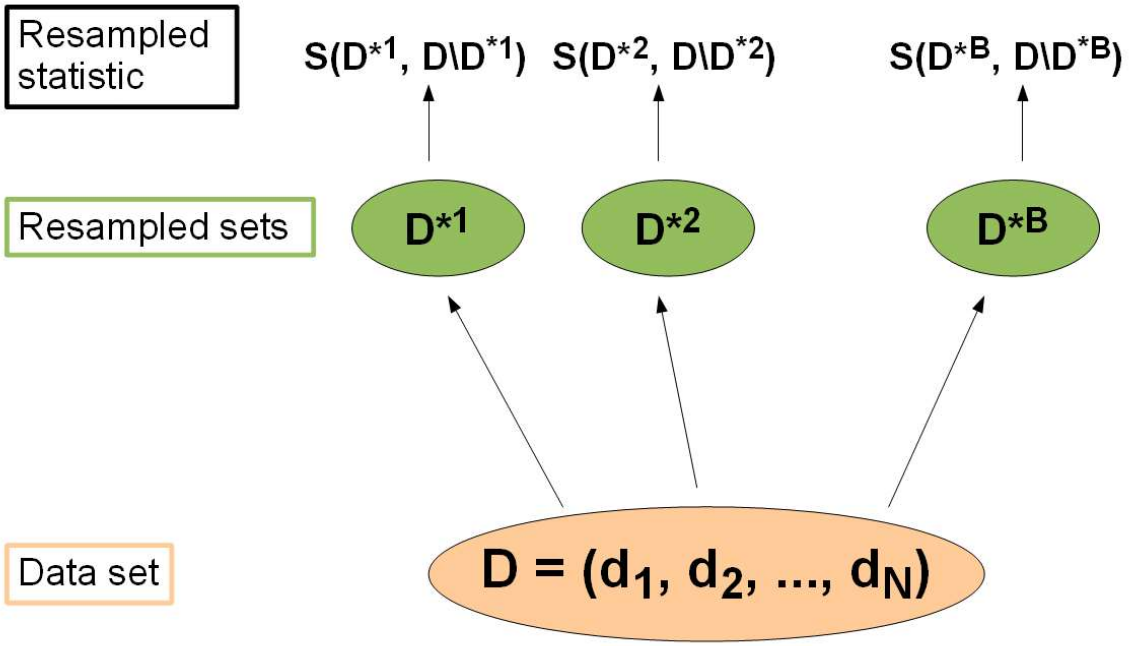
Resampling

Table of Contents

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

Resampling strategies are usually used to assess the performance of a learning algorithm: The entire data set is (repeatedly) split into training sets D^{*b} and test sets $D \setminus D^{*b}$, $b = 1, \ldots, B$. The learner is trained on each training set, predictions are made on the corresponding test set (sometimes on the training set as well) and the performance measure $S(D^{*b}, D \setminus D^{*b})$ is calculated. Then the B individual performance values are aggregated, most often by calculating the mean. There exist various different resampling strategies, for example cross-validation and bootstrap, to mention just two popular approaches.



Resampling Figure

If you want to read up on further details, the paper Resampling Strategies for Model Assessment and Selection (http://link.springer.com/chapter/10.1007%2F978-0-387-47509-7_8) by Simon is probably not a bad choice. Bernd has also published a paper Resampling methods for meta-model validation with recommendations for evolutionary computation (http://www.mitpressjournals.org/doi/pdf/10.1162/EVCO_a_00069) which contains detailed descriptions and lots of statistical background information on resampling methods.

Defining the resampling strategy

In `mlr` the resampling strategy can be defined via function `makeResampleDesc()` (`../../reference/makeResampleDesc.html`). It requires a string that specifies the resampling method and, depending on the selected strategy, further information like the number of iterations. The supported resampling strategies are:

- Cross-validation (`"CV"`),
- Leave-one-out cross-validation (`"LOO"`),
- Repeated cross-validation (`"RepCV"`),
- Out-of-bag bootstrap and other variants like *b632* (`"Bootstrap"`),
- Subsampling, also called Monte-Carlo cross-validation (`"Subsample"`),
- Holdout (training/test) (`"Holdout"`).

For example if you want to use 3-fold cross-validation type:

```
# 3-fold cross-validation
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3)
rdesc
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE
```

For holdout estimation use:

```
# Holdout estimation
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("Holdout")
rdesc
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
```

In order to save you some typing `mlr` contains some pre-defined resample descriptions for very common strategies like holdout (`hout` (`makeResampleDesc()` (`../../reference/makeResampleDesc.html`))) as well as cross-validation with different numbers of folds (e.g., `cv5` (`makeResampleDesc()` (`../../reference/makeResampleDesc.html`)) or `cv10` (`makeResampleDesc()` (`../../reference/makeResampleDesc.html`))).

```
hout
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE

cv3
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE
```

Performing the resampling

Function `resample()` (../../reference/resample.html) evaluates a Learner (`makeLearner()` (../../reference/makeLearner.html)) on a given machine learning `Task()` (../../reference/Task.html) using the selected resampling strategy (`makeResampleDesc()` (../../reference/makeResampleDesc.html)).

As a first example, the performance of linear regression (`stats::lm()` (http://www.rdocumentation.org/packages/stats/topics/lm)) on the `BostonHousing` (`mlbench::BostonHousing()` (http://www.rdocumentation.org/packages/mlbench/topics/BostonHousing)) data set is calculated using *3-fold cross-validation*.

Generally, for *K-fold cross-validation* the data set D is partitioned into K subsets of (approximately) equal size. In the b-th of the K iterations, the b-th subset is used for testing, while the union of the remaining parts forms the training set.

As usual, you can either pass a Learner (`makeLearner()` (../../reference/makeLearner.html)) object to `resample()` (../../reference/resample.html) or, as done here, provide the class name `"regr.lm"` of the learner. Since no performance measure is specified the default for regression learners (mean squared error, mse (measures.html)) is calculated.

```
## Resampling: cross-validation
## Measures:             mse
## [Resample] iter 1:    25.1371739
## [Resample] iter 2:    23.1279497
## [Resample] iter 3:    21.9152672
##
## Aggregated Result: mse.test.mean=23.3934636
##
```

```
# Specify the resampling strategy (3-fold cross-validation)
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3)

# Calculate the performance
r = resample(../../reference/resample.html)("regr.lm", bh.task, rdesc)
## Resampling: cross-validation
## Measures:             mse
## [Resample] iter 1:    25.1371739
## [Resample] iter 2:    23.1279497
## [Resample] iter 3:    21.9152672
##
## Aggregated Result: mse.test.mean=23.3934636
##

r
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm
## Aggr perf: mse.test.mean=23.3934636
## Runtime: 0.0375051
```

The result `r` is an object of class `resample()` (../../reference/resample.html) result. It contains performance results for the learner and some additional information like the runtime, predicted values, and optionally the models fitted in single resampling iterations.

```
# Peak into r
names(r)
##  [1] "learner.id"     "task.id"        "task.desc"      "measures.train"
##  [5] "measures.test"  "aggr"           "pred"           "models"
##  [9] "err.msgs"       "err.dumps"      "extract"        "runtime"

r$aggr
## mse.test.mean
##      23.39346

r$measures.test
##   iter      mse
## 1    1 25.13717
## 2    2 23.12795
## 3    3 21.91527
```

`r$measures.test` gives the performance on each of the 3 test data sets. `r$aggr` shows the aggregated performance value. Its name `"mse.test.mean"` indicates the performance measure, mse (measures.html), and the method, `test.mean` (`aggregations()` (../../reference/aggregations.html)), used to aggregate the 3 individual performances. `test.mean` (`aggregations()` (../../reference/aggregations.html)) is the default aggregation scheme for most performance measures and, as the name implies, takes the mean over the performances on the test data sets.

