05_density_based_clustering

September 29, 2021

1 Density- Based Clustering

Density-based clustering algorithms assign cluster membership based on proximity to other cluster members. They pursue the goal of identifying dense regions of arbitrary shapes and sizes. They do not require the specification of a certain number of clusters but instead rely on parameters that define the size of a neighborhood and a density threshold.

1.1 Imports & Settings

[1]:

```
%matplotlib inline
import warnings
from time import sleep
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import cm
import matplotlib.ticker as ticker
import pandas as pd
import numpy as np
from numpy import atleast_2d
from random import shuffle
from sklearn.decomposition import PCA
from sklearn.cluster import DBSCAN
from hdbscan import HDBSCAN
from sklearn.metrics import adjusted_mutual_info_score
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
from sklearn.neighbors import KDTree
```

[2]:

```
cmap = cm.get_cmap('viridis')
pd.options.display.float_format = '{:,.2f}'.format
warnings.filterwarnings('ignore')
```

1.2 Load Iris Data

[3]:

```
iris = load_iris()
iris.keys()
```

[3]:

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names',
'filename'])
```

1.3 Create DataFrame

[4]:

```
features = iris.feature_names
data = pd.DataFrame(data=np.column_stack([iris.data, iris.target]),
                    columns=features + ['label'])
data.label = data.label.astype(int)
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
sepal length (cm)    150 non-null float64
sepal width (cm)     150 non-null float64
petal length (cm)    150 non-null float64
petal width (cm)     150 non-null float64
label                150 non-null int64
dtypes: float64(4), int64(1)
memory usage: 5.9 KB
```

1.3.1 Standardize Data

[5]:

```
scaler = StandardScaler()
features_standardized = scaler.fit_transform(data[features])
n = len(data)
```

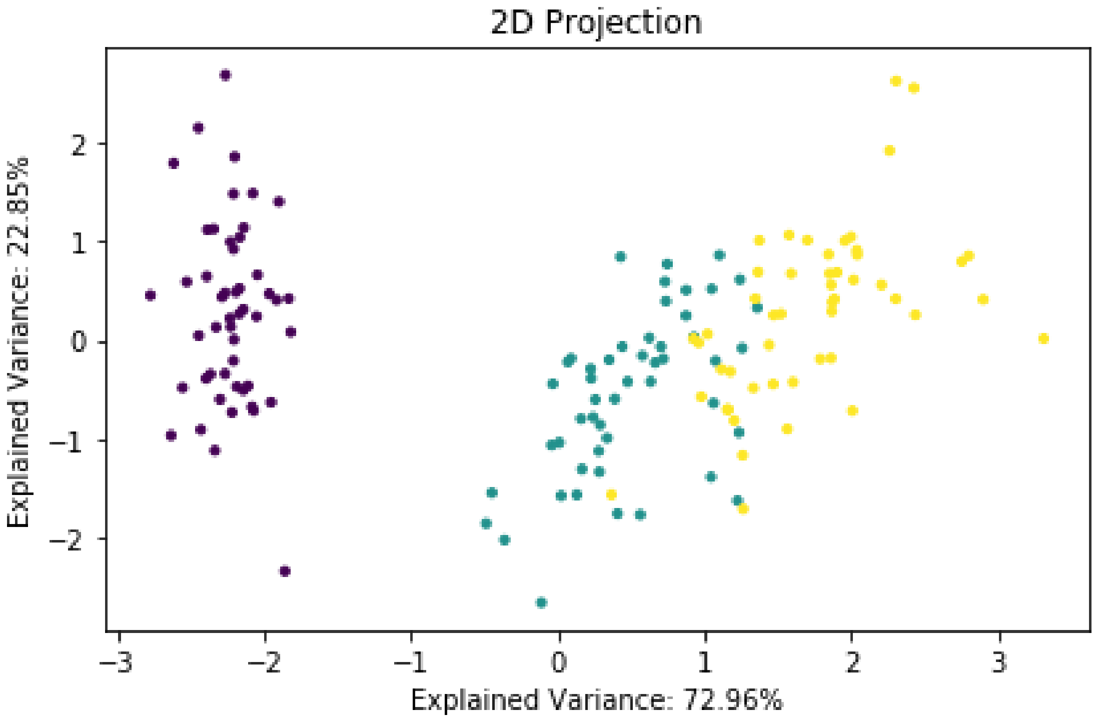
1.3.2 Reduce Dimensionality to visualize clusters

[6]:

```
pca = PCA(n_components=2)
features_2D = pca.fit_transform(features_standardized)
```

[7]:

```
ev1, ev2 = pca.explained_variance_ratio_
ax = plt.figure().gca(title='2D Projection',
                      xlabel='Explained Variance: {:.2%}'.format(ev1),
                      ylabel='Explained Variance: {:.2%}'.format(ev2))
ax.scatter(*features_2D.T, c=data.label, s=10)
plt.tight_layout();
```



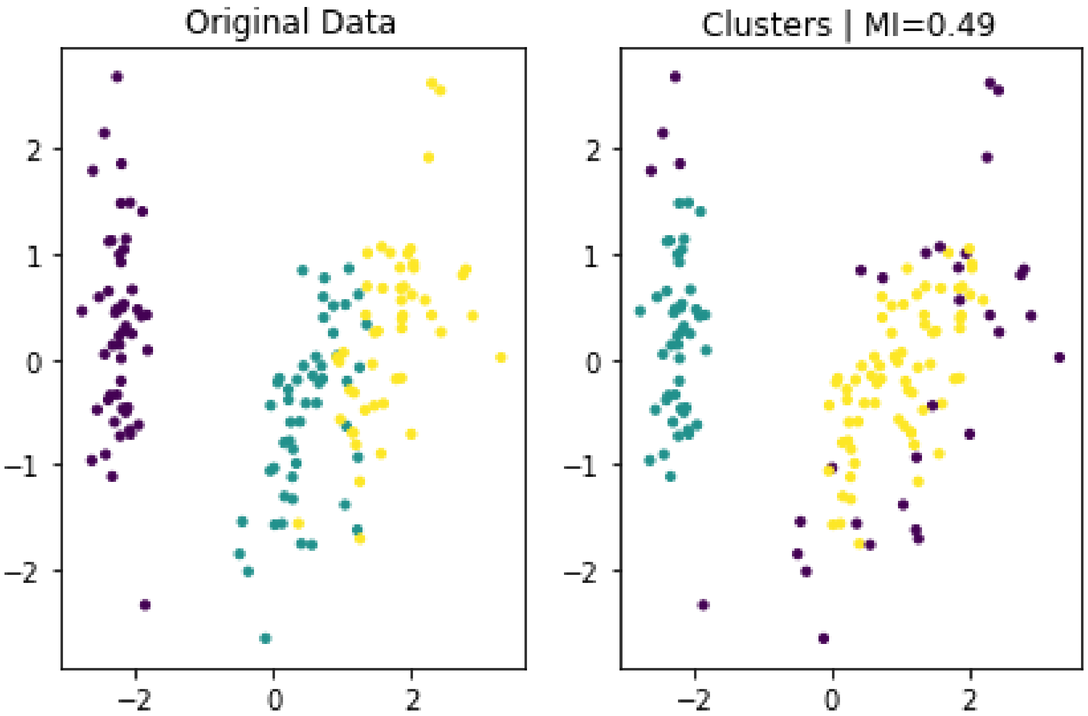
1.4 Perform DBSCAN clustering

Density-based spatial clustering of applications with noise (DBSCAN) has been developed in 1996 and awarded the 'test of time' award at the KDD conference 2014 because of the attention it has received in theory an practice.

It aims to identify core- and non-core samples, where the former extend a cluster and the latter are part of a cluster but do not have sufficient nearby neighbors to further grow the cluster. Other samples are outliers and not assigned to any cluster.

