isotree Documentation

David Cortes

May 13, 2022

CONTENTS

This is the documentation page for the *isotree* Python package. See project webpage for more details:

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

For the R version, see the CRAN webpage:

https://cran.r-project.org/web/packages/isotree/index.html

CHAPTER ONE

QUICK EXAMPLE NOTEBOOKS

- General library usage.
- As missing value imputer.
- Converting to treelite.

CHAPTER TWO

METHODS

- *IsolationForest*
- *append_trees*
- *copy*
- *decision_function*
- *export_model*
- *fit*
- *fit_predict*
- *fit_transform*
- *generate_sql*
- *get_num_nodes*
- *get_params*
- *import_model*
- *partial_fit*
- *predict*
- *predict_distance*
- *set_params*
- *subset_trees*
- *to_treelite*
- *transform*

CHAPTER THREE

ISOLATIONFOREST

class isotree.**IsolationForest**(*sample_size='auto'*, *ntrees=500*, *ndim=3*, *ntry=1*, *categ_cols=None*, *max_depth='auto'*, *ncols_per_tree=None*, *prob_pick_pooled_gain=0.0*, *prob_pick_avg_gain=0.0*, *min_gain=0.0*, *missing_action='auto'*, *new_categ_action='auto'*, *categ_split_type='subset'*, *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*, *weigh_by_kurtosis=False*, *coefs='normal'*, *assume_full_distr=True*, *build_imputer=False*, *min_imp_obs=3*, *depth_imp='higher'*, *weigh_imp_rows='inverse'*, *random_seed=1*, *nthreads=-1*, *n_estimators=None*, *max_samples=None*, *n_jobs=None*, *random_state=None*, *bootstrap=None*)

Bases: `object`

Isolation Forest model

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 aproximate 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.

Note: 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: 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).

[1] Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation forest." 2008 Eighth IEEE International Conference on Data Mining. IEEE, 2008.
[3] Hariri, Sahand, Matias Carrasco Kind, and Robert J. Brunner. "Extended Isolation Forest." arXiv preprint arXiv:1811.02141 (2018).

'SCiForest' (reference[Page 8, 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=None` despite the reference's recommendation.

'FCF' (reference[11]**):** `ndim=2`, `sample_size=256`, `max_depth=None`, `ntrees=200`, `missing_action="fail"`, `coefs="normal"`, `ntry=1`, `prob_pick_pooled_gain=1`. Might provide similar or better results with `ndim=1`. For the FCF model aimed at imputing missing values, might give better results with `ntry=10` or higher and much larger sample sizes.

Note: 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 in GitHub notebook 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=None` can offer a better speed/performance trade-off than changing `max_depth`.

Note: 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 the documentation of the parameters for more 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.

Note: See the documentation of `predict` for some considerations when serving models generated through this library.

Parameters

- **sample_size** (*str "auto", int, float(0,1), or None*) – Sample size of the data sub-samples with which each binary tree will be built. If passing 'None', each tree will be built using the full data. Recommended value in[Page 7, 1],[2],[?] is 256, while the default value in the author's code in[5] is 'None' here.

  If passing "auto", will use the full number of rows in the data, up to 10,000 (i.e. will take 'sample_size=min(nrows(X), 10000)') **when calling fit**, and the full amount of rows in the data **when calling the variants** `fit_predict` or `fit_transform`.

  If passing `None`, will take the full number of rows in the data (no sub-sampling).

[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.

[11] Cortes, David. "Revisiting randomized choices in isolation forests." arXiv preprint arXiv:2110.13402 (2021).

[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.

[5] https://sourceforge.net/projects/iforest/

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** (*int*) – Number of binary trees to build for the model. Recommended value in[?] is 100, while the default value in the author's code in[Page 8, 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.

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

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

- **ntry** (*int*) – When using `prob_pick_pooled_gain` and/or `prob_pick_avg_gain`, 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[?] is 10 (with `prob_pick_avg_gain`, for outlier detection), while the recommended value in reference[?] 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** (*None or array-like*) – Columns that hold categorical features, when the data is passed as an array or 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 array or list passed here should correspond to the column numbers, with numeration starting at zero. The maximum categorical value should not exceed 'INT_MAX' (typically $2^{31} - 1$). This might be passed either at construction time or when calling `fit` or variations of `fit`.

