A4 Report

Scikit-learn

Table of Contents

Analysis of 2 Issues

Enhancement: Test estimators are deterministic (Issue 7139):

Analysis

The enhancement asks to check if given a random state in the estimators, whether or not the output is deterministic during predicting the training data after fitting. In the issue description, we are asked to check whether or not estimators with or without a random state are deterministic. Upon further investigation, we located all the checks for the estimators and determined that the deterministic check can be done there.

Affected areas in codebase

The files this issue concerns are the following:

- /sklearn/utils/estimators_check.py

Proposed solutions and designs

We would create a new function called `check_determinism`. First, check if `random_state` is given in the parameters. For each estimator, we fit two instances of the same estimator. Then use both to predict on a subset of data and compare the resulting predictions. Since `check_determinism` is a static utility function, it has no impact on the overall architecture of scikit-learn. This function is to be used within the other estimator classes to test for determinism. Since this functionality is useful for all of the estimator classes, we decided to add it to `estimator_checks.py` as it is globally available to all the estimators.

Enhancement: Add Sparse Matrix Support For HistGradientBoostingClassifier (Issue 15336):

Analysis of Issue

The enhancement asks to add support for sparse matrices in HistGradientBoostingClassifier. In the issue description, we are asked to add support so that the size of the text does not prohibit converting sparse matrices into dense matrices. Upon further investigation, we notice a TypeError is thrown when the matrix is sparse, asking the user to use a dense matrix instead. We notice that the issue occurs during the fit and transform. Following this investigation, we determined that the issue can be simplified by compressing and decompressing the sparse matrix whenever needed.

Affected areas in codebase

The files this issue concerns are the following:

- /sklearn/ensemble/_hist_gradient_boosting/binning.py
- /sklearn/ensemble/_hist_gradient_boosting/gradient_boosting.py

Proposed solutions and designs

There are 2 parts, first is handling sparse matrices during fitting and then during transform.

Fitting

Only unique values are important. We can get the non-null values and append a zero to the list of values, allowing the matrix to be treated the same as a dense matrix.

Transform

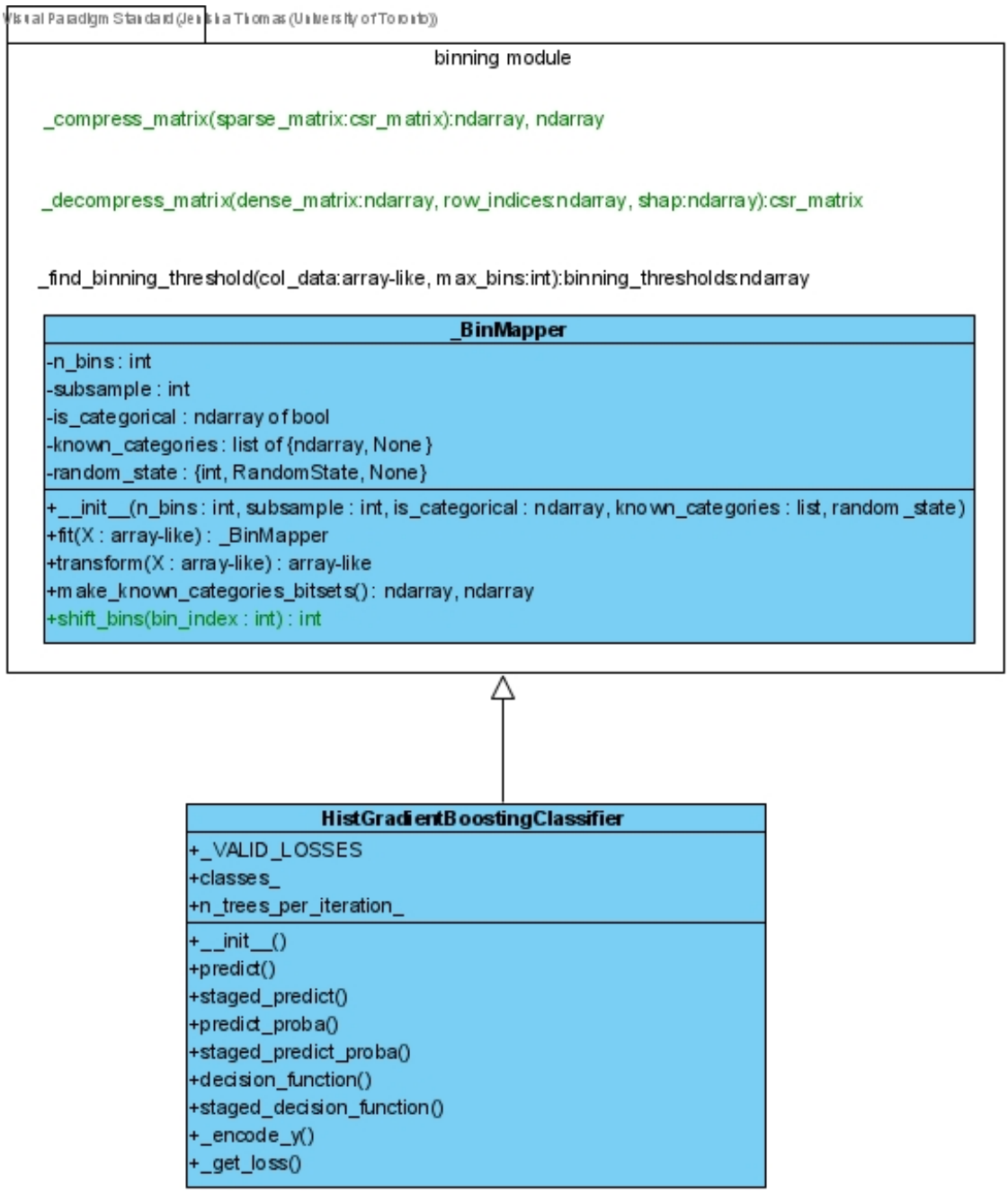
We want the bin with value zero to be mapped to the value of zero. We can take missing values as zero. The bins are right-shifted by one and the bin `bin_zero` is mapped to zero.

