Package 'mlr3filters'

July 23, 2025

Title Filter Based Feature Selection for 'mlr3'

Version 0.8.1

Description Extends 'mlr3' with filter methods for feature selection. Besides standalone filter methods built-in methods of any machine-learning algorithm are supported. Partial scoring of multivariate filter methods is supported.

License LGPL-3

URL https://mlr3filters.mlr-org.com, https://github.com/mlr-org/mlr3filters

BugReports https://github.com/mlr-org/mlr3filters/issues

Depends R (>= 3.1.0)

Imports backports, checkmate, data.table, mlr3 (>= 0.12.0), mlr3misc, paradox, R6

Suggests Boruta, care, caret, carSurv, FSelectorRcpp, knitr, lgr, mlr3learners, mlr3measures, mlr3pipelines, praznik, rpart, survival, testthat (>= 3.0.0), withr

Config/testthat/edition 3

Encoding UTF-8

NeedsCompilation no

RoxygenNote 7.3.2

Collate 'Filter.R' 'mlr_filters.R' 'FilterAUC.R' 'FilterAnova.R' 'FilterBoruta.R' 'FilterCMIM.R' 'FilterCarScore.R' 'FilterCarSurvScore.R' 'FilterCorrelation.R' 'FilterDISR.R' 'FilterFindCorrelation.R' 'FilterLearner.R' 'FilterImportance.R' 'FilterInformationGain.R' 'FilterJMI.R' 'FilterJMIM.R' 'FilterKruskalTest.R' 'FilterMIM.R' 'FilterMRMR.R' 'FilterNJMIM.R' 'FilterPerformance.R' 'FilterPermutation.R' 'FilterRelief.R' 'FilterSelectedFeatures.R' 'FilterUnivariateCox.R' 'FilterVariance.R' 'bibentries.R' 'flt.R' 'helper.R' 'reexports.R' 'zzz.R'

Author Marc Becker [cre, aut] (ORCID: <https://orcid.org/0000-0002-8115-0400>),
Patrick Schratz [aut] (ORCID: <https://orcid.org/0000-0003-0748-6624>),
Michel Lang [aut] (ORCID: <https://orcid.org/0000-0001-9754-0393>),
Bernd Bischl [aut] (ORCID: <https://orcid.org/0000-0001-6002-6980>),
Martin Binder [aut],
John Zobolas [aut] (ORCID: <https://orcid.org/0000-0002-3609-8674>)

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication 2024-11-08 11:30:02 UTC

Contents

`mlr3filters-package` *mlr3filters: Filter Based Feature Selection for 'mlr3'*

Description

Extends 'mlr3' with filter methods for feature selection. Besides standalone filter methods built-in methods of any machine-learning algorithm are supported. Partial scoring of multivariate filter methods is supported.

Author(s)

Maintainer: Marc Becker <marcbecker@posteo.de> (ORCID)

Authors:

- Patrick Schratz <patrick.schratz@gmail.com> (ORCID)
- Michel Lang <michellang@gmail.com> (ORCID)
- Bernd Bischl <bernd_bischl@gmx.net> (ORCID)
- Martin Binder <mlr.developer@mb706.com>
- John Zobolas <bblodfon@gmail.com> (ORCID)

See Also

Useful links:

- https://mlr3filters.mlr-org.com
- https://github.com/mlr-org/mlr3filters
- Report bugs at https://github.com/mlr-org/mlr3filters/issues

`Filter` *Filter Base Class*

Description

Base class for filters. Predefined filters are stored in the dictionary mlr_filters. A Filter calculates a score for each feature of a task. Important features get a large value and unimportant features get a small value. Note that filter scores may also be negative.

Details

Some features support partial scoring of the feature set: If `nfeat` is not `NULL`, only the best `nfeat` features are guaranteed to get a score. Additional features may be ignored for computational reasons, and then get a score value of `NA`.

Public fields

`id (character(1))`
: Identifier of the object. Used in tables, plot and text output.

`label (character(1))`
: Label for this object. Can be used in tables, plot and text output instead of the ID.

`task_types (character())`
: Set of supported task types, e.g. `"classif"` or `"regr"`. Can be set to the scalar value `NA` to allow any task type.

  For a complete list of possible task types (depending on the loaded packages), see `mlr_reflections$task_types$type`

`task_properties (character())`
: mlr3::Tasktask properties.

`param_set` (paradox::ParamSet)
: Set of hyperparameters.

`feature_types (character())`
: Feature types of the filter.

`packages (character())`
: Packages which this filter is relying on.

`man (character(1))`
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. Defaults to `NA`, but can be set by child classes.

`scores` Stores the calculated filter score values as named numeric vector. The vector is sorted in decreasing order with possible `NA` values last. The more important the feature, the higher the score. Tied values (this includes `NA` values) appear in a random, non-deterministic order.

Active bindings

`properties (character())`
: Properties of the filter. Currently, only `"missings"` is supported. A filter has the property `"missings"`, iff the filter can handle missing values in the features in a graceful way. Otherwise, an assertion is thrown if missing values are detected.

`hash (character(1))`
: Hash (unique identifier) for this object.