  This is ignored when the input is passed as a `DataFrame` as then it will consider columns as categorical depending on their dtype (see the documentation for `fit` for details).

- **max_depth** (*int, None, or str "auto"*) – Maximum depth of the binary trees to grow. If passing None, will build trees until each observation ends alone in a terminal node or until no further split is possible. If using "auto", 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[6]). Default setting for[?],[?],[?],[?] is "auto",

[9] Cortes, David. "Imputing missing values with unsupervised random trees." arXiv preprint arXiv:1911.06646 (2019).

[6] https://math.stackexchange.com/questions/3388518/expected-number-of-paths-required-to-separate-elements-in-a-binary-tree

but it's recommended to pass higher values if using the model for purposes other than outlier detection.

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** (*None, int, or float(0,1]*) – 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. If passing exactly 1, will assume it means taking 100% of the columns rather than taking 1 column.

  If passing `None` (the default) or zero, will use the full number of available columns.

- **prob_pick_pooled_gain** (*float(0, 1)*) – 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 a pooled standard deviation gain criterion (see references[Page 9, 9] and[?]) 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` or `prob_pick_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`.

  Be aware also 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`. Alternatively, one can also control the depth through `min_gain` (for which one might want to set `max_depth=None`).

  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_avg_gain** (*float(0, 1)*) – 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[?] and[?]) on the same variable or linear combination.

[7] Quinlan, J. Ross. C4. 5: programs for machine learning. Elsevier, 2014.

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` or `prob_pick_pooled_gain`, both the column and the split point are decided at random. Default setting for[?],[?],[?] is zero, and default for[?] is 1. This is the randomization parameter that can be passed to the author's original code in[?], but note that the code in[?] 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` or `prob_pick_pooled_gain`, the distribution of outlier scores is unlikely to be centered around 0.5.

- **min_gain** (*float > 0*) – 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.

  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** (*str, one of "divide" (single-variable only), "impute", "fail", "auto"*) – How to handle missing data at both fitting and prediction time. Options are:

  `"divide"`: (For the single-variable model only, recommended) 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"`: 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 the extended model).

  `"fail"`: Will assume there are no missing values and will trigger undefined behavior if it encounters any.

  `"auto"`: Will use "divide" for the single-variable model and "impute" for the extended model.

  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[?],[?],[?],[?] correspond to "fail" here.

- **new_categ_action** (*str, one of "weighted" (single-variable only), "impute" (extended only), "smallest", "random"*) – 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) 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) Will assign them the median value for that column that was added to the linear combination of features.

"smallest": 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": Will assing 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 `DataFrame`, when calling `predict` on new data which contains new categories (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. It also means that the `predict` function will not be thread-safe (e.g. cannot be used alongside `joblib` with a backend that uses shared memory).

"auto": Will select "weighted" for the single-variable model and "impute" for the extended model.

Ignored when passing 'categ_split_type' = 'single_categ'.

- **categ_split_type** (*str, one of "subset" or "single_categ"*) – Whether to split categorical features by assigning sub-sets of them to each branch, or by assigning a single category to a branch and the rest to the other branch. For the extended model, whether to give each category a coefficient, or only one while the rest get zero.
- **all_perm** (*bool*) – 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** (*bool*) – 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** (*bool*) – Whether to re-encode categorical variables even in case they are already passed as `pd.Categorical`. 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 `pd.Categorical` 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** (*bool*) – 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). 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** (*bool*) – Whether to sample rows with replacement or not (not recommended). Note that distance calculations, if desired, don't work well with duplicate rows.
- **penalize_range** (*bool*) – 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[?] and implemented in the authors' original code in[?]. 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)
- **standardize_data** (*bool*) – Whether to standardize the features at each node before creating alinear combination of them as suggested in[?]. This is ignored when using `ndim=1`.
- **weigh_by_kurtosis** (*bool*) – Whether to weigh each column according to the kurtosis obtained in the sub-sample that is selected for each tree as briefly proposed in[?]. 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 ~ 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.
- **coefs** (*str, one of "normal" or "uniform"*) – For the extended model, whether to sample random coefficients according to a normal distribution ~ N(0, 1) (as proposed in[?]) or according to a uniform distribution ~ Unif(-1, +1) as proposed in[?]. 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** (*bool*) – When calculating pairwise distances (see[8]), whether to assume that the fitted model represents a full population distribution (will use a standardizing criterion assuming infinite sample, and the results of the similarity between two points at prediction time will not depend on the prescence 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.

