hyperbandr tutorial

by Niklas

hyperbandr

This is an R6 implementation of the original **hyperband** algorithm https://arxiv.org/abs/1603.06560.

R6 is an encapsulated object oriented system akin to those in Java or C++, where objects contain methods in addition to data, and those methods can modify objects directly (unlike S3 and S4 which are both functional object-oriented systems, where class methods are separate from objects, and objects are not mutable).

Essentially, that means that we obtain a very generic implementation, which is working with every other R package (as long the algorithm meets the requirements of hyperband).

This tutorial contains a general introduction, four examples and a small guide to mlr:

1. General introduction of the mechanics of the **hyperbandr** package
2. Example 1: hyperband to optimize a **neural network** with mxnet and mlr (very detailed)
3. Example 2: hyperband in combination with MBO to optimize a **neural network with mxnet, mlr and mlrMBO
4. Example 3: hyperband to optimize a **gradient boosting** model with xgboost and mlr
5. Example 4: hyperband to optimize a **function** with smoof
6. appendix: introduction to mlr

1. General introduction

In order to call **hyperband**, we need to define five things:

1. a hyperparameter search space
2. a function to sample configurations
3. a function to initialize models
4. a function to train models
5. a function to evaluate the performance of a model

1: the hyperparameter search space

We begin with the hyperparameter search space. That search space includes all hyperparameters we would like to consider, as well as a reasonable range of values for each of them.

```
mySearchSpace = ...
```

2: the sampling function

Following up, we need a function to sample an arbitrary amount of hyperparameter configurations from our search space.

The inputs of that function are:

- **par.set**: the search space
- **n.configs**: the number of configurations to sample
- **...**: additional arguments to access the hyper storage (see example 2 how to utilize this feature to combine hyperband with MBO)

```
sample.fun = function(par.set, n.configs, ...) {
  ...
}
```

The sampling function must return a list of named lists, containing the sampled hyperparameter configurations. For instance, the structure of the return value of our sampling function for an arbitrary example should look like this:

```
str(sample.fun(par.set = mySearchSpace, n.configs = 2))
```

3: the initialization function

We do also need a function to initialize our models.

The inputs of that function must include:

- **r**: the amount of budget to initialize the model with
- **config**: a hyperparameter configuration
- **problem**: an object containing the data and if necessary a resampling rule

```
init.fun = function(r, config, problem) {
  ...
}
```

4: the training function

The training function takes an initialized model and continues the training process. Hyperband applies successive halving and thus eliminates a bunch of models. Instead of plainly training a new model from scratch, we rather continue training our existing model. That will save us a lot of time.

Our inputs are:

- **mod**: a model
- **budget**: the new budget allocation
- **problem**: an arbitrary object containing the data and if necessary a resampling rule

```
train.fun = function(mod, budget, problem) {
  ...
}
```

5: the performance function

Our final ingredient is the performance function. That function simply evaluates the performance of the model at its current state.

Inputs include:

- **model**: a model to evaluate
- **problem**: an arbitrary object containing the data and if necessary a resampling rule

```
performance.fun = function(model, problem) {
  ...
}
```

Now that we have defined these functions, we can finally call hyperband.

The inputs of hyperband are:

- **problem**: an arbitrary object containing the data and if necessary a resampling rule
- **max.resources**: the maximum amount of resource that can be allocated to a single configuration
  - the default is 81, that means in particular that we sample 81 configurations in our first bracket
- **prop.discard**: a control parameter to define the proportion of configurations that will be discarded in each round of successive halving
  - the default is 3, that means in particular that we eliminate 2/3 in each round of successive halving
- **max.perf**: a logical indicating whether we want to maximize (e.g. accuracy) or minimize (e.g. MSE) the performance measure
- **id**: a string generating a unique id for each model
- **par.set**: the hyperparameter search space
- **sample.fun**: the sampling function
- **train.fun**: the training function
- **performance.fun**: the performance function

```
hyperhyper = hyperband(
  problem = myProblem,
  max.resources = 81,
  prop.discard = 3,
  max.perf = TRUE or FALSE,
  id = "my id",
```

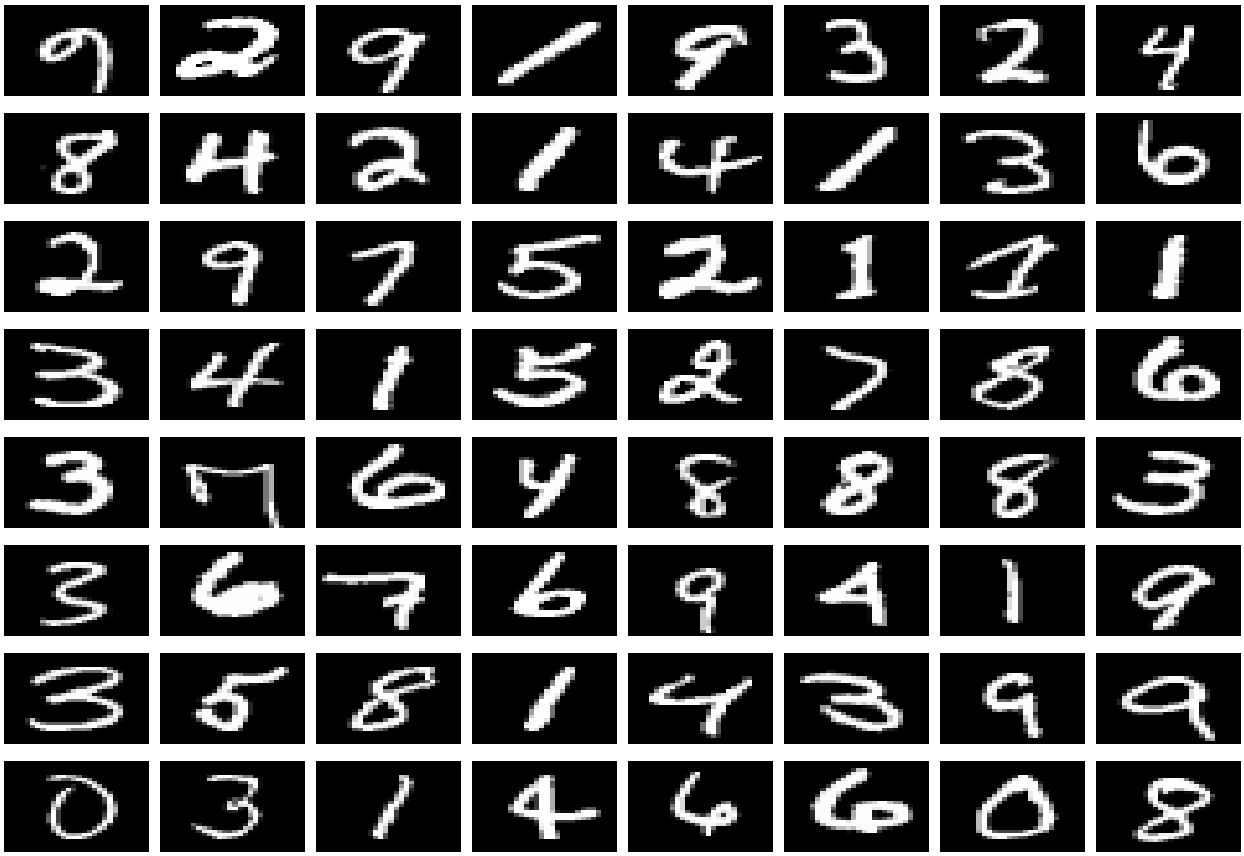
```
  par.set = mySearchSpace,
  sample.fun = sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun
)
```

We obtain according to the hyperband algorithm $floor(log_{prop.discard}(max.resources)) + 1$ brackets, which are all R6 objects. These objects contain a variety of methods, which will be discussed in the example section.