The binning logic is taking place in `_map_to_bins`, but to avoid touching cython code which is used elsewhere in scikit-learn we can instead pass a zero 1x1 matrix to `_map_to_bins`, then compress the sparse matrix into a dense matrix. Then we map the bins, shift the mapped bins, and finally decompress the dense matrix to maintain the original structure.

Note: Compressing and decompressing are done column-wise.

The majority of the changes we made didn't affect the overall architecture of scikit-learn, as we simply modified existing methods and added some static methods to be used in the existing

methods. The static methods we added were the `_compress_matrix`, used in the `transform` method of the `_BinMapper` class to convert a sparse matrix into a dense matrix, and `_decompress_matrix` which converts a dense matrix to a sparse one. One change that did affect the architecture slightly was the addition of the `shift_bin` method to the `_BinMapper` class, which is used to shift the bins to the right and assign bin_zero to bin 0.



Sparse Matrix Enhancement Implementation

User Guide

sklearn.ensemble._hist_gradient_boosting.binning

module sklearn.ensemble._hist_gradient_boosting.binning [Source]

This module contains the BinMapper class.

BinMapper is used for mapping a real-valued dataset into integer-valued bins. Bin thresholds are computed with the quantiles so that each bin contains approximately the same number of samples.

Methods

_compress_matrix(sparse_matrix)	Compress a sparse matrix into a dense matrix.
_decompress_matrix(dense_matrix, row_indices, shape)	Bin data X.
_find_binning_thresholds(col_data, max_bins)	Extract quantiles from a continuous feature

_compress_matrix(*sparse_matrix*) [Source]

Compress a sparse matrix into a dense matrix.

Compressing is done column-wise.

Parameters:

sparse_matrix : *csr_matrix, shape (n_samples, n_features)*

Returns:

dense_matrix : *ndarray, shape(n_non_zero_samples, n_features)*
The `sparse_matrix` compressed, number of rows is equal to the largest number of non-zero samples per column.

row_indices : *ndarray, shape(n_non_zero_samples, n_features)*

The initial row indices of before sample columns were compressed

_decompress_matrix(*dense_matrix, row_indices, shape*) [Source]

Decompress a dense matrix into a sparse matrix.

Decompressing is done column-wise.

