The TDMR Framework: Tuned Data Mining in R

Wolfgang Konen, Patrick Koch

Cologne University of Applied Sciences

Last update: October 2011

Overview

The TDMR framework is written in R with the aim to facilitate the training, tuning and evaluation of data mining models. It puts special emphasis on tuning these data mining models as well as simultaneously tuning certain preprocessing options. TDMR is especially designed to work with SPOT [Bart10e] as the preferred tuner, but it offers also the possibility to use other tuners, e.g., CMA-ES [Hans06], LHD [McKay79] or direct-search optimizers [BFGS, Powell] for comparision.

This document

- gives a short overview over the TDMR framework,
- explains some of the underlying concepts and
- shows an example usage: how to use TDMR on a new data mining task.

This document concentrates more on the software usage aspects of the TDMR framework. For a more scientific discussion of the underlying concepts and the results obtained, the reader is referred to [Kone10a, Kone11b].

TDMR Workflow

Phase 1: DM without SPOT

Two kinds of DM tasks, classification or regression, can be handled. The corresponding subdirectories are ClassifyTemplate/ and RegressionTemplate/.

For each DM task TASK, create one task-specific function main_TASK(opts=NULL) in the corresponding subdir, as short as possible. If called without any parameter, main_TASK() should set default parameters for opts. main_TASK() reads in the task data, does the preprocessing if necessary and then calls with the preprocessed data dset the task-independent functions tdmClassifyLoop or tdmRegressLoop, which in turn call the task-independent functions tdmClassify or tdmRegress.

A template may be copied from ClassifyTemplate/main_sonar.r. It is invoked with

> result <- main_sonar()

(*see here for an overview of elements in list* `result`).

Phase 2: Tuned Data Mining (TDMR)

A TDMR task consists of a DM task (Phase 1) plus a SPOT configuration (decision which parameters to tune within which ROI, which meta parameters to set for SPOT, …).

For each DM task a subdir TASK in TDM.SPOT.d should be created. In this subdir the files shown in Table 1 have to be created for each SPOT configuration (each TDMR task):

Table 1: Configuration files for a SPOT run

.apd	problem design: all opts-settings
.roi	SPOT ROI file, specifies which parameters to tune in which ROI
.conf	SPOT configuration file, usually with alg.func = "tdmStartSpot". Furthermore, io.apdFileName and io.roiFileName should specify the two files above.

Templates for these three files may be copied from inst/sonar/sonar_01.*.

Then, the whole SPOT tuning can be started with the following code snippet:

```
confFile = "sonar_01.conf";
tdm=list(mainCommand="result<-main_sonar(opts)"
        ,mainFile="../../tdm/demo/main_sonar.r");
spotUserConfig = list(tdm=tdm,spot.fileMode=F);
spotConfig = spot(confFile,"auto",spotConfig=spotUserConfig);
```

SPOT will first read in the settings from confFile, then append/overwrite the settings from spotConfig. SPOT will then call the generic function **tdmStartSpot**(spotConfig), which reads in the .apd file, 'sources' tdm$mainFile, changes to the directory of tdm$mainFile, and executes there tdm$mainCommand.

The only requirement on tdm$mainCommand is that it returns in

result$y

a suitable quantity to be **minimized** by SPOT.

If spot.fileMode==T, SPOT will generate .des and .aroi files (needed by SPOT internally) and the output files .bst and .res.

If spot.fileMode==F, tdmStartSpot will read the design from spotConfig$alg.currentDesign and it writes the .res data frame onto spotConfig$alg.currentResult.

Details:

- For a new task TASK, the opts-part of .apd can usually be copied from the opts-part of main_TASK.
- Usually, TASK_02.apd, TASK_03.apd, … will start with `source(TASK_01.apd, local=T)` and will only specify those elements of opts which need to be different.
- For reproducability of experiments each TDMR task should get its own task name TASK_01, TASK_02, … and the associated set of files (.apd, .conf, .roi …) should kept unchanged for further reference. DO NOT alter later the settings in a TDMR task file (unless you want to delete and overwrite the old experiment), but create a new TASK_xx with its own set of files.
- If a new parameter appears in a .roi file which never appeared in any other .roi file before, a line has to be added to tdmMapDesign.csv, specifying the mapping of this parameter to the corresponding element of opts. (*more details here*)
- For the current developper version SPOT is loaded from source files (pre-defined locations, may need adjustments in sourceSPOT.R). If you want to start SPOT simply from the CRAN package version, which has been installed as library in the usual way, set
  tdm$theSpotPath <- NA;
  If you want to load SPOT from source files in pre-defined loacations (see sourceSPOT.R), set
  tdm$theSpotPath <- "USE.SOURCE";
  If you want to load SPOT from your own source directory, set tdm$theSpotPath to this directory.
- How SPOT handles it, if confFile and spotConfig are both present, e.g. in a call
  `spot(confFile,"auto",spotConfig)`:
  - Initial defaults for all elements in spotConfig are set inside SPOT (see spotGetOptions.R).
  - If `confFile` exists (only then!), it is read and settings found in `confFile` overwrite the defaults (see spotGetOptions.R).
  - If `spot` is called with parameter spotConfig present, then the elements found in this command line parameter overwrite the settings of step 2.

Phase 3: "The Big Loop": Several TDMs with Unbiased Evaluations

"The Big Loop" is a script to start several Phase-2-TDMR tasks (usually on the same DM task), optionally with several tuners (*see here for a list of tuners*) and compare their best solutions with different modes of unbiased evaluations, e.g. on unseen test data (tdm$umode = "TST") or by starting a new, independent CV (tdm$umode = "CV") or by starting a new, independent resubsampling (tdm$umode = "RSUB").

To start the Big Loop, only one file has to be created in TDM.SPOT.d/TASK: script_all.R. A template may be copied from inst/sonar/.