2. Example 1: hyperband to optimize a neural network with mxnet and mlr

If you are not familiar with mlr, please go to the appendix for a short introduction.

We would like to use a small subset of the original MNIST data (LeCun & Cortes 2010) and tune a neural network with hyperbandr.



Our data has 6000 observations, evenly distributed on 10 classes.

```
dim(mnist)
```

```
## [1] 6000  785
```

```
table(mnist$label)
```

```
##
##   0   1   2   3   4   5   6   7   8   9
## 600 600 600 600 600 600 600 600 600 600
```

Let us create a list, which we call *problem*. That list should contain the data, as well as a resampling rule.

```
# We sample 2/3 of our data for training:
train.set = sample(nrow(mnist), size = (2/3)*nrow(mnist))

# Another 1/6 will be used for validation during training:
```

```
val.set = sample(setdiff(1:nrow(mnist), train.set), 1000)

# The remaining 1/6 will be stored for testing:
test.set = setdiff(1:nrow(mnist), c(train.set, val.set))

# Since we use mlr, we define a classification task to encapsulate the data:
task = makeClassifTask(data = mnist, target = "label")

# Finally, we define the problem list:
problem = list(data = task, train = train.set, val = val.set, test = test.set)
```

2.1: the configuration space

The ParamHelpers package provides an easy way to construct the configuration space

```
library("ParamHelpers")
# We choose to search for optimal setting of the following hyperparameters:
configSpace = makeParamSet(
  makeDiscreteParam(id = "optimizer", values = c("sgd", "rmsprop", "adam", "adagrad")),
  makeNumericParam(id = "learning.rate", lower = 0.001, upper = 0.1),
  makeNumericParam(id = "wd", lower = 0, upper = 0.01),
  makeNumericParam(id = "dropout.input", lower = 0, upper = 0.6),
  makeNumericParam(id = "dropout.layer1", lower = 0, upper = 0.6),
  makeNumericParam(id = "dropout.layer2", lower = 0, upper = 0.6),
  makeNumericParam(id = "dropout.layer3", lower = 0, upper = 0.6),
  makeLogicalParam(id = "batch.normalization1"),
  makeLogicalParam(id = "batch.normalization2"),
  makeLogicalParam(id = "batch.normalization3")
)
```

2.2: the sampling function

Now we need a function to sample configurations from our search space.

```
sample.fun = function(par.set, n.configs, ...) {
  # Sample from the par.set and remove all NAs.
  lapply(sampleValues(par = par.set, n = n.configs), function(x) x[!is.na(x)])
}
```

2.3: the initialization function

This function initializes a convolutional neural network with two conv layers as well as two dense layers. Note that we define layers = 3, the second dense layer is our output and will be automatically created by mlr. We decide to choose epochs as resources. Thus, when initializing the model, we allocate **r** resources or **r** epochs.

```
init.fun = function(r, config, problem) {
  # We begin and create a learner.
  lrn = makeLearner("classif.mxff",
```

```
    # You have to install the gpu version of mxnet in order to run this code.
    ctx = mx.gpu(),
    layers = 3,
    conv.layer1 = TRUE, conv.layer2 = TRUE,
    conv.data.shape = c(28, 28),
    num.layer1 = 8, num.layer2 = 16, num.layer3 = 64,
    conv.kernel1 = c(3,3), conv.stride1 = c(1,1),
    pool.kernel1 = c(2,2), pool.stride1 = c(2,2),
    conv.kernel2 = c(3,3), conv.stride2 = c(1,1),
    pool.kernel2 = c(2,2), pool.stride2 = c(2,2),
    array.batch.size = 128,
    begin.round = 1, num.round = r,
    # This line is very important: here we allocate the configuration to our model.
    par.vals = config
  )
  # This will start the actual training (initialization) of the model.
  mod = train(learner = lrn, task = problem$data, subset = problem$train)
  return(mod)
}
```

2.4: the training function

That function will take the initialized model and continues the training process. To this, most importantly, we have to extract the weights from our initialized model and assign them to a new learner.

```
train.fun = function(mod, budget, problem) {
  # We create a new learner and assign all hyperparameters from our initialized model.
  lrn = makeLearner("classif.mxff", ctx = mx.gpu(), par.vals = mod$learner$par.vals)
  lrn = setHyperPars(lrn,
    # In addition, we have to extract the weights and feed them into our new model .
    symbol = mod$learner.model$symbol,
    arg.params = mod$learner.model$arg.params,
    aux.params = mod$learner.model$aux.params,
    begin.round = mod$learner$par.vals$begin.round + mod$learner$par.vals$num.round,
    num.round = budget
  )
  mod = train(learner = lrn, task = problem$data, subset = problem$train)
  return(mod)
}
```

2.5: the performance function

The performance function will simply predict the validation data at each step of successive halving.

```
performance.fun = function(model, problem) {
  pred = predict(model, task = problem$data, subset = problem$val)
  # We choose accuracy as our performance measure.
  performance(pred, measures = acc)
}
```

2.6: call hyperband

Now we can call hyperband (this needs around 5 minutes on a GTX 1070).

```
hyperhyper = hyperband(
  problem = problem,
  max.resources = 81,
  prop.discard = 3,
  max.perf = TRUE,
  id = "CNN",
  par.set = configSpace,
  sample.fun =  sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

```
## Beginning with bracket 4
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 3
## Iteration 0, with 34 Algorithms left (Budget: 3)
## Iteration 1, with 11 Algorithms left (Budget: 9)
## Iteration 2, with 3 Algorithms left (Budget: 27)
## Iteration 3, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 2
## Iteration 0, with 15 Algorithms left (Budget: 9)
## Iteration 1, with 5 Algorithms left (Budget: 27)
## Iteration 2, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 1
## Iteration 0, with 8 Algorithms left (Budget: 27)
## Iteration 1, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 0
## Iteration 0, with 1 Algorithms left (Budget: 81)
```

With max.resources = 81 and prop.discard = 3, we obtain a total of 5 brackets:

```
length(hyperhyper)
```

```
## [1] 5
```

We can inspect the first bracket ..