Parameters:

dense_matrix : *ndarray, shape (n_samples, n_features)*

row_indixes : *ndarray, shape (n_samples, n_features)*

shape : *ndarray, shape (n_samples, n_features)*

Returns:

sparse_matrix : *csr_matrix, shape (n_samples, n_features)*
The `dense_matrix` decompressed, number of rows is equal to the largest number of samples per column.

_find_binning_thresholds(*col_data, max_bins*) [Source]

Extract quantiles from a continuous feature.

Missing values are ignored for finding the thresholds.

Parameters:

col_data : *array-like, shape (n_samples,)*
The continuous feature to bin.

max_bins: *int*
The maximum number of bins to use for non-missing values. If for a given feature the number of unique values is less than `"max_bins"`, then those unique values will be used to compute the bin thresholds, instead of the quantiles

Returns:

binning_thresholds : *ndarray of shape(min(max_bins, n_unique_values) - 1,)*
The increasing numeric values that can be used to separate the bins. A given value x will be mapped into bin value i iff bining_thresholds[i - 1] < x <= binning_thresholds[i]

sklearn.ensemble._hist_gradient_boosting.binning._BinMapper

class sklearn.ensemble._hist_gradient_boosting.binning._BinMapper [Source]

Transformer that maps a dataset into integer-valued bins.

For continuous features, the bins are created in a feature-wise fashion, using quantiles so that each bin contains approximately the same number of samples. For large datasets, quantiles are computed on a subset of the data to speed up the binning, but the quantiles should remain stable.

For categorical features, the raw categorical values are expected to be in [0, 254] (this is not validated here though) and each category corresponds to a bin. All categorical values must be known at initialization: transform() doesn't know how to bin unknown categorical values. Note that transform() is only used on non-training data in the case of early stopping.

Features with a small number of values may be binned into less than `"n_bins"` bins. The last bin (at index `"n_bins - 1"`) is always reserved for missing values.

Parameters

n_bins : *int, default=256*
The maximum number of bins to use (including the bin for missing values). Should be in [3, 256]. Non-missing values are binned on `"max_bins = n_bins - 1"` bins. The last bin is always reserved for missing values. If for a given feature the number of unique values is less than `"max_bins"`, then those unique values will be used to compute the bin thresholds, instead of the quantiles. For categorical features indicated by `"is_categorical"`, the docstring for `"is_categorical"` details on this procedure.

subsample : *int or None, default=2e5*
If `"n_samples > subsample"`, then `"sub_samples"` samples will be randomly chosen to compute quantiles. If `"None"`, whole data is used.

is_categorical : *ndarray of bool of shape (n_features,), default=None*
Indicates categorical features. By default, all features are continuous.

known_categories : *list of {ndarray, None} of shape (n_features,), default=none*
For each categorical feature, the array indicates the set of unique categorical values. These should be the possible values over all the data, not just the training data. For continuous features, the corresponding entry should be `None`.

random_state: *int, RandomState instance or None, default=None*
Pseudo-random number generator to control the random sub-sampling.

	Pass an `int` for reproducible output across multiple function calls. See Glossary
Attributes	**bin_thresholds_ : *list of ndarray*** For each feature, each array indicates how to map a feature into a binned feature. The semantic and size depends on the nature of the feature: ● for real-valued features, the array corresponds to the real-valued bin thresholds (the upper bound of each bin). There are `"max_bins - 1"` thresholds, where `"max_bins = n_bins - 1"` is the number of bins used for non-missing values. ● for categorical features, the array is a map from a binned category value to the raw category value. The size of the array is equal to `"min(max_bins, category_cardinality)"` where we ignore missing values in the cardinality. **n_bins_non_missing_ : *ndarray, dtype=np.uint32*** For each feature, gives the number of bins actually used for non-missing values. For features with a lot of unique values, this is equal to `"n_bins - 1"`. **is_categorical_ : *ndarray of shape (n_features,), dtype=np.uint8*** Indicator for categorical features. **missing_values_bin_idx_ : *np.uint8*** The index of the bin where missing values are mapped. This is a constant across all features. This corresponds to the last bin, and it is always equal to `"n_bins - 1"`. Note that if `"n_bins_missing_"` is less than `"n_bins - 1"` for a given feature, then there are empty (and unused) bins.