It is invoked with

```
source("script_all.R")
```

This will specify in `runList` the list of TDMR tasks and a list of tuners. For each TDMR task and each tuner

(a) the tuning process is started (if spotStep="auto") or a previous tuning result is read in from file (if spotStep="rep") and
(b) one or more unbiased evaluations are started. This is to see whether the result quality is reproducible on independently trained models and / or on independent test data.

The result is a data frame `theFinals` with one row for each TDMR task / each tuner and several columns measuring the success of the best tuning solution in different unbiased evaluations, see **Table 4.** The data frame `theFinals` is written to tdm$finalFile.

Details:

- The unbiased evaluations are done for each element of tdm$umode by calling the function unbiasedBestRun_*(…,umode,…) [*=C for classification and *=R for regression]. The function unbiasedBestRun_* reads in the best solution of a tuning run from .bst file, performs a re-run (training + test) with these best parameters.
- script_all.R 'sources' all necessary R-files, specifies a list of TDMR tasks in runList (a list of .conf files) and specifies a list of tuners in tdm$tuningMethod, e.g. c("spot","cmaes"), sets other values of tdm and calls tdmCompleteEval.
- script_all.R should be created in and called from the TASK subdir (e.g. TDM.SPOT.d/sonar/). The .conf files in runList should reside in the same directory and should be given w/o path (since TDMR will infer other files, e.g. sonar_01.apd, from it).
- DO NOT call any of the functions tdmCompleteEval, tdmDispatchTuner_tdm while being in the parent directory TDM.SPOT.d – the current workflow does not support this.
- spotList is a list of .conf files for which the tuners will be started (NULL for all from runList). If a tuner is not started for a certain .conf file it is assumed that its .bst file already exists from a prior run.
- spotStep is a list of strings (may be shorter than runList, then it is cyclically reused) which specifies the SPOT step to be invoked. If e.g. the step is "rep" ("report"), then it is assumed that the .bst file already exists.
- script_all.R starts the definition of list tdm. If some elements are not def'd, suitable defaults will be added later to tdm at the beginning of tdmCompleteEval.

TDMR Important Variables

Table 2: Overview of important variables in TDMR

<table>
<tr><td><code>opts</code></td><td>list with DM settings (used by main_TASK and its subfunctions). Parameter groups:
• <code>opts$READ.*</code> # reading the data
• <code>opts$TST.*</code> # test set and resampling
• <code>opts$PRE.*</code> # preprocessing
• <code>opts$SRF.*</code> # sorted random forest (or similar other variable rankings)
• <code>opts$RF.*</code> # Random Forest
• <code>opts$SVM.*</code> # Support Vector Machine</td></tr>
</table>

	• `opts$GD.*` # graphic device issues
`dset`	preprocessed data set (used within main_TASK and its subfunctions)
`result`	list with results from Phase 1: In the case of regression, this list contains (see tdmRegress.r): • `opts` # with some settings perhaps adjusted in tdmRegress • `last_res` # last run, last fold: result from tdmRegress • `R_train` # RMAE on training set (vector of length NRUN) • `S_train` # RMSE on training set (vector of length NRUN) • `T_train` # Theil's U for RMAE on training set (vector of length NRUN) • `*_test` # --- similar, with test set instead of training set • `y` # what to be minized by SPOT, usually `mean(R_test)` In the case of classification, this list contains (see tdmClassify.r): • `opts` # with some settings perhaps adjusted in tdmClassify • `last_res` # last run, last fold: result from tdmClassify • `C_train` # classification error on training set (vector of length NRUN) • `G_train` # gain on training set (vector of length NRUN) • `R_train` # relativ gain (% of max. gain) on training set (vector of length NRUN) • `*_test` # --- similar, with test set instead of training set • `*_test2` # --- similar, with test2 set instead of training set • `y` # what to be minized by SPOT, usually `mean(-R_test)`
`tdm`	list with settings for Phase 2 and 3. Elements are • `mainFile` (with path relative to current dir) • `mainCommand` (string, e.g. "result <- main_sonar(opts) ") • `unbiasedFunc` (string, e.g. "unbiasedBestRun_C") • `umode`: list of unbiased evaluation modes, with elements from {“TST”,”RSUB”,”CV”}, see map.des.r, tdmCompleteEval.r • `finalFile`, (string, e.g. "sonar.fin") • `withParams`: T/F, has theFinals columns with best parameters?
`finals`	see here
`envT`	environment, see here

TDMR `opts` Concept

`opts` is a long list with many parameters which control the behaviour of main_TASK, i.e. the behaviour of Phase 1. To give this long list a better structure, the parameters are grouped with key words after “`opts$`” and before “`.`” (see table above).

There are some other parameters in opts which do not fall in any of the above groups, e.g.

- opts$NRUN
- opts$VERBOSE
- opts$CLASSWT

and others.

You might either specify all opts-parameters in your application (i.e. main_TASK.r or *.apd) or you might use tdmSetOptsDefaults() and specify only those of the opts-parameters which differ from this defaults or you enter main_TASK.r with a partially filled opts and leave the rest to function tdmFillOptsDefaults (in tdmOptsDefaults.r), which is called from main_TASK after the user’s opts-settings (because some settings might depend on the user’s prior settings).

Details:

- If the list opts is extended by element X in the future, you need only to add a default specification of opts$X in function tdmFillOptsDefaults, and all functions called from main_TASK will inherit this default behaviour.
- TODO: why two function set.. and fil… : set only if opts==NULL
- opts$TST.kind="rand" triggers random resampling for the division of dset into training and test set. In the case of classification this resampling is done by **stratified sampling**: each level of the response variable appears in the training set in proportion to its relative frequency in dset, but at least with one record. This last condition is important to ensure proper functioning also in the case of 'rare' levels (most DM models will crash if a certain level does never appear in the training set). In the case of regression the sample is drawn randomly (without stratification).

TDMR RGain Concept

For **classification**: The R_-elements (i.e. result$R_train and result$R_test) can contain different things, depending on the value of opts$rgain.type:

- "rgain" or NULL [def.]: the relative gain in percent, i.e. the gain actually achieved divided by the maximal achievable gain on the given data set,
- "meanCA": mean class accuracy: For each class the accuracy on the data set is calculated and the mean over all classes is returned,
- "minCA": same as "meanCA", but with min instead of mean. For a two-class problem this is equivalent to maximizing the min(Specifity,Sensitivity) (see here).

