Stacking

Idea

The underlying idea of stacking is that a combination of models should perform better than just a single one. This is generally known as ensemble. Other ensemble methods work the same way, connecting typically multiple weak learners to build a strong one. Examples are random forest which consists of many single trees, or boosting which uses trees as well as linear models as weak components. However, there are some differences: Stacking nowadays (normally) make use of well perfoming base learners, whereas the idea of *classical* ensembles is to use learners which "performs only slightly better than random guessing" as stated in Schapire (1990). Another point where these two methods differ is that ensembles are usually aggregated employing simple average method or majority voting, whereas stacking – in addition to that – also makes use of more sophisticated aggregation methods like ensemble selection or classification/regression learners. In addtion to these classical stacking methods a new, fourth method is implemented called boosted stacking.

Basic Setup

In the following some naming conventions for the basic setup will be provided. Stacking was introduced by Wolpert (1992) as a 2-staged method. As every learning problem it starts with the data set, referred as *level 0 data*. Using this data, the so called *level 0 learners/generalizer* or *base learners* are applied. Depending on the stacking method, obtained base learner predictions are created employing cross-validation or simple training and prediction on the same observations. The obtained predictions create a new data set which is called *level 1 data*. Now the combiner is deployed, which aggregates the data to the final prediction. The combiner is also known as *level 1 generaliser*, *level 1 learner*, *aggregator* or just *combiner*. In reference to van der Laan, Polley, and Hubbard (2007) the term *super learner* is only used for the stacking method which uses a supervised method as combiner.

Four stacking methods are implemented:

- Aggregation Method,
- Super Learner,
- Ensemble Selection,

in function `makeStackedLearner` , and

- Boosted Stacking

in function `makeBoostedStackingLearner` .

Base Learners

Any learner which is implemented can be used as base learner. Use `listLearners` to find all learners which are currently implemented and use the arguments to select learners which are useful for your use case and data.

Note that currently only regression, binary and multiclass classification are supported for stacking. Moreover, it is mandatory that base learners have unique ids.

```
bls = list(makeLearner("classif.kknn"),
  makeLearner("classif.randomForest"),
  makeLearner("classif.rpart", id = "rp1", minsplit = 5),
  makeLearner("classif.rpart", id = "rp2", minsplit = 10),
  makeLearner("classif.rpart", id = "rp3", minsplit = 15),
  makeLearner("classif.rpart", id = "rp4", minsplit = 20),
  makeLearner("classif.rpart", id = "rp5", minsplit = 25)
)
```

For classification two prediction types are available: probabilites and response labels. It is useful to use `predict.type = "prob"` to obtain prediction probabilites. Using the alternative (`predict.type = "response"`) will lead to a loss in information as stated in Ting and Witten (1999). While response lables are the default as prediction type, you need to change it which can be done with only one line of code:

```
bls = lapply(bls, function(x) setPredictType(x, predict.type = "prob"))
```

The base learners for boosted stacking are created using `ModelMultiplexer` and are described in the corresponsing subsection below.

Stacking Algorithms

Aggregation Method

Aggregation is the easiest stacking method, nevertheless it is often used. For regression method “average” is always employed, that means that the final prediction is created by calculating the arithmetic mean of the predicted results. For classification, aggregation methods depend on the prediction type of the base learners as well as of the type of the final prediction. It is referenced to the table below.

Table 1: Aggregation methods for classification tasks.

Base learner prediction type	**Final prediction type**	**Method**
probabilites	probabilites	average
probabilites	labels	average
labels	probabilites	ratio
lables	labels	mode

When base learner prediction types are probabilites, the method is always the arithmetic mean ("average"). Method "ratio" counts the ratio of labels predicted from base learners in relation to the totel number of learners. Whereas "mode" means that the label will be predicted which occurs the most regarding one observation.

Note that the final prediction type needs to be specified within `makeStackedLearner` by passing `"prob"` or `"response"` to `predict.type`.

```
ave = makeStackedLearner(method = "aggregate", base.learners = bls,
  predict.type = "prob")
train.idx = sample(1:getTaskSize(iris.task), 100)
test.idx = setdiff(1:getTaskSize(iris.task), train.idx)
mod = train(ave, subsetTask(iris.task, train.idx))
pred = predict(mod, subsetTask(iris.task, test.idx))
performance(pred)
```

```
## mmce
## 0.04
```

Super Learner

Stacking using a classification or regression learner as combiner is a common approach. In the first step all base learners are run using a cross-validation. Moreover, models (=base models) are generated using the whole training set. The cross-validated prediction results are used as level 1 data. On top of this new data set the super learner is applied. All models regarding base and super learner are saved in the train object. When the stacking model is employed on new test data, predictions are generated using base models to obtain level 1 (test) data. On top of that the super model is applied which leads to the final predictions results.

The usage of cross-validation to obtain level 1 data is crucial to cope overfitting. The default is a 5-fold CV. With `use.feat` the original features can be appended to the level 1 data set, which may improve the performance in some cases.

```
base = c("regr.rpart", "regr.randomForest", "regr.svm")
lrns = lapply(base, makeLearner)
spr = makeStackedLearner(method = "superlearner",
  base.learners = lrns, predict.type = "response", resampling = cv3,
  super.learner = "regr.kknn", use.feat = TRUE)
bh.train = subsetTask(bh.task, 1:450)
bh.test = subsetTask(bh.task, 451:getTaskSize(bh.task))
mod = train(spr, bh.train)
pred = predict(mod, bh.test)
performance(pred)
```

```
##     mse
## 10.3847
```

Ensemble Selection

Ensemble selection was introduced by Caruana et al. (2004). It is a greedy forwards search, which chooses the best models from a library of (base) models. To prevent overfitting the selection takes place based on the performance obtained by cross-validated prediction results. The algorithm consists of two loops: In the outer loop a subset of models regarding **bagprop** is chosen. It is conducted as often as defined in **bagtime**. Within each iteration a fixed number (**init**) of best learners is picked depending on the **metric** defined. In addition to that an inner loop is conducted. Here as many models are added until the performance does not improve at least the value of the **tolerance** parameter *or* until **maxiter** is reached.

The parameters in detail:

- **bagtime:** An integer which determines the number of bagging iterations where models are added to the ensemble (this is not equal to the number of models added).
- **bagprop:** A numeric value between zero and one, which defines the fraction of models which will be part of every single bagging iteration.
- **init:** An integer value (starting at 1), which defines the number of best models from a bag added to the ensemble at the beginning of each iteration.
- **replace:** A logical indicator controlling if base models can be added only once or several times within a bagging iteration.
- **maxiter:** The maximal number of iterations within a bagging step (where models will be added).
- **tolerance:** When performance does not improve at least the value specified here, the inner loop is stopped (for measures which needs to be minimized). Note, that for metrics which need to be maximied, a negative value need to be specified to obtain the same result.
- **metric:** Specifies the metric which should be optimized.

To apply the model on new test data, predictions are obtained using the base models which were selected. These predictions (=level 1 data) are expanded regarding the frequency they are chosen. Finally, the predictions are combined using aggregation method.

Note, that tolerance should be carefully selected according to the metric in use. Performance differences between models are usually smaller for `acc` than for `auc`. When the tolerance is set to a negative value the adding of models will terminate at a later point (more diverse models will be added). In Caruana et al. (2004) 2000 base models were used. Regarding that setting they suggest *bagtime of 20 or 25* and *bagprop = 0.5*.

```
ensel = makeStackedLearner(method = "ensembleselection",
  base.learners = bls, predict.type = "prob",
  es.par.vals = list(bagtime = 10, init = 3, metric = logloss))
res.ensel = resample(ensel, pid.task, cv3, measures = logloss,
  model = TRUE)
performance(res.ensel$pred, measures = logloss)
```

```
##   logloss
## 0.4872104
```

Boosted Stacking

Boosted stacking is a new stacking method. It combines stacking of models in an iterative boosting fashion. The basic idea is to tune a well performing model, and then add the prediction as a new feature to the data set. The base learners are described in the `ModelMultiplexer` and the corresponding parameter space is defined in the `ParamSet` object. For tuning, a tuning algorithm (here named `ctrl`), containing the budget (here `maxit`), and a resampling strategy (here `cv10`) need to be specified. Moreover, the maximal number of boosting steps (`niter`) and the `tolerance` need to be set. If the tolerance is set to a negative value, boosting does not stop even if the performance worsen. With the argument `subsemble.prop` the observation size can be reduced. In every boosting step another subset will be created, where tuning and model fitting will be employed. The default is 80%.

```
lrns = list(
   makeLearner("classif.kknn"),
   makeLearner("classif.gbm", distribution = "bernoulli"),
   makeLearner("classif.randomForest"))
lrns = lapply(lrns, setPredictType, "prob")
mm = makeModelMultiplexer(lrns)
ps = makeModelMultiplexerParamSet(mm,
  makeIntegerParam("k", lower = 1L, upper = 20L),
  makeIntegerParam("n.trees", lower = 1L, upper = 500L),
  makeIntegerParam("interaction.depth", lower = 1L, upper = 10L),
  makeIntegerParam("ntree", lower = 1L, upper = 500L),
  makeIntegerParam("nodesize", lower = 1L, upper = 20L))
ctrl = makeTuneControlRandom(maxit = 5L)
boost = makeBoostedStackingLearner(model.multiplexer = mm,
  predict.type = "prob", resampling = cv10, mm.ps = ps,
  control = ctrl, measures = mmce, niter = 5L,
  tolerance = -.5)
res.boost = resample(boost, task = pid.task, resampling = hout,
  models = TRUE)
```

```
res.boost$models[[1]]$learner.model$score
```

```
##        classif.kknn.1 classif.randomForest.2 classif.randomForest.3
##             0.2493902              0.2295732              0.2492683
##        classif.kknn.4 classif.randomForest.5
##             0.2590244              0.2590244
```

Reducing Computation Time

Parallelization

Stacking can be computational expensive especially when many base learners or big data sets are used. In `mlr` several functionalities can be run in parallel. For parallelization the package `parallelMap` is used. The function `parallelGetRegisteredLevels` lists all levels which can be parallelized.

```
library(parallelMap)
parallelGetRegisteredLevels()
```

```
## mlr: mlr.benchmark, mlr.resample, mlr.selectFeatures, mlr.tuneParams, m
lr.stackedLearner
```

For StackedLearner `mlr.resample` and `mlr.stackedLearner` are interesting. For BoostedStacking use `mlr.resample` or `mlr.tuneParams`. See `?parallelStart` for more information about the parallelization setup. In general it is good practice to stop parallelization after its usage with `parallelStop`.

StackedLearner

Level `mlr.resample` runs the resampling procedure in parallel (in the case below it would be a maximal parallelization of three, because 3-fold CV is used). The second method (`mlr.stackedLearner`) leads to a parallelization of the single base learners in the training task. Hence, in case of a stack of seven base learners (as it is in `ensel`) a maximum of seven procedures can run at once.

```
parallelStartMulticore(cpus = 3, logging = FALSE,
  level = "mlr.stackedLearner", show.info = FALSE)
res.ensel2 = resample(ensel, pid.task, cv3, models = T)
parallelStop()
```

BoostedStacking

For boosted stacking the outer resampling (here `cv3`) or the single learners in the parameter tuning (here `maxit = 5L`) can be parallelized.

```
parallelStartMulticore(cpus = 3, logging = FALSE,
  level = "mlr.tuneParams", show.info = FALSE)
res.boost2 = resample(boost, pid.task, cv3)
parallelStop()
```

Reuse results

In case a resampling for `StackedLearner` should be run with different settings it is easiest to use `resampleStackedLearnerAgain`. This function reuses already done work from the `ResampleResult`, i.e. reuses fitted base models (needed to obtain level 1 data based on test data) and level 1 training data. Note, that models need to be present (i.e. `save.preds = TRUE` in `makeStackedLearner`). When using `save.on.disc = TRUE` in `makeStackedLearner` resampling procedure “Holdout” is allowed only (model names are not unique regarding CV fold number).

To make use of that function the `ResampleResult`, the task, and the new parameter settings need to be passed. Use `super.learner` and `use.feat` for super learner method, `es.par.vals` for ensemble selection or keep all empty for aggregation method.

```
# 2 new ensemble selection settings
reuse1 = resampleStackedLearnerAgain(obj = res.ensel,
  task = pid.task, es.par.vals = list(init = 2, bagtime = 5))
reuse2 = resampleStackedLearnerAgain(obj = res.ensel,
  task = pid.task, measures = list(mmce, auc),
  es.par.vals = list(bagprop = .8, bagtime = 15, metric = auc))
# 1 new super learner setting
reuse3 = resampleStackedLearnerAgain(obj = res.ensel,
  task = pid.task, measures = mmce, super.learner = bls[[2]],
  use.feat = TRUE)
# new aggregation setting (no parameter specified)
reuse4 = resampleStackedLearnerAgain(obj = res.ensel,
  task = pid.task, measures = list(mmce))
```

Compare performance:

```
all.res = list(Base = res.ensel, Reuse1 = reuse1,
  Reuse2 = reuse2, Reuse3 = reuse3, Reuse4 = reuse4)
extractSubList(all.res, "aggr")
```

```
## $Base
## logloss.test.mean
##         0.4872104
##
## $Reuse1
## mmce.test.mean
##      0.2447917
##
## $Reuse2
## mmce.test.mean  auc.test.mean
##      0.2773438      0.7961199
##
## $Reuse3
## mmce.test.mean
##      0.2369792
##
## $Reuse4
## mmce.test.mean
##      0.2760417
```

Compare running time (in seconds):

```
sapply(all.res, function(x) round(x$runtime, 2))
```

```
##   Base Reuse1 Reuse2 Reuse3 Reuse4
##  15.83   1.89   2.14   1.17   0.34
```

A comparable function for `BoostedStackingLearner` is not available.

Save Memory

R is not famous for being memory efficient. If you use a big number of base learners as well as large data sets computation can break due to memory limits. To overcome this problem there are two possibilites.

1. Don't save everything: you can set `models = FALSE` in resample. As a consequence no models will be saved.

```
spr = makeStackedLearner(method = "superlearner",
  base.learners = bls, predict.type = "prob", resampling = cv5,
  super.learner = "classif.randomForest")
res.withmodels = resample(spr, pid.task, cv2, models = TRUE)
res.nomodels = resample(spr, pid.task, cv2, models = FALSE)

res = list(WithModels = res.withmodels, NoModels = res.nomodels)
sapply(res, function(x) paste(round(object.size(x) / 10^6, 2), "MB"))
```

```
## WithModels   NoModels
## "13.97 MB"  "0.05 MB"
```

This comes with the disadvantage that all models will be lost, i.e. all information regarding models in super learner and selected learners in ensemble selection are not available.

Yet, you still have the possibility to save only the interesting part of the model. For random forest this might be *feature importance*.

```
res = resample(spr, pid.task, cv2, models = FALSE,
  extract = function(x) x$learner.model$super.model$learner.model$importan
ce)
paste(round(object.size(res) / 10^6, 2), "MB")
```

```
## [1] "0.05 MB"
```

2. Save the base models on disc (in your working directory).

This is only implemented for stacking methods accessible through `makeStackedLearner`.

```
spr.disc = makeStackedLearner(method = "superlearner", base.learners = bls
, predict.type = "prob", resampling = cv5, super.learner = "classif.random
Forest", save.on.disc = TRUE)
res.disc = resample(spr.disc, pid.task, cv2, models = TRUE)

res = list(WithModels = res.withmodels, OnDisc = res.disc)
sapply(res, function(x) paste(round(object.size(x) / 10^6, 2), "MB"))
```

```
## WithModels     OnDisc
## "13.97 MB"  "6.96 MB"
```

Here all models are still available and `resampleStackedLearnerAgain` can be used.

References

Caruana, R., A. Niculescu-Mizil, G. Crew, and A. Ksikes. 2004. “Ensemble Selection from Libraries of Models.” Proceedings of the twenty-first international conference on Machine learning, 18. http://www.cs.cornell.edu/~alexn/papers/shotgun.icml04.revised.rev2.pdf (http://www.cs.cornell.edu/~alexn/papers/shotgun.icml04.revised.rev2.pdf).

Schapire, R. E. 1990. “The Strength of Weak Learnability.” no. 5(2). Machine learning: 197–227.

Ting, K. M., and I. H. Witten. 1999. “Issues in Stacked Generalization,” no. 10. Journal of Artificial Intelligence Research: 271–89. http://www.jair.org/media/594/live-594-1796-jair.ps.Z (http://www.jair.org/media/594/live-594-1796-jair.ps.Z).

van der Laan, M. J., E. C. Polley, and A. E. Hubbard. 2007. “Super Learner.” no. 6(1). Statistical Applications in Genetics and Molecular Biology. http://citeseerx.ist.psu.edu/viewdoc

/download?doi=10.1.1.211.6393&rep=rep1&type=pdf (http://citeseerx.ist.psu.edu/viewdoc /download?doi=10.1.1.211.6393&rep=rep1&type=pdf).

Wolpert, D. H. 1992. “Stacked Generalization.” no. 5(2). Neural networks: 241–49. http://www.machine-learning.martinsewell.com/ensembles/stacking/Wolpert1992.pdf (http://www.machine-learning.martinsewell.com/ensembles/stacking/Wolpert1992.pdf).