

# Package ‘mlr3filters’

November 10, 2020

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

**Version** 0.4.0

**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 (>= 0.1.8), mlr3misc, paradox, R6

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

**Config/testthat/edition** 3

**Config/testthat/parallel** true

**Encoding** UTF-8

**NeedsCompilation** no

**RoxygenNote** 7.1.1

**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' 'FilterVariance.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** 2020-11-10 22:40:02 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 . . . . .	9
mlr_filters_cmim . . . . .	10
mlr_filters_correlation . . . . .	12
mlr_filters_disr . . . . .	13
mlr_filters_find_correlation . . . . .	14
mlr_filters_importance . . . . .	15
mlr_filters_information_gain . . . . .	16
mlr_filters_jmi . . . . .	17
mlr_filters_jmim . . . . .	18
mlr_filters_kruskal_test . . . . .	19
mlr_filters_mim . . . . .	20
mlr_filters_mrmr . . . . .	22
mlr_filters_njmim . . . . .	23
mlr_filters_performance . . . . .	24
mlr_filters_permutation . . . . .	25
mlr_filters_relief . . . . .	27
mlr_filters_variance . . . . .	28
<b>Index</b>	<b>30</b>

---

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:** Patrick Schratz <patrick.schratz@gmail.com> ([ORCID](#))

Authors:

- Michel Lang <michellang@gmail.com> ([ORCID](#))
- Bernd Bischl <bernd\_bischl@gmx.net> ([ORCID](#))
- Martin Binder <mlr.developer@mb706.com>

**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.

`task_type` (character(1))

Task type, e.g. "classif" or "regr".

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$new(
  id,
  task_type,
  task_properties = character(),
  param_set = ParamSet$new(),
  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".

`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.

`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 details), and defaults to the number of features in `task`. Loads required packages and then calls `private$.calculate()` of the respective subclass. 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\\_mrmm](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

---

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 <a href="#">dictionary</a> to retrieve the object.
...	(named list()) Named arguments passed to the constructor, to be set as parameters in the <a href="#">paradox::ParamSet</a> , or to be set as public field. See <a href="#">mlr3misc::dictionary_sugar_get()</a> for more details.
.keys	(character()) Keys passed to the respective <a href="#">dictionary</a> to retrieve multiple objects.

### Value

[Filter](#).

### Examples

```
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**

Other Filter: [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](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#)

**Examples**

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

---

mlr_filters_anova	<i>ANOVA F-Test Filter</i>
-------------------	----------------------------

---

**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.

## See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [Filter](#), [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](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

## Examples

```
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	<i>AUC Filter</i>
-----------------	-------------------

---

## 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.

**See Also**

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_variance](#), [mlr\\_filters](#)

**Examples**

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

---

`mlr_filters_carscore`    *Conditional Mutual Information Based Feature Selection Filter*

---

**Description**

Calculates the Correlation-Adjusted (marginal) coRelation 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](#)

Other Filter: [Filter](#), [mlr\\_filters\\_anova](#), [mlr\\_filters\\_auc](#), [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\\_variance](#), [mlr\\_filters](#)

**Examples**

```
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.

## 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.

## See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [Filter](#), [mlr\\_filters\\_anova](#), [mlr\\_filters\\_auc](#), [mlr\\_filters\\_carscore](#), [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\\_variance](#), [mlr\\_filters](#)

## Examples

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

---

```
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.

### See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [Filter](#), [mlr\\_filters\\_anova](#), [mlr\\_filters\\_auc](#), [mlr\\_filters\\_carscore](#), [mlr\\_filters\\_cmim](#), [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\\_variance](#), [mlr\\_filters](#)

### 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)
```

---

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:  $1, (k-1)/k, \dots, 1/k$  where  $k$  is the number of selected features.

## 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.

## See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [Filter](#), [mlr\\_filters\\_anova](#), [mlr\\_filters\\_auc](#), [mlr\\_filters\\_carscore](#), [mlr\\_filters\\_cmim](#), [mlr\\_filters\\_correlation](#), [mlr\\_filters\\_find\\_correlation](#), [mlr\\_filters\\_importance](#), [mlr\\_filters\\_information](#), [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\\_variance](#), [mlr\\_filters](#)

**Examples**

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

---

