Package ‘liquidSVM’

July 19, 2017

Type Package

Title A Fast and Versatile SVM Package

Version 1.2.1

Date 2017-07-19

Author Ingo Steinwart, Philipp Thomann

Copyright Ingo Steinwart, Philipp Thomann, Mohammad Farooq

Maintainer Philipp Thomann <philipp.thomann@mathematik.uni-stuttgart.de>

Description Support vector machines (SVMs) and related kernel-based learning algorithms are a well-known class of machine learning algorithms, for non-parametric classification and regression.
liquidSVM is an implementation of SVMs whose key features are:
fully integrated hyper-parameter selection,
extreme speed on both small and large data sets,
full flexibility for experts, and
inclusion of a variety of different learning scenarios:
multi-class classification, ROC, and Neyman-Pearson learning, and
least-squares, quantile, and expectile regression.

URL http://www.isa.uni-stuttgart.de/software/

License AGPL-3

Depends R (>= 2.12.0), methods

Suggests knitr, rmarkdown, deldir, testthat

Enhances mlr, ParamHelpers

VignetteBuilder knitr

RoxygenNote 6.0.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2017-07-19 21:00:12 UTC

R topics documented:

`liquidSVM-package` *liquidSVM for R*

Description

Support vector machines (SVMs) and related kernel-based learning algorithms are a well-known class of machine learning algorithms, for non-parametric classification and regression. **liquidSVM** is an implementation of SVMs whose key features are:

- fully integrated hyper-parameter selection,
- extreme speed on both small and large data sets,

- full flexibility for experts, and
- inclusion of a variety of different learning scenarios:
  - multi-class classification, ROC, and Neyman-Pearson learning, and
  - least-squares, quantile, and expectile regression

Further information is available in the following vignettes:

`demo`	liquidSVM Demo (source, pdf)
`documentation`	liquidSVM Documentation (source, pdf)

Details

In **liquidSVM** an application cycle is divided into a training phase, in which various SVM models are created and validated, a selection phase, in which the SVM models that best satisfy a certain criterion are selected, and a test phase, in which the selected models are applied to test data. These three phases are based upon several components, which can be freely combined using different components: solvers, hyper-parameter selection, working sets. All of these can be configured (see Configuration) a

For instance multi-class classification with k labels has to be delegated to several binary classifications called *tasks* either using all-vs-all ($k(k-1)/2$ tasks on the corresponding subsets) or one-vs-all (k tasks on the full data set). Every task can be split into *cells* in order to handle larger data sets (for example > 10000 samples). Now for every task and every cell, several *folds* are created to enable cross-validated hyper-parameter selection.

The following learning scenarios can be used out of the box:

- `mcSVM` binary and multi-class classification
- `lsSVM` least squares regression
- `nplSVM` Neyman-Pearson learning to classify with a specified rate on one type of error
- `rocSVM` Receivert Operating Characteristic (ROC) curve to solve multiple weighted binary classification problems.
- `qtSVM` quantile regression
- `exSVM` expectile regression
- `bsSVM` bootstrapping

To calculate kernel matrices as used by the SVM we also provide for convenience the function `kern`.

liquidSVM can benefit heavily from native compilation, hence we recommend to (re-)install it using the information provided in the installation section of the documentation vignette.

Known issues

Interruption (Ctrl-C) of running train/select/test phases is honored, but can leave the C++ library in an inconsistent state, so that it is better to save your work and restart your R session.

liquidSVM is multi-threaded and is difficult to be multi-threaded externally, see documentation

Author(s)

Ingo Steinwart `<ingo.steinwart@mathematik.uni-stuttgart.de>`, Philipp Thomann `<philipp.thomann@mathematik.`

Maintainer: Philipp Thomann `<philipp.thomann@mathematik.uni-stuttgart.de>`

References

http://www.isa.uni-stuttgart.de

See Also

init.liquidSVM, trainSVMs, predict.liquidSVM, clean.liquidSVM, and test.liquidSVM, Configuration;

Examples

```
set.seed(123)
## Multiclass classification
modelIris <- svm(Species ~ ., iris)
y <- predict(modelIris, iris)

## Least Squares
modelTrees <- svm(Height ~ Girth + Volume, trees)
y <- predict(modelTrees, trees)
plot(trees$Height, y)
test(modelTrees, trees)

## Quantile regression
modelTrees <- qtSVM(Height ~ Girth + Volume, trees, scale=TRUE)
y <- predict(modelTrees, trees)

## ROC curve
modelWarpbreaks <- rocSVM(wool ~ ., warpbreaks, scale=TRUE)
y <- test(modelWarpbreaks, warpbreaks)
plotROC(y,warpbreaks$wool)
```

---

banana — banana-bc.train, banana-bc.test banana-mc.train, *and* banana-mc.test

---

Description

Generated data set having a binary or 4-level Y variable and a two-dimensional X (first two levels resemble bananas). Both the train and the test set have 2000 samples in the binary case, and 4000 in the multi-class case. They were generated by the authors and their collaborators.

bsSVM	*Bootstrap*

Description

This routine performs bootstrap learning for all scenarios except multiclass classification.

Usage

```
bsSVM(x, y, ..., solver, ws.number = 5, ws.size = 500, do.select = TRUE)
```

Arguments

`x`	either a formula or the features
`y`	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
`...`	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
`solver`	the solver to use. Can be any of `KERNEL_RULE`, `SVM_LS_2D`, `SVM_HINGE_2D`, `SVM_QUANTILE`, `SVM_EXPECTILE_2D`
`ws.number`	number of working sets to build and train
`ws.size`	how many samples to draw from the training set for each working set
`do.select`	if `TRUE` also does the whole selection for this model

Value

an object of type `svm`. Depending on the usage this object has also `$train_errors`, `$select_errors`, and `$last_result` properties.

clean.liquidSVM	*Force to release the internal memory of the C++ objects associated to this model.*

Description

Usually this has not to be done by the user since liquidSVM harnesses garbage collection offered by R.

Usage

```
## S3 method for class 'liquidSVM'
clean(model, warn = TRUE, ...)
```

Arguments

model	the SVM model as returned by init.liquidSVM
warn	if TRUE issue warning if the model already was deleted
...	not used at the moment

See Also

init.liquidSVM

Examples

```
## Multiclass classification
modelIris <- svm(Species ~ ., iris)
y <- predict(modelIris, iris)

## Least Squares
modelTrees <- svm(Height ~ Girth + Volume, trees)
y <- predict(modelTrees, trees)
plot(trees$Height, y)
test(modelTrees, trees)

clean(modelTrees)
clean(modelIris)
# now predict(modelTrees, ...) would not be possible any more
```

---

command-args *liquidSVM command line options*

Description

Should only be used by experts! liquidSVM command line tools svm-train, svm-select, and svm-test can be used by more advanced users to get the most advanced use. These three tools have command line arguments and those can be used from R as well.

Examples

```
## Not run:
reg <- liquidData('reg-1d')
model <- init.liquidSVM(Y~., reg$train)
trainSVMs(model, command.args=list(L=2, T=2, d=1))
selectSVMs(model, command.args=list(R=0,d=2))
result <- test(model, reg$test, command.args=list(T=1, d=0))

## End(Not run)
```

`compilationInfo`	*Compilation information: whether the library was compiled using SSE2 or even AVX.*

Description

Compilation information: whether the library was compiled using SSE2 or even AVX.

Usage

```
compilationInfo()
```

Value

character with the information.

`Configuration`	*liquidSVM model configuration parameters.*

Description

Different parameters configure different aspects of training/selection/testing. The learning scenarios set many parameters to corresponding default values, and these can again be changed by the user. Therefore the order in which they are specified can be important.

Usage

```
getConfig(model, name)

setConfig(model, name, value)
```

Arguments

`model`	the model
`name`	the name
`value`	the value

Value

the value of the configuration parameter

Overview of Configuration Parameters

`display` This parameter determines the amount of output of you see at the screen: The larger its value is, the more you see. This can help as a progress indication.

`scale` If set to a true value then for every feature in the training data a scaling is calculated so that its values lie in the interval $[0, 1]$. The training then is performed using these scaled values and any testing data is scaled transparently as well.

Because SVMs are not scale-invariant any data should be scaled for two main reasons: First that all features have the same weight, and second to assure that the default gamma parameters that liquidSVM provide remain meaningful.

If you do not have scaled the data previously this is an easy option.

`threads` This parameter determines the number of cores used for computing the kernel matrices, the validation error, and the test error.

- `threads=0` (default) means that all physical cores of your CPU run one thread.
- `threads=-1` means that all but one physical cores of your CPU run one thread.

`partition_choice` This parameter determines the way the input space is partitioned. This allows larger data sets for which the kernel matrix does not fit into memory.

- `partition_choice=0` (default) disables partitioning.
- `partition_choice=6` gives usually highest speed.
- `partition_choice=5` gives usually the best test error.