In each case, TDMR seeks to minimize "–RGain", i.e. to maximize RGain.

For **regression**: The R_-elements (i.e. result$R_train and result$R_test) can contain different things, depending on the value of opts$rgain.type:

- "rmae" or NULL [def.]: the relative mean absolute error RMAE, i.e. the mean $<|y - y^{(pred)}|>$ divided by the mean $<|y|>$,
- "rmse": root mean square error.

In each case, TDMR seeks to minimize result$R_*.

Example Usage

The usage of the TDMR workflow is fairly easy. We show it for the three workflow phases and for the example of the SONAR classification task.

Phase 1: DM on task SONAR

If you want to build a DM classification model for the SONAR data (see UCI repository or package mlbench for further info on SONAR), you write a file main_sonar.r in directory ClassifyTemplate:

```
main_sonar <- function(opts=NULL) {
          tdmPath <- "../tdm";
          source(paste(tdmPath,"source.tdm.r",sep="/"));  source.tdm(tdmPath);

          if (is.null(opts)) {
            opts = tdmOptsDefaultsSet();  # set initial defaults for many elements  of opts. See tdmOptsDefaults.r
                                          # for the list of those elements and many explanatory comments
              opts$filename = "sonar.txt"
              opts$data.title <- "Sonar Data"
          }
          opts <- tdmOptsDefaultsFill (opts,".txt");    # fill in all opts params which are not yet set (see tdmOptsDefaults.r)

          tdmGraAndLogInitialize(opts);              # init graphics and log file

          #===============================================
          # PART 1: READ DATA
          #===============================================
          cat1(opts,opts$filename,": Read data ...\n")
          dset <- read.csv2(file=paste(opts$dir.data, opts$filename, sep=""), dec=".", sep=",",header=F)
             names(dset)[61] <- "Class"       # 60 columns V1,...,V60 with input data,
```

```
                                                  # one response column "Class" with levels ["M" (metal) | "R" (rock)]

        response.variable <- "Class"        # which variable is response variable

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

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

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

        tdmGraAndLogFinalize(opts);             # close graphics and log file

        result;
}
```

This function is invoked with

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

The control flow will pass through the branch if (is.null(opts)), where all defaults for opts are set with function tdmOptsDefaultsSet(). This specifies for example, that an RF model will be built. The dataset will be divided in a training part (90%) and test part (10%), based on opts$TST.kind=”rand”, opts$TST.frac=0.1. You need to specify only those things which differ from tdmOptsDefaultsSet(): in this case the filename of the SONAR dataset. Since you do not specify anything from the opts$SRF-block, you use the default SRF variable ranking (opts$SRF.kind =”xperc”, opts$SRF.Xperc=0.95). This means that the least important columns containing about 5% of the overall importance will be disregarded.

You need to specify what column in dset is response variable (classification target) and what columns are used for input (in this case all the others, because the SONAR dataset does not have ID columns or otherwise irrelevant columns).

Function tdmClassifyLoop() is started, it builds an RF model using the training data and evaluates it on training and test data.

Some output:

```
sonar.txt : Train RF (importance) ...
Target levels:  M R
sonar.txt : Saving sorted RF importance to file ./Output/sonar.txt.SRF.Class.Rdata ...
Variables sorted by importance (60 ):
[1] "V9"  "V11" "V12" "V10" …
Dropped columns (8 [=  4.8% of total importance]):
[1] "V1"  "V30" "V60" "V56" "V41" "V24" "V25" "V57"
Training cases ( 188 ):
          predicted
actual   M    R
      M  98    8
      R  22   60
total gain:   158.0 (is   0.840% of max. gain =   188.0)
```

Some graphics output:

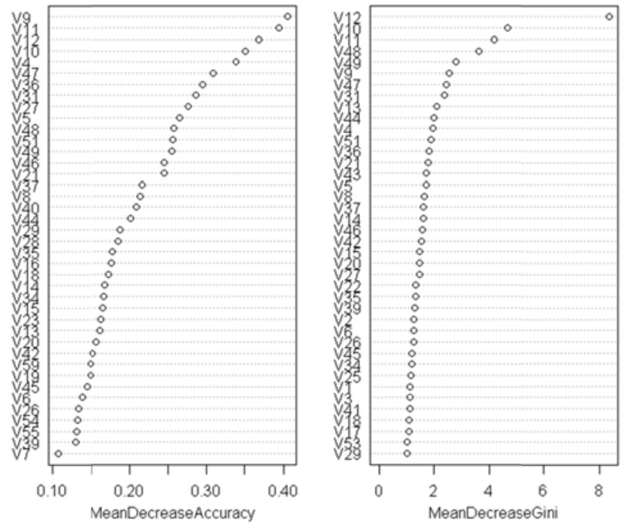


Figure 1: RF-based variable importance for task SONAR.

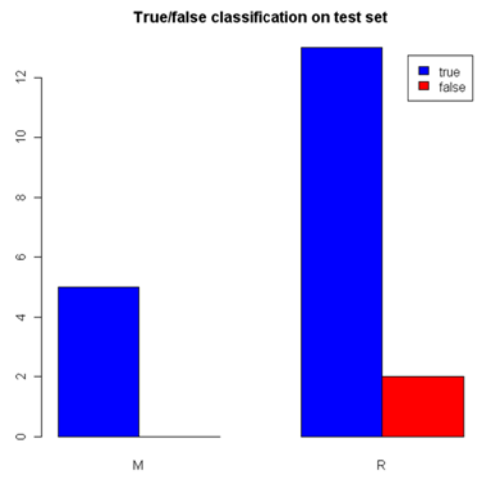


Figure 2: True/false classification for the two classes M and R of task SONAR.