```
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**

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [Filter](#), [mlr\\_filters\\_anova](#), [mlr\\_filters\\_auc](#), [mlr\\_filters\\_carscore](#), [mlr\\_filters\\_cmim](#), [mlr\\_filters\\_correlation](#), [mlr\\_filters\\_disr](#), [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\\_variance](#), [mlr\\_filters](#)

**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)
```

---

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.

**See Also**

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_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\\_variance](#), [mlr\\_filters](#)

**Examples**

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

---

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

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_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\\_variance](#), [mlr\\_filters](#)

## Examples

```
## InfoGain (default)
task = mlr3::tsk("pima")
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)
```

---

mlr_filters_jmi	<i>Joint Mutual Information Filter</i>
-----------------	----------------------------------------

---

## 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:  $1, (k-1)/k, \dots, 1/k$  where  $k$  is the number of selected features.

## 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.

### See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_kruskal\\_test](#), [mlr\\_filters\\_mim](#), [mlr\\_filters\\_mrmr](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

### Examples

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

---

mlr\_filters\_jmim

*Minimal Joint Mutual Information Maximisation 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:  $1, (k-1)/k, \dots, 1/k$  where  $k$  is the number of selected features.

### 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.

## See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_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\\_variance](#), [mlr\\_filters](#)

## Examples

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

---

`mlr_filters_kruskal_test`

*Kruskal-Wallis Test 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.

## See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_mim](#), [mlr\\_filters\\_mrmr](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [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.

## 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.

## See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_mrmr](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

## Examples

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

---

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.

## 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.

## See Also

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

## Examples

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

---

mlr_filters_njmim	<i>Minimal Normalised Joint Mutual Information Maximisation Filter</i>
-------------------	------------------------------------------------------------------------

---

## 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.

## 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.

**See Also**

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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\\_mrmm](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

**Examples**

```
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](#)

Other Filter: [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](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

**Examples**

```
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 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 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 FilterDISR 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](#)

Other Filter: [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](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_relief](#), [mlr\\_filters\\_variance](#), [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](#)

Other Filter: [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](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_variance](#), [mlr\\_filters](#)

**Examples**

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

**See Also**

Dictionary of Filters: [mlr\\_filters](#)

Other Filter: [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](#), [mlr\\_filters\\_njmim](#), [mlr\\_filters\\_performance](#), [mlr\\_filters\\_permutation](#), [mlr\\_filters\\_relief](#), [mlr\\_filters](#)

**Examples**

```
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, 9
  - mlr\_filters\_cmim, 10
  - mlr\_filters\_correlation, 12
  - mlr\_filters\_disr, 13
  - mlr\_filters\_find\_correlation, 14
  - mlr\_filters\_importance, 15
  - mlr\_filters\_information\_gain, 16
  - mlr\_filters\_jmi, 17
  - mlr\_filters\_jmim, 18
  - mlr\_filters\_kruskal\_test, 19
  - mlr\_filters\_mim, 20
  - mlr\_filters\_mrmr, 22
  - mlr\_filters\_njmim, 23
  - mlr\_filters\_performance, 24
  - mlr\_filters\_permutation, 25
  - mlr\_filters\_relief, 27
  - mlr\_filters\_variance, 28
- \* **datasets**
  - mlr\_filters, 6
- care::carscore(), 9
- caret::findCorrelation(), 14
- character(), 4
- Dictionary, 6–14, 16–22, 24, 25, 27, 28
- dictionary, 3, 6
- Filter, 3, 6–14, 16–25, 27, 28
- FilterAnova (mlr\_filters\_anova), 7
- FilterAUC (mlr\_filters\_auc), 8
- FilterCarScore (mlr\_filters\_carscore), 9
- FilterCMIM (mlr\_filters\_cmim), 10
- FilterCorrelation (mlr\_filters\_correlation), 12
- FilterDISR (mlr\_filters\_disr), 13
- FilterFindCorrelation (mlr\_filters\_find\_correlation), 14
- FilterImportance (mlr\_filters\_importance), 15
- FilterInformationGain (mlr\_filters\_information\_gain), 16
- FilterJMI (mlr\_filters\_jmi), 17
- FilterJMIM (mlr\_filters\_jmim), 18
- FilterKruskalTest (mlr\_filters\_kruskal\_test), 19
- FilterMIM (mlr\_filters\_mim), 20
- FilterMRMR (mlr\_filters\_mrmr), 22
- FilterNJMIM (mlr\_filters\_njmim), 23
- FilterPerformance (mlr\_filters\_performance), 24
- FilterPermutation (mlr\_filters\_permutation), 25
- FilterRelief (mlr\_filters\_relief), 27
- Filters, 8–14, 16–22, 24, 25, 27, 28
- FilterVariance (mlr\_filters\_variance), 28
- flt, 6
- flt(), 6
- flts (flt), 6
- FSelectorRcpp::information\_gain(), 16
- FSelectorRcpp::relief(), 27
- integer(), 5
- mlr3::Learner, 15, 24–26
- mlr3::Measure, 24–26
- mlr3::mlr\_sugar, 6
- mlr3::resample(), 24, 25
- mlr3::Resampling, 24–26
- mlr3::Task, 3–5, 25

