Package ‘mlr3filters’

January 25, 2022

**Title**  Filter Based Feature Selection for 'mlr3'

**Version**  0.5.0

**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**  care, caret, FSelectorRcpp, knitr, lgr, mlr3learners, mlr3measures, praznik, rpart, testthat (>= 3.0.0)

**Config/testthat/edition**  3

**Config/testthat/parallel**  true

**Encoding**  UTF-8

**NeedsCompilation**  no

**RoxygenNote**  7.1.2

**Collate**  'Filter.R' 'mlr_filters.R' 'FilterAUC.R' 'FilterAnova.R'
'FilterCMIM.R' 'FilterCarScore.R' 'FilterCorrelation.R'
'FilterDISR.R' 'FilterFindCorrelation.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' 'FilterVariance.R'
'bibentries.R' 'flt.R' 'helper.R' 'reexports.R' 'zzz.R'

**Author**  Patrick Schratz [aut, cre] (<https://orcid.org/0000-0003-0748-6624>),
Michel Lang [aut] (<https://orcid.org/0000-0001-9754-0393>),
Bernd Bischl [aut] (<https://orcid.org/0000-0001-6002-6980>),
Martin Binder [aut]
Maintainer  Patrick Schratz <patrick.schratz@gmail.com>
Repository  CRAN
Date/Publication  2022-01-25 16:32:46 UTC

R topics documented:

mlr3filters-package ........................................... 2
Filter ................................................................. 3
flt ................................................................. 6
mlr_filters ....................................................... 6
mlr_filters_anova ............................................... 7
mlr_filters_auc ................................................ 8
mlr_filters_carscore ........................................... 10
mlr_filters_cmim ............................................... 11
mlr_filters_correlation ....................................... 12
mlr_filters_disr ............................................... 13
mlr_filters_find_correlation ................................. 15
mlr_filters_importance ....................................... 16
mlr_filters_information_gain ................................ 17
mlr_filters_jmi .................................................. 18
mlr_filters_jmim ............................................... 20
mlr_filters_kruskal_test ..................................... 21
mlr_filters_mim ............................................... 22
mlr_filters_mrmr ............................................... 24
mlr_filters_njmim .............................................. 25
mlr_filters_performance ...................................... 26
mlr_filters_permutation ..................................... 28
mlr_filters_relief ............................................. 29
mlr_filters_selected_features ............................... 30
mlr_filters_variance ......................................... 32

Index 34
Filter

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.

task_type (character(1))
   Task type, e.g. "classif" or "regr". Can be set to NA to allow all task types.
   For a complete list of possible task types (depending on the loaded packages), see mlr_reflections$task_types$type

task_properties (character())
   mlr3::Task task 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.

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_type,
  task_properties = character(),
  param_set = ps(),
  feature_types = character(),
  packages = character(),
  man = NA_character_
)

Arguments:

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

Task_type (character())
  Types of the task the filter can operator on. E.g., "classif" or "regr". Can be set to NA to allow all task types.

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

Param_set (paradigm::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.
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()

**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 de-
tails), 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_anova, mlr_filters_auc, 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_jmim, mlr_filters_jmi, mlr_filters_kruskal_test,
mlr_filters_mim, mlr_filters_mrmr, mlrFilters_njmim, mlr_filters_performance, mlr_filters_permutation,
mlr_filters_relief, mlr_filters_selected_features, mlr_filters_variance, mlr_filters...
**Description**

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

**Usage**

```r
flt(.key, ...)
flts(.keys, ...)
```

**Arguments**

- `.key` (character(1))
  Key passed to the respective dictionary to retrieve the object.

- `...` (named list())
  Named arguments passed to the constructor, to be set as parameters in the `paramset::ParamSet`, or to be set as public field. See `mlr3misc::dictionary_sugar_get()` for more details.

- `.keys` (character())
  Keys passed to the respective dictionary to retrieve multiple objects.

**Value**

Filter.

**Examples**

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

---

**mlr_filters**

**Dictionary of Filters**

**Description**

A simple 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

R6Class object

Usage

See Dictionary.