Resampling in `mlr` works the same way for all types of learning problems and learners. Below is a classification example where a classification tree (rpart) (`rpart::rpart()` (http://www.rdocumentation.org/packages/rpart/topics/rpart)) is evaluated on the `Sonar` (`mlbench::sonar()`) data set by subsampling with 5 iterations.

In each subsampling iteration the data set D is randomly partitioned into a training and a test set according to a given percentage, e.g., 2/3 training and 1/3 test set. If there is just one iteration, the strategy is commonly called *holdout* or *test sample estimation*.

You can calculate several measures at once by passing a `list` of Measures (`makeMeasure()` (../../reference/makeMeasure.html))s to `resample()` (../../reference/resample.html). Below, the error rate (mmce (measures.html)), false positive and false negative rates (fpr (measures.html), fnr (measures.html)), and the time it takes to train the learner (timetrain (measures.html)) are estimated by *subsampling* with 5 iterations.

```
# Subsampling with 5 iterations and default split ratio 2/3
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("Subsample", ite

# Subsampling with 5 iterations and 4/5 training data
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("Subsample", ite

# Classification tree with information splitting criterion
lrn = makeLearner (../../reference/makeLearner.html)("classif.rpart", parms = lis

# Calculate the performance measures
r = resample (../../reference/resample.html)(lrn, sonar.task, rdesc, measures = l
## Resampling: subsampling
## Measures:             mmce        fpr         fnr         timetrain
## [Resample] iter 1:    0.4047619   0.5416667   0.2222222   0.0110000
## [Resample] iter 2:    0.1666667   0.1200000   0.2352941   0.0070000
## [Resample] iter 3:    0.3333333   0.1333333   0.4444444   0.0100000
## [Resample] iter 4:    0.2380952   0.3913043   0.0526316   0.0280000
## [Resample] iter 5:    0.3095238   0.2800000   0.3529412   0.0080000
##
## Aggregated Result: mmce.test.mean=0.2904762,fpr.test.mean=0.2932609,fnr.test.m
##

r
## Resample Result
## Task: Sonar-example
## Learner: classif.rpart
## Aggr perf: mmce.test.mean=0.2904762,fpr.test.mean=0.2932609,fnr.test.mean=0.26
## Runtime: 0.10692
```

If you want to add further measures afterwards, use `addRRMeasure()` (../../reference/addRRMeasure.html).

```
# Add balanced error rate (ber) and time used to predict
addRRMeasure (../../reference/addRRMeasure.html)(r, list(ber, timepredict))

## Resample Result
## Task: Sonar-example
## Learner: classif.rpart
## Aggr perf: mmce.test.mean=0.2904762,fpr.test.mean=0.2932609,fnr.test.mean=0.2
## Runtime: 0.10692
```

By default, `resample()` (../../reference/resample.html) prints progress messages and intermediate results. You can turn this off by setting `show.info = FALSE`, as done in the code chunk below. (If you are interested in suppressing these messages permanently have a look at the tutorial page about configuring mlr (configureMlr.html).)

In the above example, the Learner (`makeLearner()` (../../reference/makeLearner.html)) was explicitly constructed. For convenience you can also specify the learner as a string and pass any learner parameters via the `...` argument of `resample()` (../../reference/resample.html).

```
r = resample (../../reference/resample.html)("classif.rpart", parms = list(split
  measures = list(mmce, fpr, fnr, timetrain), show.info = FALSE)

r

## Resample Result
## Task: Sonar-example
## Learner: classif.rpart
## Aggr perf: mmce.test.mean=0.2428571,fpr.test.mean=0.2968173,fnr.test.mean=0.1
## Runtime: 0.0791025
```

Accessing resample results

Apart from the learner performance you can extract further information from the resample results, for example predicted values or the models fitted in individual resample iterations.

Predictions

Per default, the `resample()` (../../reference/resample.html) result contains the predictions made during the resampling. If you do not want to keep them, e.g., in order to conserve memory, set `keep.pred = FALSE` when calling `resample()` (../../reference/resample.html).

The predictions are stored in slot `$pred` of the resampling result, which can also be accessed by function `getRRPredictions()` (../../reference/getRRPredictions.html).

```
r$pred
## Resampled Prediction for:
## Resample description: subsampling with 5 iterations and 0.80 split rate.
## Predict: test
## Stratification: FALSE
## predict.type: response
## threshold:
## time (mean): 0.00
##    id truth response iter  set
## 1  18     R        M    1 test
## 2 189     M        M    1 test
## 3  88     R        R    1 test
## 4 121     M        R    1 test
## 5 165     M        R    1 test
## 6 111     M        M    1 test
##  ... (#rows: 210, #cols: 5)

pred = getRRPredictions (../../reference/getRRPredictions.html)(r)
pred
## Resampled Prediction for:
## Resample description: subsampling with 5 iterations and 0.80 split rate.
## Predict: test
## Stratification: FALSE
## predict.type: response
## threshold:
## time (mean): 0.00
##    id truth response iter  set
## 1  18     R        M    1 test
## 2 189     M        M    1 test
## 3  88     R        R    1 test
## 4 121     M        R    1 test
## 5 165     M        R    1 test
## 6 111     M        M    1 test
##  ... (#rows: 210, #cols: 5)
```

`pred` is an object of class `resample()` (../../reference/resample.html) Prediction. Just as a `Prediction()` (../../reference/Prediction.html) object (see the tutorial page on making predictions (predict.html) it has an element `$data` which is a `data.frame` that contains the predictions and in the case of a supervised learning problem the true values of the target variable(s). You can use `as.data.frame(Prediction()` (../../reference/Prediction.html) to directly access the `$data` slot. Moreover, all getter functions for `Prediction()` (../../reference/Prediction.html) objects like `getPredictionResponse()` (../../reference/getPredictionResponse.html) or `getPredictionProbabilities()` (../../reference/getPredictionProbabilities.html) are applicable.

```
head(as.data.frame(pred))
##    id truth response iter  set
## 1 207     M        M    1 test
## 2 127     M        M    1 test
## 3 182     M        M    1 test
## 4 166     M        M    1 test
## 5 119     M        M    1 test
## 6 114     M        R    1 test

head(getPredictionTruth (../../reference/getPredictionResponse.html)(pred))
## [1] M M M M M M
## Levels: M R

head(getPredictionResponse (../../reference/getPredictionResponse.html)(pred))
## [1] M M M M M R
## Levels: M R
```

The columns `iter` and `set` in the `data.frame` indicate the resampling iteration and the data set (`train` or `test`) for which the prediction was made.

By default, predictions are made for the test sets only. If predictions for the training set are required, set `predict = "train"` (for predictions on the train set only) or `predict = "both"` (for predictions on both train and test sets) in `makeResampleDesc()` (../../reference/makeResampleDesc.html). In any case, this is necessary for some bootstrap methods (*b632* and *b632+*) and some examples are shown later on.

Below, we use simple Holdout, i.e., split the data once into a training and test set, as resampling strategy and make predictions on both sets.

```
# Make predictions on both training and test sets
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("Holdout", predi

r = resample (../../reference/resample.html)("classif.lda", iris.task, rdesc, sh
r
## Resample Result
## Task: iris-example
## Learner: classif.lda
## Aggr perf: mmce.test.mean=0.0200000
## Runtime: 0.00993848

r$measures.train
##   iter mmce
## 1    1 0.02
```

(Please note that nonetheless the misclassification rate `r$aggr` is estimated on the test data only. How to calculate performance measures on the training sets is shown below.)