`grid_choice` This parameter determines the size of the hyper- parameter grid used during the training phase. Larger values correspond to larger grids. By default, a 10x10 grid is used. Exact descriptions are given in the next section.

`adaptivity_control` This parameter determines, whether an adaptive grid search heuristic is employed. Larger values lead to more aggressive strategies. The default `adaptivity_control = 0` disables the heuristic.

`random_seed` This parameter determines the seed for the random generator. `random_seed` = -1 uses the internal timer create the seed. All other values lead to repeatable behavior of the svm.

`folds` How many folds should be used.

Specialized configuration parameters

Parameters for regression (least-squares, quantile, and expectile)

`clipping` This parameter determines whether the decision functions should be clipped at the specified value. The value `clipping` = -1.0 leads to an adaptive clipping value, whereas `clipping` = 0 disables clipping.

Parameter for multiclass classification determine the multiclass strategy: `mc-type=0` : AvA with hinge loss. `mc-type=1` : OvA with least squares loss. `mc-type=2` : OvA with hinge loss. `mc-type=3` : AvA with least squares loss.

Parameters for Neyman-Pearson Learning

`class` The class, the `constraint` is enforced on.

`constraint` The constraint on the false alarm rate. The script actually considers a couple of values around the value of `constraint` to give the user an informed choice.

Hyperparameter Grid

For Support Vector Machines two hyperparameters need to be determined:

- `gamma` the bandwith of the kernel
- `lambda` has to be chosen such that neither over- nor underfitting happen. lambda values are the classical regularization parameter in front of the norm term.

liquidSVM has a built-in a cross-validation scheme to calculate validation errors for many values of these hyperparameters and then to choose the best pair. Since there are two parameters this means we consider a two-dimensional grid.

For both parameters either specific values can be given or a geometrically spaced grid can be specified.

`gamma_steps`, `min_gamma`, `max_gamma` specifies in the interval between `min_gamma` and `max_gamma` there should be `gamma_steps` many values

`gammas` e.g. `gammas=c(0.1,1,10,100)` will do these four gamma values

`lambda_steps`, `min_lambda`, `max_lambda` specifies in the interval between `min_lambda` and `max_lambda` there should be `lambda_steps` many values

`lambdas` e.g. `lambdas=c(0.1,1,10,100)` will do these four lambda values

`c_values` the classical term in front of the empirical error term, e.g. `c_values=c(0.1,1,10,100)` will do these four cost values (basically inverse of `lambdas`)

Note the min and max values are scaled according the the number of samples, the dimensionality of the data sets, the number of folds used, and the estimated diameter of the data set.

Using `grid_choice` allows for some general choices of these parameters

grid_choice	0	1	2
gamma_steps	10	15	20
lambda_steps	10	15	20
min_gamma	0.2	0.1	0.05
max_gamma	5.0	10.0	20.0
min_lambda	0.001	0.0001	0.00001
max_lambda	0.01	0.01	0.01

Using negative values of `grid_choice` we create a grid with listed gamma and lambda values:

grid_choice	-1
gammas	c(10.0, 5.0, 2.0, 1.0, 0.5, 0.25, 0.1, 0.05)
lambdas	c(1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001, 0.0000001)

grid_choice	-2
gammas	c(10.0, 5.0, 2.0, 1.0, 0.5, 0.25, 0.1, 0.05)
c_values	c(0.01, 0.1, 1, 10, 100, 1000, 10000)

Adaptive Grid

An adaptive grid search can be activated. The higher the values of `MAX_LAMBDA_INCREASES` and `MAX_NUMBER_OF_WORSE_GAMMAS` are set the more conservative the search strategy is. The values can be freely modified.

`ADAPTIVITY_CONTROL`	1	2
`MAX_LAMBDA_INCREASES`	4	3
`MAX_NUMBER_OF_WORSE_GAMMAS`	4	3

Cells

A major issue with SVMs is that for larger sample sizes the kernel matrix does not fit into the memory any more. Classically this gives an upper limit for the class of problems that traditional SVMs can handle without significant runtime increase. Furthermore also the time complexity is at least $O(n^2)$.

liquidSVM implements two major concepts to circumvent these issues. One is random chunks which is known well in the literature. However we prefer the new alternative of splitting the space into spatial cells and use local SVMs on every cell.

If you specify `useCells=TRUE` then the sample space X gets partitioned into a number of cells. The training is done first for cell 1 then for cell 2 and so on. Now, to predict the label for a value $x \in X$ liquidSVM first finds out to which cell this x belongs and then uses the SVM of that cell to predict a label for it.

```
If you run into memory issues turn cells on: \code{useCells=TRUE}
```

This is quite performant, since the complexity in both time and memore are both $O(\text{CELLSIZE} \times n)$ and this holds both for training as well as testing! It also can be shown that the quality of the solution is comparable, at least for moderate dimensions.

The cells can be configured using the `partition_choice`:

1. This gives a partition into random chunks of size 2000
   `VORONOI=c(1, 2000)`
2. This gives a partition into 10 random chunks
   `VORONOI=c(2, 10)`
3. This gives a Voronoi partition into cells with radius not larger than 1.0. For its creation a subsample containing at most 50.000 samples is used.
   `VORONOI=c(3, 1.0, 50000)`
4. This gives a Voronoi partition into cells with at most 2000 samples (approximately). For its creation a subsample containing at most 50.000 samples is used. A shrinking heuristic is used to reduce the number of cells.
   `VORONOI=c(4, 2000, 1, 50000)`
5. This gives a overlapping regions with at most 2000 samples (approximately). For its creation a subsample containing at most 50.000 samples is used. A stopping heuristic is used to stop the creation of regions if 0.5 * 2000 samples have not been assigned to a region, yet.
   `VORONOI=c(5, 2000, 0.5, 50000, 1)`

6. This splits the working sets into Voronoi like with `PARTITION_TYPE=4`. Unlike that case, the centers for the Voronoi partition are found by a recursive tree approach, which in many cases may be faster.
   `VORONOI=c(6, 2000, 1, 50000, 2.0, 20, 4,)`

The first parameter values correspond to `NO_PARTITION`, `RANDOM_CHUNK_BY_SIZE`, `RANDOM_CHUNK_BY_NUMBER`, `VORONOI_BY_RADIUS`, `VORONOI_BY_SIZE`, `OVERLAP_BY_SIZE`

Weights

- qt, ex: Here the number of considered tau-quantiles/expectiles as well as the considered tau-values are defined. You can freely change these values but notice that the list of tau-values is space-separated!
- npl, roc: Here, you define, which weighted classification problems will be considered. The choice is usually a bit tricky. Good luck ...

```
NPL:
WEIGHT_STEPS=10
MIN_WEIGHT=0.001
MAX_WEIGHT=0.5
GEO_WEIGHTS=1

ROC:
WEIGHT_STEPS=9
MAX_WEIGHT=0.9
MIN_WEIGHT=0.1
GEO_WEIGHTS=0
```

Grouped Cross Validation

By specifying `groupIds` when initializing an SVM samples obtain group ids. This by default also sets `FOLDS_KIND` to `GROUPED`. If the latter is the case then samples with the same group id will be put into the same fold at cross validation. This is important if e.g. there are several patients with several measurements each.

More Advanced Parameters

The following parameters should only employed by experienced users and are self-explanatory for these:

`KERNEL` specifies the kernel to use, at the moment either `GAUSS_RBF` or `POISSON`

`RETRAIN_METHOD` After training on grids and folds there are only solutions on folds. In order to construct a global solution one can either retrain on the whole training data (`SELECT_ON_ENTIRE_TRAIN_SET`) or the (partial) solutions from the training are kept and combined using voting (`SELECT_ON_EACH_FOLD` default)

`store_solutions_internally` If this is true (default in all applicable cases) then the solutions of the train phase are stored and can be just reused in the select phase. If you slowly run out of memory during the train phase maybe disable this. However then in the select phase the best models have to be trained again.

For completeness here are some values that usually get set by the learning scenario

SVM_TYPE KERNEL_RULE, SVM_LS_2D, SVM_HINGE_2D, SVM_QUANTILE, SVM_EXPECTILE_2D, SVM_TEMPLATE

LOSS_TYPE CLASSIFICATION_LOSS, MULTI_CLASS_LOSS, LEAST_SQUARES_LOSS, WEIGHTED_LEAST_SQUARES_LOSS, PINBALL_LOSS, TEMPLATE_LOSS

VOTE_SCENARIO VOTE_CLASSIFICATION, VOTE_REGRESSION, VOTE_NPL

KERNEL_MEMORY_MODEL LINE_BY_LINE, BLOCK, CACHE, EMPTY

FOLDS_KIND BLOCKS, ALTERNATING, RANDOM, STRATIFIED, GROUPED, RANDOM_SUBSET

WS_TYPE FULL_SET, MULTI_CLASS_ALL_VS_ALL, MULTI_CLASS_ONE_VS_ALL, BOOT_STRAP

`errors` *Obtain the test errors result.*

Description

After calculating the result in `test.liquidSVM` if labels were given **liquidSVM** also calculates the test error.

