python-cluster Documentation

Release 1.4.1.post1

Michel Albert

Jun 07, 2018

Contents

CHAPTER 1

Index

1.1 Changelog

1.1.1 Release 1.4.1.post1

This is a "house-keeping" commit. No new features or fixes are introduced.

- Update changelog.
- Switch doc-building to use `pipenv` & update `Pipfile` accordingly.

1.1.2 Release 1.4.1

- Fix clustering of dictionaries. See GitHub issue #28 (Tim Littlefair).

1.1.3 Release 1.4.0

- Added a "display" method to hierarchical clusters (by 1kastner).

1.1.4 Release 1.3.2 & 1.3.3

- Fix regression introduced in 1.3.1 related to package version metadata.

1.1.5 Release 1.3.1

- Don't break if the cluster is initiated with iterable elements (GitHub Issue #20).
- Fix package version metadata in setup.py

1.1.6 Release 1.3.0

- Performance improvments for hierarchical clustering (at the cost of memory)
- Cluster instances are now iterable. It will iterate over each element, resulting in a flat list of items.
- New option to specify a progress callback to hierarchical clustring. This method will be called on each iteration for hierarchical clusters. It gets two numeric values as argument: The total count of elements, and the number of processed elements. It gives users a way to present to progress on screen.
- The library now also has a `__version__` member.

1.1.7 Release 1.2.2

- Package metadata fixed.

1.1.8 Release 1.2.1

- Fixed an issue in multiprocessing code.

1.1.9 Release 1.2.0

- Multiprocessing (by loisaidasam)
- Python 3 support
- Split up one big file into smaller more logical sub-modules
- Fixed https://github.com/exhuma/python-cluster/issues/11
- Documentation update.
- Migrated to GitHub

1.1.10 Release 1.1.1b3

- Fixed bug #1727558
- Some more unit-tests
- ValueError changed to ClusteringError where appropriate

1.1.11 Release 1.1.1b2

- Fixed bug #1604859 (thanks to Willi Richert for reporting it)

1.1.12 Release 1.1.1b1

- Applied SVN patch [1535137] (thanks ajaksu)
  - Topology output supported
  - `data` and `raw_data` are now properties.

1.1.13 Release 1.1.0b1

- KMeans Clustering implemented for simple numeric tuples.

  Data in the form `[(1,1), (2,1), (5,3), ...]` can be clustered.

  Usage:

```
>>> from cluster import KMeansClustering
>>> cl = KMeansClustering([(1,1), (2,1), (5,3), ...])
>>> clusters = cl.getclusters(2)
```

  The method `getclusters` takes the amount of clusters you would like to have as parameter.

  Only numeric values are supported in the tuples. The reason for this is that the "centroid" method which I use, essentially returns a tuple of floats. So you will lose any other kind of metadata. Once I figure out a way how to recode that method, other types should be possible.

1.1.14 Release 1.0.1b2

- Optimized calculation of the hierarchical clustering by using the fact, that the generated matrix is symmetrical.

1.1.15 Release 1.0.1b1

- Implemented complete-, average-, and uclus-linkage methods. You can select one by specifying it in the constructor, for example:

```
cl = HierarchicalClustering(data, distfunc, linkage='uclus')
```

  or by setting it before starting the clustering process:

```
cl = HierarchicalClustering(data, distfunc)
cl.setLinkageMethod('uclus')
cl.cluster()
```

- Clustering is not executed on object creation, but on the first call of `getlevel`. You can force the creation of the clusters by calling the `cluster` method as shown above.

CHAPTER 2

Introduction

Implementation of cluster algorithms in pure Python.

As this is exacuted in the Python runtime, the code runs slower than similar implementations in compiled languages. You gain however to run this on pretty much any Python object. The different clustering methods have different prerequisites however which are mentioned in the different implementations.

CHAPTER 3

Example for K-Means Clustering

```
from cluster import KMeansClustering
data = [
    (8, 2),
    (7, 3),
    (2, 6),
    (3, 5),
    (3, 6),
    (1, 5),
    (8, 1),
    (3, 4),
    (8, 3),
    (9, 2),
    (2, 5),
    (9, 3)
]
cl = KMeansClustering(data)
cl.getclusters(2)
```

The above code would give the following result:

```
[
    [(8, 2), (8, 1), (8, 3), (7, 3), (9, 2), (9, 3)],
    [(3, 5), (1, 5), (3, 4), (2, 6), (2, 5), (3, 6)]
]
```

CHAPTER 4

Example for Hierarchical Clustering

```
from cluster import HierarchicalClustering
data = [791, 956, 676, 124, 564, 84, 24, 365, 594, 940, 398,
        971, 131, 365, 542, 336, 518, 835, 134, 391]
cl = HierarchicalClustering(data)
cl.getlevel(40)
```

The above code would give the following result:

```
[
    [24],
    [84, 124, 131, 134],
    [336, 365, 365, 391, 398],
    [676],
    [594, 518, 542, 564],
    [940, 956, 971],
    [791],
    [835],
]
```

Using *getlevel()* returns clusters where the distance between each cluster is no less than *level*.