A second function to extract predictions from resample results is `getRRPredictionList()` (../../reference/getRRPredictionList.html) which returns a `list` of predictions split by data set (train/test) and resampling iteration.

```
predList = getRRPredictionList (../../reference/getRRPredictionList.html)(r)
predList
## $train
## $train$`1`
## Prediction: 100 observations
## predict.type: response
## threshold:
## time: 0.00
##      id      truth   response
## 96   96 versicolor versicolor
## 130 130  virginica  virginica
## 120 120  virginica  virginica
## 77   77 versicolor versicolor
## 23   23     setosa     setosa
## 59   59 versicolor versicolor
##  ... (#rows: 100, #cols: 3)
##
##
## $test
## $test$`1`
## Prediction: 50 observations
## predict.type: response
## threshold:
## time: 0.00
##      id      truth   response
## 92   92 versicolor versicolor
## 58   58 versicolor versicolor
## 48   48     setosa     setosa
## 103 103  virginica  virginica
## 70   70 versicolor versicolor
## 82   82 versicolor versicolor
##  ... (#rows: 50, #cols: 3)
```

Learner models

In each resampling iteration a Learner (`makeLearner()` (../../reference/makeLearner.html)) is fitted on the respective training set. By default, the resulting `WrappedModel` (`makeWrappedModel()` (../../reference/makeWrappedModel.html))s are not included in the `resample()` (../../reference/resample.html) result and slot `$models` is empty. In order to keep them, set `models = TRUE` when calling `resample()` (../../reference/resample.html), as in the following survival analysis example.

```
# 3-fold cross-validation
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 3)

r = resample (../../reference/resample.html)("surv.coxph", lung.task, rdesc, show
r$models
## [[1]]
## Model for learner.id=surv.coxph; learner.class=surv.coxph
## Trained on: task.id = lung-example; obs = 111; features = 8
## Hyperparameters:
##
## [[2]]
## Model for learner.id=surv.coxph; learner.class=surv.coxph
## Trained on: task.id = lung-example; obs = 111; features = 8
## Hyperparameters:
##
## [[3]]
## Model for learner.id=surv.coxph; learner.class=surv.coxph
## Trained on: task.id = lung-example; obs = 112; features = 8
## Hyperparameters:
```

The extract option

Keeping complete fitted models can be memory-intensive if these objects are large or the number of resampling iterations is high. Alternatively, you can use the `extract` argument of `resample()` (../../reference/resample.html) to retain only the information you need. To this end you need to pass a function to `extract` which is applied to each `WrappedModel` (`makeWrappedModel()` (../../reference/makeWrappedModel.html)) object fitted in each resampling iteration.

Below, we cluster the `datasets::mtcars()` (http://www.rdocumentation.org/packages/datasets/topics/mtcars) data using the k-means algorithm with $k = 3$ and keep only the cluster centers.

```
# 3-fold cross-validation
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 3)

# Extract the compute cluster centers
r = resample (../../reference/resample.html)("cluster.kmeans", mtcars.task, rdesc
  centers = 3, extract = function(x) getLearnerModel (../../reference/getLearnerM
## Warning in rgl.init(initValue, onlyNULL): RGL: unable to open X11 display
## Warning: 'rgl_init' failed, running with rgl.useNULL = TRUE
r$extract
## [[1]]
##        mpg      cyl      disp    hp     drat       wt     qsec        vs
## 1 26.96667 4.000000  99.08333  89.5 4.076667 2.087167 18.26833 0.8333333
## 2 20.61429 5.428571 166.81429 104.0 3.715714 3.167857 19.11429 0.7142857
## 3 15.26667 8.000000 356.64444 216.0 3.251111 3.956556 16.55556 0.0000000
##          am     gear     carb
## 1 1.0000000 4.333333 1.500000
## 2 0.2857143 3.857143 3.000000
## 3 0.2222222 3.444444 3.666667
##
## [[2]]
##        mpg      cyl     disp       hp   drat       wt     qsec        vs
## 1 15.03750 8.000000 351.1750 205.0000 3.1825 4.128125 17.08375 0.0000000
## 2 21.22222 5.111111 157.6889 110.1111 3.7600 2.945000 18.75889 0.6666667
## 3 31.00000 4.000000  76.1250  62.2500 4.3275 1.896250 19.19750 1.0000000
##          am     gear     carb
## 1 0.0000000 3.000000 3.375000
## 2 0.4444444 3.888889 2.888889
## 3 1.0000000 4.000000 1.250000
##
## [[3]]
##        mpg      cyl     disp       hp     drat       wt     qsec        vs
## 1 14.68571 8.000000 384.8571 230.5714 3.378571 4.077000 16.34000 0.0000000
## 2 25.30000 4.500000 113.8125 101.2500 4.037500 2.307875 18.00125 0.7500000
## 3 16.96667 7.333333 276.1000 145.8333 2.981667 3.580000 18.20500 0.3333333
##          am     gear  carb
## 1 0.2857143 3.571429 4.000
## 2 0.7500000 4.250000 2.375
## 3 0.0000000 3.000000 2.000
```

As a second example, we extract the variable importances from fitted regression trees using function `getFeatureImportance()` (../../reference/getFeatureImportance.html). (For more detailed information on this topic see the feature selection (feature_selection.html) page.)

```
# Extract the variable importance in a regression tree
r = resample (../../reference/resample.html)("regr.rpart", bh.task, rdesc, show.
r$extract
## [[1]]
## FeatureImportance:
## Task: BostonHousing-example
##
## Learner: regr.rpart
## Measure: NA
## Contrast: NA
## Aggregation: function (x)  x
## Replace: NA
## Number of Monte-Carlo iterations: NA
## Local: FALSE
##      crim       zn    indus chas      nox       rm      age      dis rad
## 1 2689.63 867.4877 3719.711    0 2103.622 16096.32 2574.183 3647.211   0
##        tax  ptratio       b    lstat
## 1 1972.207 3712.621 395.486 8608.757
##
## [[2]]
## FeatureImportance:
## Task: BostonHousing-example
##
## Learner: regr.rpart
## Measure: NA
## Contrast: NA
## Aggregation: function (x)  x
## Replace: NA
## Number of Monte-Carlo iterations: NA
## Local: FALSE
##       crim       zn  indus chas      nox       rm      age     dis
## 1 7491.707 5423.593 7295.2    0 7348.742 14014.78 1391.373 2309.92
##        rad      tax  ptratio b    lstat
## 1 340.3975 1871.451 938.0743 0 17618.49
##
## [[3]]
## FeatureImportance:
## Task: BostonHousing-example
##
## Learner: regr.rpart
## Measure: NA
## Contrast: NA
## Aggregation: function (x)  x
## Replace: NA
## Number of Monte-Carlo iterations: NA
## Local: FALSE
##       crim       zn    indus chas     nox       rm     age      dis
## 1 2532.084 4637.312 6150.854    0 6015.01 11330.75 6843.29 2049.772
##        rad      tax  ptratio        b    lstat
## 1 525.3815 747.8954 1925.899 62.58285 17336.77
```