See Also


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 $-\log_{10}(p)$ 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:

See Also
Dictionary of Filters: mlr_filters

Examples

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

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

---

### 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

\texttt{mlr3filters::Filter} -> \texttt{FilterAUC}

Methods

Public methods:

- \texttt{FilterAUC$\text{new}()} \\
- \texttt{FilterAUC$\text{clone}()} \\

Method \texttt{\text{new}()}: Create a FilterAUC object.

Usage:

\texttt{FilterAUC$\text{new}()}

Method \texttt{\text{clone}()}: The objects of this class are cloneable with this method.

Usage:

\texttt{FilterAUC$\text{clone}(\text{deep} = \text{FALSE})}

Arguments:

- \texttt{\text{deep}}: Whether to make a deep clone.

References

For a benchmark of filter methods:


See Also

Dictionary of Filters: \texttt{mlr\_filters}

Other Filter: \texttt{Filter,mlr\_filters\_anova,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\_jmim,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\_variance,mlr\_filters}

Examples

\begin{verbatim}
  task = mlr3::tsk("pima")
  filter = flt("auc")
  filter$calculate(task)
  head(as.data.table(filter), 3)
\end{verbatim}
Conditional Mutual Information Based Feature Selection 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

Dictionary of Filters: mlr_filters

Examples

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

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

**mlr_filters_cmim**

*Minimal Conditional Mutual Information Filter*

**Description**

Minimal conditional mutual information maximisation 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:*

```r
FilterCMIM$new()
```

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

*Usage:*

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

*Arguments:*

- `deep`: Whether to make a deep clone.
References


For a benchmark of filter methods:

See Also

Dictionary of Filters: mlr_filters


Examples

```r
library(mlr3)
library(mlr3tasks)
library(mlr3filters)

# Load task
task = mlr3::tsk("iris")

# Create filter
filter = flt("cmim")

# Calculate filter scores
filter$calculate(task, nfeat = 2)

# Convert to data.table
as.data.table(filter)
```

---

### mlr3filters_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:**

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

**Arguments:**

- `deep` Whether to make a deep clone.

**References**

For a benchmark of filter methods:


**See Also**

Dictionary of Filters: `mlr_filters`


**Examples**

```r
## Pearson (default)

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

```r
## Spearman

```r
filter = FilterCorrelation$new()
filter$param_set$values = list("method" = "spearman")
filter$calculate(task)
as.data.table(filter)
```  

---

**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 $\geq 2$ 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:**

depth  Whether to make a deep clone.

References


For a benchmark of filter methods:


See Also

Dictionary of Filters: `mlr_filters`

Examples

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

---

**mlr3filters_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:*

```r
  FilterFindCorrelation$new()
```

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

*Usage:*

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

*Arguments:*

- `deep` Whether to make a deep clone.

**See Also**

Dictionary of Filters: `mlr_filters`

Examples

```r
## 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)
```

### 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 class**

`mlr3filters::Filter` -> `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.rpart"))`

**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.
mlr3 filters information gain

See Also

Dictionary of Filters: mlr3::mlr_filters
Other Filter: Filter, mlr3::mlr_filters_anova, mlr3::mlr_filters_auc, mlr3::mlr_filters_carscore, mlr3::mlr_filters_cmim, mlr3::mlr_filters_correlation, mlr3::mlr_filters_disr, mlr3::mlr_filters_find_correlation, mlr3::mlr_filters_information_gain, mlr3::mlr_filters_jmim, mlr3::mlr_filters_jmi, mlr3::mlr_filters_kruskal_test, mlr3::mlr_filters_mim, mlr3::mlr_filters_mrmr, mlr3::mlr_filters_njmim, mlr3::mlr_filters_performance, mlr3::mlr_filters_permutation, mlr3::mlr_filters_relieff, mlr3::mlr_filters_selected_features, mlr3::mlr_filters_variance, mlr3::mlr_filters

Examples

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

mlr3::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.
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


For a benchmark of filter methods:


See Also

Dictionary of Filters: mlr_filters


Examples

task = mlr3::tsk("iris")
filter = flt("jmi")
filter$calculate(task, nfeat = 2)
as.data.table(filter)
Description

Minimal joint mutual information maximisation 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:

depth Whether to make a deep clone.