The two plots in Figure 1 show the RF-based importance, where MeanDecreaseAccuracy, which has V9, V11 and V12 as the most important variables, is the more reliable measure. The right plot in Figure 2 shows the true/false classifications on the test set (which is here however rather small, so the results are not very reliable, a more reliable test set classification would be obtained with CV).

Phase 2: SPOT tuning on task SONAR

If you want to do a SPOT tuning on task SONAR, you should follow the steps described in TDMR Workflow, Phase 2 and create the three small files sonar.conf, sonar.apd and sonar.roi. The files' content may look for example like this:

sonar.conf:

```
alg.language = "sourceR"
alg.path="."
alg.func = "tdmStartSpot"
alg.resultColumn = "Y"
alg.seed = 1235

io.apdFileName = "sonar_01.apd"
io.roiFileName = "sonar_01.roi"
io.verbosity = 3;
auto.loop.steps = 50;      # number of SPOT's sequential generations
auto.loop.nevals = 100;   # concurrently, max number of algo evaluations may be specified

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.apd:

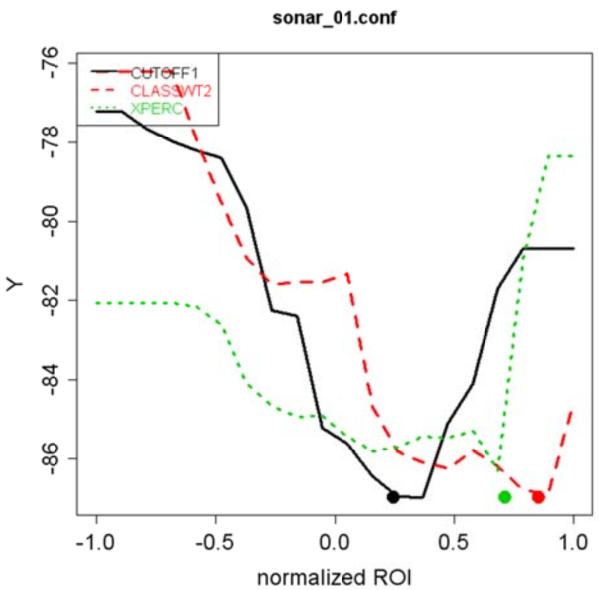
```
if (is.na(match("tdm",ls()))) tdm <- list();
tdm$mainFile <- "../../ClassifyTemplate/main_sonar.r";
tdm$mainCommand <- "result <- main_sonar(opts)";

opts = tdmSetOptsDefaults();     # set initial defaults for many elements of opts.
opts$filename = "sonar.txt"
opts$data.title <- "Sonar Data"
opts$RF.mtry = 4
opts$NRUN =  1        # how many runs with different train & test samples  - or -
                      # how many CV-runs, if TST.kind="cv"
opts$GRAPHDEV="non";
opts$GD.RESTART=F;
opts$VERBOSE= opts$SRF.verbose = 0;
```

sonar.roi:

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

The three parameter CUTOFF1, CLASSWT2 and XPERC are tuned within the borders specified by sonar.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.



After this preparation the whole SPOT tuning is started with

> sC <- spot("sonar_01.conf","auto")

(*more details*). It will generate the usual SPOT result files (see SPOT manual [Bart10e])

- sonar_01.res
- sonar_01.bst

The tuning will stop after 16 sequential steps with the configuration CONFIG=58, because the budget of auto.loop.nevals=100 evaluations is exhausted. The best solution can be seen from the last line of sonar_01.bst (or alternatively from the printout of spotConfig$alg.currentBest).

With

> spot("sonar_01.conf","rep")

the results from a prior tuning run producing sonar_01.res are read in again and a report including a sensitivity plot (see Figure 3) is made.

Details:

When spot("sonar_01.conf","auto") is invoked, the following things happen:

- SPOT is started, reads from sonar_01.conf that it has to call the inner function
  alg.func = "tdmStartSpot".
- tdmStartSpot.r in turn will read sonar_01.apd and learn from this, that

```
tdm$mainFile <- "../../ClassifyTemplate/main_sonar.r";
tdm$mainCommand <- "result <- main_sonar(opts)";
```

i.e. the DM template is main_sonar.r to be invoked with result <- main_sonar(opts). opts will be filled by tdmStartSpot according to the actual SPOT design.

- If a new parameter appears in a .roi file which never appeared in any other .roi file before, a line has to be added to tdmMapDesign.csv, specifying the mapping of this parameter to the corresponding element of opts. (*more details here*)
- Now tdm$mainCommand is started and runs the data mining process. The DM template main_sonar is provided by the user. The only requirement of SPOT or other tuners for the function main_sonar is that it returns in

  result$y

  a suitable quantity to be **minimized** by SPOT.

Phase 3: "The Big Loop" on task SONAR

To start "The Big Loop", you configure a file script_all.R in TDM.SPOT.d/sonar/, which may look like this:

```
tdm <- list(tdmPath="../../tdm" # NULL
        , unbiasedFunc="unbiasedBestRun_C"
        , umode=c("CV")     # ,"RSUB"
        , mainFile="../../ClassifyTemplate/main_sonar.r"
        , mainCommand="result <- main_sonar(opts)"
        , tuneMethod=c("spot")  #  ,  "spot"   "cmaes"   "bfgs" "lhd"
        , finalFile="sonar.fin"
        , experFile=NULL # "sonar.exp"
        , nrun=5, nfold=2        # repeats and CV-folds for the unbiased runs
        , optsVerbosity=0         # the verbosity for the unbiased runs
        , withParams=T
        , nExperim=1
        , parallelCPUs = 1        # [1] 1: sequential, >1: parallel with snowFall and this many cpus
        );
tdm$theSpotPath <- NA;
source(paste("../start.tdm.r",sep="/"),local=T);
runList = c("sonar_04.conf"); # ,"sonar_01.conf","sonar_02.conf","sonar_03.conf"); #
spotList = NULL # list() #      #  =NULL: all in runList; =list(): none
spotStep = "auto"