There is also an convenience function `getResamplingIndices()` (`../../reference/getResamplingIndices.html`) to extract the resampling indices from the `ResampleResult` object:

```
getResamplingIndices (../../reference/getResamplingIndices.html)(r)
## $train.inds
## $train.inds[[1]]
##   [1] 476 219 259  73  29 324  13 411 363 357 299 303 275 247 125 139  92
##  [18] 487   6  70 243 328 473 448 449 322 346 472  97 315  46 232 351 479
##  [35]  74  51 369 289 114 320 258  65 176 262 138 160 412 501 260 111 283
##  [52] 444 209  12 166 108 475 492 124 132 187 185 257 424 361 141 280 374
##  [69] 350 495 425 504 352 427  19 106  80 474 338 333   8 250 296 144 122
##  [86] 142 210 226 268 364 477  31 214 309 286 215 370 372 120  64 445 199
## [103] 503 330 113 271  75 103 323 193 249 230 385 344 336 127 392 441  54
## [120]  39 242 151 310 263 238 137 112  62 183 191 368 386 229 130 397  49
## [137] 184  69  14 269 255 505 365 213 102 150 126 483 399  82  60  66 152
## [154]  76 162 465 192 462 307 329 288  45 197 439  17 327 499  22 377 434
## [171]  15 360  48 451 332 221  55 356  36 494 165 486 458 387 409 101 295
## [188] 456 196  43 484 198 491  77 235 316 395 135  91 353   2 371 394 140
## [205] 452 282 153  56 415 313 147 146 500 467 182 181 349 294 100 159 342
## [222]  18 317  71 291  94 239 410 461 432 123  37 240 256 149 373 225  59
## [239] 231 482 388 382 402 442  20 121 164 105 455 281 211 318   3  86 480
## [256] 218 265 115 228 206 212 208 345 116 180 190 202 155 148 297 339 252
## [273] 416 293 391  26 274  28 279 378 433 403  33 207 319 117 419 145 358
## [290] 335 340 331  47 366 334 220  25 128 408  95 178 285 261  79 251 237
## [307] 362 167  84 308 478 421  27 367 236  50 201 406  16 471  72 312 389
## [324]  57 179 203 305 423 341 267 246 453 354 485 438 321 173
##
## $train.inds[[2]]
##   [1]   4 219 420 241  29 324 186  13  38 363  24 357   7 109 303 247 125
##  [18] 174 139 222 248 328 346 472  97 315  46 232 351 369 172 289 348 114
##  [35] 170 154 258  65 134 234 160 171 204  78 209 404  12 108 475  30 278
##  [52] 492 187 418 257 424 361 141 280 374 504 156 469 436 325  19 106 474
##  [69] 338 333 430 296 200 417 254 157 343 364 161 477  68 214 309 286 370
##  [86]  64 489 503 264  93 113  75 103  61  52 446 193 355 249 230  99 344
## [103] 336  10 502 127 490 441  54  39 242 151   9 310 263 390  44 272 238
## [120]  58 506 112 481 368 386  89 229 194  53 397  49 184  14 284 188 505
## [137] 244 365 213 102 150 233  90 399  82  66 152 463 169 398 465 401 307
## [154] 327 110 443 454  22 359 384 377 434 407 216  40 277 360  34 437 129
## [171] 221 314 498  55  36 494 175 387 253 409 104 337 413 431 298 493 196
## [188] 301  43 484 198 276 383 143 135  32 304  83 496   2 422 394 468 452
## [205] 153 415 133  23 466 467 181 349 294 100 470 159 292  35 376 302 410
## [222]  41  88 440 123  37 375 393 118 373 225  81 306 388 266 382 402 442
## [239] 131 287 164 396 119 273 281 195  98 211   3 426 223 447 400 405 265
## [256] 115 459 228 163 206 290 345 116 180 202 155 297 379  67 460  63  11
## [273] 252 416 227  26 274   1 450 177  87 245 435 428 347  33 207 429 117
## [290]  42 335 340 217 326 488  21 408 300 224 285 251 362 457  84 478 421
## [307] 136 311  27 236 406  96 270 471  72 389 414  57 179 205 168 305 341
## [324] 464 267 246 381 453 380 189 485  85 107 497 438 173   5 158
##
## $train.inds[[3]]
##   [1]   4 476 420 259 241  73 186  38 411  24   7 299 109 275 174  92 487
##  [18]   6  70 243 222 248 473 448 449 322 479  74  51 172 348 170 154 320
##  [35] 134 176 234 262 138 412 501 260 111 171 283 444 204  78 404 166  30
##  [52] 278 124 132 185 418 350 495 425 156 469 352 427 436 325  80 430   8
```

```
##  [69] 250 200 144 122 417 142 210 254 157 226 268 343 161  68  31 215 372
##  [86] 120 445 199 489 330 264  93 271 323  61  52 446 355 385  99  10 502
## [103] 392 490   9 390  44 272 137  58 506  62 183 191 481  89 194 130  53
## [120]  69 284 269 255 188 244 233 126 483  90  60 463 169  76 398 162 192
## [137] 462 401 329 288  45 197 439  17 499 110 443 454 359 384 407 216  15
## [154]  40 277  48  34 451 437 129 332 314 498 356 165 486 175 458 253 104
## [171] 337 413 431 298 101 295 493 456 301 491  77 235 316 276 383 395 143
## [188]  91  32 353 304  83 496 422 371 140 468 282  56 313 147 133 146  23
## [205] 466 500 182 470 342  18 317 292  71  35 376 291  94 239 302 461  41
## [222]  88 432 440 375 393 240 256 118 149  59 231 482  81 306 266 131  20
## [239] 121 287 105 396 455 119 273 195  98 318  86 480 426 223 447 218 400
## [256] 405 459 163 290 212 208 190 148 379  67 460 339  63  11 227 293 391
## [273]   1  28 450 177  87 245 435 279 428 378 347 433 403 319 429  42 419
## [290] 145 358 217 331  47 366 326 334 220 488  21  25 128 300  95 178 224
## [307] 261  79 237 457 167 308 136 311 367  50 201  16  96 270 312 414 205
## [324] 168 203 423 464 381 354 380 189  85 107 497 321   5 158
##
##
## $test.inds
## $test.inds[[1]]
##   [1]   1   4   5   7   9  10  11  21  23  24  30  32  34  35  38  40  41
##  [18]  42  44  52  53  58  61  63  67  68  78  81  83  85  87  88  89  90
##  [35]  93  96  98  99 104 107 109 110 118 119 129 131 133 134 136 143 154
##  [52] 156 157 158 161 163 168 169 170 171 172 174 175 177 186 188 189 194
##  [69] 195 200 204 205 216 217 222 223 224 227 233 234 241 244 245 248 253
##  [86] 254 264 266 270 272 273 276 277 278 284 287 290 292 298 300 301 302
## [103] 304 306 311 314 325 326 337 343 347 348 355 359 375 376 379 380 381
## [120] 383 384 390 393 396 398 400 401 404 405 407 413 414 417 418 420 422
## [137] 426 428 429 430 431 435 436 437 440 443 446 447 450 454 457 459 460
## [154] 463 464 466 468 469 470 481 488 489 490 493 496 497 498 502 506
##
## $test.inds[[2]]
##   [1]   6   8  15  16  17  18  20  25  28  31  45  47  48  50  51  56  59
##  [18]  60  62  69  70  71  73  74  76  77  79  80  86  91  92  94  95 101
##  [35] 105 111 120 121 122 124 126 128 130 132 137 138 140 142 144 145 146
##  [52] 147 148 149 162 165 166 167 176 178 182 183 185 190 191 192 197 199
##  [69] 201 203 208 210 212 215 218 220 226 231 235 237 239 240 243 250 255
##  [86] 256 259 260 261 262 268 269 271 275 279 282 283 288 291 293 295 299
## [103] 308 312 313 316 317 318 319 320 321 322 323 329 330 331 332 334 339
## [120] 342 350 352 353 354 356 358 366 367 371 372 378 385 391 392 395 403
## [137] 411 412 419 423 425 427 432 433 439 444 445 448 449 451 455 456 458
## [154] 461 462 473 476 479 480 482 483 486 487 491 495 499 500 501
##
## $test.inds[[3]]
##   [1]   2   3  12  13  14  19  22  26  27  29  33  36  37  39  43  46  49
##  [18]  54  55  57  64  65  66  72  75  82  84  97 100 102 103 106 108 112
##  [35] 113 114 115 116 117 123 125 127 135 139 141 150 151 152 153 155 159
##  [52] 160 164 173 179 180 181 184 187 193 196 198 202 206 207 209 211 213
##  [69] 214 219 221 225 228 229 230 232 236 238 242 246 247 249 251 252 257
##  [86] 258 263 265 267 274 280 281 285 286 289 294 296 297 303 305 307 309
## [103] 310 315 324 327 328 333 335 336 338 340 341 344 345 346 349 351 357
## [120] 360 361 362 363 364 365 368 369 370 373 374 377 382 386 387 388 389
```

```
## [137] 394 397 399 402 406 408 409 410 415 416 421 424 434 438 441 442 452
## [154] 453 465 467 471 472 474 475 477 478 484 485 492 494 503 504 505
```

