Nclustgen

Release 1.0.4

Pedro Cotovio

Apr 05, 2022

GETTING STARTED

Nclustgen is a python tool to generate biclustering and triclustering datasets programmatically.

It wraps two java packages G-Bic, and G-Tric, that serve as backend generators. If you are interested on a GUI version of this generator or on using this generator in a java environment check out those packages.

This tool adds some functionalities to the original packages, for a more fluid interaction with python libraries, like:

- Conversion to numpy arrays
- Conversion to sparse tensors
- Conversion to networkX or dgl n-partite graphs

CHAPTER

ONE

SOURCE CODE

The source code is available at: https://github.com/PedroCotovio/nclustgen.

1.1 Getting Started

1.1.1 Installation

This tool can be installed from PyPI:

```
$ pip install nclustgen
```

NOTICE: Nclustgen installs by default the dgl build with no cuda support, in case you want to use gpu you can override this by installing the correct dgl build, more information at: https://www.dgl.ai/pages/start.html.

1.1.2 Basic Usage

1.1.2.1 Biclustering Dataset

See also:

Detailed API at *Bicluster Generator*.

```
## Generate biclustering dataset

from nclustgen import BiclusterGenerator

# Initialize generator
generator = BiclusterGenerator(
     dstype='NUMERIC',
     patterns=[['CONSTANT', 'CONSTANT'], ['CONSTANT', 'NONE']],
     bktype='UNIFORM',
     in_memory=True,
     silence=True
)

# Get parameters
generator.get_params()

# Generate dataset
```

(continues on next page)

(continued from previous page)

```
x, y = generator.generate(nrows=50, ncols=100, nclusters=3)

# Build graph
graph = generator.to_graph(x, framework='dgl', device='cpu')

# Save data files
generator.save(file_name='example', single_file=True)
```

1.1.2.2 Triclustering Dataset

See also:

Detailed API at *Tricluster Generator*.

```
## Generate triclustering dataset

from nclustgen import TriclusterGenerator

# Initialize generator
generator = TriclusterGenerator(
     dstype='NUMERIC',
     patterns=[['CONSTANT', 'CONSTANT', 'CONSTANT'], ['CONSTANT', 'NONE', 'NONE']],
     bktype='UNIFORM',
     in_memory=True,
     silence=True
)

# Get parameters
generator.get_params()

# Generate dataset
x, y = generator.generate(nrows=50, ncols=100, ncontexts=10, nclusters=25)

# Build graph
graph = generator.to_graph(x, framework='dgl', device='cpu')

# Save data files
generator.save(file_name='example', single_file=True)
```

See also:

This is a basic example, more detail at *Generating Data*.

1.2 Generating Data

1.2.1 Generators

This tool provides data generators for bi-clustering and tri-clustering, both generators are based on *nclustgen.Generator.Generator*, a (dimensions) abstract class. A complete explanation of the parameters of the generator can be found in the API reference.

It can generate real-valued, integer, and categorical datasets, with different settings for cluster patterns, distributions, cluster overlapping, noise, missing values, and other parameters.

```
# Generate real-valued dataset

from nclustgen import BiclusterGenerator

# Initialize generator
generator = BiclusterGenerator(
    # Dataset type
    dstype='NUMERIC',
    # If real-valued
    realval=True,
    minval=1,
    maxval=10
)

x, y = generator.generate()
x

# Generate categorical dataset

from nclustgen import BiclusterGenerator

# Initialize generator
generator = BiclusterGenerator(
    # Dataset type
    dstype='SYMBOLIC',
    # Number of symbols
    nsymbols=10
)

x, y = generator.generate()
x
```

A seed argument can also be used to ensure reproducibility:

```
from nclustgen import BiclusterGenerator

generator = BiclusterGenerator(seed=3)

x, y = generator.generate()
x
```

To generate a dataset, the *nclustgen.Generator.Generator.generate()* method can be called. This method receives as input the dataset's shape and number of hidden clusters:

```
# Generate bicluster dataset

from nclustgen import BiclusterGenerator
generator = BiclusterGenerator()

x, y = generator.generate(nrows=100, ncols=20, nclusters=20)
x

# Generate tricluster dataset

from nclustgen import TriclusterGenerator
generator = TriclusterGenerator()

x, y = generator.generate(nrows=100, ncols=20, ncontexts=3, nclusters=20)
x
```

Different patterns can be used for *biclusters* or *triclusters*:

```
# Generate bicluster dataset
from nclustgen import BiclusterGenerator
generator = BiclusterGenerator(
    patterns = [['Additive', 'Constant'], ['Constant', 'Multiplicative']]
)

x, y = generator.generate()
x

# Generate tricluster dataset
from nclustgen import TriclusterGenerator
generator = TriclusterGenerator(
    patterns = [['Order_Preserving', 'None', 'None'], ['Constant', 'Constant', 'Constant
↪']]
)

x, y = generator.generate()
x
```

1.2.1.1 Biclustering Generator

The biclustering generator uses G-Bic a Java library as the backend generator, check this library if you prefer a graphical interface or to work with Java directly. More information can also be found there if you wish to modify the actual generator.

Patterns

The biclustering generator as specified earlier accepts a number of different bicluster patterns here is a complete list:

2D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order Preserving’, ‘None’]
1	[‘None’, ‘Order Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]
5	[‘Additive’, ‘Additive’]
6	[‘Constant’, ‘Additive’]
7	[‘Additive’, ‘Constant’]
8	[‘Multiplicative’, ‘Multiplicative’]
9	[‘Constant’, ‘Multiplicative’]
10	[‘Multiplicative’, ‘Constant’]

2D Symbolic Patterns Possible Combinations	
index	pattern combination
0	[‘Order Preserving’, ‘None’]
1	[‘None’, ‘Order Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]

See also:

Detailed API at *Bicluster Generator*.

1.2.1.2 Triclustering Generator

The triclustering generator similarly uses G-Tric a Java library as the backend generator.

Patterns

Like the biclustering generator, triclustering generator also accepts several different patterns:

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	['Order Preserving', 'None', 'None']
1	['None', 'Order Preserving', 'None']
2	['None', 'None', 'Order Preserving']
3	['Constant', 'Constant', 'Constant']
4	['None', 'Constant', 'Constant']
5	['Constant', 'Constant', 'None']
6	['Constant', 'None', 'Constant']
7	['Constant', 'None', 'None']
8	['None', 'Constant', 'None']
9	['None', 'None', 'Constant']
10	['Additive', 'Additive', 'Additive']
11	['Additive', 'Additive', 'Constant']
12	['Constant', 'Additive', 'Additive']
13	['Additive', 'Constant', 'Additive']
14	['Additive', 'Constant', 'Constant']
15	['Constant', 'Additive', 'Constant']
16	['Constant', 'Constant', 'Additive']
17	['Multiplicative', 'Multiplicative', 'Multiplicative']
18	['Multiplicative', 'Multiplicative', 'Constant']
19	['Constant', 'Multiplicative', 'Multiplicative']
20	['Multiplicative', 'Constant', 'Multiplicative']
21	['Multiplicative', 'Constant', 'Constant']
22	['Constant', 'Multiplicative', 'Constant']
23	['Constant', 'Constant', 'Multiplicative']

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	['Order Preserving', 'None', 'None']
1	['None', 'Order Preserving', 'None']
2	['None', 'None', 'Order Preserving']
3	['Constant', 'Constant', 'Constant']
4	['None', 'Constant', 'Constant']
5	['Constant', 'Constant', 'None']
6	['Constant', 'None', 'Constant']
7	['Constant', 'None', 'None']
8	['None', 'Constant', 'None']
9	['None', 'None', 'Constant']

See also:

Detailed API at *Tricluster Generator*.

1.2.2 Dense Tensors

If *nclustgen.Generator.Generator.in_memory* is True, then a dense tensor will be generated, in this case numpy is used. If you are not familiar with numpy follow this link to learn more about it: https://numpy.org/doc/stable/user/quickstart.html

```
>>> from nclustgen import BiclusterGenerator
>>> generator = BiclusterGenerator(in_memory=True)
>>> x, y = generator.generate()
>>> type(x)
<class 'numpy.ndarray'>
```

1.2.2.1 Matrix

When the generator's output is a dense matrix, it will be of shape *(nrows, ncols)*

```
>>> from nclustgen import BiclusterGenerator
>>> generator = BiclusterGenerator(in_memory=True)
>>> x, y = generator.generate(nrows=100, ncols=50)
>>> x.shape
(100, 50)
```

1.2.2.2 Tensor

On the other hand, when the generator's output is a dense tensor, it will be of shape *(ncontext, nrows, ncols)*

```
>>> from nclustgen import TriclusterGenerator
>>> generator = TriclusterGenerator(in_memory=True)
>>> x, y = generator.generate(nrows=100, ncols=50, ncontexts=30)
>>> x.shape
(30, 100, 50)
```

1.2.3 Sparse Tensors

If *nclustgen.Generator.Generator.in_memory* parameter is False, then a sparse tensor will be generated, in this case different packages are used depending on the dimensionality of the dataset. But the shape follows the standard set by the dense option.