envT <- tdmCompleteEval(runList,spotList,spotStep,tdm);
```

This file will trigger the following sequence of experiments:

- sonar_02.conf is started with tuner (a) lhd and (b) spot
- sonar_03.conf is started with tuner (a) lhd and (b) spot

This sequence of 4 tuning experiments is repeated nExperim=2 times. The corresponding 8 result lines are written to opts$finalFile. If (opts$experFile != NULL), these lines are also appended to file opts$experFile. The switch withParams=T is only sensible if both .conf files have the same set of parameters in their .roi file.

The result `theFinals` from the last experiment (4 result lines) is in file sonar.fin:

CONF	TUNER	CLASSWT2	XPERC	NRUN	NEVAL	RGain.bst	RGain.avg	RGain.OOB	sdR.OOB	RGain.RSUB	sdR.RSUB
sonar_02	lhd	12.026543	0.930197	3	36	86.70213	84.3676	84.4311	1.03715	83.73984	5.63268
sonar_02	spot	14.713475	0.981312	3	36	86.96809	84.6926	85.6287	1.03715	86.99187	7.04085
sonar_03	lhd	8.037636	0.954494	3	36	81.91489	78.6643	80.4391	1.82937	79.67480	7.45134
sonar_03	spot	7.375221	0.914740	3	35	81.91489	78.7082	78.8423	0.34571	74.79675	2.81634

Here CLASSWT2 and XPERC are the tuning parameters, the other columns are defined in **Table 4**.

In the case of the example above, the tuning process had a budget of NEVAL=36 model trainings, resulting in a best solution with class accuracy RGain.bst (in %). The average class accuracy (mean over 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. Since the classification model in this example is RF (Random Forest), an OOB-error from the 3 trainings is returned, with average RGain.OOB and standard deviation sdR.OOB. Additionally, NRUN=3 trainings are done with random subsampling (RSUB) of the data set in training and test set, resulting in an average class accuracy on the test set RGain.RSUB and the corresponding standard deviation in sdR.RSUB.

In this case the interpretation of the results is quite clear: The best configuration is sonar_02.conf with TUNER spot, since this line contains the maximum for all columns RGain.bst, RGain.avg, RGain.OOB and RGain.RSUB. Note that the standard deviation sdR.RSUB is in this case quite large (because the test set is small). A more reliable result might be obtained with "CV" instead of "RSUB".

TDMR seed Concept

In a TDMR task there are usually several places where non-deterministic decisions are made and therefore certain questions of reproducability / random variability arise:

1) Design point selection of the tuner,
2) Test/training-set division and
3) Model training (depending on the model, RF and neural nets are usually non-determininstic, but SVM is deterministic).

Part 1) is in the case of SPOT tuning controlled by the variable spot.seed in the .conf file. You may set spot.seed={any fixed number} for selecting exactly the same design points in each run. (The design point selection is however dependent on the DM process: If this process is non-deterministic (i.e. returns different y-values on the same initial design points, you will usually get different design points from sequential step 2 on.) Or you set spot.seed=tdmRandomSeed() and get in each tuning run different design points (even if you repeat the same tuning experiment and even for a deterministic DM process).

In the case of CMA-ES or other tuning algorithms, we use `set.seed(spotConfig$spot.seed)` right before we randomly select the initial design point and ensure in this way reproducibility.

Part 2) and 3) belong to the DM process and the TDMR software supports here three different cases of reproducability:

a) Sometimes you want two TDMR runs to behave exactly the same (e.g. to see if a certain software change leaves the outcome unchanged). Then you may set opts$TST.SEED={any fixed number} and opts$MOD.SEED={any fixed number}.
b) Sometimes you want the test set selection (RSUB or CV) to be deterministic, but the model training process non-deterministic. This is the case if you want to formulate problem tasks of exactly the same difficulty and to see how different models – or the same model in different runs – perform on these tasks. Then you may set opts$TST.SEED={any fixed number}, opts$MOD.SEED=NULL.
c) Sometimes you want both parts, test set selection and model training, to be non-deterministic. This is if you want to see the full variability of a certain solution approach, i.e. if you want to measure the degree of reproducability in a whole experiment. Then you may set opts$TST.SEED= NULL; opts$MOD.SEED=NULL.

(The case {TST.SEED= tdmRandomSeed(); MOD.SEED=any value} is a fourth possibility, but it has – as far as I can see – no practical application).

Here **tdmRandomSeed** is a function which returns a different integer seed each time it is called. This is even true, if it is called multiple times within the same second (where a function like Sys.time() would return the same number). This can easily happen in parallel execution mode, where processes on different slaves usually will be started in the same second.

A second aspect of random variability: We usually want each run through main_TASK (loop over `i in 1:opts$NRUN` in tdmClassifyLoop) <u>and</u> each repeat during tuning (loop over `r in 1:des$REPEATS[k]` in tdmStart*) to explore different random regions, even in the case where all seed settings (spot.seed, opts$TST.SEED an opts$MOD.SEED) are fixed. We achieve this by storing the loop variables `i` and `r` in `opts$i` and `opts$rep`, resp., and use in tdmClassify.r the specific seeds

```
newseed=opts$MOD.SEED+(opts$i-1)+opts$NRUN*(opts$rep-1);
```

and

```
newseed=opts$TST.SEED+(opts$i-1)+opts$NRUN*(opts$rep-1);
```

In this way, each run through main_TASK gets a different seed. If opts$*.SEED is {any fixed number}, the whole process is however exactly reproducible.

for (opts$rep = 1 : des$REPEATS[k])

main_TASK

for (opts$i = 1 : opts$NRUN)

tdmClassifyLoop

Why is `opts$MOD.SEED=tdmRandomSeed()` and `opts$MOD.SEED=NULL` different? – The first statement selects a random seed at the time of definition time of `opts$MOD.SEED`, but uses it then throughout the whole tuning process, i.e. each design point evaluation within this tuning has the same `opts$MOD.SEED`. The second statement, `opts$MOD.SEED=NULL`, is different: Each time we pass through tdmClassify (start of response.variable-loop) we execute the statement

