Package: isotree (via r-universe)

October 14, 2026

Type Package

Title Isolation-Based Outlier Detection

Version 0.6.1-5

Maintainer David Cortes <david.cortes.rivera@gmail.com>

URL https://github.com/david-cortes/isotree

BugReports https://github.com/david-cortes/isotree/issues

Description Fast and multi-threaded implementation of isolation forest (Liu, Ting, Zhou (2008) <doi:10.1109/ICDM.2008.17>), extended isolation forest (Hariri, Kind, Brunner (2018) <doi:10.48550/arXiv.1811.02141>), SCiForest (Liu, Ting, Zhou (2010) <doi:10.1007/978-3-642-15883-4_18>), fair-cut forest (Cortes (2021) <doi:10.48550/arXiv.2110.13402>), robust random-cut forest (Guha, Mishra, Roy, Schrijvers (2016) <http://proceedings.mlr.press/v48/guha16.html>), and customizable variations of them, for isolation-based outlier detection, clustered outlier detection, distance or similarity approximation (Cortes (2019) <doi:10.48550/arXiv.1910.12362>), isolation kernel calculation (Ting, Zhu, Zhou (2018) <doi:10.1145/3219819.3219990>), and imputation of missing values (Cortes (2019) <doi:10.48550/arXiv.1911.06646>), based on random or guided decision tree splitting, and providing different metrics for scoring anomalies based on isolation depth or density (Cortes (2021) <doi:10.48550/arXiv.2111.11639>). Provides simple heuristics for fitting the model to categorical columns and handling missing data, and offers options for varying between random and guided splits, and for using different splitting criteria.

License BSD_2_clause + file LICENSE

Depends R (>= 4.3.0)

Imports Rcpp (>= 1.0.1), jsonlite (>= 1.7.3), RhpcBLASctl, methods

Suggests MASS, outliertree, DiagrammeR, mlbench, MLmetrics, kernlab, knitr, rmarkdown, kableExtra

Enhances Matrix, SparseM

LinkingTo Rcpp

VignetteBuilder knitr

RoxygenNote 7.3.2

NeedsCompilation yes

Author David Cortes [aut, cre, cph], Thibaut Goetghebuer-Planchon [cph] (Copyright holder of included robinmap library), David Blackman [cph] (Copyright holder of original xoshiro code), Sebastiano Vigna [cph] (Copyright holder of original xoshiro code), NumPy Developers [cph] (Copyright holder of formatted ziggurat tables), SciPy Developers [cph] (Copyright holder of parts of digamma implementation), Enthought Inc [cph] (Copyright holder of parts of digamma implementation), Stephen Moshier [cph] (Copyright holder of parts of digamma implementation)

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

Date/Publication 2026-03-05 06:10:44 UTC

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

RemoteRef HEAD

RemoteSha cc0551a3aa83652e06ae80a4bf441d530b16691c

Contents

isolation.forest . . . 3
isotree.add.tree . . . 25
isotree.append.trees . . . 27
isotree.build.indexer . . . 28
isotree.deep.copy . . . 29
isotree.drop.imputer . . . 30
isotree.drop.indexer . . . 31
isotree.drop.reference.points . . . 32
isotree.export.model . . . 32
isotree.get.num.nodes . . . 34
isotree.import.model . . . 35
isotree.is.same . . . 36
isotree.plot.tree . . . 37
isotree.restore.handle . . . 38
isotree.set.nthreads . . . 39
isotree.set.reference.points . . . 40
isotree.subset.trees . . . 41
isotree.to.graphviz . . . 42
isotree.to.json . . . 44
isotree.to.sql . . . 45
length.isolation_forest . . . 48
predict.isolation_forest . . . 48

print.isolation_forest 53
summary.isolation_forest 54
variable.names.isolation_forest 55

Index **56**

`isolation.forest` *Create Isolation Forest Model*

Description

Isolation Forest is an algorithm originally developed for outlier detection that consists in splitting sub-samples of the data according to some attribute/feature/column at random. The idea is that, the rarer the observation, the more likely it is that a random uniform split on some feature would put outliers alone in one branch, and the fewer splits it will take to isolate an outlier observation like this. The concept is extended to splitting hyperplanes in the extended model (i.e. splitting by more than one column at a time), and to guided (not entirely random) splits in the SCiForest and FCF models that aim at isolating outliers faster and/or finding clustered outliers.

This version adds heuristics to handle missing data and categorical variables. Can be used to approximate pairwise distances by checking the depth after which two observations become separated, and to approximate densities by fitting trees beyond balanced-tree limit. Offers options to vary between randomized and deterministic splits too.

Important: The default parameters in this software do not correspond to the suggested parameters in any of the references (see section "Matching models from references"). In particular, the following default values are likely to cause huge differences when compared to the defaults in other software: ‘ndim‘, ‘sample_size‘, ‘ntrees‘. The defaults here are nevertheless more likely to result in better models. In order to mimic the Python library "scikit-learn" for example, one would need to pass ‘ndim=1‘, ‘sample_size=256‘, ‘ntrees=100‘, ‘missing_action="fail"‘, ‘nthreads=1‘.

Note that the default parameters will not scale to large datasets. In particular, if the amount of data is large, it’s suggested to set a smaller sample size for each tree (parameter ‘sample_size‘), and to fit fewer of them (parameter ‘ntrees‘). As well, the default option for ‘missing_action‘ might slow things down significantly (see below for details). These defaults can also result in very big model sizes in memory and as serialized files (e.g. models that weight over 10GB) when the number of rows in the data is large. Using fewer trees, smaller sample sizes, and shallower trees can help to reduce model sizes if that becomes a problem.

The model offers many tunable parameters (see reference [11] for a comparison). The most likely candidate to tune is ‘prob_pick_pooled_gain‘, for which higher values tend to result in a better ability to flag outliers in multimodal datasets, at the expense of poorer generalizability to inputs with values outside the variables’ ranges to which the model was fit (see plots generated from the examples for a better idea of the difference). The next candidate to tune is ‘sample_size‘ - the default is to use all rows, but in some datasets introducing sub-sampling can help, especially for the single-variable model. In smaller datasets, one might also want to experiment with ‘weigh_by_kurtosis‘ and perhaps lower ‘ndim‘.If using ‘prob_pick_pooled_gain‘, models are likely to benefit from deeper trees (controlled by ‘max_depth‘), but using large samples and/or deeper trees can result in significantly slower model fitting and predictions - in such cases, using ‘min_gain‘ (with a value like 0.25) with ‘max_depth=NULL‘ can offer a better speed/performance trade-off than changing ‘max_depth‘.

If the data has categorical variables and these are more important important for determining outlierness compared to numerical columns, one might want to experiment with ‘ndim=1‘, ‘categ_split_type="single_categ"‘, and ‘scoring_metric="density"‘; while for all-numeric datasets - especially if there are missing values - one might want to experiment with ‘ndim=2‘ or ‘ndim=3‘.

For small datasets, one might also want to experiment with ‘ndim=1‘, ‘scoring_metric="adj_depth"‘ and ‘penalize_range=TRUE‘.

Usage

```
isolation.forest(
  data,
  sample_size = min(nrow(data), 10000L),
  ntrees = 500,
  ndim = 1,
  ntry = 1,
  categ_cols = NULL,
  max_depth = ceiling(log2(sample_size)),
  ncols_per_tree = ncol(data),
  prob_pick_pooled_gain = 0,
  prob_pick_avg_gain = 0,
  prob_pick_full_gain = 0,
  prob_pick_dens = 0,
  prob_pick_col_by_range = 0,
  prob_pick_col_by_var = 0,
  prob_pick_col_by_kurt = 0,
  min_gain = 0,
  missing_action = ifelse(ndim > 1, "impute", "divide"),
  new_categ_action = ifelse(ndim > 1, "impute", "weighted"),
  categ_split_type = ifelse(ndim > 1, "subset", "single_categ"),
  all_perm = FALSE,
  coef_by_prop = FALSE,
  recode_categ = FALSE,
  weights_as_sample_prob = TRUE,
  sample_with_replacement = FALSE,
  penalize_range = FALSE,
  standardize_data = TRUE,
  scoring_metric = "depth",
  fast_bratio = TRUE,
  weigh_by_kurtosis = FALSE,
  coefs = "uniform",
  assume_full_distr = TRUE,
  build_imputer = FALSE,
  output_imputations = FALSE,
  min_imp_obs = 3,
  depth_imp = "higher",
  weigh_imp_rows = "inverse",
  output_score = FALSE,
  output_dist = FALSE,
```

```
  square_dist = FALSE,
  sample_weights = NULL,
  column_weights = NULL,
  lazy_serialization = TRUE,
  seed = 1,
  use_long_double = FALSE,
  nthreads = parallel::detectCores()
)
```

Arguments

data	Data to which to fit the model. Supported inputs type are: • A ‘data.frame‘, also accepted as ‘data.table‘ or ‘tibble‘. • A ‘matrix‘ object from base R. • A sparse matrix in CSC format, either from package ‘Matrix‘ (class ‘dgCMatrix‘) or from package ‘SparseM‘ (class ‘matrix.csc‘). If passing a ‘data.frame‘, will assume that columns are: • Numerical, if they are of types ‘numeric‘, ‘integer‘, ‘Date‘, ‘POSIXct‘. • Categorical, if they are of type ‘character‘, ‘factor‘, ‘bool‘. Note that, if factors are ordered, the order will be ignored here. Other input and column types are not supported.
sample_size	Sample size of the data sub-samples with which each binary tree will be built. Recommended value in references [1], [2], [3], [4] is 256, while the default value in the author’s code in reference [5] is ‘nrow(data)‘. If passing ‘NULL‘, will take the full number of rows in the data (no sub-sampling). If passing a number between zero and one, will assume it means taking a sample size that represents that proportion of the rows in the data. Note that sub-sampling is incompatible with ‘output_score‘, ‘output_dist‘, and ‘output_imputations‘, and if any of those options is requested, ‘sample_size‘ will be overridden. Hint: seeing a distribution of scores which is on average too far below 0.5 could mean that the model needs more trees and/or bigger samples to reach convergence (unless using non-random splits, in which case the distribution is likely to be centered around a much lower number), or that the distributions in the data are too skewed for random uniform splits.
ntrees	Number of binary trees to build for the model. Recommended value in reference [1] is 100, while the default value in the author’s code in reference [5] is 10. In general, the number of trees required for good results is higher when (a) there are many columns, (b) there are categorical variables, (c) categorical variables have many categories, (d) ‘ndim‘ is high, (e) ‘prob_pick_pooled_gain‘ is used, (f) ‘scoring_metric="density"‘ or ‘scoring_metric="boxed_density"‘ are used. Hint: seeing a distribution of scores which is on average too far below 0.5 could mean that the model needs more trees and/or bigger samples to reach convergence (unless using non-random splits, in which case the distribution is likely to be centered around a much lower number), or that the distributions in the data are too skewed for random uniform splits.

`ndim` Number of columns to combine to produce a split. If passing 1, will produce the single-variable model described in references [1] and [2], while if passing values greater than 1, will produce the extended model described in references [3] and [4]. Recommended value in reference [4] is 2, while [3] recommends a low value such as 2 or 3. Models with values higher than 1 are referred hereafter as the extended model (as in reference [3]).

If passing 'NULL', will assume it means using the full number of columns in the data.

Note that, when using 'ndim>1' plus 'standardize_data=TRUE', the variables are standardized at each step as suggested in [4], which makes the models slightly different than in [3].

In general, when the data has categorical variables, models with 'ndim=1' plus 'categ_split_type="single_categ"' tend to produce better results, while models 'ndim>1' tend to produce better results for numerical-only data, especially in the presence of missing values.

`ntry` When using any of 'prob_pick_pooled_gain', 'prob_pick_avg_gain', 'prob_pick_full_gain', 'prob_pick_dens', how many variables (with 'ndim=1') or linear combinations (with 'ndim>1') to try for determining the best one according to gain.

Recommended value in reference [4] is 10 (with 'prob_pick_avg_gain', for outlier detection), while the recommended value in reference [11] is 1 (with 'prob_pick_pooled_gain', for outlier detection), and the recommended value in reference [9] is 10 to 20 (with 'prob_pick_pooled_gain', for missing value imputations).

`categ_cols` Columns that hold categorical features, when the data is passed as a matrix (either dense or sparse). Can be passed as an integer vector (numeration starting at 1) denoting the indices of the columns that are categorical, or as a character vector denoting the names of the columns that are categorical, assuming that 'data' has column names.

Categorical columns should contain only integer values with a continuous numeration starting at **zero** (not at one as is typical in R packages), and with negative values and NA/NaN taken as missing. The maximum categorical value should not exceed '.Machine$integer.max' (typically $2^{31} - 1$).

This is ignored when the input is passed as a 'data.frame' as then it will consider columns as categorical depending on their type/class (see the documentation for 'data' for details).

`max_depth` Maximum depth of the binary trees to grow. By default, will limit it to the corresponding depth of a balanced binary tree with number of terminal nodes corresponding to the sub-sample size (the reason being that, if trying to detect outliers, an outlier will only be so if it turns out to be isolated with shorter average depth than usual, which corresponds to a balanced tree depth). When a terminal node has more than 1 observation, the remaining isolation depth for them is estimated assuming the data and splits are both uniformly random (separation depth follows a similar process with expected value calculated as in reference [6]). Default setting for references [1], [2], [3], [4] is the same as the default here, but it's recommended to pass higher values if using the model for purposes other than outlier detection.

If passing 'NULL' or zero, will not limit the depth of the trees (that is, will grow them until each observation is isolated or until no further split is possible).

Note that models that use 'prob_pick_pooled_gain' or 'prob_pick_avg_gain' are likely to benefit from deeper trees (larger 'max_depth'), but deeper trees can result in much slower model fitting and predictions.

If using pooled gain, one might want to substitute 'max_depth' with 'min_gain'.

`ncols_per_tree` Number of columns to use (have as potential candidates for splitting at each iteration) in each tree, somewhat similar to the 'mtry' parameter of random forests. In general, this is only relevant when using non-random splits and/or weighted column choices.

