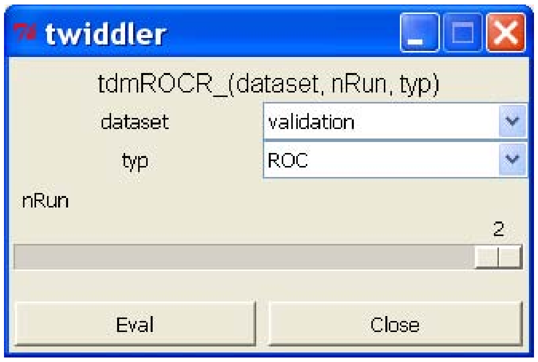


Figure 7: Twiddler interface for `tdmROCR(result)`. The user may select the dataset (`training` or `validation`), the type of plot (ROC, lift, or precision-recall) and the number of the run (only if `Opts(result)$NRUN>1`).

3.7 Lesson 7: Tuner CMA-ES (rCMA)

`demo/demo07cma_j.r`
`demo02sonar/sonar_03.conf`

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.

3.7.1 Fixing problems with the `rJava` installation

On some operating systems, especially Windows 7, it may happen that the command `require(rJava)` in `demo07cma_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`.

3.8 Lesson 8: Parallel TDMR

`demo/demo08parallel.r`
`demo02sonar/sonar_04.conf`

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).

3.9 Lesson 9: Tuning with fewer data

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

3.9.1 Tuning

We add an extra feature to this demo lesson: Suppose you have a large dataset and you want to do quick tuning runs. To reduce the tuning time (of course at the price of a somewhat reduced tuning quality) 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. After tuning has finished, you may want to use the best parameters found by tuning and to perform a high-quality training and evaluation on the full dataset to assess the real strength of the tuning result.

Tuning is started with the function `tdmTuneIt`:

[10]Alternatively, you may reduce `opts$RF.samp`, the `sampsize` parameter in case of Random Forest, to a small value. These settings are all done in the APD-file `wine_01.apd`, as specified in `wine_01.conf`.

```
#*# ---------ex-winequality/start_wine.r ---------
path <- paste(find.package("TDMR"), "examples/ex-winequality",sep="/");
oldwd=getwd();  setwd(path);
tdm=list(mainFile="main_wine.r"
         ,runList="wine_01.conf"  # in "wine_01.apd": opts£READ.NROW=600
         ,umode="SP_T"
         ,U.saveModel=F
         ,optsVerbosity=1
         ,nrun=2
);
source(tdm$mainFile);

#
# perform a complete tuning, but only with the first 600 records
#
envT <- tdmEnvTMakeNew(tdm); # construct envT from the TDMR settings in tdm
opts <- tdmEnvTGetOpts(envT,1);
dataObj <- tdmSplitTestData(opts,tdm);
envT <- tdmTuneIt(envT,"auto",dataObj);  # start the tuning loop

## Warning in randomForest.default(x, y):  The response has five or fewer unique
values.  Are you sure you want to do regression?

setwd(oldwd);
```

As usual, the file `main_wine.r` containts the function `main_wine with the template for the data mining proc`
to read the data. This tuning experiment is repeated `nExperim=1` time. The corresponding 1 result line is written to data frame `envT$theFinals`:

```
print(envT$theFinals);

##      CONF TUNER NEXP     XPERC NEVAL RGain.bst RGain.avg Time.TRN NRUN
## 1 wine_01  spot    1 0.9330384    11   65.0463  64.26768     5.74    2
##   RGain.TRN  sdR.TRN RGain.SP_T  sdR.SP_T Time.TST
## 1  63.88889 1.964186   57.91667 0.5892557     1.01
```

Tuning runs are rather short, to make this example run quickly. Do not expect good numeric results.

3.9.2 Retrain on bigger data set

The tuning results are saved in `wine_01.RData`. The following code from `final_wine.r` shows how to retrain with these tuning results.

We load this file, then set `opts$READ.NROW = -1`. This means that we now read **all** data with `tdmSplitTestData` and split them into 10% test data (the default) and 90% training data (since we specify with `tdm$TST.valifrac=0` that we want no validation 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 `tdmTuneIt` with the new dataset `dataObj`, the new `opts`, and with `spotStep="rep"` 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="/");
oldwd <- getwd();  setwd(path);

tdm=list(mainFile="main_wine.r"
         ,runList="wine_01.conf"
         ,umode="SP_T"
         ,TST.valiFrac=0
         ,U.saveModel=F
         ,optsVerbosity=1
         ,nrun=2
);
source(tdm$mainFile);

#
# re-use prior tuning result (spotStep="rep"); do only spot-report and
# unbiased eval on best tuning result.
# But do so by training a model on 80% of all 4898 records.
#
tdm <- tdmDefaultsFill(tdm)
load(tdm$filenameEnvT);      # envT
opts <- tdmEnvTGetOpts(envT,1);
opts$READ.NROW=-1;           # read all 4898 records of winequality-white.csv
opts$RF.samp=5000;
envT$tdm$optsVerbosity <- 1;
envT$tdm$U.saveModel <- tdm$U.saveModel;
envT$tdm$TST.valiFrac <- tdm$TST.valiFrac;
dataObj <- tdmSplitTestData(opts,envT$tdm);
envT <- tdmEnvTSetOpts(envT,opts);
envT <- tdmTuneIt(envT,"rep",dataObj);

setwd(oldwd);
```

The result of the new unbiased training + evaluation runs is 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 0.9330384    11   65.0463  64.26768     0.03    2
##   RGain.TRN  sdR.TRN RGain.SP_T  sdR.SP_T Time.TST
## 1  67.81067 0.884109    71.1951 0.1444549     9.94
```

Note that we get a higher gain (lower error) on `RGain.TRN` and `RGain.SP_T` than we had after tuning in Sec. 3.9.1. This is due to the increased number of data used during the unbiased training and evaluation runs.

