The TDMR 2.2 Package: Tuned Data Mining in R

Wolfgang Konen, Patrick Koch,
Cologne University of Applied Sciences

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

Contents

1 Using TDMR

1.1 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. TDMR is especially designed to work with SPOT (Bartz-Beielstein, 2010) as the preferred tuner, but it offers also the possibility to use other tuners, e.g., CMA-ES (Hansen, 2006), LHD (McKay et al., 1979) or direct-search optimizers [BFGS, Powell] for comparison.

This document (TDMR-docu.pdf, Konen and Koch (2012a))

- gives a short overview over the TDMR framework,
- explains some of the underlying concepts and
- gives more details for the developer.

This document should be read in conjunction with the companion document TDMR-tutorial.pdf (Konen and Koch, 2012b), which shows example usages of TDMR in the form of lessons.

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

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

1.2 Installing TDMR

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

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

Then, library TDMR is loaded with

```
library(TDMR);

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

[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).

Table 1: Elements of `result`

`result`	list with results from Level 1:	
	In case of classification, object of class `TDMclassifier`, containing:	
	opts	# with some settings perhaps adjusted in tdmClassify
	lastRes	# last run, last fold: object of class `tdmClass`, see Tab. 4
	C_train	# classification error on training set (vector of length NRUN)
	G_train	# gain on training set (vector of length NRUN)
	R_train	# relative gain (% of max. gain) on training set (vector of length NRUN)
	*_test	# — similar, with validation 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)
	sd.y	# standard deviation of y over the opts$NRUN runs
	In case of regression, object of class `TDMregressor`, containing:	
	opts	# with some settings perhaps adjusted in tdmRegress
	lastRes	# last run, last fold: object of class `tdmRegr`, see Table 4
	R_train	# RMAE on training set (vector of length NRUN)
	S_train	# RMSE on training set (vector of length NRUN)
	T_train	# Theil's U for RMAE on training set (vector of length NRUN)
	*_test	# — similar, with validation set instead of training set
	y	# the quantity to be minized by the tuner, usually mean(R_test)
	sd.y	# standard deviation of y over the opts$NRUN runs

2 TDMR Workflow

2.1 Level 1: DM without Tuning

Two kinds of DM tasks, classification or regression, can be handled. For each DM task `TASK`, create one task-specific function `main_TASK(opts=NULL)`, as short as possible. If called without any parameter, `main_TASK()` should set default parameters for `opts` via `tdmOptsDefaultsSet()`. `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 `inst/demo02sonar/main_sonar.r`[2]. The template is invoked with

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

See **Lesson 1** in TDMR-tutorial.pdf (Konen and Koch, 2012b) for a complete example.

See Table 1 for an overview of elements in list `result`.

[2]Here and in the following `inst/` refers to the directory where the package TDMR is installed. Use `find.package("TDMR")` to locate this directory.

2.2 Level 2: Tuned Data Mining in R

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

If you want to do such a tuning on task SONAR, you should follow the steps described in TDMR-tutorial.pdf (see Konen and Koch (2012b), Sec. 3.2 *Lesson 2*) and create in addition to `main_sonar.r` and `opts` from Lesson 1 a SPOT configuration object `ctrlSC` and a TDMR configuration object `tdm`. This can be all done within `demo02sonar.r`:

```
demo(demo02sonar,ask=F);
```

This script will define a `main_TASK` in `tdm$mainFunc`, reads the data, sets the parameter for tuning and for TDMR and starts the tuning process, which repeatedly executes `tdm$mainFunc`. The only requirement on `tdm$mainFunc` is that it returns in

```
result$y
```

a suitable quantity to be minimized by SPOT.

See **Lesson 2** in TDMR-tutorial.pdf (Konen and Koch, 2012b) for the complete example.

2.3 Level 3: „The Big Loop“

„The Big Loop“ (several TDM runs with unbiased evaluations) is a script to start several Level-2-TDMR tasks (usually on the same DM task), optionally with

- several configuration objects `ctrlSC` and
- several tuners (see Table 6 for a list of tuners) and
- several experiments with different seeds.

The modes of unbiased evaluations allow to compare the best solutions obtained by the tuners. Different modes are availabe, e.g. to use unseen test data (`tdm$umode = "TST"`) or to start a new, independent CV (`tdm$umode = "CV"`) or to start a new, independent re-sampling (`tdm$umode = "RSUB"`).

To start the Big Loop, only one script file has to be created in the user directory. A template may be copied from `demo/demo03sonar.r`. It is invoked with

```
demo(demo03sonar,ask=F);
```

See TDMR-tutorial.pdf (see Konen and Koch (2012b), Sec. 3.3 *Lesson 3*) for the full details.

This will specify in the functions `controlSC` the list of TDMR configuration and in `tdm$tuneMethod` a list of tuners. For each TDMR task and each tuner

(a) the tuning process is started 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 `envT$theFinals` with one row for each TDMR run (triple CONF, tuner, experiment). Several columns measure the success of the best tuning solution in different unbiased evaluations, see Table 2.

More detailed results are returned in the environment `envT`. See Sec. 5.4 and Table 5 for more details on `envT`.

See **Lesson 3** in TDMR-tutorial.pdf (Konen and Koch, 2012b) for the complete example.

3 TDMR Experiment Concept

TDMR Level 3 („The Big Loop“) allows

(a) to conduct experiments, where different TDMR configurations, different tuners are tried on the same task;

(b) to repeat certain experiments of kind (a) multiple times with different seeds (`tdm$nExperim` > 1).

Each TDMR experiment consist of three parts:

Model building:
- During model building (training) and tuning the user starts with a data set, which is partitioned into training and validation set.
- The relative gain achieved on the validation set acts as performance measure for the tuning process.
- In the case of `opts$TST.kind=="cv"` or in the case `opts$NRUN` > 1 multiple models are build, each with its own training and validation set. In this case multiple relative gains are averaged to get the performance measure for the tuning process.

Tuning:
- The above model building process is started several times with different model parameters and preprocessing parameters (design points). The tuning process uses the performance measure returned to guide the search for better parameters.
- As a result of the tuning process, a best parameter set is established. It has a certain performance measure attached to it, but this measure might be too optimistic (e.g. due to validation data being used as training data in a prior tuning step or due to extensive search for good solutions in a noisy environment)

Unbiased Evaluation (Test):
- Once a best parameter set is established, an unbiased performance evaluation is recommended. This evaluation is done by calling `unbiasedRun()` with the object `dataObj` containing a split into test and training-validation data. See Sec. 4.2 on "Training, Validation, and Test Set".
- If `tdm$nrun` > 1, multiple calls to `unbiasedRun()` are executed. The performance measure returned is the average over all runs.

Table 2: Elements of data frame `envT$theFinals`

theFinals$	Description	Condition
– columns obtained from the tuning process –		
CONF	the base name of the configuration	
TUNER	the value of `tdm$tuneMethod`	
–PARAMS–	all the tuned parameters appearing in the ROI section of `controlSC`	`tdm$withParams==TRUE`
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$Y`)	
Time.TRN	time needed for tuning (see `tdm$timeMode` in Appendix C)	
– columns obtained from the unbiased runs –		
NRUN	# of runs with different test & train samples in `unbiasedRun` or # of unbiased CV-runs. Usually `NRUN = tdm$nrun`, see function `map.opts` in `tdmMapDesign.r`.	
RGain.TRN	mean training RGain (averaged over all unbiased runs). This is OOB training RGain in case `opts$MOD.method==*.RF` and the normal training RGain for all other model training methods	
sdR.TRN	std. dev. of RGain.TRN	
RGain.RSUB	mean test RGain (test set = random subsample)	if `tdm$umode=="RSUB"`
sdR.RSUB	std. dev. of RGain.RSUB	if `tdm$umode=="RSUB"`
RGain.TST	mean test RGain (test set = separate data, user-provided)	if `tdm$umode=="TST"`
sdR.TST	std. dev. of RGain.TST	if `tdm$umode=="TST"`
RGain.SP_T	mean test RGain (test set = split-test prior to tuning)	if `tdm$umode=="SP_T"`
sdR.SP_T	std. dev. of RGain.RSUB	if `tdm$umode=="SP_T"`
RGain.CV	mean test RGain (test set = CV, cross validation with `tdm$nfold` CV-folds	if `tdm$umode=="CV"`
sdR.CV	std. dev. of RGain.CV	if `tdm$umode=="CV"`
Time.TST	time needed for unbiased runs (see `tdm$timeMode` in Appendix C)	

Note that from the `RGain` columns with "if `tdm$umode`" only one will be selected, since `tdm$umode` has exactly one value. The same for the `sdR` columns.
For more information on `RGain` see Sec. 5.2. Mean and standard deviation of `RGain` are obtained by averaging over all unbiased runs.
In the case of regression experiments, each `RGain` has to be replaced by `RMAE` in the table above, see Sec. 5.2 (TDMR `RGain` Concept) for further explanation.

4 TDMR Data Reading and Data Split in Train / Validation / Test Data

4.1 Data Reading

TDMR reads the task data according to function `opts$READ.TrnFn`, usually from `opts$filename` (but this depends on `opts$READ.TrnFn`). Optionally, if `opts$READ.TstFn!=NULL`, test data are also read according to this function. Both function have the signature `function(opts)` and they have to return a data frame. Data are read prior to tuning when the object `dataObj` is constructed