`phash (character(1))`
: Hash (unique identifier) for this partial object, excluding some components which are varied systematically during tuning (parameter values) or feature selection (feature names).

Methods

Public methods:

- `Filter$new()`
- `Filter$format()`
- `Filter$print()`
- `Filter$help()`
- `Filter$calculate()`

- `Filter$clone()`

Method `new()`**:** Create a Filter object.

Usage:

```
Filter$new(
  id,
  task_types,
  task_properties = character(),
  param_set = ps(),
  feature_types = character(),
  packages = character(),
  label = NA_character_,
  man = NA_character_
)
```

Arguments:

`id (character(1))`
: Identifier for the filter.

`task_types (character())`
: Types of the task the filter can operator on. E.g., `"classif"` or `"regr"`. Can be set to scalar `NA` to allow any task type.

`task_properties (character())`
: Required task properties, see mlr3::Task. Must be a subset of `mlr_reflections$task_properties`.

`param_set (paradox::ParamSet)`
: Set of hyperparameters.

`feature_types (character())`
: Feature types the filter operates on. Must be a subset of `mlr_reflections$task_feature_types`.

`packages (character())`
: Set of required packages. Note that these packages will be loaded via `requireNamespace()`, and are not attached.

`label (character(1))`
: Label for the new instance.

`man (character(1))`
: String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

Method `format()`**:** Format helper for Filter class

Usage:

```
Filter$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer for Filter class

Usage:

```
Filter$print()
```

Method `help()`**:** Opens the corresponding help page referenced by field `$man`.

Usage:

```
Filter$help()
```

Method `calculate()`**:** Calculates the filter score values for the provided mlr3::Task and stores them in field `scores`. `nfeat` determines the minimum number of features to score (see details), and defaults to the number of features in `task`. Loads required packages and then calls `private$.calculate()` of the respective subclass.
This private method is is expected to return a numeric vector, uniquely named with (a subset of) feature names. The returned vector may have missing values. Features with missing values as well as features with no calculated score are automatically ranked last, in a random order. If the task has no rows, each feature gets the score `NA`.

Usage:

```
Filter$calculate(task, nfeat = NULL)
```

Arguments:

`task` (mlr3::Task)
mlr3::Task to calculate the filter scores for.

`nfeat` (`integer()`)
The minimum number of features to calculate filter scores for.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Filter$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Filter: `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

`flt` *Syntactic Sugar for Filter Construction*

Description

These functions complements mlr_filters with a function in the spirit of mlr3::mlr_sugar.

Usage

```
flt(.key, ...)

flts(.keys, ...)
```

Arguments

.key	(character(1)) Key passed to the respective dictionary to retrieve the object.
...	(any) Additional arguments.
.keys	(character()) Keys passed to the respective dictionary to retrieve multiple objects.

Value

Filter.

Examples

```
flt("correlation", method = "kendall")
flts(c("mrmr", "jmim"))
```

---

mlr_filters *Dictionary of Filters*

Description

A simple mlr3misc::Dictionary storing objects of class Filter. Each Filter has an associated help page, see mlr_filters_[id].

This dictionary can get populated with additional filters by add-on packages.

For a more convenient way to retrieve and construct filters, see flt().

Usage

```
mlr_filters
```

Format

R6::R6Class object

Usage

See mlr3misc::Dictionary.

See Also

Other Filter: Filter, mlr_filters_anova, mlr_filters_auc, mlr_filters_boruta, mlr_filters_carscore, mlr_filters_carsurvscore, mlr_filters_cmim, mlr_filters_correlation, mlr_filters_disr, mlr_filters_find_correlation, mlr_filters_importance, mlr_filters_information_gain, mlr_filters_jmi, mlr_filters_jmim, mlr_filters_kruskal_test, mlr_filters_mim, mlr_filters_mrmr, mlr_filters_njmim, mlr_filters_performance, mlr_filters_permutation, mlr_filters_relief, mlr_filters_selected_features, mlr_filters_univariate_cox, mlr_filters_variance

Examples

```
mlr_filters$keys()
as.data.table(mlr_filters)
mlr_filters$get("mim")
flt("anova")
```

`mlr_filters_anova` *ANOVA F-Test Filter*

Description

ANOVA F-Test filter calling `stats::aov()`. Note that this is equivalent to a t-test for binary classification.

The filter value is `-log10(p)` where `p` is the p-value. This transformation is necessary to ensure numerical stability for very small p-values.

Super class

`mlr3filters::Filter` -> `FilterAnova`

Methods

Public methods:

- `FilterAnova$new()`
- `FilterAnova$clone()`

Method `new()`**:** Create a FilterAnova object.

Usage:

```
FilterAnova$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterAnova$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
task = mlr3::tsk("iris")
filter = flt("anova")
filter$calculate(task)
head(as.data.table(filter), 3)

# transform to p-value
10^(-filter$scores)

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("anova"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_auc` *AUC Filter*

Description

Area under the (ROC) Curve filter, analogously to `mlr3measures::auc()` from **mlr3measures**. Missing values of the features are removed before calculating the AUC. If the AUC is undefined for the input, it is set to 0.5 (random classifier). The absolute value of the difference between the AUC and 0.5 is used as final filter value.