Note: Due to a bug in earlier releases, the elements of the input data *must be* sortable!

CHAPTER 5

API

5.1 cluster

class cluster.cluster.**Cluster**(*level*, **args*)

Bases: object

A collection of items. This is internally used to detect clustered items in the data so we could distinguish other collection types (lists, dicts, ...) from the actual clusters. This means that you could also create clusters of lists with this class.

display(*depth=0*)

Pretty-prints this cluster. Useful for debuging.

getlevel(*threshold*)

Retrieve all clusters up to a specific level threshold. This level-threshold represents the maximum distance between two clusters. So the lower you set this threshold, the more clusters you will receive and the higher you set it, you will receive less but bigger clusters.

Parameters **threshold** – The level threshold:

Note: It is debatable whether the value passed into this method should really be as strongly linked to the real cluster-levels as it is right now. The end-user will not know the range of this value unless s/he first inspects the top-level cluster. So instead you might argue that a value ranging from 0 to 1 might be a more useful approach.

---

topology()

Returns the structure (topology) of the cluster as tuples.

Output from cl.data:

```
[<Cluster@0.833333333333(['CVS',
 <Cluster@0.818181818182(['34.xls',
 <Cluster@0.789473684211([<Cluster@0.555555555556(['0.txt',
```

(continues on next page)

(continued from previous page)

```
 <Cluster@0.181818181818(['ChangeLog', 'ChangeLog.txt'])>])>,
 <Cluster@0.684210526316(['20060730.py',
 <Cluster@0.684210526316(['.cvsignore',
 <Cluster@0.647058823529(['About.py', <Cluster@0.625(['.idlerc',
 '.pylint.d'])>])>])>])>])>])>]
```

Corresponding output from cl.topo():

```
('CVS', ('34.xls', (('0.txt', ('ChangeLog', 'ChangeLog.txt')),
('20060730.py', ('.cvsignore', ('About.py',
('.idlerc', '.pylint.d')))))))
```

5.2 cluster.matrix

class cluster.matrix.**Matrix**(*data*, *combinfunc*, *symmetric=False*, *diagonal=None*)

Bases: object

Object representation of the item-item matrix.

genmatrix(*num_processes=1*)

Actually generate the matrix

Parameters **num_processes** – If you want to use multiprocessing to split up the work and run `combinfunc()` in parallel, specify `num_processes > 1` and this number of workers will be spun up, the work is split up amongst them evenly.

worker()

Multiprocessing task function run by worker processes

5.3 cluster.method.base

class cluster.method.base.**BaseClusterMethod**(*input*, *distance_function*, *progress_callback=None*)

Bases: object

The base class of all clustering methods.

Parameters **input** – a list of objects

Distance_function a function returning the distance - or opposite of similarity `(distance = -similarity)` - of two items from the input. In other words, the closer the two items are related, the smaller this value needs to be. With 0 meaning they are exactly the same.

Note: The distance function should always return the absolute distance between two given items of the list. Say:

```
distance(input[1], input[4]) = distance(input[4], input[1])
```

This is very important for the clustering algorithm to work! Naturally, the data returned by the distance function MUST be a comparable datatype, so you can perform arithmetic comparisons on them (< or >)! The simplest examples would be floats or ints. But as long as they are comparable, it's ok.

---

data
Returns the data that is currently in process.

raw_data
Returns the raw data (data without being clustered).

topo()
Returns the structure (topology) of the cluster.

See *topology()* for more information.

5.4 cluster.method.hierarchical

class cluster.method.hierarchical.**HierarchicalClustering**(*data*, *distance_function*, *linkage=None*, *num_processes=1*, *progress_callback=None*)

Bases: *cluster.method.base.BaseClusterMethod*

Implementation of the hierarchical clustering method as explained in a tutorial by *matteucc*.

Object prerequisites:

- Items must be sortable (See issue #11)
- Items must be hashable.

Example:

```
>>> from cluster import HierarchicalClustering
>>> # or: from cluster import *
>>> cl = HierarchicalClustering([123,334,345,242,234,1,3],
        lambda x,y: float(abs(x-y)))
>>> cl.getlevel(90)
[[345, 334], [234, 242], [123], [3, 1]]
```

Note that all of the returned clusters are more than 90 (`getlevel(90)`) apart.

See *BaseClusterMethod* for more details.

Parameters

- **data** – The collection of items to be clustered.
- **distance_function** – A function which takes two elements of `data` and returns a distance between both elements (note that the distance should not be returned as negative value!)
- **linkage** – The method used to determine the distance between two clusters. See *set_linkage_method()* for possible values.
- **num_processes** – If you want to use multiprocessing to split up the work and run `genmatrix()` in parallel, specify num_processes > 1 and this number of workers will be spun up, the work split up amongst them evenly.
- **progress_callback** – A function to be called on each iteration to publish the progress. The function is called with two integer arguments which represent the total number of elements in the cluster, and the remaining elements to be clustered.

cluster(*matrix=None*, *level=None*, *sequence=None*)
Perform hierarchical clustering.

Parameters

- **matrix** – The 2D list that is currently under processing. The matrix contains the distances of each item with each other
- **level** – The current level of clustering
- **sequence** – The sequence number of the clustering

display()
Prints a simple dendogram-like representation of the full cluster to the console.

getlevel(*threshold*)
Returns all clusters with a maximum distance of *threshold* in between each other

Parameters **threshold** – the maximum distance between clusters.

See *getlevel()*

publish_progress(*total*, *current*)
If a progress function was supplied, this will call that function with the total number of elements, and the remaining number of elements.

Parameters

- **total** – The total number of elements.
- **remaining** – The remaining number of elements.

set_linkage_method(*method*)
Sets the method to determine the distance between two clusters.

Parameters **method** – The method to use. It can be one of `'single'`, `'complete'`, `'average'` or `'uclus'`, or a callable. The callable should take two collections as parameters and return a distance value between both collections.

5.5 cluster.method.kmeans

class cluster.method.kmeans.**KMeansClustering**(*data*, *distance=None*, *equality=None*)
Bases: `object`

Implementation of the kmeans clustering method as explained in a tutorial by *matteucc*.

Example:

```
>>> from cluster import KMeansClustering
>>> cl = KMeansClustering([(1,1), (2,1), (5,3), ...])
>>> clusters = cl.getclusters(2)
```

Parameters

- **data** – A list of tuples or integers.
- **distance** – A function determining the distance between two items. Default (if `None` is passed): It assumes the tuples contain numeric values and appiles a generalised form of the euclidian-distance algorithm on them.
- **equality** – A function to test equality of items. By default the standard python equality operator (==) is applied.

Raises **ValueError** – if the list contains heterogeneous items or if the distance between items cannot be determined.

assign_item(*item*, *origin*)
Assigns an item from a given cluster to the closest located cluster.

Parameters

- **item** – the item to be moved.
- **origin** – the originating cluster.

getclusters(*count*)
Generates *count* clusters.

Parameters **count** – The amount of clusters that should be generated. count must be greater than `1`.

Raises ***ClusteringError*** – if *count* is out of bounds.

initialise_clusters(*input_*, *clustercount*)
Initialises the clusters by distributing the items from the data. evenly across n clusters

Parameters

- **input** – the data set (a list of tuples).
- **clustercount** – the amount of clusters (n).

move_item(*item*, *origin*, *destination*)
Moves an item from one cluster to anoter cluster.

Parameters

- **item** – the item to be moved.
- **origin** – the originating cluster.
- **destination** – the target cluster.

5.6 cluster.util

exception cluster.util.**ClusteringError**
Bases: exceptions.Exception

cluster.util.**centroid**(*data*, *method=<function median>*)
returns the central vector of a list of vectors

cluster.util.**dotproduct**(*a*, *b*)
Calculates the dotproduct between two vecors

cluster.util.**flatten**(*L*)
Flattens a list.

Example:

```
>>> flatten([a,b,[c,d,[e,f]]])
[a,b,c,d,e,f]
```

cluster.util.**fullyflatten**(*container*)
Completely flattens out a cluster and returns a one-dimensional set containing the cluster's items. This is useful in cases where some items of the cluster are clusters in their own right and you only want the items.

Parameters **container** – the container to flatten.

cluster.util.**magnitude**(*a*)
calculates the magnitude of a vecor

cluster.util.**mean**(*numbers*)
Returns the arithmetic mean of a numeric list. see: http://mail.python.org/pipermail/python-list/2004-December/294990.html

cluster.util.**median**(*numbers*)
Return the median of the list of numbers. see: http://mail.python.org/pipermail/python-list/2004-December/294990.html

cluster.util.**minkowski_distance**(*x*, *y*, *p=2*)
Calculates the minkowski distance between two points.

Parameters

- **x** – the first point
- **y** – the second point
- **p** – the order of the minkowski algorithm. If *p=1* it is equal to the manhatten distance, if *p=2* it is equal to the euclidian distance. The higher the order, the closer it converges to the Chebyshev distance, which has *p=infinity*.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

W