1.2.3.1 Matrix

When the generator's output is a sparse matrix, scipy's csr_matrix will be used.

```
>>> from nclustgen import BiclusterGenerator
>>> generator = BiclusterGenerator(in_memory=False)
>>> x, y = generator.generate()
>>> type(x)
<class 'scipy.sparse.csr.csr_matrix'>
```

1.2.3.2 Tensor

On the other hand, when the generator's output is a sparse tensor a sparse's COO object will be outputted.

```
>>> from nclustgen import TriclusterGenerator
>>> generator = TriclusterGenerator(in_memory=False)
>>> x, y = generator.generate()
>>> type(x)
<class 'sparse._coo.core.COO'>
```

1.2.4 Graphs

The *nclustgen.Generator.Generator.to_graph()* method allows for either a bipartite or tripartite graph to be generated, depending on the datasets dimension.

The datasets shape will be transformed in the following way:

number of nodes = *nrows* + *ncols (+ ncontexts)*

number of edges = *nrows* * *ncols (* ncontexts * 3)*

The graphs can be outputted in two different formats as a NetworkX Multigraph, or as a DGL heterograph with a pytorch backend.

The networkX is a very well known framework to deal with graph data, while DGL is a more recent library mainly for deep learning with graphs, so if you intend to use this data for deep learning models DGL is recommended, otherwise, networkX will probably be a better option.

```
>>> from nclustgen import BiclusterGenerator
>>> generator = BiclusterGenerator()
>>> x, y = generator.generate(100, 50)
>>> g = generator.to_graph(framework='dgl')
>>> g
<networkx.classes.graph.Graph object at 0x10a011d60>
>>> len(g.nodes) == 100 + 50
True
>>> len(g.edges) == 100 * 50
True
>>> g = generator.to_graph(framework='dgl')
>>> g
Graph(num_nodes={'col': 50, 'row': 100},
      num_edges={('row', 'elem', 'col'): 5000},
      metagraph=[('row', 'col', 'elem')])
>>> g.num_nodes() == 100 + 50
True
>>> g.num_edges() == 100 * 50
True
```

In case dgl framework is being used the *nclustgen.Generator.Generator.to_graph()* method can also receive two additional parameters, the *device* and *cuda* parameters. The first determines if the tensors are stored in cpu or gpu memory, the second is only used for gpu devices and sets the index of the gpu device to be used in multi-gpu machines if that's not the case ignore it as it defaults to 0.

```
>>> g = generator.to_graph(framework='dgl', device='gpu', cuda=0)
>>> g.device
device(type='gpu')
```

1.3 Bicluster Generator

See also:

This module inherits from *Generator*, check it out for more info.

1.3.1 Bicluster Generator

class nclustgen.BiclusterGen.**BiclusterGenerator**(**args*, ***kwargs*)

Bases: *nclustgen.Generator.Generator*

This class provides an implementation for two-dimensional datasets with hidden biclusters.

Examples

```
>>> from nclustgen import BiclusterGenerator
>>> generator = BiclusterGenerator(
...     dstype='NUMERIC',
...     patterns=[['CONSTANT', 'CONSTANT'], ['CONSTANT', 'NONE']],
...     bktype='UNIFORM',
...     in_memory=True,
...     silence=True
... )
>>> generator.get_params()
{'X': None, 'Y': None, 'background': ['UNIFORM'], 'clusterdistribution': [['UNIFORM
↪', 4, 4], ['UNIFORM', 4, 4]],
'contiguity': 'NONE', 'dstype': 'NUMERIC', 'errors': (0.0, 0.0, 0.0),
↪'generatedDataset': None, 'graph': None,
'in_memory': 'True', 'maxclustsperoverlappedarea': 0, 'maxpercofoverlappingelements
↪': 0.0, 'maxval': 10.0,
'minval': -10.0, 'missing': (0.0, 0.0), 'cuda': 2, 'noise': (0.0, 0.0, 0.0),
'patterns': [['CONSTANT', 'CONSTANT'], ['CONSTANT', 'NONE']],
↪'percofoverlappingclusts': 0.0,
'percofoverlappingcolumns': 1.0, 'percofoverlappingcontexts': 1.0,
↪'percofoverlappingrows': 1.0,
'plaidcoherency': 'NO_OVERLAPPING', 'realval': True, 'seed': -1, 'silenced': True,
↪'time_profile': None}
>>> x, y = generator.generate(nrows=50, ncols=100, nclusters=3)
>>> x
array([[-4.43, -8.2 , -0.34, ...,  8.85,  9.24,  6.13],
       [ 9.28,  9.45,  5.46, ...,  7.83,  8.67, -6.48],
       [-9.97, -2.14, -6.58, ...,  1.23,  5.64, -7.29],
       ...,
       [-5.12,  1.11, -3.44, ..., -7.45, -0.21,  2.21],
       [-0.96,  5.43, -3.28, ...,  9.58, -0.73,  3.99],
       [-0.75,  8.91, -6.91, ..., -9.22,  0.43, -4.46]])
>>> y
```

(continues on next page)

(continued from previous page)

```
[[[1, 9, 37, 46], [13, 25, 32, 79]], [[17, 29, 39, 46], [0, 5, 74, 90]], [[21, 30,␣
↪39, 42], [8, 46, 60, 93]]]
>>> graph = generator.to_graph(x, framework='dgl', device='cpu')
>>> graph
Graph(num_nodes={'col': 100, 'row': 50},
      num_edges={('row', 'elem', 'col'): 5000},
      metagraph=[('row', 'col', 'elem')])
>>> generator.save(file_name='example', single_file=True)
```

Parameters

- **n** (*int, internal*) – Determines dimensionality (e.g. Bi/Tri clustering). Should only be used by subclasses.
- **dstype** (*{'NUMERIC', 'SYMBOLIC'}, default 'Numeric'*) – Type of Dataset to be generated, numeric or symbolic(categorical).
- **patterns** (*list or array, default [['CONSTANT', 'CONSTANT']]*) – Defines the type of patterns that will be hidden in the data.

  Shape: (number of patterns, number of dimensions)

  Patterns_Set: {CONSTANT, ADDITIVE, MULTIPLICATIVE, ORDER_PRESERVING, NONE}

  Numeric_Patterns_Set: {CONSTANT, ADDITIVE, MULTIPLICATIVE, ORDER_PRESERVING, NONE}

  Symbolic_Patterns_Set: {CONSTANT, ORDER_PRESERVING, NONE}

  Pattern_Combinations:

2D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’]
1	[‘None’, ‘Order_Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]
5	[‘Additive’, ‘Additive’]
6	[‘Constant’, ‘Additive’]
7	[‘Additive’, ‘Constant’]
8	[‘Multiplicative’, ‘Multiplicative’]
9	[‘Constant’, ‘Multiplicative’]
10	[‘Multiplicative’, ‘Constant’]

2D Symbolic Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’]
1	[‘None’, ‘Order_Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’, ‘None’]
1	[‘None’, ‘Order_Preserving’, ‘None’]
2	[‘None’, ‘None’, ‘Order_Preserving’]
3	[‘Constant’, ‘Constant’, ‘Constant’]
4	[‘None’, ‘Constant’, ‘Constant’]
5	[‘Constant’, ‘Constant’, ‘None’]
6	[‘Constant’, ‘None’, ‘Constant’]
7	[‘Constant’, ‘None’, ‘None’]
8	[‘None’, ‘Constant’, ‘None’]
9	[‘None’, ‘None’, ‘Constant’]
10	[‘Additive’, ‘Additive’, ‘Additive’]
11	[‘Additive’, ‘Additive’, ‘Constant’]
12	[‘Constant’, ‘Additive’, ‘Additive’]
13	[‘Additive’, ‘Constant’, ‘Additive’]
14	[‘Additive’, ‘Constant’, ‘Constant’]
15	[‘Constant’, ‘Additive’, ‘Constant’]
16	[‘Constant’, ‘Constant’, ‘Additive’]
17	[‘Multiplicative’, ‘Multiplicative’, ‘Multiplicative’]
18	[‘Multiplicative’, ‘Multiplicative’, ‘Constant’]
19	[‘Constant’, ‘Multiplicative’, ‘Multiplicative’]
20	[‘Multiplicative’, ‘Constant’, ‘Multiplicative’]
21	[‘Multiplicative’, ‘Constant’, ‘Constant’]
22	[‘Constant’, ‘Multiplicative’, ‘Constant’]
23	[‘Constant’, ‘Constant’, ‘Multiplicative’]

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’, ‘None’]
1	[‘None’, ‘Order_Preserving’, ‘None’]
2	[‘None’, ‘None’, ‘Order_Preserving’]
3	[‘Constant’, ‘Constant’, ‘Constant’]
4	[‘None’, ‘Constant’, ‘Constant’]
5	[‘Constant’, ‘Constant’, ‘None’]
6	[‘Constant’, ‘None’, ‘Constant’]
7	[‘Constant’, ‘None’, ‘None’]
8	[‘None’, ‘Constant’, ‘None’]
9	[‘None’, ‘None’, ‘Constant’]

- **bktype** (*{'NORMAL', 'UNIFORM', 'DISCRETE', 'MISSING'}, default 'UNIFORM'*) – Determines the distribution used to generate the background values.
- **clusterdistribution** (*list or array, default [['UNIFORM', 4.0, 4.0], ['UNIFORM', 4.0, 4.0]]*) – Distribution used to calculate the size of a cluster.