Super class

`mlr3filters::Filter` -> `FilterAUC`

Methods

Public methods:

- `FilterAUC$new()`
- `FilterAUC$clone()`

Method `new()`**:** Create a FilterAUC object.

Usage:

```
FilterAUC$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterAUC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
task = mlr3::tsk("sonar")
filter = flt("auc")
filter$calculate(task)
head(as.data.table(filter), 3)

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("auc"), filter.frac = 0.5) %>>%
```

```
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_boruta` *Burota Filter*

Description

Filter using the Boruta algorithm for feature selection. If `keep = "tentative"`, confirmed and tentative features are returned. Note that there is no ordering in the selected features. Selected features get a score of 1, deselected features get a score of 0. The order of selected features is random. In combination with **mlr3pipelines**, only the filter criterion `cutoff` makes sense.

Initial parameter values

- `num.threads`:
  - Actual default: `NULL`, triggering auto-detection of the number of CPUs.
  - Adjusted value: 1.
  - Reason for change: Conflicting with parallelization via **future**.

Super class

`mlr3filters::Filter` -> `FilterBoruta`

Methods

Public methods:

- `FilterBoruta$new()`
- `FilterBoruta$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
FilterBoruta$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterBoruta$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kursa MB, Rudnicki WR (2010). "Feature Selection with the Boruta Package." *Journal of Statistical Software*, **36**(11), 1-13.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("Boruta")) {
 task = mlr3::tsk("sonar")
 filter = flt("boruta")
 filter$calculate(task)
 as.data.table(filter)
}
```

---

mlr_filters_carscore *Correlation-Adjusted Marignal Correlation Score Filter*

Description

Calculates the Correlation-Adjusted (marginal) coRrelation scores (short CAR scores) implemented in `care::carscore()` in package **care**. The CAR scores for a set of features are defined as the correlations between the target and the decorrelated features. The filter returns the absolute value of the calculated scores.

Argument `verbose` defaults to `FALSE`.

Super class

`mlr3filters::Filter` -> `FilterCarScore`

Methods

Public methods:

- `FilterCarScore$new()`
- `FilterCarScore$clone()`

Method `new()`**:** Create a FilterCarScore object.

Usage:

```
FilterCarScore$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterCarScore$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: Filter, mlr_filters, mlr_filters_anova, mlr_filters_auc, mlr_filters_boruta, mlr_filters_carsurvscore, mlr_filters_cmim, mlr_filters_correlation, mlr_filters_disr, mlr_filters_find_correlation, mlr_filters_importance, mlr_filters_information_gain, mlr_filters_jmi, mlr_filters_jmim, mlr_filters_kruskal_test, mlr_filters_mim, mlr_filters_mrmr, mlr_filters_njmim, mlr_filters_performance, mlr_filters_permutation, mlr_filters_relief, mlr_filters_selected_features, mlr_filters_univariate_cox, mlr_filters_variance

Examples

```
if (requireNamespace("care")) {
  task = mlr3::tsk("mtcars")
  filter = flt("carscore")
  filter$calculate(task)
  head(as.data.table(filter), 3)

  ## changing the filter settings
  filter = flt("carscore")
  filter$param_set$values = list("diagonal" = TRUE)
  filter$calculate(task)
  head(as.data.table(filter), 3)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "care", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("mtcars")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("carscore"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("regr.rpart"))

  graph$train(task)
}
```

---

mlr_filters_carsurvscore

Correlation-Adjusted Survival Score Filter

Description

Calculates CARS scores for right-censored survival tasks. Calls the implementation in `carSurv::carSurvScore()` in package **carSurv**.

Super class

`mlr3filters::Filter` -> `FilterCarSurvScore`

Methods

Public methods:

- `FilterCarSurvScore$new()`
- `FilterCarSurvScore$clone()`

Method `new()`**:** Create a FilterCarSurvScore object.

Usage:

```
FilterCarSurvScore$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterCarSurvScore$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Bommert A, Welchowski T, Schmid M, Rahnenführer J (2021). "Benchmark of filter methods for feature selection in high-dimensional gene expression survival data." *Briefings in Bioinformatics*, **23**(1). doi:10.1093/bib/bbab354.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

`mlr_filters_cmim` *Minimal Conditional Mutual Information Maximization Filter*

Description

Minimal conditional mutual information maximization filter calling `praznik::CMIM()` from package **praznik**.

This filter supports partial scoring (see Filter).

Details

As the scores calculated by the **praznik** package are not monotone due to the greedy forward fashion, the returned scores simply reflect the selection order: `1`, `(k-1)/k`, ..., `1/k` where `k` is the number of selected features.

Threading is disabled by default (hyperparameter `threads` is set to 1). Set to a number `>= 2` to enable threading, or to `0` for auto-detecting the number of available cores.

Super class

`mlr3filters::Filter` -> `FilterCMIM`

Methods

Public methods:

- `FilterCMIM$new()`
- `FilterCMIM$clone()`

Method `new()`**:** Create a FilterCMIM object.