If passing a number between zero and one, will assume it means taking a sample size that represents that proportion of the columns in the data. Note that, if passing exactly 1, will assume it means taking 100% of the columns, rather than taking a single column.

If passing 'NULL', will use the full number of columns in the data.

`prob_pick_pooled_gain`

his parameter indicates the probability of choosing the threshold on which to split a variable (with 'ndim=1') or a linear combination of variables (when using 'ndim>1') as the threshold that maximizes a pooled standard deviation gain criterion (see references [9] and [11]) on the same variable or linear combination, similarly to regression trees such as CART.

If using 'ntry>1', will try several variables or linear combinations thereof and choose the one in which the largest standardized gain can be achieved.

For categorical variables with 'ndim=1', will use shannon entropy instead (like in [7]).

Compared to a simple averaged gain, this tends to result in more evenly-divided splits and more clustered groups when they are smaller. Recommended to pass higher values when used for imputation of missing values. When used for outlier detection, datasets with multimodal distributions usually see better performance under this type of splits.

Note that, since this makes the trees more even and thus it takes more steps to produce isolated nodes, the resulting object will be heavier. When splits are not made according to any of 'prob_pick_avg_gain', 'prob_pick_pooled_gain', 'prob_pick_full_gain', 'prob_pick_dens', both the column and the split point are decided at random. Note that, if passing value 1 (100%) with no sub-sampling and using the single-variable model, every single tree will have the exact same splits.

Be aware that 'penalize_range' can also have a large impact when using 'prob_pick_pooled_gain'.

Under this option, models are likely to produce better results when increasing 'max_depth'. Alternatively, one can also control the depth through 'min_gain' (for which one might want to set 'max_depth=NULL').

Important detail: if using any of 'prob_pick_avg_gain' or 'prob_pick_pooled_gain', 'prob_pick_full_gain', 'prob_pick_dens', the distribution of outlier scores is unlikely to be centered around 0.5.

`prob_pick_avg_gain`

This parameter indicates the probability of choosing the threshold on which to split a variable (with 'ndim=1') or a linear combination of variables (when using

'ndim>1') as the threshold that maximizes an averaged standard deviation gain criterion (see references [4] and [11]) on the same variable or linear combination.

If using 'ntry>1', will try several variables or linear combinations thereof and choose the one in which the largest standardized gain can be achieved.

For categorical variables with 'ndim=1', will take the expected standard deviation that would be gotten if the column were converted to numerical by assigning to each category a random number '~ Unif(0, 1)' and calculate gain with those assumed standard deviations.

Compared to a pooled gain, this tends to result in more cases in which a single observation or very few of them are put into one branch. Typically, datasets with outliers defined by extreme values in some column more or less independently of the rest, usually see better performance under this type of split. Recommended to use sub-samples (parameter 'sample_size') when passing this parameter. Note that, since this will create isolated nodes faster, the resulting object will be lighter (use less memory).

When splits are not made according to any of 'prob_pick_avg_gain', 'prob_pick_pooled_gain', 'prob_pick_full_gain', 'prob_pick_dens', both the column and the split point are decided at random. Default setting for [1], [2], [3] is zero, and default for [4] is 1. This is the randomization parameter that can be passed to the author's original code in [5], but note that the code in [5] suffers from a mathematical error in the calculation of running standard deviations, so the results from it might not match with this library's.

Be aware that, if passing a value of 1 (100%) with no sub-sampling and using the single-variable model, every single tree will have the exact same splits.

Under this option, models are likely to produce better results when increasing 'max_depth'.

Important detail: if using either 'prob_pick_avg_gain', 'prob_pick_pooled_gain', 'prob_pick_full_gain', 'prob_pick_dens', the distribution of outlier scores is unlikely to be centered around 0.5.

`prob_pick_full_gain`

This parameter indicates the probability of choosing the threshold on which to split a variable (with 'ndim=1') or a linear combination of variables (when using 'ndim>1') as the threshold that minimizes the pooled sums of variances of all columns (or a subset of them if using 'ncols_per_tree').

In general, this is much slower to evaluate than the other gain types, and does not tend to lead to better results. When using this option, one might want to use a different scoring metric (particulatly '"density"', '"boxed_density2"' or '"boxed_ratio"'). Note that the calculations are all done through the (exact) sorted-indices approach, while is much slower than the (approximate) histogram approach used by other decision tree software.

Be aware that the data is not standardized in any way for the variance calculations, thus the scales of features will make a large difference under this option, which might not make it suitable for all types of data.

his option is not compatible with categorical data, and 'min_gain' does not apply to it.

When splits are not made according to any of ‘prob_pick_avg_gain‘, ‘prob_pick_pooled_gain‘, ‘prob_pick_full_gain‘, ‘prob_pick_dens‘, both the column and the split point are decided at random. Default setting for references [1], [2], [3], [4] is zero.

`prob_pick_dens` This parameter indicates the probability of choosing the threshold on which to split a variable (with ‘ndim=1‘) or a linear combination of variables (when using ‘ndim>1‘) as the threshold that maximizes the pooled densities of the branch distributions.

The ‘min_gain‘ option does not apply to this type of splits.

When splits are not made according to any of ‘prob_pick_avg_gain‘, ‘prob_pick_pooled_gain‘, ‘prob_pick_full_gain‘, ‘prob_pick_dens‘, both the column and the split point are decided at random. Default setting for [1], [2], [3], [4] is zero.

`prob_pick_col_by_range`

When using ‘ndim=1‘, this denotes the probability of choosing the column to split with a probability proportional to the range spanned by each column within a node as proposed in reference [12].

When using ‘ndim>1‘, this denotes the probability of choosing columns to create a hyperplane with a probability proportional to the range spanned by each column within a node.

This option is not compatible with categorical data. If passing column weights, the effect will be multiplicative.

Be aware that the data is not standardized in any way for the range calculations, thus the scales of features will make a large difference under this option, which might not make it suitable for all types of data.

If there are infinite values, all columns having infinite values will be treated as having the same weight, and will be chosen before every other column with non-infinite values.

Note that the proposed RRCF model from [12] uses a different scoring metric for producing anomaly scores, while this library uses isolation depth regardless of how columns are chosen, thus results are likely to be different from those of other software implementations. Nevertheless, as explored in [11], isolation depth as a scoring metric typically provides better results than the "co-displacement" metric from [12] under these split types.

`prob_pick_col_by_var`

When using ‘ndim=1‘, this denotes the probability of choosing the column to split with a probability proportional to the variance of each column within a node.

When using ‘ndim>1‘, this denotes the probability of choosing columns to create a hyperplane with a probability proportional to the variance of each column within a node.

For categorical data, it will calculate the expected variance if the column were converted to numerical by assigning to each category a random number ‘~ Unif(0, 1)‘, which depending on the number of categories and their distribution, produces numbers typically a bit smaller than standardized numerical variables.

Note that when using sparse matrices, the calculation of variance will rely on a procedure that uses sums of squares, which has less numerical precision than the calculation used for dense inputs, and as such, the results might differ slightly.

Be aware that this calculated variance is not standardized in any way, so the scales of features will make a large difference under this option.

If passing column weights, the effect will be multiplicative.

If passing a ‘missing_action‘ different than "fail", infinite values will be ignored for the variance calculation. Otherwise, all columns with infinite values will have the same probability and will be chosen before columns with non-infinite values.

`prob_pick_col_by_kurt`

When using ‘ndim=1‘, this denotes the probability of choosing the column to split with a probability proportional to the kurtosis of each column **within a node** (unlike the option ‘weigh_by_kurtosis‘ which calculates this metric only at the root).

When using ‘ndim>1‘, this denotes the probability of choosing columns to create a hyperplane with a probability proportional to the kurtosis of each column within a node.

For categorical data, it will calculate the expected kurtosis if the column were converted to numerical by assigning to each category a random number ‘~ Unif(0, 1)‘.

Note that when using sparse matrices, the calculation of kurtosis will rely on a procedure that uses sums of squares and higher-power numbers, which has less numerical precision than the calculation used for dense inputs, and as such, the results might differ slightly.

If passing column weights, the effect will be multiplicative. This option is not compatible with ‘weigh_by_kurtosis‘.

If passing a ‘missing_action‘ different than "fail", infinite values will be ignored for the kurtosis calculation. Otherwise, all columns with infinite values will have the same probability and will be chosen before columns with non-infinite values.

If using ‘missing_action="impute"‘, the calculation of kurtosis will not use imputed values in order not to favor columns with missing values (which would increase kurtosis by all having the same central value).

Be aware that kurtosis can be a rather slow metric to calculate.

`min_gain` Minimum gain that a split threshold needs to produce in order to proceed with a split. Only used when the splits are decided by a variance gain criterion (‘prob_pick_pooled_gain‘ or ‘prob_pick_avg_gain‘, but not ‘prob_pick_full_gain‘ nor ‘prob_pick_dens‘). If the highest possible gain in the evaluated splits at a node is below this threshold, that node becomes a terminal node.

This can be used as a more sophisticated depth control when using pooled gain (note that ‘max_depth‘ still applies on top of this heuristic).

`missing_action` How to handle missing data at both fitting and prediction time. Options are

- ‘"divide"‘ (for the single-variable model only, recommended), which will follow both branches and combine the result with the weight given by the fraction of the data that went to each branch when fitting the model. The weights are determined during fitting by producing a split based on the non-missing values only, and calculating which fraction of the non-missing values go to each branch (missing values are then sent to both branches with those weights to continue the fitting procedure).

- ‘"impute"‘, which will assign observations to the branch with the most observations in the single-variable model, or fill in missing values with the median of each column of the sample from which the split was made in the extended model (recommended for it) (but note that the calculation of medians does not take into account sample weights when using ‘weights_as_sample_prob=FALSE‘). When using ‘ndim=1‘, gain calculations will use median-imputed values for missing data under this option.
- ‘"fail"‘, which will assume there are no missing values and will trigger undefined behavior if it encounters any.

In the extended model, infinite values will be treated as missing. Passing ‘"fail"‘ will produce faster fitting and prediction times along with decreased model object sizes.

Models from references [1], [2], [3], [4] correspond to ‘"fail"‘ here.

Typically, models with ‘ndim>1‘ are less affected by missing data that models with ‘ndim=1‘.

`new_categ_action`

What to do after splitting a categorical feature when new data that reaches that split has categories that the sub-sample from which the split was done did not have. Options are

- ‘"weighted"‘ (for the single-variable model only, recommended), which will follow both branches and combine the result with weight given by the fraction of the data that went to each branch when fitting the model.
- ‘"impute"‘ (for the extended model only, recommended) which will assign them the median value for that column that was added to the linear combination of features (but note that this median calculation does not use sample weights when using ‘weights_as_sample_prob=FALSE‘).
- ‘"smallest"‘, which in the single-variable case will assign all observations with unseen categories in the split to the branch that had fewer observations when fitting the model, and in the extended case will assign them the coefficient of the least common category.
- ‘"random"‘, which will assign a branch (coefficient in the extended model) at random for each category beforehand, even if no observations had that category when fitting the model. Note that this can produce biased results when deciding splits by a gain criterion.
  Important: under this option, if the model is fitted to a ‘data.frame‘, when calling ‘predict‘ on new data which contains new factor levels (unseen in the data to which the model was fitted), they will be added to the model's state on-the-fly. This means that, if calling ‘predict‘ on data which has new categories, there might be inconsistencies in the results if predictions are done in parallel or if passing the same data in batches or with different row orders.

Ignored when passing ‘categ_split_type‘ = ‘"single_categ"‘.

`categ_split_type`

Whether to split categorical features by assigning sub-sets of them to each branch (by passing ‘"subset"‘ there), or by assigning a single category to a branch and the rest to the other branch (by passing ‘"single_categ"‘ here). For the extended

model, whether to give each category a coefficient (‘"subset"‘), or only one while the rest get zero (‘"single_categ"‘).

`all_perm` When doing categorical variable splits by pooled gain with ‘ndim=1‘ (single-variable model), whether to consider all possible permutations of variables to assign to each branch or not. If ‘FALSE‘, will sort the categories by their frequency and make a grouping in this sorted order. Note that the number of combinations evaluated (if ‘TRUE‘) is the factorial of the number of present categories in a given column (minus 2). For averaged gain, the best split is always to put the second most-frequent category in a separate branch, so not evaluating all permutations (passing ‘FALSE‘) will make it possible to select other splits that respect the sorted frequency order. Ignored when not using categorical variables or not doing splits by pooled gain or using ‘ndim>1‘.

`coef_by_prop` In the extended model, whether to sort the randomly-generated coefficients for categories according to their relative frequency in the tree node. This might provide better results when using categorical variables with too many categories, but is not recommended, and not reflective of real "categorical-ness". Ignored for the single-variable model (‘ndim=1‘) and/or when not using categorical variables.

`recode_categ` Whether to re-encode categorical variables even in case they are already passed as factors. This is recommended as it will eliminate potentially redundant categorical levels if they have no observations, but if the categorical variables are already of type ‘factor‘ with only the levels that are present, it can be skipped for slightly faster fitting times. You’ll likely want to pass ‘FALSE‘ here if merging several models into one through isotree.append.trees.

`weights_as_sample_prob`

If passing sample (row) weights when fitting the model, whether to consider those weights as row sampling weights (i.e. the higher the weights, the more likely the observation will end up included in each tree sub-sample), or as distribution density weights (i.e. putting a weight of two is the same as if the row appeared twice, thus higher weight makes it less of an outlier, but does not give it a higher chance of being sampled if the data uses sub-sampling).

`sample_with_replacement`

Whether to sample rows with replacement or not (not recommended). Note that distance calculations, if desired, don’t work when there are duplicate rows.

This option is not compatible with ‘output_score‘, ‘output_dist‘, ‘output_imputations‘.

`penalize_range` Whether to penalize (add -1 to the terminal depth) observations at prediction time that have a value of the chosen split variable (linear combination in extended model) that falls outside of a pre-determined reasonable range in the data being split (given by ‘2 * range‘ in data and centered around the split point), as proposed in reference [4] and implemented in the authors’ original code in reference [5]. Not used in single-variable model when splitting by categorical variables.