References


For a benchmark of filter methods:

**mlr_filters_kruskal_test**

**See Also**

Dictionary of Filters: `mlr_filters`


**Examples**

```r
task = mlr3::tsk("iris")
filter = flt("jmi")
filter$calculate(task, nfeat = 2)
as.data.table(filter)
```

**Description**

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

The filter value is \(-\log_{10}(p)\) 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.
References

For a benchmark of filter methods:


See Also

Dictionary of Filters: mlr_filters


Examples

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

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

---

**mlr_filters_mim**  
Conditional Mutual Information Based Feature Selection 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


For a benchmark of filter methods:


See Also

Dictionary of Filters: mlr_filters


Examples

task = mlr3::tsk("iris")
filter = flt("mim")
filter$calculate(task, nfeat = 2)
as.data.table(filter)
Minimum redundancy maximal relevancy filter calling \texttt{praznik::MRMR()} in package \texttt{praznik}. This filter supports partial scoring (see \texttt{Filter}).

As the scores calculated by the \texttt{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 \texttt{threads} is set to 1). Set to a number \texttt{>= 2} to enable threading, or to \texttt{0} for auto-detecting the number of available cores.

\textbf{Super class}

\texttt{mlr3filters::Filter} -> \texttt{FilterMRMR}

\textbf{Methods}

\textbf{Public methods:}

- \texttt{FilterMRMR$new()}
- \texttt{FilterMRMR$clone()}

\textbf{Method} \texttt{new()}: Create a \texttt{FilterMRMR} object.

\textit{Usage:}

\texttt{FilterMRMR$new()}

\textbf{Method} \texttt{clone()}: The objects of this class are cloneable with this method.

\textit{Usage:}

\texttt{FilterMRMR$clone(deep = FALSE)}

\textit{Arguments:}

- \texttt{deep} Whether to make a deep clone.

\textbf{References}


For a benchmark of filter methods:

### **mlr_filters_njmim**

**Minimal Normalised Joint Mutual Information Maximisation Filter**

**Description**

Minimal normalised joint mutual information maximisation 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:

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

Arguments:

depth Whether to make a deep clone.

References


For a benchmark of filter methods:


See Also

Dictionary of Filters: `mlr_filters`


Examples

```r
task = mlr3::tsk("iris")
filter = flt("njmim")
filter$calculate(task, nfeat = 2)
as.data.table(filter)
```

---

**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 class**

`mlr3filters::Filter` → `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.rpart"),
  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

Dictionary of Filters: mlr_filters

Examples

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

---

**mlr_filters_permutation**

*Permutation 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 permutated 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 class**

`mlr3filters::Filter` -> `FilterPermutation`

**Public fields**

- `learner` (mlr3::Learner)
- `resampling` (mlr3::Resampling)
- `measure` (mlr3::Measure)

**Methods**

**Public methods:**

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

**Method** `new()`: Create a `FilterPermutation` object.
Usage:
FilterPermutation$new(
    learner = mlr3::lrn("classif.rpart"),
    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

Dictionary of Filters: mlr_filters


---

**mlr_filters_relief**  Information Gain 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.

See Also

Dictionary of Filters: `mlr_filters`


Examples

```r
## Relief (default)
task = mlr3::tsk("pima")
filter = flt("relief")
filter$calculate(task)
head(filter$scores, 3)
```

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 class

\texttt{mlr3filters::Filter} -> \texttt{FilterSelectedFeatures}

Public fields

learner (\texttt{mlr3::Learner})

Learner to extract the importance values from.

Methods

Public methods:

\begin{itemize}
  \item \texttt{FilterSelectedFeatures\$new()}
  \item \texttt{FilterSelectedFeatures\$clone()}
\end{itemize}

Method \texttt{new()}: Create a FilterImportance object.

\textit{Usage:}

\texttt{FilterSelectedFeatures\$new(learner = mlr3::lrn("classif.rpart"))}

\textit{Arguments:}

learner (\texttt{mlr3::Learner})

Learner to extract the importance values from.

Method \texttt{clone()}: The objects of this class are cloneable with this method.

\textit{Usage:}

\texttt{FilterSelectedFeatures\$clone(deep = FALSE)}

\textit{Arguments:}

depth Whether to make a deep clone.