```
set.seed(tdmRandomSeed())
```

which selects a new random seed for each design point and each run.

Details

(RNG = random number generator)

- If TST.SEED=NULL, the RNG seed will be set to (a different) number via tdmRandomSeed() in each pass through the nrun-loop of tdmClassifyLoop / tdmRegressLoop (at start of loop).
- If MOD.SEED= NULL, the RNG seed will be set to (a different) number via tdmRandomSeed() in each pass through the response.variable-loop of tdmClassify / tdmRegress (at start of step 4.3 "model training").
- Before Nov'2010 the TDMR software would not modify RNG seed in any way if TST.SEED=NULL. But we noticed that with a call from SPOT two runs would exactly produce the same results in this case. The reason is that SPOT fixes the RNG seed for each configuration in the same way and so we got the same model training and test set results. To change this, we moved to the new behaviour, where each *.SEED=NULL leads to a "random" RNG-seed at appropriate places.
- TODO: Check if reproducability is fulfilled, if all seed-params are non-NULL (!)

TDMR Tuner Concept

How to use different tuners

If you want to tune a TDMR-task with two tuners SPOT and CMA-ES: Simply specify

tdm$tuneMethod = c("spot","cmaes")

in script_all.R and set the variable spotStep to "auto". The tuning results (.bst and .res files) will be copied into subdirs "spot" and "cmaes" of the directory from which you start script_all.R.

Table 3: Tuners availabe in TDMR

tdm$tuneMethod	**Description**

spot	Sequential Parameter Optimization Toolbox
lhd	Latin Hypercube Design (truncated SPOT, all budget for the initial step)
cmaes	Covariance Matrix Adaption ES
powell	Powell's Method (direct & local search)
bfgs	Broyden, Fletcher, Goldfarb and Shannon method (direct & local search)
bobyqa	direct & local search, with constraint handling

How to integrate new tuners

Originally TDMR was only written for SPOT as tuning method.

In November 2010, we started to add other tuners to aid the comparision with SPOT on the same footing. As the first other tuner, we introduced CMA-ES (Niko Hansen, R-package by Olaf Mersmann and others). Since comparision with SPOT is the main issue, CMA-ES was wrapped in such a way in tdmDispatchTuner.r that the behaviour and output is very similar to SPOT.

This has the following implications which should also be obeyed when adding other tuners to TDMR:

- Each tuning method has a unique name (e.g. "spot", "cmaes"): this name is an allowed entry for tdm$tuneMethod and finals$TUNER and it is the name of a subdir in TDM.SPOT.d/TASK/.
- Each tuner writes result files (.bst, .res) in a fashion similar to SPOT. These result files are copied to the above mentioned subdir at the end of tuning. This facilitates later comparision of results from different tuners.
- Each tuner supports at least two values for spotStep: "auto" and "rep" (="report"). In the latter case it is assumed that .bst and .res already exist (in their subdir) and they are usually analysed with spot(confFile,"rep",…).
- Each tuner reads in the .conf file and infers from spotConfig the tuner settings (e.g. budget for function calls, max repeats, …) and tries to make its tuning behaviour as similar to these settings as possible.

For the current CMA-ES tuner the relevant source code for integration in TDMR is in functions tdmDispatchTuner and cmaesTuner (both in tdmDispatchTuner.r) and in tdmStartCMA.r.

These functions may be used as templates for the integration of other tuners in the future.

TDMR Experiment Concept

TDMR Phase 3 ("The Big Loop") allows

(a) to conduct experiments, where different .conf files, different tuners, different unbiased evaluations, … are tried on the same task;
(b) to repeat certain experiments of kind (a) multiple times with different seeds (tdm$nExperim>1).

Details:

- Each experiment of kind (a) initially deletes file tdm$finalFile, if it exists, and then writes for each combination {.conf file, tuner} it encounters a line to tdm$finalFile (usually a file with suffix .fin). This line is a one-row data frame `finals` which is built in unbiasedBestRun_C.r (classification) and contains the columns listed in **Table 4**.

Table 4: Elements of data frame `finals`

finals$	Description	Condition
<<columns obtained from the tuning process>>		
CONF	the base name of the .conf file	
TUNER	the value of tdm$tuneMethod	

{PARAMS}	all tuned parameters appearing in .roi file	if tdm$withParams==T
NEVAL	tuning budget, i.e. # of model evaluations during tuning (rows in data frame res)	
RGain.bst	best solution (RGain) obtained from tuning	
RGain.avg	average RGain during tuning (mean of res$Y)	
<<columns obtained from the unbiased runs>>		
NRUN	# of runs with different test & train samples in unbiasedBestRun_*.r or # of unbiased CV-runs. Usually NRUN = tdm$nrun, see fct map.opts in tdmMapDesign.r.	
RGain.OOB	mean OOB training error (averaged over all unbiased runs)	if opts$method = *.RF
sdR.OOB	std. dev. of RGain.OOB	if opts$method = *.RF
RGain.TRN	mean training error (averaged over all unbiased runs)	if opts$method ≠ *.RF
sdR.TRN	std. dev. of RGain.TRN	if opts$method ≠ *.RF
RGain.RSUB	mean test RGain (test set = random subsample)	if tdm$umode has "RSUB"
sdR.RSUB	std. dev. of RGain.RSUB (averaged over all unbiased runs)	if tdm$umode has "RSUB"
RGain.TST	mean test RGain (test set = separate data, user-provided)	if tdm$umode has "TST"
sdR.TST	std. dev. of RGain.TST (averaged over all unbiased runs)	if tdm$umode has "TST"
RGain.CV	mean test RGain (test set = CV, cross validation with tdm$nfold CV-folds	if tdm$umode has "CV"
sdR.CV	std. dev. of RGain.CV (averaged over all unbiased runs)	if tdm$umode has "CV"

- In the case of regression experiments (unbiasedBestRun_R.r) each "RGain" has to be replaced by "RMAE" in the table above, see here for further explanation.