```
hyperhyper[[1]]
```

```
## <Bracket>
##   Public:
##     adjust: 27
##     B: 405
##     bracket.storage: BracketStorage, R6
##     clone: function (deep = FALSE)
##     configurations: list
```

```
##     filterTopKModels: function (k)
##     getBudgetAllocation: function ()
##     getNumberOfModelsToSelect: function ()
##     getPerformances: function ()
##     getTopKModels: function (k)
##     id: CNN
##     initialize: function (problem, max.perf, max.resources, prop.discard, s,
##     iteration: 4
##     max.perf: TRUE
##     max.resources: NULL
##     models: list
##     n.configs: 1
##     par.set: ParamSet
##     printState: function ()
##     prop.discard: 3
##     r.config: 1
##     run: function ()
##     s: 4
##     sample.fun: NULL
##     step: function ()
##     visPerformances: function (make.labs = TRUE, ...)
```

.. and for instance check it's performance by calling the getPerformance() method:

```
hyperhyper[[1]]$getPerformances()
```

```
## [1] 0.971
```

We can also inspect the architecture of the best model of bracket 1:

```
hyperhyper[[1]]$models[[1]]$model
```

```
## Model for learner.id=classif.mxff; learner.class=classif.mxff
## Trained on: task.id = mnist; obs = 4000; features = 784
## Hyperparameters: learning.rate=0.0442,array.layout=rowmajor,verbose=FALSE,optimizer=adagrad,wd=0.006
```

Now let's see which bracket yielded the best performance:

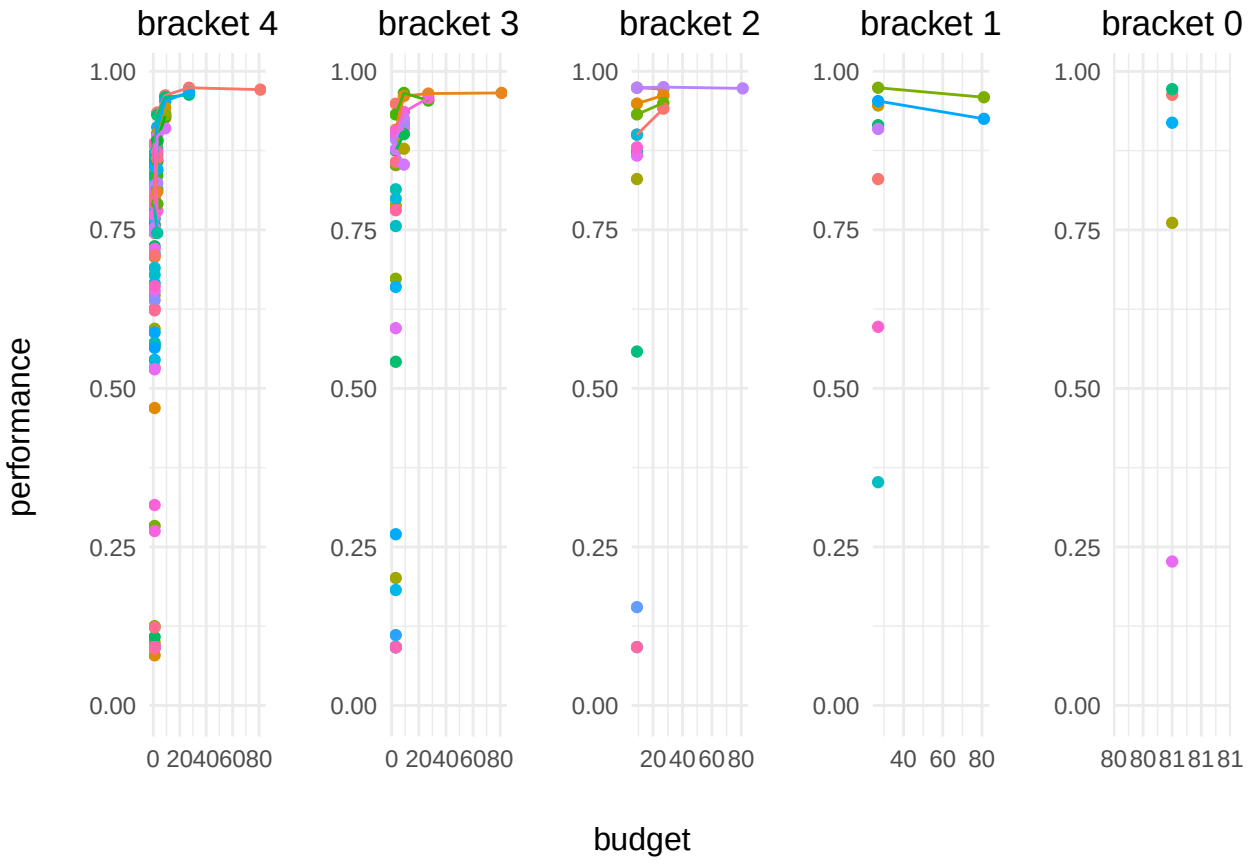
```
lapply(hyperhyper, function(x) x$getPerformances())
```

```
## [[1]]
## [1] 0.971
##
## [[2]]
## [1] 0.966
##
## [[3]]
## [1] 0.973
##
## [[4]]
## [1] 0.959
##
## [[5]]
```

```
## [1] 0.972
```

We can call the hyperVis function to visualize all brackets:

```
hyperVis(hyperhyper)
```



Let us use the best model over all brackets and predict the test data:

```
# Extract the best model:
best.mod.index = which.max(unlist(lapply(hyperhyper, function(x) x$getPerformances())))
best.mod = hyperhyper[[best.mod.index]]$models[[1]]$model

# Predict the test data:
performance(predict(object = best.mod, task = problem$data, subset = problem$test),
            measures = acc)
```

```
##   acc
## 0.977
```

2.7: additional features

The hyperbandr package can also compute single bracket objects. For demonstration purposes we shrink our hyperparameter search space. Computing single bracket objects requires us to input some new parameters:

- **s**: the s'th bracket which we would like to compute
- **B**: the (approximate) total amount of resources, that will be spend in that bracket
  - the formula from the hyperband paper to compute B is B = (sMax + 1) * max.resources

```
# Smaller config space for demonstration purposes.
configSpace = makeParamSet(
  makeDiscreteParam(id = "optimizer", values = c("sgd", "adam")),
  makeNumericParam(id = "learning.rate", lower = 0.001, upper = 0.1),
  makeLogicalParam(id = "batch.normalization"))
```

```
brack = bracket$new(
  problem = problem,
  max.perf = TRUE,
  max.resources = 81,
  prop.discard = 3,
  s = 4,
  B = (4 + 1)*81,
  id = "nnet_bracket",
  par.set = configSpace,
  sample.fun = sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

Each bracket object has a bracket storage object which is basically just another R6 class. The bracket storage shows us the hyperparameters, the current budget and the performance in an equation-ish style.

```
## [1] 81  5
```

```
##    optimizer learning.rate batch.normalization current_budget     y
## 1        sgd   0.092580048                TRUE              1 0.399
## 2        sgd   0.073175152               FALSE              1 0.092
## 3       adam   0.056124215               FALSE              1 0.092
## 4        sgd   0.065107343                TRUE              1 0.250
## 5        sgd   0.083932671               FALSE              1 0.092
## 6        sgd   0.088801177                TRUE              1 0.503
## 7        sgd   0.008158432                TRUE              1 0.092
## 8        sgd   0.031451005                TRUE              1 0.126
## 9        sgd   0.033258030                TRUE              1 0.170
## 10      adam   0.032635399               FALSE              1 0.870
```

We call the step() method to conduct one round of successive halving. Or just complete the bracket by calling the run() method. That means we conduct successive halving according to the rules described in the hyperband paper until only one configuration is left.

```
brack$run()
```