Methods

`fit`(X[, y])	Fit data X by computing the binning thresholds.
`transform`(X)	Bin data X.
`make_known_categories_bitsets`()	Create bitsets of known categories.
`shift_bins`(bin_index, bin_zero)	Shift bins [1: bin_zeros - 1] right, assign bin_zero to bin 0

fit(*self, X, y=None*) [Source]

Fit data X by computing the binning thresholds.

The last bin is reserved for missing values, whether missing values are present in the data or not.

Parameters:

X : *array-like of shape (n_samples, n_features)*
The data to bin.

y: *None*
Ignored.

Returns:

self : *object*

transform(*self, X*) [Source]

Bin data X.

Missing values will be mapped to the last bin.

For categorical features, the mapping will be incorrect for unknown categories. Since the BinMapper is given known_categories of the entire training data (i.e. before the call to `train_test_split()` in case of early-stopping), this never happens.

Parameters:

X : *array-like of shape (n_samples, n_features)*
The data to bin.

y: *None*
Ignored.

Returns:

binned : *array-like of shape (n_samples, n_features)*
The binned data (fortran-aligned).

make_known_categories_bitsets(*self*) [Source]

Create bitsets of known categories.

Returns:

known_cat_bitsets : *ndarray of shape (n_categorical_features, 8)*
Array of bitsets of known categories, for each categorical feature.

f_idx_map : *ndarray of shape (n_features,)*
Map from original feature index to the corresponding index in the `known_cat_bitsets` array.

shift_bins(*self, bin_index, bin_zero*) [Source]

Shift the `bins [1: bin_zeros - 1]` to the right and assign bin_zero to bin 0

Parameters:

bin_index : *int*
The initial index of bin.

bin_zero : *int*
Bin zero index.

Returns:

bin_index : *int*
The shifted index of bin..

User Acceptance Tests

For _compress_matrix:

1. Create a sparse matrix X with dimension m x n where m can be equal to n
2. Provide the statement **compressed_X = binning._compress_matrix(X)**
3. The expected outcome of this function is that the **sparse** matrix will be converted to a **dense** matrix, whose number of rows is equal to the largest number of non-zero samples per column. The dense matrix can be accessed using **compressed_X[0]**

For _decompress_matrix:

1. Create a sparse matrix X with dimension m x n where m can be equal to n
2. Provide the statement **decompressed_X = binning._decompress_matrix(X, row_indices, shape)**
3. The expected outcome of this function is that the **dense** matrix will be converted to a **sparse** matrix, whose number of rows is equal to the largest number of samples per column

For prediction of the sparse matrix:

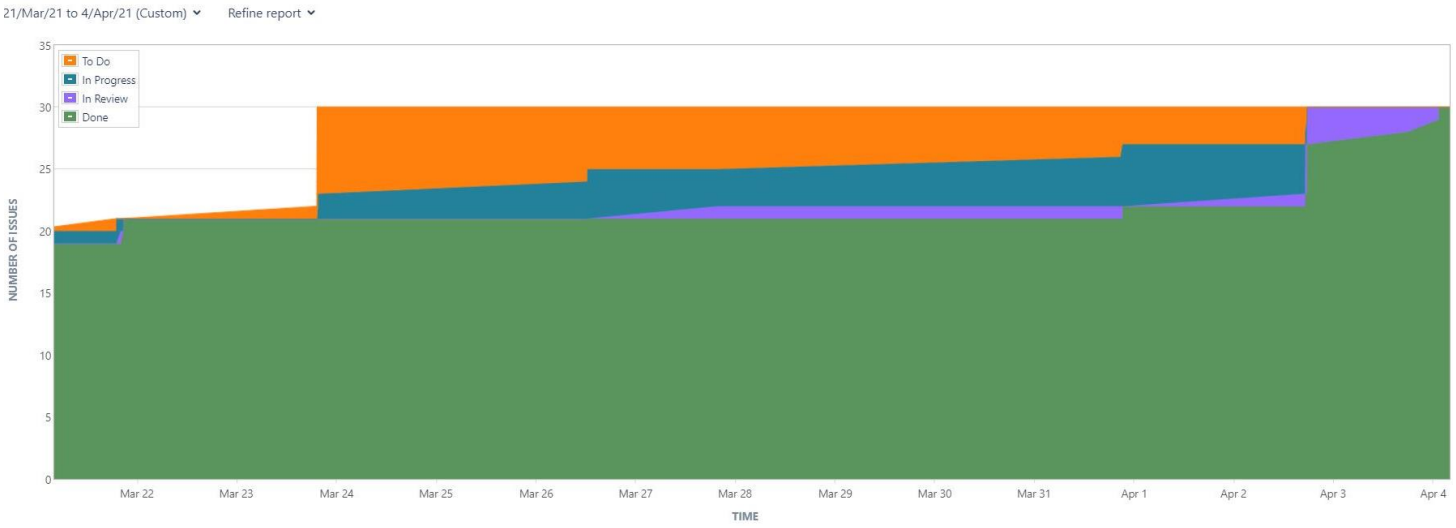
1. Create a sparse matrix X with dimension m x n where m can be equal to n
2. Create a matrix Y with any dimension
3. Provide the statement **clf = HistGradientBoostingClassifier()** along with fitting X and Y using **clf.fit(X, Y)** and then predict on X using **clf.predict(X)**
4. The expected outcome of this function is that a prediction is made on the classes of X