[8] Cortes, David. "Distance approximation using Isolation Forests." arXiv preprint arXiv:1910.12362 (2019).

This will not be assumed when the distances are calculated as the model is being fit (see documentation for method 'fit_transform').

- **build_imputer** (*bool*) – 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.
- **min_imp_obs** (*int*) – Minimum number of observations with which an imputation value can be produced. Ignored if passing 'build_imputer' = 'False'.
- **depth_imp** (*str, one of "higher", "lower", "same"*) – 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** (*str, one of "inverse", "prop", "flat"*) – 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 "proportional" 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'.
- **random_seed** (*int*) – Seed that will be used for random number generation.
- **nthreads** (*int*) – Number of parallel threads to use. If passing a negative number, will use the same formula as joblib does for calculating number of threads (which is n_cpus + 1 + n_jobs - i.e. pass -1 to use all available threads). 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.
- **n_estimators** (*None or int*) – Synonym for `ntrees`, kept for better compatibility with scikit-learn.
- **max_samples** (*None or int*) – Synonym for `sample_size`, kept for better compatibility with scikit-learn.
- **n_jobs** (*None or int*) – Synonym for `nthreads`, kept for better compatibility with scikit-learn.
- **random_state** (*None, int, or RandomState*) – Synonym for `random_seed`, kept for better compatibility with scikit-learn.
- **bootstrap** (*None or bool*) – Synonym for `sample_with_replacement`, kept for better compatibility with scikit-learn.

Variables

- **`cols_numeric`** (*`array(n_num_features,)`*) – Array with the names of the columns that were taken as numerical (Only when fitting the model to a DataFrame object).
- **`cols_categ`** (*`array(n_categ_features,)`*) – Array with the names of the columns that were taken as categorical (Only when fitting the model to a DataFrame object).
- **`is_fitted`** (*`bool`*) – Indicator telling whether the model has been fit to data or not.

References

append_trees(*other*)

Appends isolation trees from another Isolation Forest model into this one

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 `np.array` and `scipy.sparse.csc_matrix`).

If the data has categorical variables, the models should have been built with parameter `recode_categ=False` in the class constructor, and the categorical columns passed as type `pd.Categorical` with the same encoding - 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: 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 Python process when using it.

Note: This function is not thread-safe - that is, it will produce problems if one tries to call this function on the same model object in parallel through e.g. `joblib` with a shared-memory backend (which is not the default for joblib).

Parameters **other** (*IsolationForest*) – Another Isolation Forest model from which trees will be appended to this model. It will not be modified during the call to this function.

Returns **self** – This object.

Return type obj

copy()

Get a deep copy of this object

Returns **copied** – A deep copy of this object

Return type obj

decision_function(*X*)

Wrapper for 'predict' with 'output=score'

This function is kept for compatibility with SciKit-Learn.

Parameters **X** (*array or array-like (n_samples, n_features)*) – Observations for which to predict outlierness or average isolation depth. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSC or CSR matrix.

Returns **score** – Outlier scores for the rows in 'X' (the higher, the most anomalous).

Return type array(n_samples,)

drop_imputer()

Drops the imputer sub-object from this model object

Drops the imputer sub-object from this 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.

Returns **self** – This object

Return type obj

`export_model`(*file*, *add_metada_file=False*)

Export Isolation Forest model

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

This function is not suggested to be used for passing models to and from Python - in such case, one can use `pickle` instead, although the function still works correctly for serializing objects between Python processes.

Note that, if the model was fitted to a `DataFrame`, the column names must be something exportable as JSON, and must be something that R could use as column names (for example, using integers as column names is valid in pandas but not in R).

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.

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 `pickle`.