This option is not supported when using density-based outlier scoring metrics.

It’s recommended to turn this off for faster predictions on sparse CSC matrices.

Note that this can make a very large difference in the results when using ‘prob_pick_pooled_gain‘.

Be aware that this option can make the distribution of outlier scores a bit different (i.e. not centered around 0.5).

`standardize_data`
: Whether to standardize the features at each node before creating alinear combination of them as suggested in [4]. This is ignored when using ‘ndim=1‘.

`scoring_metric`
: Metric to use for determining outlier scores (see reference [13]). Options are:

- "depth": Will use isolation depth as proposed in reference [1]. This is typically the safest choice and plays well with all model types offered by this library.
- "density": Will set scores for each terminal node as the ratio between the fraction of points in the sub-sample that end up in that node and the fraction of the volume in the feature space which defines the node according to the splits that lead to it. If using ‘ndim=1‘, for categorical variables, this is defined in terms of number of categories that go towards each side of the split divided by number of categories in the observations that reached that node.
  The standardized outlier score from density for a given observation is calculated as the negative of the logarithm of the geometric mean from the per-tree densities, which unlike the standardized score produced from depth, is unbounded, but just like the standardized score from depth, has a natural threshold for defining outlierness, which in this case is zero is instead of 0.5. The non-standardized outlier score is calculated as the geometric mean, while the per-tree scores are calculated as the density values.
  This might lead to better predictions when using ‘ndim=1‘, particularly in the presence of categorical variables. Note however that using density requires more trees for convergence of scores (i.e. good results) compared to isolation-based metrics.
  This option is incompatible with ‘penalize_range‘.
- "adj_depth": Will use an adjusted isolation depth that takes into account the number of points that go to each side of a given split vs. the fraction of the range of that feature that each side of the split occupies, by a metric as follows:
  $d = \frac{2}{(1+\frac{1}{2p}}$
  Where p is defined as:
  $p = \frac{n_s}{n_t} \Big/ \frac{r_s}{r_t}$
  With n_t being the number of points that reach a given node, n_s the number of points that are sent to a given side of the split/branch at that node, r_t being the range (maximum minus minimum) of the splitting feature or linear combination among the points that reached the node, and r_s being the range of the same feature or linear combination among the points that are sent to this same side of the split/branch. This makes each split add a number between zero and two to the isolation depth, with this number's probabilistic distribution being centered around 1 and thus the expected isolation depth remains the same as in the original ‘"depth"‘ metric, but having more variability around the extremes.
  Scores (standardized, non-standardized, per-tree) are aggregated in the same way as for ‘"depth"‘.
  This might lead to better predictions when using ‘ndim=1‘, particularly in the presence of categorical variables and for smaller datasets, and for smaller datasets, might make sense to combine it with ‘penalize_range=TRUE‘.

- "adj_density": Will use the same metric from ‘"adj_depth"‘, but applied multiplicatively instead of additively. The expected value for this adjusted density is not strictly the same as for isolation, but using the expected isolation depth as standardizing criterion tends to produce similar standardized score distributions (centered around 0.5).
  Scores (standardized, non-standardized, per-tree) are aggregated in the same way as for ‘"depth"‘.
  This option is incompatible with ‘penalize_range‘.
- "boxed_ratio": Will set the scores for each terminal node as the ratio between the volume of the boxed feature space for the node as defined by the smallest and largest values from the split conditions for each column (bounded by the variable ranges in the sample) and the variable ranges in the tree sample. If using ‘ndim=1‘, for categorical variables this is defined in terms of number of categories. If using ‘ndim=>1‘, this is defined in terms of the maximum achievable value for the splitting linear combination determined from the minimum and maximum values for each variable among the points in the sample, and as such, it has a rather different meaning compared to the score obtained with ‘ndim=1‘ - boxed ratio scores with ‘ndim>1‘ typically provide very poor quality results and this metric is thus not recommended to use in the extended model. With ‘ndim>1‘, it also has a tendency of producing too small values which round to zero.
  The standardized outlier score from boxed ratio for a given observation is calculated simply as the the average from the per-tree boxed ratios. This metric has a lower bound of zero and a theoretical upper bound of one, but in practice the scores tend to be very small numbers close to zero, and its distribution across different datasets is rather unpredictable. In order to keep rankings comparable with the rest of the metrics, the non-standardized outlier scores are calculated as the negative of the average instead. The per-tree scores are calculated as the ratios.
  This metric can be calculated in a fast-but-not-so-precise way, and in a low-but-precise way, which is controlled by parameter ‘fast_bratio‘. Usually, both should give the same results, but in some fatasets, the fast way can lead to numerical inaccuracies due to roundoffs very close to zero.
  This metric might lead to better predictions in datasets with many rows when using ‘ndim=1‘ and a relatively small ‘sample_size‘. Note that more trees are required for convergence of scores when using this metric. In some datasets, this metric might result in very bad predictions, to the point that taking its inverse produces a much better ranking of outliers.
  This option is incompatible with ‘penalize_range‘.
- "boxed_density2": Will set the score as the ratio between the fraction of points within the sample that end up in a given terminal node and the boxed ratio metric.
  Aggregation of scores (standardized, non-standardized, per-tree) is done in the same way as for density, and it also has a natural threshold at zero for determining outliers and inliers.
  This metric is typically usable with ‘ndim>1‘, but tends to produce much bigger values compared to ‘ndim=1‘.
  Albeit unintuitively, in many datasets, one can usually get better results

with metric ‘"boxed_density"‘ instead.
The calculation of this metric is also controlled by ‘fast_bratio‘.
This option is incompatible with ‘penalize_range‘.

- "boxed_density": Will set the score as the ratio between the fraction of points within the sample that end up in a given terminal node and the ratio between the boxed volume of the feature space in the sample and the boxed volume of a node given by the split conditions (inverse as in ‘"boxed_density2"‘). This metric does not have any theoretical or intuitive justification behind its existence, and it is perhaps illogical to use it as a scoring metric, but tends to produce good results in some datasets.
  The standardized outlier scores are defined as the negative of the geometric mean of this metric, while the non-standardized scores are the geometric mean, and the per-tree scores are simply the 'density' values.
  The calculation of this metric is also controlled by ‘fast_bratio‘.
  This option is incompatible with ‘penalize_range‘.

`fast_bratio` When using "boxed" metrics for scoring, whether to calculate them in a fast way through cumulative sum of logarithms of ratios after each split, or in a slower way as sum of logarithms of a single ratio per column for each terminal node.

Usually, both methods should give the same results, but in some datasets, particularly when variables have too small or too large ranges, the first method can be prone to numerical inaccuracies due to roundoff close to zero.

Note that this does not affect calculations for models with ‘ndim>1‘, since given the split types, the calculation for them is different.

`weigh_by_kurtosis`
Whether to weigh each column according to the kurtosis obtained in the subsample that is selected for each tree as briefly proposed in reference [1]. Note that this is only done at the beginning of each tree sample. For categorical columns, will calculate expected kurtosis if the column were converted to numerical by assigning to each category a random number ‘~ Unif(0, 1)‘.

Note that when using sparse matrices, the calculation of kurtosis will rely on a procedure that uses sums of squares and higher-power numbers, which has less numerical precision than the calculation used for dense inputs, and as such, the results might differ slightly.

Using this option makes the model more likely to pick the columns that have anomalous values when viewed as a 1-d distribution, and can bring a large improvement in some datasets.

This is intended as a cheap feature selector, while the parameter ‘prob_pick_col_by_kurt‘ provides the option to do this at each node in the tree for a different overall type of model.

If passing column weights or using weighted column choices proportional to some other metric (‘prob_pick_col_by_range‘, ‘prob_pick_col_by_var‘), the effect will be multiplicative.

If passing ‘missing_action="fail"‘ and the data has infinite values, columns with rows having infinite values will get a weight of zero. If passing a different value for missing action, infinite values will be ignored in the kurtosis calculation.

If using ‘missing_action="impute"‘, the calculation of kurtosis will not use imputed values in order not to favor columns with missing values (which would

increase kurtosis by all having the same central value).

`coefs` For the extended model, whether to sample random coefficients according to a normal distribution ‘~ N(0, 1)‘ (as proposed in reference [4]) or according to a uniform distribution ‘~ Unif(-1, +1)‘ as proposed in reference [3]. Ignored for the single-variable model. Note that, for categorical variables, the coefficients will be sampled ~ N (0,1) regardless - in order for both types of variables to have transformations in similar ranges (which will tend to boost the importance of categorical variables), pass ‘"uniform"‘ here.

`assume_full_distr`

When calculating pairwise distances (see reference [8]), whether to assume that the fitted model represents a full population distribution (will use a standardizing criterion assuming infinite sample as in reference [6], and the results of the similarity between two points at prediction time will not depend on the presence of any third point that is similar to them, but will differ more compared to the pairwise distances between points from which the model was fit). If passing ‘FALSE‘, will calculate pairwise distances as if the new observations at prediction time were added to the sample to which each tree was fit, which will make the distances between two points potentially vary according to other newly introduced points. This will not be assumed when the distances are calculated as the model is being fit (see documentation for parameter ‘output_dist‘).

This was added for experimentation purposes only and it’s not recommended to pass ‘FALSE‘. Note that when calculating distances using a tree indexer (after calling isotree.build.indexer), there might be slight discrepancies between the numbers produced with or without the indexer due to what are considered "additional" observations in this calculation.

`build_imputer` Whether to construct missing-value imputers so that later this same model could be used to impute missing values of new (or the same) observations. Be aware that this will significantly increase the memory requirements and serialized object sizes. Note that this is not related to ’missing_action’ as missing values inside the model are treated differently and follow their own imputation or division strategy.

`output_imputations`

Whether to output imputed missing values for ‘data‘. Passing ‘TRUE‘ here will force ‘build_imputer‘ to ‘TRUE‘. Note that, for sparse matrix inputs, even though the output will be sparse, it will generate a dense representation of each row with missing values.

This is not supported when using sub-sampling, and if sub-sampling is specified, will override it using the full number of rows.

`min_imp_obs` Minimum number of observations with which an imputation value can be produced. Ignored if passing ‘build_imputer‘ = ‘FALSE‘.

`depth_imp` How to weight observations according to their depth when used for imputing missing values. Passing ‘"higher"‘ will weigh observations higher the further down the tree (away from the root node) the terminal node is, while ‘"lower"‘ will do the opposite, and ‘"same"‘ will not modify the weights according to node depth in the tree. Implemented for testing purposes and not recommended to change from the default. Ignored when passing ‘build_imputer‘ = ‘FALSE‘.

`weigh_imp_rows` How to weight node sizes when used for imputing missing values. Passing ‘"inverse"‘ will weigh a node inversely proportional to the number of observations that end up there, while ‘"prop"‘ will weight them heavier the more observations there are, and ‘"flat"‘ will weigh all nodes the same in this regard regardless of how many observations end up there. Implemented for testing purposes and not recommended to change from the default. Ignored when passing ‘build_imputer‘ = ‘FALSE‘.

`output_score` Whether to output outlierness scores for the input data, which will be calculated as the model is being fit and it’s thus faster. Cannot be done when using sub-samples of the data for each tree (in such case will later need to call the ‘predict‘ function on the same data). If using ‘penalize_range‘, the results from this might differ a bit from those of ‘predict‘ called after.

This is not supported when using sub-sampling, and if sub-sampling is specified, will override it using the full number of rows.

`output_dist` Whether to output pairwise distances for the input data, which will be calculated as the model is being fit and it’s thus faster. Cannot be done when using sub-samples of the data for each tree (in such case will later need to call the ‘predict‘ function on the same data). If using ‘penalize_range‘, the results from this might differ a bit from those of ‘predict‘ called after.

This is not supported when using sub-sampling, and if sub-sampling is specified, will override it using the full number of rows.

Note that it might be much faster to calculate distances through a fitted model object with isotree.build.indexer instead or calculating them while fitting like this.

`square_dist` If passing ‘output_dist‘ = ‘TRUE‘, whether to return a full square matrix or just the upper-triangular part, in which the entry for pair (i,j) with 1 <= i < j <= n is located at position p(i, j) = ((i - 1) * (n - i/2) + j - i).

`sample_weights` Sample observation weights for each row of ‘data‘, with higher weights indicating either higher sampling probability (i.e. the observation has a larger effect on the fitted model, if using sub-samples), or distribution density (i.e. if the weight is two, it has the same effect of including the same data point twice), according to parameter ‘weights_as_sample_prob‘. Not supported when calculating pairwise distances while the model is being fit (done by passing ‘output_dist‘ = ‘TRUE‘).

If ‘data‘ is a ‘data.frame‘ and the variable passed here matches to the name of a column in ‘data‘ (with or without enclosing ‘sample_weights‘ in quotes), it will assume the weights are to be taken as that column name.

`column_weights` Sampling weights for each column in ‘data‘. Ignored when picking columns by deterministic criterion. If passing ‘NULL‘, each column will have a uniform weight. If used along with kurtosis weights, the effect is multiplicative.

Note that, if passing a data.frame with both numeric and categorical columns, the column names must not be repeated, otherwise the column weights passed here will not end up matching. If passing a ‘data.frame‘ to ‘data‘, will assume the column order is the same as in there, regardless of whether the entries passed to ‘column_weights‘ are named or not.

`lazy_serialization`

Whether to use a lazy serialization mechanism for the model C++ objects through the ALTREP system, which would only call the serialization and de-serialization methods when needed.