Usage:

```
FilterCMIM$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterCMIM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kursa MB (2021). "Praznik: High performance information-based feature selection." *SoftwareX*, **16**, 100819. doi:10.1016/j.softx.2021.100819.

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: Filter, mlr_filters, mlr_filters_anova, mlr_filters_auc, mlr_filters_boruta, mlr_filters_carscore, mlr_filters_carsurvscore, mlr_filters_correlation, mlr_filters_disr, mlr_filters_find_correlation, mlr_filters_importance, mlr_filters_information_gain, mlr_filters_jmi, mlr_filters_jmim, mlr_filters_kruskal_test, mlr_filters_mim, mlr_filters_mrmr, mlr_filters_njmim, mlr_filters_performance, mlr_filters_permutation, mlr_filters_relief, mlr_filters_selected_features, mlr_filters_univariate_cox, mlr_filters_variance

Examples

```
if (requireNamespace("praznik")) {
  task = mlr3::tsk("iris")
  filter = flt("cmim")
  filter$calculate(task, nfeat = 2)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "praznik"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("cmim"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

---

mlr_filters_correlation

Correlation Filter

Description

Simple correlation filter calling stats::cor(). The filter score is the absolute value of the correlation.

Super class

mlr3filters::Filter -> FilterCorrelation

Methods

Public methods:

- `FilterCorrelation$new()`
- `FilterCorrelation$clone()`

Method `new()`**:** Create a FilterCorrelation object.

Usage:

```
FilterCorrelation$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterCorrelation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

This filter, in its default settings, can handle missing values in the features. However, the resulting filter scores may be misleading or at least difficult to compare if some features have a large proportion of missing values.

If a feature has no non-missing value, the resulting score will be `NA`. Missing scores appear in a random, non-deterministic order at the end of the vector of scores.

References

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
## Pearson (default)
task = mlr3::tsk("mtcars")
filter = flt("correlation")
filter$calculate(task)
as.data.table(filter)

## Spearman
filter = FilterCorrelation$new()
filter$param_set$values = list("method" = "spearman")
filter$calculate(task)
as.data.table(filter)
if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("mtcars")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("correlation"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("regr.rpart"))

  graph$train(task)
}
```

`mlr_filters_disr` *Double Input Symmetrical Relevance Filter*

Description

Double input symmetrical relevance filter calling `praznik::DISR()` from package **praznik**.

This filter supports partial scoring (see Filter).

Details

As the scores calculated by the **praznik** package are not monotone due to the greedy forward fashion, the returned scores simply reflect the selection order: `1`, `(k-1)/k`, ..., `1/k` where `k` is the number of selected features.

Threading is disabled by default (hyperparameter `threads` is set to 1). Set to a number `>= 2` to enable threading, or to `0` for auto-detecting the number of available cores.

Super class

`mlr3filters::Filter` -> `FilterDISR`

Methods

Public methods:

- `FilterDISR$new()`
- `FilterDISR$clone()`

Method `new()`**:** Create a FilterDISR object.

Usage:

```
FilterDISR$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterDISR$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kursa MB (2021). "Praznik: High performance information-based feature selection." *SoftwareX*, **16**, 100819. doi:10.1016/j.softx.2021.100819.

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("praznik")) {
  task = mlr3::tsk("iris")
  filter = flt("disr")
  filter$calculate(task)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "praznik"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")
```

```
  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("disr"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_find_correlation` *Correlation Filter*

Description

Simple filter emulating `caret::findCorrelation(exact = FALSE)`.

This gives each feature a score between 0 and 1 that is *one minus* the cutoff value for which it is excluded when using `caret::findCorrelation()`. The negative is used because `caret::findCorrelation()` excludes everything *above* a cutoff, while filters exclude everything below a cutoff. Here the filter scores are shifted by +1 to get positive values for to align with the way other filters work.

Subsequently `caret::findCorrelation(cutoff = 0.9)` lists the same features that are excluded with `FilterFindCorrelation` at score 0.1 (= 1 - 0.9).

Super class

`mlr3filters::Filter` -> `FilterFindCorrelation`

Methods

Public methods:

- `FilterFindCorrelation$new()`
- `FilterFindCorrelation$clone()`

Method `new()`**:** Create a FilterFindCorrelation object.

Usage:

```
FilterFindCorrelation$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterFindCorrelation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
# Pearson (default)
task = mlr3::tsk("mtcars")
filter = flt("find_correlation")
filter$calculate(task)
as.data.table(filter)

## Spearman
filter = flt("find_correlation", method = "spearman")
filter$calculate(task)
as.data.table(filter)

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("find_correlation"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_importance`

Filter for Embedded Feature Selection via Variable Importance

---

Description

Variable Importance filter using embedded feature selection of machine learning algorithms. Takes a mlr3::Learner which is capable of extracting the variable importance (property "importance"), fits the model and extracts the importance values to use as filter scores.

Super classes

`mlr3filters::Filter` -> `mlr3filters::FilterLearner` -> `FilterImportance`

Public fields

`learner` (mlr3::Learner)
Learner to extract the importance values from.

Methods

Public methods:

- `FilterImportance$new()`
- `FilterImportance$clone()`

Method `new()`**:** Create a FilterImportance object.

Usage:

```
FilterImportance$new(learner = mlr3::lrn("classif.featureless"))
```

Arguments:

`learner` (mlr3::Learner)
Learner to extract the importance values from.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterImportance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("rpart")) {
  task = mlr3::tsk("iris")
  learner = mlr3::lrn("classif.rpart")
  filter = flt("importance", learner = learner)
  filter$calculate(task)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "mlr3learners"), quietly = TRUE)) {
  library("mlr3learners")
  library("mlr3pipelines")
```

```
  task = mlr3::tsk("sonar")

  learner = mlr3::lrn("classif.rpart")

  # Note: `filter.frac` is selected randomly and should be tuned.

 graph = po("filter", filter = flt("importance", learner = learner), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.log_reg"))

  graph$train(task)
}
```

`mlr_filters_information_gain`
Information Gain Filter

Description

Information gain filter calling `FSelectorRcpp::information_gain()` in package **FSelectorRcpp**. Set parameter `"type"` to `"gainratio"` to calculate the gain ratio, or set to `"symuncert"` to calculate the symmetrical uncertainty (see `FSelectorRcpp::information_gain()`). Default is `"infogain"`.

Argument `equal` defaults to `FALSE` for classification tasks, and to `TRUE` for regression tasks.

Super class

`mlr3filters::Filter` -> `FilterInformationGain`

Methods

Public methods:

- `FilterInformationGain$new()`
- `FilterInformationGain$clone()`

Method `new()`**:** Create a FilterInformationGain object.

Usage:

```
FilterInformationGain$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterInformationGain$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: Filter, mlr_filters, mlr_filters_anova, mlr_filters_auc, mlr_filters_boruta, mlr_filters_carscore, mlr_filters_carsurvscore, mlr_filters_cmim, mlr_filters_correlation, mlr_filters_disr, mlr_filters_find_correlation, mlr_filters_importance, mlr_filters_jmi, mlr_filters_jmim, mlr_filters_kruskal_test, mlr_filters_mim, mlr_filters_mrmr, mlr_filters_njmim, mlr_filters_performance, mlr_filters_permutation, mlr_filters_relief, mlr_filters_selected_features, mlr_filters_univariate_cox, mlr_filters_variance

Examples

```
if (requireNamespace("FSelectorRcpp")) {
  ## InfoGain (default)
  task = mlr3::tsk("sonar")
  filter = flt("information_gain")
  filter$calculate(task)
  head(filter$scores, 3)
  as.data.table(filter)

  ## GainRatio

  filterGR = flt("information_gain")
  filterGR$param_set$values = list("type" = "gainratio")
  filterGR$calculate(task)
  head(as.data.table(filterGR), 3)

}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "FSelectorRcpp", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("information_gain"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)

}
```

---

mlr_filters_jmi *Joint Mutual Information Filter*

Description

Joint mutual information filter calling praznik::JMI() in package **praznik**.

This filter supports partial scoring (see Filter).

Details

As the scores calculated by the **praznik** package are not monotone due to the greedy forward fashion, the returned scores simply reflect the selection order: `1`, `(k-1)/k`, ..., `1/k` where `k` is the number of selected features.

Threading is disabled by default (hyperparameter `threads` is set to 1). Set to a number `>= 2` to enable threading, or to `0` for auto-detecting the number of available cores.

Super class

`mlr3filters::Filter` -> `FilterJMI`

Methods

Public methods:

- `FilterJMI$new()`
- `FilterJMI$clone()`

Method `new()`**:** Create a FilterJMI object.

Usage:

```
FilterJMI$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterJMI$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kursa MB (2021). "Praznik: High performance information-based feature selection." *SoftwareX*, **16**, 100819. doi:10.1016/j.softx.2021.100819.

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("praznik")) {
  task = mlr3::tsk("iris")
  filter = flt("jmi")
  filter$calculate(task, nfeat = 2)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "praznik"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("jmi"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

---

mlr_filters_jmim *Minimal Joint Mutual Information Maximization Filter*

Description

Minimal joint mutual information maximization filter calling `praznik::JMIM()` in package **praznik**.

This filter supports partial scoring (see Filter).

Details

As the scores calculated by the **praznik** package are not monotone due to the greedy forward fashion, the returned scores simply reflect the selection order: `1`, `(k-1)/k`, ..., `1/k` where `k` is the number of selected features.

Threading is disabled by default (hyperparameter `threads` is set to 1). Set to a number `>= 2` to enable threading, or to `0` for auto-detecting the number of available cores.

Super class

`mlr3filters::Filter` -> `FilterJMIM`

Methods

Public methods:

- `FilterJMIM$new()`
- `FilterJMIM$clone()`

Method `new()`**:** Create a FilterJMIM object.

Usage:

```
FilterJMIM$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterJMIM$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

Kursa MB (2021). "Praznik: High performance information-based feature selection." *SoftwareX*, **16**, 100819. doi:10.1016/j.softx.2021.100819.

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("praznik")) {
  task = mlr3::tsk("iris")
  filter = flt("jmim")
  filter$calculate(task, nfeat = 2)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "praznik"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("jmim"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_kruskal_test` *Kruskal-Wallis Test Filter*

Description

Kruskal-Wallis rank sum test filter calling `stats::kruskal.test()`.

The filter value is `-log10(p)` where `p` is the p-value. This transformation is necessary to ensure numerical stability for very small p-values.

Super class

`mlr3filters::Filter` -> `FilterKruskalTest`

Methods

Public methods:

- `FilterKruskalTest$new()`
- `FilterKruskalTest$clone()`

Method `new()`**:** Create a FilterKruskalTest object.

Usage:

```
FilterKruskalTest$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterKruskalTest$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

This filter, in its default settings, can handle missing values in the features. However, the resulting filter scores may be misleading or at least difficult to compare if some features have a large proportion of missing values.

If a feature has not at least one non-missing observation per label, the resulting score will be NA. Missing scores appear in a random, non-deterministic order at the end of the vector of scores.

References

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
task = mlr3::tsk("iris")
filter = flt("kruskal_test")
filter$calculate(task)
as.data.table(filter)

# transform to p-value
10^(-filter$scores)

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("kruskal_test"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_mim` *Mutual Information Maximization Filter*

Description

Conditional mutual information based feature selection filter calling `praznik::MIM()` in package **praznik**.

This filter supports partial scoring (see Filter).

Details

As the scores calculated by the **praznik** package are not monotone due to the greedy forward fashion, the returned scores simply reflect the selection order: `1`, `(k-1)/k`, ..., `1/k` where `k` is the number of selected features.

Threading is disabled by default (hyperparameter `threads` is set to 1). Set to a number `>= 2` to enable threading, or to `0` for auto-detecting the number of available cores.

Super class

`mlr3filters::Filter` -> `FilterMIM`

Methods

Public methods:

- `FilterMIM$new()`
- `FilterMIM$clone()`

Method `new()`**:** Create a FilterMIM object.

Usage:

```
FilterMIM$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterMIM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kursa MB (2021). "Praznik: High performance information-based feature selection." *SoftwareX*, **16**, 100819. doi:10.1016/j.softx.2021.100819.

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("praznik")) {
  task = mlr3::tsk("iris")
  filter = flt("mim")
  filter$calculate(task, nfeat = 2)
  as.data.table(filter)
```

```
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "praznik"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("mim"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_mrmr` *Minimum Redundancy Maximal Relevancy Filter*

Description

Minimum redundancy maximal relevancy filter calling `praznik::MRMR()` in package **praznik**.

This filter supports partial scoring (see Filter).

Details

As the scores calculated by the **praznik** package are not monotone due to the greedy forward fashion, the returned scores simply reflect the selection order: `1`, `(k-1)/k`, ..., `1/k` where `k` is the number of selected features.

Threading is disabled by default (hyperparameter `threads` is set to 1). Set to a number `>= 2` to enable threading, or to `0` for auto-detecting the number of available cores.

Super class

`mlr3filters::Filter` -> `FilterMRMR`

Methods

Public methods:

- `FilterMRMR$new()`
- `FilterMRMR$clone()`

Method `new()`**:** Create a FilterMRMR object.

Usage:

```
FilterMRMR$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterMRMR$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kursa MB (2021). "Praznik: High performance information-based feature selection." *SoftwareX*, **16**, 100819. doi:10.1016/j.softx.2021.100819.

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("praznik")) {
  task = mlr3::tsk("iris")
  filter = flt("mrmr")
  filter$calculate(task, nfeat = 2)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "praznik"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("mrmr"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

---

mlr_filters_njmim *Minimal Normalised Joint Mutual Information Maximization Filter*

Description

Minimal normalised joint mutual information maximization filter calling `praznik::NJMIM()` from package **praznik**.

This filter supports partial scoring (see Filter).

Details

As the scores calculated by the **praznik** package are not monotone due to the greedy forward fashion, the returned scores simply reflect the selection order: `1`, `(k-1)/k`, ..., `1/k` where `k` is the number of selected features.

Threading is disabled by default (hyperparameter `threads` is set to 1). Set to a number `>= 2` to enable threading, or to `0` for auto-detecting the number of available cores.

Super class

`mlr3filters::Filter` -> `FilterNJMIM`

Methods

Public methods:

- `FilterNJMIM$new()`
- `FilterNJMIM$clone()`

Method `new()`**:** Create a FilterNJMIM object.

Usage:

```
FilterNJMIM$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterNJMIM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kursa MB (2021). "Praznik: High performance information-based feature selection." *SoftwareX*, **16**, 100819. doi:10.1016/j.softx.2021.100819.

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("praznik")) {
  task = mlr3::tsk("iris")
  filter = flt("njmim")
  filter$calculate(task, nfeat = 2)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart", "praznik"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("njmim"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

---

mlr_filters_performance

Predictive Performance Filter

Description

Filter which uses the predictive performance of a mlr3::Learner as filter score. Performs a `mlr3::resample()` for each feature separately. The filter score is the aggregated performance of the mlr3::Measure, or the negated aggregated performance if the measure has to be minimized.

Super classes

`mlr3filters::Filter` -> `mlr3filters::FilterLearner` -> `FilterPerformance`

Public fields

`learner` (mlr3::Learner)

`resampling` (mlr3::Resampling)

`measure` (mlr3::Measure)

Methods

Public methods:

- `FilterPerformance$new()`

- `FilterPerformance$clone()`

Method `new()`**:** Create a FilterDISR object.

Usage:

```
FilterPerformance$new(
  learner = mlr3::lrn("classif.featureless"),
  resampling = mlr3::rsmp("holdout"),
  measure = NULL
)
```

Arguments:

`learner` (mlr3::Learner)
mlr3::Learner to use for model fitting.

`resampling` (mlr3::Resampling)
mlr3::Resampling to be used within resampling.

`measure` (mlr3::Measure)
mlr3::Measure to be used for evaluating the performance.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterPerformance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("rpart")) {
  task = mlr3::tsk("iris")
  learner = mlr3::lrn("classif.rpart")
  filter = flt("performance", learner = learner)
  filter$calculate(task)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("iris")
```

```
  l = lrn("classif.rpart")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("performance", learner = l), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_permutation`

Permutation Score Filter

Description

The permutation filter randomly permutes the values of a single feature in a mlr3::Task to break the association with the response. The permuted feature, together with the unmodified features, is used to perform a `mlr3::resample()`. The permutation filter score is the difference between the aggregated performance of the mlr3::Measure and the performance estimated on the unmodified mlr3::Task.

Parameters

`standardize logical(1)`
: Standardize feature importance by maximum score.

`nmc integer(1)`
: Number of Monte-Carlo iterations to use in computing the feature importance.

Super classes

`mlr3filters::Filter` -> `mlr3filters::FilterLearner` -> `FilterPermutation`

Public fields

`learner` (mlr3::Learner)

`resampling` (mlr3::Resampling)

`measure` (mlr3::Measure)

Active bindings

`hash (character(1))`
: Hash (unique identifier) for this object.

`phash (character(1))`
: Hash (unique identifier) for this partial object, excluding some components which are varied systematically during tuning (parameter values) or feature selection (feature names).

Methods

Public methods:

- `FilterPermutation$new()`
- `FilterPermutation$clone()`

Method `new()`**:** Create a FilterPermutation object.

Usage:

```
FilterPermutation$new(
  learner = mlr3::lrn("classif.featureless"),
  resampling = mlr3::rsmp("holdout"),
  measure = NULL
)
```

Arguments:

`learner` (mlr3::Learner)
: mlr3::Learner to use for model fitting.

`resampling` (mlr3::Resampling)
: mlr3::Resampling to be used within resampling.

`measure` (mlr3::Measure)
: mlr3::Measure to be used for evaluating the performance.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterPermutation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("rpart")) {
  learner = mlr3::lrn("classif.rpart")
  resampling = mlr3::rsmp("holdout")
  measure = mlr3::msr("classif.acc")
 filter = flt("permutation", learner = learner, measure = measure, resampling = resampling,
    nmc = 2)
  task = mlr3::tsk("iris")
```

```
  filter$calculate(task)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("iris")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("permutation", nmc = 2), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

`mlr_filters_relief` *RELIEF Filter*

Description

Information gain filter calling `FSelectorRcpp::relief()` in package **FSelectorRcpp**.

Super class

`mlr3filters::Filter` -> `FilterRelief`

Methods

Public methods:

- `FilterRelief$new()`
- `FilterRelief$clone()`

Method `new()`**:** Create a FilterRelief object.

Usage:

```
FilterRelief$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterRelief$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

This filter can handle missing values in the features. However, the resulting filter scores may be misleading or at least difficult to compare if some features have a large proportion of missing values.

If a feature has no non-missing observation, the resulting score will be (close to) 0.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: Filter, mlr_filters, mlr_filters_anova, mlr_filters_auc, mlr_filters_boruta, mlr_filters_carscore, mlr_filters_carsurvscore, mlr_filters_cmim, mlr_filters_correlation, mlr_filters_disr, mlr_filters_find_correlation, mlr_filters_importance, mlr_filters_information_gain, mlr_filters_jmi, mlr_filters_jmim, mlr_filters_kruskal_test, mlr_filters_mim, mlr_filters_mrmr, mlr_filters_njmim, mlr_filters_performance, mlr_filters_permutation, mlr_filters_selected_features, mlr_filters_univariate_cox, mlr_filters_variance

Examples

```
if (requireNamespace("FSelectorRcpp")) {
  ## Relief (default)
  task = mlr3::tsk("iris")
  filter = flt("relief")
  filter$calculate(task)
  head(filter$scores, 3)
  as.data.table(filter)
}

if (mlr3misc::require_namespaces(c("mlr3pipelines", "FSelectorRcpp", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("iris")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("relief"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

---

mlr_filters_selected_features

Filter for Embedded Feature Selection

Description

Filter using embedded feature selection of machine learning algorithms. Takes a mlr3::Learner which is capable of extracting the selected features (property "selected_features"), fits the model and extracts the selected features.

Note that contrary to mlr_filters_importance, there is no ordering in the selected features. Selected features get a score of 1, deselected features get a score of 0. The order of selected features is random and different from the order in the learner. In combination with **mlr3pipelines**, only the filter criterion `cutoff` makes sense.

Super classes

`mlr3filters::Filter` -> `mlr3filters::FilterLearner` -> `FilterSelectedFeatures`

Public fields

`learner` (mlr3::Learner)
Learner to extract the importance values from.

Methods

Public methods:

- `FilterSelectedFeatures$new()`
- `FilterSelectedFeatures$clone()`

Method `new()`**:** Create a FilterImportance object.

Usage:

```
FilterSelectedFeatures$new(learner = mlr3::lrn("classif.featureless"))
```

Arguments:

`learner` (mlr3::Learner)
Learner to extract the selected features from.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterSelectedFeatures$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_univariate_cox`, `mlr_filters_variance`

Examples

```
if (requireNamespace("rpart")) {
  task = mlr3::tsk("iris")
  learner = mlr3::lrn("classif.rpart")
  filter = flt("selected_features", learner = learner)
  filter$calculate(task)
  as.data.table(filter)
}
```

```
if (mlr3misc::require_namespaces(c("mlr3pipelines", "mlr3learners", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  library("mlr3learners")
  task = mlr3::tsk("sonar")

  filter = flt("selected_features", learner = lrn("classif.rpart"))

  # Note: All filter scores are either 0 or 1, i.e. setting `filter.cutoff = 0.5` means that
  # we select all "selected features".

  graph = po("filter", filter = filter, filter.cutoff = 0.5) %>>%
    po("learner", mlr3::lrn("classif.log_reg"))

  graph$train(task)
}
```

`mlr_filters_univariate_cox`

Univariate Cox Survival Filter

Description

Calculates scores for assessing the relationship between individual features and the time-to-event outcome (right-censored survival data) using a univariate Cox proportional hazards model. The goal is to determine which features have a statistically significant association with the event of interest, typically in the context of clinical or biomedical research.

This filter fits a Cox Proportional Hazards model using each feature independently and extracts the p-value that quantifies the significance of the feature's impact on survival. The filter value is `-log10(p)` where `p` is the p-value. This transformation is necessary to ensure numerical stability for very small p-values. Also higher values denote more important features. The filter works only for numeric features so please ensure that factor variables are properly encoded, e.g. using PipeOpEncode.

Super class

`mlr3filters::Filter` -> `FilterUnivariateCox`

Methods

Public methods:

- `FilterUnivariateCox$new()`
- `FilterUnivariateCox$clone()`

Method `new()`**:** Create a FilterUnivariateCox object.

Usage:

```
FilterUnivariateCox$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterUnivariateCox$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_variance`

Examples

```
filter = flt("univariate_cox")
filter
```

`mlr_filters_variance` *Variance Filter*

Description

Variance filter calling `stats::var()`.

Argument `na.rm` defaults to `TRUE` here.

Super class

`mlr3filters::Filter` -> `FilterVariance`

Methods

Public methods:

- `FilterVariance$new()`
- `FilterVariance$clone()`

Method `new()`**:** Create a FilterVariance object.

Usage:

```
FilterVariance$new()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FilterVariance$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

For a benchmark of filter methods:

Bommert A, Sun X, Bischl B, Rahnenführer J, Lang M (2020). "Benchmark for filter methods for feature selection in high-dimensional classification data." *Computational Statistics & Data Analysis*, **143**, 106839. doi:10.1016/j.csda.2019.106839.

See Also

- PipeOpFilter for filter-based feature selection.
- Dictionary of Filters: mlr_filters

Other Filter: `Filter`, `mlr_filters`, `mlr_filters_anova`, `mlr_filters_auc`, `mlr_filters_boruta`, `mlr_filters_carscore`, `mlr_filters_carsurvscore`, `mlr_filters_cmim`, `mlr_filters_correlation`, `mlr_filters_disr`, `mlr_filters_find_correlation`, `mlr_filters_importance`, `mlr_filters_information_gain`, `mlr_filters_jmi`, `mlr_filters_jmim`, `mlr_filters_kruskal_test`, `mlr_filters_mim`, `mlr_filters_mrmr`, `mlr_filters_njmim`, `mlr_filters_performance`, `mlr_filters_permutation`, `mlr_filters_relief`, `mlr_filters_selected_features`, `mlr_filters_univariate_cox`

Examples

```
task = mlr3::tsk("mtcars")
filter = flt("variance")
filter$calculate(task)
head(filter$scores, 3)
as.data.table(filter)

if (mlr3misc::require_namespaces(c("mlr3pipelines", "rpart"), quietly = TRUE)) {
  library("mlr3pipelines")
  task = mlr3::tsk("spam")

  # Note: `filter.frac` is selected randomly and should be tuned.

  graph = po("filter", filter = flt("variance"), filter.frac = 0.5) %>>%
    po("learner", mlr3::lrn("classif.rpart"))

  graph$train(task)
}
```

Index