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

Note: While in earlier versions of this library this functionality used to be faster than `pickle`, starting with version 0.3.0, this function and `pickle` should have similar timings and it's recommended to use `pickle` for serializing objects across Python processes.

Note: **Important:** The model treats boolean variables as categorical. Thus, if the model was fit to a `DataFrame` 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. Also, if using some of Pandas' own Boolean types, these might end up as non-boolean categorical, and if importing the model into R, you might need to pass values as e.g. `"True"` instead of `TRUE` (look at the `.metadata` file to determine this).

Note: 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 curruptions without an informative error message). **This last point applies also to models saved through pickle**. 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.

Note: On Windows, if compiling this library with a compiler other than MSVC or MINGW, there might be issues exporting models larger than 2GB.

Parameters

- **file** (*str*) – The output file path into which to export the model. Must be a file name, not a file handle.
- **add_metada_file** (*bool*) – 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.

Returns **self** – This object.

Return type obj

`fit`(*X*, *y=None*, *sample_weights=None*, *column_weights=None*, *categ_cols=None*)

Fit isolation forest model to data

Parameters

- **X** (*array or array-like (n_samples, n_features)*) – Data to which to fit the model. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSC matrix. If passing a DataFrame, will assume that columns are:
  - Numeric, if their dtype is a subtype of NumPy's 'number' or 'datetime64'.
  - Categorical, if their dtype is 'object', 'Categorical', or 'bool'. Note that, if *Categorical* dtypes are ordered, the order will be ignored here.

Other dtypes are not supported.

Note that, if passing NumPy arrays, they are used in column-major order (a.k.a. "Fortran arrays"), and if they are not already in column-major format, will need to create a copy of the data.

- **y** (*None*) – Not used. Kept as argument for compatibility with SciKit-learn pipelining.
- **sample_weights** (*None or array(n_samples,)*) – Sample observation weights for each row of 'X', 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' in the model constructor method.
- **column_weights** (*None or array(n_features,)*) – Sampling weights for each column in 'X'. Ignored when picking columns by deterministic criterion. If passing None, each column will have a uniform weight. Cannot be used when weighting by kurtosis.
- **categ_cols** (*None or array-like*) – Columns that hold categorical features, when the data is passed as an array or 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 array or list passed here should correspond to the column numbers, with numeration starting at zero. The maximum categorical value should not exceed 'INT_MAX' (typically $2^{31} - 1$). This might be passed either at construction time or when calling `fit` or variations of `fit`.

  This is ignored when the input is passed as a `DataFrame` as then it will consider columns as categorical depending on their dtype.

Returns **self** – This object.

Return type obj

`fit_predict`(*X*, *column_weights=None*, *output_outlierness='score'*, *output_distance=None*, *square_mat=False*, *output_imputed=False*, *categ_cols=None*)

Fit the model in-place and produce isolation or separation depths along the way

See the documentation of other methods ('init', 'fit', 'predict', 'predict_distance') for details.

Note: The data must NOT contain any duplicate rows.

Note: This function will be faster at predicting average depths than calling 'fit' + 'predict' separately when using full row samples.

Note: If using 'penalize_range' = 'True', the resulting scores/depths from this function might differ a bit from those of 'fit' + 'predict' ran separately.

Note: Sample weights are not supported for this method.

Note: When using multiple threads, there can be small differences in the predicted scores or average depth or separation/distance between runs due to roundoff error.

Parameters

- **X** (*array or array-like (n_samples, n_features)*) – Data to which to fit the model. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSC matrix. If passing a DataFrame, will assume that columns are:
  - Numeric, if their dtype is a subtype of NumPy's 'number' or 'datetime64'.
  - Categorical, if their dtype is 'object', 'Categorical', or 'bool'. Note that, if *Categorical* dtypes are ordered, the order will be ignored here.