Passing 'TRUE' here has the following effects:

- Fitting the model will not immediately trigger serialization of the model object, not will it need to allocate extra memory for the serialized bytes - instead, these serialized bytes will only get materialized when calling serialization functions such as 'save' or 'saveRDS'.
- The resulting object will not be possible to serialize with library 'qs' ('qs::qsave'), nor with other serialization libraries that do not work with R's ALTREP system.
- If restoring a session with saved objects or loading serialized models through 'load', if there is not enough memory to de-serialize the model object, this will be manifested as silent failures in which the object simply disappears from the environment without leaving any trace or error message.
- When reading a model through 'readRDS', if there's an error (such as 'insufficient memory'), it will fail to load the R object at all.
- Since this uses a workaround for which the ALTREP system was not initially designed, calling methods such as 'str' on the resulting object will result in errors being displayed when it comes to the external pointer fields.

If passing 'FALSE', on the other hand:

- Immediately after fitting the model, this fitted model will be serialized into memory-contiguous raw bytes from which the C++ object can then be reconstructed.
- If the model gets de-serialized from a saved file (for example through 'load', 'readRDS', 'qs::qread', or by restarting R sessions), the underlying C++ model object will be lost, and as such will need to be restored (de-serialized from the serialized bytes) the first time a method like 'predict' gets called on it (which means the first call will be slower and will result in additional memory allocations).

`seed` Seed that will be used for random number generation.

`use_long_double`

Whether to use 'long double' (extended precision) type for more precise calculations about standard deviations, means, ratios, weights, gain, and other potential aggregates. This makes such calculations accurate to a larger number of decimals (provided that the compiler used has wider long doubles than doubles) and it is highly recommended to use when the input data has a number of rows or columns exceeding 2^{53} (an unlikely scenario), and also highly recommended to use when the input data has problematic scales (e.g. numbers that differ from each other by something like 10^{-100} or columns that include values like 10^{100}, 10^{-10}, and 10^{-100} and still need to be sensitive to a difference of 10^{-10}), but will make the calculations slower, the more so in platforms in which 'long double' is a software-emulated type (e.g. Power8 platforms). Note that some platforms (most notably windows with the msvc compiler) do not make any difference between 'double' and 'long double'.

If 'long double' is not going to be used, the library can be compiled without support for it (making the library size smaller) by defining an environment variable 'NO_LONG_DOUBLE' before installing this package (e.g. through 'Sys.setenv("NO_LONG_DOUBLE" = "1")' before running the 'install.packages' command). If R itself was compiled without 'long double' support, this library will follow suit and disable long double too.

This option is not available on Windows, due to lack of support in some compilers (e.g. msvc) and lack of thread-safety in the calculations in others (e.g. mingw).

`nthreads` Number of parallel threads to use. Note that, the more threads, the more memory will be allocated, even if the thread does not end up being used. Be aware that most of the operations are bound by memory bandwidth, which means that adding more threads will not result in a linear speed-up. For some types of data (e.g. large sparse matrices with small sample sizes), adding more threads might result in only a very modest speed up (e.g. 1.5x faster with 4x more threads), even if all threads look fully utilized.

Details

If requesting outlier scores or depths or separation/distance while fitting the model and using multiple threads, there can be small differences in the predicted scores/depth/separation/distance between runs due to roundoff error.

Value

If passing 'output_score' = 'FALSE', 'output_dist' = 'FALSE', and 'output_imputations' = 'FALSE' (the defaults), will output an 'isolation_forest' object from which 'predict' method can then be called on new data.

If passing 'TRUE' to any of the former options, will output a list with entries:

- 'model': the 'isolation_forest' object from which new predictions can be made.
- 'scores': a vector with the outlier score for each inpuit observation (if passing 'output_score' = 'TRUE').
- 'dist': the distances (either a 'dist' object or a square matrix), if passing 'output_dist' = 'TRUE'.
- 'imputed': the input data with missing values imputed according to the model (if passing 'output_imputations' = 'TRUE').

Matching models from references

Shorthands for parameter combinations that match some of the references:

- 'iForest' (reference [1]): 'ndim=1', 'sample_size=256', 'max_depth=8', 'ntrees=100', 'missing_action="fail"'.
- 'EIF' (reference [3]): 'ndim=2', 'sample_size=256', 'max_depth=8', 'ntrees=100', 'missing_action="fail"', 'coefs="uniform"', 'standardize_data=False' (plus standardizing the data **before** passing it).

- 'SCiForest' (reference [4]): 'ndim=2', 'sample_size=256', 'max_depth=8', 'ntrees=100', 'missing_action="fail"', 'coefs="normal"', 'ntry=10', 'prob_pick_avg_gain=1', 'penalize_range=True'. Might provide much better results with 'max_depth=NULL' despite the reference's recommendation.
- 'FCF' (reference [11]): 'ndim=2', 'sample_size=256', 'max_depth=NULL', 'ntrees=200', 'missing_action="fail"', 'coefs="normal"', 'ntry=1', 'prob_pick_pooled_gain=1'. Might provide similar or better results with 'ndim=1' and/or sample size as low as 32. For the FCF model aimed at imputing missing values, might give better results with 'ntry=10' or higher and much larger sample sizes.
- 'RRCF' (reference [12]): 'ndim=1', 'prob_pick_col_by_range=1', 'sample_size=256' or more, 'max_depth=NULL', 'ntrees=100' or more, 'missing_action="fail"'. Note however that reference [12] proposed a different method for calculation of anomaly scores, while this library uses isolation depth just like for 'iForest', so results might differ significantly from those of other libraries. Nevertheless, experiments in reference [11] suggest that isolation depth might be a better scoring metric for this model.

Model serving considerations

If the model is built with 'nthreads>1', the prediction function predict.isolation_forest will use OpenMP for parallelization. In a linux setup, one usually has GNU's "gomp" as OpenMP as backend, which will hang when used in a forked process - for example, if one tries to call this prediction function from 'RestRserve', which uses process forking for parallelization, it will cause the whole application to freeze. A potential fix in these cases is to pass 'nthreads=1' to 'predict', or to set the number of threads to 1 in the model object (e.g. 'model$nthreads <- 1L' or calling isotree.set.nthreads), or to compile this library without OpenMP (requires manually altering the 'Makevars' file), or to use a non-GNU OpenMP backend (such as LLVM's 'libomp'. This should not be an issue when using this library normally in e.g. an RStudio session.

The R objects that hold the models contain heap-allocated C++ objects which do not map to R types and which thus do not survive serializations the same way R objects do. In order to make model objects serializable (i.e. usable with 'save', 'saveRDS', and similar), the package offers two mechanisms: (a) a 'lazy_serialization' option which uses the ALTREP system as a workaround, by defining classes with serialization methods but without datapointer methods (see the docs for 'lazy_serialization' for more info); (b) a more theoretically correct way in which raw bytes are produced alongside the model and from which the C++ objects can be reconstructed. When using the lazy serialization system, C++ objects are restored automatically on load and the serialized bytes then discarded, but this is not the case when using the serialized bytes approach. For model serving, one would usually want to drop these serialized bytes after having loaded a model through 'readRDS' or similar (note that reconstructing the C++ object will first require calling isotree.restore.handle, which is done automatically when calling 'predict' and similar), as they can increase memory usage by a large amount. These redundant raw bytes can be dropped as follows: 'model$cpp_objects$model$ser <- NULL' (and an additional 'model$cpp_objects$imputer$ser <- NULL' when using 'build_imputer=TRUE', and 'model$cpp_objects$indexer$ser <- NULL' when building a node indexer). After that, one might want to force garbage collection through 'gc()'.

Usually, for serving purposes, one wants a setup as minimalistic as possible (e.g. smaller docker images). This library can be made smaller and faster to compile by disabling some features - particularly, the library will by default build with support for calculation of aggregated metrics (such as standard deviations) in 'long double' precision (an extended precision type), which is a functionality that's unlikely to get used (default is not to use this type as it is slower, and calculations done in the

'predict' function do not use it for anything). Support for 'long double' can be disable at compile time by setting up an environment variable 'NO_LONG_DOUBLE' before installing the package (e.g. by issuing command 'Sys.setenv("NO_LONG_DOUBLE" = "1")' before 'install.packages').

References

1. Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation forest." 2008 Eighth IEEE International Conference on Data Mining. IEEE, 2008.
2. Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation-based anomaly detection." ACM Transactions on Knowledge Discovery from Data (TKDD) 6.1 (2012): 3.
3. Hariri, Sahand, Matias Carrasco Kind, and Robert J. Brunner. "Extended Isolation Forest." arXiv preprint arXiv:1811.02141 (2018).
4. Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. "On detecting clustered anomalies using SCiForest." Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, Berlin, Heidelberg, 2010.
5. https://sourceforge.net/projects/iforest/
6. https://math.stackexchange.com/questions/3388518/expected-number-of-paths-required-to-separate
7. Quinlan, J. Ross. "C4. 5: programs for machine learning." Elsevier, 2014.
8. Cortes, David. "Distance approximation using Isolation Forests." arXiv preprint arXiv:1910.12362 (2019).
9. Cortes, David. "Imputing missing values with unsupervised random trees." arXiv preprint arXiv:1911.06646 (2019).
10. https://math.stackexchange.com/questions/3333220/expected-average-depth-in-random-binary-tree-
11. Cortes, David. "Revisiting randomized choices in isolation forests." arXiv preprint arXiv:2110.13402 (2021).
12. Guha, Sudipto, et al. "Robust random cut forest based anomaly detection on streams." International conference on machine learning. PMLR, 2016.
13. Cortes, David. "Isolation forests: looking beyond tree depth." arXiv preprint arXiv:2111.11639 (2021).
14. Ting, Kai Ming, Yue Zhu, and Zhi-Hua Zhou. "Isolation kernel and its effect on SVM." Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018.

See Also

predict.isolation_forest, isotree.add.tree isotree.restore.handle

Examples

```
### Example 1: detect an obvious outlier
### (Random data from a standard normal distribution)
library(isotree)
set.seed(1)
m <- 100
n <- 2
X <- matrix(rnorm(m * n), nrow = m)
```

```
### Will now add obvious outlier point (3, 3) to the data
X <- rbind(X, c(3, 3))

### Fit a small isolation forest model
iso <- isolation.forest(X, ntrees = 10, nthreads = 1)

### Check which row has the highest outlier score
pred <- predict(iso, X)
cat("Point with highest outlier score: ",
    X[which.max(pred), ], "\n")


### Example 2: plotting outlier regions
### This example shows predicted outlier score in a small
### grid, with a model fit to a bi-modal distribution. As can
### be seen, the extended model is able to detect high
### outlierness outside of both regions, without having false
### ghost regions of low-outlierness where there isn't any data
library(isotree)
oldpar <- par(mfrow = c(2, 2), mar = c(2.5,2.2,2,2.5))

### Randomly-generated data from different distributions
set.seed(1)
group1 <- data.frame(x = rnorm(1000, -1, .4),
    y = rnorm(1000, -1, .2))
group2 <- data.frame(x = rnorm(1000, +1, .2),
    y = rnorm(1000, +1, .4))
X = rbind(group1, group2)

### Add an obvious outlier which is within the 1d ranges
### (As an interesting test, remove the outlier and see what happens,
###  or check how its score changes when using sub-sampling or
###  changing the scoring metric for 'ndim=1')
X = rbind(X, c(-1, 1))

### Produce heatmaps
pts = seq(-3, 3, .1)
space_d <- expand.grid(x = pts, y = pts)
plot.space <- function(Z, ttl) {
    image(pts, pts, matrix(Z, nrow = length(pts)),
      col = rev(heat.colors(50)),
      main = ttl, cex.main = 1.4,
      xlim = c(-3, 3), ylim = c(-3, 3),
      xlab = "", ylab = "")
    par(new = TRUE)
    plot(X, type = "p", xlim = c(-3, 3), ylim = c(-3, 3),
     col = "#0000801A",
     axes = FALSE, main = "",
     xlab = "", ylab = "")
}

### Now try out different variations of the model
```

```
### Single-variable model
iso_simple = isolation.forest(
    X, ndim=1,
    ntrees=100,
    nthreads=1,
    penalize_range=FALSE,
    prob_pick_pooled_gain=0,
    prob_pick_avg_gain=0)
Z1 <- predict(iso_simple, space_d)
plot.space(Z1, "Isolation Forest")

### Extended model
iso_ext = isolation.forest(
     X, ndim=2,
     ntrees=100,
     nthreads=1,
     penalize_range=FALSE,
     prob_pick_pooled_gain=0,
     prob_pick_avg_gain=0)
Z2 <- predict(iso_ext, space_d)
plot.space(Z2, "Extended Isolation Forest")

### SCiForest
iso_sci = isolation.forest(
     X, ndim=2, ntry=1,
     coefs="normal",
     ntrees=100,
     nthreads=1,
     penalize_range=TRUE,
     prob_pick_pooled_gain=0,
     prob_pick_avg_gain=1)
Z3 <- predict(iso_sci, space_d)
plot.space(Z3, "SCiForest")

### Fair-cut forest
iso_fcf = isolation.forest(
     X, ndim=2,
     ntrees=100,
     nthreads=1,
     penalize_range=FALSE,
     prob_pick_pooled_gain=1,
     prob_pick_avg_gain=0)
Z4 <- predict(iso_fcf, space_d)
plot.space(Z4, "Fair-Cut Forest")
par(oldpar)

### (As another interesting variation, try setting
###  'penalize_range=TRUE' for the last model)

### Example 3: calculating pairwise distances,
### with a short validation against euclidean dist.
library(isotree)
```

```
### Generate random data with 3 dimensions
set.seed(1)
m <- 100
n <- 3
X <- matrix(rnorm(m * n), nrow=m, ncol=n)

### Fit isolation forest model
iso <- isolation.forest(X, ndim=2, ntrees=100, nthreads=1)

### Calculate distances with the model
### (this can be accelerated with 'isotree.build.indexer')
D_iso <- predict(iso, X, type = "dist")

### Check that it correlates with euclidean distance
D_euc <- dist(X, method = "euclidean")

cat(sprintf("Correlation with euclidean distance: %f\n",
    cor(D_euc, D_iso)))
### (Note that euclidean distance will never take
###  any correlations between variables into account,
###  which the isolation forest model can do)


### Example 4: imputing missing values
### (requires package MASS)
library(isotree)

### Generate random data, set some values as NA
if (require("MASS")) {
  set.seed(1)
  S <- crossprod(matrix(rnorm(5 * 5), nrow = 5))
  mu <- rnorm(5)
  X <- MASS::mvrnorm(1000, mu, S)
  X_na <- X
  values_NA <- matrix(runif(1000 * 5) < .15, nrow = 1000)
  X_na[values_NA] = NA

  ### Impute missing values with model
  iso <- isolation.forest(
      X_na,
      build_imputer = TRUE,
      prob_pick_pooled_gain = 1,
      ndim = 2,
      ntry = 10,
      nthreads = 1
  )
  X_imputed <- predict(iso, X_na, type = "impute")
  cat(sprintf("MSE for imputed values w/model: %f\n",
      mean((X[values_NA] - X_imputed[values_NA])^2)))

  ### Compare against simple mean imputation
  X_means <- apply(X, 2, mean)
```

```
  X_imp_mean <- X_na
  for (cl in 1:5)
      X_imp_mean[values_NA[,cl], cl] <- X_means[cl]
  cat(sprintf("MSE for imputed values w/means: %f\n",
      mean((X[values_NA] - X_imp_mean[values_NA])^2)))
}



#### A more interesting example
#### (requires package outliertree)

### Compare outliers returned by these different methods,
### and see why some of the outliers returned by the
### isolation forest could be flagged as outliers
if (require("outliertree")) {
  hypothyroid <- outliertree::hypothyroid

  iso <- isolation.forest(hypothyroid, nthreads=1)
  pred_iso <- predict(iso, hypothyroid)
  otree <- outliertree::outlier.tree(
      hypothyroid,
      z_outlier = 6,
      pct_outliers = 0.02,
      outliers_print = 20,
      nthreads = 1)

  ### Now compare against the top
  ### outliers from isolation forest
  head(hypothyroid[order(-pred_iso), ], 20)
}
```

`isotree.add.tree` *Add additional (single) tree to isolation forest model*

Description

Adds a single tree fit to the full (non-subsampled) data passed here. Must have the same columns as previously-fitted data. Categorical columns, if any, may have new categories.