- If tdm$experFile is not NULL, then the same one-row data frame `finals` is also appended to the file tdm$experFile. Usually, tdm$experFile is a file with `.exp` as suffix. This file is never deleted by the TDMR system, only the user may delete it. tdm$experFile serves the purpose to accumulate experiments carried out multiple times (with different random seeds). This multiple-experiment execution may be done either directly, within one 'big-loop' experiment, if tdm$nExperim>1, or it may be done subsequently by the user when starting script_all.R again at a later point in time with the same tdm$experFile defined.
- An .exp file can be analyzed with scripts like exp_summ.r in TDM.SPOT.d/appAcid/.

TDMR Design Mappping Concept

Each variable appearing in.roi file (and thus in .des file) has to be mapped on its corresponding value in list opts. This is done in the file tdmMapDesign.csv:

roiValue	optsValue	isInt
MTRY	opts$RF.mtry	1
XPERC	opts$SRF.Xperc	0
…	…	…

If a variable is defined with isInt=1, it is rounded in opts$... to the next integer even if it is a non-integer in the design file.

~~The file tdmMapDesign.csv exists twice, once in ClassifyTemplate/ and once in RegressionTemplate/, (because classification and regression might define different sets of parameters)~~

The base file **tdmMapDesign.csv** is read from <packageDir> = `.find.package("TDMR")` [1]. If in the <dir_of_main_task> = `dirname(tdm$mainFile)` an additional file **userMapDesign.csv** exists, it is additionally read in and added to the relevant data frame. The file userMapDesign.csv makes the mapping modifiable and extendable by the user without the necessity to modify the corresponding source file tdmMapDesign.r.

These files are read in when starting tdmCompleteEval via function tdmMapDesLoad and the corresponding data frames are added to `envT$map` and `envT$mapUser`, resp. This is for later use by function tdmMapDesApply; this function can called from the parallel slaves, which might have no access to a file system.

How to add a new tuning variable

TODO

Details

- We have beneath {tdmMapDesLoad, tdmMapDesApply} a second pair of functions {tdmMapDesSpot$load, tdmMapDesSpot$apply} with exactly the same functionality. Why? – The second pair of functions is for use in tdmStartSpot(spotConfig) where we have no access to envT due to the calling syntax of spot(). Instead the object tdmMapDesSpot store the maps in local, permanent storage of this object's environment. The first pair of functions is for use in tdmStartOther, especially when called by a separate R process when using the tuner cma_j. In this case the local, permanent storage mechanism does not work across different R sessions. Here we need the envT-based solution of the first pair, since the environment envT can be restored across R sessions easily via save & load.

TDMR parallel computing concept

How to use parallel computing

TDMR supports parallel computing through the packages snow and snowfall [Knaus08, Knaus09].

TODO: <<Describe sfInit-part>>

- We want to parallelize the tdmDispatchTuner-calls which are currently inside the 3-fold loop {tdm$nExperim, runList, tdm$tuneMethod). Therefore, the for-loops should be rewritten as sapply commands, which can be transformed to sfSapply. Define suitable inner functions for sapply.
- Four operating modes:

tdm$parallelCPUs	tdm$fileMode	mode
=1	FALSE	sequential, everything is returned via environment envT, no files are written
=1	TRUE	sequential, everything is returned via environment envT, and logged on several files
>1	FALSE	parallel, everything is returned via environment envT, no files are written or read
>1	TRUE	parallel, everything is returned via environment envT, and logged on several files

  =1/TRUE is the current state of the source code (May'2011).
  >1/TRUE is the parallel mode viable on maanvs-clusters at GM.
  >1/FALSE is the parallel mode needed for LIDO (TU DO). It requires more software redesign, since the code should make no file access (no sourcing, no data set reading!) below the call to tdmDispatchTuner.
  The switch tdm$fileMode==FALSE is not yet ready (as of June'2011), but should be available in the near future.

[1] resp. from tdm$tdmPath/inst/ for the developer version.

TODO:

- How to deal with the sourcing inside files (tdmDispatchTuner, tdmStartSpot, main_TASK)?
- How to transport the DM data dset if tdm$fileMode=FALSE? opts$dset?
- OK try to fill envT$opts at a high level in the calling hierarchy (before the parallel branches), so that we do not need source(pdFile) at lower levels (might not work on parallel clusters)

Environment envT

The environment envT is used to pass necessary information to and back from the parallel slaves. It replaces in nearly all cases the need for file reading or file writing. (File writing is however still possible for the sequential case or for parallel slaves supporting file access. File writing might be beneficial to trace the progress of parallel or sequential tuning processes while they are running and to log the resulting informations.)

Environment **envT** is constructed in tdmCompleteEval. **Table 5** shows its elements and it shows in the 3rd column which function adds these elements to envT:

Table 5: Elements of environment envT

variable	remark	function
bst	data frame with contents of **last** .bst file	tdmStartOther or spotTuner, lhdTuner
bstGrid	list with all bst data frames, bstGrid[[k]] retrieves the kth data frame	tdmCompleteEval or populateEnvT
getBst(conf,tuner,n)	function returning from bstGrid the bst data frame for configuration file `conf`, tuning method `tuner` and experiment `n`	tdmCompleteEval
res	data frame with contents of **last** .res file	tdmStart* or tdmCompleteEval
resGrid	list with all res data frames, resGrid[[k]] retrieves the kth data frame	tdmCompleteEval or populateEnvT
getRes(conf,tuner,n)	function returning from resGrid the res data frame for configuration file `conf`, tuning method `tuner` and experiment `n`	tdmCompleteEval
result	list with results of tdm$mainCommand as called in the **last** unbiased evaluation	unbiasedBestRun_C or unbiasedBestRun_R
theFinals	data frame with one row for each res file, see Table 4	tdmCompleteEval or populateEnvT
tdm		
opts		
spotConfig		tdmCompleteEval

envT is used to pass information back and forth between different fcts of TDMR, where envT$opts and envT$tdm pass info into tdmStart*, while envT$res and envT$bst are used to pass info back from tdmStart* to the main level.