  Shape: number of dimensions, 3 -> param1(str), param2(float), param3(float)

  The first parameter(param1) is always the type of distribution {‘NORMAL’, ‘UNIFORM’}. If param1==UNIFORM, then param2 and param3 represents the min and max, respectively. If param1==NORMAL, then param2 and param3 represents the mean and standard deviation, respectively.

- **contiguity** (*{'COLUMNS', 'CONTEXTS', 'NONE'}, default None*) – Contiguity can occur on COLUMNS or CONTEXTS. To avoid contiguity use None.

  If dimensionality == 2 and contiguity == 'CONTEXTS' it defaults to None.

- **plaidcoherency** (*{'ADDITIVE', 'MULTIPLICATIVE', 'INTERPOLED', 'NONE', 'NO_OVERLAPPING'}, default 'NO_OVERLAPPING'*) – Enforces the type of plaid coherency. To avoid plaid coherency use NONE, to avoid any overlapping use 'NO_OVERLAPPING'.

- **percofoverlappingclusters** (*float, default 0.0*) – Percentage of overlapping clusters. Defines how many clusters are allowed to overlap.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **maxclustsperoverlappedarea** (*int, default 0*) – Maximum number of clusters overlapped per area. Maximum number of clusters that can overlap together.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0, nclusters]

- **maxpercofoverlappingelements** (*float, default 0.0*) – Maximum percentage of values shared by overlapped clusters.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **percofoverlappingrows** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters rows.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **percofoverlappingcolumns** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters columns.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **percofoverlappingcontexts** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters contexts.

  Not used if plaidcoherency == 'NO_OVERLAPPING' or cuda >= 3.

  Range: [0,1]

- **percmissingsonbackground** (*float, 0.0*) – Percentage of missing values on the background, that is, values that do not belong to planted clusters.

  Range: [0,1]

- **percmissingsonclusters** (*float, 0.0*) – Maximum percentage of missing values on each cluster.

  Range: [0,1]

- **percnoiseonbackground** (*float, 0.0*) – Percentage of noisy values on background, that is, values with added noise.

  Range: [0,1]

- **percnoiseonclusters** (*float, 0.0*) – Maximum percentage of noisy values on each cluster.

  Range: [0,1]

- **percnoisedeviation** (*int or float, 0.0*) – Percentage of symbol on noisy values deviation, that is, the maximum difference between the current symbol on the matrix and the one that will replaced it to be considered noise.

  If dstype == Numeric then percnoisedeviation -> float else int.

  Ex: Let Alphabet = [1,2,3,4,5] and CurrentSymbol = 3, if the noiseDeviation is '1', then CurrentSymbol will be, randomly, replaced by either '2' or '4'. If noiseDeviation is '2', CurrentSymbol can be replaced by either '1','2','4' or '5'.

- **percerroesonbackground** (*float, 0.0*) – Percentage of error values on background. Similar as noise, a new value is considered an error if the difference between it and the current value in the matrix is greater than noiseDeviation.

  Ex: Alphabet = [1,2,3,4,5], If currentValue = 2, and errorDeviation = 2, to turn currentValue an error, it's value must be replaced by '5', that is the only possible value that respects abs(currentValue - newValue) > noiseDeviation

  Range: [0,1]

- **percerrorsonclusters** (*float, 0.0*) – Percentage of errors values on background. Similar as noise, a new value is considered an error if the difference between it and the current value in the matrix is greater than noiseDeviation.

  Ex: Alphabet = [1,2,3,4,5], If currentValue = 2, and errorDeviation = 2, to turn currentValue an error, it's value must be replaced by '5', that is the only possible value that respects abs(currentValue - newValue) > noiseDeviation

  Range: [0,1]

- **percerrorondeviation** (*int or float, 0.0*) – Percentage of symbol on error values deviation, that is, the maximum difference between the current symbol on the matrix and the one that will replaced it to be considered error.

  If dstype == Numeric then percnoisedeviation -> float else int.

- **silence** (*bool, default False*) – If True them the class does not print to the console.

- **seed** (*int, default -1*) – Seed to initialize random objects.

  If seed is None or -1 then random objects are initialized without a seed.

- **timeprofile** (*{'RANDOM', 'MONONICALLY_INCREASING', 'MONONICALLY_DECREASING', None}, default None*) – It determines a time profile for the ORDER_PRESERVING pattern. Only used if ORDER_PRESERVING in patterns.

  If None and ORDER_PRESERVING in patterns it defaults to 'RANDOM'.

- **realval** (*bool, default True*) – Indicates if the dataset is real valued. Only used when dstype == 'NUMERIC'.

- **minval** (*int or float, default -10.0*) – Dataset's minimum value. Only used when dstype == 'NUMERIC'.

- **maxval** (*int or float, default 10.0*) – Dataset's maximum value. Only used when dstype == 'NUMERIC'.

- **symbols** (*list or array of strings, default None*) – Dataset's alphabet (list of possible values/symbols it can contain). Only used if dstype == 'SYMBOLIC'.

  Shape: alphabets length

- **nsymbols** (*int, default 10*) – Defines the length of the alphabet, instead of defining specific symbols this parameter can be passed, and a list of strings will be create with range(1, cuda), where cuda represents this parameter.

  Only used if dstype == 'SYMBOLIC' and symbols is None.

- **mean** (*int or float, default 14.0*) – Mean for the background's distribution. Only used when bktype == 'NORMAL'.

- **stdev** (*int or float, default 7.0*) – Standard deviation for the background's distribution. Only used when bktype == 'NORMAL'.

- **probs** (*list or array of floats*) – Background weighted distribution probabilities. Only used when bktype == 'DISCRETE'. No default probabilities, if probs is None and bktype == 'DISCRETE', bktype defaults to 'UNIFORM'.

  Shape: Number of symbols or possible integers

  Range: [0,1]

  Math: sum(probs) == 1

- **in_memory** (*bool, default None*) – Determines if generated datasets return dense or sparse matrix (True/False).

  If None then if the generated dataset's size is larger then 10**5 it defaults to sparse, else outputs dense.

  Note: This parameter can be overwritten in the generate method.

_n
Dimensionality.

Type int

_stdout
default System.out

Type System object (java)

dstype
Type of Dataset to be generated, numeric or symbolic(categorical).

Type {'NUMERIC', 'SYMBOLIC'}

patterns
Type of patterns that will be hidden in the data.

Type list

clusterdistribution
Distribution used to calculate the size of a cluster.

Type list

contiguity
Data contiguity.

Type {'COLUMNS', 'CONTEXTS', 'NONE'}

time_profile
Time profile for the ORDER_PRESERVING pattern.

Type {'RANDOM', 'MONONICALLY_INCREASING', 'MONONICALLY_DECREASING', None}

seed
Seed to initialize random objects.

Type int

realval
If the dataset is real valued.

Type bool

minval
Dataset's minimum value.

Type float

maxval
Dataset's maximum value.

Type float

noise
Dataset's noise settings.

Type tuple

errors
Dataset's error settings.

Type tuple

missing
Dataset's missing settings.

Type tuple

symbols
Dataset's alphabet.

Type list

nsymbols
Length of the alphabet.

Type int

plaidcoherency
Type of plaid coherency.

Type {'ADDITIVE', 'MULTIPLICATIVE', 'INTERPOLED', 'NONE', 'NO_OVERLAPPING'}

percofoverlappingclusts
Percentage of overlapping clusters.

Type float

maxclustsperoverlappedarea
Maximum number of clusters overlapped per area.

Type int

maxpercofoverlappingelements
Maximum percentage of values shared by overlapped clusters.

Type float

percofoverlappingrows
Percentage of allowed amount of overlaping across clusters rows.

Type float

percofoverlappingcolumns
Percentage of allowed amount of overlaping across clusters columns.

Type float

percofoverlappingcontexts
Percentage of allowed amount of overlaping across clusters contexts.

Type float

background
Dataset's background settings

Type list

generatedDataset
Generated dataset.

Type Dataset object (java)

X
Generated dataset as tensor.

Type dense or sparse tensor

Y
Hidden cluster labels.

Type list

graph
N-partite graph

Type Graph object

in_memory
If dataset should be saved in memory (dense format)

Type bool

silenced
If prints to the console.

Type bool

save(*extension='default'*, *file_name='example'*, *path=None*, *single_file=None*, ***kwargs*)
Saves data files to chosen path.

Parameters

- **extension** (*{'default', 'csv'}, default 'default'*) – Extension of saved data file. If default, uses Java class default.
- **file_name** (*str, default 'example_dataset'*) – Saved files prefix.

- **path** (*str, default None*) – Path to save files. If None then files are saved in the current working directory.
- **single_file** (*Bool, default None.*) – If False dataset is saved in multiple data files. If None then if the dataset's size is larger then 10**5 it defaults to False, else True.
- ****kwargs** (*any, default None*) – Additional keywords that are passed on.

