Package: opdisDownsampling (via r-universe)

October 4, 2026

Type Package

Title Optimal Distribution Preserving Down-Sampling of Bio-Medical Data

Version 1.7

Description An optimized method for distribution-preserving class-proportional down-sampling of bio-medical data <doi:10.1371/journal.pone.0255838>.

Depends R (>= 3.5.0)

Imports parallel, graphics, methods, stats, caTools, pracma, twosamples, doParallel, pbmcapply, foreach, utils, Rcpp (>= 1.0.0)

LazyData true

LinkingTo Rcpp

License GPL-3

URL https://github.com/JornLotsch/opdisDownsampling

Encoding UTF-8

Maintainer Jorn Lotsch <j.lotsch@em.uni-frankfurt.de>

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

Date/Publication 2026-09-04 13:55:25 UTC

RemoteUrl https://github.com/jornlotsch/opdisdownsampling

RemoteRef HEAD

RemoteSha 4ea25815c592a54de164dfa7596500804efb55e5

Contents

FlowcytometricData 2
get_seed 2
GMMartificialData 4
opdisDownsampling 5

Index **8**

FlowcytometricData *Example data of hematologic marker expression.*

Description

Data set of 6 flow cytometry-based lymphoma makers from 55,843 cells from healthy subjects (class 1) and 55,843 cells from lymphoma patients (class 2).

Usage

```
data("FlowcytometricData")
```

Details

Size 111686 x 6 , stored in `FlowcytometricData$[Var_1,Var_2,Var_3,Var_4,Var_5,Var_6]`
Classes 2, stored in `FlowcytometricData$Cls`

Examples

```
data(FlowcytometricData)
str(FlowcytometricData)
```

get_seed *Random Seed Recovery from RNG State*

Description

Functions for recovering the original seed value that produced the current random number generator state. Provides both R and C++ implementations with the C++ version offering significantly improved performance for large search spaces.

Usage

```
get_seed(range = NULL, fallback_seed = 42, max_search = 2147483647,
  step_size = 50000, use_cpp = TRUE, ...)

get_seed_cpp(range = NULL, fallback_seed = 42, max_search = 2147483647,
  step_size = 50000, batch_size = 10000, verbose = TRUE)
```

Arguments

range	Optional integer vector of specific seed values to search. If provided, only these seeds will be tested instead of systematic range searching.
fallback_seed	Integer seed value to return if no matching seed is found during the search process (default: 42).
max_search	Maximum seed value to search up to when performing systematic range searching. Must be a positive integer within the valid range for R's random number generator (default: 2147483647).
step_size	Step size for systematic range searching when no specific range is provided. Larger values speed up search but may miss the target seed if it falls between steps (default: 50000).
use_cpp	Logical; if TRUE, uses the fast C++ implementation via get_seed_cpp(). If FALSE, falls back to the slower R implementation with a warning (default: TRUE).
batch_size	Integer specifying the number of seeds to process in each C++ batch operation. Larger batches are more memory efficient but require more RAM. Only used in get_seed_cpp() (default: 10000).
verbose	Logical; if TRUE, prints progress information during the search process including batch progress and timing information. Only used in get_seed_cpp() (default: TRUE).
...	Additional arguments passed to get_seed_cpp() when use_cpp = TRUE.

Details

The functions work by systematically testing seed values to find one that reproduces the current RNG state stored in .Random.seed. The search process:

- Tests each candidate seed by setting it and comparing the resulting RNG state
- Uses efficient C++ implementation for faster processing of large search spaces
- Supports both targeted searching (via range parameter) and systematic range searching
- Employs batched processing to optimize memory usage and performance

Performance Considerations:

The C++ implementation (get_seed_cpp()) provides significant performance improvements:

- Batch processing reduces overhead for large search spaces
- Optimized memory management prevents excessive RAM usage
- Native C++ random number generation matching R's implementation
- Progress reporting for long-running searches

Search Strategy:

- If range is provided: Tests only the specified seed values
- If range is NULL: Performs systematic search from 1 to max_search in steps of step_size
- Search terminates immediately when a matching seed is found

- Returns `fallback_seed` if no match is found within the search parameters

Memory Management:

The C++ implementation uses batched processing controlled by `batch_size` to:

- Process large search ranges without excessive memory allocation
- Provide regular progress updates during long searches
- Allow interruption of long-running operations

Value

Returns an integer representing the seed value that reproduces the current random number generator state. If no matching seed is found within the search parameters, returns the `fallback_seed` value.

Note

- Requires an active RNG state (i.e., `.Random.seed` must exist)
- Large search ranges may take considerable time even with C++ optimization
- The search is deterministic but computationally intensive
- Consider using smaller `step_size` values if the initial search fails

See Also

`set.seed`, `.Random.seed`

Examples

```
## Basic seed recovery after generating random numbers
set.seed(123)
recovered_seed <- get_seed()
print(recovered_seed)
```

GMMartificialData *Example data an artificial Gaussian mixture.*

Description

Dataset of 30000 instances with 10 variables that are Gaussian mixtures and belong to classes Cls = 1, 2, or 3, with different means and standard deviations and equal weights of 0.5, 0.4, and 0.1, respectively.

Usage

```
data("GMMartificialData")
```

Details

Size 30000 x 10, stored in `GMMartificialData$[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10]`

Classes 3, stored in `GMMartificialData$Cls`

Examples

```
data(GMMartificialData)
str(GMMartificialData)
```

---

opdisDownsampling *Optimal Distribution Preserving Down-Sampling of Bio-Medical Data*

Description

The package provides functions for optimal distribution-preserving down-sampling of large (bio-medical) data sets. It draws statistically representative subsets of data while preserving the class proportions and original data distribution.