  Other dtypes are not supported.
- **column_weights** (*None or array(n_features,)*) – Sampling weights for each column in 'X'. Ignored when picking columns by deterministic criterion. If passing None, each column will have a uniform weight. Cannot be used when weighting by kurtosis. Note that, if passing a DataFrame with both numeric and categorical columns, the column names must not be repeated, otherwise the column weights passed here will not end up matching.
- **output_outlierness** (*None or str in ["score", "avg_depth"]*) – Desired type of outlierness output. If passing "score", will output standardized outlier score. If passing "avg_depth" will output average isolation depth without standardizing. If passing 'None', will skip outlierness calculations.
- **output_distance** (*None or str in ["dist", "avg_sep"]*) – Type of distance output to produce. If passing "dist", will standardize the average separation depths. If passing "avg_sep", will output the average separation depth without standardizing it (note that lower separation depth means furthest distance). If passing 'None', will skip distance calculations.
- **square_mat** (*bool*) – Whether to produce a full square matrix with the distances. If passing 'False', will output only the upper triangular part as a 1-d array in which entry (i,j) with 0 <= i < j < n is located at position p(i,j) = (i * (n - (i+1)/2) + j - i - 1). Ignored when passing 'output_distance' = 'None'.
- **output_imputed** (*bool*) – Whether to output the data with imputed missing values. Model object must have been initialized with 'build_imputer' = 'True'.
- **categ_cols** (*None or array-like*) – Columns that hold categorical features, when the data is passed as an array or 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 array or list passed here should correspond to the column numbers, with numeration starting at zero. The maximum categorical value should not exceed 'INT_MAX' (typically $2^{31} - 1$). This might be passed either at construction time or when calling `fit` or variations of `fit`.

  This is ignored when the input is passed as a `DataFrame` as then it will consider columns as categorical depending on their dtype.

Returns **output** – Requested outputs about isolation depth (outlierness), pairwise separation depth (distance), and/or imputed missing values. If passing either 'output_distance' or 'output_imputed', will return a dictionary with keys "pred" (array(n_samples,)), "dist" (array(n_samples * (n_samples - 1) / 2,) or array(n_samples, n_samples)), "imputed" (array-like(n_samples, n_columns)), according to whether each output type is present.

Return type array(n_samples,), or dict

fit_transform(*X*, *y=None*, *column_weights=None*, *categ_cols=None*)

SciKit-Learn pipeline-compatible version of 'fit_predict'

Will fit the model and output imputed missing values. Intended to be used as part of SciKit-learn pipelining. Note that this is just a wrapper over 'fit_predict' with parameter 'output_imputed' = 'True'. See the documentation of 'fit_predict' for details.

Parameters

- **X** (*array or array-like (n_samples, n_features)*) – Data to which to fit the model and whose missing values need to be imputed. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSC matrix (see the documentation of `fit` for more details).

  If the model was fit to a DataFrame with categorical columns, must also be a DataFrame.

- **y** (*None*) – Not used. Kept for compatibility with SciKit-Learn.

- **column_weights** (*None or array(n_features,)*) – Sampling weights for each column in 'X'. Ignored when picking columns by deterministic criterion. If passing None, each column will have a uniform weight. Cannot be used when weighting by kurtosis. Note that, if passing a DataFrame with both numeric and categorical columns, the column names must not be repeated, otherwise the column weights passed here will not end up matching.

- **categ_cols** (*None or array-like*) – Columns that hold categorical features, when the data is passed as an array or 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 array or list passed here should correspond to the column numbers, with numeration starting at zero. The maximum categorical value should not exceed 'INT_MAX' (typically $2^{31} - 1$). This might be passed either at construction time or when calling `fit` or variations of `fit`.

  This is ignored when the input is passed as a `DataFrame` as then it will consider columns as categorical depending on their dtype.

Returns **imputed** – Input data 'X' with missing values imputed according to the model.

Return type array-like(n_samples, n_columns)

`generate_sql`(*enclose='doublequotes'*, *output_tree_num=False*, *tree=None*, *table_from=None*, *select_as='outlier_score'*, *column_names=None*, *column_names_categ=None*)

Generate SQL statements representing the model prediction function

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 zero).

Note: 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.

Note: 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.

Note: The generated SQL statements will not include range penalizations, thus predictions might differ from calls to `predict` when using `penalize_range=True`.

Note: 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).

Note: 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).

Parameters

- **enclose** (*str*) – 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"`: Will enclose them as `"column_name"` - this will work for e.g. PostgreSQL.