```
opts <- tdmEnvTGetOpts(envT);
dataObj <- tdmReadAndSplit(opts,tdm,nExp);
```

where `tdmReadAndSplit` lets the function `tdmReadDataset` do the read work (using the options `opts$READ.TrnFn`, `opts$READ.TstFn`, `opts$READ.TXT`, and `opts$READ.NROW`).

This object `dataObj` is passed into `tdmBigLoop` or `tdmTuneIt`[3]. Inside `tdmBigLoop` or `tdmTuneIt`, `dataObj` is passed on to `tdmDispatchTuner` and `unbiasedRun`, where the training-validation data and the test data are extracted with

```
dset <- dsetTrnVa(dataObj);
tset <- dsetTest(dataObj);
```

and passed on to `main_TASK(..., dset=dset,tset=tset)`.

4.2 Training, Validation, and Test Set

4.2.1 Principles

In data mining we know three kind of data or data sets:

1. **Training set**: the data for learning or model training.
2. **Validation set**: the data used to obtain a performance measure of the trained model. The performance on the validation data is used to guide the tuning process.
3. **Test set**: When training and tuning is finished, we build a final model. To estimate the quality of the model for new data, we test its performance on test data. Usually, the test data were not seen by the model or the tuner. The user should NOT use the performance on the test data in any way to tune the model further.

[3]This is at least recommended behaviour. For downward compatibility, there exists the possibility to enter `tdmBigLoop` or `tdmTuneIt` with `dataObj=NULL` (not recommended). Then two further branches come into effect: a) if `opts$READ.INI=TRUE`: read the data at the beginning of `bigLoopStep` in `tdmBigLoop.r`; b) if `opts$READ.INI=FALSE`: set `dset=NULL,tset=NULL` and the data reading is done in `main_TASK`, for each tuning step anew.

Usually, the split into test set on the one side and training/validation set on the other side is done *once* prior to the tuning process. During tuning, many tuning steps are possible, each containing at least one model training and each step may have a new separation of the training/validation set into a training part and a validation part.

4.2.2 Test Set Splitting

How can we split the data into test set on one side and training/validation set (which we will abbreviate with TrnVaSet in the following) on the other side?

TDMR offers within the call

```
dataObj <- tdmReadAndSplit(opts,tdm,nExp);
```

four options [the value in brackets denotes the choice for `tdm$umode`]:

1. **TDMR sets a random fraction of the data aside for testing** [`"SP_T"`, **split-test**]**.**

   This is done once before the tuning starts. The test set (the data which are set aside for testing) is used only in unbiased evaluation. The whole procedure can be repeated (if `tdm$nExperim` > 1) and another random test set is set aside.

   This is the recommended option, it has a completely independent test set and allows to assess the variability due to varying test set selection.

   To use this option, set `tdm$umode="SP_T"` and `tdm$TST.testFrac` to the desired random fraction to be set aside (default is 10%). The splitting is coded in the column `dset$tdmSplit` with `0` for all records belonging to TrnVaSet and `1` for test data. Set `tdm$SPLIT.SEED=<number>` if you want reproducible splits (but varying for each experiment which has a different `<number>`).

2. **TDMR makes CV with different test set folds** [`"SP_CV"`]**.** TODO.

3. **User-defined test set splitting** [`"TST"`]**.** The user provides two reading functions `opts$READ.TrnFn` and `opts$READ.TstFn`. [4] TDMR reads both, adds then a new column `opts$TST.COL` to the data frames with `0` for the train/validataion data and `1` for the test data. Finally, both data frames are bound together into one data frame `dset`. [5]

4. **Test set is part of TrnVaSet** [`"RSUB"` **or** `"CV"`] **.** (NOTE: This option is strongly discouraged, since the test set is already visible during training and tuning, which inevitably leads to overfitting and / or oversearching. But sometimes you may have only very few data and cannot afford to set test data aside.)

   The whole data is used for training/validation and later also as the reservoir from which the test set sample is drawn.

[4] Often these functions will make use of the data file names `opts$filename` and `opts$filetest`, resp.

[5] There might be the special case that the user does not have training and test data in two different files, but instead she has a data frame or file containing both training and test records in different rows, distinguished by the value in a special column USER.COL. If this is the case, provide suitable functions `opts$READ.TrnFn` and `opts$READ.TstFn` which extract the appropriate subsets from the data frame. You might wish to subtract column USER.COL from the data frames returned.

To use this option, set `tdm$umode="RSUB"` or `tdm$umode="CV"`. In case `"RSUB"` set `tdm$TST.testFrac` to the desired random fraction to be drawn from the whole data (default is 20%). In case `"CV"` set `tdm$nfold` to the desired number of CV folds (default is 5).

With each of these choices for `tdm$umode`, the following happens during unbiased evaluation: A "fresh" model is build using (all or a fraction of) the data in TrnVaSet for training. Then this model is evaluated on the test data and the performance (relative gain or RMAE) on these test data is an unbiased estimator of the model's performance on new data.

4.2.3 `main_TASK` and its training/validation/test logic

The signature of function `main_TASK` is

```
main_TASK(opts=NULL, dset=NULL, tset=NULL)
```

It is usually called in three cases

Case 1 by the user (solo ML task or user-defined tuning procedure)

Case 2 from TDMR during tuning

Case 3 from TDMR during unbiased evaluation

In Case 2 the syntax is `main_TASK(opts,dset)`, where `dset = dsetTrnVa(dataObj)` and `tset = NULL`.

In Case 3 the syntax is `main_TASK(opts,dset,tset)`, where `dset = dsetTrnVa(dataObj)` and in addition `tset = dsetTest(dataObj)`.

How does `main_TASK` split into training and validation data (during tuning) or into training and test data (during unbiased evaluation)?

If `tset==NULL`, then `tdmClassifyLoop` takes care of splitting `dset` into training and validation data: Three options are supported here, which are distinguished by the value of `opts$TST.kind`:

1. `"rand"` = **Random Subsampling**: Sample a fraction `opts$TST.valiFrac` from `dset` (the train-validation-data) and set it aside for validation. Use the rest for training, if `opts$TST.trnFrac` is `NULL`. If `opts$TST.trnFrac` is defined (and if it is $\leq 1 -$ `opts$TST.valiFrac`, otherwise error), then use only a random fraction `opts$TST.trnFrac` of the non-validation data from `dset` for training.
2. `"cv"` = **Cross Validation**: Split `dset` into `opts$TST.nfold` folds and use them for cross validation.
3. `"col"` = **User-Defined Column**: All records with a `0` in column `opts$TST.COL` are used for training, the rest for validation.

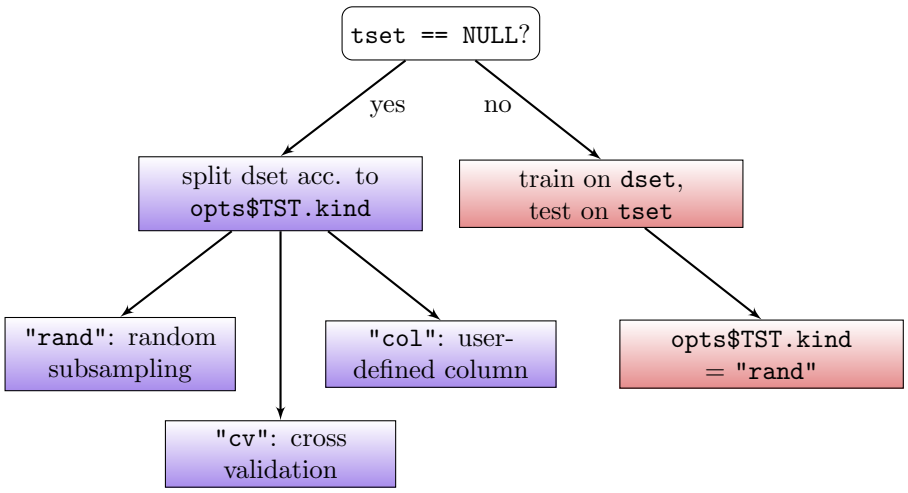

Figure 1: Modes of data splitting in `main_TASK`. Blue boxes: modes of splitting into training and validation set. Maroon boxes: splitting into training and test set.

The split into training and validation data is done in `tdmClassifyLoop`, i.e. for each call of `main_TASK`. This allows to construct different splits train/vali for each tuning step or each repeated run.

Note that the fractions `opts$TST.valiFrac` and `opts$TST.trnFrac` are relative to the number of rows in `dset`. `dset` may be the TrnVaSet defined above or the complete dataset.

`opts$TST.kind =="col"` in combination with `tdm$umode="TST"` above is normally NOT recommended (the same data are specified for test set and validation set). But it is o.k. in the special case of `opts$MOD.method =="RF"` or `=="MC.RF"` (Random Forest): Then the validation data are in fact never used, since RF uses its own validation measure with the OOB-error on the training data.

If `tset != NULL`, only `opts$TST.kind=="rand"` is allowed. Training data are taken from `dset`, by choosing a random subsample (fraction `opts$TST.trnFrac`).[6]

4.3 Examples

```
opts$READ.TrnFn=readDmc2010Trn
opts$READ.TstFn=readDmc2010Tst
opts$filename="dmc2010_train.txt"
opts$filetest="dmc2010_test.txt"
opts$TST.kind="col"
```

[6] If `opts$TST.trnFrac==NULL`, set it to 1 − `opts$TST.valiFrac`. Set `opts$TST.trnFrac=1` (and `opts$TST.valiFrac=0`), if you want to use all data from `dset` for training.

