Package: ExhaustiveSearch (via r-universe)

October 24, 2026

Type Package

Title A Fast and Scalable Exhaustive Feature Selection Framework

Version 1.0.2

Description The goal of this package is to provide an easy to use, fast and scalable exhaustive search framework. Exhaustive feature selections typically require a very large number of models to be fitted and evaluated. Execution speed and memory management are crucial factors here. This package provides solutions for both. Execution speed is optimized by using a multi-threaded C++ backend, and memory issues are solved by by only storing the best results during execution and thus keeping memory usage constant.

License GPL (>= 3)

Encoding UTF-8

Imports Rcpp

LinkingTo Rcpp, RcppArmadillo

RoxygenNote 7.3.3

URL https://github.com/RudolfJagdhuber/ExhaustiveSearch

BugReports https://github.com/RudolfJagdhuber/ExhaustiveSearch/issues

Suggests mlbench

NeedsCompilation yes

Author Rudolf Jagdhuber [aut, cre], Jorge Nocedal [cph] (lbfgs c library), Naoaki Okazaki [cph] (lbfgs c library)

Maintainer Rudolf Jagdhuber <r.jagdhuber@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2025-09-19 08:40:30 UTC

RemoteUrl https://github.com/cran/ExhaustiveSearch

RemoteRef HEAD

RemoteSha c3fcbf4bf8d39c7f7373b50893ce112778825c4b

Contents

ExhaustiveSearch 2
getFeatures 5
print.ExhaustiveSearch 6
resultTable 7

Index **9**

`ExhaustiveSearch` *Exhaustive feature selection*

Description

Performs an exhaustive feature selection. `ExhaustiveSearch()` is a fast and scalable implementation of an exhaustive feature selection framework. It is particularly suited for huge tasks, which would typically not be possible due to memory limitations. The current version allows to compute linear and logistic regression models and compare them with respect to AIC or MSE.

Usage

```
ExhaustiveSearch(
  formula,
  data,
  family = NULL,
  performanceMeasure = NULL,
  combsUpTo = NULL,
  nResults = 5000,
  nThreads = NULL,
  testSetIDs = NULL,
  errorVal = -1,
  quietly = FALSE,
  checkLarge = TRUE
)
```

Arguments

`formula`	An object of class formula (or one that can be coerced to that class): a symbolic description of the feature and response structure. All combinations of features on the right hand side of the formula are evaluated.
`data`	A data.frame (or object coercible by `as.data.frame()` to a data.frame) containing the variables in the model.
`family`	A character string naming the family function similar to the parameter in `glm()`. Currently options are 'gaussian' or 'binomial'. If not specified, the function tries to guess it from the response variable.

`performanceMeasure`
: A character string naming the performance measure to compare models by. Currently available options are 'AIC' (Akaike's An Information Criterion) or 'MSE' (Mean Squared Error).

`combsUpTo`
: An integer of length 1 to set an upper limit to the number of features in a combination. This can be useful to drastically reduce the total number of combinations to a feasible size.

`nResults`
: An integer of length 1 to define the size of the final ranking list. The default (5000) provides a good trade-off of memory usage and result size. Set this value to `Inf` to store all models.

`nThreads`
: Number of threads to use. The default is to detect the available number of threads automatically.

`testSetIDs`
: A vector of row indices of data, which define the test set partition. If this parameter is `NULL` (default), models are trained and evaluated on the full data set. If it is set, models are trained on `data[-testSetIDs,]` and tested on `data[testSetIDs,]`.

`errorVal`
: A numeric value defining what performance result is returned if the model could not be fitted. The default (-1) makes those models appear at the top of the result ranking.

`quietly`
: logical. If set to `TRUE` (default), status and runtime updates are printed to the console.

`checkLarge`
: logical. Very large calls get stopped by a safety net. This parameter can be used to execute these calls anyway.

Details

An exhaustive search evaluates all setups of a combinatorial task. In feature and model selection application, exhaustive searches are often referred to as *optimal* search strategies, as they test each setup and therefore ensure to find the best solution. The main downside of this approach is the possibly enormous computational complexity of the task. `ExhaustiveSearch()` provides an easy to use and efficient framework for these tasks.

Its main characteristics are:

- Combinations are iteratively generated on the fly,
- Model fitting and evalution is performed multi-threaded in C++,
- Only a fixed amount of models are stored to keep memory usage small.

Therefore, the framework of this package is able to evaluate huge tasks of billions of models, while only being limited by run-time.

Currently, ordinary linear regression models similar to `lm()` and logistic regression models similar to `glm()` (with parameter `family = "binomial"`) can be fitted. The model type is specified via the `family` parameter. All model results of the C++ backend are identical to what would be obtained by `glm()` or `lm()`. For that, the logistic regression also uses the same L-BFGS optimizer as `glm()`.

To assess the quality of a model, the `performanceMeasure` options 'AIC' (Akaike's An Information Criterion) and 'MSE' (Mean Squared Error) are implemented. Note that the AIC can only be computed on the training data, while it is recommended for the MSE to be computed on independent

test data. If performanceMeasure is not set, it will be decided according to the definition of a test data set.

While this framework is able to handle very large amounts of combinations, an exhaustive search of every theoretical combination can still be unfeasible. However, a possible way to drastically limit the total number of combinations is to define an upper bound for the size of a combination. For example, evaluating all combinations of 500 features (3.3e150) is obviously impossible. But if we only consider combinations of up to 3 features, this number reduces to around 21 million, which could easily be evaluated by this framework in less than a minute (16 threads). Setting an upper limit is thus a very powerful option to enable high dimensional analyses. It is implemented by the parameter combsUpTo.