  `"squarebraces"`: Will enclose them as `[column_name]` - this will work for e.g. SQL Server.

  `"none"`: Will output the column names as-is (e.g. `column_name`)
- **output_tree_num** (*bool*) – Whether to make the statements return the terminal node number instead of the isolation depth. The numeration will start at zero.
- **tree** (*int or None*) – Tree for which to generate SQL statements. If passed, will generate the statements only for that single tree. If passing 'None', will generate statements for all trees in the model.
- **table_from** (*str or None*) – 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** (*str*) – Alias to give to the generated outlier score in the select statement. Ignored when not passing `table_from`.
- **column_names** (*None or list[str]*) – Column names to use for the **numeric** columns. If not passed and the model was fit to a `DataFrame`, will use the column names from that `DataFrame`, which can be found under `self.cols_numeric_`. If not passing it and the model was fit to data in a format other than `DataFrame`, 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** (*None or list[str]*) – Column names to use for the **categorical** columns. If not passed, will use the column names from the `DataFrame` to which the model was fit. These can be found under `self.cols_categ_`.

Returns **sql** – A list of SQL statements for each tree as strings, or the SQL statement for a single tree if passing 'tree', or a single select-from SQL statement with all the trees concatenated if passing `table_from`.

Return type list[str] or str

`get_num_nodes()`

Get number of nodes per tree

Gets the number of nodes per tree, along with the number of terminal nodes.

Returns **nodes** – A tuple in which the first element denotes the total number of nodes in each tree, and the second element denotes the number of terminal nodes. Both are returned as arrays having one entry per tree.

Return type tuple(array(n_trees,), array(n_trees,))

`get_params`(*deep=True*)

Get parameters for this estimator.

Kept for compatibility with scikit-learn.

Parameters **deep** (*bool*) – Ignored.

Returns **params** – Parameter names mapped to their values.

Return type dict

`static import_model`(*file*)

Load an Isolation Forest model exported from R or Python

Loads a serialized Isolation Forest model as produced and exported by the function `export_model` or by the R version of this package. Note that the metadata must be something importable in Python - e.g. column names must be valid for Pandas.

It's recommended to generate a '.metadata' file (passing `add_metada_file=True`) and to visually inspect said file in any case.

See the documentation for `export_model` for details about compatibility of the generated files across different machines and versions.

Note:

This is a static class method - that is, it should be called like this: `iso = IsolationForest.import_model(...)`

(i.e. no parentheses after *IsolationForest*)

Note: While in earlier versions of this library this functionality used to be faster than `pickle`, starting with version 0.3.0, this function and `pickle` should have similar timings and it's recommended to use `pickle` for serializing objects across Python processes.

Parameters **file** (*str*) – The input file path containing an exported model along with its metadata file. Must be a file name, not a file handle.

Returns **iso** – An Isolation Forest model object reconstructed from the serialized file and ready to use.

Return type IsolationForest

`partial_fit`(*X*, *sample_weights=None*, *column_weights=None*)

Add additional (single) tree to isolation forest model

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

Note: If constructing trees with different sample sizes, the outlier scores 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 construction argument (if that is `None` or `"auto"`, will then set it as the sample size of the first tree).

Note: This function is not thread-safe - that is, it will produce problems if one tries to call this function on the same model object in parallel through e.g. `joblib` with a shared-memory backend (which is not the default for joblib).

Parameters

- **X** (*array or array-like (n_samples, n_features)*) – Data to which to fit the new tree. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSC matrix. If passing a DataFrame, will assume that columns are:
  - Numeric, if their dtype is a subtype of NumPy's 'number' or 'datetime64'.
  - Categorical, if their dtype is 'object', 'Categorical', or 'bool'. Note that, if *Categorical* dtypes are ordered, the order will be ignored here. Categorical columns, if any, may have new categories.

  Other dtypes are not supported.
- **sample_weights** (*None or array(n_samples,)*) – 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 'None', model must have been built with 'weights_as_sample_prob' = 'False'.
- **column_weights** (*None or array(n_features,)*) – Sampling weights for each column in 'X'. Ignored when picking columns by deterministic criterion. If passing None, each column will have a uniform weight. Cannot be used when weighting by kurtosis.