Usage

```
isotree.add.tree(
  model,
  data,
  sample_weights = NULL,
  column_weights = NULL,
  refdata = NULL
)
```

Arguments

`model`	An Isolation Forest object as returned by isolation.forest, to which an additional tree will be added. **This object will be modified in-place**.
`data`	A ‘data.frame‘, ‘data.table‘, ‘tibble‘, ‘matrix‘, or sparse matrix (from package ‘Matrix‘ or ‘SparseM‘, CSC format) to which to fit the new tree.
`sample_weights`	Sample observation weights for each row of ’X’, with higher weights indicating distribution density (i.e. if the weight is two, it has the same effect of including the same data point twice). If not ‘NULL‘, model must have been built with ‘weights_as_sample_prob‘ = ‘FALSE‘.
`column_weights`	Sampling weights for each column in ‘data‘. Ignored when picking columns by deterministic criterion. If passing ‘NULL‘, each column will have a uniform weight. If used along with kurtosis weights, the effect is multiplicative.
`refdata`	Reference points for distance and/or kernel calculations, if these were previously added to the model object through isotree.set.reference.points. Must correspond to the same points that were passed in the call to that function. If sparse, only CSC format is supported. This is ignored if the model has no stored reference points.

Details

If constructing trees with different sample sizes, the outlier scores with depth-based metrics will not be centered around 0.5 and might have a very skewed distribution. The standardizing constant for the scores will be taken according to the sample size passed in the model construction argument.

If trees are going to be fit to samples of different sizes, it’s strongly recommended to use density-based scoring metrics instead.

Be aware that, if an out-of-memory error occurs, the resulting object might be rendered unusable (might crash when calling certain functions).

For safety purposes, the model object can be deep copied (including the underlying C++ object) through function isotree.deep.copy before undergoing an in-place modification like this.

If this function is going to be called frequently, it’s highly recommended to use ‘lazy_serialization=TRUE‘ as then it will not need to copy over serialized bytes.

Value

The same ‘model‘ object now modified, as invisible.

See Also

isolation.forest isotree.restore.handle

`isotree.append.trees` *Append isolation trees from one model into another*

Description

This function is intended for merging models **that use the same hyperparameters** but were fitted to different subsets of data.

In order for this to work, both models must have been fit to data in the same format - that is, same number of columns, same order of the columns, and same column types, although not necessarily same object classes (e.g. can mix ‘base::matrix‘ and ‘Matrix::dgCMatrix‘).

If the data has categorical variables, the models should have been built with parameter ‘recode_categ=FALSE‘ in the call to isolation.forest, and the categorical columns passed as type ‘factor‘ with the same ‘levels‘ - otherwise different models might be using different encodings for each categorical column, which will not be preserved as only the trees will be appended without any associated metadata.

Note that this function will not perform any checks on the inputs, and passing two incompatible models (e.g. fit to different numbers of columns) will result in wrong results and potentially crashing the R process when using the resulting object.

Also be aware that the first input will be modified in-place.

If using ‘lazy_serialization=FALSE‘, this will trigger a re-serialization so it will be slower than if using ‘lazy_serialization=TRUE‘.

Usage

```
isotree.append.trees(model, other)
```

Arguments

`model`	An Isolation Forest model (as returned by function isolation.forest) to which trees from ‘other‘ (another Isolation Forest model) will be appended into. **Will be modified in-place**, and on exit will contain the resulting merged model.
`other`	Another Isolation Forest model, from which trees will be appended into ‘model‘. It will not be modified during the call to this function.

Details

Be aware that, if an out-of-memory error occurs, the resulting object might be rendered unusable (might crash when calling certain functions).

For safety purposes, the model object can be deep copied (including the underlying C++ object) through function isotree.deep.copy before undergoing an in-place modification like this.

Value

The same input ‘model‘ object, now with the new trees appended, returned as invisible.

Examples

```
library(isotree)

### Generate two random sets of data
m <- 100
n <- 2
set.seed(1)
X1 <- matrix(rnorm(m*n), nrow=m)
X2 <- matrix(rnorm(m*n), nrow=m)

### Fit a model to each dataset
iso1 <- isolation.forest(X1, ntrees=3, ndim=2, nthreads=1)
iso2 <- isolation.forest(X2, ntrees=2, ndim=2, nthreads=1)

### Check the terminal nodes for some observations
nodes1 <- predict(iso1, head(X1, 3), type="tree_num")
nodes2 <- predict(iso2, head(X1, 3), type="tree_num")

### Check also the average isolation depths
nodes1.depths <- predict(iso1, head(X1, 3), type="avg_depth")
nodes2.depths <- predict(iso2, head(X1, 3), type="avg_depth")

### Append the trees from 'iso2' into 'iso1'
iso1 <- isotree.append.trees(iso1, iso2)

### Check that it predicts the same as the two models
nodes.comb <- predict(iso1, head(X1, 3), type="tree_num")
nodes.comb == cbind(nodes1, nodes2)

### The new predicted scores will be a weighted average
### (Be aware that, due to round-off, it will not match with '==')
nodes.comb.depths <- predict(iso1, head(X1, 3), type="avg_depth")
nodes.comb.depths
(3*nodes1.depths + 2*nodes2.depths) / 5
```

---

isotree.build.indexer *Build Indexer for Faster Terminal Node Predictions and/or Distance Calculations*

Description

Builds an index of terminal nodes for faster prediction of terminal node numbers (calling ‘predict‘ with ‘type="tree_num"‘).

Optionally, can also pre-calculate terminal node distances in order to speed up distance calculations (calling ‘predict‘ with ‘type="dist"‘ or ‘type="avg_sep"‘).

Usage

```
isotree.build.indexer(model, with_distances = FALSE, nthreads = model$nthreads)
```

Arguments

`model`	An Isolation Forest model (as returned by function isolation.forest) for which an indexer for terminal node numbers and/or distances will be added. **The object will be modified in-place**.
`with_distances`	Whether to also pre-calculate node distances in order to speed up ‘predict‘ with ‘type="dist"‘ or ‘type="avg_sep"‘. Note that this will consume a lot more memory and make the resulting object significantly heavier.
`nthreads`	Number of parallel threads to use.

Details

This feature is not available for models that use ‘missing_action="divide"‘ or ‘new_categ_action="weighted"‘ (which are the defaults when passing ‘ndim=1‘).

Value

The same ‘model‘ object (as invisible), but now with an indexer added to it. Note the input object is modified in-place regardless.

See Also

isotree.drop.indexer

`isotree.deep.copy` *Deep-Copy an Isolation Forest Model Object*

Description

Generates a deep copy of a model object, including the C++ objects inside it. This function is only meaningful if one intends to call a function that modifies the internal C++ objects - currently, the only such function are isotree.add.tree and isotree.append.trees - as otherwise R’s objects follow a copy-on-write logic.

Usage

```
isotree.deep.copy(model)
```

Arguments

`model`	An ‘isolation_forest‘ model object.

Value

A new ‘isolation_forest‘ object, with deep-copied C++ objects.

See Also

isotree.is.same

`isotree.drop.imputer` *Drop Imputer Sub-Object from Isolation Forest Model Object*

Description

Drops the imputer sub-object from an isolation forest model object, if it was fitted with data imputation capabilities. The imputer, if constructed, is likely to be a very heavy object which might not be needed for all purposes.

Usage

```
isotree.drop.imputer(model, manually_delete_cpp = TRUE)
```

Arguments

`model`	An ‘isolation_forest‘ model object.
`manually_delete_cpp`	Whether to manually delete the underlying C++ object after calling this function. If passing ‘FALSE‘, memory will not be freed until the underlying R ’externalptr’ object is garbage-collected, which typically happens after the next call to ‘gc()‘. If passing ‘TRUE‘, will manually delete the C++ object held in the ’externalptr’ object before nullifying it. Note that, if somehow one assigned the pointer address to some other R variable through e.g. a deep copy of the ’externalptr’ object (that happened without copying the full model object where this R variable is stored), then other pointers pointing at the same address might trigger crashes at the moment they are used. Note that, unless one starts manually fiddling with the internals of model objects and assigning variables to/from them, it should not be possible to end up in a situation in which an ’externalptr’ object ends up deep-copied, especially when using ‘lazy_serialization=TRUE‘.

Value

The same ‘model‘ object, but now with the imputer removed. **Note that ‘model‘ is modified inplace in any event**.

`isotree.drop.indexer` *Drop Indexer Sub-Object from Isolation Forest Model Object*

Description

Drops the indexer sub-object from an isolation forest model object, if it was constructed. The indexer, if constructed, is likely to be a very heavy object which might not be needed for all purposes.

Usage

```
isotree.drop.indexer(model, manually_delete_cpp = TRUE)
```

Arguments

`model`	An ‘isolation_forest‘ model object.
`manually_delete_cpp`	Whether to manually delete the underlying C++ object after calling this function. If passing ‘FALSE‘, memory will not be freed until the underlying R ’externalptr’ object is garbage-collected, which typically happens after the next call to ‘gc()‘. If passing ‘TRUE‘, will manually delete the C++ object held in the ’externalptr’ object before nullifying it. Note that, if somehow one assigned the pointer address to some other R variable through e.g. a deep copy of the ’externalptr’ object (that happened without copying the full model object where this R variable is stored), then other pointers pointing at the same address might trigger crashes at the moment they are used. Note that, unless one starts manually fiddling with the internals of model objects and assigning variables to/from them, it should not be possible to end up in a situation in which an ’externalptr’ object ends up deep-copied, especially when using ‘lazy_serialization=TRUE‘.

Details

Note that reference points as added through isotree.set.reference.points are associated with the indexer object and will also be dropped if any were added.

Value

The same ‘model‘ object, but now with the indexer removed. **Note that ‘model‘ is modified inplace in any event**.

See Also

isotree.build.indexer

isotree.drop.reference.points *Drop Reference Points from Isolation Forest Model Object*

Description

Drops any reference points used for distance and/or kernel calculations from the model object, if any were set through isotree.set.reference.points.

Usage

```
isotree.drop.reference.points(model)
```

Arguments

model	An ‘isolation_forest‘ model object.

Value

The same ‘model‘ object, but now with the reference points removed. **Note that ‘model‘ is modified in-place in any event**.

See Also

isotree.set.reference.points

isotree.export.model *Export Isolation Forest model*

Description

Save Isolation Forest model to a serialized file along with its metadata, in order to be used in the Python or the C++ versions of this package.

This function is not suggested to be used for passing models to and from R - in such case, one can use ‘saveRDS‘ and ‘readRDS‘ instead, although the function still works correctly for serializing objects between R sessions.

Note that, if the model was fitted to a ‘data.frame‘, the column names must be something exportable as JSON, and must be something that Python’s Pandas could use as column names (e.g. strings/character).

Can optionally generate a JSON file with metadata such as the column names and the levels of categorical variables, which can be inspected visually in order to detect potential issues (e.g. character encoding) or to make sure that the columns are of the right types.

Requires the ‘jsonlite‘ package in order to work.

Usage

```
isotree.export.model(model, file, add_metadata_file = FALSE)
```

Arguments

`model`	An Isolation Forest model as returned by function isolation.forest.
`file`	File path where to save the model. File connections are not accepted, only file paths
`add_metadata_file`	Whether to generate a JSON file with metadata, which will have the same name as the model but will end in '.metadata'. This file is not used by the de-serialization function, it's only meant to be inspected manually, since such contents will already be written in the produced model file.

Details

The metadata file, if produced, will contain, among other things, the encoding that was used for categorical columns - this is under 'data_info.cat_levels', as an array of arrays by column, with the first entry for each column corresponding to category 0, second to category 1, and so on (the C++ version takes them as integers). When passing 'categ_cols', there will be no encoding but it will save the maximum category integer and the column numbers instead of names.

The serialized file can be used in the C++ version by reading it as a binary file and de-serializing its contents using the C++ function 'deserialize_combined' (recommended to use 'inspect_serialized_object' beforehand).

Be aware that this function will write raw bytes from memory as-is without compression, so the file sizes can end up being much larger than when using 'saveRDS'.