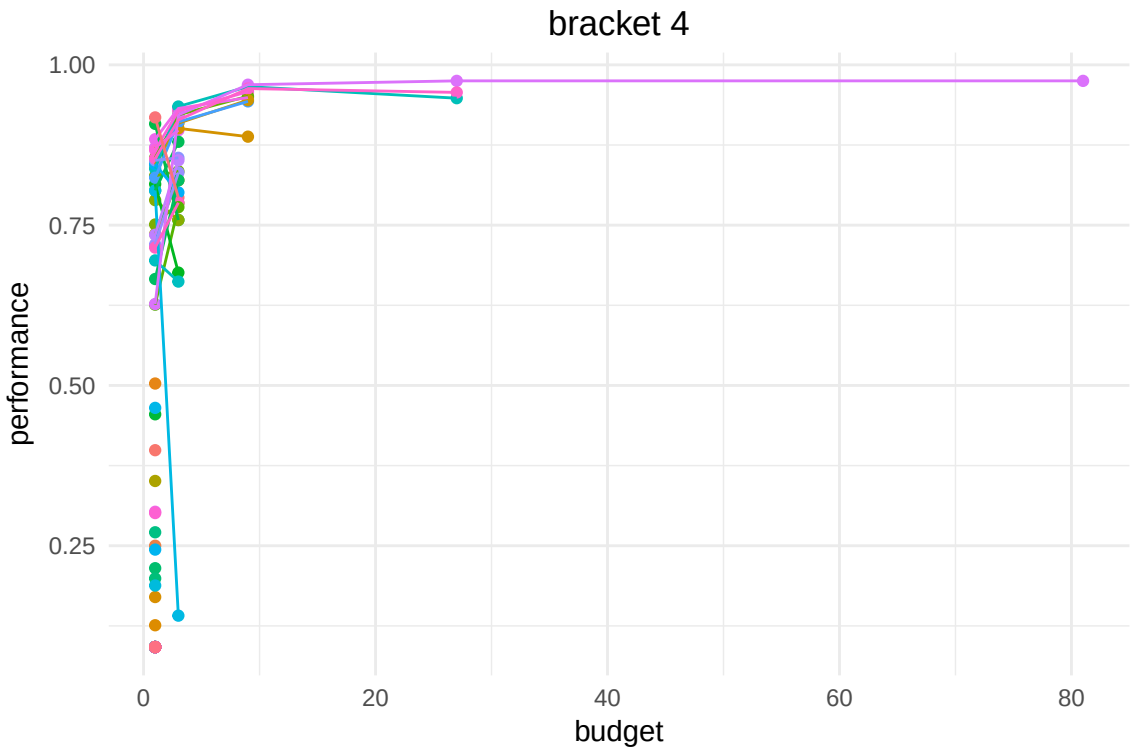
```
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
```

While we call the run() method, we continuously write new lines to our bracket storage object.

```
## [1] 121   5
```

Bracket objects have a visPerformances() method to immediately visualize the bracket.

```
brack$visPerformances()
```



Beside the graphic investigation, we can also extract the best models performance by simply calling the getPerformance() method.

```
brack$getPerformances()
```

```
## [1] 0.975
```

Each bracket object contains multiple algorithm objects. The hyperbandr package allows us to create these algorithm objects soley and manipulate them. The input values are almost identical to those seen in the bracket object or when calling hyperband.

```
set.seed(1337)
myConfig = sample.fun(par.set = configSpace, n.configs = 1)[[1]]

obj = algorithm$new(
  problem = problem,
  id = "nnet",
  configuration = myConfig,
  initial.budget = 1,
```

```
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

We can inspect architecture of our algorithm object by calling configuration:

```
# You can also call obj$model for much more details, but that would not fit on the page.
obj$configuration
```

```
## $optimizer
## [1] "adam"
##
## $learning.rate
## [1] 0.05690947
##
## $batch.normalization
## [1] TRUE
```

Similar to the bracket object, each algorithm object has a algorithm storage object which is basically just another R6 class. The algorithm storage shows us the hyperparameters, the current budget and the performance in an equation-ish style.

```
obj$algorithm.result$data.matrix
```

```
##   optimizer learning.rate batch.normalization current_budget     y
## 1      adam    0.05690947                TRUE              1 0.826
```

The algorithm object does also have a getPerformance() method.

```
obj$getPerformance()
```

```
##   acc
## 0.826
```

By calling the continue() method, we can continue training our algorithm object by an arbitrary amount of budget (here: epochs).

```
obj$continue(1)
```

Like before, in each step we write new lines to our algorithm object. That enables us so track the behaviour of our algorithm object when allocating more resources.

```
obj$algorithm.result$data.matrix
```

```
##   optimizer learning.rate batch.normalization current_budget     y
## 1      adam    0.05690947                TRUE              1 0.826
## 2      adam    0.05690947                TRUE              2 0.913
```

So let us call continue(1) for 18 times to obtain a total of 20 iterations.

```
invisible(capture.output(replicate(18, obj$continue(1))))
```

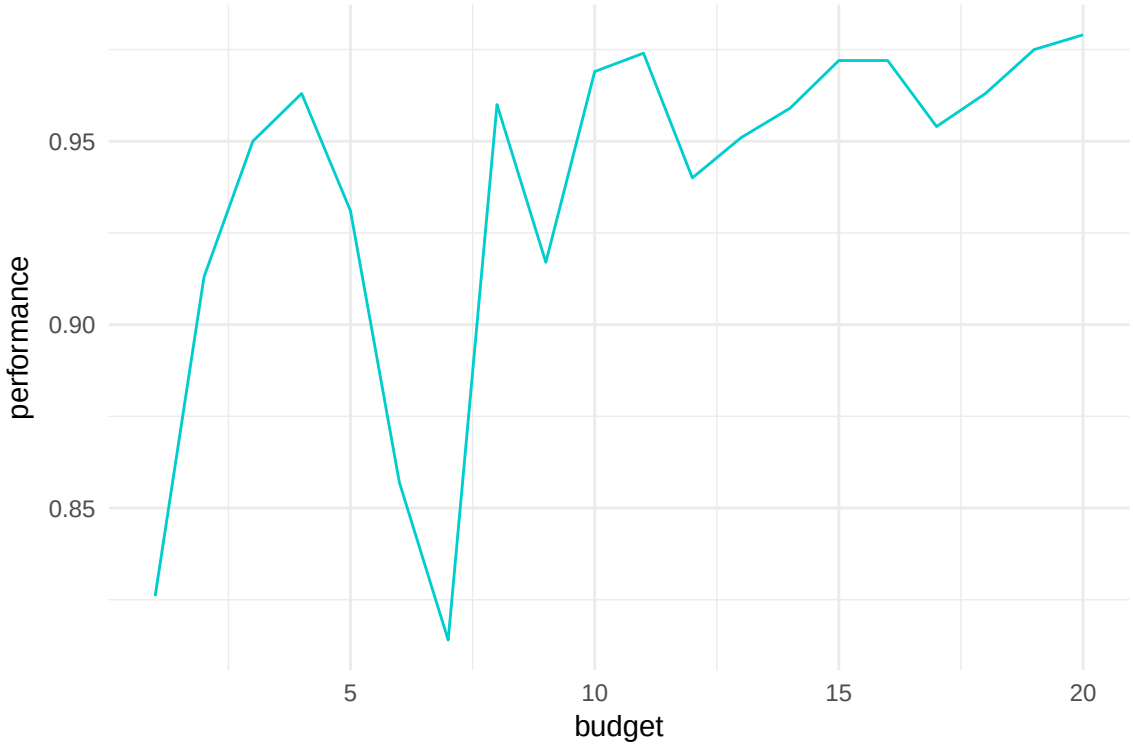
This will write 18 additional lines to our algorithm storage:

```
obj$algorithm.result$data.matrix
```

```
##    optimizer learning.rate batch.normalization current_budget     y
## 1       adam    0.05690947                 TRUE              1 0.826
## 2       adam    0.05690947                 TRUE              2 0.913
## 3       adam    0.05690947                 TRUE              3 0.950
## 4       adam    0.05690947                 TRUE              4 0.963
## 5       adam    0.05690947                 TRUE              5 0.931
## 6       adam    0.05690947                 TRUE              6 0.857
## 7       adam    0.05690947                 TRUE              7 0.814
## 8       adam    0.05690947                 TRUE              8 0.960
## 9       adam    0.05690947                 TRUE              9 0.917
## 10      adam    0.05690947                 TRUE             10 0.969
## 11      adam    0.05690947                 TRUE             11 0.974
## 12      adam    0.05690947                 TRUE             12 0.940
## 13      adam    0.05690947                 TRUE             13 0.951
## 14      adam    0.05690947                 TRUE             14 0.959
## 15      adam    0.05690947                 TRUE             15 0.972
## 16      adam    0.05690947                 TRUE             16 0.972
## 17      adam    0.05690947                 TRUE             17 0.954
## 18      adam    0.05690947                 TRUE             18 0.963
## 19      adam    0.05690947                 TRUE             19 0.975
## 20      adam    0.05690947                 TRUE             20 0.979
```

To visualize the training process and the development of our validation error, we simply call the visPerformance() method:

```
obj$visPerformance()
```


performance
0.95
0.90
0.85
5
10
15
20
budget

3. Example 2: hyperband in combination with MBO to optimize a neural network with mxnet, mlr and mlrMBO

Recall the bracket storage object of example 1:

Each bracket has a bracket storage, containing all configurations in that bracket as well as their corresponding performance values, e.g.:

```
head(brack$bracket.storage$data.matrix)
```

```
##   optimizer learning.rate batch.normalization current_budget     y
## 1       sgd    0.09258005                 TRUE              1 0.399
## 2       sgd    0.07317515                FALSE              1 0.092
## 3      adam    0.05612421                FALSE              1 0.092
## 4       sgd    0.06510734                 TRUE              1 0.250
## 5       sgd    0.08393267                FALSE              1 0.092
## 6       sgd    0.08880118                 TRUE              1 0.503
```

At each step of successive halving, we write new lines to the bracket storage object. Consequently, configurations which survived one step of successive halving occur at least two times in the bracket storage.

When we call hyperband, another R6 class called hyper storage is automatically being created. That hyper storage object takes bracket storage objects and concatenates them. Thus, the hyper storage contains the accumulated information of all configurations over all brackets, which have been computed so far.

For instance, if we begin with the third bracket, the hyper storage object contains all configurations and performance values of the first and the second bracket.

Instead of random sampling configurations in the third bracket, we could exploit the information in the hyper storage object to propose new configurations in a model based fashion (MBO).

To this we simply have to adjust our sampling function.

One potential implementation could look like this:

```
library("mlrMBO")
library("ranger")

sample.fun.mbo = function(par.set, n.configs, hyper.storage) {
  # if the hyper storage is empty, sample from our search space
  if (dim(hyper.storage)[[1]] == 0) {
    lapply(sampleValues(par = par.set, n = n.configs), function(x) x[!is.na(x)])
  # else, propose configurations via MBO
  } else {
    catf("Proposing points")
    ctrl = makeMBOControl(propose.points = n.configs)
    # set the infill criterion
    ctrl = setMBOControlInfill(ctrl, crit = crit.cb)
    designMBO = data.table(hyper.storage)
    # we have to keep in mind, that some configurations occur multiple times,
    # thus we have to aggregate their performance according to some rule:
    # here we simply do this by electing the best performance that we observe
    designMBO = data.frame(designMBO[, max(y), by = names(configSpace$pars)])
    colnames(designMBO) = colnames(hyper.storage)[-(length(configSpace$pars) + 1)]
    # initSMBO enables us to conduct human-in-the-loop MBO
    opt.state = initSMBO(
```

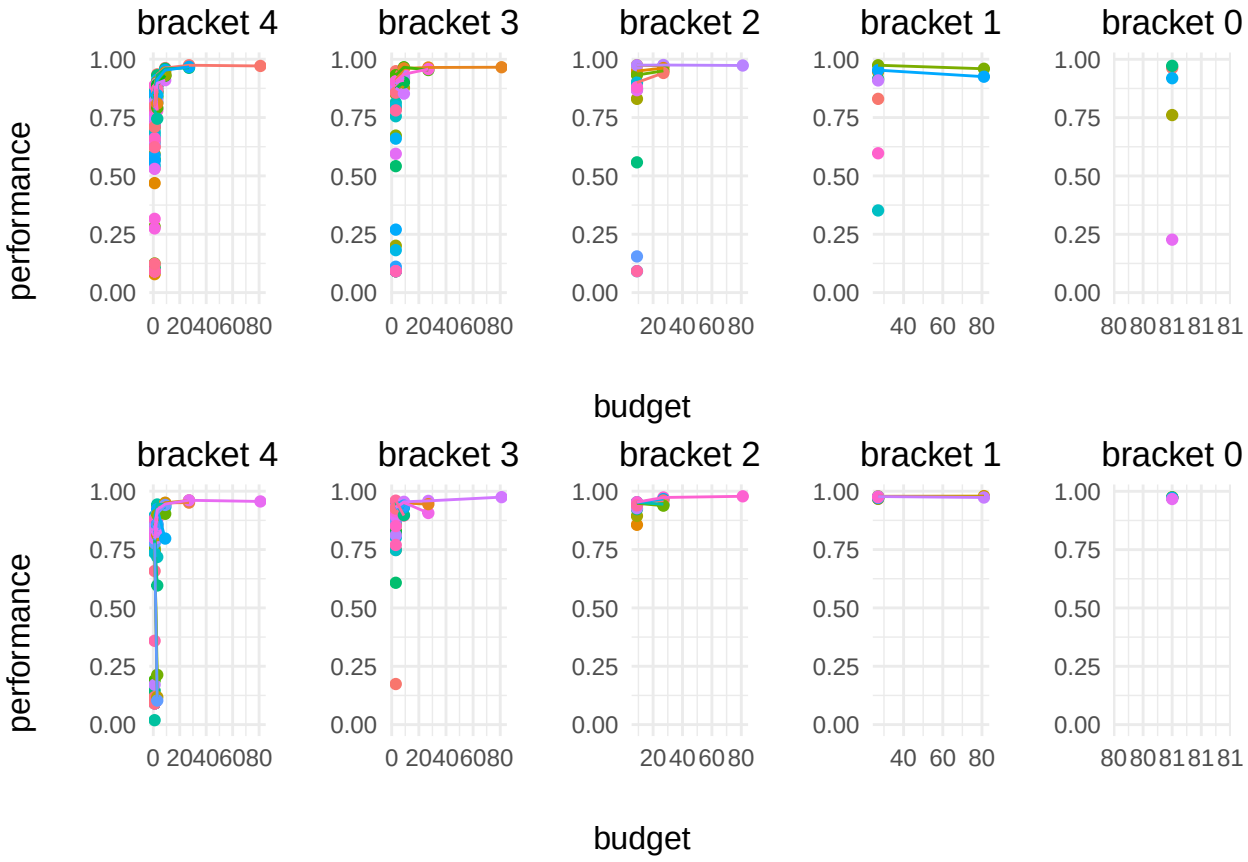
```
      par.set = configSpace,
      design = designMBO,
      control = ctrl,
      minimize = FALSE,
      noisy = FALSE)
    # based on the surrogate model, proposePoints yields us our configurations
    prop = proposePoints(opt.state)
    propPoints = prop$prop.points
    rownames(propPoints) = c()
    propPoints = convertRowsToList(propPoints, name.list = FALSE, name.vector = TRUE)
    return(propPoints)
  }
}
```

Now we simply run hyperband with the new sampling function