Examples

```
>>> generator = BiclusterGenerator(silence=True)
>>> generator.generate()
>>> generator.save(file_name='BicFiles', single_file=False)
>>> generator.save(extension='csv', file_name='BicFiles', delimiter=';')
```

1.3.2 Bicluster Generator byConfig

class nclustgen.BiclusterGen.**BiclusterGeneratorbyConfig**(*file_path=None*)

Bases: *nclustgen.BiclusterGen.BiclusterGenerator*

This class initializes the generator via configuration file.

Examples

```
>>> from nclustgen import BiclusterGeneratorbyConfig
>>> generator = BiclusterGeneratorbyConfig('example.json')
>>> generator.get_params()
{'X': None, 'Y': None, 'background': ['UNIFORM'], 'clusterdistribution': [['UNIFORM
↪', 4, 4], ['UNIFORM', 4, 4]],
'contiguity': 'NONE', 'dstype': 'NUMERIC', 'errors': (0.0, 0.0, 0.0),
↪'generatedDataset': None, 'graph': None,
'in_memory': 'True', 'maxclustsperoverlappedarea': 0, 'maxpercofoverlappingelements
↪': 0.0, 'maxval': 10.0,
'minval': -10.0, 'missing': (0.0, 0.0), 'noise': (0.0, 0.0, 0.0),
'patterns': [['CONSTANT', 'CONSTANT'], ['CONSTANT', 'NONE']],
↪'percofoverlappingclusts': 0.0,
'percofoverlappingcolumns': 1.0, 'percofoverlappingcontexts': 1.0,
↪'percofoverlappingrows': 1.0,
'plaidcoherency': 'NO_OVERLAPPING', 'realval': True, 'seed': -1, 'silenced': False,
↪'time_profile': None}
>>> x, y = generator.generate(nrows=50, ncols=100, nclusters=3)
>>> x
array([[-4.67,  3.57,  2.38, ..., -7.41, -4.14,  4.64],
       [-8.31,  8.06,  1.33, ..., -7.24, -2.62, -5.59],
       [-4.68, -5.43, -1.81, ..., -0.49, -1.34,  0.68],
       ...,
       [ 1.85,  9.55,  8.1 , ..., -2.5 ,  2.41, -5.54],
       [-2.09,  0.73,  6.38, ...,  0.46, -8.97,  4.46],
       [-7.21,  6.6 , -9.78, ..., -6.29, -7.24, -2.98]])
```

Parameters

file_path: str, default None Determines the path to the configuration file. If None then no parameters are passed to class.

1.4 Tricluster Generator

See also:

This module inherits from *Generator*, check it out for more info.

1.4.1 Tricluster Generator

class nclustgen.TriclusterGen.**TriclusterGenerator**(**args*, ***kwargs*)

Bases: *nclustgen.Generator.Generator*

This class provides an implementation for three-dimensional datasets with hidden triclusters.

Examples

```
>>> from nclustgen import TriclusterGenerator
>>> generator = TriclusterGenerator(
...     dstype='NUMERIC',
...     patterns=[['CONSTANT', 'CONSTANT', 'CONSTANT'], ['CONSTANT', 'NONE', 'NONE
↪']],
...     bktype='UNIFORM',
...     in_memory=True,
...     silence=True
... )
>>> generator.get_params()
{'X': None, 'Y': None, 'background': ['UNIFORM'], 'clusterdistribution': [['UNIFORM
↪', 4, 4], ['UNIFORM', 4, 4],
['UNIFORM', 4, 4]], 'contiguity': 'NONE', 'dstype': 'NUMERIC', 'errors': (0.0, 0.0,␣
↪0.0),
'generatedDataset': None, 'graph': None, 'in_memory': 'True',
↪'maxclustsperoverlappedarea': 0,
'maxpercofoverlappingelements': 0.0, 'maxval': 10.0, 'minval': -10.0, 'missing': (0.
↪0, 0.0), 'cuda': 3,
'noise': (0.0, 0.0, 0.0), 'patterns': [['CONSTANT', 'CONSTANT', 'CONSTANT'], [
↪'CONSTANT', 'NONE', 'NONE']],
'percofoverlappingclusts': 0.0, 'percofoverlappingcolumns': 1.0,
↪'percofoverlappingcontexts': 1.0,
'percofoverlappingrows': 1.0, 'plaidcoherency': 'NO_OVERLAPPING', 'realval': True,
↪'seed': -1,
'silenced': False, 'time_profile': None}
>>> x, y = generator.generate(nrows=50, ncols=100, ncontexts=5, nclusters=3)
>>> x
array([[[-1.29, -4.92, -2.49, ..., -9.17, -5.19,  6.66],
        [ 5.41, -3.04, -1.58, ..., -3.44,  1.99,  9.84],
        [-1.88, -9.09,  5.06, ..., -8.96, -8.4 ,  3.56],
        ...,
        [ 8.74,  4.07,  0.6 , ...,  6.73, -1.3 ,  5.  ],
        [ 3.33,  4.66, -5.72, ...,  0.55,  5.82, -3.17],
        [ 2.32, -9.29,  3.95, ...,  3.61,  3.93, -6.76]],
       [[ 4.34,  8.59, -1.96, ...,  0.88,  8.52, -7.85],
        [ 1.87, -8.59, -9.78, ...,  5.33, -7.45,  3.1 ],
        [-6.86, -3.93,  7.73, ...,  3.21,  6.54, -7.13],
        ...,
```

(continues on next page)

(continued from previous page)

```
        [-5.75,  9.91, -4.76, ...,  0.94, -9.2 , -1.32],
        [ 3.11, -8.26, -2.32, ..., -5.08,  5.33,  2.52],
        [-4.18,  7.98,  8.42, ...,  4.21, -0.03, -7.51]],
       [[ 1.22, -5.69, -8.72, ...,  5.78,  8.74,  1.44],
        [ 3.41, -7.45,  7.01, ...,  8.93, -6.01,  0.18],
        [ 3.8 ,  2.92, -1.87, ..., -1.16, -3.31, -3.02],
        ...,
        [ 4.82, -9.82,  0.31, ...,  9.91, -0.45,  7.86],
        [ 7.24,  8.28, -3.13, ...,  9.12, -0.47,  6.16],
        [-6.61, -7.34, -0.56, ...,  1.41, -1.7 ,  6.22]],
       [[ 3.46,  7.85, -8.23, ...,  1.33,  2.82, -4.05],
        [-8.87, -6.42,  2.28, ...,  9.72, -1.75,  5.01],
        [-0.26, -3.25, -9.16, ..., -1.69,  6.96,  4.63],
        ...,
        [-5.36,  2.84, -2.09, ...,  0.33, -2.88,  3.43],
        [ 5.72,  1.11,  2.11, ...,  0.27, -5.95,  3.39],
        [-7.02, -3.85, -5.44, ...,  1.64, -1.24, -2.74]],
       [[-2.39, -9.27, -8.12, ..., -7.86,  7.54,  4.99],
        [ 2.06,  3.84, -2.99, ...,  4.82, -9.29, -9.23],
        [ 0.21, -5.85, -8.45, ...,  4.35, -2.69,  0.34],
        ...,
        [-0.52, -2.59,  7.63, ..., -8.07, -3.51,  2.7 ],
        [ 4.93, -1.55, -0.65, ..., -0.87,  8.53,  9.97],
        [ 8.03,  2.32, -4.76, ..., -2.03, -4.48, -5.56]]])
>>> y
[[[8, 16, 17, 35], [36, 55, 69, 88], [0, 2, 3, 4]], [[7, 21, 33, 35], [22, 57, 65,␣
↪75], [0, 1, 2, 4]],
[[9, 19, 23, 27], [12, 19, 59, 72], [1, 2, 3, 4]]]
>>> graph = generator.to_graph(x, framework='dgl', device='cpu')
>>> graph
Graph(num_nodes={'col': 100, 'ctx': 5, 'row': 50},
      num_edges={('col', 'elem', 'ctx'): 500, ('row', 'elem', 'col'): 5000, ('row',
↪'elem', 'ctx'): 250},
      metagraph=[('col', 'ctx', 'elem'), ('row', 'col', 'elem'), ('row', 'ctx',
↪'elem')])
>>> generator.save(file_name='example', single_file=True)
```

Parameters

- **n** (*int, internal*) – Determines dimensionality (e.g. Bi/Tri clustering). Should only be used by subclasses.
- **dstype** (*{'NUMERIC', 'SYMBOLIC'}, default 'Numeric'*) – Type of Dataset to be generated, numeric or symbolic(categorical).
- **patterns** (*list or array, default [['CONSTANT', 'CONSTANT']]*) – Defines the type of patterns that will be hidden in the data.