Returns **self** – This object.

Return type obj

`predict`(*X*, *output='score'*)

Predict outlierness based on average isolation depth

Calculates the approximate depth that it takes to isolate an observation according to the fitted model splits. Can output either the average depth, or a standardized outlier score based on whether it takes more or fewer splits than average to isolate observations. In the standardized outlier score metric, 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).

Note: Depending on the model parameters, it might be possible to convert the models to 'treelite' format for faster predictions or for easier model serving. See method `to_treelite` for details.

Note: If the model was built with 'nthreads>1', this prediction function 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 'flask'+'gunicorn', which uses process forking for parallelization, it will cause the whole application to freeze; and if using kubernetes on top of a different backend such as 'falcon', might cause it to run slower than needed or to hang too. A potential fix in these cases is to set the number of threads to 1 in the object (e.g. 'model.nthreads = 1'), or to use a different version of this library compiled without OpenMP (requires manually altering the 'setup.py' file), or to use a non-GNU OpenMP backend. This should not be an issue when using this library normally in e.g. a jupyter notebook.

Note: The more threads that are set for the model, the higher the memory requirements will be as each thread will allocate an array with one entry per row.

Note: In order to save memory when fitting and serializing models, the functionality for outputting terminal node number 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.

Note: 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 'fit_predict' functionality.

Note: If using non-random splits (parameters `prob_pick_avg_gain`, `prob_pick_pooled_gain`) and/or range penalizations (which are off by default), the distribution of scores might not be centered around 0.5.

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

Parameters

- **X** (*array or array-like (n_samples, n_features)*) – Observations for which to predict outlierness or average isolation depth. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSC or CSR matrix.

  If 'X' 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 format as it will be much faster when the number of trees or rows is large.

  While the 'X' used by `fit` always needs to be in column-major order, predictions can be done on data that is in either row-major or column-major orders, with row-major being faster for dense data.

- **output** (*str, one of "score", "avg_depth", "tree_num", "tree_depths"*) – Desired type of output. Options are:

  `"score"`: Will output standardized outlier scores.

  `"avg_depth"`: Will output unstandardized average isolation depths.

  `"tree_num"`: Will output the index of the terminal node under each tree in the model.

  `"tree_depths"`: Will output non-standardized per-tree isolation depths (note that they will not include range penalties from `penalize_range=True`).

Returns **score** – Requested output type for each row accoring to parameter 'output' (outlier scores, average isolation depth, terminal node indices, or per-tree isolation depths).

Return type array(n_samples,) or array(n_samples, n_trees)

`predict_distance`(*X*, *output='dist'*, *square_mat=False*, *X_ref=None*)

Predict approximate distances between points

Predict approximate pairwise distances between points or individual distances between two sets of points based on how many splits it takes to separate them. Can output either the average number of paths, or a standardized metric (in the same way as the outlier score) in which values closer to zero indicate nearer points, closer to one further away points, and closer to 0.5 average distance.

Note: 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 combination.

Parameters

- **X** (*array or array-like (n_samples, n_features)*) – Observations for which to calculate approximate pairwise distances, or first group for distances between sets of points. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSC matrix.
- **output** (*str, one of "dist", "avg_sep"*) – Type of output to produce. If passing "dist", will standardize the average separation depths. If passing "avg_sep", will output the average separation depth without standardizing it (note that lower separation depth means furthest distance).
- **square_mat** (*bool*) – Whether to produce a full square matrix with the pairwise distances. If passing 'False', will output only the upper triangular part as a 1-d array in which entry (i,j) with 0 <= i < j < n is located at position p(i,j) = (i * (n - (i+1)/2) + j - i - 1). Ignored when passing `X_ref`.
- **X_ref** (*array or array-like (n_ref, n_features)*) – Second group of observations. If passing it, will calculate distances between each point in `X` and each point in `X_ref`. If passing `None` (the default), will calculate pairwise distances between the points in `X`. Must be of the same type as `X` (e.g. array, DataFrame, CSC).