The metadata is not used in the C++ version, but is necessary for the R and Python versions.

Note that the model treats boolean/logical variables as categorical. Thus, if the model was fit to a 'data.frame' with boolean columns, when importing this model into C++, they need to be encoded in the same order - e.g. the model might encode 'TRUE' as zero and 'FALSE' as one - you need to look at the metadata for this.

The files produced by this function will be compatible between:

- Different operating systems.
- Different compilers.
- Different Python/R versions.
- Systems with different 'size_t' width (e.g. 32-bit and 64-bit), as long as the file was produced on a system that was either 32-bit or 64-bit, and as long as each saved value fits within the range of the machine's 'size_t' type.
- Systems with different 'int' width, as long as the file was produced on a system that was 16-bit, 32-bit, or 64-bit, and as long as each saved value fits within the range of the machine's int type.
- Systems with different bit endianness (e.g. x86 and PPC64 in non-le mode).

- Versions of this package from 0.3.0 onwards, **but only forwards compatible** (e.g. a model saved with versions 0.3.0 to 0.3.5 can be loaded under version 0.3.6, but not the other way around, and attempting to do so will cause crashes and memory corruption without an informative error message). **This last point applies also to models saved through save, saveRDS, qsave, and similar**. Note that loading a model produced by an earlier version of the library might be slightly slower.

But will not be compatible between:

- Systems with different floating point numeric representations (e.g. standard IEEE754 vs. a base-10 system).
- Versions of this package earlier than 0.3.0.

This pretty much guarantees that a given file can be serialized and de-serialized in the same machine in which it was built, regardless of how the library was compiled.

Reading a serialized model that was produced in a platform with different characteristics (e.g. 32-bit vs. 64-bit) will be much slower.

On Windows, if compiling this library with a compiler other than MSVC or MINGW, (not currently supported by CRAN's build systems at the moment of writing) there might be issues exporting models larger than 2GB.

In non-windows systems, if the file name contains non-ascii characters, the file name must be in the system's native encoding. In windows, file names with non-ascii characters are supported as long as the package is compiled with GCC5 or newer.

Note that, while 'readRDS' and 'load' will not make any changes to the serialized format of the objects, reading a serialized model from a file will forcibly re-serialize, using the system's own setup (e.g. 32-bit vs. 64-bit, endianness, etc.), and as such can be used to convert formats.

Value

The same 'model' object that was passed as input, as invisible.

See Also

isotree.import.model isotree.restore.handle

`isotree.get.num.nodes` *Get Number of Nodes per Tree*

Description

Get Number of Nodes per Tree

Usage

```
isotree.get.num.nodes(model)
```

Arguments

`model`	An Isolation Forest model as produced by function ‘isolation.forest‘.

Value

A list with entries ‘"total"‘ and ‘"terminal"‘, both of which are integer vectors with length equal to the number of trees. ‘"total"‘ contains the total number of nodes that each tree has, while ‘"terminal"‘ contains the number of terminal nodes per tree.

`isotree.import.model` *Load an Isolation Forest model exported from Python*

Description

Loads a serialized Isolation Forest model as produced and exported by the Python version of this package. Note that the metadata must be something importable in R - e.g. column names must be valid for R (numbers are valid for Python’s pandas, but not for R, for example).

It’s recommended to generate a ’.metadata’ file (passing ‘add_metada_file=TRUE‘) and to visually inspect said file in any case.

This function is not meant to be used for passing models to and from R - in such case, one can use ‘saveRDS‘ and ‘readRDS‘ instead as they will likely result in smaller file sizes (although this function will still work correctly for serialization within R).

Usage

```
isotree.import.model(file, lazy_serialization = TRUE)
```

Arguments

`file`	Path to the saved isolation forest model. Must be a file path, not a file connection, and the character encoding should correspond to the system’s native encoding.
`lazy_serialization`	Whether to use lazy serialization through the ALTREP system for the resulting objects. See the documentation for this same parameter in isolation.forest for details.

Details

If the model was fit to a ‘DataFrame‘ using Pandas’ own Boolean types, take a look at the metadata to check if these columns will be taken as booleans (R logicals) or as categoricals with string values ‘"True"‘ and ‘"False"‘.

See the documentation for isotree.export.model for details about compatibility of the generated files across different machines and versions.

Value

An isolation forest model, as if it had been constructed through isolation.forest.

See Also

isotree.export.model isotree.restore.handle

`isotree.is.same` *Check if two Isolation Forest Models Share the Same C++ Object*

Description

Checks if two isolation forest models, as produced by functions like isolation.forest, have a reference to the same underlying C++ object.

When this is the case, functions that produce in-place modifications, such as isotree.build.indexer, will produce changes in all of the R variables that share the same C++ object.

Two R variables will have the same C++ object when assigning one variable to another, but will have different C++ objects when these R objects are serialized and deserialized or when calling isotree.deep.copy.

Usage

```
isotree.is.same(obj1, obj2)
```

Arguments

`obj1`	First model to compare (against ‘obj2‘).
`obj2`	Second model to compare (against ‘obj1‘).

Value

A logical (boolean) value which will be ‘TRUE‘ when both models have a reference to the same C++ object, or ‘FALSE‘ otherwise.

See Also

isotree.deep.copy

Examples

```
library(isotree)
data(mtcars)
model <- isolation.forest(mtcars, ntrees = 10, nthreads = 1, ndim = 1)

model_shallow_copy <- model
isotree.is.same(model, model_shallow_copy)

model_deep_copy <- isotree.deep.copy(model)
isotree.is.same(model, model_deep_copy)

isotree.add.tree(model_shallow_copy, mtcars)
```

```
length(isotree.get.num.nodes(model_shallow_copy)$total)
length(isotree.get.num.nodes(model)$total)
length(isotree.get.num.nodes(model_deep_copy)$total)
```

`isotree.plot.tree` *Plot Tree from Isolation Forest Model*

Description

Plots a given tree from an isolation forest model.

Requires the 'DiagrammeR' library to be installed.

Note that this is just a wrapper over isotree.to.graphviz + 'DiagrammeR::grViz'.

Usage

```
isotree.plot.tree(
  model,
  output_tree_num = FALSE,
  tree = 1L,
  column_names = NULL,
  column_names_categ = NULL,
  nthreads = model$nthreads,
  width = NULL,
  height = NULL
)
```

Arguments

`model`	An Isolation Forest object as returned by isolation.forest.
`output_tree_num`	Whether to make the statements / outputs return the terminal node number instead of the isolation depth. The numeration will start at one.
`tree`	Tree for which to generate SQL statements or other outputs. If passed, will generate the statements only for that single tree. If passing 'NULL', will generate statements for all trees in the model.
`column_names`	Column names to use for the **numeric** columns. If not passed and the model was fit to a 'data.frame', will use the column names from that 'data.frame', which can be found under 'model$metadata$cols_num'. If not passing it and the model was fit to data in a format other than 'data.frame', the columns will be named 'column_N' in the resulting SQL statement. Note that the names will be taken verbatim - this function will not do any checks for e.g. whether they constitute valid SQL or not when exporting to SQL, and will not escape characters such as double quotation marks when exporting to SQL.
`column_names_categ`	Column names to use for the **categorical** columns. If not passed, will use the column names from the 'data.frame' to which the model was fit. These can be found under 'model$metadata$cols_cat'.

nthreads	Number of parallel threads to use.
width	Width for the plot, to pass to 'DiagrammeR::grViz'.
height	Height for the plot, to pass to 'DiagrammeR::grViz'.

Details

In general, isolation forest trees tend to be rather large, and the contents on the nodes can be very long when using 'ndim>1' - if the idea is to get easily visualizable trees, one might want to use parameters like 'ndim=1', 'sample_size=256', 'max_depth=8'.

Value

An 'htmlwidget' object that contains the plot.

---

isotree.restore.handle

Unpack isolation forest model after de-serializing

Description

After persisting an isolation forest model object through 'saveRDS', 'save', or restarting a session, the underlying C++ objects that constitute the isolation forest model and which live only on the C++ heap memory are not saved along, and depending on parameter 'lazy_serialization', might not get automatically restored after loading a saved model through 'readRDS' or 'load'.

The model object however keeps serialized versions of the C++ objects as raw bytes, from which the C++ objects can be reconstructed, and are done so automatically upon de-serialization when using 'lazy_serialization=TRUE', but otherwise, the C++ objects will only get de-serialized after calling 'predict', 'print', 'summary', or 'isotree.add.tree' on the freshly-loaded object from 'readRDS' or 'load'.

This function allows to automatically de-serialize the object ("complete" or "restore" the handle) without having to call any function that would do extra processing when one uses 'lazy_serialization=FALSE' (calling the function is **not** needed when using 'lazy_serialization=TRUE').

It is an analog to XGBoost's 'xgb.Booster.complete' and CatBoost's 'catboost.restore_handle' functions.

If the model was built with 'lazy_serialization=TRUE', this function will not do anything to the object.

Usage

```
isotree.restore.handle(model)
```

Arguments

model	An Isolation Forest object as returned by 'isolation.forest', which has been just loaded from a disk file through 'readRDS', 'load', or a session restart, and which was constructed with 'lazy_serialization=FALSE'.

Details

If using this function to de-serialize a model in a production system, one might want to delete the serialized bytes inside the object afterwards in order to free up memory. These are under ‘model$cpp_objects$(model,imputer,indexer)$ser‘ - e.g.: ‘model$cpp_objects$model$ser = NULL; gc()‘.

Value

The same model object that was passed as input. Object is modified in-place however, so it does not need to be re-assigned.

Examples

```
### Warning: this example will generate a temporary .Rds
### file in your temp folder, and will then delete it

### First, create a model from random data
library(isotree)
set.seed(1)
X <- matrix(rnorm(100), nrow = 20)
iso <- isolation.forest(X, ntrees=10, nthreads=1, lazy_serialization=FALSE)

### Now serialize the model
temp_file <- file.path(tempdir(), "iso.Rds")
saveRDS(iso, temp_file)
iso2 <- readRDS(temp_file)
file.remove(temp_file)

cat("Model pointer after loading is this: \n")
print(iso2$cpp_objects$model$ptr)

### now unpack it
isotree.restore.handle(iso2)

cat("Model pointer after unpacking is this: \n")
print(iso2$cpp_objects$model$ptr)

### Note that this function is not needed when using lazy_serialization=TRUE
iso_lazy <- isolation.forest(X, ntrees=10, nthreads=1, lazy_serialization=TRUE)
temp_file_lazy <- file.path(tempdir(), "iso_lazy.Rds")
saveRDS(iso_lazy, temp_file_lazy)
iso_lazy2 <- readRDS(temp_file_lazy)
file.remove(temp_file_lazy)
cat("Model pointer after unpacking lazy-serialized: \n")
print(iso_lazy2$cpp_objects$model$ptr)
```

`isotree.set.nthreads` *Set Number of Threads for Isolation Forest Model Object*

Description

Changes the number of threads that an isolation forest model object will use when calling functions such as ‘predict‘.

Usage

```
isotree.set.nthreads(model, nthreads = 1L)
```

Arguments

`model`	An Isolation Forest model (as returned by function isolation.forest) for which an indexer for terminal node numbers and/or distances will be added. **The object will be modified in-place**.
`nthreads`	Number of threads to set for this model object to use.

Value

The same ‘model‘ object (as invisible), but now with a different configured number of threadst. Note the input object is modified in-place regardless.

`isotree.set.reference.points` *Set Reference Points to Calculate Distances or Kernels With*

Description

Sets some points as pre-defined landmarks with respect to which distances and/or isolation kernel values will be calculated for arbitrary new points in calls to ‘predict‘ with types ‘"dist"‘, ‘"avg_sep"‘, ‘"kernel"‘. If any points have already been set as references in the model object, they will be overwritten with the new points passed here.

Be aware that adding reference points requires building a tree indexer.

Usage

```
isotree.set.reference.points(
  model,
  data,
  with_distances = FALSE,
  nthreads = model$nthreads
)
```

Arguments

`model`	An Isolation Forest model (as returned by function isolation.forest) for which reference points for distance and/or kernel calculations will be set. **The object will be modified in-place**. If there were any previous references, they will be overwritten with the new ones passed here.
`data`	Observations to set as reference points for future distance and/or isolation kernel calculations. Same format as for predict.isolation_forest.
`with_distances`	Whether to pre-calculate node distances (this is required to calculate distance from arbitrary points to the reference points). Note that reference points for distances can only be set when using 'assume_full_distr=FALSE' (which is the default).
`nthreads`	Number of parallel threads to use.

Details

Note that points are added in terms of their terminal node indices, but the raw data about them is not kept - thus, calling isotree.add.tree later on a model with reference points requires passing those reference points again to add their node indices to the new tree.

If using 'lazy_serialization=TRUE', and the process fails while setting references (e.g. due to out-of-memory errors), previous references that the model might have had will be lost.

Value

The same 'model' object (as invisible), but now with added reference points that can be used for new distance and/or kernel calculations with respect to other arbitrary points.

See Also

isotree.build.indexer

`isotree.subset.trees` *Subset trees of a given model*

Description

Creates a new isolation forest model containing only selected trees of a given isolation forest model object. Note that, if using 'lazy_serialization=FALSE', this will re-trigger serialization.

Usage

```
isotree.subset.trees(model, trees_take)

## S3 method for class 'isolation_forest'
x[i]
```

Arguments

`model`, `x`	An ‘isolation_forest‘ model object.
`trees_take`, `i`	Indices of the trees of ‘model‘ to copy over to a new model, as an integer vector. Must be integers with numeration starting at one

Value

A new isolation forest model object, containing only the subset of trees from this ‘model‘ that was specified under ‘trees_take‘.

`isotree.to.graphviz` *Generate GraphViz Dot Representation of Tree*

Description

Generate GraphViz representations of model trees in ’dot’ format - either separately per tree (the default), or for a single tree if needed (if passing ‘tree‘) Can also be made to output terminal node numbers (numeration starting at one).

These can be loaded as graphs through e.g. ‘DiagrammeR::grViz(x)‘, where ‘x‘ would be the output of this function for a given tree.

Graph format is based on XGBoost’s.

Usage

```
isotree.to.graphviz(
  model,
  output_tree_num = FALSE,
  tree = NULL,
  column_names = NULL,
  column_names_categ = NULL,
  nthreads = model$nthreads
)
```

Arguments

`model`	An Isolation Forest object as returned by isolation.forest.
`output_tree_num`	Whether to make the statements / outputs return the terminal node number instead of the isolation depth. The numeration will start at one.
`tree`	Tree for which to generate SQL statements or other outputs. If passed, will generate the statements only for that single tree. If passing ‘NULL‘, will generate statements for all trees in the model.

`column_names` Column names to use for the **numeric** columns. If not passed and the model was fit to a ‘data.frame‘, will use the column names from that ‘data.frame‘, which can be found under ‘model$metadata$cols_num‘. If not passing it and the model was fit to data in a format other than ‘data.frame‘, the columns will be named ‘column_N‘ in the resulting SQL statement. Note that the names will be taken verbatim - this function will not do any checks for e.g. whether they constitute valid SQL or not when exporting to SQL, and will not escape characters such as double quotation marks when exporting to SQL.

`column_names_categ` Column names to use for the **categorical** columns. If not passed, will use the column names from the ‘data.frame‘ to which the model was fit. These can be found under ‘model$metadata$cols_cat‘.

`nthreads` Number of parallel threads to use.

Details

- The generated graphs will not include range penalizations, thus predictions might differ from calls to ‘predict‘ when using ‘penalize_range=TRUE‘.
- The generated graphs will only include handling of missing values when using ‘missing_action="impute"‘. When using the single-variable model with categorical variables + subset splits, the rule buckets might be incomplete due to not including categories that were not present in a given node - this last point can be avoided by using ‘new_categ_action="smallest"‘, ‘new_categ_action="random"‘, or ‘missing_action="impute"‘ (in the latter case will treat them as missing, but the ‘predict‘ function might treat them differently).
- If using ‘scoring_metric="density"‘ or ‘scoring_metric="boxed_ratio"‘ plus ‘output_tree_num=FALSE‘, the outputs will correspond to the logarithm of the density rather than the density.

Value

If passing ‘tree=NULL‘, will return a list with one element per tree in the model, where each element consists of an R character / string with the ’dot’ format representation of the tree. If passing ‘tree‘, the output will be instead a single character / string element with the ’dot’ representation for that tree.

Examples

```
## Not run:
library(isotree)
set.seed(123)
X <- matrix(rnorm(100 * 3), nrow = 100)
model <- isolation.forest(X, ndim=1, max_depth=3, ntrees=2, nthreads=1)
model_as_graphviz <- isotree.to.graphviz(model)

# These can be parsed and plotted with library 'DiagrammeR'
if (require("DiagrammeR")) {
    # first tree
    DiagrammeR::grViz(model_as_graphviz[[1]])

    DiagrammeR::grViz(model_as_graphviz[[1]])
```

```
}

## End(Not run)
```

`isotree.to.json` *Generate JSON representations of model trees*

Description

Generates a JSON representation of either a single tree in the model, or of all the trees in the model.

The JSON for a given tree will consist of a sub-json/list for each node, where nodes are indexed by their number (base-1 indexing) as keys in these JSONs (note that they are strings, not numbers, in order to conform to JSON format).

Nodes will in turn consist of another map/list indicating whether they are terminal nodes or not, their score and terminal node index if terminal, or otherwise the split conditions, nodes to follow when the condition is or isn't met, and other aspects such as imputation values if applicable, acceptable ranges when using range penalizations, fraction of the data that went into the left node if recorded, among others.

Note that the JSON structure will be very different for models that have 'ndim=1' than for models that have 'ndim>1'. In the case of 'ndim=1', the conditions are based on the value of only one variable, but for 'ndim=2', they will consist of a linear combination of different columns (which is expressed as a list of JSONs with one entry per column that goes into the calculation) - for numeric columns for example, these will be expressed in the json by a coefficient for the given column, and a centering that needs to be applied, with the score from that column being added as

$$\text{coef} \times (x - \text{centering})$$

and the imputation value being applied in replacement of this formula in the case of missing values for that column (depending on the model parameters); while in the case of categorical columns, might either have a different coefficient for each possible category ('categ_split_type="subset"'), or a single category that gets a non-zero coefficient while the others get zeros ('categ_split_type="single_categ"').

The JSONs might contain redundant information in order to ease understanding of the model logic - for example, when using 'ndim>1' and 'categ_split_type="single_categ"', the coefficient for the non-chosen categories will always be zero, but is nevertheless added to every node's JSON, even if not needed.

Usage

```
isotree.to.json(
  model,
  output_tree_num = FALSE,
  tree = NULL,
  column_names = NULL,
  column_names_categ = NULL,
  as_str = FALSE,
  nthreads = model$nthreads
)
```

Arguments

`model`	An Isolation Forest object as returned by isolation.forest.
`output_tree_num`	Whether to make the statements / outputs return the terminal node number instead of the isolation depth. The numeration will start at one.
`tree`	Tree for which to generate SQL statements or other outputs. If passed, will generate the statements only for that single tree. If passing ‘NULL‘, will generate statements for all trees in the model.
`column_names`	Column names to use for the **numeric** columns. If not passed and the model was fit to a ‘data.frame‘, will use the column names from that ‘data.frame‘, which can be found under ‘model$metadata$cols_num‘. If not passing it and the model was fit to data in a format other than ‘data.frame‘, the columns will be named ‘column_N‘ in the resulting SQL statement. Note that the names will be taken verbatim - this function will not do any checks for e.g. whether they constitute valid SQL or not when exporting to SQL, and will not escape characters such as double quotation marks when exporting to SQL.
`column_names_categ`	Column names to use for the **categorical** columns. If not passed, will use the column names from the ‘data.frame‘ to which the model was fit. These can be found under ‘model$metadata$cols_cat‘.
`as_str`	Whether to return the result as raw JSON strings (returned as R’s character type) instead of being parsed into R lists (internally, it uses ‘jsonlite::fromJSON‘).
`nthreads`	Number of parallel threads to use.

Details

- If using ‘scoring_metric="density"‘ or ‘scoring_metric="boxed_ratio"‘ plus ‘output_tree_num=FALSE‘, the outputs will correspond to the logarithm of the density rather than the density.

Value

Either a list of lists (when passing ‘as_str=FALSE‘) or a vector of characters (when passing ‘as_str=TRUE‘), or a single such list or character element if passing ‘tree‘.

`isotree.to.sql` *Generate SQL statements from Isolation Forest model*

Description

Generate SQL statements - either separately per tree (the default), for a single tree if needed (if passing ‘tree‘), or for all trees concatenated together (if passing ‘table_from‘). Can also be made to output terminal node numbers (numeration starting at one).

Some important considerations:

- Making predictions through SQL is much less efficient than from the model itself, as each terminal node will have to check all of the conditions that lead to it instead of passing observations down a tree.
- If constructed with the default arguments, the model will not perform any sub-sampling, which can lead to very big trees. If it was fit to a large dataset, the generated SQL might consist of gigabytes of text, and might lay well beyond the character limit of commands accepted by SQL vendors.
- The generated SQL statements will not include range penalizations, thus predictions might differ from calls to ‘predict‘ when using ‘penalize_range=TRUE‘.
- The generated SQL statements will only include handling of missing values when using ‘missing_action="impute"‘. When using the single-variable model with categorical variables + subset splits, the rule buckets might be incomplete due to not including categories that were not present in a given node - this last point can be avoided by using ‘new_categ_action="smallest"‘, ‘new_categ_action="random"‘, or ‘missing_action="impute"‘ (in the latter case will treat them as missing, but the ‘predict‘ function might treat them differently).
- The resulting statements will include all the tree conditions as-is, with no simplification. Thus, there might be lots of redundant conditions in a given terminal node (e.g. "X > 2" and "X > 1", the second of which is redundant).
- If using ‘scoring_metric="density"‘ or ‘scoring_metric="boxed_ratio"‘ plus ‘output_tree_num=FALSE‘, the outputs will correspond to the logarithm of the density rather than the density.

Usage

```
isotree.to.sql(
  model,
  enclose = "doublequotes",
  output_tree_num = FALSE,
  tree = NULL,
  table_from = NULL,
  select_as = "outlier_score",
  column_names = NULL,
  column_names_categ = NULL,
  nthreads = model$nthreads
)
```

Arguments

model	An Isolation Forest object as returned by isolation.forest.
enclose	With which symbols to enclose the column names in the select statement so as to make them SQL compatible in case they include characters like dots. Options are: • ‘"doublequotes"‘, which will enclose them as ‘"column_name"‘ - this will work for e.g. PostgreSQL. • ‘"squarebraces"‘, which will enclose them as ‘[column_name]‘ - this will work for e.g. SQL Server. • ‘"none"‘, which will output the column names as-is (e.g. ‘column_name‘)

output_tree_num
: Whether to make the statements / outputs return the terminal node number instead of the isolation depth. The numeration will start at one.

tree
: Tree for which to generate SQL statements or other outputs. If passed, will generate the statements only for that single tree. If passing 'NULL', will generate statements for all trees in the model.

table_from
: If passing this, will generate a single select statement for the outlier score from all trees, selecting the data from the table name passed here. In this case, will always output the outlier score, regardless of what is passed under 'output_tree_num'.

select_as
: Alias to give to the generated outlier score in the select statement. Ignored when not passing 'table_from'.

column_names
: Column names to use for the **numeric** columns. If not passed and the model was fit to a 'data.frame', will use the column names from that 'data.frame', which can be found under 'model$metadata$cols_num'. If not passing it and the model was fit to data in a format other than 'data.frame', the columns will be named 'column_N' in the resulting SQL statement. Note that the names will be taken verbatim - this function will not do any checks for e.g. whether they constitute valid SQL or not when exporting to SQL, and will not escape characters such as double quotation marks when exporting to SQL.

column_names_categ
: Column names to use for the **categorical** columns. If not passed, will use the column names from the 'data.frame' to which the model was fit. These can be found under 'model$metadata$cols_cat'.

nthreads
: Number of parallel threads to use.

Value

- If passing neither 'tree' nor 'table_from', will return a list of 'character' objects, containing at each entry the SQL statement for the corresponding tree.
- If passing 'tree', will return a single 'character' object with the SQL statement representing that tree.
- If passing 'table_from', will return a single 'character' object with the full SQL select statement for the outlier score, selecting the columns from the table name passed under 'table_from'.

Examples

```
library(isotree)
data(iris)
set.seed(1)
iso <- isolation.forest(iris, ntrees=2, sample_size=16, ndim=1, nthreads=1)
sql_forest <- isotree.to.sql(iso, table_from="my_iris_table")
cat(sql_forest)
```

`length.isolation_forest`

Get Number of Trees in Model

Description

Returns the number of trees in an isolation forest model.

Usage

```
## S3 method for class 'isolation_forest'
length(x)
```

Arguments

x	An isolation forest model, as returned by function isolation.forest.

Value

The number of trees in the model, as an integer.

`predict.isolation_forest`

Predict method for Isolation Forest

Description

Predict method for Isolation Forest

Usage

```
## S3 method for class 'isolation_forest'
predict(
  object,
  newdata,
  type = "score",
  square_mat = ifelse(type == "kernel", TRUE, FALSE),
  refdata = NULL,
  use_reference_points = TRUE,
  nthreads = min(object$nthreads, RhpcBLASctl::get_num_cores()),
  ...
)
```

Arguments

`object`	An Isolation Forest object as returned by isolation.forest.
`newdata`	A ‘data.frame‘, ‘data.table‘, ‘tibble‘, ‘matrix‘, or sparse matrix (from package ‘Matrix‘ or ‘SparseM‘, CSC/dgCMatrix supported for outlierness, distance, kernels; CSR/dgRMatrix supported for outlierness and imputations) for which to predict outlierness, distance, kernels, or imputations of missing values. If ‘newdata‘ is sparse and one wants to obtain the outlier score or average depth or tree numbers, it’s highly recommended to pass it in CSC (‘dgCMatrix‘) format as it will be much faster when the number of trees or rows is large.
`type`	Type of prediction to output. Options are: • ‘"score"‘ for the standardized outlier score - for isolation-based metrics (the default), values closer to 1 indicate more outlierness, while values closer to 0.5 indicate average outlierness, and close to 0 more averageness (harder to isolate). For all scoring metrics, higher values indicate more outlierness. • ‘"avg_depth"‘ for the non-standardized average isolation depth or density or log-density. For ‘scoring_metric="density"‘, will output the geometric mean instead. See the documentation for ‘scoring_metric‘ for more details about the calculations for density-based metrics. For all scoring metrics, higher values indicate less outlierness. • ‘"dist"‘ for approximate pairwise or between-points distances (must pass more than 1 row) - these are standardized in the same way as outlierness, values closer to zero indicate nearer points, closer to one further away points, and closer to 0.5 average distance. To make this computation faster, it is highly recommended to build a node indexer with isotree.build.indexer (with ‘with_distances=TRUE‘) before calling this function. • ‘"avg_sep"‘ for the non-standardized average separation depth. To make this computation faster, it is highly recommended to build a node indexer with isotree.build.indexer (with ‘with_distances=TRUE‘) before calling this function. • ‘"kernel"‘ for pairwise or between-points isolation kernel calculations (also known as proximity matrix), which denotes the fraction of trees in which two observations end up in the same terminal node. This is typically not as good quality as the separation distance, but it’s much faster to calculate, and has other potential uses - for example, this "kernel" can be used as an estimate of the correlations between residuals for a generalized least-squares regression, for which distance might not be as appropriate. Note that building an indexer will not speed up kernel/proximity calculations unless it has reference points. This calculation can be sped up significantly by setting reference points in the model object through isotree.set.reference.points, and it’s highly recommended to do so if this calculation is going to be performed repeatedly. • ‘"kernel_raw"‘ for the isolation kernel or proximity matrix, but having as output the number of trees instead of the fraction of total trees. • ‘"tree_num"‘ for the terminal node number for each tree - if choosing this option, will return a list containing both the average isolation depth and

the terminal node numbers, under entries ‘avg_depth‘ and ‘tree_num‘, respectively. If this calculation is going to be perform frequently, it’s recommended to build node indices through isotree.build.indexer.

