tabmat

QuantCo, Inc.

May 06, 2024

CONTENTS:

Please see the project README for a broad overview of what `tabmat` is.

CHAPTER

ONE

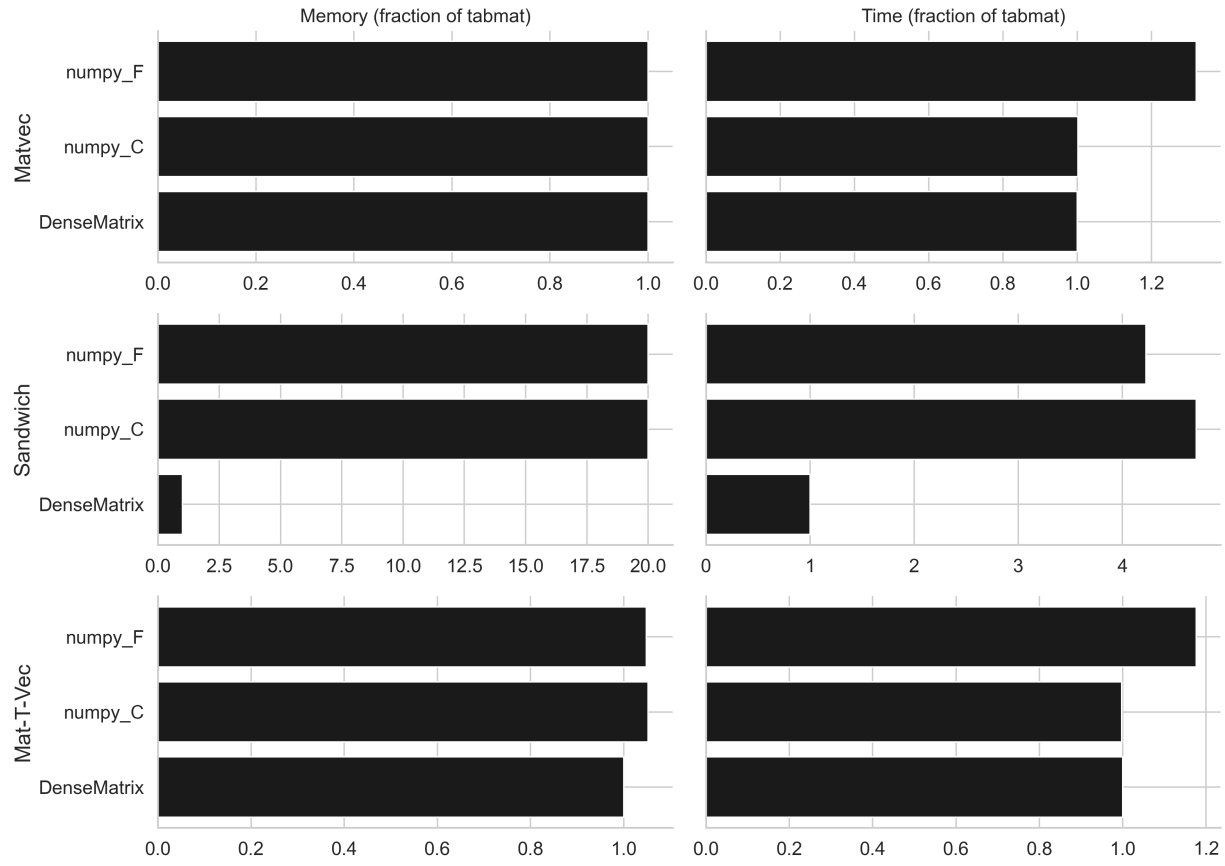
BENCHMARKS

To generate the data to run all the benchmarks: `python src/tabmat/benchmark/generate_matrices.py`. Then, to run all the benchmarks: `python src/tabmat/benchmark/main.py`. To produce or update these figures, open `src/tabmat/benchmark/visualize_benchmarks.py` as a notebook via `jupytext`.

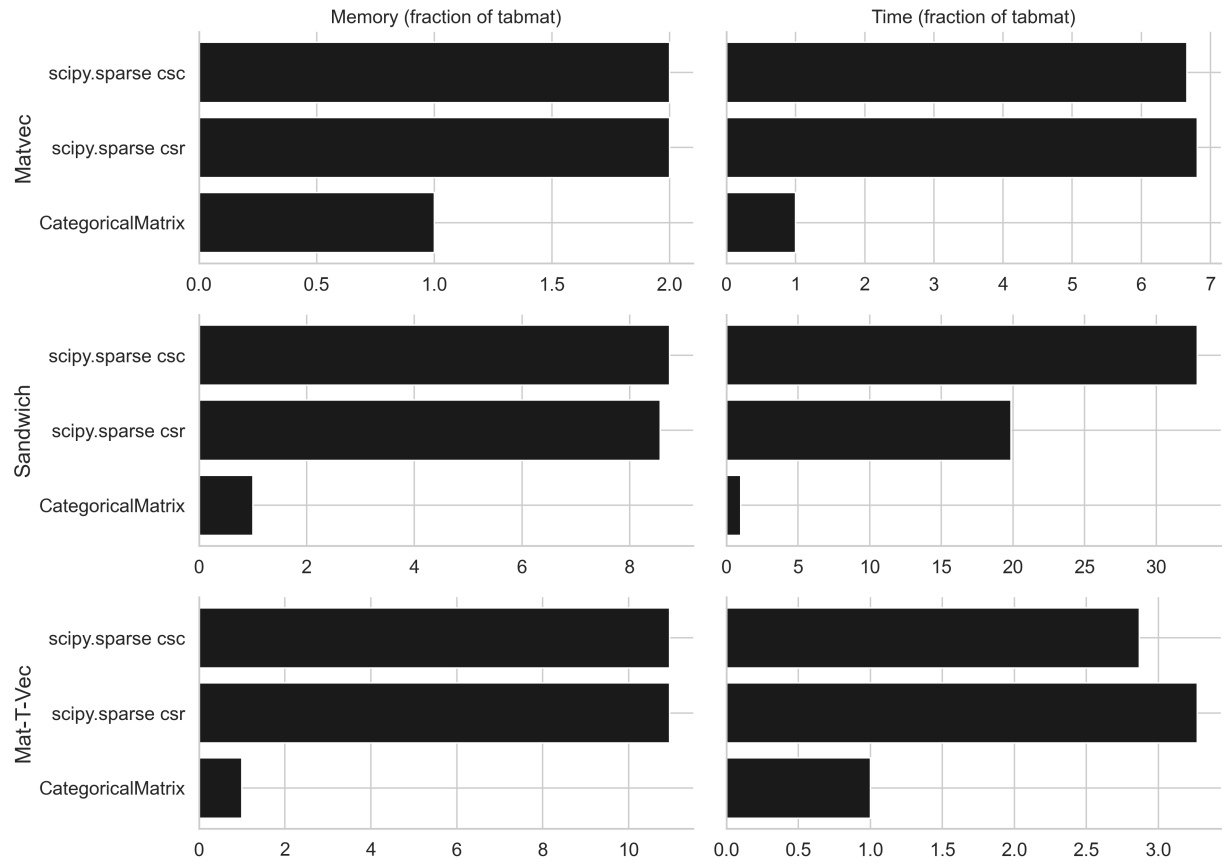
For more info on the benchmark CLI: `python src/tabmat/benchmark/main.py --help`.

1.1 Performance

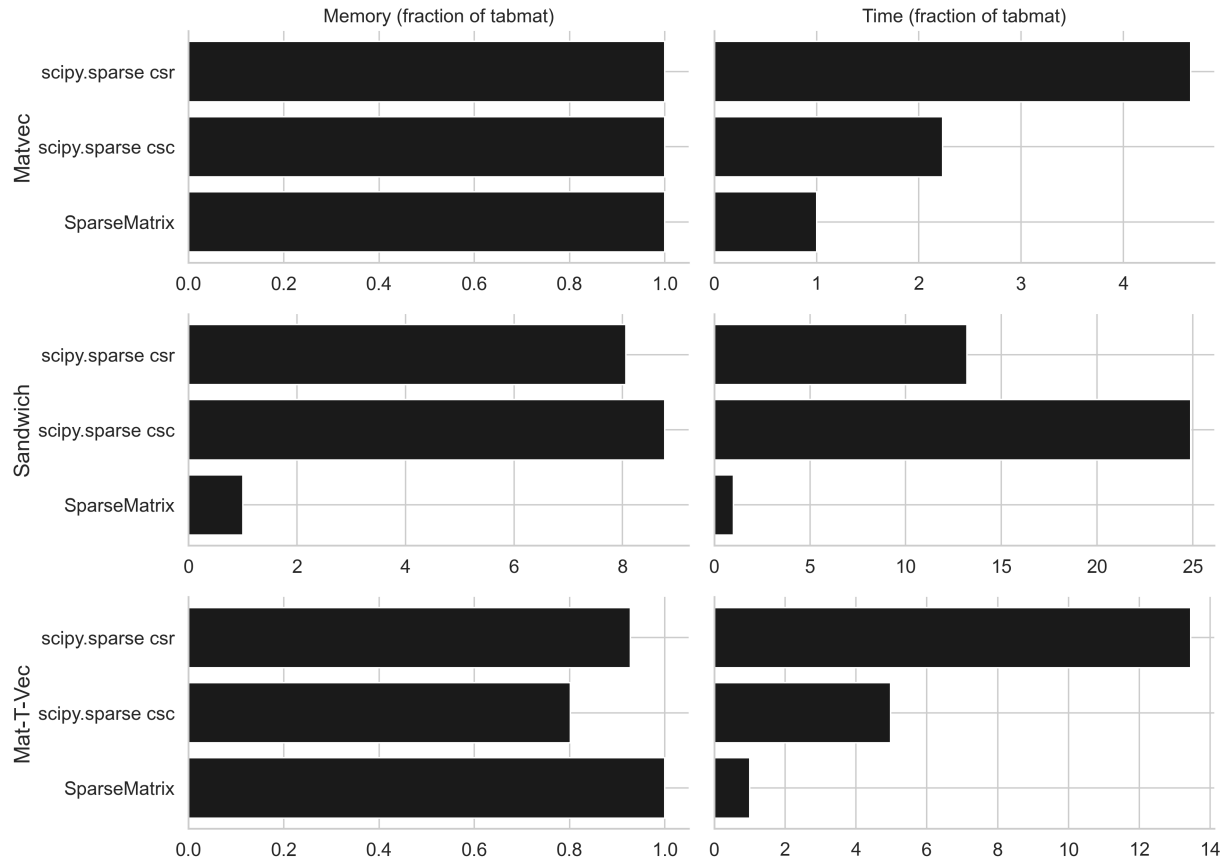
Dense matrix, 4M x 10:



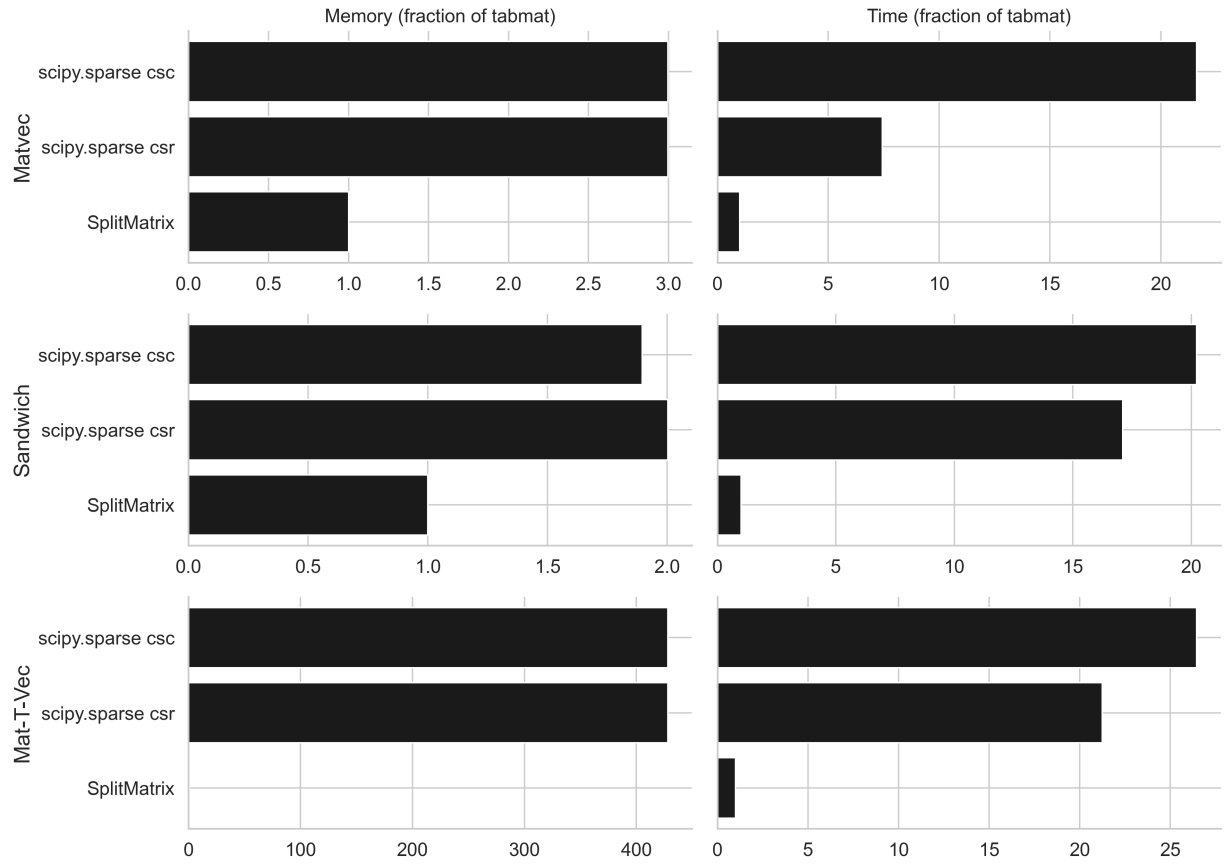
One-hot encoded categorical variable, 1M x 100k:



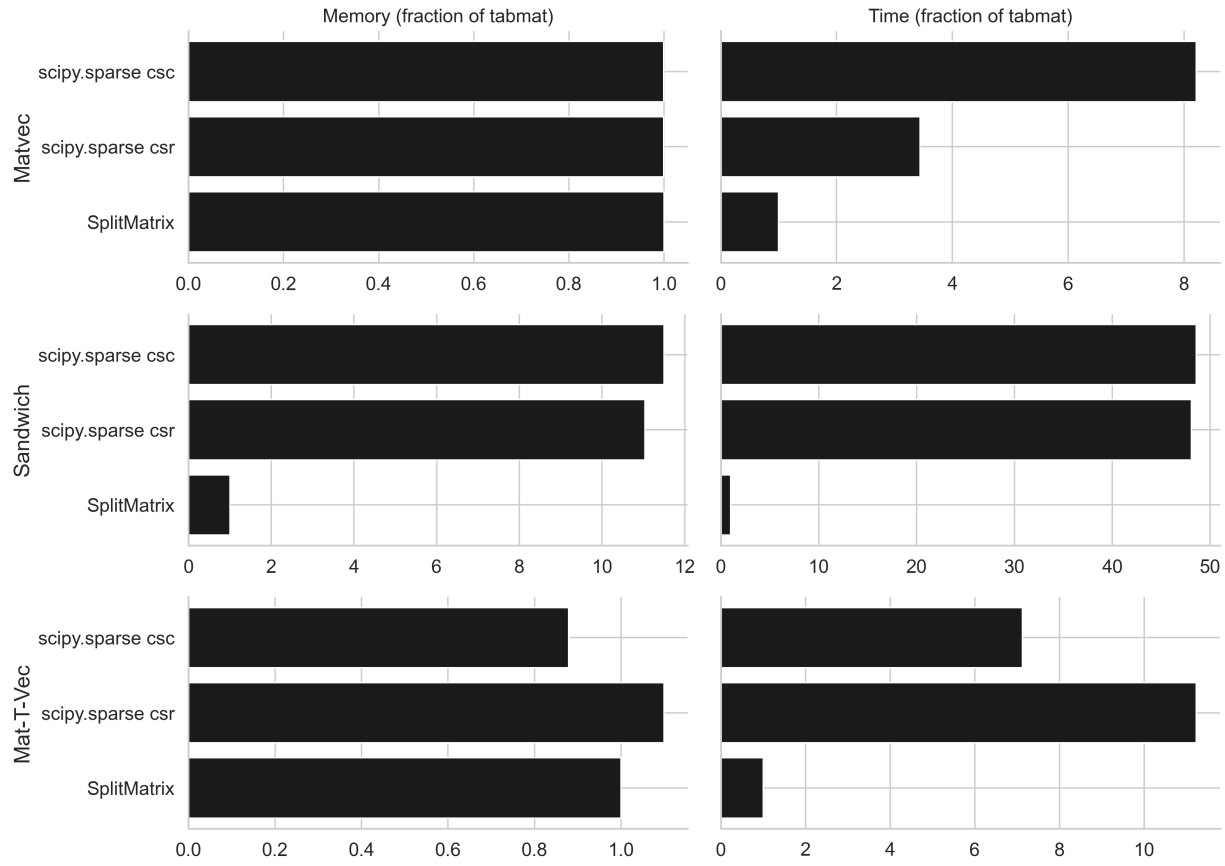
Sparse matrix, 1M x 1k:



Two categorical matrices, 1M x 2k:



Dense matrix plus two categorical matrices, 3M x (dense=5, cat1=10, cat2=1000).


Memory (fraction of tabmat)
Time (fraction of tabmat)
Matvec
Sandwich
Mat-T-Vec
scipy.sparse csc
scipy.sparse csr
SplitMatrix
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12
20
30
40
50

CHAPTER

TWO

TABMAT PACKAGE

tabmat.**from_pandas**(*df*, *dtype=<class 'numpy.float64'>*, *sparse_threshold=0.1*, *cat_threshold=4*, *object_as_cat=False*, *cat_position='expand'*, *drop_first=False*, *categorical_format='{name}[{category}]'*, *cat_missing_method='fail'*, *cat_missing_name='(MISSING)'*)

Transform a pandas.DataFrame into an efficient SplitMatrix. For most users, this will be the primary way to construct tabmat objects from their data.

Parameters

- **df** (*pd.DataFrame*) – pandas DataFrame to be converted.
- **dtype** (*np.dtype, default np.float64*) – dtype of all sub-matrices of the resulting SplitMatrix.
- **sparse_threshold** (*float, default 0.1*) – Density threshold below which numerical columns will be stored in a sparse format.
- **cat_threshold** (*int, default 4*) – Number of levels of a categorical column under which the column will be stored as sparse one-hot-encoded columns instead of CategoricalMatrix
- **object_as_cat** (*bool, default False*) – If True, DataFrame columns stored as python objects will be treated as categorical columns.
- **cat_position** (*str {'end'|'expand'}, default 'expand'*) – Position of the categorical variable in the index. If “last”, all the categoricals (including the ones that did not satisfy cat_threshold) will be placed at the end of the index list. If “expand”, all the variables will remain in the same order.
- **drop_first** (*bool, default False*) – If true, categoricals variables will have their first category dropped. This allows multiple categorical variables to be included in an unregularized model. If False, all categories are included.
- **cat_missing_method** (*str {'fail'|'zero'|'convert'}, default 'fail'*) – How to handle missing values in categorical columns: - if ‘fail’, raise an error if there are missing values. - if ‘zero’, missing values will represent all-zero indicator columns. - if ‘convert’, missing values will be converted to the ‘(MISSING)’ category.
- **cat_missing_name** (*str, default '(MISSING)'*) – Name of the category to which missing values will be converted if `cat_missing_method='convert'`.
- **categorical_format** (*str*)

Return type

SplitMatrix

tabmat.**from_csc**(*mat*, *threshold=0.1*, *column_names=None*, *term_names=None*)

Convert a CSC-format sparse matrix into a `SplitMatrix`.

The `threshold` parameter specifies the density below which a column is treated as sparse.

Parameters
mat (*csc_matrix*)

class tabmat.**MatrixBase**

Bases: `ABC`

Base class for all matrix classes. `MatrixBase` cannot be instantiated.

property A: ndarray

Convert self into an np.ndarray. Synonym for `toarray()`.

property column_names

Column names of the matrix.

abstract get_names(*type='column'*, *missing_prefix=None*, *indices=None*)

Get column names.

For columns that do not have a name, a default name is created using the following pattern: `"{missing_prefix}{start_index + i}"` where `i` is the index of the column.

Parameters

- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).
- **missing_prefix** (*Optional[str], default None*) – Prefix to use for columns that do not have a name. If None, then no default name is created.
- **indices** (*list[int] | None*) – The indices used for columns that do not have a name. If `None`, then the indices are `list(range(self.shape[1]))`.

Returns
Column names.

Return type
list[Optional[str]]

abstract matvec(*other*, *cols=None*, *out=None*)

Perform: self[:, cols] @ other[cols], so result[i] = sum_j self[i, j] other[j].

The 'cols' parameter allows restricting to a subset of the matrix without making a copy. If provided:

```
result[i] = sum_{j in cols} self[i, j] other[j].
```

If 'out' is provided, we modify 'out' in place by adding the output of this operation to it.

Parameters

- **cols** (*ndarray*)
- **out** (*ndarray*)

abstract sandwich(*d*, *rows=None*, *cols=None*)

Perform a sandwich product: (self[rows, cols].T * d[rows]) @ self[rows, cols].

The rows and cols parameters allow restricting to a subset of the matrix without making a copy.

Parameters

- **d** (*ndarray*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)

Return type
ndarray

set_names(*names*, *type='column'*)

Set column names.

Parameters

- **names** (*list[Optional[str]]*) – Names to set.
- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).

standardize(*weights*, *center_predictors*, *scale_predictors*)

Return a StandardizedMatrix along with the column means and column standard deviations.

It is often useful to modify a dataset so that each column has mean zero and standard deviation one. This function does this "standardization" without modifying the underlying dataset by storing shifting and scaling factors that are then used whenever an operation is performed with the new StandardizedMatrix.

Note: If center_predictors is False, col_means will be zeros.

Note: If scale_predictors is False, col_stds will be None.

Parameters

- **weights** (*ndarray*)
- **center_predictors** (*bool*)
- **scale_predictors** (*bool*)

Return type
tuple[*Any*, *ndarray*, *ndarray* | None]

property term_names

Term names of the matrix.

For differences between column names and term names, see `get_names`.

abstract toarray()

Convert self into an np.ndarray.

Return type
ndarray

abstract transpose_matvec(*vec*, *rows=None*, *cols=None*, *out=None*)

Perform: self[rows, cols].T @ vec[rows], so result[i] = sum_j self[j, i] vec[j].

The rows and cols parameters allow restricting to a subset of the matrix without making a copy.

If 'rows' and 'cols' are provided:

```
result[i] = sum_{j in rows} self[j, cols[i]] vec[j].
```

Note that the length of the output is len(cols).

If `out` is provided:

```
out[cols[i]] += sum_{j in rows} self[j, cols[i]] vec[j]
```

Parameters

- **vec** (*ndarray | list*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

class tabmat.**DenseMatrix**(*input_array*, *column_names=None*, *term_names=None*)

Bases: *MatrixBase*

A `numpy.ndarray` subclass with several additional functions that allow it to share the MatrixBase API with SparseMatrix and CategoricalMatrix.

In particular, we have added:

- The `sandwich` product
- `getcol` to support the same interface as SparseMatrix for retrieving a single column
- `toarray`
- `matvec`

property T

Returns a view of the array with axes transposed.

astype(*dtype*, *order='K'*, *casting='unsafe'*, *copy=True*)

Copy of the array, cast to a specified type.

property dtype

Data-type of the array's elements.

get_names(*type='column'*, *missing_prefix=None*, *indices=None*)

Get column names.

For columns that do not have a name, a default name is created using the following pattern: `"{missing_prefix}{start_index + i}"` where `i` is the index of the column.

Parameters

- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).
- **missing_prefix** (*Optional[str], default None*) – Prefix to use for columns that do not have a name. If None, then no default name is created.
- **indices** (*list[int] | None*) – The indices used for columns that do not have a name. If `None`, then the indices are `list(range(self.shape[1]))`.

Returns
Column names.

Return type
list[Optional[str]]

`getcol`(*i*)

Return matrix column at specified index.

`matvec`(*vec*, *cols=None*, *out=None*)

Perform self[:, cols] @ other[cols].

Parameters

- **vec** (*ndarray | list*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

`multiply`(*other*)

Element-wise multiplication.

This assumes that `other` is a vector of size `self.shape[0]`.

`property ndim`

Number of array dimensions.

`sandwich`(*d*, *rows=None*, *cols=None*)

Perform a sandwich product: X.T @ diag(d) @ X.

Parameters

- **d** (*ndarray*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)

Return type
ndarray

`set_names`(*names*, *type='column'*)

Set column names.

Parameters

- **names** (*list[Optional[str]]*) – Names to set.
- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).

`property shape`

Tuple of array dimensions.

`toarray`()

Return array representation of matrix.

`transpose()`

Returns a view of the array with axes transposed.

`transpose_matvec(vec, rows=None, cols=None, out=None)`

Perform: self[rows, cols].T @ vec[rows].

Parameters

- **vec** (*ndarray | list*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

`unpack()`

Return the underlying numpy.ndarray.

`class tabmat.SparseMatrix(input_array, shape=None, dtype=None, copy=False, column_names=None, term_names=None)`

Bases: *MatrixBase*

A scipy.sparse csc matrix subclass that allows such objects to conform to the `MatrixBase` interface.

SparseMatrix is instantiated in the same way as scipy.sparse.csc_matrix.

Parameters

- **shape** (*tuple[int, int]*)
- **dtype** (*dtype*)

`property T`

Returns a view of the array with axes transposed.

`property array_csc`

Return the CSC representation of the matrix.

`property array_csr`

Cache the CSR representation of the matrix.

`astype(dtype, order='K', casting='unsafe', copy=True)`

Return SparseMatrix cast to new type.

`property data`

Data of the matrix.

`dot(other)`

Return the dot product as a scipy sparse matrix.

`property dtype`

Data-type of the array's elements.

`get_names(type='column', missing_prefix=None, indices=None)`

Get column names.

For columns that do not have a name, a default name is created using the following pattern: `"{missing_prefix}{start_index + i}"` where `i` is the index of the column.

Parameters

- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).
- **missing_prefix** (*Optional[str], default None*) – Prefix to use for columns that do not have a name. If None, then no default name is created.
- **indices** (*list[int] | None*) – The indices used for columns that do not have a name. If `None`, then the indices are `list(range(self.shape[1]))`.

Returns
Column names.

Return type
list[Optional[str]]

getcol(*i*)

Return matrix column at specified index.

property indices

Indices of the matrix.

property indptr

Indptr of the matrix.

matvec(*vec*, *cols=None*, *out=None*)

Perform self[:, cols] @ other[cols].

Parameters

- **cols** (*ndarray*)
- **out** (*ndarray*)

multiply(*other*)

Element-wise multiplication.

See `scipy.sparse.csc_matrix.multiply`. The method is taken almost directly from the parent class except that `other` is assumed to be a vector of size `self.shape[0]`.

property ndim

Number of array dimensions.

sandwich(*d*, *rows=None*, *cols=None*)

Perform a sandwich product: X.T @ diag(d) @ X.

Parameters

- **d** (*ndarray*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)

Return type
ndarray

sandwich_dense(*B*, *d*, *rows*, *L_cols*, *R_cols*)

Perform a sandwich product: self.T @ diag(d) @ B.

Parameters

- **B** (*ndarray*)
- **d** (*ndarray*)
- **rows** (*ndarray*)
- **L_cols** (*ndarray*)
- **R_cols** (*ndarray*)

Return type
ndarray

set_names(*names*, *type='column'*)

Set column names.

Parameters

- **names** (*list[Optional[str]]*) – Names to set.
- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).

property shape

Tuple of array dimensions.

toarray()

Return a dense ndarray representation of the matrix.

tocsc(*copy=False*)

Return the matrix in CSC format.

transpose()

Returns a view of the array with axes transposed.

transpose_matvec(*vec*, *rows=None*, *cols=None*, *out=None*)

Perform: self[rows, cols].T @ vec[rows].

Parameters

- **vec** (*ndarray | list*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

unpack()

Return the underlying scipy.sparse.csc_matrix.

class tabmat.**CategoricalMatrix**(*cat_vec*, *drop_first=False*, *dtype=<class 'numpy.float64'>*, *column_name=None*, *term_name=None*, *column_name_format='{name}[{category}]'*, *cat_missing_method='fail'*, *cat_missing_name='(MISSING)'*)

Bases: *MatrixBase*

A faster, more memory efficient sparse matrix adapted to the specific settings of a one-hot encoded categorical variable.

Parameters

- **cat_vec** (*list | ndarray | Categorical*) – array-like vector of categorical data.
- **drop_first** (*bool*) – drop the first level of the dummy encoding. This allows a CategoricalMatrix to be used in an unregularized setting.
- **cat_missing_method** (*str {'fail'|'zero'|'convert'}, default 'fail'*) –
  - if 'fail', raise an error if there are missing values.
  - if 'zero', missing values will represent all-zero indicator columns.
  - if 'convert', missing values will be converted to the `cat_missing_name` category.
- **cat_missing_name** (*str, default '(MISSING)'*) – Name of the category to which missing values will be converted if `cat_missing_method='convert'`. If this category already exists, an error will be raised.
- **dtype** (*numpy.dtype*) – data type
- **column_name** (*str | None*)
- **term_name** (*str | None*)
- **column_name_format** (*str*)

astype(*dtype*, *order='K'*, *casting='unsafe'*, *copy=True*)

Return CategoricalMatrix cast to new type.

get_names(*type='column'*, *missing_prefix=None*, *indices=None*)

Get column names.

For columns that do not have a name, a default name is created using the following pattern: `"{missing_prefix}{start_index + i}"` where `i` is the index of the column.

Parameters

- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).
- **missing_prefix** (*Optional[str], default None*) – Prefix to use for columns that do not have a name. If None, then no default name is created.
- **indices** (*list[int] | None*) – The indices used for columns that do not have a name. If `None`, then the indices are `list(range(self.shape[1]))`.

Returns
Column names.

Return type
list[Optional[str]]

getcol(*i*)

Return matrix column at specified index.

Parameters
i (*int*)

Return type
SparseMatrix

matvec(*other*, *cols=None*, *out=None*)

Multiply self with vector 'other', and add vector 'out' if it is present.

out[i] += sum_j mat[i, j] other[j] = other[mat.indices[i]]

The cols parameter allows restricting to a subset of the matrix without making a copy.

If out is None, then a new array will be returned.

Test: test_matrices::test_matvec

Parameters

- **other** (*list | ndarray*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

multiply(*other*)

Element-wise multiplication.

This assumes that `other` is a vector of size `self.shape[0]`.

Return type
SparseMatrix

recover_orig()

Return 1d numpy array with same data as what was initially fed to __init__.

Test: matrix/test_categorical_matrix::test_recover_orig

Return type
ndarray

sandwich(*d*, *rows=None*, *cols=None*)

Perform a sandwich product: X.T @ diag(d) @ X.

```
sandwich(self, d)[i, j] = (self.T @ diag(d) @ self)[i, j]
    = sum_k (self[k, i] (diag(d) @ self)[k, j])
    = sum_k self[k, i] sum_m diag(d)[k, m] self[m, j]
    = sum_k self[k, i] d[k] self[k, j]
    = 0 if i != j
sandwich(self, d)[i, i] = sum_k self[k, i] ** 2 * d(k)
```

The rows and cols parameters allow restricting to a subset of the matrix without making a copy.

Parameters

- **d** (*ndarray | list*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)

Return type
dia_matrix

set_names(*names*, *type='column'*)

Set column names.

Parameters

- **names** (*list[Optional[str]]*) – Names to set.
- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).

`to_sparse_matrix()`

Return a tabmat.SparseMatrix representation.

`toarray()`

Return array representation of matrix.

Return type
ndarray

`tocsr()`

Return scipy csr representation of matrix.

Return type
csr_matrix

`transpose_matvec`(*vec*, *rows=None*, *cols=None*, *out=None*)

Perform: self[rows, cols].T @ vec[rows].

```
for i in cols: out[i] += sum_{j in rows} self[j, i] vec[j]
self[j, i] = 1(indices[j] == i)


for j in rows:
    for i in cols:
        out[i] += (indices[j] == i) * vec[j]
```

If cols == range(self.shape[1]), then for every row j, there will be exactly
one relevant column, so you can do

```
for j in rows,
    out[indices[j]] += vec[j]
```

The rows and cols parameters allow restricting to a subset of the matrix without making a copy.

If out is None, then a new array will be returned.

Test: tests/test_matrices::test_transpose_matvec

Parameters

- **vec** (*ndarray | list*)
- **rows** (*ndarray | None*)
- **cols** (*ndarray | None*)
- **out** (*ndarray | None*)

Return type
ndarray

unpack()

Return the underlying pandas.Categorical.

class tabmat.**SplitMatrix**(*matrices*, *indices=None*)

Bases: *MatrixBase*

A class for matrices with sparse, dense and categorical parts.

For real-world tabular data, it's common for the same dataset to contain a mix of columns that are naturally dense, naturally sparse and naturally categorical. Representing each of these sets of columns in the format that is most natural allows for a significant speedup in matrix multiplications compared to representations that are entirely dense or entirely sparse.

Initialize a SplitMatrix directly with a list of `matrices` and a list of column `indices` for each matrix. Most of the time, it will be best to use *tabmat.from_pandas()* or *tabmat.from_csc()* to initialize a `SplitMatrix`.

Parameters

- **matrices** (*Sequence[MatrixBase]*) – The sub-matrices composing the columns of this SplitMatrix.
- **indices** (*list[ndarray] | None*) – If `indices` is not None, then for each matrix passed in `matrices`, `indices` must contain the set of columns which that matrix covers.

astype(*dtype*, *order='K'*, *casting='unsafe'*, *copy=True*)

Return SplitMatrix cast to new type.

get_names(*type='column'*, *missing_prefix=None*, *indices=None*)

Get column names.

For columns that do not have a name, a default name is created using the following pattern: `"{missing_prefix}{start_index + i}"` where `i` is the index of the column.

Parameters

- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).
- **missing_prefix** (*Optional[str], default None*) – Prefix to use for columns that do not have a name. If None, then no default name is created.
- **indices** (*list[int] | None*) – The indices used for columns that do not have a name. If `None`, then the indices are `list(range(self.shape[1]))`.

Returns
Column names.

Return type
list[Optional[str]]

getcol(*i*)

Return matrix column at specified index.

Parameters
i (*int*)

Return type
ndarray | *csr_matrix*

matvec(*v*, *cols=None*, *out=None*)

Perform self[:, cols] @ other[cols].

Parameters

- **v** (*ndarray*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

multiply(*other*)

Element-wise multiplication.

This assumes that `other` is a vector of size `self.shape[0]`.

sandwich(*d*, *rows=None*, *cols=None*)

Perform a sandwich product: X.T @ diag(d) @ X.

Parameters

- **d** (*ndarray | list*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)

Return type
ndarray

set_names(*names*, *type='column'*)

Set column names.

Parameters

- **names** (*list[Optional[str]]*) – Names to set.
- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).

toarray()

Return array representation of matrix.

Return type
ndarray

transpose_matvec(*v*, *rows=None*, *cols=None*, *out=None*)

Perform: self[rows, cols].T @ vec[rows].

```
self.transpose_matvec(v, rows, cols) = self[rows, cols].T @ v[rows]
self.transpose_matvec(v, rows, cols)[i]
    = sum_{j in rows} self[j, cols[i]] v[j]
    = sum_{j in rows} sum_{mat in self.matrices} 1(cols[i] in mat)
                                                 self[j, cols[i]] v[j]
```

Parameters

- **v** (*ndarray | list*)

- **rows** (*ndarray*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

class tabmat.**StandardizedMatrix**(*mat*, *shift*, *mult=None*)

Bases: `object`

StandardizedMatrix allows for storing a matrix standardized to have columns that have mean zero and standard deviation one without modifying underlying sparse matrices.

To be precise, for a StandardizedMatrix:

```
self[i, j] = (self.mult[j] * self.mat[i, j]) + self.shift[j]
```

This class is returned from *MatrixBase.standardize*.

Parameters

- **mat** (MatrixBase)
- **shift** (*ndarray | list*)
- **mult** (*ndarray | list*)

property A: ndarray

Return array representation of self.

astype(*dtype*, *order='K'*, *casting='unsafe'*, *copy=True*)

Return StandardizedMatrix cast to new type.

property column_names

Column names of the matrix.

get_names(*type='column'*, *missing_prefix=None*, *indices=None*)

Get column names.

For columns that do not have a name, a default name is created using the following pattern: `"{missing_prefix}{start_index + i}"` where `i` is the index of the column.

Parameters

- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).
- **missing_prefix** (*Optional[str], default None*) – Prefix to use for columns that do not have a name. If None, then no default name is created.
- **indices** (*list[int] | None*) – The indices used for columns that do not have a name. If `None`, then the indices are `list(range(self.shape[1]))`.

Returns
Column names.

Return type
list[Optional[str]]

getcol(*i*)

Return matrix column at specified index.