Stratification, Blocking and Grouping

- *Stratification* with respect to a categorical variable makes sure that all its values are present in each training and test set in approximately the same proportion as in the original data set. Stratification is possible with regard to categorical target variables (and thus for supervised classification and survival analysis) or categorical explanatory variables.
- *Blocking* refers to the situation that subsets of observations belong together and must not be separated during resampling. Hence, for one train/test set pair the entire block is either in the training set or in the test set.
- *Grouping* means that the folds are composed out of a factor vector given by the user. In this setting no repetitions are possible as all folds are predefined. The approach can also be used in a nested resampling setting. Note the subtle but important difference to "Blocking": In "Blocking" factor levels are respected when splitting into train and test (e.g. the test set could be composed out of two given factor levels) whereas in "Grouping" the folds will strictly follow the factor level grouping (meaning that the test set will always only consist of one factor level).

Stratification with respect to the target variable(s)

For classification, it is usually desirable to have the same proportion of the classes in all of the partitions of the original data set. This is particularly useful in the case of imbalanced classes and small data sets. Otherwise, it may happen that observations of less frequent classes are missing in some of the training sets which can decrease the performance of the learner, or lead to model crashes. In order to conduct stratified resampling, set `stratify = TRUE` in `makeResampleDesc()` (../../reference/makeResampleDesc.html).

```
# 3-fold cross-validation
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3,

r = resample(../../reference/resample.html)("classif.lda", iris.task, rdesc, sho
r
## Resample Result
## Task: iris-example
## Learner: classif.lda
## Aggr perf: mmce.test.mean=0.0200000
## Runtime: 0.0176296
```

Stratification is also available for survival tasks. Here the stratification balances the censoring rate.

Stratification with respect to explanatory variables

Sometimes it is required to also stratify on the input data, e.g., to ensure that all subgroups are represented in all training and test sets. To stratify on the input columns, specify `factor` columns of your task data via `stratify.cols`.

```
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3,

r = resample(../../reference/resample.html)("regr.rpart", bh.task, rdesc, show.i
r
## Resample Result
## Task: BostonHousing-example
## Learner: regr.rpart
## Aggr perf: mse.test.mean=23.8843587
## Runtime: 0.0268815
```

Blocking: CV with flexible predefined indices

If some observations "belong together" and must not be separated when splitting the data into training and test sets for resampling, you can supply this information via a `blocking` factor when creating the task (task.html).

```
# 5 blocks containing 30 observations each
task = makeClassifTask(../../reference/Task.html)(data = iris, target = "Species
task
## Supervised task: iris
## Type: classif
## Target: Species
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: TRUE
## Has coordinates: FALSE
## Classes: 3
##     setosa versicolor  virginica
##         50         50         50
## Positive class: NA
```

When performing a simple "CV" resampling and inspecting the result, we see that the training indices in fold 1 correspond to the specified grouping set in `blocking` in the task. To initiate this method, we need to set `blocking.cv = TRUE` when creating the resample description

object.

```
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", iters = 3
p = resample (../../reference/resample.html)("classif.lda", task, rdesc)
## Resampling: cross-validation
## Measures:             mmce
## [Resample] iter 1:    0.0500000
## [Resample] iter 2:    0.0500000
## [Resample] iter 3:    0.0000000
##
## Aggregated Result: mmce.test.mean=0.0333333
##

sort(p$pred$instance$train.inds[[1]])
##  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
## [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
## [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
## [52]  52  53  54  55  56  57  58  59  60 121 122 123 124 125 126 127 128
## [69] 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145
## [86] 146 147 148 149 150
```

However, please note the effects of this method: The created folds will not have the same size! Here, Fold 1 has a 120/30 split while the other two folds have a 90/60 split.

```
lapply(p$pred$instance$train.inds, function(x) length(x))
## [[1]]
## [1] 90
##
## [[2]]
## [1] 90
##
## [[3]]
## [1] 120
```

This is caused by the fact that we supplied five groups that must belong together but only used a three fold resampling strategy here.

Grouping: CV with fixed predefined indices

There is a second way of using predefined indices in resampling in `mlr`: Constructing the folds based on the supplied indices in `blocking`. We refer to this method here as “grouping” to distinguish it from “blocking”. This method is more restrictive in the way that it will always use the number of levels supplied via `blocking` as the number of folds. To use this method, we need to set `fixed = TRUE` instead of `blocking.cv` when creating the resampling description object.

We can leave out the `iters` argument, as it will be set internally to the number of supplied factor levels.

```
rdesc = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", fixed = TR
p = resample (../../reference/resample.html)("classif.lda", task, rdesc)
## Warning in instantiateResampleInstance.CVDesc(desc, size, task): Adjusting
## levels to match number of blocking levels.
## Resampling: cross-validation
## Measures:             mmce
## [Resample] iter 1:    0.1000000
## [Resample] iter 2:    0.0000000
## [Resample] iter 3:    0.0000000
## [Resample] iter 4:    0.0000000
## [Resample] iter 5:    0.1000000
##
## Aggregated Result: mmce.test.mean=0.0400000
##
sort(p$pred$instance$train.inds[[1]])
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
## [103] 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
## [120] 120
```

You can see that we automatically created five folds in which the test set always corresponds to one factor level.

Doing it this way also means that we cannot do repeated CV because there is no way to create multiple shuffled folds of this fixed arrangement.

```
lapply(p$pred$instance$train.inds, function(x) length(x))
## [[1]]
## [1] 120
##
## [[2]]
## [1] 120
##
## [[3]]
## [1] 120
##
## [[4]]
## [1] 120
##
## [[5]]
## [1] 120
```

However, this method can also be used in nested resampling settings (e.g. in hyperparameter tuning). In the inner level, the factor levels are honored and the function simply creates one fold less than in the outer level.