- ‘"tree_depths"‘ for the non-standardized isolation depth or expected isolation depth or density or log-density for each tree (note that they will not include range penalties from ‘penalize_range=TRUE‘). See the documentation for ‘scoring_metric‘ for more details about the calculations for density-based metrics.
- ‘"impute"‘ for imputation of missing values in ‘newdata‘.

`square_mat` When passing ‘type‘ = ‘"dist‘ or ‘"avg_sep"‘ or ‘"kernel"‘ or ‘"kernel_raw"‘ with no ‘refdata‘, whether to return a full square matrix (returned as a numeric ‘matrix‘ object) or just its upper-triangular part (returned as a ‘dist‘ object and compatible with functions such as ‘hclust‘), in which the entry for pair (i,j) with 1 <= i < j <= n is located at position p(i, j) = ((i - 1) * (n - i/2) + j - i).

Ignored when not predicting distance/separation/kernels or when passing ‘refdata‘ or ‘use_reference_points=TRUE‘ plus having reference points.

`refdata` If passing this and calculating distances or average separation depths or kernels, will calculate distances between each point in ‘newdata‘ and each point in ‘refdata‘, outputting a matrix in which points in ‘newdata‘ correspond to rows and points in ‘refdata‘ correspond to columns. Must be of the same type as ‘newdata‘ (e.g. ‘data.frame‘, ‘matrix‘, ‘dgCMatrix‘, etc.). If this is not passed, and type is ‘"dist"‘ or ‘"avg_sep"‘ or ‘"kernel"‘ or ‘"kernel_raw"‘, will calculate pairwise distances/separation between the points in ‘newdata‘.

Note that, if ‘refdata‘ is passed and and the model object has an indexer with reference points added (through isotree.set.reference.points), those reference points will be ignored for the calculation.

`use_reference_points`

When the model object has an indexer with reference points (which can be added through isotree.set.reference.points) and passing ‘type="dist"‘ or ‘"avg_sep"‘ or ‘"kernel"‘ or ‘"kernel_raw"‘, whether to calculate the distances/kernels from ‘newdata‘ to those reference points instead of the pairwise distances between points in ‘newdata‘.

This is ignored when passing ‘refdata‘ or when the model object does not contain an indexer or the indexer does not contain reference points.

`nthreads` Number of parallel threads to use. **Note:** for better performance, it’s recommended to set the number of threads to the number of **physical** CPU cores, which in a typical desktop CPU, corresponds to half the number of threads (see details for more information).

Shorthand for best performance: ‘nthreads = RhpcBLASctl::get_num_cores()‘

`...` Not used.

Details

The standardized outlier score for isolation-based metrics is calculated according to the original paper’s formula: $2^{-\frac{\bar{d}}{c(n)}}$, where $\bar{d}$ is the average depth under each tree at which an observation becomes isolated (a remainder is extrapolated if the actual terminal node is not isolated), and $c(n)$

is the expected isolation depth if observations were uniformly random (see references under isolation.forest for details). The actual calculation of $c(n)$ however differs from the paper as this package uses more exact procedures for calculation of harmonic numbers.

For density-based matrics, see the documentation for 'scoring_metric' in isolation.forest for details about the score calculations.

The distribution of outlier scores for isolation-based metrics should be centered around 0.5, unless using non-random splits (parameters 'prob_pick_avg_gain', 'prob_pick_pooled_gain', 'prob_pick_full_gain', 'prob_pick_dens') and/or range penalizations, or having distributions which are too skewed. For 'scoring_metric="density"', most of the values should be negative, and while zero can be used as a natural score threshold, the scores are unlikely to be centered around zero.

The more threads that are set for the model, the higher the memory requirement will be as each thread will allocate an array with one entry per row (outlierness) or combination (distance), with an exception being calculation of distances/kernels to reference points, which do not do this.

For multi-threaded predictions on many rows, it is recommended to set the number of threads to the number of physical cores of the CPU rather than the number of logical cores, as it will typically have better performance that way. Assuming a typical x86-64 desktop CPU, this typically involves dividing the number of threads by 2 - for example: 'model$nthreads <- RhpcBLASctl::get_num_cores()'

Outlierness predictions for sparse data will be much slower than for dense data. Not recommended to pass sparse matrices unless they are too big to fit in memory.

Note that after loading a serialized object from 'isolation.forest' through 'readRDS' or 'load', if it was constructed with 'lazy_serialization=FALSE' it will only de-serialize the underlying C++ object upon running 'predict', 'print', or 'summary', so the first run will be slower, while subsequent runs will be faster as the C++ object will already be in-memory. This does not apply when using 'lazy_serialization=TRUE'.

In order to save memory when fitting and serializing models, the functionality for outputting terminal node numbers will generate index mappings on the fly for all tree nodes, even if passing only 1 row, so it's only recommended for batch predictions. If this type of prediction is desired, it can be sped up by building an index of terminal nodes through isotree.build.indexer, which will avoid having to recompute these every time.

The outlier scores/depth predict functionality is optimized for making predictions on one or a few rows at a time - for making large batches of predictions, it might be faster to use the option 'output_score=TRUE' in 'isolation.forest'.

When making predictions on CSC matrices with many rows using multiple threads, there can be small differences between runs due to roundoff error.

When imputing missing values, the input may contain new columns (i.e. not present when the model was fitted), which will be output as-is.

If passing 'type="dist"' or 'type="avg_sep"', by default, it will do the calculation through a procedure that counts steps as observations are passed down the trees, which is especially slow and not recommended for more than a few thousand observations. If this calculation is going to be called repeatedly and/or it is going to be called for a large number of rows, it's highly recommended to build node distance indexes beforehand through isotree.build.indexer with option 'with_distances=TRUE', as then the computation will be done based on terminal node indices instead, which is a much faster procedure. If distance calculations are all going to be performed with respect to a fixed set of points, it's highly recommended to set those points as references through isotree.set.reference.points.

If using ‘assume_full_distr=FALSE‘ (not recommended to use such option), distance predictions with and without an indexer will differ slightly due to differences in what they count towards "additional" observations in the calculation.

Value

The requested prediction type, which can be:

- A numeric vector with one entry per row in ‘newdata‘ (for output types ‘"score"‘ and ‘"avg_depth"‘).
- An integer matrix with number of rows matching to rows in ‘newdata‘ and number of columns matching to the number of trees in the model, indicating the terminal node number under each tree for each observation, with trees as columns, for output type ‘"tree_num"‘.
- A numeric matrix with rows matching to those in ‘newdata‘ and one column per tree in the model, for output type ‘"tree_depths"‘.
- A numeric square matrix or ‘dist‘ object which consists of a vector with the upper triangular part of a square matrix, (for output types ‘"dist"‘, ‘"avg_sep"‘, ‘"kernel"‘, ‘"kernel_raw"‘; with no ‘refdata‘ and no reference points or ‘use_reference_points=FALSE‘).
- A numeric matrix with points in ‘newdata‘ as rows and points in ‘refdata‘ as columns (for output types ‘"dist"‘, ‘"avg_sep"‘, ‘"kernel"‘, ‘"kernel_raw"‘; with ‘refdata‘).
- A numeric matrix with points in ‘newdata‘ as rows and reference points set through isotree.set.reference.points as columns (for output types ‘"dist"‘, ‘"avg_sep"‘, ‘"kernel"‘, ‘"kernel_raw"‘; with ‘use_reference_points=TRUE‘ and no ‘refdata‘).
- The same type as the input ‘newdata‘ (for output type ‘"impute"‘).

Model serving considerations

If the model is built with ‘nthreads>1‘, the prediction function predict.isolation_forest will use OpenMP for parallelization. In a linux setup, one usually has GNU’s "gomp" as OpenMP as backend, which will hang when used in a forked process - for example, if one tries to call this prediction function from ‘RestRserve‘, which uses process forking for parallelization, it will cause the whole application to freeze. A potential fix in these cases is to pass ‘nthreads=1‘ to ‘predict‘, or to set the number of threads to 1 in the model object (e.g. ‘model$nthreads <- 1L‘ or calling isotree.set.nthreads), or to compile this library without OpenMP (requires manually altering the ‘Makevars‘ file), or to use a non-GNU OpenMP backend (such as LLVM’s ’libomp’. This should not be an issue when using this library normally in e.g. an RStudio session.

The R objects that hold the models contain heap-allocated C++ objects which do not map to R types and which thus do not survive serializations the same way R objects do. In order to make model objects serializable (i.e. usable with ‘save‘, ‘saveRDS‘, and similar), the package offers two mechanisms: (a) a ’lazy_serialization’ option which uses the ALTREP system as a workaround, by defining classes with serialization methods but without datapointer methods (see the docs for ‘lazy_serialization‘ for more info); (b) a more theoretically correct way in which raw bytes are produced alongside the model and from which the C++ objects can be reconstructed. When using the lazy serialization system, C++ objects are restored automatically on load and the serialized bytes then discarded, but this is not the case when using the serialized bytes approach. For model serving, one would usually want to drop these serialized bytes after having loaded a model through ‘readRDS‘ or similar (note that reconstructing the C++ object will first require calling isotree.restore.handle, which is done automatically when calling ‘predict‘ and similar), as they can

increase memory usage by a large amount. These redundant raw bytes can be dropped as follows: ‘model$cpp_objects$model$ser <- NULL‘ (and an additional ‘model$cpp_objects$imputer$ser <- NULL‘ when using ‘build_imputer=TRUE‘, and ‘model$cpp_objects$indexer$ser <- NULL‘ when building a node indexer). After that, one might want to force garbage collection through ‘gc()‘.

Usually, for serving purposes, one wants a setup as minimalistic as possible (e.g. smaller docker images). This library can be made smaller and faster to compile by disabling some features - particularly, the library will by default build with support for calculation of aggregated metrics (such as standard deviations) in ’long double’ precision (an extended precision type), which is a functionality that’s unlikely to get used (default is not to use this type as it is slower, and calculations done in the ‘predict‘ function do not use it for anything). Support for ’long double’ can be disable at compile time by setting up an environment variable ‘NO_LONG_DOUBLE‘ before installing the package (e.g. by issuing command ‘Sys.setenv("NO_LONG_DOUBLE" = "1")‘ before ‘install.packages‘).

See Also

isolation.forest isotree.restore.handle isotree.build.indexer isotree.set.reference.points

---

`print.isolation_forest` *Print summary information from Isolation Forest model*

Description

Displays the most general characteristics of an isolation forest model (same as ‘summary‘).

Usage

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

Arguments

`x`	An Isolation Forest model as produced by function ‘isolation.forest‘.
`...`	Not used.

Details

Note that after loading a serialized object from ‘isolation.forest‘ through ‘readRDS‘ or ‘load‘, when using ‘lazy_serialization=FALSE‘, it will only de-serialize the underlying C++ object upon running ‘predict‘, ‘print‘, or ‘summary‘, so the first run will be slower, while subsequent runs will be faster as the C++ object will already be in-memory. This does not apply when using ‘lazy_serialization=TRUE‘.

Value

The same model that was passed as input.

See Also

isolation.forest

`summary.isolation_forest` *Print summary information from Isolation Forest model*

Description

Displays the most general characteristics of an isolation forest model (same as ‘print‘).

Usage

```
## S3 method for class 'isolation_forest'
summary(object, ...)
```

Arguments

`object`	An Isolation Forest model as produced by function ‘isolation.forest‘.
`...`	Not used.

Details

Note that after loading a serialized object from ‘isolation.forest‘ through ‘readRDS‘ or ‘load‘, when using ‘lazy_serialization=FALSE‘, it will only de-serialize the underlying C++ object upon running ‘predict‘, ‘print‘, or ‘summary‘, so the first run will be slower, while subsequent runs will be faster as the C++ object will already be in-memory. This does not apply when using ‘lazy_serialization=TRUE‘.

Value

No return value.

See Also

isolation.forest

`variable.names.isolation_forest` *Get Variable Names for Isolation Forest Model*

Description

Returns the names of the input data columns / variables to which an isolation forest model was fitted.

If the data did not have column names, it will make them up as "column_1..N".

Note that columns will always be reordered so that numeric columns come first, followed by categorical columns.

Usage

```
## S3 method for class 'isolation_forest'
variable.names(object, ...)
```

Arguments

`object`	An isolation forest model, as returned by function isolation.forest.
`...`	Not used.

Value

A character vector containing the column / variable names.

Index

[.isolation_forest
(isotree.subset.trees), 41

isolation.forest, 3, *26, 27*, *29*, *33*, *35–37*, *40–42*, *45, 46*, *48, 49*, *51*, *53–55*
isotree.add.tree, *21*, 25, *29*, *41*
isotree.append.trees, *12*, 27, *29*
isotree.build.indexer, *16, 17*, 28, *31*, *36*, *41*, *49–51*, *53*
isotree.deep.copy, *26, 27*, 29, *36*
isotree.drop.imputer, 30
isotree.drop.indexer, *29*, 31
isotree.drop.reference.points, 32
isotree.export.model, 32, *35, 36*
isotree.get.num.nodes, 34
isotree.import.model, *34*, 35
isotree.is.same, *29*, 36
isotree.plot.tree, 37
isotree.restore.handle, *20, 21*, *26*, *34*, *36*, 38, *52, 53*
isotree.set.nthreads, *20*, 39, *52*
isotree.set.reference.points, *26*, *31, 32*, 40, *49–53*
isotree.subset.trees, 41
isotree.to.graphviz, *37*, 42
isotree.to.json, 44
isotree.to.sql, 45

length.isolation_forest, 48

predict.isolation_forest, *20, 21*, *41*, 48, *52*
print.isolation_forest, 53

summary.isolation_forest, 54

variable.names.isolation_forest, 55