A core element of why this framework does not require more memory if tasks get larger is that at any point the best models are stored in a list of fixed size. Therefore, sub-optimal models are not saved and do not take space and time to be handled. The parameter defining the size of the models, which are actively stored is nResults. Large values here can impair performance or even cause errors, if the system memory runs out and should always be set with care. The function will however warn you beforehand if you set a very large value here.

The parameter testSetIDs can be used to split the data into a training and testing partition. If it is not set, all models will be trained and tested on the full data set. If it is set, the data will be split beforehand into data[testSetIDs,] and data[-testSetIDs,].

The development version of this package can be found at https://github.com/RudolfJagdhuber/ExhaustiveSearch. Issues or requests are handled on this page.

Value

Object of class ExhaustiveSearch with elements

nModels	The total number of evaluated models.
runtimeSec	The total runtime of the exhaustive search in seconds.
ranking	A list of the performance values and the featureIDs. The i-th element of both correspond. The featureIDs refer to the elements of featureNames. Formatted results of these rankings can e.g. be obtained with getFeatures(), or resultTable().
featureNames	The feature names in the given data. featureIDs in the ranking element refer to this vector.
batchInfo	A list of information on the batches, into which the total task has been partitioned. List elements are the number of batches, the number of elements per batch, and the combination boundaries that define the batches.
setup	A list of input parameters from the function call.

Author(s)

Rudolf Jagdhuber

See Also

resultTable(), getFeatures()

Examples

```
## Linear Regression on mtcars data
data(mtcars)

##  Exhaustive search of 1023 models compared by AIC
ES <- ExhaustiveSearch(mpg ~ ., data = mtcars, family = "gaussian",
  performanceMeasure = "AIC")
print(ES)

## Same setup, but compared by MSE on a test set partition
testIDs <- sample(nrow(mtcars), round(1/3 * nrow(mtcars)))
ES2 <- ExhaustiveSearch(mpg ~ ., data = mtcars, family = "gaussian",
  performanceMeasure = "MSE", testSetIDs = testIDs)
print(ES2)


## Not run:
## Logistic Regression on Ionosphere Data
data("Ionosphere", package = "mlbench")

## Only combinations of up to 3 features! -> 5488 models instead of 4 billion
ES3 <- ExhaustiveSearch((Class == "good") ~ ., data = Ionosphere[,-c(1, 2)],
  family = "binomial", combsUpTo = 3)
print(ES3)

## End(Not run)
```

`getFeatures` *Extract the feature sets from an ExhaustiveSearch object*

Description

A simple function to get a vector of feature names for one or more elements of an ExhaustiveSearch object.

Usage

```
getFeatures(ESResult, ranks)
```

Arguments

`ESResult`	a result object from an exhaustive search.
`ranks`	a numeric value or vector defining which elements should be returned.

Value

If `ranks` is a single value, a vector of feature names is returned. If an intercept is included, the first element of this vector is "1". If `ranks` includes multiple values, a list of such vectors is returned.

Author(s)

Rudolf Jagdhuber

See Also

ExhaustiveSearch()

Examples

```
## Exhaustive search on the mtcars data
data(mtcars)
ES <- ExhaustiveSearch(mpg ~ ., data = mtcars, family = "gaussian")

## Get the feature combinations of the top 3 models
getFeatures(ES, 1:3)

## Get the feature combination of the 531th best model
getFeatures(ES, 531)
```

---

print.ExhaustiveSearch *Print ExhaustiveSearch*

Description

Prints a compact summary of the results of an ExhaustiveSearch object.

Usage

```
## S3 method for class 'ExhaustiveSearch'
print(x, ...)
```

Arguments

x	Object of class 'ExhaustiveSearch'.
...	Further arguments passed to or from other methods.

Value

No return value. The function is only called to print results to the console.

Author(s)

Rudolf Jagdhuber

See Also

ExhaustiveSearch()

`resultTable` *Overview of the top exhaustive search results*

Description

Extract the top `n` results of an exhaustive search and present them as a `data.frame` object.

Usage

```
resultTable(ESResult, n = Inf, insertStart = "")
```

Arguments

`ESResult`	a result object from an exhaustive search.
`n`	number of results to be returned. The default (`Inf`) returns every result available in `ESResult`.
`insertStart`	used for additional spacing when printing. The value of `insertStart` gets printed in front of every feature combination to increase the space to the printed performance measure.

Details

The result of an exhaustive search is given by an object of class ExhaustiveSearch, which is a list of encoded feature combinations and performance values. This function decodes the feature combinations and presents them in a `data.frame` together with the respective performance values

Value

A `data.frame` with two columns. The first one shows the performance values and the second shows the decoded feature set collapsed with plus signs.

Author(s)

Rudolf Jagdhuber

See Also

`ExhaustiveSearch()`

Examples

```
## Exhaustive search on the mtcars data
data(mtcars)
ES <- ExhaustiveSearch(mpg ~ ., data = mtcars, family = "gaussian")

## Summary data.frame of the top 5 models
resultTable(ES, 5)
```

```
## Return a data.frame of all stored models
res <- resultTable(ES)
str(res)

## Add custom characters for printing
resultTable(ES, 1, "  <->  ")
```

Index

as.data.frame(), *2*

character, *2, 3*

data.frame, *2*

ExhaustiveSearch, 2
ExhaustiveSearch(), *6, 7*

formula, *2*

getFeatures, 5
getFeatures(), *4*
glm(), *2, 3*

lm(), *3*
logical, *3*

print.ExhaustiveSearch, 6

resultTable, 7
resultTable(), *4*