```
hyperhyperMBO = hyperband(
  problem = problem,
  max.resources = 81,
  prop.discard = 3,
  max.perf = TRUE,
  id = "CNN",
  par.set = configSpace,
  sample.fun =  sample.fun.mbo,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

```
## Beginning with bracket 4
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 3
## Proposing points
## Iteration 0, with 34 Algorithms left (Budget: 3)
## Iteration 1, with 11 Algorithms left (Budget: 9)
## Iteration 2, with 3 Algorithms left (Budget: 27)
## Iteration 3, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 2
## Proposing points
## Iteration 0, with 15 Algorithms left (Budget: 9)
## Iteration 1, with 5 Algorithms left (Budget: 27)
## Iteration 2, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 1
## Proposing points
## Iteration 0, with 8 Algorithms left (Budget: 27)
## Iteration 1, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 0
## Proposing points
## Iteration 0, with 1 Algorithms left (Budget: 81)
```

Let us compare the results of our vanilla hyperband and the combination of hyperband with MBO:



text

text

4. Example 3: hyperband to optimize a gradient boosting model with xgboost and mlr

```
train.set = xgb.DMatrix(agaricus.train$data, label = agaricus.train$label)
test.set = xgb.DMatrix(agaricus.test$data, label = agaricus.test$label)

problem = list(train = train.set, val = test.set)
```

```
rm(train.set)
rm(test.set)
```

text

```
configSpace = makeParamSet(
  makeIntegerParam("max_depth", lower = 3, upper = 15, default = 3),
  makeNumericParam("colsample_bytree", lower = 0.3, upper = 1, default = 0.6),
  makeNumericParam("subsample", lower = 0.3, upper = 1, default = 0.6)
)
```

text

```
sample.fun = function(par.set, n.configs, ...) {
  lapply(sampleValues(par = par.set, n = n.configs), function(x) x[!is.na(x)])
}
```

text

```
init.fun = function(r, config, problem) {
  watchlist = list(eval = problem$val, train = problem$train)
  capture.output({mod = xgb.train(config, problem$train, nrounds = r, watchlist, verbose = 1)})
  return(mod)
}
```

text

```
train.fun = function(mod, budget, problem) {
  watchlist = list(eval = problem$val, train = problem$train)
  capture.output({mod = xgb.train(xgb_model = mod,
    nrounds = budget, params = mod$params, problem$train, watchlist, verbose = 1)})
  return(mod)
}
```

text

```
performance.fun = function(model, problem) {
  tail(model$evaluation_log$eval_rmse, n = 1)
}
```

text

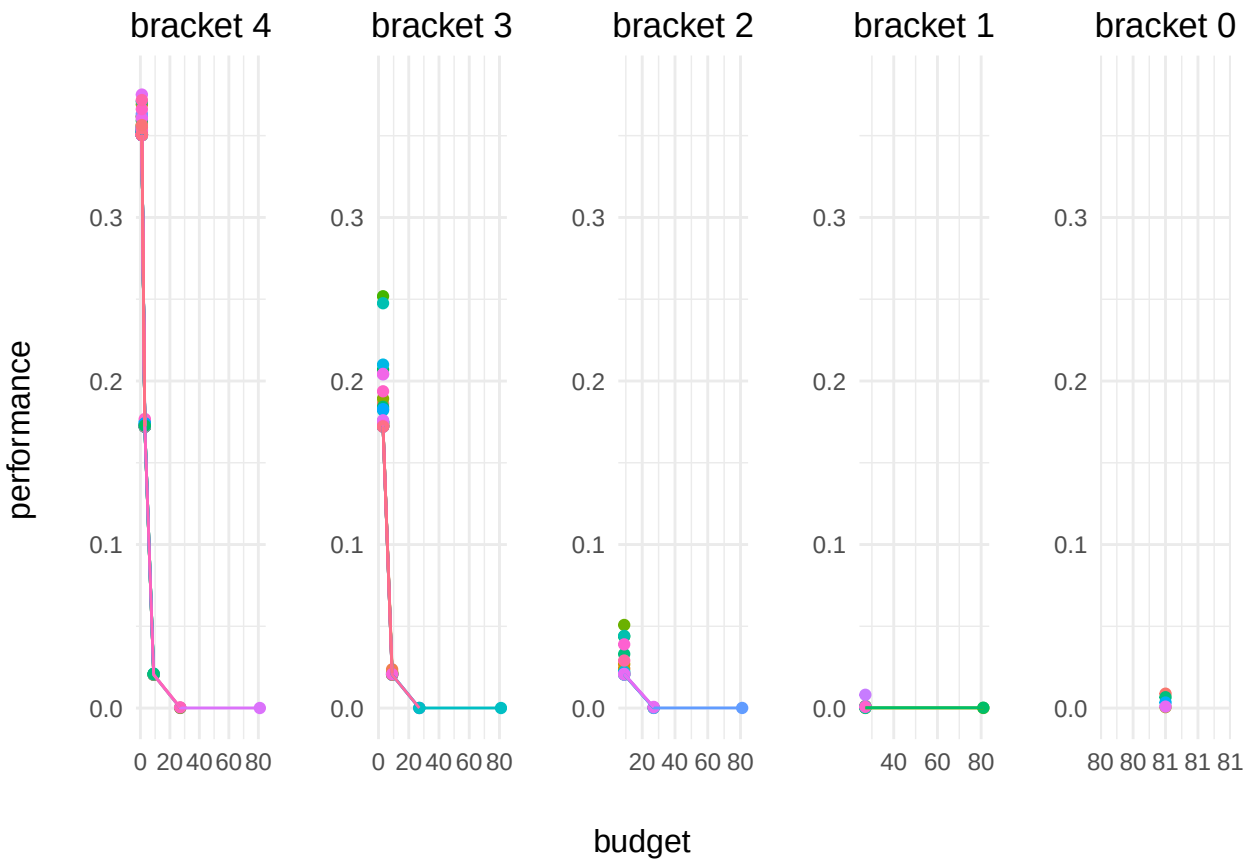
```r
hyperhyper = hyperband(
  problem = problem,
  max.resources = 81,
  prop.discard = 3,
  max.perf = FALSE,
  id = "xgboost",
  par.set = configSpace,
  sample.fun =  sample.fun,
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

```
## Beginning with bracket 4
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 3
## Iteration 0, with 34 Algorithms left (Budget: 3)
## Iteration 1, with 11 Algorithms left (Budget: 9)
## Iteration 2, with 3 Algorithms left (Budget: 27)
## Iteration 3, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 2
## Iteration 0, with 15 Algorithms left (Budget: 9)
## Iteration 1, with 5 Algorithms left (Budget: 27)
## Iteration 2, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 1
## Iteration 0, with 8 Algorithms left (Budget: 27)
## Iteration 1, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 0
## Iteration 0, with 1 Algorithms left (Budget: 81)
```

text