Please note that the `iters` argument has no effect in `makeResampleDesc()` (../../reference/makeResampleDesc.html) if `fixed = TRUE`. The number of folds will be automatically set based on the supplied number of factor levels via `blocking`. In the inner level, the number of folds will simply be one less than in the outer level.

```
# test fixed in nested resampling
lrn = makeLearner (../../reference/makeLearner.html)("classif.lda")
ctrl ← makeTuneControlRandom (../../reference/makeTuneControlRandom.html)(maxit
ps ← makeParamSet(makeNumericParam("nu", lower = 2, upper = 20))
inner = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", fixed = TR
outer = makeResampleDesc (../../reference/makeResampleDesc.html)("CV", fixed = TR
tune_wrapper = makeTuneWrapper (../../reference/makeTuneWrapper.html)(lrn, resamp
  control = ctrl, show.info = FALSE)

p = resample (../../reference/resample.html)(tune_wrapper, task, outer, show.info
  extract = getTuneResult)
```

To check on the inner resampling indices, you can call `getResamplingIndices(inner = TRUE)` (../../reference/getResamplingIndices.html). You can see that for every outer fold (List of 5), four inner folds were created that respect the grouping supplied via the `blocking` argument.

Of course you can also use a normal random sampling “CV” description in the inner level by just setting `fixed = FALSE`.

```
str(getResamplingIndices (../../reference/getResamplingIndices.html)(p, inner = 
## List of 5
##  $ :List of 2
##   ..$ train.inds:List of 4
##   .. ..$ : int [1:90] 3 150 120 97 133 124 116 29 93 22 ...
##   .. ..$ : int [1:90] 3 150 67 68 65 133 124 29 22 84 ...
##   .. ..$ : int [1:90] 150 120 97 67 68 65 133 124 116 93 ...
##   .. ..$ : int [1:90] 3 120 97 67 68 65 116 29 93 22 ...
##   ..$ test.inds :List of 4
##   .. ..$ : int [1:30] 70 79 61 63 64 80 68 62 69 81 ...
##   .. ..$ : int [1:30] 91 115 105 116 110 102 94 101 98 99 ...
##   .. ..$ : int [1:30] 10 29 21 9 1 12 30 3 22 2 ...
##   .. ..$ : int [1:30] 138 144 134 129 136 121 131 146 128 139 ...
##  $ :List of 2
##   ..$ train.inds:List of 4
##   .. ..$ : int [1:90] 130 104 144 88 100 125 92 84 111 135 ...
##   .. ..$ : int [1:90] 130 52 55 144 39 88 42 32 125 84 ...
##   .. ..$ : int [1:90] 130 104 52 55 144 39 42 100 32 125 ...
##   .. ..$ : int [1:90] 104 52 55 39 88 42 100 32 92 84 ...
##   ..$ test.inds :List of 4
##   .. ..$ : int [1:30] 55 50 60 54 39 45 38 35 32 47 ...
##   .. ..$ : int [1:30] 91 115 105 116 110 102 94 101 98 99 ...
##   .. ..$ : int [1:30] 70 79 61 63 64 80 68 62 69 81 ...
##   .. ..$ : int [1:30] 138 144 134 129 136 121 131 146 128 139 ...
##  $ :List of 2
##   ..$ train.inds:List of 4
##   .. ..$ : int [1:90] 75 49 148 70 50 124 31 76 39 130 ...
##   .. ..$ : int [1:90] 75 49 15 8 22 28 70 50 14 31 ...
##   .. ..$ : int [1:90] 75 15 8 22 28 148 70 124 14 76 ...
##   .. ..$ : int [1:90] 49 15 8 22 28 148 50 124 14 31 ...
##   ..$ test.inds :List of 4
##   .. ..$ : int [1:30] 10 29 21 9 1 12 30 3 22 2 ...
##   .. ..$ : int [1:30] 138 144 134 129 136 121 131 146 128 139 ...
##   .. ..$ : int [1:30] 55 50 60 54 39 45 38 35 32 47 ...
##   .. ..$ : int [1:30] 70 79 61 63 64 80 68 62 69 81 ...
##  $ :List of 2
##   ..$ train.inds:List of 4
##   .. ..$ : int [1:90] 123 107 34 33 103 124 120 112 101 109 ...
##   .. ..$ : int [1:90] 123 107 103 124 120 112 18 101 109 19 ...
##   .. ..$ : int [1:90] 123 34 33 124 18 19 8 137 51 141 ...
##   .. ..$ : int [1:90] 107 34 33 103 120 112 18 101 109 19 ...
##   ..$ test.inds :List of 4
##   .. ..$ : int [1:30] 10 29 21 9 1 12 30 3 22 2 ...
##   .. ..$ : int [1:30] 55 50 60 54 39 45 38 35 32 47 ...
##   .. ..$ : int [1:30] 91 115 105 116 110 102 94 101 98 99 ...
##   .. ..$ : int [1:30] 138 144 134 129 136 121 131 146 128 139 ...
##  $ :List of 2
##   ..$ train.inds:List of 4
##   .. ..$ : int [1:90] 17 19 35 27 18 29 85 5 2 10 ...
##   .. ..$ : int [1:90] 116 35 109 111 120 85 49 105 61 106 ...
##   .. ..$ : int [1:90] 17 116 19 27 109 111 120 18 29 85 ...
```

```
##   ..  ..$ : int [1:90] 17 116 19 35 27 109 111 120 18 29  ...
##   ..$ test.inds :List of 4
##   ..  ..$ : int [1:30] 91 115 105 116 110 102 94 101 98 99  ...
##   ..  ..$ : int [1:30] 10 29 21 9 1 12 30 3 22 2  ...
##   ..  ..$ : int [1:30] 55 50 60 54 39 45 38 35 32 47  ...
##   ..  ..$ : int [1:30] 70 79 61 63 64 80 68 62 69 81  ...
```

Resample descriptions and resample instances

As already mentioned, you can specify a resampling strategy using function `makeResampleDesc()` (../../reference/makeResampleDesc.html).

```
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3)
rdesc
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE

str(rdesc)
## List of 6
##  $ fixed      : logi FALSE
##  $ blocking.cv: logi FALSE
##  $ id         : chr "cross-validation"
##  $ iters      : int 3
##  $ predict    : chr "test"
##  $ stratify   : logi FALSE
##  - attr(*, "class")= chr [1:2] "CVDesc" "ResampleDesc"

str(makeResampleDesc(../../reference/makeResampleDesc.html)("Subsample", strati
## List of 8
##  $ split        : num 0.667
##  $ id           : chr "subsampling"
##  $ iters        : int 30
##  $ predict      : chr "test"
##  $ stratify     : logi FALSE
##  $ stratify.cols: chr "chas"
##  $ fixed        : logi FALSE
##  $ blocking.cv  : logi FALSE
##  - attr(*, "class")= chr [1:2] "SubsampleDesc" "ResampleDesc"
```

The result `rdesc` inherits from class `ResampleDesc` (`makeResampleDesc()` (../../reference/makeResampleDesc.html)) (short for resample description) and, in principle, contains all necessary information about the resampling strategy including the number of iterations, the proportion of training and test sets, stratification variables, etc.