See Also

Dictionary of Filters: \texttt{mlr_filters}

Other Filter: \texttt{Filter, mlr\_filters\_anova, mlr\_filters\_auc, mlr\_filters\_carscore, mlr\_filters\_cmim,}
\texttt{mlr\_filters\_correlation, mlr\_filters\_dissr, mlr\_filters\_find\_correlation, mlr\_filters\_importance,}
\texttt{mlr\_filters\_information\_gain, mlr\_filters\_jmim, mlr\_filters\_jmi, mlr\_filters\_kruskal\_test,}
\texttt{mlr\_filters\_mim, mlr\_filters\_mrmr, mlr\_filters\_njmim, mlr\_filters\_performance, mlr\_filters\_permutation,}
\texttt{mlr\_filters\_relief, mlr\_filters\_variance, mlr\_filters}

Examples

\begin{verbatim}
  task = mlr3::tsk("iris")
  learner = mlr3::lrn("classif.rpart")
  filter = flt("selected_features", learner = learner)
  filter\$calculate(task)
  as.data.table(filter)
\end{verbatim}
**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:*

depth Whether to make a deep clone.

**References**

For a benchmark of filter methods:


**See Also**

Dictionary of Filters: `mlr_filters`

Other Filter: `Filter, mlr3filters::anova, mlr3filters::auc, mlr3filters::carscore, mlr3filters::cmim, mlr3filters::correlation, mlr3filters::disr, mlr3filters::find_correlation, mlr3filters::importance, mlr3filters::information_gain, mlr3filters::jmim, mlr3filters::jmi, mlr3filters::kruskal_test, mlr3filters::mim, mlr3filters::mmr, mlr3filters::njmim, mlr3filters::performance, mlr3filters::permutation, mlr3filters::relief, mlr3filters::selected_features, mlr3filters`
Examples

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

* Dictionary
  mlr_filters, 6
* Filter
  Filter, 3
  mlr_filters, 6
  mlr_filters_anova, 7
  mlr_filters_auc, 8
  mlr_filters_carscore, 10
  mlr_filters_cmim, 11
  mlr_filters_correlation, 12
  mlr_filters_disr, 13
  mlr_filters_find_correlation, 15
  mlr_filters_importance, 16
  mlr_filters_information_gain, 17
  mlr_filters_jmi, 18
  mlr_filters_jmim, 20
  mlr_filters_kruskal_test, 21
  mlr_filters_mim, 22
  mlr_filters_mrmr, 24
  mlr_filters_njmim, 25
  mlr_filters_performance, 26
  mlr_filters_permutation, 28
  mlr_filters_relief, 29
  mlr_filters_selected_features, 30
  mlr_filters_variance, 32
* datasets
  mlr_filters, 6

care::carscore(), 10
caret::findCorrelation(), 15
character(), 4

dictionary, 3, 6

Filter, 3, 6–15, 17–27, 29–32
FilterAnova (mlr_filters_anova), 7
FilterAUC (mlr_filters_auc), 8
FilterCarScore (mlr_filters_carscore), 10
FilterCMIM (mlr_filters_cmim), 11
FilterCorrelation (mlr_filters_correlation), 12
FilterDISR (mlr_filters_disr), 13
FilterFindCorrelation (mlr_filters_find_correlation), 15
FilterImportance (mlr_filters_importance), 16
FilterInformationGain (mlr_filters_information_gain), 17
FilterJMI (mlr_filters_jmi), 18
FilterJMIM (mlr_filters_jmim), 20
FilterKruskalTest (mlr_filters_kruskal_test), 21
FilterMIM (mlr_filters_mim), 22
FilterMRMR (mlr_filters_mrmr), 24
FilterNJMIM (mlr_filters_njmim), 25
FilterPerformance (mlr_filters_performance), 26
FilterPermutation (mlr_filters_permutation), 28
FilterRelief (mlr_filters_relief), 29
FilterSelectedFeatures (mlr_filters_selected_features), 30
FilterVariance (mlr_filters_variance), 32
flt, 6
flt(), 6
flts(flt), 6
FSelectorRcpp::information_gain(), 17
FSelectorRcpp::relief(), 29
integer(), 5