- mlr3filters (mlr3filters-package), 2
- mlr3filters-package, 2
- mlr3filters::Filter, 7–9, 11–19, 21–24, 26–28
- mlr3measures::auc(), 8
- mlr3misc::dictionary\_sugar\_get(), 6
- mlr\_filters, 3, 5, 6, 6, 8–14, 16–22, 24, 25, 27, 28
- mlr\_filters\_anova, 5, 7, 7, 9–14, 16–22, 24, 25, 27, 28
- mlr\_filters\_auc, 5, 7, 8, 8, 10–14, 16–22, 24, 25, 27, 28
- mlr\_filters\_carscore, 5, 7–9, 9, 11–14, 16–22, 24, 25, 27, 28
- mlr\_filters\_cmim, 5, 7–10, 10, 12–14, 16–22, 24, 25, 27, 28
- mlr\_filters\_correlation, 5, 7–11, 12, 13, 14, 16–22, 24, 25, 27, 28
- mlr\_filters\_disr, 5, 7–12, 13, 14, 16–22, 24, 25, 27, 28
- mlr\_filters\_find\_correlation, 5, 7–13, 14, 16–22, 24, 25, 27, 28
- mlr\_filters\_importance, 5, 7–14, 15, 17–22, 24, 25, 27, 28
- mlr\_filters\_information\_gain, 5, 7–14, 16, 16, 18–22, 24, 25, 27, 28
- mlr\_filters\_jmi, 5, 7–14, 16, 17, 17, 19–22, 24, 25, 27, 28
- mlr\_filters\_jmim, 5, 7–14, 16–18, 18, 20–22, 24, 25, 27, 28
- mlr\_filters\_kruskal\_test, 5, 7–14, 16–19, 19, 21, 22, 24, 25, 27, 28
- mlr\_filters\_mim, 5, 7–14, 16–20, 20, 22, 24, 25, 27, 28
- mlr\_filters\_mrmr, 5, 7–14, 16–21, 22, 24, 25, 27, 28
- mlr\_filters\_njmim, 5, 7–14, 16–22, 23, 25, 27, 28
- mlr\_filters\_performance, 5, 7–14, 16–22, 24, 24, 27, 28
- mlr\_filters\_permutation, 5, 7–14, 16–22, 24, 25, 25, 27, 28
- mlr\_filters\_relief, 5, 7–14, 16–22, 24, 25, 27, 27, 28
- mlr\_filters\_variance, 5, 7–14, 16–22, 24, 25, 27, 28
- mlr\_reflections\$task\_feature\_types, 4
- mlr\_reflections\$task\_properties, 4
- mlr\_reflections\$task\_types\$type, 3
- paradox::ParamSet, 3, 4, 6
- praznik::CMIM(), 10
- praznik::DISR(), 13
- praznik::JMI(), 17
- praznik::JMIM(), 18
- praznik::MIM(), 20
- praznik::MRMR(), 22
- praznik::NJMIM(), 23
- R6Class, 7
- requireNamespace(), 4
- stats::aov(), 7
- stats::cor(), 12
- stats::kruskal.test(), 19