Given either the size of the data set at hand or the `Task()` (../../reference/Task.html), function `makeResampleInstance()` (../../reference/makeResampleInstance.html) draws the training and test sets according to the `ResampleDesc` (`makeResampleDesc()`

(../../reference/makeResampleDesc.html)).

```
# Create a resample instance based an a task
rin = makeResampleInstance (../../reference/makeResampleInstance.html)(rdesc, ir
rin
## Resample instance for 150 cases.
## Resample description: cross-validation with 3 iterations.
## Predict: test
## Stratification: FALSE

str(rin)
## List of 5
##  $ desc      :List of 6
##   ..$ fixed      : logi FALSE
##   ..$ blocking.cv: logi FALSE
##   ..$ id         : chr "cross-validation"
##   ..$ iters      : int 3
##   ..$ predict    : chr "test"
##   ..$ stratify   : logi FALSE
##   ..- attr(*, "class")= chr [1:2] "CVDesc" "ResampleDesc"
##  $ size      : int 150
##  $ train.inds:List of 3
##   ..$ : int [1:100] 16 69 106 67 83 104 129 36 76 105 ...
##   ..$ : int [1:100] 3 62 69 106 27 42 104 129 76 105 ...
##   ..$ : int [1:100] 3 62 16 67 27 83 42 36 71 30 ...
##  $ test.inds :List of 3
##   ..$ : int [1:50] 3 7 8 14 17 18 22 27 29 30 ...
##   ..$ : int [1:50] 9 10 12 16 19 20 25 28 33 36 ...
##   ..$ : int [1:50] 1 2 4 5 6 11 13 15 21 23 ...
##  $ group     : Factor w/ 0 levels:
##  - attr(*, "class")= chr "ResampleInstance"

# Create a resample instance given the size of the data set
rin = makeResampleInstance (../../reference/makeResampleInstance.html)(rdesc, si
str(rin)
## List of 5
##  $ desc      :List of 6
##   ..$ fixed      : logi FALSE
##   ..$ blocking.cv: logi FALSE
##   ..$ id         : chr "cross-validation"
##   ..$ iters      : int 3
##   ..$ predict    : chr "test"
##   ..$ stratify   : logi FALSE
##   ..- attr(*, "class")= chr [1:2] "CVDesc" "ResampleDesc"
##  $ size      : int 150
##  $ train.inds:List of 3
##   ..$ : int [1:100] 79 63 132 42 62 72 134 82 1 81 ...
##   ..$ : int [1:100] 112 26 121 63 132 131 24 141 93 86 ...
##   ..$ : int [1:100] 112 79 26 121 42 131 24 141 93 86 ...
##  $ test.inds :List of 3
##   ..$ : int [1:50] 14 17 18 19 24 25 26 27 31 35 ...
##   ..$ : int [1:50] 1 4 6 10 12 23 37 40 42 44 ...
##   ..$ : int [1:50] 2 3 5 7 8 9 11 13 15 16 ...
```

```
##  $ group     : Factor w/ 0 levels:
##  - attr(*, "class")= chr "ResampleInstance"

# Access the indices of the training observations in iteration 3
rin$train.inds[[3]]
##   [1] 112  79  26 121  42 131  24 141  93  86  72  82   1 115  98 110  27
##  [18]  81  50  18 105  90 150 119  69 147 111 128  67  64  38  45  52  17
##  [35]  51 124  48  41  80 108  96 145 148  89 133 127  19 142 104 101  92
##  [52]  99 117  36  12 125  66 126  91  14 123   4 149  40  31  71  85 129
##  [69]  56  95 107  49  44 114  35  59  77  46  94 102 130  23 139  74   6
##  [86] 120  70  10 140 106  76  60 103  88 113  37  25  65  78  58
```

The result `rin` inherits from class `ResampleInstance` (`makeResampleInstance()` (../../reference/makeResampleInstance.html)) and contains `list`s of index vectors for the train and test sets.

If a `ResampleDesc` (`makeResampleDesc()` (../../reference/makeResampleDesc.html)) is passed to `resample()` (../../reference/resample.html), it is instantiated internally. Naturally, it is also possible to pass a `ResampleInstance` (`makeResampleInstance()` (../../reference/makeResampleInstance.html)) directly.

While the separation between resample descriptions, resample instances, and the `resample()` (../../reference/resample.html) function itself seems overly complicated, it has several advantages:

- Resample instances readily allow for paired experiments, that is comparing the performance of several learners on exactly the same training and test sets. This is particularly useful if you want to add another method to a comparison experiment you already did. Moreover, you can store the resample instance along with your data in order to be able to reproduce your results later on.

```
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3)
rin = makeResampleInstance(../../reference/makeResampleInstance.html)(rdesc, tas

# Calculate the performance of two learners based on the same resample instance
r.lda = resample(../../reference/resample.html)("classif.lda", iris.task, rin, s
r.rpart = resample(../../reference/resample.html)("classif.rpart", iris.task, ri
r.lda$aggr
## mmce.test.mean
##           0.02

r.rpart$aggr
## mmce.test.mean
##     0.06666667
```

- In order to add further resampling methods you can simply derive from the `ResampleDesc` (`makeResampleDesc()` (../../reference/makeResampleDesc.html)) and `ResampleInstance` (`makeResampleInstance()` (../../reference/makeResampleInstance.html)) classes, but you do neither have to

touch `resample()` (../../reference/resample.html) nor any further methods that use the resampling strategy.

Usually, when calling `makeResampleInstance()` (../../reference/makeResampleInstance.html) the train and test index sets are drawn randomly. Mainly for *holdout* (*test sample*) *estimation* you might want full control about the training and tests set and specify them manually. This can be done using function `makeFixedHoldoutInstance()` (../../reference/makeFixedHoldoutInstance.html).

```
rin = makeFixedHoldoutInstance (../../reference/makeFixedHoldoutInstance.html)(t
rin
## Resample instance for 150 cases.
## Resample description: holdout with 0.67 split rate.
## Predict: test
## Stratification: FALSE
```

Aggregating performance values

In each resampling iteration $b = 1, \ldots, B$ we get performance values $S(D^{*b}, D \setminus D^{*b})$ (for each measure we wish to calculate), which are then aggregated to an overall performance.

For the great majority of common resampling strategies (like holdout, cross-validation, subsampling) performance values are calculated on the test data sets only and for most measures aggregated by taking the mean (`test.mean` (`aggregations()` (../../reference/aggregations.html))).

Each performance `Measure` (`makeMeasure()` (../../reference/makeMeasure.html)) in `mlr` has a corresponding default aggregation method which is stored in slot `$aggr`. The default aggregation for most measures is `test.mean` (`aggregations()` (../../reference/aggregations.html)). One exception is the root mean square error (rmse (measures.html)).

```
# Mean misclassification error
mmce$aggr
## Aggregation function: test.mean

mmce$aggr$fun
## function (task, perf.test, perf.train, measure, group, pred)
## mean(perf.test)
## <bytecode: 0×c8eade8>
## <environment: namespace:mlr>

# Root mean square error
rmse$aggr
## Aggregation function: test.rmse

rmse$aggr$fun
## function (task, perf.test, perf.train, measure, group, pred)
## sqrt(mean(perf.test^2))
## <bytecode: 0×23ad72d8>
## <environment: namespace:mlr>
```

You can change the aggregation method of a `Measure` (`makeMeasure()` (../../reference/makeMeasure.html)) via function `setAggregation()` (../../reference/setAggregation.html). All available aggregation schemes are listed on the `aggregations()` (../../reference/aggregations.html) documentation page.