  Shape: (number of patterns, number of dimensions)

  Patterns_Set: {CONSTANT, ADDITIVE, MULTIPLICATIVE, ORDER_PRESERVING, NONE}

  Numeric_Patterns_Set: {CONSTANT, ADDITIVE, MULTIPLICATIVE, ORDER_PRESERVING, NONE}

Symbolic_Patterns_Set: {CONSTANT, ORDER_PRESERVING, NONE}

Pattern_Combinations:

2D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’]
1	[‘None’, ‘Order_Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]
5	[‘Additive’, ‘Additive’]
6	[‘Constant’, ‘Additive’]
7	[‘Additive’, ‘Constant’]
8	[‘Multiplicative’, ‘Multiplicative’]
9	[‘Constant’, ‘Multiplicative’]
10	[‘Multiplicative’, ‘Constant’]

2D Symbolic Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’]
1	[‘None’, ‘Order_Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’, ‘None’]
1	[‘None’, ‘Order_Preserving’, ‘None’]
2	[‘None’, ‘None’, ‘Order_Preserving’]
3	[‘Constant’, ‘Constant’, ‘Constant’]
4	[‘None’, ‘Constant’, ‘Constant’]
5	[‘Constant’, ‘Constant’, ‘None’]
6	[‘Constant’, ‘None’, ‘Constant’]
7	[‘Constant’, ‘None’, ‘None’]
8	[‘None’, ‘Constant’, ‘None’]
9	[‘None’, ‘None’, ‘Constant’]
10	[‘Additive’, ‘Additive’, ‘Additive’]
11	[‘Additive’, ‘Additive’, ‘Constant’]
12	[‘Constant’, ‘Additive’, ‘Additive’]
13	[‘Additive’, ‘Constant’, ‘Additive’]
14	[‘Additive’, ‘Constant’, ‘Constant’]
15	[‘Constant’, ‘Additive’, ‘Constant’]
16	[‘Constant’, ‘Constant’, ‘Additive’]
17	[‘Multiplicative’, ‘Multiplicative’, ‘Multiplicative’]
18	[‘Multiplicative’, ‘Multiplicative’, ‘Constant’]
19	[‘Constant’, ‘Multiplicative’, ‘Multiplicative’]
20	[‘Multiplicative’, ‘Constant’, ‘Multiplicative’]
21	[‘Multiplicative’, ‘Constant’, ‘Constant’]
22	[‘Constant’, ‘Multiplicative’, ‘Constant’]
23	[‘Constant’, ‘Constant’, ‘Multiplicative’]

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’, ‘None’]
1	[‘None’, ‘Order_Preserving’, ‘None’]
2	[‘None’, ‘None’, ‘Order_Preserving’]
3	[‘Constant’, ‘Constant’, ‘Constant’]
4	[‘None’, ‘Constant’, ‘Constant’]
5	[‘Constant’, ‘Constant’, ‘None’]
6	[‘Constant’, ‘None’, ‘Constant’]
7	[‘Constant’, ‘None’, ‘None’]
8	[‘None’, ‘Constant’, ‘None’]
9	[‘None’, ‘None’, ‘Constant’]

- **bktype** (*{'NORMAL', 'UNIFORM', 'DISCRETE', 'MISSING'}, default 'UNIFORM'*) – Determines the distribution used to generate the background values.
- **clusterdistribution** (*list or array, default [['UNIFORM', 4.0, 4.0], ['UNIFORM', 4.0, 4.0]]*) – Distribution used to calculate the size of a cluster.

  Shape: number of dimensions, 3 -> param1(str), param2(float), param3(float)

  The first parameter(param1) is always the type of distribution {‘NORMAL’, ‘UNIFORM’}. If param1==UNIFORM, then param2 and param3 represents the min and max, respectively. If param1==NORMAL, then param2 and param3 represents the mean and standard deviation, respectively.

- **contiguity** (*{'COLUMNS', 'CONTEXTS', 'NONE'}, default None*) – Contiguity can occur on COLUMNS or CONTEXTS. To avoid contiguity use None.

  If dimensionality == 2 and contiguity == 'CONTEXTS' it defaults to None.

- **plaidcoherency** (*{'ADDITIVE', 'MULTIPLICATIVE', 'INTERPOLED', 'NONE', 'NO_OVERLAPPING'}, default 'NO_OVERLAPPING'*) – Enforces the type of plaid coherency. To avoid plaid coherency use NONE, to avoid any overlapping use 'NO_OVERLAPPING'.

- **percofoverlappingclusters** (*float, default 0.0*) – Percentage of overlapping clusters. Defines how many clusters are allowed to overlap.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **maxclustsperoverlappedarea** (*int, default 0*) – Maximum number of clusters overlapped per area. Maximum number of clusters that can overlap together.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0, nclusters]

- **maxpercofoverlappingelements** (*float, default 0.0*) – Maximum percentage of values shared by overlapped clusters.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **percofoverlappingrows** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters rows.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **percofoverlappingcolumns** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters columns.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **percofoverlappingcontexts** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters contexts.

  Not used if plaidcoherency == 'NO_OVERLAPPING' or cuda >= 3.

  Range: [0,1]

- **percmissingsonbackground** (*float, 0.0*) – Percentage of missing values on the background, that is, values that do not belong to planted clusters.

  Range: [0,1]

- **percmissingsonclusters** (*float, 0.0*) – Maximum percentage of missing values on each cluster.

  Range: [0,1]

- **percnoiseonbackground** (*float, 0.0*) – Percentage of noisy values on background, that is, values with added noise.

  Range: [0,1]

- **percnoiseonclusters** (*float, 0.0*) – Maximum percentage of noisy values on each cluster.

  Range: [0,1]

- **percnoisedeviation** (*int or float, 0.0*) – Percentage of symbol on noisy values deviation, that is, the maximum difference between the current symbol on the matrix and the one that will replaced it to be considered noise.

  If dstype == Numeric then percnoisedeviation -> float else int.

  Ex: Let Alphabet = [1,2,3,4,5] and CurrentSymbol = 3, if the noiseDeviation is '1', then CurrentSymbol will be, randomly, replaced by either '2' or '4'. If noiseDeviation is '2', CurrentSymbol can be replaced by either '1','2','4' or '5'.

- **percerroesonbackground** (*float, 0.0*) – Percentage of error values on background. Similar as noise, a new value is considered an error if the difference between it and the current value in the matrix is greater than noiseDeviation.

  Ex: Alphabet = [1,2,3,4,5], If currentValue = 2, and errorDeviation = 2, to turn currentValue an error, it's value must be replaced by '5', that is the only possible value that respects abs(currentValue - newValue) > noiseDeviation

  Range: [0,1]

- **percerrorsonclusters** (*float, 0.0*) – Percentage of errors values on background. Similar as noise, a new value is considered an error if the difference between it and the current value in the matrix is greater than noiseDeviation.

  Ex: Alphabet = [1,2,3,4,5], If currentValue = 2, and errorDeviation = 2, to turn currentValue an error, it's value must be replaced by '5', that is the only possible value that respects abs(currentValue - newValue) > noiseDeviation

  Range: [0,1]

- **percerrorondeviation** (*int or float, 0.0*) – Percentage of symbol on error values deviation, that is, the maximum difference between the current symbol on the matrix and the one that will replaced it to be considered error.

  If dstype == Numeric then percnoisedeviation -> float else int.

- **silence** (*bool, default False*) – If True them the class does not print to the console.

- **seed** (*int, default -1*) – Seed to initialize random objects.

  If seed is None or -1 then random objects are initialized without a seed.

- **timeprofile** (*{'RANDOM', 'MONONICALLY_INCREASING', 'MONONICALLY_DECREASING', None}, default None*) – It determines a time profile for the ORDER_PRESERVING pattern. Only used if ORDER_PRESERVING in patterns.

  If None and ORDER_PRESERVING in patterns it defaults to 'RANDOM'.

- **realval** (*bool, default True*) – Indicates if the dataset is real valued. Only used when dstype == 'NUMERIC'.

- **minval** (*int or float, default -10.0*) – Dataset's minimum value. Only used when dstype == 'NUMERIC'.

- **maxval** (*int or float, default 10.0*) – Dataset's maximum value. Only used when dstype == 'NUMERIC'.

- **symbols** (*list or array of strings, default None*) – Dataset's alphabet (list of possible values/symbols it can contain). Only used if dstype == 'SYMBOLIC'.

  Shape: alphabets length

- **nsymbols** (*int, default 10*) – Defines the length of the alphabet, instead of defining specific symbols this parameter can be passed, and a list of strings will be create with range(1, cuda), where cuda represents this parameter.

  Only used if dstype == 'SYMBOLIC' and symbols is None.

- **mean** (*int or float, default 14.0*) – Mean for the background's distribution. Only used when bktype == 'NORMAL'.

- **stdev** (*int or float, default 7.0*) – Standard deviation for the background's distribution. Only used when bktype == 'NORMAL'.

- **probs** (*list or array of floats*) – Background weighted distribution probabilities. Only used when bktype == 'DISCRETE'. No default probabilities, if probs is None and bktype == 'DISCRETE', bktype defaults to 'UNIFORM'.

  Shape: Number of symbols or possible integers

  Range: [0,1]

  Math: sum(probs) == 1

- **in_memory** (*bool, default None*) – Determines if generated datasets return dense or sparse matrix (True/False).

  If None then if the generated dataset's size is larger then 10**5 it defaults to sparse, else outputs dense.

  Note: This parameter can be overwritten in the generate method.

_n
Dimensionality.

Type int

_stdout
default System.out

Type System object (java)

dstype
Type of Dataset to be generated, numeric or symbolic(categorical).

Type {'NUMERIC', 'SYMBOLIC'}

patterns
Type of patterns that will be hidden in the data.

Type list

clusterdistribution
Distribution used to calculate the size of a cluster.

Type list

contiguity
Data contiguity.

Type {'COLUMNS', 'CONTEXTS', 'NONE'}

time_profile
Time profile for the ORDER_PRESERVING pattern.

Type {'RANDOM', 'MONONICALLY_INCREASING', 'MONONICALLY_DECREASING', None}

seed
Seed to initialize random objects.

Type int

realval
If the dataset is real valued.

Type bool

minval
Dataset's minimum value.

Type float

maxval
Dataset's maximum value.

Type float

noise
Dataset's noise settings.

Type tuple

errors
Dataset's error settings.

Type tuple

missing
Dataset's missing settings.

Type tuple

symbols
Dataset's alphabet.

Type list

nsymbols
Length of the alphabet.

Type int

plaidcoherency
Type of plaid coherency.

Type {'ADDITIVE', 'MULTIPLICATIVE', 'INTERPOLED', 'NONE', 'NO_OVERLAPPING'}

percofoverlappingclusts
Percentage of overlapping clusters.

Type float

maxclustsperoverlappedarea
Maximum number of clusters overlapped per area.

Type int

maxpercofoverlappingelements
Maximum percentage of values shared by overlapped clusters.

Type float

percofoverlappingrows
Percentage of allowed amount of overlaping across clusters rows.

Type float

percofoverlappingcolumns
Percentage of allowed amount of overlaping across clusters columns.

Type float

percofoverlappingcontexts
Percentage of allowed amount of overlaping across clusters contexts.

Type float

background
Dataset's background settings

Type list

generatedDataset
Generated dataset.

Type Dataset object (java)

X
Generated dataset as tensor.

Type dense or sparse tensor

Y
Hidden cluster labels.

Type list

graph
N-partite graph

Type Graph object

in_memory
If dataset should be saved in memory (dense format)

Type bool

silenced
If prints to the console.

Type bool

save(*extension='default'*, *file_name='example'*, *path=None*, *single_file=None*, ***kwargs*)
Saves data files to chosen path.

Parameters

- **extension** (*{'default', 'csv'}, default 'default'*) – Extension of saved data file. If default, uses Java class default. Else it returns a data file per context.
- **file_name** (*str, default 'example_dataset'*) – Saved files prefix.

- **path** (*str, default None*) – Path to save files. If None then files are saved in the current working directory.
- **single_file** (*Bool, default None.*) – If False dataset is saved in multiple data files. If None then if the dataset's size is larger then 10**5 it defaults to False, else True. Only used if extension=='default'.
- ****kwargs** (*any, default None*) – Additional keywords that are passed on.

Examples

```
>>> generator = TriclusterGenerator(silence=True)
>>> generator.generate()
>>> generator.save(file_name='TricFiles', single_file=False)
>>> generator.save(extension='csv', file_name='TricFiles', delimiter=';')
```

1.4.2 Tricluster Generator byConfig

class nclustgen.TriclusterGen.**TriclusterGeneratorbyConfig**(*file_path=None*)

Bases: *nclustgen.TriclusterGen.TriclusterGenerator*

This class initializes the generator via configuration file.

Examples

```
>>> from nclustgen import TriclusterGeneratorbyConfig
>>> generator = TriclusterGeneratorbyConfig('example.json')
>>> x, y = generator.generate(nrows=50, ncols=100, ncontexte=4, nclusters=2)
>>> x
array([[[ 3.94, -7.62, -2.68, ..., -1.66,  4.41, -3.8 ],
        [-2.27, -7.19, -3.42, ...,  7.19, -2.9 , -6.03],
        [-8.91, -9.46, -7.98, ..., -0.78, -7.66, -4.96],
        ...,
        [-7.93,  9.79,  2.95, ...,  2.01,  7.99,  6.15],
        [-4.25, -3.81, -1.43, ..., -0.61, -5.36, -8.09],
        [ 0.4 , -5.36, -3.68, ...,  8.5 ,  6.8 , -7.34]],
       [[ 0.62, -1.18, -3.07, ...,  0.23, -8.38,  2.96],
        [ 6.37,  4.63,  6.15, ...,  9.13,  9.6 ,  9.5 ],
        [-5.33,  0.15,  1.65, ...,  5.73, -4.64, -6.47],
        ...,
        [ 9.16,  4.75,  3.06, ...,  3.76, -3.09, -6.96],
        [ 3.6 ,  5.54, -0.2 , ...,  1.09,  9.23, -0.62],
        [ 2.68, -6.15, -8.99, ...,  8.65,  9.89,  7.63]],
       [[ 0.55, -1.03,  6.35, ...,  3.88,  5.96, -6.52],
        [-0.71,  7.99,  2.56, ..., -7.15,  0.33,  7.9 ],
        [ 0.86,  2.99,  3.69, ...,  1.57, -5.23,  4.59],
        ...,
        [ 4.2 ,  4.03, -9.11, ...,  5.28,  6.09,  1.19],
        [-0.31,  7.71,  7.57, ..., -3.57, -9.67, -9.89],
        [ 6.55,  4.69, -9.96, ..., -8.9 ,  7.31, -0.13]]])
```

Parameters

file_path: str, default None Determines the path to the configuration file. If None then no parameters are passed to class.

1.5 Generator

1.5.1 Generator

class nclustgen.Generator.**Generator**(*n*, *dstype='NUMERIC'*, *patterns=None*, *bktype='UNIFORM'*, *clusterdistribution=None*, *contiguity=None*, *plaidcoherency='NO_OVERLAPPING'*, *percofoverlappingclusters=0.0*, *maxclustsperoverlappedarea=0*, *maxpercofoverlappingelements=0.0*, *percofoverlappingrows=1.0*, *percofoverlappingcolumns=1.0*, *percofoverlappingcontexts=1.0*, *percmissingsonbackground=0.0*, *percmissingsonclusters=0.0*, *percnoiseonbackground=0.0*, *percnoiseonclusters=0.0*, *percnoisedeviation=0.0*, *percerroesonbackground=0.0*, *percerrorsonclusters=0.0*, *percerrorondeviation=0.0*, *silence=False*, *seed=None*, **args*, ***kwargs*)

Bases: `object`

Abstract class from where dimensional specific subclasses should inherit. Should not be called directly. This class abstracts dimensionality providing core implemented methods and abstract methods that should be implemented for any n-clustering generator.

Parameters

- **n** (*int, internal*) – Determines dimensionality (e.g. Bi/Tri clustering). Should only be used by subclasses.
- **dstype** (*{'NUMERIC', 'SYMBOLIC'}, default 'Numeric'*) – Type of Dataset to be generated, numeric or symbolic(categorical).
- **patterns** (*list or array, default [['CONSTANT', 'CONSTANT']]*) – Defines the type of patterns that will be hidden in the data.

  Shape: (number of patterns, number of dimensions)

  Patterns_Set: {CONSTANT, ADDITIVE, MULTIPLICATIVE, ORDER_PRESERVING, NONE}

  Numeric_Patterns_Set: {CONSTANT, ADDITIVE, MULTIPLICATIVE, ORDER_PRESERVING, NONE}

  Symbolic_Patterns_Set: {CONSTANT, ORDER_PRESERVING, NONE}

  Pattern_Combinations:

2D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’]
1	[‘None’, ‘Order_Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]
5	[‘Additive’, ‘Additive’]
6	[‘Constant’, ‘Additive’]
7	[‘Additive’, ‘Constant’]
8	[‘Multiplicative’, ‘Multiplicative’]
9	[‘Constant’, ‘Multiplicative’]
10	[‘Multiplicative’, ‘Constant’]

2D Symbolic Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’]
1	[‘None’, ‘Order_Preserving’]
2	[‘Constant’, ‘Constant’]
3	[‘None’, ‘Constant’]
4	[‘Constant’, ‘None’]

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’, ‘None’]
1	[‘None’, ‘Order_Preserving’, ‘None’]
2	[‘None’, ‘None’, ‘Order_Preserving’]
3	[‘Constant’, ‘Constant’, ‘Constant’]
4	[‘None’, ‘Constant’, ‘Constant’]
5	[‘Constant’, ‘Constant’, ‘None’]
6	[‘Constant’, ‘None’, ‘Constant’]
7	[‘Constant’, ‘None’, ‘None’]
8	[‘None’, ‘Constant’, ‘None’]
9	[‘None’, ‘None’, ‘Constant’]
10	[‘Additive’, ‘Additive’, ‘Additive’]
11	[‘Additive’, ‘Additive’, ‘Constant’]
12	[‘Constant’, ‘Additive’, ‘Additive’]
13	[‘Additive’, ‘Constant’, ‘Additive’]
14	[‘Additive’, ‘Constant’, ‘Constant’]
15	[‘Constant’, ‘Additive’, ‘Constant’]
16	[‘Constant’, ‘Constant’, ‘Additive’]
17	[‘Multiplicative’, ‘Multiplicative’, ‘Multiplicative’]
18	[‘Multiplicative’, ‘Multiplicative’, ‘Constant’]
19	[‘Constant’, ‘Multiplicative’, ‘Multiplicative’]
20	[‘Multiplicative’, ‘Constant’, ‘Multiplicative’]
21	[‘Multiplicative’, ‘Constant’, ‘Constant’]
22	[‘Constant’, ‘Multiplicative’, ‘Constant’]
23	[‘Constant’, ‘Constant’, ‘Multiplicative’]

3D Numeric Patterns Possible Combinations	
index	pattern combination
0	[‘Order_Preserving’, ‘None’, ‘None’]
1	[‘None’, ‘Order_Preserving’, ‘None’]
2	[‘None’, ‘None’, ‘Order_Preserving’]
3	[‘Constant’, ‘Constant’, ‘Constant’]
4	[‘None’, ‘Constant’, ‘Constant’]
5	[‘Constant’, ‘Constant’, ‘None’]
6	[‘Constant’, ‘None’, ‘Constant’]
7	[‘Constant’, ‘None’, ‘None’]
8	[‘None’, ‘Constant’, ‘None’]
9	[‘None’, ‘None’, ‘Constant’]

- **bktype** (*{'NORMAL', 'UNIFORM', 'DISCRETE', 'MISSING'}, default 'UNIFORM'*) – Determines the distribution used to generate the background values.
- **clusterdistribution** (*list or array, default [['UNIFORM', 4.0, 4.0], ['UNIFORM', 4.0, 4.0]]*) – Distribution used to calculate the size of a cluster.

  Shape: number of dimensions, 3 -> param1(str), param2(float), param3(float)

  The first parameter(param1) is always the type of distribution {‘NORMAL’, ‘UNIFORM’}. If param1==UNIFORM, then param2 and param3 represents the min and max, respectively. If param1==NORMAL, then param2 and param3 represents the mean and standard deviation, respectively.
- **contiguity** (*{'COLUMNS', 'CONTEXTS', 'NONE'}, default None*) – Contiguity can occur on COLUMNS or CONTEXTS. To avoid contiguity use None.

  If dimensionality == 2 and contiguity == ‘CONTEXTS’ it defaults to None.
- **plaidcoherency** (*{'ADDITIVE', 'MULTIPLICATIVE', 'INTERPOLED', 'NONE', 'NO_OVERLAPPING'}, default 'NO_OVERLAPPING'*) – Enforces the type of plaid coherency. To avoid plaid coherency use NONE, to avoid any overlapping use ‘NO_OVERLAPPING’.
- **percofoverlappingclusters** (*float, default 0.0*) – Percentage of overlapping clusters. Defines how many clusters are allowed to overlap.

  Not used if plaidcoherency == ‘NO_OVERLAPPING’.

  Range: [0,1]
- **maxclustsperoverlappedarea** (*int, default 0*) – Maximum number of clusters overlapped per area. Maximum number of clusters that can overlap together.

  Not used if plaidcoherency == ‘NO_OVERLAPPING’.

  Range: [0, nclusters]
- **maxpercofoverlappingelements** (*float, default 0.0*) – Maximum percentage of values shared by overlapped clusters.

  Not used if plaidcoherency == ‘NO_OVERLAPPING’.

  Range: [0,1]
- **percofoverlappingrows** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters rows.

  Not used if plaidcoherency == ‘NO_OVERLAPPING’.

Range: [0,1]

- **percofoverlappingcolumns** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters columns.

  Not used if plaidcoherency == 'NO_OVERLAPPING'.

  Range: [0,1]

- **percofoverlappingcontexts** (*float, default 1.0*) – Percentage of allowed amount of overlaping across clusters contexts.

  Not used if plaidcoherency == 'NO_OVERLAPPING' or cuda >= 3.

  Range: [0,1]

- **percmissingsonbackground** (*float, 0.0*) – Percentage of missing values on the background, that is, values that do not belong to planted clusters.

  Range: [0,1]

- **percmissingsonclusters** (*float, 0.0*) – Maximum percentage of missing values on each cluster.

  Range: [0,1]

- **percnoiseonbackground** (*float, 0.0*) – Percentage of noisy values on background, that is, values with added noise.

  Range: [0,1]

- **percnoiseonclusters** (*float, 0.0*) – Maximum percentage of noisy values on each cluster.

  Range: [0,1]

- **percnoisedeviation** (*int or float, 0.0*) – Percentage of symbol on noisy values deviation, that is, the maximum difference between the current symbol on the matrix and the one that will replaced it to be considered noise.

  If dstype == Numeric then percnoisedeviation -> float else int.

  Ex: Let Alphabet = [1,2,3,4,5] and CurrentSymbol = 3, if the noiseDeviation is '1', then CurrentSymbol will be, randomly, replaced by either '2' or '4'. If noiseDeviation is '2', CurrentSymbol can be replaced by either '1','2','4' or '5'.

- **percerroesonbackground** (*float, 0.0*) – Percentage of error values on background. Similar as noise, a new value is considered an error if the difference between it and the current value in the matrix is greater than noiseDeviation.

  Ex: Alphabet = [1,2,3,4,5], If currentValue = 2, and errorDeviation = 2, to turn currentValue an error, it's value must be replaced by '5', that is the only possible value that respects abs(currentValue - newValue) > noiseDeviation

  Range: [0,1]

- **percerrorsonclusters** (*float, 0.0*) – Percentage of errors values on background. Similar as noise, a new value is considered an error if the difference between it and the current value in the matrix is greater than noiseDeviation.

  Ex: Alphabet = [1,2,3,4,5], If currentValue = 2, and errorDeviation = 2, to turn currentValue an error, it's value must be replaced by '5', that is the only possible value that respects abs(currentValue - newValue) > noiseDeviation

  Range: [0,1]

- **percerrorondeviation** (*int or float, 0.0*) – Percentage of symbol on error values deviation, that is, the maximum difference between the current symbol on the matrix and the one that will replaced it to be considered error.

  If dstype == Numeric then percnoisedeviation -> float else int.

- **silence** (*bool, default False*) – If True them the class does not print to the console.

- **seed** (*int, default -1*) – Seed to initialize random objects.

  If seed is None or -1 then random objects are initialized without a seed.

- **timeprofile** (*{'RANDOM', 'MONONICALLY_INCREASING', 'MONONICALLY_DECREASING', None}, default None*) – It determines a time profile for the ORDER_PRESERVING pattern. Only used if ORDER_PRESERVING in patterns.

  If None and ORDER_PRESERVING in patterns it defaults to 'RANDOM'.

- **realval** (*bool, default True*) – Indicates if the dataset is real valued. Only used when dstype == 'NUMERIC'.

- **minval** (*int or float, default -10.0*) – Dataset's minimum value. Only used when dstype == 'NUMERIC'.

- **maxval** (*int or float, default 10.0*) – Dataset's maximum value. Only used when dstype == 'NUMERIC'.

- **symbols** (*list or array of strings, default None*) – Dataset's alphabet (list of possible values/symbols it can contain). Only used if dstype == 'SYMBOLIC'.

  Shape: alphabets length

- **nsymbols** (*int, default 10*) – Defines the length of the alphabet, instead of defining specific symbols this parameter can be passed, and a list of strings will be create with range(1, cuda), where cuda represents this parameter.

  Only used if dstype == 'SYMBOLIC' and symbols is None.

- **mean** (*int or float, default 14.0*) – Mean for the background's distribution. Only used when bktype == 'NORMAL'.

- **stdev** (*int or float, default 7.0*) – Standard deviation for the background's distribution. Only used when bktype == 'NORMAL'.

- **probs** (*list or array of floats*) – Background weighted distribution probabilities. Only used when bktype == 'DISCRETE'. No default probabilities, if probs is None and bktype == 'DISCRETE', bktype defaults to 'UNIFORM'.

  Shape: Number of symbols or possible integers

  Range: [0,1]

  Math: sum(probs) == 1

- **in_memory** (*bool, default None*) – Determines if generated datasets return dense or sparse matrix (True/False).

  If None then if the generated dataset's size is larger then 10**5 it defaults to sparse, else outputs dense.

Note: This parameter can be overwritten in the generate method.

_n
Dimensionality.

Type int

_stdout
default System.out

Type System object (java)

dstype
Type of Dataset to be generated, numeric or symbolic(categorical).

Type {'NUMERIC', 'SYMBOLIC'}

patterns
Type of patterns that will be hidden in the data.

Type list

clusterdistribution
Distribution used to calculate the size of a cluster.

Type list

contiguity
Data contiguity.

Type {'COLUMNS', 'CONTEXTS', 'NONE'}

time_profile
Time profile for the ORDER_PRESERVING pattern.

Type {'RANDOM', 'MONONICALLY_INCREASING', 'MONONICALLY_DECREASING', None}

seed
Seed to initialize random objects.