```r
hyperVis(hyperhyper)
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



text

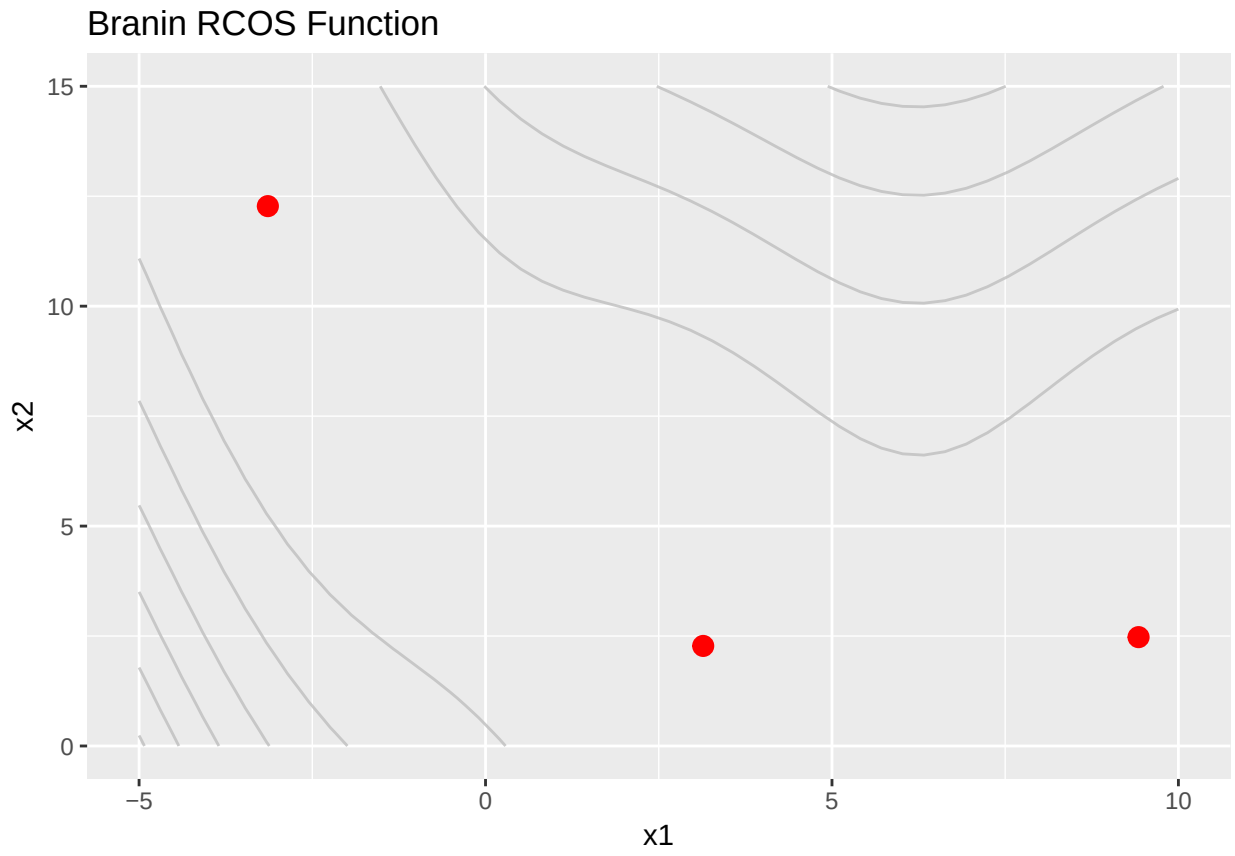
```
lapply(hyperhyper, function(x) x$getPerformances())
```

```
## [[1]]
## [1] 4.5e-05
##
## [[2]]
## [1] 4.7e-05
##
## [[3]]
## [1] 7.9e-05
##
## [[4]]
## [1] 6.6e-05
##
## [[5]]
## [1] 0.000584
```

5. Example 4: hyperband to optimize a function with smoof

```
library("smoof")
braninProb = makeBraninFunction()
```

text




```
## Single-objective function
## Name: Branin RCOS Function
## Description: no description
## Tags: single-objective, continuous, differentiable, non-separable, non-scalable, multimodal
## Noisy: FALSE
## Minimize: TRUE
## Constraints: TRUE
## Number of parameters: 2
##            Type len Def        Constr Req Tunable Trafo
## x numericvector   2   - -5,0 to 10,15   -    TRUE     -
## Global optimum objective value of 0.3979 at
##          x1     x2
## 1 -3.141593 12.275
## 2  3.141593  2.275
## 3  9.424778  2.475
```

```
getParamSet(braninProb)
```

```
##            Type len Def        Constr Req Tunable Trafo
```

```
## x numericvector   2   - -5,0 to 10,15   -    TRUE     -
```

text

```
configSpace = makeParamSet(
    makeNumericParam(id = "x1", lower = -5, upper = 10.1))
```

text

```
sample.fun = function(par.set, n.configs, ...) {
  sampleValues(par = par.set, n = n.configs)
}
```

text

```
init.fun = function(r, config, problem) {
  x1 = unname(unlist(config))
  x2 = runif(1, 0, 15)
  mod = c(x1, x2)
  return(mod)
}
```

text

```
train.fun = function(mod, budget, problem) {
  for(i in seq_len(budget)) {
    mod.new = c(mod[[1]], mod[[2]] + rnorm(1, sd = 3))
    if(performance.fun(mod.new) < performance.fun(mod))
      mod = mod.new
  }
  return(mod)
}
```

text

```
performance.fun = function(model, problem) {
  braninProb(c(model[[1]], model[[2]]))
}
```