Unit Testing Instructions

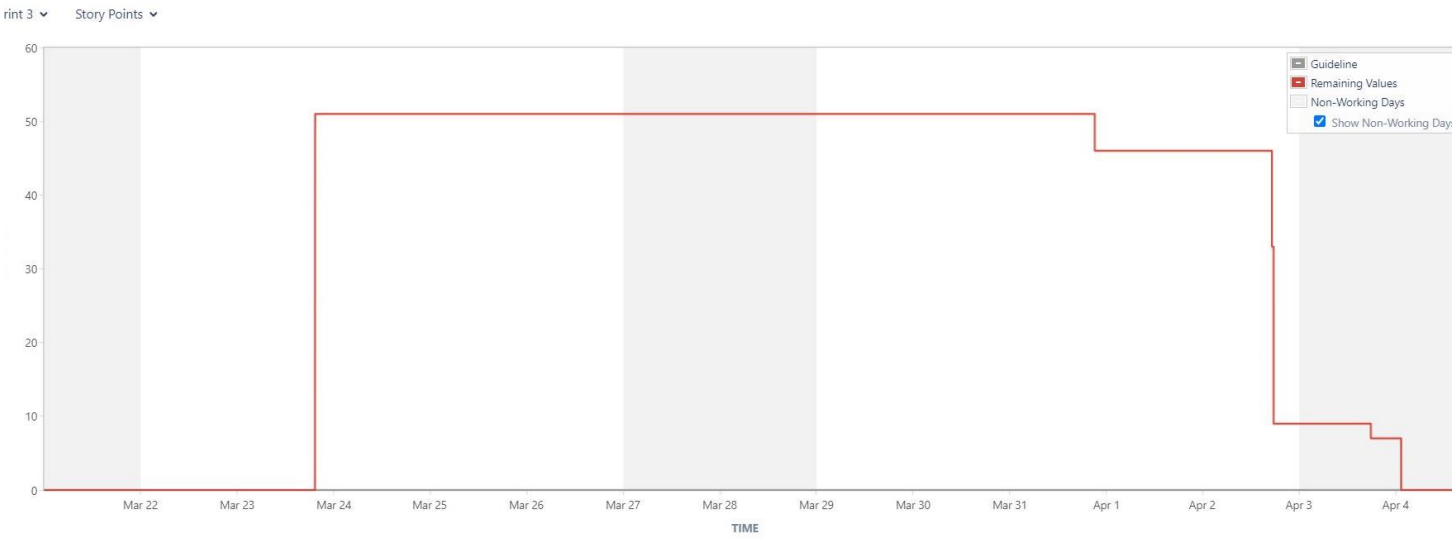
The cases can be run by using the command "`pytest 15336-testsuite.py`" in your terminal opened in the `a4` directory after successfully installing scikit-learn.

Development Process

JIRA



Cumulative Flow Diagram



Burndown Chart