It uses a parameter eps for the radius of the neighborhood and min_samples for the number of members required for core samples. It is deterministic and exclusive and has difficulties with clusters of different density and high-dimensional data. It can be challenging to tune the parameters to the requisite density, especially as it is often not constant.

[8]:
```
clusterer = DBSCAN()
data['clusters'] = clusterer.fit_predict(features_standardized)
fig, axes = plt.subplots(ncols=2)
labels, clusters = data.label, data.clusters
mi = adjusted_mutual_info_score(labels, clusters)
axes[0].scatter(*features_2D.T, c=data.label, s=10)
axes[0].set_title('Original Data')
axes[1].scatter(*features_2D.T, c=data.clusters, s=10)
axes[1].set_title('Clusters | MI={:.2f}'.format(mi))
plt.tight_layout()
```



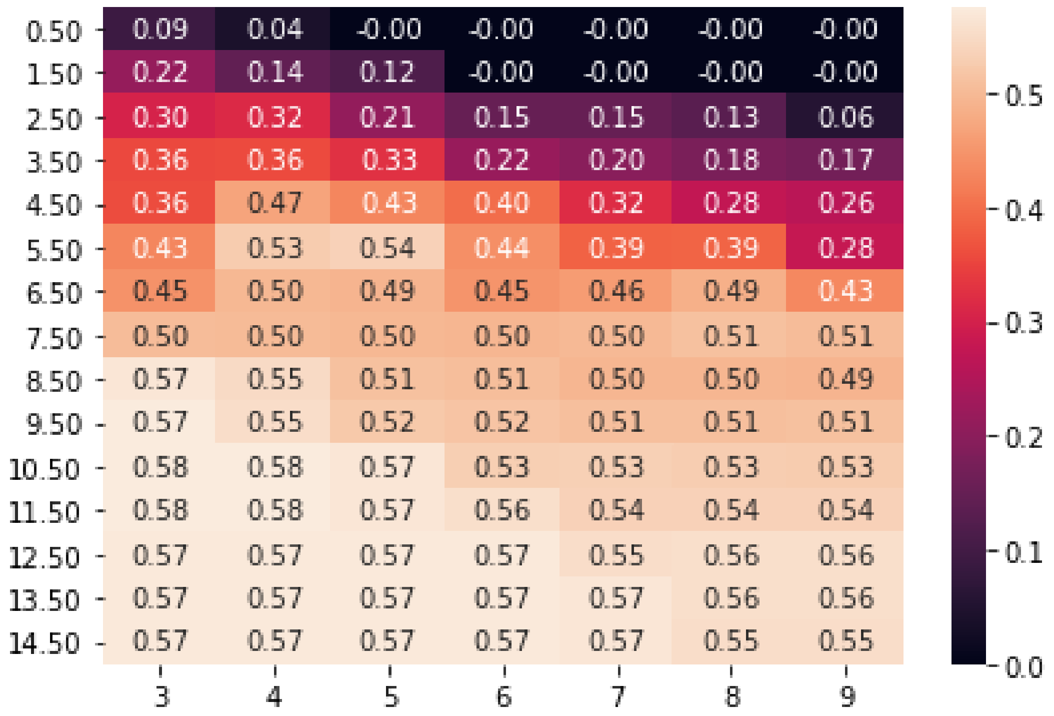
1.4.1 Compare parameter settings

[9]:

```
eps_range = np.arange(.2, .91, .05)
min_samples_range = list(range(3, 10))
labels = data.label
mi = {}
for eps in eps_range:
    for min_samples in min_samples_range:
        clusterer = DBSCAN(eps=eps, min_samples=min_samples)
        clusters = clusterer.fit_predict(features_standardized)
        mi[(eps, min_samples)] = adjusted_mutual_info_score(clusters, labels)
```

[10]:

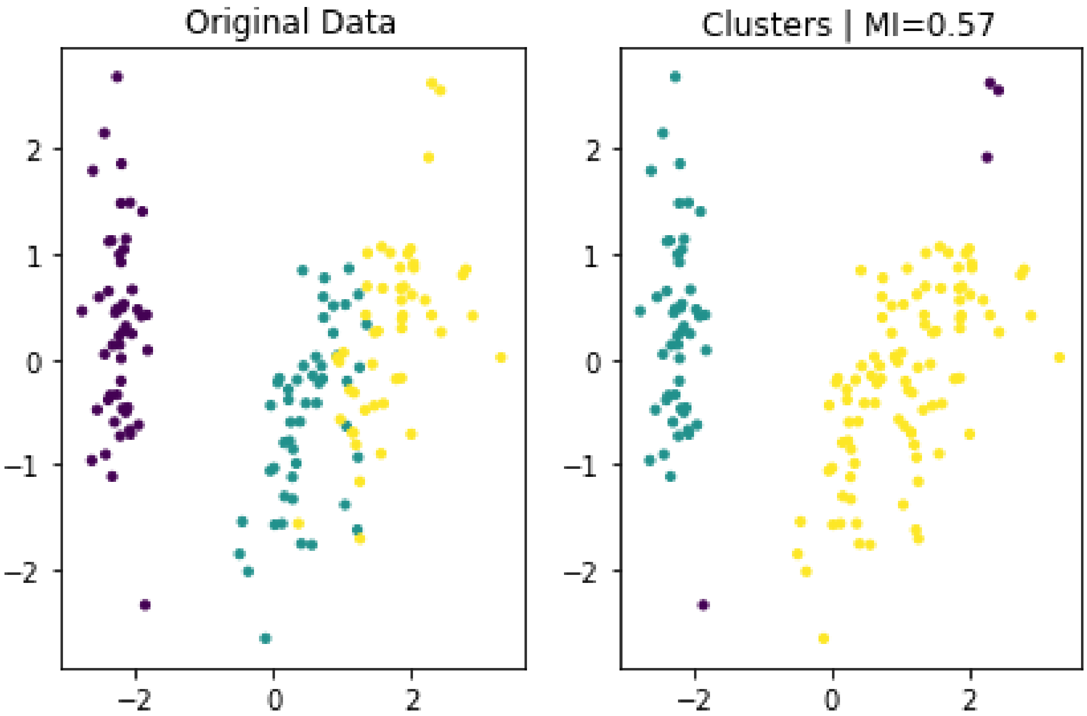
```
results = pd.Series(mi)
results.index = pd.MultiIndex.from_tuples(results.index)
fig, axes = plt.subplots()
ax = sns.heatmap(results.unstack(), annot=True, fmt='.2f')
ax.yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.2f'))
plt.tight_layout()
```



	3	4	5	6	7	8	9
0.50	0.09	0.04	-0.00	-0.00	-0.00	-0.00	-0.00
1.50	0.22	0.14	0.12	-0.00	-0.00	-0.00	-0.00
2.50	0.30	0.32	0.21	0.15	0.15	0.13	0.06
3.50	0.36	0.36	0.33	0.22	0.20	0.18	0.17
4.50	0.36	0.47	0.43	0.40	0.32	0.28	0.26
5.50	0.43	0.53	0.54	0.44	0.39	0.39	0.28
6.50	0.45	0.50	0.49	0.45	0.46	0.49	0.43
7.50	0.50	0.50	0.50	0.50	0.50	0.51	0.51
8.50	0.57	0.55	0.51	0.51	0.50	0.50	0.49
9.50	0.57	0.55	0.52	0.52	0.51	0.51	0.51
10.50	0.58	0.58	0.57	0.53	0.53	0.53	0.53
11.50	0.58	0.58	0.57	0.56	0.54	0.54	0.54
12.50	0.57	0.57	0.57	0.57	0.55	0.56	0.56
13.50	0.57	0.57	0.57	0.57	0.57	0.56	0.56
14.50	0.57	0.57	0.57	0.57	0.57	0.55	0.55

1.4.2 Run again

[11]:
```
clusterer = DBSCAN(eps=.8, min_samples=5)
data['clusters'] = clusterer.fit_predict(features_standardized)
fig, axes = plt.subplots(ncols=2)
labels, clusters = data.label, data.clusters
mi = adjusted_mutual_info_score(labels, clusters)
axes[0].scatter(*features_2D.T, c=data.label, s=10)