```
opts$TST.COL="TST"
opts$MOD.method="RF"
tdm$umode="TST"
```

Read the data prior to tuning, with train-set from `dmc2010_train.txt`, test set from `dmc2010_test.txt`. This is coded with `0` and `1` in column `TST` of the data frame `dset` . With `opts$TST.kind="col"` we specify that all `TST==0` data are used for training. The model `RF` (Random Forest) needs no validation data, since the performance measure is "OOB on the training set".

```
opts$filename="sonar.txt"
opts$TST.testFrac=0.15
opts$TST.kind="cv"
opts$TST.nfold=5
tdm$umode="SP_T"
```

Read the data prior to tuning from sonar.txt, split them by random subsampling: 15% into test set and 85% into train+validation set. This is coded with `0` and `1` in column "tdmSplit" of data frame dset. During tuning, the train+validation set is further split by cross validation with 5 folds (new split in each tuning step). The unbiased run uses all 85% train+validation data for training and reports the performance on the 15% test set data.

Details: `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).

Table 3: Overview of important variables in TDMR

`opts`	list with DM settings (used by `main_TASK` and its subfunctions). Parameter groups: `opts$READ.*` # reading the data `opts$TST.*` # training / validation / test set and resampling `opts$PRE.*` # preprocessing `opts$SRF.*` # sorted random forest (or similar other variable rankings) `opts$MOD.*` # genera model issues `opts$CLS.*` # classification issues `opts$RF.*` # Random Forest `opts$SVM.*` # Support Vector Machine `opts$GD.*` # graphic device issues See Appendix B or `?tdmOptsDefaultsSet` for a complete list of all elements in `opts`.
`dset`	preprocessed data set (used by `main_TASK` and its subfunctions)
`result`	list with results from Level 1, see Table 1
`finals`	see Table 2
`lastRes`	list with results from tdmClassify/tdmRegress, see Table 4
`envT`	environment, see Table 5
`tdm`	list with all options for controlling TDMR, see Appendix C or `?tdmDefaultsFill` for a complete list of all elements in `tdm`

5 TDMR Important Variables

5.1 Variable opts

opts is a long list with many parameters which control the behaviour of main_TASK, i.e. the behaviour of Level 1. To give this long list a better structure, the parameters are grouped with key words after "opts$" and before "." (see Table 3 above).

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

- opts$NRUN
- opts$VERBOSE

and others.

You should create opts with tdmOptsDefaultsSet() and specify in your application (i.e. main_TASK or controlDM) only those elements of opts which differ from these defaults. Or you enter main_TASK 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 these settings of the user).

The accessor function Opts(envT$result) returns the element envT$result$lastRes$opts.

- For "type safety", every object opts should be created as