Details

- We have in tdmCompleteEval only one parallelization mode (parallel over experiments, tuners and .conf files). We decided that it is sufficient to have one strategy for parallelization, for all values of tdm$parallelCPUs. We decided that it is dangerous to have nested sfSapply-calls.

- When does sfSapply return? – The snowfall manual says that sfSapply first hands out nCPU jobs to the CPUs, then waits for <u>all</u> (!) jobs to return and then hands out another nCPU jobs until all jobs are finished. sfSapply returns when the last job is finished. Therefore it is not clear what happens with nested sfSapply-calls and we make our design in such a way that no such nested calls appear.
- We added column NEXP (=envT$nExp) to tdm$finalFile and tdm$experFile. So it might be that older .fin and .exp files can no longer be merged with the new ones.
- In case tdm$nExperim>1 we write now on different .fin files, e.g.
  sonar-e01.fin, sonar-e02.fin, …
  This is to avoid that parallel executing tasks will remove or write on the same .fin file concurrently.
- How and when is the res data frame passed back from SPOT? (we get an error with spot.fileMode=F). The bst data frame is in spotConfig$alg.currentBest. – Answer: With the new SPOT package version (>0.1.1372) and with spot.fileMode==F, the res data frame is passed back in **spotConfig$alg.currentResult**. The user function spotConfig$alg.func is responsible for writing this data frame. We do this for both values of spot.fileMode: we start in fcts spotTuner and lhdTuner a new data frame **spotConfig$ alg.currentResult** (initially NULL) and fill it consecutively in tdmStartSpot.

TDMR Graphic Device Concept

*Utility Functions tdmGraphic**

These functions are defined in tdmGraphicUtils.r and should provide a consistent interface to different graphics device choices.

The different choices for opts$GRAPHDEV are

- "pdf": plot everything in one multipage pdf file opts$GRAPHFILE
- "png": each plot goes into a new png file in opts$GD.PNGDIR
- "win": each plot goes into a new window (X11())
- "non": all plots are suppressed (former opts$DO.GRAPHICS=F)

	opts$GRAPHDEV			
utility function	**"pdf"**	**"png"**	**"win"**	**"non"**
tdmGraphicInit	open multipage pdf	(create and) clear PNGDIR	-	-
tdmGraphicNewWin	-	open new png file in PNGDIR	open new window	-
tdmGraphicCloseWin	-	close png file	-	-
tdmGraphicCloseDev	close all open pdf devices	close all open png devices	close all devices (graphics.off())	-

tdmGraphicCloseWin does not close any X11()-window (because we want to look at it), but it closes the last open .png file with dev.off(), so that you can look at this .png file with an image viewer.

GD.RESTART, Case 1: main_TASK solo

if GD.RESTART==F: No window is closed, no graphic device restarted.

If GD.RESTART==T we want the following behaviour:

- close initially any windows from previous runs
- not too many windows open (e.g. if NRUN=5, nfold=10, the repeated generation of windows can easily lead to s.th. like 250 open windows)
- the important windows should be open long enough to view them (at least shortly)
- in the end, the last round of windows should remain open.

We achieve this behaviour with the following actions in the code for the case GD.RESTART==T:

- close all open windows when starting main_TASK
- close all open windows before starting the last loop (i==NRUN, k=the.nfold) of tdmClassify

- close all open windows when starting the graphics part (Part 4.7) of tdmClassify *UNLESS* we are in the last loop (i==NRUN, k=the.nfold); this assures that the windows remain open before the graphics part, that is during the time consuming training part.
- if GD.CLOSE==T and GD.GRAPHDEV!="win": close in the end any open .png or .pdf

GD.RESTART, Case 2: During SPOT-Run "auto"

This will normally have GD.RESTART=F: No window is closed, no graphic device restarted; but also GD.GRAPHDEV="non", so that no graphic is issued from main_TASK, only the graphics from SPOT.

GD.RESTART, Case 3: During unbiased runs

This will normally have also GD.RESTART=F and GD.GRAPHDEV="non": No graphics. But you might as well set GD.RESTART=T and choose any of the active GD.GRAPHDEV's before calling unbiaseBestRun_*, if you want the plots from the last round of unbiasedBestRun_*.

Summary

This report has shown how to use TDMR, the **T**uned **D**ata **M**ining framework in R. The examples shown should make the reader familiar with the concepts and the workflow phases of TDMR. They are deliberately made with fairly small datasets in order to facilitate quick reproducability. For results on larger datasets the reader is referred to [Kone10a, Kone11b].

References

[Bart10e] T. Bartz-Beielstein. SPOT: An R package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization. Technical Report http://arxiv.org/abs/1006.4645 . CIOP Technical Report 05-10, FH Köln, June 2010.

[Hans06] N. Hansen. The CMA evolution strategy: a comparing review. In: J. 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.

[Kone10a] W. Konen, P. Koch, O. Flasch, T. Bartz-Beielstein. Parameter-tuned data mining: A general framework. In F. Hoffmann and E. Hüllermeier, editors, Proceedings 20. Workshop Computational Intelligence. Universitätsverlag Karlsruhe, 2010.

[Kone11b] W. Konen, P. Koch, O. Flasch, T. Bartz-Beielstein. Tuned Data Mining: A Benchmark Study on Different Tuners. CIOP Technical Report 02-11, FH Köln, January 2011.

[Knaus08] Jochen Knaus, Parallel computing in R with sfCluster/snowfall, TR IMBI Uni Freiburg, http://www.imbi.uni-freiburg.de/parallel/.

[Knaus09] Knaus, J. and Porzelius, C. and Binder, H. and Schwarzer, G., Easier parallel computing in R with snowfall and sfCluster. The R Journal, 1:5459, 2009.

[McKay79] McKay, M.D.; Beckman, R.J.; Conover, W.J. (May 1979). "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code" (JSTOR Abstract). *Technometrics* (American Statistical Association) **21** (2): 239–245. doi:10.2307/1268522. OSTI 5236110. ISSN 0040-1706. http://www.jstor.org/pss/1268522.