Example: One measure with different aggregations

The aggregation schemes `test.median` (`aggregations()` (../../reference/aggregations.html)), `test.min` (`aggregations()` (../../reference/aggregations.html)), and `text.max` (`aggregations()` (../../reference/aggregations.html)) compute the median, minimum, and maximum of the performance values on the test sets.

```
## Resampling: cross-validation
## Measures:             mse       mse       mse       mse
## [Resample] iter 1:    21.043561321.043561321.043561321.0435613
## [Resample] iter 2:    27.737793427.737793427.737793427.7377934
## [Resample] iter 3:    23.526411123.526411123.526411123.5264111
##
## Aggregated Result: mse.test.mean=24.1025886,mse.test.median=23.5264111,mse.tes
##
```

```
mseTestMedian = setAggregation(../../reference/setAggregation.html)(mse, test.me
mseTestMin = setAggregation(../../reference/setAggregation.html)(mse, test.min)
mseTestMax = setAggregation(../../reference/setAggregation.html)(mse, test.max)

mseTestMedian
## Name: Mean of squared errors
## Performance measure: mse
## Properties: regr,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: Inf
## Aggregated by: test.median
## Arguments:
## Note: Defined as: mean((response - truth)^2)

rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3)
r = resample(../../reference/resample.html)("regr.lm", bh.task, rdesc, measures
## Resampling: cross-validation
## Measures:             mse       mse       mse       mse
## [Resample] iter 1:    24.078202624.078202624.078202624.0782026
## [Resample] iter 2:    29.498307729.498307729.498307729.4983077
## [Resample] iter 3:    18.689471818.689471818.689471818.6894718
##
## Aggregated Result: mse.test.mean=24.0886607,mse.test.median=24.0782026,mse.tes
##

r
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm
## Aggr perf: mse.test.mean=24.0886607,mse.test.median=24.0782026,mse.test.min=18
## Runtime: 0.0288048

r$aggr
##   mse.test.mean mse.test.median    mse.test.min    mse.test.max
##        24.08866        24.07820        18.68947        29.49831
```

Example: Calculating the training error

Below we calculate the mean misclassification error (mmce (measures.html)) on the training and the test data sets. Note that we have to set `predict = "both"` when calling `makeResampleDesc()` (../../reference/makeResampleDesc.html) in order to get predictions on both training and test sets.

```
mmceTrainMean = setAggregation(../../reference/setAggregation.html)(mmce, train.
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("CV", iters = 3,
r = resample(../../reference/resample.html)("classif.rpart", iris.task, rdesc, m
## Resampling: cross-validation
## Measures:             mmce.train   mmce.test
## [Resample] iter 1:    0.0300000    0.1000000
## [Resample] iter 2:    0.0500000    0.0200000
## [Resample] iter 3:    0.0200000    0.0800000
##
## Aggregated Result: mmce.test.mean=0.0666667,mmce.train.mean=0.0333333
##

r$measures.train
##   iter mmce mmce
## 1    1 0.03 0.03
## 2    2 0.05 0.05
## 3    3 0.02 0.02

r$aggr
##  mmce.test.mean mmce.train.mean
##      0.06666667      0.03333333
```

Example: Bootstrap

In *out-of-bag bootstrap estimation* B new data sets $D^{*1}, \ldots, D^{*B}$ are drawn from the data set D with replacement, each of the same size as D. In the b-th iteration, D^{*b} forms the training set, while the remaining elements from D, i.e., $D \setminus D^{*b}$, form the test set.

The *b632* and *b632+* variants calculate a convex combination of the training performance and the out-of-bag bootstrap performance and thus require predictions on the training sets and an appropriate aggregation strategy.

```
# Use bootstrap as resampling strategy and predict on both train and test sets
rdesc = makeResampleDesc(../../reference/makeResampleDesc.html)("Bootstrap", pre

# Set aggregation schemes for b632 and b632+ bootstrap
mmceB632 = setAggregation(../../reference/setAggregation.html)(mmce, b632)
mmceB632plus = setAggregation(../../reference/setAggregation.html)(mmce, b632plu

mmceB632
## Name: Mean misclassification error
## Performance measure: mmce
## Properties: classif,classif.multi,req.pred,req.truth
## Minimize: TRUE
## Best: 0; Worst: 1
## Aggregated by: b632
## Arguments: list()
## Note: Defined as: mean(response != truth)

r = resample(../../reference/resample.html)("classif.rpart", iris.task, rdesc, r
  show.info = FALSE)
head(r$measures.train)
##   iter       mmce       mmce       mmce
## 1    1 0.04000000 0.04000000 0.04000000
## 2    2 0.04000000 0.04000000 0.04000000
## 3    3 0.02666667 0.02666667 0.02666667
## 4    4 0.02000000 0.02000000 0.02000000
## 5    5 0.02000000 0.02000000 0.02000000
## 6    6 0.02000000 0.02000000 0.02000000

# Compare misclassification rates for out-of-bag, b632, and b632+ bootstrap
r$aggr
## mmce.test.mean      mmce.b632  mmce.b632plus
##     0.05751504     0.04640817     0.04751579
```

Convenience functions

The functionality described on this page allows for much control and flexibility. However, when quickly trying out some learners, it can get tedious to type all the code for defining the resampling strategy, setting the aggregation scheme and so on. As mentioned above, `mlr` includes some pre-defined resample description objects for frequently used strategies like, e.g., 5-fold cross-validation (`cv5` (`makeResampleDesc()` (../../reference/makeResampleDesc.html))). Moreover, `mlr` provides special functions for the most common resampling methods, for example `holdout` (`resample()` (../../reference/resample.html)), `crossval` (`resample()` (../../reference/resample.html)), or `bootstrapB632` (`resample()` (../../reference/resample.html)).

```
## Resampling: cross-validation
## Measures:             mmce      ber
## [Resample] iter 1:    0.0200000 0.0185185
## [Resample] iter 2:    0.0000000 0.0000000
## [Resample] iter 3:    0.0600000 0.0640523
##
## Aggregated Result: mmce.test.mean=0.0266667,ber.test.mean=0.0275236
##
## Resampling: OOB bootstrapping
## Measures:             mse.train   mae.train   mse.test    mae.test
## [Resample] iter 1:    17.8574197  3.0527442   25.0329081  3.4522670
## [Resample] iter 2:    21.2690350  3.2048943   21.0377962  3.4237734
## [Resample] iter 3:    16.3170950  2.8646150   30.1866248  3.8006483
##
## Aggregated Result: mse.b632plus=23.0152530,mae.b632plus=3.3816457
##
```

```
crossval (../../reference/resample.html)("classif.lda", iris.task, iters = 3, mea
## Resampling: cross-validation
## Measures:             mmce      ber
## [Resample] iter 1:    0.0200000 0.0238095
## [Resample] iter 2:    0.0400000 0.0370370
## [Resample] iter 3:    0.0000000 0.0000000
##
## Aggregated Result: mmce.test.mean=0.0200000,ber.test.mean=0.0202822
##
## Resample Result
## Task: iris-example
## Learner: classif.lda
## Aggr perf: mmce.test.mean=0.0200000,ber.test.mean=0.0202822
## Runtime: 0.0205457

bootstrapB632plus (../../reference/resample.html)("regr.lm", bh.task, iters = 3,
## Resampling: OOB bootstrapping
## Measures:             mse.train   mae.train   mse.test    mae.test
## [Resample] iter 1:    24.6425511  3.4107320   16.3415466  3.0123263
## [Resample] iter 2:    17.0963191  2.9809210   29.6056968  3.7131236
## [Resample] iter 3:    23.1440608  3.5079975   24.4183753  3.3467443
##
## Aggregated Result: mse.b632plus=22.9359054,mae.b632plus=3.3459751
##
## Resample Result
## Task: BostonHousing-example
## Learner: regr.lm
## Aggr perf: mse.b632plus=22.9359054,mae.b632plus=3.3459751
## Runtime: 0.0419419
```