  ```
  opts = tdmOptsDefaultsSet()
  ```

  and not with opts = list().
- If the list opts is extended by element X in the future, you need only to add a default specification of opts$X in function tdmOptsDefaultsSet, and all functions called from main_TASK will inherit this default behaviour.
- tdmOptsDefaultsSet calls finally the internal function tdmOptsDefaultsFill(opts) , and this fills in further defaults derived from actual settings of opts (e.g. opts$LOGFILE is an element which is derived from opts$filename as <opts$filename>.log).

5.2 TDMR RGain Concept

5.2.1 Classification

The **total gain** is defined as the sum of the pointwise product gainmat*confmat. Here confmat is the confusion matrix (actual vs. predicted cases) and gainmat is the gain associated with each possible outcome.[7]

The R_-elements (i.e. result$R_train and result$R_test, referred to as "RGain" in different places of this document) can contain different performance measures, depending on the value of opts$rgain.type:

[7] If there are for example different costs for different types of misclassification, the gain matrix can be defined with zeros on the diagonal and a negative gain "−cost" for each non-diagonal element (negative cost matrix).

Table 4: Elements of `lastRes`. The items last* are specific for the *last* model (the one built for the last response variable in the last run and the last fold)

`lastRes` list with results from `tdmClassify/tdmRegress`:	
In the case of classification, an object of class `tdmClass` with:	
opts	# with some settings perhaps adjusted
d_train	# training set + predicted class column(s)
d_test	# test set + predicted class column(s)
d_dis	# disregard set + predicted class column(s)
allEVAL	# data frame with evaluation measures, one row for each response # variable, the columns are explained in Sec. 5.3
lastCm*	# confusion matrix for * = train or test
lastModel	# the trained model (for last response variable)
lastPred	# name of prediction column
lastProbs	# a list with three probability matrices (row: records, col: classes) # v_train, v_test, v_dis if the model provides probabilities.
In the case of regression, an object of class `tdmRegre` with:	
opts	# with some settings perhaps adjusted
d_train	# training set + predicted regression column(s)
d_test	# test set + predicted regression column(s)
allRMAE	# data frame with rows = response.variables and columns according to Sec. 5.2.2 (RMAE = relative mean absolute error): # `$rmae.train`: RMAE on training set # `$theil.train`: Theil's U [RMAE] on training set # `$rmae.test`: RMAE on test set # `$theil.test`: Theil's U [RMAE] on test set
allRMSE	# data frame with rows = response.variables and columns according to Sec. 5.2.2 (RMSE = root mean square error): # `$rmse.train`: RMSE on training set # `$theil.train`: Theil's U [RMSE] on training set # `$rmse.test`: RMSE on test set # `$theil.test`: Theil's U [RMSE] on test set
lastModel	# the trained model (for last response variable)

- `"rgain"` or `NULL` [def.]: the relative gain in percent, i.e. the total gain actually achieved divided by the maximal achievable gain on the given data set,
- `"meanCA"` : mean class accuracy: For each class the accuracy (`1 − error rate`) 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(Specificity,Sensitivity).

For **binary classification** there are additional options for `opts$rgain.type`, three of them based on package `ROCR` (Sing et al., 2005):

- `"arROC"`: area under ROC curve (a number in [0,1]),
- `"arLIFT"`: area between lift curve and horizontal line 1.0,
- `"arPRE"`: area under precision-recall curve (a number in [0,1]).
- `"bYouden"`: balanced Youden index (a number in [0,1]),

The **balanced Youden index** B_{youden} is based on specificity and sensitivity, two measures based on the 2×2 confusion matrix:

$$B_{youden} = \min(\text{Sensitivity}, \text{Specificity}) \tag{1}$$

with

$$\text{Sensitivity} = \frac{TP}{TP+FN} \tag{2}$$

$$\text{Specificity} = \frac{TN}{TN+FP} \tag{3}$$

In each classification case, TDMR seeks to minimize $-$`result$R_train` , i.e. to *maximize* `result$R_train`.

5.2.2 Regression

For **regression**: The `R_`-elements (i.e. `result$R_train` and `result$R_test`, referred to as "RGain" in different places of this document) can contain different things, depending on the value of `opts$rgain.type` (with y_i = true response and p_i = predicted response) :

- `"rmae"` or `NULL` [def.]: the relative mean absolute error RMAE, i.e.

$$RMAE = \frac{\frac{1}{N}\sum_{i=1}^{N}|y_i - p_i|}{\frac{1}{N}\sum_{i=1}^{N}|y_i|}$$

,

- `"rmse"`, root mean square error:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - p_i)^2}$$

  ,

- `"made"`, mean absolute deviation:

$$MADE = \frac{1}{N}\sum_{i=1}^{N}|y_i - p_i|$$

In each regression case, TDMR seeks to *minimize* `result$R_train`.

In addition, the measure Theil's U is returned in `lastRes$allRMSE$theil.*`, which is in the case of RMSE:

$$\text{Theil's U} = \frac{RMSE}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - p_i^{(naive)})^2}},$$

where $p_i^{(naive)}$ is the prediction of a *naive prediction model*. This naive model is by default the mean of the response variable on the training data set (but other naive models could be used as well). The meaning of Theil's U is: If it is greater than 1, then the model is of no use, because it is beaten by the naive model. If it is smaller than 1, the model has some predictive power.

In the case of RMAE, Theil's U is defined similarly:

$$\text{Theil's U} = \frac{RMAE}{\frac{1}{N}\sum_{i=1}^{N}|y_i - p_i^{(naive)}|},$$

5.3 Classes `tdmClass` and `tdmRegre`

Function `tdmClassify` returns as result an object of class `tdmClass` and function `tdmRegress` returns an object of class `tdmRegre`. Both objects are lists. Within list `result` (Tab. 1), the element `lastRes` is an object of either `tdmClass` or `tdmRegre` (Tab. 4) .

Objects of class `tdmClass` (Tab. 4) contain a data frame `allEVAL`. The 9 evaluation measures in `allEVAL` are

`cerr.*`	misclassification error,
`gain.*`	total gain and
`rgain.*`	relative gain, i.e. total gain divided by max. achievable gain in `*`

where `* = [trn | tst | tst2 ]` stands for `[` training set `|` test set `|` test set with special treatment `]` and the special treatment is either `opts$test2.string = "no postproc"` or `"default cutoff"`.

Table 5: Elements of environment envT. The 3rd column shows which function adds the specified element to `envT`.

variable	remark	function
bst	data frame (see Appendix E)	tdmStartOther or spotTuner, lhdTuner
bstGrid	list with all bst data frames, bstGrid[[k]] retrieves the kth data frame	tdmBigLoop or populateEnvT
getBst (conf,tuner,n)	function returning from bstGrid the bst data frame for configuration file conf, tuning method tuner and experiment n	tdmBigLoop
res	data frame (see Appendix E)	tdmStart* or tdmBigLoop
resGrid	list with all res data frames, resGrid[[k]] retrieves the kth data frame	tdmBigLoop or populateEnvT
getRes (conf,tuner,n)	function returning from resGrid the res data frame for configuration file conf, tuning method tuner and experiment n	tdmBigLoop
result	list with results of tdm$mainFunc as called in the last unbiased evaluation, see Table 1	`unbiasedRun`
runList	a list of configuration names	tdmBigLoop
spotConfig	see package SPOT	tdmBigLoop
tdm	see Appendix C	tdmBigLoop
theFinals	data frame with one row for each res file, see Table 2	tdmBigLoop or populateEnvT
tunerVal	the value of `tdmDispatchTuner` (which can be a long list in case of SPOT)	tdmDispatchTuner

5.4 Environment envT

The environment envT is used for several purposes in TDMR

- to report results from a call to tdmBigLoop (Level 3) back to the user,
- to communicate information between different parts of TDMR,
- to pass necessary information to and back from the parallel slaves, see Sec. 6.2 *Environment envT for parallel mode*.

Environment envT is constructed in tdmEnvTMakeNew, with some elements filled in later by other functions. Table 5 shows the elements of envT.

envT is used to pass information back and forth between different functions of TDMR, where envT$sCList[[k]]$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. Note that the variable opts with various settings for the DM process is returned in several variables of envT:

envT$result$opts,
envT$result$lastRes$opts,
envT$tunerVal$opts,
envT$spotConfig$opts,
envT$sCList[[k]]$opts.

6 TDMR parallel computing concept

6.1 How to use parallel computing

TDMR supports parallel computing through the packages snow and package parallel. Parallelization of TDMR's level-3-tasks is very easy, you simply have to set tdm$parallelCPUs to a suitable value > 1. This will invoke the parSapply-mechanism of parallel.

Note that a certain parSapply will try to spawn always tdm$parallelCPUs processes, but if the last process(es) are less than this number, parSapply will wait for the slowest to complete before the next parSapply takes over. So it is a good idea to bundle as many processes as possible into one parSapply, if you want an even load distribution over time. But on the other hand, it has also advantages to send several tdmBigLoop's because every such call will have its own envT, which is saved on its own .RData file at the end of function tdmBigLoop and so the intermediate results are preserved, even if the parallel cluster should crash.

6.2 Environment envT for parallel mode

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.)

See Sec. 5.4 and Table 5 for more information on envT.

7 Variable-length vectors in TDMR classification

When running TDMR for classification, some possible tuning parameters need special treatment. These are (in the case of RF or similar learning algorithms):

- `sampsize`
- `cutoff`
- `classwt`

We explain the details in the following.

7.1 `sampsize`

The parameter `sampsize` in a call to randomForest can be either

a) a scalar, then it is the total sample size

b) a vector of length `n.class` = „number of levels in response.variable“, then it is the size of each strata (number of samples with that class level), so the sum of this vector is the overall sample size.

TDMR allows to tune the `sampsize` variables in either case a) or b), in case b) up to a limit of `n.class`=5. A ROI data frame can contain lines with SAMPSIZE1, SAMPSIZE2, SAMPSIZE3, SAMPSIZE5, SAMPSIZE5 which are mapped to `opts$RF.samp[i]`, i=1,...,5.

If *only* SAMPSIZE1 is present in ROI data frame, then `opts$RF.samp` is a scalar, which is case a) above. Otherwise, we have case b).

In more detail:

For classification:

- SAMPSIZEi in ROI will be mapped to `opts$RF.samp[i]`. If the user wants to tune just a scalar sampize, she defines only SAMPSIZE1 in ROI.
- After mapping, `opts$RF.samp` has to be a scalar or a vector of length `n.class`. That is, `controlDM` is responsible for setting all `opts$RF.samp[i]` that do not appear in ROI (because they shall not be tuned).
- Prior to training the model on data to.model, a call tdmModAdjustSampsizeC will check all this and will throw errors, if not fulfilled. In addition,

```
tdmModAdjustSampsizeC(opts$RF.samp,...)
```

will compare `opts$RF.samp[i]` with the number of records for each class level in the training set to.model and clip it, if necessary. The result is a vector `opts$RF.sampsize`, which is guaranteed to work in train.rf for a call `randomForest(...,sampsize,...)`.

- If importance check is enabled (SRF), then a similar call

```
tdmModAdjustSampsizeC(opts$SRF.samp,...)
```

  will be done before importance check. Currently, `opts$SRF.samp` will be only a scalar (if not set otherwise in `controlDM`). It is not (yet) in the set of tunable parameters.

For regression:

- very much the same, only tdmModAdjustSampsizeC is replaced by tdmModAdjustSampsizeR;
- this takes care of the fact, that for regression, `sampsize` can only be a scalar (or NULL).

7.2 `cutoff`

The parameter `cutoff` in a call to randomForest (for classification only) can be either

a) not present, then `cutoff[i] = 1/n.class` is the default, where `n.class` = „number of levels in response.variable“

b) a vector of length `n.class`, whose sum has to be exactly 1.

TDMR allows to tune the `cutoff` variables in b) up to a limit of n=`n.class`=5. A ROI data frame can contain lines with CUTOFF1, CUTOFF2, CUTOFF3, CUTOFF4, CUTOFF5 which are mapped to `opts$CLS.cutoff[i]`, i=1,...,5.

It is a bit tricky to ensure for c_i = CUTOFFi the constraint $\sum_{i=1}^{n} c_i = 1$.

This is because a tuning of any CUTOFFi tells the tuner to select a random value from [lower,upper] as specified in ROI, independent of the other CUTOFFk. Therefore a design point will almost always violate the sum constraint. Even if we map the violating design points to legal ones, the problem remains that many different design points are mapped to the same configuration.

How to cure? - The short story is: It is not wise to tune all $c_i, i = 1, ., n$. Instead: Set one $c_i = -1$ in `controlDM`. Specify positive values for the `n.class`-1 other c_i either in `controlDM` or in the ROI section of `controlSC`. This reduces the tuning complexity because at most `n.class`-1 cutoffs need to be tuned. Example:

```
opts$CLS.cutoff = c(0.1, -1, 0.5)
```

Then TDMR (with function tdmModAdjustCutoff) will take care to set the negative cutoff to „$1 - \sum$ (other cutoffs)“, i.e. $c_2 = 1 - 0.6 = 0.4$ in the example above.

In more detail:

- `opts$CLS.cutoff` is the cutoff for model training. `opts$SRF.cutoff` is the cutoff for the randomForest used during importance check.

- `controlDM` may or may not specify values for `opts$CLS.cutoff` and `opts$SRF.cutoff`. E.g.

```
opts$CLS.cutoff=c(0.1, 0.1, -1)
```

signaling that `opts$CLS.cutoff[3]` gets the remainder to 1. If it does not specify anything, the default

```
opts$CLS.cutoff=NULL
opts$SRF.cutoff=opts$CLS.cutoff
```

is taken (function tdmOptsDefaultsSet). If any cutoff is NULL, there will be no cutoff argument in the call to randomForest.

- Now the design point according to the ROI section of `controlSC` is mapped, e.g. with

```
CUTOFF1 = 0.243
CUTOFF2 = 0.115
```

we get `opts$CLS.cutoff=c(0.243, 0.115, -1)`.

Note: Any settings in the ROI section of `controlSC` will overwrite prior settings in `controlDM`. This means that the setting `opts$CLS.opt[1]=0.1` in `controlDM` is overwritten by the appearance of CUTOFF1 in the ROI section of `controlSC`

- The setting `opts$CLS.cutoff=c(0.243, 0.115, -1)` is passed on to TDMR and TDMR (with function `tdmModAdjustCutoff`) takes care to map the cutoff vector to valid cutoff vector (all $c_i > 0$ and $\sum c_i = 1$). It takes care of some special cases:
  - If the cutoff vector has length `n.class`-1, it adds a -1 at the end.
  - If exactly one cutoff is negative, it is set to „$1 - \sum$(other cutoffs)“. If more than one cutoff is negative, it throws an error.
  - If $\sum$(other cutoffs) ≥ 1, it scales all those elements to sum 0.9. Why 0.9? – Because then the left-over cutoff can get a positive value 0.1. A warning „sum ≥ 1“is issued. There is no problem if this warning occurs only for some design points, it can happen sometimes for certain ROI regions. But if it happens very often, the user may change the ROI, so that the left-over cutoff is not always 0.1.
  - It is a good idea to tune the smaller cutoffs and have the largest cutoff as left-over, in this case warnings will occur less often or never.

7.3 `classwt`

The parameter `classwt` in a call to randomForest (for classification only) can be either

a) not present, then all class levels get the same weight,

b) a vector of length `n.class`, where `n.class` = „number of levels in response.variable“.

TDMR allows to tune the `classwt` variables in case b) up to a limit of n=`n.class`=5. A ROI data frame can contain lines with CLASSWT1, CLASSWT2, CLASSWT3, CLASSWT5, CLASSWT5 which are mapped to `opts$CLS.CLASSWT[i]`, i=1,...,5.

Similar to `cutoff`, CLASSWTi tells the tuner to select a random value from [lower,upper] as specified in ROI, independent of the other CLASSWTk. Similar to `cutoff`, only the relative weight to the other CLASSWTi is important. Therefore, it is not wise to tune all CLASSWTi , i = 1,.,n. Instead: Set one CLASSWTi in`controlDM`. Specify positive values for the `n.class`−1 other CLASSWTi either in ROI section of `controlSC` or in `controlDM`. This reduces the tuning complexity because at most `n.class`−1 variables in `classwt` need to be tuned.

8 Example Usage

The usage of the TDMR workflow is fairly easy. We show several example lessons in the accompanying document TDMR-tutorial.pdf (Konen and Koch, 2012b).

9 Frequently Asked Questions (FAQ)

See the FAQ section in the accompanying document TDMR-tutorial.pdf (Konen and Koch, 2012b).

10 TDMR for Developers

This section contains more details about some aspects of TDMR. It can be skipped on first reading.

10.1 TDMR Tuner Concept

10.1.1 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 `demo03sonar.r`. The tuning results are in `envT$bst` and `envT$res` and in `envT$bstGrid` and `envT$resGrid` (see Appendix E and Table 5) .

10.1.2 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 (Hansen, 2006). Since comparison 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"`, `"cma_j"`): 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 reads the tuner configuration from `controlSC` 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.

10.2 How to integrate new machine learning algorithms

Assume you want to add a new algorithm named `ALGO`, similar to the existing algorithms RF or SVM.

- Add a new function `train.ALGO` in `tdmClassify.r` and/or `tdmRegress.r`.
- Add a new function `apply.ALGO` in `tdmClassify.r` and/or `tdmRegress.r`.
- Add a new choice `"ALGO"` to `opts$MOD.method`.
- Add the tunable parameters of `ALGO` as `opts$ALGO.*` to the list `opts`, see `tdmOptsDefaults.r`.
- Add for each parameter `opts$ALGO.*` a suitable mapping in `tdmMapDesign.csv`.

10.3 Details on TDMR parallel computing concept

We parallelize the `tdmDispatchTuner`-calls which are currently inside the 3-fold loop

```
(sCList,tdm\$tuneMethod,tdm\$nExperim).
```

Therefore, these loops are written as `sapply` commands, which can be transformed to `parSapply`. Additional remarks:

- We have in `tdmBigLoop` only one parallelization mode (parallel over experiments, tuners and configurations). 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 `parSapply`-calls.
- When does `parSapply` return? – The manual says that `parSapply` first hands out `nCPU` jobs to the CPUs, then waits for all (!) jobs to return and then hands out another `nCPU` jobs until all jobs are finished. `parSapply` returns when the last job is finished. Therefore it is not clear what happens with nested `parSapply`-calls and we make our design in such a way that no such nested calls appear.

10.4 TDMR Design Mappping Concept

Each variable appearing in the ROI section of `controlSC` (and thus in .des file) has to be mapped on its corresponding value in list opts. This is done in the file `tdmMapDesign.csv` (see Appendix A):

roiValue	optsValue	isInt
MTRY	opts$RF.mtry	1
...		

If a variable is defined with `isInt=1`, it is rounded in `opts$...` to the next integer, even if it is non-integer in the design file. The base file `tdmMapDesign.csv` is read from `<packageDir> = .find.package("TDMR" )`.[8] If in the `<dir_of_main_task> = dirname(tdm$mainFile)` an additional file `userMapDesign.csv` exists, it is additionally read and added to the relevant

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

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.

10.4.1 How to add a new tuning variable

There are several options:

- (user) add a new line to `userMapDesign.csv` or
- (developer) add a new line to `tdmMapDesign.csv` or
- (optional, for developer) add a line to `tdmOptsDefaultsSet()`, if it is a new variable `opts$...` and if all existing and further tasks should have a default setting for this variable

Details We have in tdmMapDesign.r 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 tdmMapDesLoad, tdmMapDesApply 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.

10.5 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 seedSPOT in the configuration control (`controlSC, ctrlSC`). You may set seedSPOT=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 `seedSPOT=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$seedSPOT) 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= NULL; MOD.SEED=any value` is a fourth possibility, but it has –as far as I can see – no practical application).

What happens if `opts$*.SEED` is `NULL`? – In this case, TDMR will execute

```
opts$TST.SEED = tdmRandomSeed()
```

in `tdmClassify` before each usage of `opts$*.SEED. (* = MOD, TST)`. Here, `tdmRandomSeed` is a function which returns a different integer seed each time it is called. This is even true, if it is called multiple times within the same second (where a function like `Sys.time()` would return the same number). This can easily happen in parallel execution mode, where processes on different slaves usually will be started in the same second. A second aspect of random variability: We usually want each run through `main_TASK` (loop over `i in 1:opts$NRUN` in `tdmClassifyLoop`) and each repeat during tuning (loop over `r in 1:des$REPEATS[k]` in `tdmStart*`) to explore different random regions, even in the case where all seed settings (`seedSPOT, opts$TST.SEED` and `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.

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.

New Jan'2012: When `opts$*.SEED (* = MOD, TST)` is the string `"algSeed"`, then TDMR will set the relevant seed to `opts$ALG.SEED`, which is the seed `spotConfig$alg.seed+r` from SPOT, where `spotConfig$alg.seed` is set by the user (reproducibility) and `r` is the repeat-number for the design point in question (ensure that each repeat gets another seed to explore the random variability).

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.

10.6 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$GD.PNGDIR
- "win" : each plot goes into a new window (X11())
- "rstudio": each plot goes to the RStudio plot window
- "non" : all plots are suppressed (former opts$DO.GRAPHICS=F)

`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 `unbiaseRun_*`, if you want the plots from the last round of `unbiasedRun_*`.

Table 6: Tuners availabe in TDMR

`tdm$tuneMethod`	Description
`spot`	Sequential Parameter Optimization Toolbox, Bartz-Beielstein (2010)
`lhd`	Latin Hypercube Design, McKay et al. (1979) (truncated SPOT, all budget for the initial step)
`cmaes`	Covariance Matrix Adaption ES, Hansen (2006) (R-version, package cmaes)
`cma_j`	Covariance Matrix Adaption ES, Hansen (2006) (Java-version, interfaced to R via package rCMA)
`powell`	Powell's method, Powell (1970) (direct & local search)
`bfgs`	Broyden, Fletcher, Goldfarb and Shannon method, Shanno (1985) (direct & local search)

Table 7: Graphic Utility Functions

	`opts$GRAPHDEV`				
utility function	`pdf`	`png`	`win`	`rstudio`	`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())	clear all plots	-

11 Summary

This report has shown how to use TDMR, the Tuned Data Mining framework in R. The examples shown should make the reader familiar with the concepts and the workflow phases of TDMR. More examples are shown in the companion document TDMR-tutorial.pdf (Konen and Koch, 2012b). They are deliberately made with fairly small datasets in order to facilitate quick reproducability. For results on larger datasets the reader is referred to Konen et al. (2010, 2011).

A Appendix A: tdmMapDesign.csv

```
# For each variable which appears in ROI section of controlSC:
#    set its counterpart in list opts.
# For each variable not appearing:
#    leave its optsValue at its default from controlDM.
  roiValue;    optsValue;           isInt
  PCA.npc;     opts$PRE.PCA.npc;    1
  SFA.npc;     opts$PRE.SFA.npc;    1
  SFA.PPRANGE;opts$PRE.SFA.PPRANGE;1
  SFA.ODIM;    opts$PRE.SFA.ODIM;   1
  NCOPIES;     opts$ncopies;        1
  TRNFRAC;     opts$TST.trnFrac;    0
  XPERC;       opts$SRF.XPerc;      0
  NDROP;       opts$SRF.ndrop;      1
  MTRY;        opts$RF.mtry;        1
  NODESIZE;    opts$RF.nodesize;    1
  NTREE;       opts$RF.ntree;       1
  SVMkernel;   opts$SVM.kernel;     1
  SVMdegree;   opts$SVM.degree;     1
  ADAcoeflrn; opts$ADA.coeflearn; 1
  ADAmfinal;   opts$ADA.mfinal;     1
  EPSILON;     opts$SVM.epsilon;    0
  GAMMA;       opts$SVM.gamma;      0
  TOLERANCE;   opts$SVM.tolerance; 0
  SIGMA;       opts$SVM.sigma;      0
  COST;        opts$SVM.cost;       0
  COEF0;       opts$SVM.coef0;      0
  SAMPSIZE1;   opts$RF.samp[1];     1
  SAMPSIZE2;   opts$RF.samp[2];     1
  SAMPSIZE3;   opts$RF.samp[3];     1
  SAMPSIZE4;   opts$RF.samp[4];     1
  SAMPSIZE5;   opts$RF.samp[5];     1
  CLASSWT1;    opts$CLS.CLASSWT[1];0
  CLASSWT2;    opts$CLS.CLASSWT[2];0
  CLASSWT3;    opts$CLS.CLASSWT[3];0
  CLASSWT4;    opts$CLS.CLASSWT[4];0
  CLASSWT5;    opts$CLS.CLASSWT[5];0
#
  CUTOFF1;     opts$CLS.cutoff[1]; 0
  CUTOFF2;     opts$CLS.cutoff[2]; 0
  CUTOFF3;     opts$CLS.cutoff[3]; 0
  CUTOFF4;     opts$CLS.cutoff[4]; 0
  CUTOFF5;     opts$CLS.cutoff[5]; 0
```

B Appendix B: List `opts`

List `opts` contains all options relevant for controlling a DM task.

This table – with proper hyperlinks – is as well obtained by typing `?tdmOptsDefaultsSet` within an R session.

[`path = dir(tdm$mainFile)`, if `tdm$mainFile` is defined, else the current dir. The elements `dir.txt`, `dir.data`, `dir.output` are relative to `path`.]

Element	Description
dir.txt	[`data`] where to find .txt/.csv files
dir.data	[`data`] where to find other data files, including .Rdata
dir.output	[`Output`] where to put output files
filename	[`"default.txt"`] the task data
filetest	[NULL] the test data, only relevant for READ.TstFn!=NULL
data.title	[`"Default Data"`] title for plots
READ.TXT	[TRUE] =TRUE: read data from .csv and save as .Rdata, =FALSE: read from .Rdata
READ.NROW	[−1] read this amount of rows or −1 for 'read all rows'
READ.TrnFn	function to be passed into `tdmReadDataset`. Signature: `function(opts)` returning a data frame. It reads the train-validation data.
READ.TstFn	[NULL] function to be passed into `tdmReadDataset`. Signature: `function(opts)` returning a data frame. It reads a separate test data file. If NULL, this reading step is skipped.
READ.INI	[TRUE] read the task data initially, i.e. prior to tuning, using `tdmReadDataset`. If =FALSE, the data are read anew in each pass through `main_TASK`, i.e. in each tuning step (deprecated).
TST.kind	[`"rand"`] How to split the data into train and validation set. One of the choices from `[ "cv" | "rand" | "col" ]` for `[` cross validadation `|` random sample `|` column with train/test flag `]`, see Sec. 4.2 and `tdmModCreateCVindex` for details
TST.COL	[`"TST.COL"`] name of column with train/test/disregard-flag
TST.NFOLD	[3] number of CV-folds (only for `TST.kind=="cv"`)
TST.valiFrac	[0.1] set this fraction of data aside for validation (only for `TST.kind=="rand"`)
TST.testFrac	[0.1] set prior to tuning this fraction of data aside for testing (if `tdm$umode=="SP_T"` and `opts$READ.INI==TRUE`) or set this fraction of data aside for testing after tuning (if `tdm$umode=="RSUB"` or `=="CV"`)
TST.SEED	[NULL] a seed for the random test set selection (tdmRandomSeed) and random validation set selection. (tdmClassifyLoop). If NULL, use tdmRandomSeed.
NRUN	[2] how many runs with different train & test samples – or – how many CV-runs, if `opts$TST.kind="cv"`

PRE.PCA	`["none"]` PCA preprocessing: `[ "(default)"none" | "linear" ]` for `[` don't `|` normal pca (prcomp) `]`
PRE.PCA.REPLACE	[TRUE] =TRUE: replace with the PCA columns the original numerical columns, =FALSE: add the PCA columns
PRE.PCA.npc	[0] if > 0: add monomials of degree 2 for the first PRE.PCA.npc columns (PCs)
PRE.SFA	`["none"]` SFA preprocessing (see package rSFA: `[ "none" | "2nd" ]` for `[` don't `|` normal SFA with 2nd degree expansion `]`
PRE.SFA.REPLACE	[FALSE] =TRUE: replace the original numerical columns with the SFA columns; =FALSE: add the SFA columns
PRE.SFA.npc	[0] if > 0: add monomials of degree 2 for the first PRE.SFA.npc columns
PRE.SFA.PPRANGE	[11] number of inputs after SFA preprocessing, only those inputs enter into SFA expansion
PRE.SFA.ODIM	[5] number of SFA output dimensions (slowest signals) to return
PRE.SFA.doPB	[TRUE] =TRUE/FALSE: do / don't do parametric bootstrap for SFA in case of marginal training data
PRE.SFA.fctPB	[sfaPBootstrap] the function to call in case of parametric bootstrap, see sfaPBootstrap in package rSFA for its interface description
PRE.Xpgroup	[0.99] bind the fraction 1− PRE.Xpgroup in column OTHER (see tdmPreGroupLevels)
PRE.MaxLevel	[32] if there are N cases, bind the $N - 32 + 1$ least frequent cases in column OTHER (see tdmPreGroupLevels)
SRF.kind	`["xperc" (default) |"ndrop" |"nkeep" |"none" ]` the method used for feature selection, see `tdmModSortedRFimport`
SRF.ndrop	[0] how many variables to drop (if `SRF.kind=="ndrop"`)
SRF.XPerc	[0.95] if ≥ 0, keep that importance percentage, starting with the most important variables (if `SRF.kind=="xperc"`)
SRF.calc	[TRUE] =TRUE: calculate importance & save on SRF.file, =FALSE: load from SRF.file (`SRF.file = Output/<filename>.SRF.<response.variable>.Rdata)`
SRF.ntree	[50] number of RF trees
SRF.samp	sampsize for RF in importance estimation. See RF.samp for further info on sampsize.
SRF.verbose	[2]
SRF.maxS	[40] how many variables to show in plot
SRF.minlsi	[1] a lower bound for the length of `SRF$input.variables`
SRF.method	`["RFimp"]`
SRF.scale	[TRUE] option 'scale' for call importance() in tdmModSortedRFimport
MOD.SEED	[NULL] a seed for the random model initialization (if model is non-deterministic). If NULL, use tdmRandomSeed.
MOD.method	`["RF" (default) |"MC.RF" |"SVM" |"NB" ]` use `[` RF `|` MetaCost-RF `|` SVM `|` Naive Bayes `]` in `tdmClassify`

	`["RF" (default) |"SVM" |"LM" ]` use `[` RF `|` SVM `|` linear model `]` in `tdmRegress`
RF.ntree	[500]
RF.samp	[1000] sampsize for RF in model training. If RF.samp is a scalar, then it specifies the total size of the sample. For classification, it can also be a vector of length n.class (= number of levels in response variable), then it specifies the size of each strata. The sum of the vector is the total sample size.
RF.mtry	[NULL]
RF.nodesize	[1]
RF.OOB	[TRUE] if =TRUE, return OOB-training set error as tuning measure; if =FALSE, return validation set error
RF.p.all	[FALSE]
SVM.cost	[1.0]
SVM.C	[1] needed only for regression
SVM.epsilon	[0.005] needed only for regression
SVM.gamma	[0.005]
SVM.tolerance	[0.008]
ADA.coeflearn	[1] =1: "Breiman", =2: "Freund", =3: "Zhu" as value for boosting(...,coeflearn,...) (AdaBoost)
ADA.mfinal	[10] number of trees in AdaBoost = mfinal boosting(...,mfinal,...)
ADA.rpart.minsplit	[20] minimum number of observations in a node in order for a split to be attempted
CLS.cutoff	[NULL] vote fractions for the classes (vector of length n.class = number of levels in response variable). The class i with maximum ratio (% votes)/CLS.cutoff[i] wins. If NULL, then each class gets the cutoff 1/n.class (i.e. majority vote wins). The smaller CLS.cutoff[i], the more likely class i will win.
CLS.CLASSWT	[NULL] class weights for the n.class classes, e.g. c(A=10,B=20) for a 2-class problem with classes A and B (the higher, the more costly is a misclassification of that real class). It should be a named vector with the same length and names as the levels of the response variable. If no names are given, the levels of the response variables in lexicographical order will be attached in `tdmClassify`. CLS.CLASSWT=NULL for no weights.
CLS.gainmat	[NULL] (n.class x n.class) gain matrix. If NULL, CLS.gainmat will be set to unit matrix in `tdmClassify`.
rgain.type	`["rgain" (default) |"meanCA" |"minCA" ]` in case classification: The measure Rgain returned from tdmClassifyLoop in `result$R_*` is `[` relative gain (i.e. gain/gainmax) `|` mean class accuracy `|` minimum class accuracy `]` (see Sec. 5.2.1). The goal is to *maximize* Rgain. For binary classification there are the additional measures `["arROC"	"arLIFT"	"arPRE"]`, see `tdmModConfmat`.

	For regression, the goal is to *minimize* `result$R_*` returned from tdmRegress. In this case, possible values are rgain.type = `["rmae" (default) |"rmse"	"made"]` which stands for `[` relative mean absolute error `|` root mean squared error `|` mean absolute deviation `]` (see Sec. 5.2.2).
ncopies	[0] if > 0, activate tdmParaBootstrap in `tdmClassify`
fct.postproc	[NULL] name of a function with signature (pred, dframe, opts) where pred is the prediction of the model on the data frame dframe and opts is this list. This function may do some postprocessing on pred and it returns a (potentially modified) pred. This function will be called in `tdmClassify` if it is not NULL.
GD.DEVICE	[`"win"`] =`"win"`: all graphics to (several) windows (`X11`) =`"rstudio"`: all graphics to RStudio plot window =`"pdf"`: all graphics to one multi-page PDF =`"png"`: all graphics in separate PNG files in `opts$GD.PNGDIR` =`"non"`: no graphics at all This affects TDMR graphics, not SPOT (or other tuner) graphics
GD.RESTART	[TRUE] =TRUE: restart the graphics device (i.e. close all 'old' windows or re-open multi-page pdf) in each call to tdmClassify or tdmRegress, resp. =FALSE: leave all windows open (suitable for calls from SPOT) or write more pages in same pdf.
GD.CLOSE	[TRUE] =TRUE: close graphics device `"png"`, `"pdf"` at the end of `main_*.r` (suitable for `main_*.r` solo) or =FALSE: do not close (suitable for call from `tdmStartSpot`, where all windows should remain open)
APPLY_TIME	[FALSE]
VERBOSE	[2] =2: print much output, =1: less, =0: none

Additional settings from `tdmOptsDefaultsFill(opts)`, which depend on the already def'd elements of opts: [`*` is the stripped part of `opts$filename` (w/o suffix).]

Element	Description
PDFFILE	[`"*_pic.pdf"`] file for multipage graphics in case `opts$GD.DEVICE="pdf"`
GD.PNGDIR	[`"PNG*"`] directory for `.png` files in case `opts$GD.DEVICE="png"`
LOGFILE	[`"*.log"`] where to log the output
EVALFILE	[`"*_eval.csv"`] file with evaluation results allEVAL
SRF.samp	sample size for SRF, derived from `opts$RF.samp`
SRF.cutoff	[`opts$CLS.cutoff`] cutoff used during SRF modeling
rgain.string	one out of `c("RGain","MeanCA","MinCA","RMAE","RMSE")`, depending on `opts$rgain.type`

C Appendix C: List `tdm`

List `tdm` contains all options relevant for controlling TDMR.

This table – with proper hyperlinks – is as well obtained by typing `?tdmDefaultsFill` within an R session.

Element	Description
mainFile	[NULL] if not NULL, source this file from the current dir. It should contain the definition of `tdm$mainFunc`.
mainFunc	`sub(".r","",basename(tdm$mainFile),fixed=TRUE)`, if `tdm$mainFile` is set and `tdm$mainFunc` is NULL, else `"mainFunc"`. This is the name of the function called by `tdmStartSpot` and `unbiasedRun`
CMA.propertyFile	[NULL] (only for CMA-ES Java tuner) see `cma_jTuner`.
CMA.populationSize	[NULL] (only for CMA-ES Java tuner) see `cma_jTuner`.
filenameEnvT	[NULL] filename where `tdmBigLoop` will save a small version of environment `envT`. If NULL, save `envT` to `sub(".conf",".RData",tdm$runList[1])`.
nExperim	[1]
nfold	[10] number of CV-folds for unbiased runs (only for `umode="CV"`)
nrun	[5] number of unbiased runs
optsVerbosity	[0] the verbosity for the unbiased runs
parallelCPUs	[1] = 1: sequential, > 1: parallel execution with this many CPUs (package parallel)
parallelFuncs	[NULL] in case `tdm$parallelCPUs`> 1: a string vector with functions which are clusterExport'ed in addition to `tdm$mainFunc`.
path	[NULL] where to save/load `envT`. If NULL, `path` is set to the actual working directory at the time when `tdmEnvTMakeNew` is executed
runList	a list of configuration names
stratified	[NULL] see `tdmReadAndSplit`
tdmPath	[NULL] from where to source the R sources. If NULL load library TDMR instead.
test2.string	`["default cutoff"]`
theSpotPath	[NA] use SPOT's package version
timeMode	[1] 1: proc time, 2: system time, 3: elapsed time (columns `Time.TST` and `Time.TRN` in `envT$theFinals`, see Table 2)
tstCol	`["TST"]` `opts$TST.COL` for unbiased runs (only for `umode="TST"`)
tuneMethod	`["spot"]` other choices: `"cmaes"`, `"bfgs"`, ..., see `tdmDispatchTuner`
U.saveModel	[FALSE] if TRUE, save the last model, which is trained in `unbiasedRun`, onto `filenameEnvT`
umode	`["RSUB"]`, one out of [`"RSUB" | "CV" | "TST" | "SP_T"`], see `unbiasedRun`
unbiasedFunc	`["unbiasedRun"]` name of function to call for unbiased evaluation

withParams	[TRUE] include the columns with tuned parameters in final results
TST.trnFrac	[NULL] train set fraction (of all train-vali data), *overwrites* `opts$TST.trnFrac` if not NULL.
TST.valiFrac	[NULL] validation set fraction (of all train-vali data), *overwrites* `opts$TST.valiFrac` if not NULL.
TST.testFrac	[0.2] test set fraction (of *all* data) for unbiased runs (only for `umode="RSUB"` or `="SP_T"`)

Note

The settings `tdm$TST.trnFrac` and `tdm$TST.valiFrac` allow to set programmatically certain values for `opts$TST.trnFrac` and `opts$TST.valiFrac` *after* `opts` has been constructed (e. g. via `controlDM`). So use `tdm$TST.trnFrac` and `tdm$TST.valiFrac` with CAUTION!

For `tdm$timeMode`, the 'user time' is the CPU time charged for the execution of user instructions of the calling process. The 'system time' is the CPU time charged for execution by the system on behalf of the calling process. The 'elapsed time' is the 'real' (wall-clock) time since the process was started.

D Appendix D: List `ctrlSC`

List `ctrlSC` contains all control options for SPOT or other tuners.

This table – with proper hyperlinks – is as well obtained by typing `?defaultSC` within an R session.

Element	Description
alg.roi	["NEEDS_TO_BE_SET"] a data frame with columns `lower`, `upper`, `type`, `row.names`, each a vector with as many entries as there are parameter to be tuned
opts	["NEEDS_TO_BE_SET"] the list `opts` of controls for DM (see App. B or type `?tdmOptsDefaultsSet` in `R`) is internally attached to `ctrlSC` in order to transport these controls to the DM task to be tuned.
sCName	["NEEDS_TO_BE_SET.conf"] a string ending on `.conf`, the configuration name. `envT` will be written to file `scName.RData`.
alg.resultColumn	["Y"] column name containing results
funEvals	[20] spot's `funEvals`, the budget of function evaluations
OCBA	[FALSE] spot's `OCBA`
seedSPOT	[1] seed for the random number generator used for SPOT
plots	[FALSE] TRUE: make a line plot showing progress
design	[designLHD] spot's `design`, a function that creates the initial design of experiment
designControl.size	[10] number of initial design points
designControl.replicates	[2] number of initial repeats
seq.merge.func	[mean] how to merge `Y` over replicates: `mean` or `min`
replicates	[2] number of repeats for the same model design point
noise	[TRUE] whether the object function has noise or not (necessary if `replicates > 1`)
model	[buildKriging] spot's `model`, a function that builds the surrogate model
optimizer	[optimLHD] spot's `optimizer`, the optimizer to use when optimizing on `model`
optimizerControl.funEvals	[100] optimizer budget
optimizerControl.retries	[2] optimizer retries

Correspondences between spotConfig (SPOT 1.x) and control (SPOT 2.x)

spotConfig (SPOT 1.x)	control (SPOT 2.x)	remarks
auto.loop.nevals	funEvals	
auto.loop.steps	-	
alg.roi$type	types	
["FLOAT","INT"]	["numeric","integer","factor"]	see also designControl$types
init.design.func	design	

`[string]`	`[function]`	
init.design.size	designControl$size	
init.design.retries	designControl$retries	
init.design.repeats	designControl$replicates	
seq.design.maxRepeats	replicates	see also designControl$replicates
seq.predictionModel.func	model	
`[string]`	`[function]`	
seq.design.size	optimizerControl$funEval	how many optimizer evals
seq.design.retries	optimizerControl$retries	
spot.ocba	OCBA	
spot.seed	seedSPOT	
io.verbosity	?	

E Appendix E: Data frames `envT$res` and `envT$bst`

Data frame `envT$res` contains results from each run of tuner SPOT or other tuners producing equivalent output. Data frame `res` has the following columns:

- `Y`: the performance (-Rgain for classification, RMAE for regression) on the validation data with a specific configuration of the tuning variables (a design point), a value to be minimized.
- columns between `Y` and `SEED`: the values of the configuration (i.e. the design point in parameter space)
- `SEED`: seed for random number generator
- `STEP`: SPOT performs its tuning in several steps. The first step is the initial design (step 0 in the example), where 10 design points are drawn via LHS (latin hypercube design).
- `CONFIG`: the configuration number (design point number)
- `REP`: how many repeats were done for a `CONFIG`.

```
> envT$res
           Y   CUTOFF1  CLASSWT2     XPERC SEED STEP CONFIG REP
1  -80.66667 0.6160287  8.143644 0.9500583 1236    0      1   1
2  -77.33333 0.3870275  9.631080 0.9114447 1236    0      2   1
3  -71.33333 0.2636683 13.205968 0.9399057 1236    0      3   1
4  -76.66667 0.5369056 14.286117 0.9857516 1236    0      4   1
5  -69.33333 0.7396785 10.203108 0.9723459 1236    0      5   1
6  -70.66667 0.3264214  7.332868 0.9660157 1236    0      6   1
7  -69.33333 0.1994577  5.100756 0.9212448 1236    0      7   1
8  -54.00000 0.1001375  6.198328 0.9995300 1236    0      8   1
9  -80.00000 0.5198881 12.832352 0.9488441 1236    0      9   1
10 -79.33333 0.6618317 11.971008 0.9080284 1236    0     10   1
11 -76.00000 0.6160287  8.143644 0.9500583 1237    1      1   1
12 -78.66667 0.5378022 12.584936 0.9530634 1236    1     11   1
13 -78.66667 0.5378022 12.584936 0.9530634 1237    1     11   2
14 -80.66667 0.4854926 12.055882 0.9491663 1236    1     12   1
15 -79.33333 0.4854926 12.055882 0.9491663 1237    1     12   2
16 -73.33333 0.6527187 12.708480 0.9063501 1236    1     13   1
17 -76.66667 0.6527187 12.708480 0.9063501 1237    1     13   2
18 -82.66667 0.5198881 12.832352 0.9488441 1237    2      9   1
19 -79.33333 0.4602900 11.756751 0.9545721 1236    2     14   1
20 -78.00000 0.4602900 11.756751 0.9545721 1237    2     14   2
21 -80.66667 0.6530202 12.215844 0.9574122 1236    2     15   1
22 -80.00000 0.6530202 12.215844 0.9574122 1237    2     15   2
23 -78.00000 0.5514645 12.624498 0.9086569 1236    2     16   1
24 -80.00000 0.5514645 12.624498 0.9086569 1237    2     16   2
25 -77.33333 0.4912961 11.326311 0.9611196 1236    3     17   1
```

```
26 -84.66667 0.4912961 11.326311 0.9611196 1237    3     17   2
27 -82.00000 0.5236395 11.964197 0.9307988 1236    3     18   1
28 -81.33333 0.5236395 11.964197 0.9307988 1237    3     18   2
29 -81.33333 0.4928852 13.998331 0.9564483 1236    3     19   1
30 -82.00000 0.4928852 13.998331 0.9564483 1237    3     19   2
```

Let's explain the `res` example above (which is truncated after STEP=3): In the first step (STEP=0) 10 configurations (design points) are generated by LHS and tested on the data mining task with SEED=1236. It turns out that CONFIG=4 has the lowest Y=-86.00.

In the next step (STEP=1), three new configurations 11,12,13 are generated (using a metamodel built upon the previous step) and all of them are evaluated with two repeats (SEED=1236,1237). To make a fair comparison, CONFIG=4 is evaluated a second time (SEED=1237). In all cases, the configurations are judged by the **average** Y of the two repeats, which are in this case

CONFIG	<Y>
4	-84.00
11	-81.00
12	-79.00
13	-81.33

This means that CONFIG=4 survives STEP=1 as the best configuration. Now the next step (STEP=2) starts, where three new configurations 14,15,16 are evaluated for two repeats. In this case CONFIG=14 gets an even better average Y = 84.66.

The information which configuration is the best one in each step of the tuning algorithm is contained as a summary in data frame `envT$bst`. Most columns have exactly the same meaning as in data frame `envT$res`, with the addition of the new column COUNT which gives the number of repeats being executed for CONFIG at the beginning of STEP.

```
> envT$bst
             Y   CUTOFF1 CLASSWT2     XPERC COUNT CONFIG STEP
4   -86.00000 0.5369056 14.28612 0.9857516     1      4    1
41  -84.00000 0.5369056 14.28612 0.9857516     2      4    2
14  -84.66667 0.5364093 14.43284 0.9608062     2     14    3
141 -84.66667 0.5364093 14.43284 0.9608062     2     14    4
142 -84.66667 0.5364093 14.43284 0.9608062     2     14    5
143 -84.66667 0.5364093 14.43284 0.9608062     2     14    6
144 -84.66667 0.5364093 14.43284 0.9608062     2     14    7
145 -84.66667 0.5364093 14.43284 0.9608062     2     14    8
```

We see that CONFIG=4 is with Y=-86.00 the best configuration at the beginning of STEP=1. At the beginning of STEP=2 this is reduced to Y=-84.00 due to the 2nd repeat, but CONFIG=4 is still the best. At the beginning of STEP=3 however, CONFIG=14 takes over with Y=-84.66 (average of 2 repeats) and it remains the best configuration until the end.

Note: Although the aim of the tuner is to minimize Y, we might sometimes see an increase in Y in the sequence of rows in `envT$bst`. This can be the case if we go from a lower COUNT to

a higher COUNT. The reason is that at COUNT=1 we may have a 'lucky' validation set which produces a small Rgain for a certain CONFIG. If we repeat the run with another seed (another validation set), the same CONFIG may result in a higher Rgain, so that the 'lucky' Y for that CONFIG cannot be confirmed at COUNT=2. The higher COUNT has the statistically more sound value for Y.

References

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

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

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

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

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

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

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

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

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

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

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

McKay, M. D., Beckman, R. J., and Conover, W. J. (1979). A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245.

Powell, M. (1970). *A new algorithm for unconstrained optimization*. United Kingdom Atomic Energy Authority.

Shanno, D. (1985). On Broyden-Fletcher-Goldfarb-Shanno method. *Journal of Optimization Theory and Applications*, 46(1):87–94.

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

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