Usage

```
errors(y, showall = FALSE)
```

Arguments

`y`	the results of `test.liquidSVM`
`showall`	show the more detailed errors as well.

Details

Depending on the learning scenario there can be multiple errors: usually there is one per task, and `mcSVM` adds in front the global classification error. In the latter case the names give an information for what task the error was computed.

For each error also the positive and negative validation error can be shonw using `showall` for example in `rocSVM`.

Value

for all tasks the global and optionally also the positive/negative errors. Depending on the learning scenario there can be also a overall error (e.g. in multi-class classification).

See Also

`test.liquidSVM`

Examples

```
modelTrees <- svm(Height ~ Girth + Volume, trees[1:10, ])  # least squares

y <- test(modelTrees,trees[-1:-10,])
errors(y)

## Not run:
banana <- liquidData('banana-bc')
s_banana <- rocSVM(Y~., banana$test)
result <- test(s_banana, banana$train)
errors(result, showall=TRUE)

## End(Not run)
```

`exSVM` *Expectile Regression*

Description

This routine performs non-parametric, asymmetric least squares regression using SVMs. The tested estimators are therefore estimating the conditional tau-expectiles of Y given X. By default, estimators for five different tau values are computed. `svmExpectileRegression` is a simple alias of `exSVM`.

Usage

```
exSVM(x, y, ..., weights = c(0.05, 0.1, 0.5, 0.9, 0.95), clipping = -1,
  do.select = TRUE)

svmExpectileRegression(x, y, ..., weights = c(0.05, 0.1, 0.5, 0.9, 0.95),
  clipping = -1, do.select = TRUE)
```

Arguments

`x`	either a formula or the features
`y`	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
`...`	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
`weights`	the expectiles that should be estimated
`clipping`	absolute value where the estimated labels will be clipped. -1 (the default) leads to an adaptive clipping value, whereas 0 disables clipping.
`do.select`	if `TRUE` also does the whole selection for this model

Value

an object of type `svm`. Depending on the usage this object has also `$train_errors`, `$select_errors`, and `$last_result` properties.

Examples

```
## Not run:
tt <- ttsplit(quakes)
model <- exSVM(mag~., tt$train, display=1)
result <- test(model, tt$test)

errors(result)[2] ## is the same as
mean(ifelse(result[,2]<tt$test$mag, .1,.9) * (result[,2]-tt$test$mag)^2)

## End(Not run)
```

---

getCover — *Get Cover of partitioned SVM*

Description

If you use `voronoi=3` or `voronoi=4` this retrieves the voronoi centers that have been found.

Usage

```
getCover(model, task = 1)
```

Arguments

`model`	the model
`task`	the task between 1 and number of tasks

Value

the indices of the samples in the training data set that were used as Voronoi partition centers.

Note

This is not tested thoroughly so use in production is at your own risk.

Examples

```
## Not run:
banana <- liquidData('banana-mc')
model <- mcSVM(Y~.,banana$train, voronoi=c(4,500),d=1)
# task 4 is predicting 2 vs 3
cover <- getCover(model,task=4)
centers <- cover$samples
```

```
# we are considering task 4 and hence only show labels 2 and 3:
bananaSub <- banana$train[banana$train$Y %in% c(2,3),]
plot(bananaSub[,-1],col=bananaSub$Y)
points(centers,pch='x',cex=2)

if(require(deldir)){
  voronoi <- deldir::deldir(centers$X1,centers$X2,rw=c(range(bananaSub$X1),range(bananaSub$X2)))
   plot(voronoi,wlines="tess",add=TRUE, lty=1)
   text(centers$X1,centers$X2,1:nrow(centers),pos=1)
}

# let us calculate for every sample in this task which cell it belongs to
distances <- as.matrix(dist(model$train_data))
cells <- apply(distances[model$train_labels %in% c(2,3),cover$indices],1,which.min)
# and you can check that the cell sizes are as reported in the training phase for task 4
table(cells)

## End(Not run)
```

`getSolution` *Retrieve the solution of an SVM*

Description

Gives the solution of an SVM that has been trained and selected in an ad-hoc list.

Usage

```
getSolution(model, task = 1, cell = 1, fold = 1)
```

Arguments

`model`	the model
`task`	the task between 1 and number of tasks
`cell`	the cell between 1 and number of cells
`fold`	the fold between 1 and number of folds

Details

liquidSVM splits all problems into tasks (e.g. for multiclass classification or if using multiple weights), then each task is split into cells (maybe only a global one), and every cell then is trained in one or more folds to yiele a solution. Hence these coordinates have to be specified.

Value

a list with three entries: the offset of the solution (not yet implemented), the indices of the support vectors in the training data set, and the coefficients of the support vectors

Note

This is not tested thoroughly so use in production is at your own risk.

Examples

```
## Not run:
# simple example: regression of sinus curve
x <- seq(0,1,by=.01)
y <- sin(x*10)
a <- lapply(1:5, function(i)getSolution(model <- lsSVM(x,y,d=1), 1,1,i))
plot(x,y,type='l',ylim=c(-5,5));
for(i in 1:5) lines(coeff~samples, data=a[[i]],col=i)

# a more typical example
banana <- liquidData('banana-mc')
model <- mcSVM(Y~.,banana$train,d=1)
# task 4 is predicting 2 vs 3, there is only cell 1 here
solution <- getSolution(model,task=4,cell=1,fold=1)
supportvecs <- solution$samples
# we are considering task 4 and hence only show labels 2 and 3:
bananaSub <- banana$train[banana$train$Y %in% c(2,3),]
plot(bananaSub[,-1],col=bananaSub$Y)
points(supportvecs,pch='x',cex=2)

## End(Not run)
```

`init.liquidSVM` *Initialize an SVM object.*

Description

Should only be used by experts! This initializes a `svm` object and allocates in C++ an SVM model to which it keeps a reference.

Usage

```
init.liquidSVM(x, y, ...)

## S3 method for class 'formula'
init.liquidSVM(x, y, ..., d = NULL)

## Default S3 method:
init.liquidSVM(x, y, scenario = NULL, useCells = NULL,
  ..., sampleWeights = NULL, groupIds = NULL, ids = NULL, d = NULL)
```

Arguments

x	either a formula or the features
y	either the data or the labels corresponding to the features x. It can be a character in which case the data is loaded using liquidData. If it is of type liquidData then after training and selection the model is tested using the testing data (y$test).
...	configuration parameters, see Configuration. Can be threads=2, display=1, gpus=1, etc.
d	level of display information
scenario	configures the model for a learning scenario: E.g. scenario='ls', scenario='mc', scenario='npl', etc. Unlike the specialized functions qtSVM, exSVM, nplSVM etc. this does not trigger the correct select
useCells	if TRUE partitions the problem (equivalent to partition_choice=6)
sampleWeights	vector of weights for every sample or NULL (default) [currently has no effect]
groupIds	vector of integer group ids for every sample or NULL (default). If not NULL this will do group-wise folds, see folds_kind='GROUPED'.
ids	vector of integer ids for every sample or NULL (default) [currently has no effect]

Details

Since it binds heap memory it has to be released using clean.liquidSVM which is also performed at garbage collection.

The training data can either be provided using a formula and a corresponding data.frame or the features and the labels are given directly.

Value

an object of type svm

Methods (by class)

- formula: Initialize SVM model using a a formula and data
- default: Initialize SVM model using a data frame and a label vector

See Also

svm, predict.liquidSVM, test.liquidSVM and clean.liquidSVM

Examples

```
modelTrees <- init.liquidSVM(Height ~ Girth + Volume, trees[1:20, ])  # least squares
modelIris <- init.liquidSVM(Species ~ ., iris)  # multiclass classification
```

kern — *Calculates the kernel matrix.*

Description

Calculates the kernel matrix.

Usage

```
kern(data, gamma = 1, type = c("gaussian.rbf", "poisson"),
  threads = getOption("liquidSVM.default.threads", 1))
```

Arguments

`data`	the data set
`gamma`	the gamma-parameter
`type`	kernel type to use: one of "gaussian.rbf","poisson"
`threads`	how many threads to be used

Value

kernel matrix

Examples

```
kern(trees)
image(kern(trees, 2, "pois"))
```

liquidData — *Loads or downloads training and testing data*

Description

This looks at several locations to find a *name*`.train.csv` and *name*`.test.csv`. If it does then it loads or downloads it, parses it, and returns an `liquidData`-object. The files also can be gzipped having names *name*`.train.csv.gz` and *name*`.test.csv.gz`.

Usage

```
liquidData(name, factor_cols, header = FALSE, loc = c(".", "~/sml/data",
  "~/liquidData", system.file("data", package = "liquidSVM"), "../../../data",
  "http://www.isa.uni-stuttgart.de/liquidData"), prob = NULL,
  testSize = NULL, trainSize = NULL, stratified = NULL)

ttsplit(data, target = NULL, testProb = 0.2, testSize = NULL,
  stratified = NULL)

sample.liquidData(liquidData, prob = 0.2, trainSize = NULL,
  testSize = NULL, stratified = NULL)

## S3 method for class 'liquidData'
print(x, ...)
```

Arguments

`name`	name of the data set. If not given then a list of available names in `loc` is returned
`factor_cols`	list of column numbers that are factors (or list of header names, if `header=TRUE`)
`header`	do the data files have headers
`loc`	vector of locations where the data should be searched for
`prob`	probability of sample being put into test set
`testSize`	size of the test set. If stratified, this will only be approximately fulfilled.
`trainSize`	size of the train set. If stratified, this will only be approximately fulfilled.
`stratified`	whether sampling should be done separately in every bin defined by the unique values of the target column. Also can be index or name of the column in `data` that should be used to define bins.
`data`	the given data set
`target`	optional name or index of the target variable. If both this and `stratified` are not specified there will be no stratification.
`testProb`	probability of sample being put into test set
`liquidData`	the given liquidData
`x`	the model to print
`...`	other arguments to print.default

Value

if name is specified an liquidData object: an environment with $train and $test datasets as well as $name and optionally $target as name of the target variable. If no name is spacified a character vector of available names in `loc`.

See Also

`ttsplit`

Examples

```
banana <- liquidData('banana-mc')

## to get a smaller sample
liquidData('banana-mc',prob=0.2)
## if you disable stratified then there is some variance in the group sizes:
liquidData('banana-mc',prob=0.2, stratified=FALSE)

## Not run:
## to downlad a file from our web directory

liquidData("gisette")

## To get a list of available names:
liquidData()

## End(Not run)
## to produce an liquidData from some dataset
ttsplit(iris)
# the following will be stratified
ttsplit(iris,'Species')

# specify a testSize:
ttsplit(trees, testSize=10)
## example for sample.liquidData
banana <- liquidData('banana-mc')
sample.liquidData(banana, prob=0.1)
# this is equivalent to
liquidData('banana-mc', prob=0.1)
## example for print
banana <- liquidData("banana-mc")
print(banana)
```

`liquidSVM-class` *A Reference Class to represent a liquidSVM model.*

Description

A Reference Class to represent a liquidSVM model.

Fields

`cookie` this is used in C++ to access the model in memory lsSVM *Least Squares Regression*

Description

This routine performs non-parametric least squares regression using SVMs. The tested estimators are therefore estimating the conditional means of Y given X. `svmRegression` is a simple alias of `lsSVM`.

Usage

```
lsSVM(x, y, ..., clipping = -1, do.select = TRUE)

svmRegression(x, y, ..., clipping = -1, do.select = TRUE)
```

Arguments

`x`	either a formula or the features
`y`	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
`...`	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
`clipping`	absolute value where the estimated labels will be clipped. -1 (the default) leads to an adaptive clipping value, whereas 0 disables clipping.
`do.select`	if `TRUE` also does the whole selection for this model

Details

This is the default for `svm` if the labels are not a factor.

Value

an object of type `svm`. Depending on the usage this object has also `$train_errors`, `$select_errors`, and `$last_result` properties.

Examples

```
tt <- ttsplit(quakes)
model <- lsSVM(mag~., tt$train, display=1)
result <- test(model, tt$test)

errors(result) ## is the same as
mean( (tt$test$mag-result)^2 )
```

mcSVM — *Multiclass Classification*

Description

This routine is intended for both binary and multiclass classification. The binary classification is treated by an SVM solver for the classical hinge loss, and for the multiclass case, one-verus-all and all-versus-all reductions to binary classification for the hinge and the least squares loss are provided. The error of the very first task is the overall classification error. `svmMulticlass` is a simple alias of `mcSVM`

Usage

```
mcSVM(x, y, ..., predict.prob = FALSE, mc_type = c("AvA_hinge", "OvA_ls",
  "OvA_hinge", "AvA_ls"), do.select = TRUE)

svmMulticlass(x, y, ..., predict.prob = FALSE, mc_type = c("AvA_hinge",
  "OvA_ls", "OvA_hinge", "AvA_ls"), do.select = TRUE)
```

Arguments

`x`	either a formula or the features
`y`	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
`...`	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
`predict.prob`	If `TRUE` then a LS-svm will be trained and the conditional probabilities for the binary classification problems will be estimated. This also restricts the choices of `mc_type` to `c("OvA_ls","AvA_ls")`.
`mc_type`	configures the the multiclass variants for All-vs-All / One-vs-All and with hinge or least squares loss.
`do.select`	if `TRUE` also does the whole selection for this model

Details

Please look at the demo-vignette (`vignette('demo')`) for more examples.

`mcSVM` is best used with `factor`-labels. If there are just two levels in the factor, or just two unique values if it is `numeric` than a binary classification is performed. Else, by using the parameter `mc_type` different combinations of all-vs-all (`AvA`) and one-vs-all (`OvA`) and hinge (`hinge`) and least squares loss (`ls`) can be used.

If a test is performed then not only the final decision is returned but also the results of the intermediate binary classifications. This is indicated in the column names. If the training labels are given by a `factor` then the final decision will be encoded in this factor. If this is the case and `AvA_hinge` is used, then also the binary classification problems will receive the corresponding label...

Value

an object of type svm. Depending on the usage this object has also $train_errors, $select_errors, and $last_result properties.

See Also

Configuration

Examples

```
model <- mcSVM(Species ~ ., iris)
model <- mcSVM(Species ~ ., iris, mc_type="OvA")
model <- mcSVM(Species ~ ., iris, mc.type="AvA_hi")
model <- mcSVM(Species ~ ., iris, predict.prob=TRUE)

## Not run:
## a worked example can be seen at

vignette("demo",package="liquidSVM")

## End(Not run)
```

---

mlr-liquidSVM *liquidSVM functions for mlr*

Description

Allow for liquidSVM lsSVM and mcSVM to be used in the mlr framework.

Usage

```
makeRLearner.regr.liquidSVM()

trainLearner.regr.liquidSVM(.learner, .task, .subset, .weights = NULL,
  partition_choice = 0, partition_param = -1, ...)

predictLearner.regr.liquidSVM(.learner, .model, .newdata, ...)

makeRLearner.classif.liquidSVM()

trainLearner.classif.liquidSVM(.learner, .task, .subset, .weights = NULL,
  partition_choice = 0, partition_param = -1, ...)

predictLearner.classif.liquidSVM(.learner, .model, .newdata, ...)
```

Arguments

`.learner`	see mlr-Documentation
`.task`	see mlr-Documentation
`.subset`	see mlr-Documentation
`.weights`	see mlr-Documentation
`partition_choice`	the partition choice, see Configuration
`partition_param`	a further param for partition choice, see Configuration
`...`	other parameters, see Configuration
`.model`	the trained mlr-model, see mlr-Documentation
`.newdata`	the test features, see mlr-Documentation

Note

In order that mlr can find our learners liquidSVM has to be loaded using e.g. `library(liquidSVM)` `model <- train(...)`

Examples

```
## Not run:
if(require(mlr)){
library(liquidSVM)

## Define a regression task
task <- makeRegrTask(id = "trees", data = trees, target = "Volume")
## Define the learner
lrn <- makeLearner("regr.liquidSVM", display=1)
## Train the model use mlr::train to get the correct train function
model <- train(lrn,task)
pred <- predict(model, task=task)
performance(pred)

## Define a classification task
task <- makeClassifTask(id = "iris", data = iris, target = "Species")

## Define the learner
lrn <- makeLearner("classif.liquidSVM", display=1)
model <- train(lrn,task)
pred <- predict(model, task=task)
performance(pred)

## or for probabilities
lrn <- makeLearner("classif.liquidSVM", display=1, predict.type='prob')
model <- train(lrn,task)
pred <- predict(model, task=task)
performance(pred)

} # end if(require(mlr))
```

```
## End(Not run)
```

`nplSVM` *Neyman-Pearson-Learning*

Description

This routine provides binary classifiers that satisfy a predefined error rate on one type of error and that simlutaneously minimize the other type of error. For convenience some points on the ROC curve around the predefined error rate are returned. `nplNPL` performs Neyman-Pearson-Learning for classification.

Usage

```
nplSVM(x, y, ..., class = 1, constraint = 0.05, constraint.factors = c(3,
  4, 6, 9, 12)/6, do.select = TRUE)
```

Arguments

`x`	either a formula or the features
`y`	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
`...`	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
`class`	is the normal class (the other class becomes the alarm class)
`constraint`	gives the false alarm rate which should be achieved
`constraint.factors`	specifies the factors around `constraint`
`do.select`	if `TRUE` also does the whole selection for this model

Details

Please look at the demo-vignette (`vignette('demo')`) for more examples. The labels should only have value `c(1,-1)`.

`min_weight`, `max_weight`, `weight_steps`: you might have to define which weighted classification problems will be considered. The choice is usually a bit tricky. Good luck ...

Value

an object of type `svm`. Depending on the usage this object has also `$train_errors`, `$select_errors`, and `$last_result` properties.

Examples

```
## Not run:
model <- nplSVM(Y ~ ., 'banana-bc', display=1)

## a worked example can be seen at
vignette("demo",package="liquidSVM")

## End(Not run)
```

---

plotROC *Plots the ROC curve for a result or model*

Description

This can be used either using `rocSVM` or `lsSVM`

Usage

```
plotROC(x, correct, posValue = NULL, xlim = 0:1, ylim = 0:1, asp = 1,
  type = NULL, pch = "x", add = FALSE, ...)
```

Arguments

`x`	either the result from a `test` or a model
`correct`	either the true values or testing data for the model
`posValue`	the label marking the positive value. If `NULL` (default) then the larger value.
`xlim`	sets better defaults for `plot.default`
`ylim`	sets better defaults for `plot.default`
`asp`	sets better defaults for `plot.default`
`type`	sets better defaults for `plot.default`
`pch`	sets better defaults for `plot.default`
`add`	if 'FALSE' (default) produces a new plot and if 'TRUE' adds to existing plot.
`...`	gets passed to `plot.default`

See Also

rocSVM, lsSVM

`rocSVM`

Examples

```
## Not run:
banana <- liquidData('banana-bc')
model <- rocSVM(Y~.,banana$train)

plotROC(model ,banana$test)
# or:
result <- test(model, banana$test)
plotROC(result, banana$test$Y)

model.ls <- lsSVM(Y~., banana$train)
result <- plotROC(model.ls, banana$test)

## End(Not run)
```

---

predict.liquidSVM *Predicts labels of new data using the selected SVM.*

Description

After training and selection the SVM provides means to compute predictions for new input features. If you have also labels consider using `test.liquidSVM`.

Usage

```
## S3 method for class 'liquidSVM'
predict(object, newdata, ...)
```

Arguments

`object`	the SVM model as returned by `init.liquidSVM`
`newdata`	data frame of features to predict. If it has all the explanatory variables of `formula`, then the respective subset is taken.
`...`	other parameters passed to `test.liquidSVM`

Details

In the multi-result learning scenarios this returns all the predictions corresponding to the different quantiles, expectiles, etc. For multi-class classification, if the model was setup with `predict.prob=TRUE` Then this will return only the probability columns and not the prediction.

Value

the predicted values of test

See Also

`init.liquidSVM` and `test.liquidSVM`

Examples

```
## Multiclass classification
modelIris <- svm(Species ~ ., iris)
y <- predict(modelIris, iris)

## Least Squares
modelTrees <- svm(Height ~ Girth + Volume, trees)
y <- predict(modelTrees, trees)
plot(trees$Height, y)
```

`print.liquidSVM` *Printing an SVM model.*

Description

Printing an SVM model.

Usage

```
## S3 method for class 'liquidSVM'
print(x, ...)
```

Arguments

`x`	the model to print
`...`	other arguments to print.default

Examples

```
s_iris <- svm(solver='hinge', Species ~ ., iris)  # multiclass classification
print(s_iris)
```

`qtSVM` *Quantile Regression*

Description

This routine performs non-parametric and quantile regression using SVMs. The tested estimators are therefore estimating the conditional tau-quantiles of Y given X. By default, estimators for five different tau values are computed. `svmQuantileRegression` is a simple alias of `qtSVM`.

Usage

```
qtSVM(x, y, ..., weights = c(0.05, 0.1, 0.5, 0.9, 0.95), clipping = -1,
  do.select = TRUE)

svmQuantileRegression(x, y, ..., weights = c(0.05, 0.1, 0.5, 0.9, 0.95),
  clipping = -1, do.select = TRUE)
```

Arguments

`x`	either a formula or the features
`y`	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
`...`	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
`weights`	the quantiles that should be estimated
`clipping`	absolute value where the estimated labels will be clipped. -1 (the default) leads to an adaptive clipping value, whereas 0 disables clipping.
`do.select`	if `TRUE` also does the whole selection for this model

Value

an object of type `svm`. Depending on the usage this object has also `$train_errors`, `$select_errors`, and `$last_result` properties.

Examples

```
## Not run:
tt <- ttsplit(quakes)
model <- qtSVM(mag~., tt$train, display=1)
result <- test(model, tt$test)

errors(result)[2] ## is the same as
mean(ifelse(result[,2]<tt$test$mag, -.1,.9) * (result[,2]-tt$test$mag))

## End(Not run)
```

`read.liquidSVM` *Read and Write Solution from and to File*

Description

Reads or writes the solution from or to a file. The format of the solutions is the same as used in the command line version of liquidSVM. In addition also configuration data is written and by default also the training data. This can be interchanged also with the other bindings.

Usage

```
read.liquidSVM(filename, ...)

write.liquidSVM(model, filename)

serialize.liquidSVM(model, writeData = TRUE)

unserialize.liquidSVM(obj, ...)
```

Arguments

`filename`	the filename to read from/save to. Can be relative to the working directory.
`...`	passed to `init.liquidSVM`
`model`	the model
`writeData`	whether the training data should be serialized in the stream
`obj`	the data to unserialize

Details

The command line version of liquidSVM saves solutions after select in files of the name *data*`.sol` or *data*`.fsol` and uses those in the test-phase. `read.liquidSVM` and `write.liquidSVM` read and write the same format at the specified path. If you give a `filename` using extension `.fsol` the training data is written to the file and read from it. On the other hand, if you use the `.sol` format, you need to be able to reproduce the same data again once you read the solution. `readSolution` creates a new svm object.

Note

This is not tested thoroughly so use in production is at your own risk. Furthermore the serialize/unserialize hooks write temporary files.

See Also

`init.liquidSVM`, `write.liquidSVM`

Examples

```
## Not run:
banana <- liquidData('banana-bc')
modelOrig <- mcSVM(Y~., banana$train)
write.liquidSVM(modelOrig, "banana-bc.fsol")
write.liquidSVM(modelOrig, "banana-bc.sol")
clean(modelOrig) # delete the SVM object

# now we read it back from the file
modelRead <- read.liquidSVM("banana-bc.fsol")
# No need to train/select the data!
errors(test(modelRead, banana$test))

# to read the model where no data was saved we have to make sure, we get the same training data:
banana <- liquidData('banana-bc')
# then we can read it
modelDataExternal <- read.liquidSVM("banana-bc.sol", Y~., banana$train)
result <- test(modelDataExternal, banana$test)

# to serialize an object use:
banana <- liquidData('banana-bc')
modelOrig <- mcSVM(Y~., banana$train)
# we serialize it into a raw vector
```

```
obj <- serialize.liquidSVM(modelOrig)
clean(modelOrig) # delete the SVM object

# now we unserialize it from that raw vector
modelUnserialized <- unserialize.liquidSVM(obj)
errors(test(modelUnserialized, banana$test))

## End(Not run)
```

`reg-1d` — `reg-1d.train` *and* `reg-1d.test`

Description

Generated data set having a continuous Y variable and a one-dimensional X variable.

Details

Both the train and the test set have 2000 samples. They were generated by the authors and their collaborators.

`rocSVM` — *Receiver Operating Characteristic curve (ROC curve)*

Description

This routine provides several points on the ROC curve by solving multiple weighted binary classification problems. It is only suitable to binary classification data.

Usage

```
rocSVM(x, y, ..., weight_steps = 9, do.select = TRUE)
```

Arguments

`x`	either a formula or the features
`y`	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
`...`	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
`weight_steps`	indicates how many weights between `min_weight` and `max_weight` will be used
`do.select`	if `TRUE` also does the whole selection for this model

Details

Please look at the demo-vignette (`vignette('demo')`) for more examples. The labels should only have value `c(1,-1)`.

`min_weight`, `max_weight`, `weight_steps`: you might have to define which weighted classification problems will be considered. The choice is usually a bit tricky. Good luck ...

Value

an object of type `svm`. Depending on the usage this object has also `$train_errors`, `$select_errors`, and `$last_result` properties.

See Also

`plotROC`

Examples

```
## Not run:
banana <- liquidData('banana-bc')
model <- rocSVM(Y ~ ., banana$train, display=1)
plotROC(model,banana$test)

## a worked example can be seen at
vignette("demo",package="liquidSVM")

## End(Not run)
```

`selectSVMs` *Selects the best hyper-parameters of all the trained SVMs.*

Description

Should only be used by experts! This selects for every task and cell the best hyper-parameter based on the validation errors in the folds. This is saved and will afterwards be used in the evaluation of the decision functions.

Usage

```
selectSVMs(model, command.args = NULL, ..., d = NULL,
  warn.suboptimal = getOption("liquidSVM.warn.suboptimal", TRUE))
```

Arguments

model	the svm-model
command.args	further arguments aranged in a list, corresponding to the arguments of the command line interface to svm-select, e.g. list(d=2,R=0) is equivalent to svm-select -d 2 -R 0. See command-args for details.
...	parameters passed to selection phase e.g. retrain_method="select_on_entire_train_set"
d	level of display information
warn.suboptimal	if TRUE this will issue a warning if the boundary of the hyper-parameter grid was hit too many times. The default can be changed by setting options(liquidSVM.warn.suboptimal=F

Details

Some learning scenarios have to perform several selection runs: for instance in quantile regression for every quantile. This is done by specifying weight_number ranging from 1 to the number of quantiles.

See command-args for details.

Value

a table giving training and validation errors and more internal statistic for all the SVMs that were selected. This is also recorded in model$select_errors.

Documentation for command-line parameters of svm-select

The following parameters can be used as well:

- h=[<level>]
  Displays all help messages.

  Meaning of specific values:

  <level> = 0 => short help messages
  <level> = 1 => detailed help messages
  Allowed values:

  <level>: 0 or 1
  Default values:

  <level> = 0

- N=c(<class>,<constraint>)
  Replaces the best validation error in the search for the best hyper-parameter

  combination by an NPL criterion, in which the best detection rate is searched
  for given the false alarm constraint <constraint> on class <class>.
  Allowed values:

  <class>: -1 or 1

<constraint>: float between 0.0 and 1.0
Default values:

Option is deactivated.

- R=<method>
  Selects the method that produces decision functions from the different folds.

  Meaning of specific values:

  <method> = 0 => select for best average validation error
  <method> = 1 => on each fold select for best validation error
  Allowed values:

  <method>: integer between 0 and 1
  Default values:

  <method> = 1

- W=<number>
  Restrict the search for the best hyper-parameters to weights with the number

  <number>.
  Meaning of specific values:

  <number> = 0 => all weights are considered.
  Default values:

  <number> = 0

See Also

command-args, `svm`, `init.liquidSVM`, `selectSVMs`, `predict.liquidSVM`, `test.liquidSVM` and `clean.liquidSVM`

`setDisplay` *Set display info mode that controls how much information is displayed by liquidSVM C++ routines. Usually you will use* `display=d` *in* `svm(...)` *etc.*

Description

Set display info mode that controls how much information is displayed by liquidSVM C++ routines. Usually you will use `display=d` in `svm(...)` etc.

Usage

```
setDisplay(d = 1)
```

Arguments

d	the display information

---

svm *Convenience function to initialize, train, select, and optionally test an SVM.*

Description

The model is inited using the features and labels provided and training and selection is performed. If the labels are given as a `factor` classification is performed else least squares regression. If testing data is provided then this is used to calculate predictions and if test labels are provided also the test error and both are saved in `$last_result` of the returned `svm` object.

Usage

```
svm(x, y, ..., do.select = TRUE, testdata = NULL, testdata_labels = NULL,
  scenario = NULL, d = NULL, scale = TRUE, predict.prob = FALSE)
```

Arguments

x	either a formula or the features
y	either the data or the labels corresponding to the features `x`. It can be a `character` in which case the data is loaded using `liquidData`. If it is of type `liquidData` then after `training` and `selection` the model is `test`ed using the testing data (`y$test`).
...	configuration parameters, see Configuration. Can be `threads=2, display=1, gpus=1,` etc.
do.select	can be set to a list to args to be passed to the select phase
testdata	if supplied then also testing is performed. If this is `NULL` but `y` is of type `liquidData` then `y$test` is used.
testdata_labels	the labels used if testing is also perfomed.
scenario	configures the model for a learning scenario: E.g. `scenario='ls', scenario='mc', scenario='npl',` etc. Unlike the specialized functions `qtSVM, exSVM, nplSVM` etc. this does not trigger the correct `select`
d	level of display information
scale	if `TRUE` scales the features in the internal representation to values between 0 and 1.
predict.prob	If `TRUE` then a LS-svm will be trained and the conditional probabilities for the binary classification problems will be estimated. This also restricts the choices of `mc_type` to `c("OvA_ls","AvA_ls")`.

Details

The training data can either be provided using a formula and a corresponding `data.frame` or the features and the labels are given directly.

`svm` has one more difference to `lsSVM` and `mcSVM` because it uses `scale=TRUE` by default and the others do not.

Value

an object of type `svm`. Depending on the usage this object has also `$train_errors`, `$select_errors`, and `$last_result` properties.

See Also

`lsSVM`, `mcSVM`, `init.liquidSVM`, `trainSVMs`, `selectSVMs`

Examples

```
# since Species is a factor the following performs multiclass classification
modelIris <- svm(Species ~ ., iris)
# equivalently
modelIris <- svm(iris[,1:4], iris$Species)

# since Height is numeric the following performs least-squares regression
modelTrees <- svm(Height ~ Girth + Volume, trees)
# equivalently
modelTrees <- svm(trees[,c(1,3)],trees$Height)
```

`test.liquidSVM` *Tests new data using the selected SVM.*

Description

After training and selection the SVM provides means to evaluate labels for new input features. If you do not have labels consider using `predict.liquidSVM`. The errors for all tasks and cells are returned attached to the result (see `errors`).

Usage

```
## S3 method for class 'liquidSVM'
test(model, newdata, labels = NULL, command.args = NULL,
  ..., d = NULL)
```

Arguments

model	the SVM model as returned by init.liquidSVM
newdata	data frame of features to predict. If it has all the explanatory variables of formula, then the respective subset is taken. NAs will be removed.
labels	the known labels to test against. If NULL then they are retrieved from newdata using the original formula.
command.args	further arguments aranged in a list, corresponding to the arguments of the command line interface to svm-select, e.g. list(d=2,R=0) is equivalent to svm-select -d 2 -R 0. See command-args for details.
...	other configuration parameters passed to testing phase
d	level of display information

Details

If the SVM has multiple tasks the result will have corresponding columns. For mcSVM the first column gives the global vote and the other columns give the result for the corresponding binary classification problem indicated by the column name.

For convenience the latest result is always saved in model$last_result.

Value

predictions for all tasks together with errors (see errors). This is also recorded in model$last_result.

Documentation for command-line parameters of svm-test

The following parameters can be used as well:

- GPU=c(<use_gpus>,[<GPU_offset>])
  Flag controlling whether the GPU support is used. If <use_gpus> = 1, then each

  CPU thread gets a thread on a GPU. In the case of multiple GPUs, these threads are uniformly distributed among the available GPUs. The optional <GPU_offset> is added to the CPU thread number before the GPU is added before distributing the threads to the GPUs. This makes it possible to avoid that two or more independent processes use the same GPU, if more than one GPU is available.
  Allowed values:

  <use_gpus>: bool
  <use_offset>: non-negative integer.
  Default values:

  <gpus> = 0
  <gpu_offset> = 0
  Unfortunately, this option is not activated for the binaries you are currently

  using. Install CUDA and recompile to activate this option.

- `h=[<level>]`
  Displays all help messages.

  Meaning of specific values:

  <level> = 0 => short help messages
  <level> = 1 => detailed help messages
  Allowed values:

  <level>: 0 or 1
  Default values:

  <level> = 0

- `L=c(<loss>,[<neg_weight>,<pos_weight>])`
  Sets the loss that is used to compute empirical errors. The optional weights can

  only be set, if <loss> specifies a loss that has weights.
  Meaning of specific values:

  <loss> = 0 => binary classification loss
  <loss> = 1 => multiclass class
  <loss> = 2 => least squares loss
  <loss> = 3 => weighted least squares loss
  <loss> = 6 => your own template loss
  Allowed values:

  <loss>: integer between 0 and 2
  <neg_weight>: float > 0.0
  <pos_weight>: float > 0.0
  Default values:

  <loss> = 0
  <neg_weight> = 1.0
  <pos_weight> = 1.0

- `T=c(<threads>,[<thread_id_offset>])`
  Sets the number of threads that are going to be used. Each thread is

  assigned to a logical processor on the system, so that the number of allowed threads is bounded by the number of logical processors. On systems with activated hyperthreading each physical core runs one thread, if <threads> does not exceed the number of physical cores. Since hyper-threads on the same core share resources, using more threads than cores does usually not increase the performance significantly, and may even decrease it. The optional <thread_id_offset> is added before distributing the threads to the cores. This makes it possible to avoid that two or more independent processes use the same physical cores.
  Example: To run 2 processes with 3 threads each on a 6-core system call the first process with -T 3 0 and the second one with -T 3 3 .
  Meaning of specific values:

<threads> = 0 => 4 threads are used (all physical cores run one thread)
<threads> = -1 => 3 threads are used (all but one of the physical cores run one thread)
Allowed values:

<threads>: integer between -1 and 4
<thread_id_offset>: integer between 0 and 4
Default values:

<threads> = 0
<thread_id_offset> = 0

- `v=c(<weighted>,<scenario>,[<npl_class>])`
  Sets the weighted vote method to combine decision functions from different

  folds. If <weighted> = 1, then weights are computed with the help of the validation error, otherwise, equal weights are used. In the classification scenario, the decision function values are first transformed to -1 and +1, before a weighted vote is performed, in the regression scenario, the bare function values are used in the vote. In the weighted NPL scenario, the weights are computed according to the validation error on the samples with label <npl_class>, the rest is like in the classification scenario.
  <npl_class> can only be set for the NPL scenario.
  Meaning of specific values:

  <scenario> = 0 => classification
  <scenario> = 1 => regression
  <scenario> = 2 => NPL
  Allowed values:

  <weighted>: 0 or 1
  <scenario>: integer between 0 and 2
  <npl_class>: -1 or 1
  Default values:

  <weighted> = 1
  <scenario> = 0
  <npl_class> = 1

- `o=<display_roc_style>`
  Sets a flag that decides, wheather classification errors are displayed by

  true positive and false positives.
  Allowed values:

  <display_roc_style>: 0 or 1
  Default values:

  <display_roc_style>: Depends on option -v

See Also

command-args, init.liquidSVM, errors

Examples

```
modelTrees <- svm(Height ~ Girth + Volume, trees[1:10, ])  # least squares
result <- test(modelTrees, trees[11:31, ], trees$Height[11:31])
errors(result)
```

---

trainSVMs *Trains an SVM object.*

Description

Should only be used by experts! This uses the **liquidSVM** C++ implementation to solve all SVMs on the hyper-parameter grid.

Usage

```
trainSVMs(model, ..., solver = c("kernel.rule", "ls", "hinge", "quantile"),
  command.args = NULL, do.select = FALSE, useCells = FALSE, d = NULL)
```

Arguments

model	the svm-model
...	configuration parameters set before training
solver	solver to use: one of "kernel.rule","ls","hinge","quantile","expectile"
command.args	further arguments aranged in a list, corresponding to the arguments of the command line interface to `svm-train`, e.g. `list(d=2,W=2)` is equivalent to `svm-train -d 2 -W 2`. See command-args for details.
do.select	if not `FALSE` then the model is selected. This parameter can be used as a list of named arguments to be passed to the select phase
useCells	if `TRUE` partitions the problem (equivalent to `partition_choice=6`)
d	level of display information

Details

SVMs are solved for all tasks/cells/folds and entries in the hyper-parameter grid and can afterwards be selected using `selectSVMs`. A model even can be retrained using other parameters, reusing the training data. The training phase is usually the most time-consuming phase, and therefore for bigger problems it is recommended to use `display=1` to get some progress information.

See command-args for details.

Value

a table giving training and validation errors and more internal statistic for every SVM that was trained. This is also recorded in `model$train_errors`.

Documentation for command-line parameters of svm-train

The following parameters can be used as well:

- `f=c(<kind>,<number>,[<train_fraction>],[<neg_fraction>])`
  Selects the fold generation method and the number of folds. If <train_fraction>

  < 1.0, then the folds for training are generated from a subset with the
  specified size and the remaining samples are used for validation.
  Meaning of specific values:
  <kind> = 1 => each fold is a contiguous block
  <kind> = 2 => alternating fold assignmend
  <kind> = 3 => random
  <kind> = 4 => stratified random
  <kind> = 5 => random respecting group information of samples
  <kind> = 6 => random subset (<train_fraction> and <neg_fraction> required)
  Allowed values:

  <kind>: integer between 1 and 6
  <number>: integer >= 1
  <train_fraction>: float > 0.0 and <= 1.0
  <neg_fraction>: float > 0.0 and < 1.0
  Default values:

  <kind> = 3
  <number> = 5
  <train_fraction> = 1.00

- `g=c(<size>,<min_gamma>,<max_gamma>,[<scale>])`

- `g=<gamma_list>`
  The first variant sets the size <size> of the gamma grid and its endpoints

  <min_gamma> and <max_gamma>.
  The second variant uses <gamma_list> for the gamma grid.
  Meaning of specific values:

  <scale> Flag indicating whether <min_gamma> and <max_gamma> are scaled
  based on the sample size, the dimension, and the diameter.
  Allowed values:

  <size>: integer >= 1
  <min_gamma>: float > 0.0
  <max_gamma>: float > 0.0
  <scale>: bool

Default values:

<size> = 10
<min_gamma> = 0.200
<max_gamma> = 5.000
<scale> = 1

- `GPU=c(<use_gpus>,[<GPU_offset>])`
  Flag controlling whether the GPU support is used. If <use_gpus> = 1, then each

  CPU thread gets a thread on a GPU. In the case of multiple GPUs, these threads are uniformly distributed among the available GPUs. The optional <GPU_offset> is added to the CPU thread number before the GPU is added before distributing the threads to the GPUs. This makes it possible to avoid that two or more independent processes use the same GPU, if more than one GPU is available.
  Allowed values:

  <use_gpus>: bool
  <use_offset>: non-negative integer.
  Default values:

  <gpus> = 0
  <gpu_offset> = 0
  Unfortunately, this option is not activated for the binaries you are currently

  using. Install CUDA and recompile to activate this option.

- `h=[<level>]`
  Displays all help messages.

  Meaning of specific values:

  <level> = 0 => short help messages
  <level> = 1 => detailed help messages
  Allowed values:

  <level>: 0 or 1
  Default values:

  <level> = 0

- `i=c(<cold>,<warm>)`
  Selects the cold and warm start initialization methods of the solver. In

  general, this option should only be used in particular situations such as the implementation and testing of a new solver or when using the kernel cache.
  Meaning of specific values:

  For values between 0 and 6, both <cold> and <warm> have the same meaning as in Steinwart et al, 'Training SVMs without offset', JMLR 2011. These are:

0 Sets all coefficients to zero.
1 Sets all coefficients to C.
2 Uses the coefficients of the previous solution.
3 Multiplies all coefficients by C_new/C_old.
4 Multiplies all unbounded SVs by C_new/C_old.
5 Multiplies all coefficients by C_old/C_new.
6 Multiplies all unbounded SVs by C_old/C_new.
Allowed values:

Depends on the solver, but the range of <cold> is always a subset of the range of <warm>.
Default values:

Depending on the solver, the (hopefully) most efficient method is chosen.

- `k=c(<type>,[aux-file],[<Tr_mm_Pr>,[<size_P>],<Tr_mm>,[<size>],<Va_mm_Pr>,<Va_mm>])`
  Selects the type of kernel and optionally the memory model for the kernel matrices.

  Meaning of specific values:

  <type> = 0 => Gaussian RBF
  <type> = 1 => Poisson
  <type> = 3 => Experimental hierarchical Gauss kernel
  <aux_file> => Name of the file that contains additional information for the hierarchical Gauss kernel. Only this kernel type requires this option.
  <X_mm_Y> = 0 => not contiguously stored matrix
  <X_mm_Y> = 1 => contiguously stored matrix
  <X_mm_Y> = 2 => cached matrix
  <X_mm_Y> = 3 => no matrix stored
  <size_Y> => size of kernel cache in MB
  Here, X=Tr stands for the training matrix and X=Va for the validation matrix. In both cases, Y=Pr stands for the pre-kernel matrix, which stores the distances between the samples. If <Tr_mm_Pr> is set, then the other three flags <X_mm_Y> need to be set, too. The values <sizeY> must only be set if a cache is chosen.
  NOTICE: Not all possible combinations are allowed.
  Allowed values:

  <type>: integer between 0 and 3
  <X_mm_Y>: integer between 0 and 3
  <size_Y>: integer not smaller than 1
  Default values:

  <type> = 0
  <X_mm_Y> = 1
  <size_Y> = 1024
  <size> = 512

- `l=c(<size>,<min_lambda>,<max_lambda>,[<scale>])`

- `l=c(<lambda_list>,[<interpret_as_C>])`
  The first variant sets the size <size> of the lambda grid and its endpoints

  <min_lambda> and <max_lambda>.
  The second variant uses <lambda_list>, after ordering, for the lambda grid.
  Meaning of specific values:

  <scale> Flag indicating whether <min_lambda> is internally
  devided by the average number of samples per fold.
  <interpret_as_C> Flag indicating whether the lambda list should be
  interpreted as a list of C values
  Allowed values:

  <size>: integer >= 1
  <min_lambda>: float > 0.0
  <max_lambda>: float > 0.0
  <scale>: bool
  <interpret_as_C>: bool
  Default values:

  <size> = 10
  <min_lambda> = 0.001
  <max_lambda> = 0.100
  <scale> = 1
  <scale> = 0

- `L=c(<loss>,[<clipp>],[<neg_weight>,<pos_weight>])`
  Sets the loss that is used to compute empirical errors. The optional <clipp> value

  specifies where the predictions are clipped during validation. The optional weights
  can only be set if <loss> specifies a loss that has weights.
  Meaning of specific values:

  <loss> = 0 => binary classification loss
  <loss> = 2 => least squares loss
  <loss> = 3 => weighted least squares loss
  <loss> = 4 => pinball loss
  <loss> = 5 => hinge loss
  <loss> = 6 => your own template loss
  <clipp> = -1.0 => clipp at smallest possible value (depends on labels)
  <clipp> = 0.0 => no clipping is applied
  Allowed values:

  <loss>: values listed above
  <neg_weight>: float >= -1.0
  <neg_weight>: float > 0.0
  <pos_weight>: float > 0.0
  Default values:

  <loss> = native loss of solver chosen by option -S
  <clipp> = -1.000
  <neg_weight> = <weight1> set by option -W

<pos_weight> = <weight2> set by option -W

- `P=c(1,[<size>])`
- `P=c(2,[<number>])`
- `P=c(3,[<radius>],[<subset_size>])`
- `P=c(4,[<size>],[<reduce>],[<subset_size>])`
- `P=c(5,[<size>],[<ignore_fraction>],[<subset_size>],[<covers>])`
- `P=c(6,[<size>],[<reduce>],[<subset_size>],[<covers>],[<shrink_factor>])`
  [<max_width>] [<max_depth>]
  Selects the working set partition method.

  Meaning of specific values:

  <type> = 0 => do not split the working sets
  <type> = 1 => split the working sets in random chunks using maximum <size> of
  each chunk.
  Default values are:
  <size> = 2000
  <type> = 2 => split the working sets in random chunks using <number> of
  chunks.
  Default values are:
  <size> = 10
  <type> = 3 => split the working sets into Voronoi subsets of radius <radius>.
  If [subset_size] is set, a subset of this size is used to faster
  create the Voronoi partition. If subset_size == 0, the entire
  data set is used, otherwise, the radius is only approximately
  ensured.
  Default values are:
  <radius> = 1.000
  <subset_size> = 0
  <type> = 4 => split the working sets into Voronoi subsets of maximal size
  <size>. The optional flag <reduce> controls whether a heuristic
  to reduce the number of cells is used. If [subset_size] is set,
  a subset of this size is used to faster create the Voronoi
  partition. If subset_size == 0, the entire data set is used,
  otherwise, the maximal size is only approximately ensured.
  Default values are:
  <size> = 2000
  <reduce> = 1
  <subset_size> = 50000
  <type> = 5 => devide the working sets into overlapping regions of maximal
  size <size>. The process of creating regions is stopped when

<size> * <ignore_fraction> samples have not been assigned to
a region. These samples will then be assigned to the closest
region. If <subset_size> is set, a subset of this size is
used to find the regions. If subset_size == 0, the entire
data set is used. Finally, <covers> controls the number of
times the process of finding regions is repeated.
Default values are:.
<size> = 2000
<ignore_fraction> = 0.5
<subset_size> = 50000
<covers> = 1
<type> = 6 => split the working sets into Voronoi subsets of maximal size
<size>. The optional flag <reduce> controls whether a heuristic
to reduce the number of cells is used. If [subset_size] is set,
a subset of this size is used to faster create the Voronoi
partition. If subset_size == 0, the entire data set is used,
otherwise, the maximal size is only approximately ensured.
Unlike for <type> = 4, the centers for the Voronoi partition are
found by a recursive tree approach, which in many cases may be
faster. <shrink_factor> describes by which factor the number of
samples should at least be decreased. The recursion is stoppend
when either <max_width> * <size> is greater than the current
working subset or the <max_tree_depth> is reached. For both
parameters, a value of 0 means that the corresponding condition
above is ignored.
Default values (so far, they are only a brave guess) are:
<size> = 2000
<reduce> = 1
<subset_size> = 50000
<shrink_factor> = 2.0000
<max_width> = 20
<max_tree_depth> = 4
Allowed values:

<type>: integer between 0 and 6
<size>: positive integer
<number>: positive integer
<radius>: positive real
<subset_size>: non-negative integer
<reduce>: bool
<covers>: positive integer
<shrink_factor>: real > 1.0
<max_width>: non-negative integer
<max_tree_depth>: non-negative integer
Default values:

<type> = 0

- `r=<seed>`
  Initializes the random number generator with <seed>.

  Meaning of specific values:

  <seed> = -1 => a random seed based on the internal timer is used
  Allowed values:

  <seed>: integer between -1 and 2147483647
  Default values:

  <seed> = -1

- `s=c(<clipp>,[<stop_eps>])`
  Sets the value at which the loss is clipped in the solver to <value>. The

  optional parameter <stop_eps> sets the threshold in the stopping criterion
  of the solver.
  Meaning of specific values:

  <clipp> = -1.0 => Depending on the solver type clipp either at the
  smallest possible value (depends on labels), or
  do not clipp.
  <clipp> = 0.0 => no clipping is applied
  Allowed values:

  <clipp>: -1.0 or float >= 0.0.
  In addition, if <clipp> > 0.0, then <clipp> must not be smaller
  than the largest absolute value of the samples.
  <stop_eps>: float > 0.0
  Default values:

  <clipp> = -1.0
  <stop_eps> = 0.0010

- `S=c(<solver>,[<NNs>])`
  Selects the SVM solver <solver> and the number <NNs> of nearest neighbors used

  in the working set selection strategy (2D-solvers only).
  Meaning of specific values:

  <solver> = 0 => kernel rule for classification
  <solver> = 1 => LS-SVM with 2D-solver
  <solver> = 2 => HINGE-SVM with 2D-solver
  <solver> = 3 => QUANTILE-SVM with 2D-solver
  <solver> = 4 => EXPECTILE-SVM with 2D-solver
  <solver> = 5 => Your SVM solver implemented in template_svm.*
  Allowed values:

  <solver>: integer between 0 and 5
  <NNs>: integer between 0 and 100
  Default values:

<solver> = 2
<NNs> = depends on the solver

- `T=c(<threads>,[<thread_id_offset>])`
  Sets the number of threads that are going to be used. Each thread is

  assigned to a logical processor on the system, so that the number of allowed threads is bounded by the number of logical processors. On systems with activated hyperthreading each physical core runs one thread, if <threads> does not exceed the number of physical cores. Since hyper-threads on the same core share resources, using more threads than cores does usually not increase the performance significantly, and may even decrease it. The optional <thread_id_offset> is added before distributing the threads to the cores. This makes it possible to avoid that two or more independent processes use the same physical cores.
  Example: To run 2 processes with 3 threads each on a 6-core system call the first process with -T 3 0 and the second one with -T 3 3 .
  Meaning of specific values:

  <threads> = 0 => 4 threads are used (all physical cores run one thread)
  <threads> = -1 => 3 threads are used (all but one of the physical cores run one thread)
  Allowed values:

  <threads>: integer between -1 and 4
  <thread_id_offset>: integer between 0 and 4
  Default values:

  <threads> = 0
  <thread_id_offset> = 0

- `w=c(<neg_weight>,<pos_weight>)`
- `w=c(<min_weight>,<max_weight>,<size>,[<geometric>,<swap>])`
- `w=c(<weight_list>,[<swap>])`
  Sets values for the weights, solvers should be trained with. For solvers

  that do not have weights this option is ignored.
  The first variants sets a pair of values.
  The second variant computes a sequence of weights of length <size>.
  The third variant takes the list of weights.
  Meaning of specific values:

  size> = 1 => <weight1> is the negative weight and <weight2> is the positive weight.
  <size> > 1 => <size> many pairs are computed, where the positive weights are between <min_weight> and <max_weight> and the negative weights are 1 - pos_weight.

<geometric> Flag indicating whether the intermediate positive weights are geometrically or arithmetically distributed.
<swap> Flag indicating whether the role of the positive and negative weights are interchanged.
Allowed values:

<... weight ...>: float > 0.0 and < 1.0
<size>: integer > 0
<geometric>: bool
<swap>: bool
Default values:

<weight1> = 1.0
<weight2> = 1.0
<size> = 1
<geometric> = 0
<swap> = 0

- `W=<type>`
  Selects the working set selection method.

  Meaning of specific values:

  <type> = 0 => take the entire data set
  <type> = 1 => multiclass 'all versus all'
  <type> = 2 => multiclass 'one versus all'
  <type> = 3 => bootstrap with <number> resamples of size <size>
  Allowed values:

  <type>: integer between 0 and 3
  Default values:

  <type> = 0

See Also

`command-args`, `svm`, `init.liquidSVM`, `selectSVMs`, `predict.liquidSVM`, `test.liquidSVM` and `clean.liquidSVM`

`write.liquidData` *Write Smldata*

Description

Write `liquidData` in such a way that it is understood by liquidSVM command line utilities.

Usage

```
write.liquidData(data, location = ".", label = 1, name = NULL,
  type = "csv")
```

Arguments

`data`	the liquidData to write
`location`	the location to write `name.train.csv` and `name.test.csv`
`label`	the column with this index or this name will become the label column, and be written as the first column.
`name`	the name of the file. If `NULL` (default) then takes the `data$name`
`type`	the format of output. At the moment only `"csv"` is supported.

Index