Returns a StandardizedMatrix.

```
>>> from scipy import sparse as sps
>>> x = StandardizedMatrix(SparseMatrix(sps.eye(3).tocsc()), shift=[0, 1, -2])
>>> col_1 = x.getcol(1)
>>> isinstance(col_1, StandardizedMatrix)
True
>>> col_1.A
array([[1.],
       [2.],
       [1.]])
```

Parameters
i (*int*)

matvec(*other_mat*, *cols=None*, *out=None*)

Perform self[:, cols] @ other[cols].

This function returns a dense output, so it is best geared for the matrix-vector case.

Parameters

- **other_mat** (*ndarray | list*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

multiply(*other*)

Element-wise multiplication.

Note that the output of this function is always a DenseMatrix and might require a lot more memory. This assumes that `other` is a vector of size `self.shape[0]`.

Return type
DenseMatrix

sandwich(*d*, *rows=None*, *cols=None*)

Perform a sandwich product: X.T @ diag(d) @ X.

Parameters

- **d** (*ndarray*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)

Return type
ndarray

set_names(*names*, *type='column'*)

Set column names.

Parameters

- **names** (*list[Optional[str]]*) – Names to set.
- **type** (*str {'column'|'term'}*) – Whether to get column names or term names. The main difference is that a categorical submatrix counts as one term, but can count as multiple columns. Furthermore, matrices created from formulas distinguish between columns and terms (c.f. `formulaic` docs).

property term_names

Term names of the matrix.

For differences between column names and term names, see `get_names`.

toarray()

Return array representation of matrix.

Return type
ndarray

transpose_matvec(*other*, *rows=None*, *cols=None*, *out=None*)

Perform: self[rows, cols].T @ vec[rows].

Let self.shape = (N, K) and other.shape = (M, N). Let shift_mat = outer(ones(N), shift)

(X.T @ other)[k, i] = (X.mat.T @ other)[k, i] + (shift_mat @ other)[k, i] (shift_mat @ other)[k, i] = (outer(shift, ones(N)) @ other)[k, i] = sum_j outer(shift, ones(N))[k, j] other[j, i] = sum_j shift[k] other[j, i] = shift[k] other.sum(0)[i] = outer(shift, other.sum(0))[k, i]

With row and col restrictions:

self.transpose_matvec(other, rows, cols)[i, j]

= self.mat.transpose_matvec(other, rows, cols)[i, j]

- (outer(self.shift, ones(N))[rows, cols] @ other[cols])

= self.mat.transpose_matvec(other, rows, cols)[i, j]

- shift[cols[i]] other.sum(0)[rows[j]

Parameters

- **other** (*ndarray | list*)
- **rows** (*ndarray*)
- **cols** (*ndarray*)
- **out** (*ndarray*)

Return type
ndarray

unstandardize()

Get unstandardized (base) matrix.

Return type
MatrixBase

CHAPTER

THREE

CHANGELOG

3.1 4.0.0 - 2024-04-23

Breaking changes:

- To unify the API, `DenseMatrix` does not inherit from `np.ndarray` anymore. To convert a `DenseMatrix` to a `np.ndarray`, use `DenseMatrix.unpack()`.
- Similarly, `SparseMatrix` does not inherit from `sps.csc_matrix` anymore. To convert a `SparseMatrix` to a `sps.csc_matrix`, use `SparseMatrix.unpack()`.

New features:

- Added column name and term name metadata to `MatrixBase` objects. These are automatically populated when initializing a `MatrixBase` from a `pandas.DataFrame`. In addition, they can be accessed and modified via the `MatrixBase.column_names` and `MatrixBase.term_names` properties.
- Added a formula interface for creating tabmat matrices from pandas data frames. See `tabmat.from_formula()` for details.
- Added support for missing values in `CategoricalMatrix` by either creating a separate category for them or treating them as all-zero rows.
- Added support for handling missing categorical values in pandas data frames.

Bug fix:

- Added cython compiler directive `legacy_implicit_noexcept = True` to fix performance regression with cython 3.

Other changes:

- Refactored the pre-commit hooks to use ruff.
- Refactored `CategoricalMatrix.transpose_matvec()` to be deterministic when using OpenMP.
- Adjusted transformation to sparse format in *`tabmat.from_pandas()`* to future changes in pandas.

3.2 3.1.13 - 2023-10-17

Other changes:

- Pypi release is now done using trusted publisher.
- Fix build and upload of `x86_64` wheels on Linux.

3.3 3.1.12 - 2023-10-16

Other changes:

- Fixed macos arm64 wheels with proper linkage.

3.4 3.1.11 - 2023-10-13

Other changes:

- Improve the performance of `from_pandas` in the case of low-cardinality categorical variables.
- Require Python>=3.9 in line with NEP 29
- Build and test with Python 3.12 in CI.
- Fixed macos arm64 wheels with proper linkage.

3.5 3.1.10 - 2023-06-23

Bug fixes:

- We fixed a bug in the dense sandwich product, which would previously segfault for very large matrices.
- Fixed the column order when initializing a `SplitMatrix` from a list containing other `SplitMatrix` objects.
- Fixed `getcol` not respecting the `drop_first` attribute of a `CategoricalMatrix`.

3.6 3.1.9 - 2023-06-16

Other changes:

- Support building on architectures that are unsupported by xsimd.

3.7 3.1.8 - 2023-06-13

Other changes:

- The C++ types have been refactored. Loop indices are now using the `Py_ssize_t` type. Integers now have a templated type as well.
- The documentation for `matvec` and `matvec_transpose` has been updated to reflect actual behavior.
- Checks for dimension mismatch in `matvec` and `matvec_transpose` arguments have been added.
- Remove upper pin on xsimd.

3.8 3.1.7 - 2022-03-28

Bug fix:

- We fixed a bug in the cross sandwich product, which would previously segfault for very large matrices.

3.9 3.1.6 - 2022-03-27

Bug fix:

- We fixed a bug in the dense sandwich product, which would previously segfault for very large F-contiguous matrices.

3.10 3.1.5 - 2022-03-20

Bug fix:

- We fixed a bug in the dense matrix-vector and sandwich products, which would previously segfault for very large matrices.

3.11 3.1.4 - 2022-02-07

Bug fix:

- Fixed the loading of jemalloc in Apple Silicon wheels.

3.12 3.1.3 - 2022-01-26

Other changes:

- Build and upload wheels for Apple Silicon.