Returns **dist** – Approximate distances or average separation depth between points, according to parameter 'output'. Shape and size depends on parameter `square_mat`, or `X_ref` if passed.

Return type array(n_samples * (n_samples - 1) / 2,) or array(n_samples, n_samples) or array(n_samples, n_ref)

`set_params`(***params*)

Set the parameters of this estimator.

Kept for compatibility with scikit-learn.

Note: Setting any parameter other than the number of threads will reset the model - that is, if it was fitted to some data, the fitted model will be lost, and it will need to be refitted before being able to make predictions.

Parameters ****params** (*dict*) – Estimator parameters.

Returns **self** – Estimator instance.

Return type estimator instance

`subset_trees`(*trees_take*)

Subset trees of a given model

Creates a new model containing only selected trees of this model object.

Parameters **trees_take** (*array_like(n,)*) – Indices of the trees of this model to copy over to the new model. Must be integers with numeration starting at zero.

Returns **new_model** – A new IsolationForest model object, containing only the subset of trees from this object that was specified under 'trees_take'.

Return type obj

`to_treelite`(*use_float32=False*)

Convert model to 'treelite' format

Converts an IsolationForest model to a 'treelite' object, which can be compiled into a small standalone runtime library for smaller models and usually faster predictions:

https://treelite.readthedocs.io/en/latest/index.html

A couple notes about this conversion:

- It is only possible to convert to 'treelite' when using `ndim=1` (which is not the default).
- The 'treelite' and 'treelite_runtime' libraries must be installed for this to work.
- The options for handling missing values in 'treelite' are more limited. This function will always produce models that force `missing_action="impute"`, regardless of how the IsolationForest model itself handles them.
- The options for handling unseen categories in categorical variables are also more limited in 'treelite'. It's not possible to convert models that use `new_categ_action="weighted"`, and categories that were not present within the training data (which are not meant to be passed to 'treelite') will always be sent to the right side of the split, which might produce different results from `predict`.
- Some features such as range penalizations will not be kept in the 'treelite' model.
- While this library always uses C 'double' precision (typically 'float64') for model objects and prediction outputs, 'treelite' (a) can use 'float32' precision, (b) converts floating point numbers to a decimal representation and back to floating point; which combined can result in some precision loss which leads to producing slightly different predictions from the `predict` function in this package.
- The output returned from a compiled 'treelite' model when calling `predict` will be the average isolation depth, as it does not (yet?) support the standardized outlier score from isolation forests.
- If the model was fit to a DataFrame having a mixture of numerical and categorical columns, the resulting 'treelite' object will be built assuming all the numerical columns come before the categorical columns, regardless of which order they originally had in the data that was passed to 'fit'. In such cases, it is possible to check the order of the columns under attributes `self.cols_numeric_` and `self.cols_categ_`.
- Categorical columns in 'treelite' are passed as integer values. if the model was fit to a DataFrame with categorical columns, the encoding that is used can be found under `self._cat_mapping`.
- The 'treelite' object returned by this function will not yet have been compiled. It's necessary to call `compile` and `export_lib` afterwards in order to be able to use it.

Parameters **use_float32** (*bool*) – Whether to use 'float32' type for the model. This is typically faster but has less precision than the typical 'float64' (outside of this conversion, models from this library always use 'float64').

Returns **model** – A 'treelite' model object.

Return type obj

`transform`(*X*)

Impute missing values in the data using isolation forest model

---

Note: In order to use this functionality, the model must have been built with imputation capabilities ('build_imputer' = 'True').

Note: Categorical columns, if imputed with a model fit to a DataFrame, will always come out with pandas categorical dtype.

Note: The input may contain new columns (i.e. not present when the model was fitted), which will be output as-is.

Parameters **X** (*array or array-like (n_samples, n_features)*) – Data for which missing values should be imputed. Can pass a NumPy array, Pandas DataFrame, or SciPy sparse CSR matrix.

If the model was fit to a DataFrame with categorical columns, must also be a DataFrame.

Returns **X_imputed** – Object of the same type and dimensions as 'X', but with missing values already imputed. Categorical columns will be output as pandas's 'Categorical' regardless of their dtype in 'X'.

Return type array or array-like (n_samples, n_features)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX