Iterated F-Racing for mixed spaces and dependencies

Table of Contents

- Tuning across whole model spaces with ModelMultiplexer
- Multi-criteria evaluation and optimization

Source: `vignettes/tutorial/advanced_tune.Rmd` (https://github.com/mlr-org/mlr/blob/master/vignettes/tutorial/advanced_tune.Rmd)

The package supports a larger number of tuning algorithms, which can all be looked up and selected via `TuneControl() (../../reference/TuneControl.html)`. One of the cooler algorithms is iterated F-racing from the `irace::irace() (http://www.rdocumentation.org/packages/irace/topics/irace)` package (technical description here (http://iridia.ulb.ac.be/IridiaTrSeries/link/IridiaTr2011-004.pdf)). This not only works for arbitrary parameter types (numeric, integer, discrete, logical), but also for so-called dependent / hierarchical parameters:

```
ps = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeDiscreteParam("kernel", values = c("vanilladot", "polydot", "rbfdot")),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x,
    requires = quote(kernel == "rbfdot")),
  makeIntegerParam("degree", lower = 2L, upper = 5L,
    requires = quote(kernel == "polydot"))
)
ctrl = makeTuneControlIrace (../../reference/makeTuneControlIrace.html)(maxExper
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("Holdout")
res = tuneParams (../../reference/tuneParams.html)("classif.ksvm", iris.task, rde
  show.info = FALSE)
df = as.data.frame(res$opt.path)
print(head(df[, -ncol(df)]))
##           C     kernel    sigma degree mmce.test.mean dob eol
## 1  -3.877823    polydot       NA      2           0.02   1  NA
## 2   9.665740 vanilladot       NA     NA           0.06   1  NA
## 3   7.951264    polydot       NA      3           0.10   1  NA
## 4   1.699949    polydot       NA      2           0.06   1  NA
## 5 -11.033144     rbfdot 8.088702     NA           0.72   1  NA
## 6  -6.076261 vanilladot       NA     NA           0.14   1  NA
##   error.message
## 1          <NA>
## 2          <NA>
## 3          <NA>
## 4          <NA>
## 5          <NA>
## 6          <NA>
```

See how we made the kernel parameters like `sigma` and `degree` dependent on the `kernel` selection parameters? This approach allows you to tune parameters of multiple kernels at once, efficiently concentrating on the ones which work best for your given data set.

Tuning across whole model spaces with ModelMultiplexer

We can now take the following example even one step further. If we use the `makeModelMultiplexer() (../../reference/makeModelMultiplexer.html)` we can tune over different model classes at once, just as we did with the SVM kernels above.

```
base.learners = list(
  makeLearner (../../reference/makeLearner.html)("classif.ksvm"),
  makeLearner (../../reference/makeLearner.html)("classif.randomForest")
)
lrn = makeModelMultiplexer (../../reference/makeModelMultiplexer.html)(base.learn
```

Function `makeModelMultiplexerParamSet()` `(../../reference/makeModelMultiplexerParamSet.html)` offers a simple way to construct a parameter set for tuning: The parameter names are prefixed automatically and the `requires` element is set, too, to make all parameters subordinate to `selected.learner`.

```
ps = makeModelMultiplexerParamSet (../../reference/makeModelMultiplexerParamSet.h
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeIntegerParam("ntree", lower = 1L, upper = 500L)
)
print(ps)
##                              Type len Def
## selected.learner          discrete   -   -
## classif.ksvm.sigma         numeric   -   -
## classif.randomForest.ntree  integer   -   -
##                                                         Constr Req Tunable
## selected.learner          classif.ksvm,classif.randomForest   -    TRUE
## classif.ksvm.sigma                                  -12 to 12   Y    TRUE
## classif.randomForest.ntree                           1 to 500   Y    TRUE
##                           Trafo
## selected.learner              -
## classif.ksvm.sigma            Y
## classif.randomForest.ntree     -

rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 2L
ctrl = makeTuneControlIrace (../../reference/makeTuneControlIrace.html)(maxExperi
res = tuneParams (../../reference/tuneParams.html)(lrn, iris.task, rdesc, par.set
  show.info = FALSE)
df = as.data.frame(res$opt.path)
print(head(df[, -ncol(df)]))
##       selected.learner classif.ksvm.sigma classif.randomForest.ntree
## 1 classif.randomForest                 NA                        253
## 2 classif.randomForest                 NA                        265
## 3 classif.randomForest                 NA                        124
## 4         classif.ksvm          -5.804419                         NA
## 5 classif.randomForest                 NA                        234
## 6         classif.ksvm          -3.147588                         NA
##   mmce.test.mean dob eol error.message
## 1     0.06000000   1  NA          <NA>
## 2     0.06666667   1  NA          <NA>
## 3     0.06000000   1  NA          <NA>
## 4     0.14000000   1  NA          <NA>
## 5     0.06000000   1  NA          <NA>
## 6     0.06000000   1  NA          <NA>
```

Multi-criteria evaluation and optimization

During tuning you might want to optimize multiple, potentially conflicting, performance measures simultaneously.

In the following example we aim to minimize both, the false positive and the false negative rates (`fpr` and `fnr`). We again tune the hyperparameters of an SVM (function `kernlab::ksvm() (http://www.rdocumentation.org/packages/kernlab/topics/ksvm)`) with a radial basis kernel and use `sonar.task() (../../reference/sonar.task.html)` for illustration. As search strategy we choose a random search.

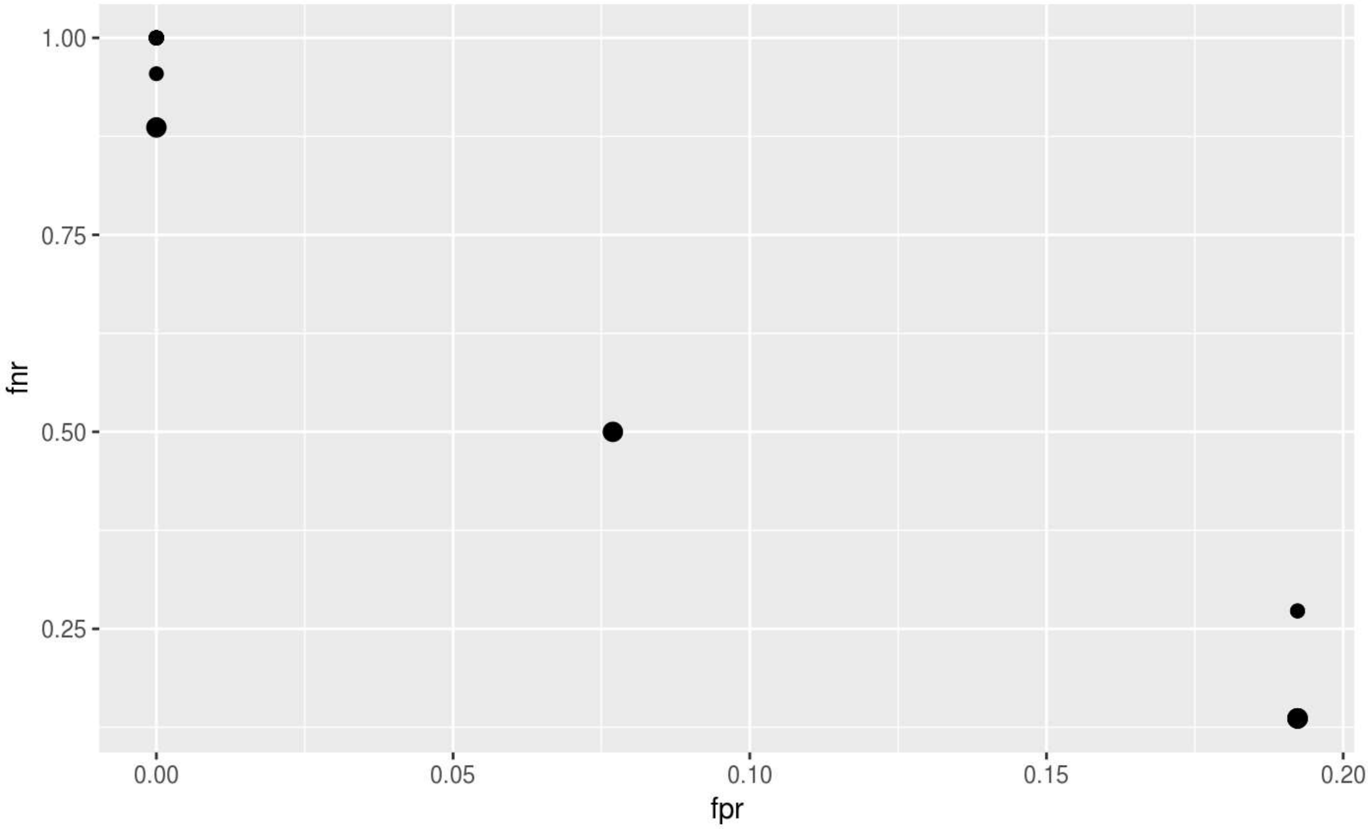
For all available multi-criteria tuning algorithms see `TuneMultiCritControl() (../../reference/TuneMultiCritControl.html)`.

```
ps = makeParamSet(
  makeNumericParam("C", lower = -12, upper = 12, trafo = function(x) 2^x),
  makeNumericParam("sigma", lower = -12, upper = 12, trafo = function(x) 2^x)
)
ctrl = makeTuneMultiCritControlRandom (../../reference/TuneMultiCritControl.html
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("Holdout")
res = tuneParamsMultiCrit (../../reference/tuneParamsMultiCrit.html)("classif.ksv
  resampling = rdesc, par.set = ps,
  measures = list(fpr, fnr), control = ctrl, show.info = FALSE)
res
## Tune multicrit result:
## Points on front: 5

print(head(df[, -ncol(df)]))
##       selected.learner classif.ksvm.sigma classif.randomForest.ntree
## 1 classif.randomForest                 NA                        253
## 2 classif.randomForest                 NA                        265
## 3 classif.randomForest                 NA                        124
## 4         classif.ksvm          -5.804419                         NA
## 5 classif.randomForest                 NA                        234
## 6         classif.ksvm          -3.147588                         NA
##   mmce.test.mean dob eol error.message
## 1     0.06000000   1  NA          <NA>
## 2     0.06666667   1  NA          <NA>
## 3     0.06000000   1  NA          <NA>
## 4     0.14000000   1  NA          <NA>
## 5     0.06000000   1  NA          <NA>
## 6     0.06000000   1  NA          <NA>
```

The results can be visualized with function `plotTuneMultiCritResult() (../../reference/plotTuneMultiCritResult.html)`. The plot shows the false positive and false negative rates for all parameter settings evaluated during tuning. Points on the Pareto front are slightly increased.

```
plotTuneMultiCritResult (../../reference/plotTuneMultiCritResult.html)(res)
```



1.00
0.75
0.50
0.25
fnr
0.00
0.05
0.10
0.15
0.20
fpr