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 and test it on the same / other data?

As an 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="/");
oldwd <- getwd(); setwd(path);
envT <- tdmEnvTLoad("demo03.RData");
source(envT$tdm$mainFile);
source("sonar_06.apd")      # opts
opts$READ.NROW=-1;
dataObj <- tdmSplitTestData(opts,envT$tdm);
envT <- tdmBigLoop(envT,"rep",dataObj);
setwd(oldwd);
```

This will reload the tuning results from `demo03.RData`. Then all data will be read with `tdmSplitTestData` 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.)

`tdmTuneIt` will use the best tuning parameters previously found, 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);
```

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.

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

- `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 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 report step in SPOT. If we have the setting

```
seq.predictionModel.func = "spotPredictRandomForest";
```

then the metamodel used in SPOT is 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.4 Why are there two similar functions `tdmTuneIt` and `tdmBigLoop`? Which function should I use when?

If you only have a single CONF file, 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]

If you want to process multiple CONF files in one TDMR experiment, then `tdmBigLoop` is the recommended choice. If each CONF file 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 CONF file needs different data: If `dataObj` is not an argument to `tdmBigLoop` then `dataObj` is constructed anew on each loop-pass inside `tdmBigLoop`.

A.5 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 it is then not possible to reload the `.RData` file with `envT` and do directly new predictions. You would have to re-train a model first with appropriate training data.

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

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` file in directory `tdm$path`, the tuner will tune this variable. TDMR maps `SRF.SAMP` to `opts$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.

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)
demo05visMeta	Interactive visualization of RES data frames generated by `demo04cpu` and their metamodels
demo06ROCR	Visualization of classification results using package `ROCR`
demo07cma_j	Tuning demo for tuner `cma_j` (CMAES, Java version through `rCMA`, runs on all platforms)
demo08parallel	Demo for parallel execution (8 experiments of type `demo03sonar` on 4 cores)

References

Thomas Bartz-Beielstein. 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`, June 2010.

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

Patrick Koch and Wolfgang Konen. Efficient sampling and handling of variance in tuning data mining models. In Carlos Coello Coello, Vincenzo Cutello, et al., editors, *PPSN'2012: 12th International Conference on Parallel Problem Solving From Nature, Taormina*, pages 195–205, Heidelberg, September 2012. Springer. URL `http://www.gm.fh-koeln.de/ciopwebpub/Koch12a.d/Koch12a.pdf`.

Patrick Koch and Wolfgang Konen. Subsampling strategies in svm ensembles. In Frank Hoffmann and Eyke Hüllermeier, editors, *Proceedings 23. Workshop Computational Intelligence*, pages 119–134. Universitätsverlag Karlsruhe, 2013. URL `http://www.gm.fh-koeln.de/~konen/Publikationen/kochGMA2013.pdf`.

Patrick Koch, Bernd Bischl, Oliver Flasch, Thomas Bartz-Beielstein, Claus Weihs, and Wolfgang Konen. Tuning and evolution of support vector kernels. *Evolutionary Intelligence*, 5:153–170, 2012. URL `http://www.gm.fh-koeln.de/~konen/Publikationen/Koch11a-EvolIntel.pdf`.

Patrick Koch, Tobias Wagner, Michael T. M. Emmerich, Thomas Bäck, and Wolfgang Konen. Efficient multi-criteria optimization on noisy machine learning problems. *Applied Soft Computing*, (accepted for publication):1, 2014.

W. Konen and P. Koch. 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, 2012a. URL `http://www.gm.fh-koeln.de/ciopwebpub/Kone12a.d/Kone12a.pdf`. Last update: March, 2015.

W. Konen and P. Koch. 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, 2012b. URL `http://www.gm.fh-koeln.de/ciopwebpub/Kone12b.d/Kone12b.pdf`. Last update: March, 2015.

W. Konen, P. Koch, O. Flasch, and T. Bartz-Beielstein. 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`, 2010. URL `http://www.gm.fh-koeln.de/~konen/Publikationen/GMACI10_tunedDM.pdf`.

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

Wolfgang Konen. 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, May 2011. URL `http://www.gm.fh-koeln.de/ciopwebpub/Kone11c.d/Kone11c.pdf`. e-print published at http://arxiv.org/abs/1105.1951 and Dagstuhl Preprint Archive, Workshop 11181 "Organic Computing – Design of Self-Organizing Systems".

T. Sing, O. Sander, N. Beerenwinkel, and T. Lengauer. ROCR: visualizing classifier performance in R. *Bioinformatics*, 21(20):3940–3941, 2005. URL `http://rocr.bioinf.mpi-sb.mpg.de/`.

Jörg Stork, Ricardo Ramos, Patrick Koch, and Wolfgang Konen. SVM ensembles are better when different kernel types are combined. In Berthold Lausen, editor, *European Conference on Data Analysis (ECDA13)*. GfKl, 2013. URL `http://www.gm.fh-koeln.de/~konen/Publikationen/storkECDA-2013.pdf`.