Type int

realval
If the dataset is real valued.

Type bool

minval
Dataset's minimum value.

Type float

maxval
Dataset's maximum value.

Type float

noise
Dataset's noise settings.

Type tuple

errors
Dataset's error settings.

Type tuple

missing
Dataset's missing settings.

Type tuple

symbols
Dataset's alphabet.

Type list

nsymbols
Length of the alphabet.

Type int

plaidcoherency
Type of plaid coherency.

Type {'ADDITIVE', 'MULTIPLICATIVE', 'INTERPOLED', 'NONE', 'NO_OVERLAPPING'}

percofoverlappingclusts
Percentage of overlapping clusters.

Type float

maxclustsperoverlappedarea
Maximum number of clusters overlapped per area.

Type int

maxpercofoverlappingelements
Maximum percentage of values shared by overlapped clusters.

Type float

percofoverlappingrows
Percentage of allowed amount of overlaping across clusters rows.

Type float

percofoverlappingcolumns
Percentage of allowed amount of overlaping across clusters columns.

Type float

percofoverlappingcontexts
Percentage of allowed amount of overlaping across clusters contexts.

Type float

background
Dataset's background settings

Type list

generatedDataset
Generated dataset.

Type Dataset object (java)

X
Generated dataset as tensor.

Type dense or sparse tensor

Y
Hidden cluster labels.

Type list

graph
N-partite graph

Type Graph object

in_memory
If dataset should be saved in memory (dense format)

Type bool

silenced
If prints to the console.

Type bool

get_params()
Returns the classes attributes.

Returns Values of class attributes.

Return type dict

Examples

```
>>> generator = BiclusterGenerator()
>>> generator.get_params()
{'X': None, 'Y': None, 'background': ['UNIFORM'], 'clusterdistribution': [[
↪'UNIFORM', 4.0, 4.0],
['UNIFORM', 4.0, 4.0]], 'contiguity': 'NONE', 'dstype': 'NUMERIC', 'errors': (0.
↪0, 0.0, 0.0),
'generatedDataset': None, 'graph': None, 'in_memory': None,
↪'maxclustsperoverlappedarea': 0,
'maxpercofoverlappingelements': 0.0, 'maxval': 10.0, 'minval': -10.0, 'missing
↪': (0.0, 0.0),
'noise': (0.0, 0.0, 0.0), 'patterns': [['CONSTANT', 'CONSTANT']],
↪'percofoverlappingclusts': 0.0,
'percofoverlappingcolumns': 1.0, 'percofoverlappingcontexts': 1.0,
↪'percofoverlappingrows': 1.0,
'plaidcoherency': 'NO_OVERLAPPING', 'realval': True, 'seed': -1, 'silenced':␣
↪False, 'time_profile': None}
```

property cluster_info
Returns clusters info.

Returns Hidden cluster info.

Return type dict

Examples

```
>>> generator = BiclusterGenerator(silence=True)
>>> generator.generate(no_return=True)
>>> generator.cluster_info
{'0': {'%Errors': '0', 'Type': 'Numeric', '%Missings': '0', '%Noise': '0', 'X':␣
↪[15, 51, 63, 92],
'Y': [7, 29, 35, 94], 'RowPattern': 'Constant', 'ColumnPattern': 'Constant',
'Data': [['-8.61', '-8.61', '-8.61', '-8.61'], ['-8.61', '-8.61', '-8.61', '-8.
↪61'],
['-8.61', '-8.61', '-8.61', '-8.61'], ['-8.61', '-8.61', '-8.61', '-8.61']],
'PlaidCoherency': 'No Overlapping', '#rows': 4, '#columns': 4}}
```

property coverage

Returns clusters dataset coverage.

Returns Percentage of cluster coverage.

Return type float

Examples

```
>>> generator = BiclusterGenerator(silence=True)
>>> generator.generate(no_return=True)
>>> generator.coverage
0.16
```

abstract save(*extension='default'*, *file_name='example_dataset'*, *path=None*, *single_file=None*, ***kwargs*)

Saves data files to chosen path.

Parameters

- **extension** (*{'default', 'csv'}, default 'default'*) – Extension of saved data file.
- **file_name** (*str, default 'example_dataset'*) – Saved files prefix.
- **path** (*str, default None*) – Path to save files. If None then files are saved in the current working directory.
- **single_file** (*Bool, default None.*) – If False dataset is saved in multiple data files. If None then if the dataset's size is larger then 10**5 it defaults to False, else True. Only used if extension=='default'.
- ****kwargs** (*any, default None*) – Additional keywords that are passed on.

to_tensor(*generatedDataset=None*, *in_memory=None*, *keys=None*)

Returns generated dataset as somekind of tensor and hidden cluster labels.

Parameters

- **generatedDataset** (*Dataset object*) – Generated dataset (java object).
- **in_memory** (*bool, default None*) – Determines if generated datasets return dense or sparse matrix (True/False).

  If None then if the generated dataset's size is larger then 10**5 it defaults to sparse, else outputs dense.
- **keys** (*list, default ['X', 'Y', 'Z']*) – Axis keys. Do not overwrite, unless you are using a different dataset object.

Returns

- *dense or sparse tensor* – Generated dataset as tensor.

  Shape: (ncontexts, nrows, ncols) or (nrows, ncols)

- *list* – Hidden cluster labels.

  Shape: (nclusters, any)

Examples

```
>>> generator = BiclusterGenerator(silence=True)
>>> generator.generate(no_return=True)
>>> x, y = generator.to_tensor(generatedDataset=generator.generatedDataset, in_
↪memory=True)
>>> x
array([[-4.15,  9.88,  7.69, ...,  3.68,  1.72, -6.95],
       [ 7.37,  2.63, -0.13, ..., -2.53,  2.03,  8.03],
       [ 4.28,  0.36,  8.66, ..., -1.11,  6.28, -1.03],
       ...,
       [-9.25, -9.15, -4.68, ...,  2.06, -6.19,  2.54],
       [ 2.63, -3.03,  3.8 , ...,  4.13, -4.17,  7.68],
       [-1.98,  8.02,  1.89, ...,  3.59,  4.27,  6.4 ]])
```

to_graph(*x=None*, *framework='networkx'*, *device='cpu'*, ***kwargs*)

Returns a n-partite graph, where n==dim.

Parameters

- **x** (*numpy array*) – Data array.
- **framework** (*{networkx, dgl}, default 'networkx'*) – Backend to use to build graph.
- **device** (*{'cpu', 'gpu'}, default 'cpu'*) – Type of device for storing the tensor. Only used if framework==dgl.
- ****kwargs** (*any, default None*) – Additional keywords that are passed on.

Returns

N-partite graph, where n==dim.

Shape: (nrows + ncols + ncontexts, nrows * ncols * ncontexts * 3(dim)) or (nrows + ncols, nrows * ncols)

Return type Graph object

Examples

```
>>> generator = BiclusterGenerator(silence=True)
>>> X, y = generator.generate()
>>> g = generator.to_graph(X, framework='dgl')
Graph(num_nodes={'col': 100, 'row': 100},
      num_edges={('row', 'elem', 'col'): 10000},
      metagraph=[('row', 'col', 'elem')])
```

generate(*nrows=100*, *ncols=100*, *ncontexts=3*, *nclusters=1*, *no_return=False*, ***kwargs*)

Generates dataset, and may return somekind of tensor and hidden cluster labels.

Parameters

- **nrows** (*int, default 100*) – Number of rows in generated dataset.
- **ncols** (*int, default 100*) – Number of columns in generated dataset.
- **ncontexts** (*int, default 3*) – Number of contexts in generated dataset. Only used if dim >= 3.
- **nclusters** (*int, default 1*) – Number of clusters in generated dataset.
- **no_return** (*bool, default False*) – If True method returns None.
- ****kwargs** (*any, default None*) – Additional keywords that are passed on.

Returns

- *dense or sparse tensor* – Generated dataset as tensor.

  Shape: (ncontexts, nrows, ncols) or (nrows, ncols)

- *list* – Hidden cluster labels.

  Shape: (nclusters, any)

- *None* – If no_return==True.

Examples

```
>>> gen = BiclusterGenerator(silence=True)
>>> x, y = gen.generate(nrows=100, ncols=200, nclusters=20, in_memory=True)
>>> x
array([[-7.36,  4.88,  8.42, ..., -5.04, -4.93,  6.35],
       [-7.1 ,  0.47, -2.58, ..., -3.03,  0.42,  8.76],
       [-8.08,  4.19,  2.53, ..., -4.3 ,  7.54,  0.94],
       ...,
       [-0.52,  0.38,  6.98, ..., -7.6 ,  5.71,  9.24],
       [-1.28, -3.55, -3.13, ..., -4.17, -6.05, -9.87],
       [-5.79, -6.05, -2.24, ...,  1.88,  1.97,  6.05]])
```

static shutdownJVM()

Shuts down JVM.

> **Caution:** If the JVM is shutdown it cannot be restarted on the same session.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

Symbols

B

C

D

E

G

I

M

N

P

R

S

T

X

Y