Usage

```
opdisDownsampling(Data, Cls, Size, Seed = "simple", nTrials = 1000,
  TestStat = "ad", MaxCores = getOption("mc.cores", 2L),
  PCAimportance = FALSE, CheckRemoved = FALSE, JobSize = 0, verbose = FALSE)
```

Arguments

`Data`	Numeric data as a vector, matrix, or data frame. Each row represents an instance, each column a variable.
`Cls`	Optional vector with class labels for each instance in `Data`. If missing, all instances are treated as belonging to the same class.
`Size`	The number (integer) or proportion (0<Size<1) of instances to draw from the dataset. The reduction is class proportional and aims to preserve the variable distributions.
`Seed`	Seed value. Options: `"auto"` for seed recovery, `"simple"` to generate and report a seed using the current RNG state, or an integer for exact reproducibility. Use integers for systematic testing and fully reproducible analyses.
`nTrials`	Number of random sampling trials used to find the optimal subset (default: 1000).
`TestStat`	Character string defining the statistical test used to assess distribution similarity. Available options are: • `"ad"`: Anderson–Darling statistic • `"kuiper"`: Kuiper statistic • `"cvm"`: Cramér–von Mises statistic • `"wass"`: Wasserstein distance

- "dts": Distributional Transform Statistic
- "ks": Kolmogorov–Smirnov statistic
- "kld": Kullback–Leibler divergence (via KullbLeiblKLD2())
- "amrdd": Average Mean Root of Distributional Differences (via amrdd())
- "euc": Euclidean distance (via EucDist())
- "nent": Absolute normalized entropy difference (via abs_norm_entropy_diff())

MaxCores — Maximum number of CPU cores to use for parallel computing (default is value stored in getOption("mc.cores"), or 2 if missing).

PCAimportance — Logical; if TRUE, only variables deemed important by principal component analysis are used in computing similarity statistics.

CheckRemoved — Logical; if TRUE, the optimization also considers distribution similarity between the removed data and the original, selecting the trial that minimizes distributional differences for both the retained and the removed subset (default: FALSE).

JobSize — Integer specifying the number of trials to process in each chunk. If 0 (default), no chunking is applied. If NULL, an automatic chunk size is calculated based on data dimensions, number of trials, available system memory, and number of processor cores. A positive integer manually sets the chunk size.

verbose — Logical; if TRUE, prints diagnostic information about chunk-size selection, including data dimensions, estimated memory usage, and the chosen chunking strategy. Useful for understanding memory usage patterns and debugging performance issues.

Details

Chunked processing can be used to reduce memory usage when dealing with large datasets or high numbers of trials. Set JobSize = NULL to enable automatic memory-aware chunk-size calculation. The automatic chunking strategy considers:

- Data size, defined by number of rows and columns
- Available system memory, detected on Linux systems
- Number of processor cores
- Number of trials to perform

Set JobSize = 0 to process all trials in a single batch. Set JobSize to a positive integer to manually define the number of trials processed per chunk.

Variable Selection Method:

If PCAimportance = TRUE, PCA-based variable selection is used. Variables are ranked by their loadings in the first principal components, and variables with higher importance scores are used for distribution comparisons.

Value

Returns a list with the following elements:

ReducedData — Down-sampled data set (as data frame or matrix) including only the selected instances.

`RemovedData` Data not included in the sample.

`ReducedInstances`
: Row indices (or names) of the selected instances from the original data set.

`RemovedInstances`
: Row indices (or names) of the unselected instances from the original data set.

Author(s)

Jorn Lotsch

References

Lotsch, J., Malkusch, S., Ultsch, A. (2021):\ Optimal distribution-preserving downsampling of large biomedical data sets.\ *PLoS ONE* 16(8): e0255838. doi:10.1371/journal.pone.0255838

Examples

```
## Example: Down-sample the Iris dataset to 50 points
data(iris)
Iris50percent <- opdisDownsampling(Data = iris[,1:4], Cls = as.integer(iris$Species),
  Size = 50, Seed = 42, MaxCores = 1)

## Example: Down-sample with custom chunk size and verbose output
data(iris)
Iris50percent <- opdisDownsampling(Data = iris[,1:4], Cls = as.integer(iris$Species),
  Size = 50, Seed = 42, MaxCores = 1, JobSize = 25, verbose = TRUE)

## Example: Use PCA-based variable selection
data(iris)
Iris_pca <- opdisDownsampling(Data = iris[,1:4], Cls = as.integer(iris$Species),
  Size = 50, Seed = 42, PCAimportance = TRUE, MaxCores = 1)

## Example: Memory-efficient processing of large dataset with many trials
## Not run:
# For large datasets, automatic chunking can reduce memory usage
LargeDataSample <- opdisDownsampling(Data = large_dataset,
  Size = 0.1, Seed = 42, nTrials = 5000, JobSize = NULL, verbose = TRUE)

## End(Not run)
```

Index

∗ **RNG**
    get_seed, 2
∗ **random**
    get_seed, 2
∗ **reproducibility**
    get_seed, 2
∗ **seed**
    get_seed, 2
∗ **state recovery**
    get_seed, 2
.Random.seed, *4*

FlowcytometricData, 2

get_seed, 2
get_seed_cpp *(*get_seed*)*, 2
GMMartificialData, 4

opdisDownsampling, 5

set.seed, *4*