

Package ‘isotree’

March 1, 2021

Type Package

Title Isolation-Based Outlier Detection

Version 0.2.7

Date 2021-02-28

Author David Cortes

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) <arXiv:1811.02141>), SCiForest (Liu, Ting, Zhou (2010) <doi:10.1007/978-3-642-15883-4_18>), and fair-cut forest (Cortes (2019) <arXiv:1911.06646>), for isolation-based outlier detection, clustered outlier detection, distance or similarity approximation (Cortes (2019) <arXiv:1910.12362>), and imputation of missing values (Cortes (2019) <arXiv:1911.06646>), based on random or guided decision tree splitting. 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

Imports Rcpp (>= 1.0.1)

Suggests MASS, outlierTree, jsonlite, readr

Enhances Matrix, SparseM

LinkingTo Rcpp, Rcereal

RoxygenNote 7.1.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2021-03-01 19:20:02 UTC

R topics documented:

add.isolation.tree	2
append.trees	3
deepcopy.isotree	5
export.isotree.model	5
get.num.nodes	7
isolation.forest	7
isotree.to.sql	20
load.isotree.model	22
predict.isolation_forest	23
print.isolation_forest	26
summary.isolation_forest	26
unpack.isolation.forest	27

Index	29
--------------	-----------

add.isolation.tree	<i>Add additional (single) tree to isolation forest model</i>
--------------------	---

Description

Adds a single tree fit to the full (non-subsampled) data passed here. Must have the same columns as previously-fitted data.

Usage

```
add.isolation.tree(model, df, sample_weights = NULL, column_weights = NULL)
```

Arguments

model	An Isolation Forest object as returned by ‘isolation.forest’, to which an additional tree will be added. The result of this function must be reassigned to ‘model’, and the old ‘model’ should not be used any further.
df	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 ‘df’. Ignored when picking columns by deterministic criterion. If passing ‘NULL’, each column will have a uniform weight. Cannot be used when weighting by kurtosis.

Details

Important: this function will modify the model object in-place, but this modification will only affect the R object in the environment in which it was called. If trying to use the same model object in e.g. its parent environment, it will lead to issues due to the C++ object being modified but the R object remaining the same, so if this method is used inside a function, make sure to output the newly-modified R object and have it replace the old R object outside the calling function too.

The model object can be deep copied (including the underlying C++ object) through function [deep-copy.isotree](#).

Value

No return value. The model is modified in-place.

See Also

[isolation.forest](#) [unpack.isolation.forest](#)

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](#) (which is **not** the default), 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 it.

Also be aware that the result **must** be reassigned to the first input, as the first input will no longer work correctly after appending more trees to it.

Important: the result of this function must be reassigned to 'model' in order for it to work properly - e.g. 'model <- append.trees(model, other)'.

Usage

```
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. The result of this function must be reassigned to 'model', and the old 'model' should not be used any further.
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

Important: this function will modify the model object in-place, but this modification will only affect the R object in the environment in which it was called. If trying to use the same model object in e.g. its parent environment, it will lead to issues due to the C++ object being modified but the R object remaining the same, so if this method is used inside a function, make sure to output the newly-modified R object and have it replace the old R object outside the calling function too.

The model object can be deep copied (including the underlying C++ object) through function [deep-copy.isotree](#).

Value

The updated 'model' object, to which 'model' needs to be reassigned (i.e. you need to use it as follows: 'model <- append.trees(model, other)').

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, nthreads=1)
iso2 <- isolation.forest(X2, ntrees=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")

### Append the trees from 'iso2' into 'iso1'
iso1 <- 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$tree_num == cbind(nodes1$tree_num, nodes2$tree_num)

### The new predicted scores will be a weighted average
```

```
### (Be aware that, due to round-off, it will not match with '==')
nodes.comb$avg_depth
(3*nodes1$avg_depth + 2*nodes2$avg_depth) / 5
```

deepcopy.isotree *Deep-Copy an Isolation Forest Model Object*

Description

Deep-Copy an Isolation Forest Model Object

Usage

```
deepcopy.isotree(model)
```

Arguments

model An 'isolation_forest' model object.

Details

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 [add.isolation.tree](#) and [append.trees](#) - as otherwise R's objects follow a copy-on-write logic.

Value

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

export.isotree.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).

It is recommended to visually inspect the produced '.metadata' file in any case.

Usage

```
export.isotree.model(model, file, ...)
```

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
...	Additional arguments to pass to writeBin - you might want to pass extra parameters if passing files between different CPU architectures or similar.

Details

This function will create 2 files: the serialized model, in binary format, with the name passed in 'file'; and a metadata file in JSON format with the same name but ending in '.metadata'. The second file should **NOT** be edited manually, except for the field 'nthreads' if desired.

If the model was built with 'build_imputer=TRUE', there will also be a third binary file ending in '.imputer'.

The metadata 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. This metadata is written to a JSON file using the 'jsonlite' package, which must be installed in order for this to work.

The serialized file can be used in the C++ version by reading it as a binary raw file and de-serializing its contents with the 'cereal' library or using the provided C++ functions for de-serialization. If using 'ndim=1', it will be an object of class 'IsoForest', and if using 'ndim>1', will be an object of class 'ExtIsoForest'. The imputer file, if produced, will be an object of class 'Imputer'.

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 Python version.

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.

Value

No return value.

References

<https://uscilab.github.io/cereal/>

See Also

[load.isotree.model](#) [writeBin](#) [unpack.isolation.forest](#)

get.num.nodes	<i>Get Number of Nodes per Tree</i>
---------------	-------------------------------------

Description

Get Number of Nodes per Tree

Usage

```
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.

isolation.forest	<i>Create Isolation Forest Model</i>
------------------	--------------------------------------

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 model that aim at isolating outliers faster and 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. 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 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. 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 the training data ('df') at the expense of hindered performance when making predictions on new data (calling function 'predict') and 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'.

Usage

```
isolation.forest(
  df,
  sample_size = NROW(df),
  ntrees = 500,
  ndim = min(3, NCOL(df)),
  ntry = 3,
  categ_cols = NULL,
  max_depth = ceiling(log2(sample_size)),
  ncols_per_tree = NCOL(df),
  prob_pick_avg_gain = 0,
  prob_pick_pooled_gain = 0,
  prob_split_avg_gain = 0,
  prob_split_pooled_gain = 0,
  min_gain = 0,
  missing_action = ifelse(ndim > 1, "impute", "divide"),
  new_categ_action = ifelse(ndim > 1, "impute", "weighted"),
  categ_split_type = "subset",
  all_perm = FALSE,
  coef_by_prop = FALSE,
  recode_categ = TRUE,
  weights_as_sample_prob = TRUE,
  sample_with_replacement = FALSE,
  penalize_range = FALSE,
  weigh_by_kurtosis = FALSE,
  coefs = "normal",
  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,
random_seed = 1,
nthreads = parallel::detectCores()
)

```

Arguments

- df** 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’.
- 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(df)’ (same as in here).
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.
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.
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 [3]).
Note that, when using ‘ndim>1’, the variables are standardized at each step as suggested in [4], which makes the models slightly different than in [3].

ntry	In the extended model with non-random splits, how many random combinations to try for determining the best gain. Only used when deciding splits by gain (see documentation for parameters 'prob_pick_avg_gain' and 'prob_pick_pooled_gain'). Recommended value in reference [4] is 10. Ignored for single-variable model.
categ_cols	Columns that hold categorical features (should be an integer vector), when the data is passed as a matrix. Categorical columns should contain only integer values with a continuous numeration starting at zero, with negative values and NaN taken as missing, and the vector passed here should correspond to the column numbers, with numeration starting at one. 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 'df' 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.
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 weighting by kurtosis. 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.
prob_pick_avg_gain	<ul style="list-style-type: none"> • For the single-variable model ('ndim=1'), this parameter indicates the probability of making each split by choosing a column and split point in that same column as both the column and split point that gives the largest averaged gain (as proposed in [4]) across all available columns and possible splits in each column. Note that this implies evaluating every single column in the sample data when this type of split happens, which will potentially make the model fitting much slower, but has no impact on prediction time. For categorical variables, 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 '$\sim \text{Unif}(0, 1)$' and calculate gain with those assumed standard deviations. • For the extended model, this parameter indicates the probability that the split point in the chosen linear combination of variables will be decided by this averaged gain criterion. <p>Compared to a pooled average, this tends to result in more cases in which a single observation or very few of them are put into one branch. Recommended to use sub-samples (parameter 'sample_size') when passing this pa-</p>

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_split_avg_gain', 'prob_split_pooled_gain', both the column and the split point are decided at random. Default setting for references [1], [2], [3] is zero, and default for reference [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.

Note 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.

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

prob_pick_pooled_gain

- For the single-variable model ('ndim=1'), this parameter indicates the probability of making each split by choosing a column and split point in that same column as both the column and split point that gives the largest pooled gain (as used in decision tree classifiers such as C4.5 in [7]) across all available columns and possible splits in each column. Note that this implies evaluating every single column in the sample data when this type of split happens, which will potentially make the model fitting much slower, but has no impact on prediction time. For categorical variables, will use shannon entropy instead (like in [7]).
- For the extended model, this parameter indicates the probability that the split point in the chosen linear combination of variables will be decided by this pooled gain criterion.

Compared to a simple average, 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, higher values of this parameter result in models that are able to better flag outliers in the training data, but generalize poorly to outliers in new data and to values of variables outside of the ranges from the training data. Passing small 'sample_size' and high values of this parameter will tend to flag too many outliers.

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 (use more memory). When splits are not made according to any of 'prob_pick_avg_gain', 'prob_pick_pooled_gain', 'prob_split_avg_gain', 'prob_split_pooled_gain', 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'.

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

prob_split_avg_gain

Probability of making each split by selecting a column at random and determining the split point as that which gives the highest averaged gain. Not supported

for the extended model as the splits are on linear combinations of variables. See the documentation for parameter `'prob_pick_avg_gain'` for more details.

`prob_split_pooled_gain`

Probability of making each split by selecting a column at random and determining the split point as that which gives the highest pooled gain. Not supported for the extended model as the splits are on linear combinations of variables. See the documentation for parameter `'prob_pick_pooled_gain'` for more details.

`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 gain criterion (either pooled or averaged). If the highest possible gain in the evaluated splits at a node is below this threshold, that node becomes a terminal node.

`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.
- `"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).
- `"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.

`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.
- `"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.

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'` (regular 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 regular 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 [append.trees](#).
- `weights_as_sample_prob` If passing `'sample_weights'` argument, 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). Note that sampling weight is only used when sub-sampling data for each tree, which is not the default in this implementation.
- `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.
- `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.
- 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)

<code>weigh_by_kurtosis</code>	<p>Whether to weigh each column according to the kurtosis obtained in the sub-sample 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, so if not using sub-samples, it's better to pass column weights calculated externally. For categorical columns, will calculate expected kurtosis if the column was converted to numerical by assigning to each category a random number '$\sim \text{Unif}(0, 1)$'.</p> <p>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.</p> <p>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.</p>
<code>coefs</code>	<p>For the extended model, whether to sample random coefficients according to a normal distribution '$\sim N(0, 1)$' (as proposed in reference [3]) or according to a uniform distribution '$\sim \text{Unif}(-1, +1)$' as proposed in reference [4]. Ignored for the single-variable model. Note that, for categorical variables, the coefficients will be sampled $\sim N(0,1)$ 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 <code>"uniform"</code> here.</p>
<code>assume_full_distr</code>	<p>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 <code>'FALSE'</code>, 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 <code>'output_dist'</code>).</p>
<code>build_imputer</code>	<p>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 <code>'missing_action'</code> as missing values inside the model are treated differently and follow their own imputation or division strategy.</p>
<code>output_imputations</code>	<p>Whether to output imputed missing values for <code>'df'</code>. Passing <code>'TRUE'</code> here will force <code>'build_imputer'</code> to <code>'TRUE'</code>. 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.</p>
<code>min_imp_obs</code>	<p>Minimum number of observations with which an imputation value can be produced. Ignored if passing <code>'build_imputer' = 'FALSE'</code>.</p>
<code>depth_imp</code>	<p>How to weight observations according to their depth when used for imputing missing values. Passing <code>"higher"</code> will weigh observations higher the further</p>

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 outlieriness 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.
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.
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 $1 \leq i < j \leq n$ is located at position $p(i, j) = ((i - 1) * (n - i/2) + j - i)$.
sample_weights	Sample observation weights for each row of 'df', 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').
column_weights	Sampling weights for each column in 'df'. Ignored when picking columns by deterministic criterion. If passing 'NULL', each column will have a uniform weight. Cannot be used when weighting by kurtosis. 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 'df', will assume the column order is the same as in there, regardless of whether the entries passed to 'column_weights' are named or not.
random_seed	Seed that will be used for random number generation.
nthreads	Number of parallel threads to use. If passing a negative number, will use the maximum number of available threads in the system. 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

When using more than one dimension for splits (i.e. splitting hyperplanes, see `'ndim'`) and when calculating gain, the variables are standardized at each step, so there is no need to center/scale the data beforehand. The gain calculations are also standardized according to the standard deviation when using `'ntry>1'` or `'ndim==1'`, in order to avoid differences in the magnitudes of the coefficients.

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 input observation (if passing `'output_score' = 'TRUE'`).
- `'dist'`: the distances (either a 1-d vector with the upper-triangular part 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'`).

References

- Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation forest." 2008 Eighth IEEE International Conference on Data Mining. IEEE, 2008.
- 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.
- Hariri, Sahand, Matias Carrasco Kind, and Robert J. Brunner. "Extended Isolation Forest." arXiv preprint arXiv:1811.02141 (2018).
- 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.
- <https://sourceforge.net/projects/iforest/>
- <https://math.stackexchange.com/questions/3388518/expected-number-of-paths-required-to-separate>
- Quinlan, J. Ross. "C4. 5: programs for machine learning." Elsevier, 2014.
- Cortes, David. "Distance approximation using Isolation Forests." arXiv preprint arXiv:1910.12362 (2019).
- Cortes, David. "Imputing missing values with unsupervised random trees." arXiv preprint arXiv:1911.06646 (2019).
- <https://math.stackexchange.com/questions/3333220/expected-average-depth-in-random-binary-tree->

See Also

[predict.isolation_forest](#), [add.isolation.tree](#) [unpack.isolation.forest](#)

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 and see what happens,
### or check how its score changes when using sub-sampling)
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 ouy 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,
  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)

```

```

### 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, ntrees=100, nthreads=1)

### Calculate distances with the model
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 <- matrix(rnorm(5 * 5), nrow = 5)
  S <- t(S) %*% S
  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,
    ntry = 10)
  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 (c1 in 1:5)
  X_imp_mean[values_NA[,c1], c1] <- X_means[c1]
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 outlierstree)

### 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")) {
  hypothroid <- outlierstree::hypothyroid

  iso <- isolation.forest(hypothyroid, nthreads=1)
  pred_iso <- predict(iso, hypothroid)
  otree <- outlierstree::outlier.tree(
    hypothroid,
    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.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).

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
)
```

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: <ul style="list-style-type: none"> • "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 return the terminal node number instead of the isolation depth. The numeration will start at one.
tree	Tree for which to generate SQL statements. 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 whether they constitute valid SQL or not, and will not escape characters such as double quotation marks.
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'.

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)
```

load.isotree.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 not valid names for R). It's recommended to visually inspect the '.metadata' file in any case.

This function is not meant to be used for passing models to and from R - in such case, you can use 'saveRDS' and 'readRDS' instead.

Usage

```
load.isotree.model(file)
```

Arguments

file Path to the saved isolation forest model along with its metadata file, and imputer file if produced. Must be a file path, not a file connection.

Details

Internally, this function uses `'readr::read_file_raw'` (from the `'readr'` package) and `'jsonlite::fromJSON'` (from the `'jsonlite'` package). Be sure to have those installed and that the files are readable through them.

Note: 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"` or `"False"`.

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 `'modelcpp_objserialized'` (plus `'modelcpp_objimp_ser'` if building with imputer) - e.g.: `'modelcpp_objserialized = NULL; modelcpp_objimp_ser = NULL; gc()'`.

Value

An isolation forest model, as if it had been constructed through [isolation.forest](#).

See Also

[export.isotree.model](#) [unpack.isolation.forest](#)

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 = FALSE,
  refdata = NULL,
  ...
)
```

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 format for distance and outlieriness, or CSR/dgRMatrix format for outlieriness and imputations) for which to predict outlieriness, distance, or imputations of missing values. Note that when passing 'type' = "impute" and 'newdata' is a sparse matrix, under some situations it might get modified in-place. Note also that, if using sparse matrices from package 'Matrix', converting to 'dgRMatrix' (when required) might require using 'as(m, "RsparseMatrix")' instead of 'dgRMatrix' directly. Nevertheless, 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: <ul style="list-style-type: none"> • "score" for the standardized outlier score, where values closer to 1 indicate more outlieriness, while values closer to 0.5 indicate average outlieriness, and close to 0 more averageness (harder to isolate). • "avg_depth" for the non-standardized average isolation depth. • "dist" for approximate pairwise or between-points distances (must pass more than 1 row) - these are standardized in the same way as outlieriness, values closer to zero indicate nearer points, closer to one further away points, and closer to 0.5 average distance. • "avg_sep" for the non-standardized average separation depth. • "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. • "impute" for imputation of missing values in 'newdata'.
square_mat	When passing 'type' = "dist" or "avg_sep" with no 'refdata', whether to return a full square matrix or just the upper-triangular part, in which the entry for pair (i,j) with $1 \leq i < j \leq n$ is located at position $p(i, j) = ((i - 1) * (n - i/2) + j - i)$. Ignored when not predicting distance/separation or when passing 'refdata'.
refdata	If passing this and calculating distance or average separation depth, 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", will calculate pairwise distances/separation between the points in 'newdata'.
...	Not used.

Details

The standardized outlier score 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.

The distribution of outlier scores should be centered around 0.5, unless using non-random splits (parameters `'prob_pick_avg_gain'`, `'prob_pick_pooled_gain'`, `'prob_split_avg_gain'`, `'prob_split_pooled_gain'`) and/or range penalizations (which are on by default).

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).

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'`, 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.

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.

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 imputing missing values, if the input data is a `'data.frame'` and the model was fit to a `'data.frame'`, the input may contain new columns (i.e. not present when the model was fitted), which will be output as-is. If it is a matrix or sparse matrix, should not contain any new columns.

Value

The requested prediction type, which can be:

- A numeric vector with one entry per row in `'newdata'` (for output types `'score'`, `'avg_depth'`).
- A list with entries `'avg_depth'` (numeric vector) and `'tree_num'` (integer matrix indicating the terminal node number under each tree for each observation, with trees as columns), for output type `'tree_num'`.
- A numeric square matrix or vector with the upper triangular part of a square matrix (for output types `'dist'`, `'avg_sep'`, with no `'refdata'`).
- A numeric matrix with points in `'newdata'` as rows and points in `'refdata'` as columns (for output types `'dist'`, `'avg_sep'`, with `'refdata'`).
- The same type as the input `'newdata'` (for output type `'impute'`).

See Also

[isolation.forest](#) [unpack.isolation.forest](#)

```
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', 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.

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', 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.

Value

No return value.

See Also

[isolation.forest](#)

unpack.isolation.forest

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, thus not 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 after calling 'predict', 'print', 'summary', or 'add.isolation.tree' on the freshly-loaded object from 'readRDS' or 'load'.

But due to R's environments system (as opposed to other systems such as Python which can use pass-by-reference), they will only be re-constructed in the environment that is calling 'predict', 'print', etc. and not in higher-up environments (i.e. if calling 'predict' on the object from inside different functions, each function will have to reconstruct the C++ objects independently and they will only live within the function that called 'predict').

This function serves as an environment-level unpacker that will reconstruct the C++ object in the environment in which it is called (i.e. if it's desired to call 'predict' from inside multiple functions, use this function before passing the freshly-loaded model object to those other functions, and then they will not need to reconstruct the C++ objects anymore), in the same way as 'predict' or 'print', but without producing any outputs or messages.

It is an equivalent to XGBoost's 'xgb.Booster.complete' function.

Usage

```
unpack.isolation.forest(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.

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_obj\$serialized’ (plus ‘model\$cpp_obj\$imp_ser’ if building with imputer) - e.g.: ‘model\$cpp_obj\$serialized = NULL; model\$cpp_obj\$imp_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
library(isotree)
set.seed(1)
X <- matrix(rnorm(100), nrow = 20)
iso <- isolation.forest(X, ntrees=10, nthreads=1)
temp_file <- file.path(tempdir(), "iso.Rds")
saveRDS(iso, temp_file)
iso2 <- readRDS(temp_file)
file.remove(temp_file)

### will de-serialize inside, but object is short-lived
wrap_predict <- function(model, data) {
  pred <- predict(model, data)
  cat("pointer inside function is this: ")
  print(model$cpp_obj$ptr)
  return(pred)
}
temp <- wrap_predict(iso2, X)
cat("pointer outside function is this: \n")
print(iso2$cpp_obj$ptr) ### pointer to the C++ object

### now unpack the C++ object beforehand
unpack.isolation.forest(iso2)
print("after unpacking beforehand")
temp <- wrap_predict(iso2, X)
cat("pointer outside function is this: \n")
print(iso2$cpp_obj$ptr)
```

Index

`add.isolation.tree`, [2](#), [5](#), [16](#)
`append.trees`, [3](#), [5](#), [13](#)
`deepcopy.isotree`, [3](#), [4](#), [5](#)
`export.isotree.model`, [5](#), [23](#)
`get.num.nodes`, [7](#)
`isolation.forest`, [3](#), [4](#), [6](#), [7](#), [21](#), [23](#), [25–27](#)
`isotree.to.sql`, [20](#)
`load.isotree.model`, [6](#), [22](#)
`predict.isolation_forest`, [16](#), [23](#)
`print.isolation_forest`, [26](#)
`summary.isolation_forest`, [26](#)
`unpack.isolation.forest`, [3](#), [6](#), [16](#), [23](#), [25](#),
[27](#)
`writeBin`, [6](#)