3.13 3.1.2 - 2022-07-01

Other changes:

- Next attempt to build wheel for PyPI without `march=native`.

3.14 3.1.1 - 2022-07-01

Other changes:

- Add Python 3.10 support to CI (remove Python 3.6).
- Build wheel for PyPI without `march=native`.

3.15 3.1.0 - 2022-03-07

New feature

- *tabmat.CategoricalMatrix* now accepts a *drop_first* argurment. This allows the user to drop the first column of a CategoricalMatrix to avoid multicollinearity problems in unregularized models.
- *tabmat.StandardizedMatrix* and *tabmat.MatrixBase* now support the *multiply* method.

3.16 3.0.8 - 2022-01-03

Bug fix

- Always use 64bit integers for indexing in `tabmat.ext.sparse.sparse_sandwich()` to avoid segmentation faults on very wide problems.

3.17 3.0.7 - 2021-11-23

Bug fix

- Disable the use of static TLS in the Linux wheels to avoid issues with too small TLS on some distributions.

3.18 3.0.6 - 2021-11-11

Bug fix

- We fixed a bug in *tabmat.SplitMatrix.matvec()*, where incorrect matrix vector products were computed when a `SplitMatrix` did not contain any dense components.

3.19 3.0.5 - 2021-11-05

Other changes

- We are now specifying the run time dependencies in `setup.py`, so that missing dependencies are automatically installed from PyPI when installing `tabmat` via pip.

3.20 3.0.4 - 2021-11-03

Other changes

- tabmat is now available on PyPI and will be automatically updated when a new release is published.

3.21 3.0.3 - 2021-10-15

Bug fix

- We now support `xsimd>=8` and support alternative jemalloc installations.

3.22 3.0.2 - 2021-10-14

Bug fix

- Allow to link to alternatively suffixed jemalloc installation to work around #113 .

3.23 3.0.1 - 2021-10-07

Bug fix

- The license was mistakenly left as proprietary. Corrected to BSD-3-Clause.

Other changes

- ReadTheDocs integration.
- CONTRIBUTING.md
- Correct pyproject.toml to work with PEP-517

3.24 3.0.0 - 2021-10-07

Breaking changes:

- The package has been renamed to `tabmat`. CELEBRATE!
- The `one_over_var_inf_to_val()` function has been made private.
- The `csc_to_split()` function has been re-named to *tabmat.from_csc()* to match the *tabmat.from_pandas()* function.

- The `tabmat.MatrixBase.get_col_means()` and `tabmat.MatrixBase.get_col_stds()` methods have been made private.
- The `cross_sandwich()` method has also been made private.

Bug fix

- `StandardizedMatrix.transpose_matvec()` was giving the wrong answer when the *out* parameter was provided. This is now fixed.
- `SplitMatrix.__repr__()` now calls the *__repr__* method of component matrices instead of *__str__*.

Other changes

- Optimized the *tabmat.SparseMatrix.matvec()* and *tabmat.SparseMatrix.transpose_matvec()* for when `rows` and `cols` are None.
- Implemented `CategoricalMatrix.__rmul__()`
- Reorganizing the documentation and updating the text to match the current API.
- Enable indexing the rows of a `CategoricalMatrix`. Previously `CategoricalMatrix.__getitem__()` only supported column indexing.
- Allow creating a `SplitMatrix` from a list of any `MatrixBase` objects including another `SplitMatrix`.
- Reduced memory usage in *tabmat.SplitMatrix.matvec()*.

3.25 2.0.3 - 2021-07-15

Bug fix

- In `SplitMatrix.sandwich()`, when a col subset was specified, incorrect output was produced if the components of the indices array were not sorted. `SplitMatrix.__init__()` now checks for sorted indices and maintains sorted index lists when combining matrices.

Other changes

- `SplitMatrix.__init__()` now filters out any empty matrices.
- `StandardizedMatrix.sandwich()` passes `rows=None` and `cols=None` onwards to the underlying matrix instead of replacing them with full arrays of indices. This should improve performance slightly.
- `SplitMatrix.__repr__()` now includes the type of the underlying matrix objects in the string output.

3.26 2.0.2 - 2021-06-24

Bug fix

Sparse matrices now accept 64-bit indices on Windows.

3.27 2.0.1 - 2021-06-20

Bug fix:

Split matrices now also work on Windows.

3.28 2.0.0 - 2021-06-17

Breaking changes:

We renamed several public functions to make them private. These include functions in `tabmat.benchmark` that are unlikely to be used outside of this package as well as

- `tabmat.dense_matrix._matvec_helper()`
- `tabmat.sparse_matrix._matvec_helper()`.
- `tabmat.split_matrix._prepare_out_array()`.

Other changes:

- We removed the dependency on `sparse_dot_mkl`. We now use `scipy.sparse.csr_matvec()` instead of `sparse_dot_mkl.dot_product_mkl()` on all platforms, because the former suffered from poor performance, especially on narrow problems. This also means that we removed the function `tabmat.sparse_matrix._dot_product_maybe_mkl()`.
- We updated the pre-commit hooks and made sure the code is line with the new hooks.

3.29 1.0.6 - 2020-04-26

Other changes:

We are now also making releases for Windows.

3.30 1.0.5 - 2020-04-26

Other changes:

Still trying.

3.31 1.0.4 - 2020-04-26

Other changes:

We are trying to make releases for Windows.

3.32 1.0.3 - 2020-04-21

Bug fixes:

- Added a check that matrices are two-dimensional in the `SplitMatrix.__init__`
- Replace `np.int` with `np.int64` where appropriate due to NumPy deprecation of `np.int`.

3.33 1.0.2 - 2020-04-20

Other changes:

- Added Python 3.9 support.
- Use `scipy.sparse` dot product when MKL isn't available.

3.34 1.0.1 - 2020-11-25

Bug fixes:

- Handling for nulls when setting up a `CategoricalMatrix`
- Fixes to make several functions work with both row and col restrictions and out

Other changes:

- Added various tests and documentation improvements

3.35 1.0.0 - 2020-11-11

Breaking change:

- Rename *dot* to *matvec*. Our *dot* function supports matrix-vector multiplication for every subclass, but only supports matrix-matrix multiplication for some. We therefore rename it to *matvec* in line with other libraries.

Bug fix:

- Fix a bug in *matvec* for categorical components when the number of categories exceeds the number of rows.

3.36 0.0.6 - 2020-08-03

See git history.

genindex

INDEX

T

U