text

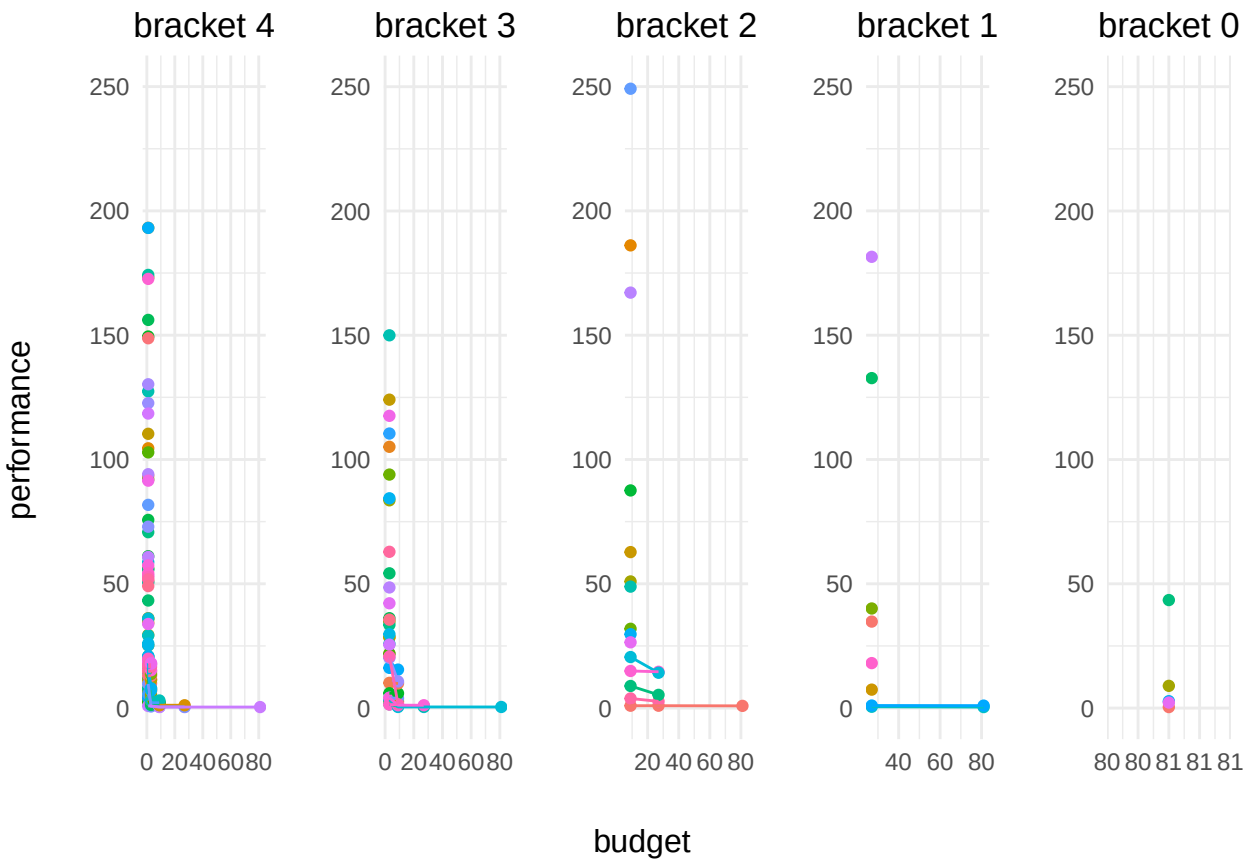
```
hyperhyper = hyperband(
  problem = braninProb,
  max.resources = 81,
  prop.discard = 3,
  max.perf = FALSE,
  id = "branin",
  par.set = configSpace,
  sample.fun =  sample.fun,
```

```
  init.fun = init.fun,
  train.fun = train.fun,
  performance.fun = performance.fun)
```

```
## Beginning with bracket 4
## Iteration 0, with 81 Algorithms left (Budget: 1)
## Iteration 1, with 27 Algorithms left (Budget: 3)
## Iteration 2, with 9 Algorithms left (Budget: 9)
## Iteration 3, with 3 Algorithms left (Budget: 27)
## Iteration 4, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 3
## Iteration 0, with 34 Algorithms left (Budget: 3)
## Iteration 1, with 11 Algorithms left (Budget: 9)
## Iteration 2, with 3 Algorithms left (Budget: 27)
## Iteration 3, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 2
## Iteration 0, with 15 Algorithms left (Budget: 9)
## Iteration 1, with 5 Algorithms left (Budget: 27)
## Iteration 2, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 1
## Iteration 0, with 8 Algorithms left (Budget: 27)
## Iteration 1, with 1 Algorithms left (Budget: 81)
## Beginning with bracket 0
## Iteration 0, with 1 Algorithms left (Budget: 81)
```

text

```
hyperVis(hyperhyper)
```



text

```
lapply(hyperhyper, function(x) x$getPerformances())
```

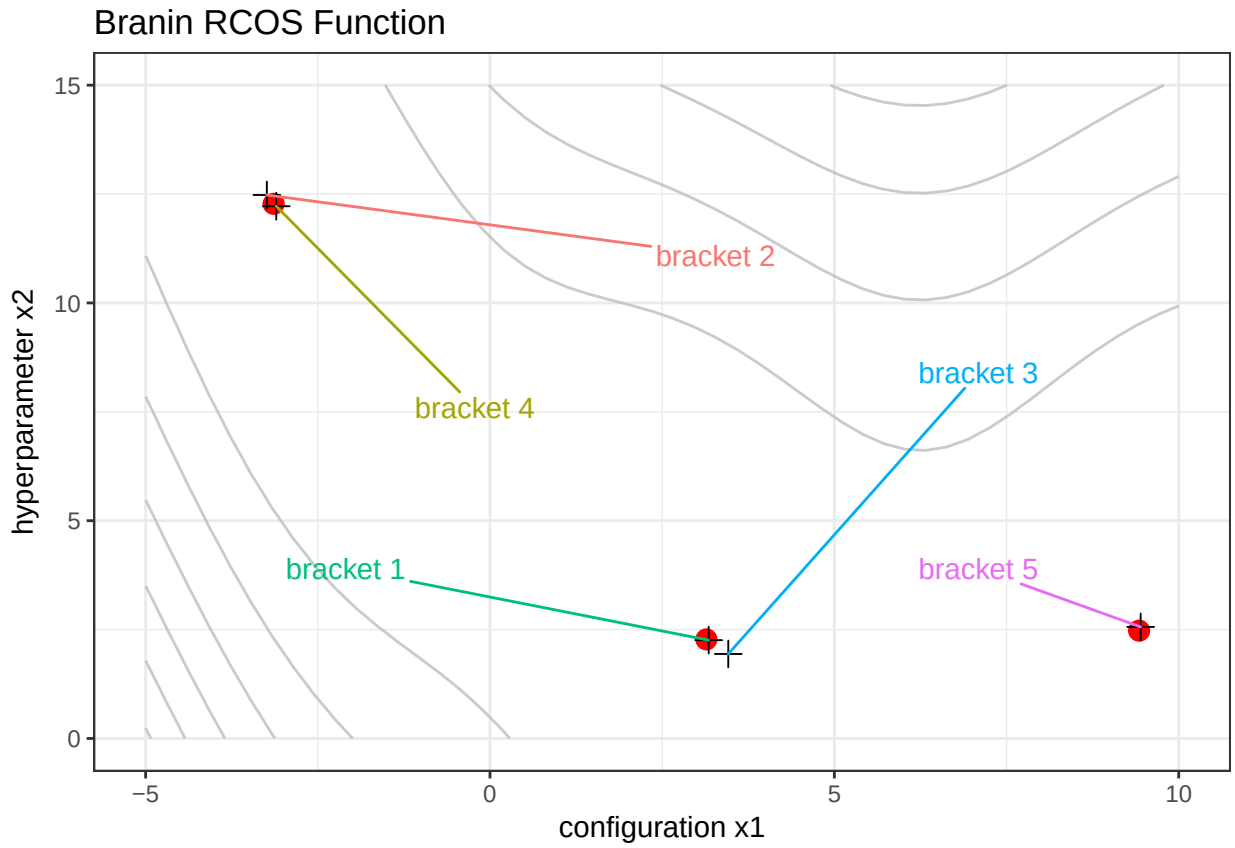
```
## [[1]]
## [1] 0.403344
##
## [[2]]
## [1] 0.4456938
##
## [[3]]
## [1] 0.8893165
##
## [[4]]
## [1] 0.4061799
##
## [[5]]
## [1] 0.4047745
```

text

```
results = lapply(hyperhyper, function(x) x$models[[1]]$model)
data = data.frame(matrix(unlist(results), ncol = 2, byrow = TRUE))
rownames(data) = c("bracket 1", "bracket 2", "bracket 3", "bracket 4", "bracket 5")
```

```r
colnames(data) = c("x1", "x2")

(vis = vis +
  geom_point(data = data, mapping = aes(x = x1, y = x2), shape = 3, size = 3) +
  geom_text_repel(data = data,
                  mapping = aes(x = x1, y = x2, color = factor(x1)),
                  label = rownames(data),
                  max.iter = 10000,
                  force = 3,
                  size = 4,
                  box.padding = unit(5, "lines")) +
  theme_bw() +
  theme(legend.position = "none")) +
  scale_x_continuous(name = "configuration x1") +
  scale_y_continuous(name = "hyperparameter x2")
```



text

6. appendix: introduction to mlr