axes[0].set_title('Original Data')
axes[1].scatter(*features_2D.T, c=data.clusters, s=10)
axes[1].set_title('Clusters | MI={:.2f}'.format(mi))
plt.tight_layout()
```



1.4.3 DBSCAN in python

Recursive dbscan

[12]:
```python
def run_dbscan(point, members):
    members.add(point)
    neighbors = kdtree.query_radius(atleast_2d(data_[point]), eps)[0]
    if len(neighbors) < min_samples:
        return members | set(neighbors)
    else:
        for neighbor in set(neighbors) - set(members):
            members.update(run_dbscan(neighbor, members))
    return members
```

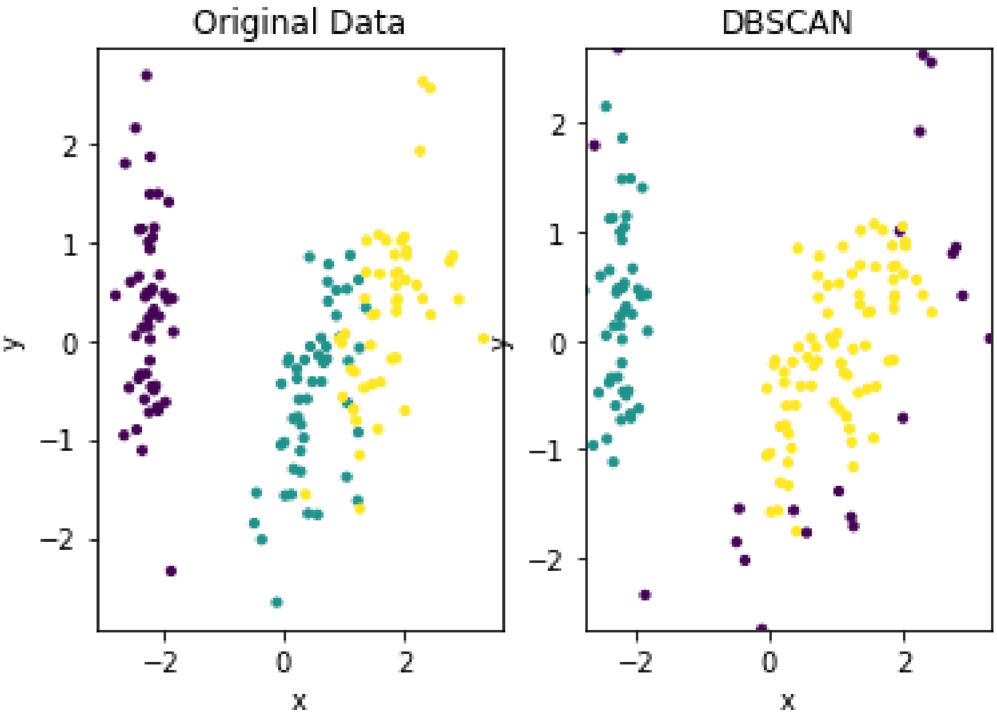
Dynamic Plotting

[13]:
```python
def plot_dbscan(data, assignments, axes, delay=.5):
    for ax in axes:
        ax.clear()
    xmin, ymin = data[['x', 'y']].min()
    xmax, ymax = data[['x', 'y']].max()
    data.plot.scatter(x='x', y='y', c=data.label, cmap=cmap, s=10,
                      title='Original Data', ax=axes[0], colorbar=False)
    plot_data.clusters = plot_data.index.map(assignments.get)
    db_data= data.fillna(0)[data.clusters.notnull()]
    db_data.plot.scatter(x='x', y='y', cmap=cmap, colorbar=False,
                         xlim=(xmin, xmax), ylim=(ymin, ymax),
                         c=db_data.clusters, s=10, title='DBSCAN', ax=axes[1])
    fig.canvas.draw()
    sleep(delay)
```

DBSCAN Execution

```
[14]: eps, min_samples = .6, 5
data_ = features_standardized.copy()
kdtree = KDTree(data_)

to_do = list(range(len(data_)))
plot_data = pd.DataFrame(data=np.c_[features_2D, labels],
                        columns=['x', 'y', 'label']).assign(clusters=np.nan)
shuffle(to_do)
n_clusters = 1
fig, axes = plt.subplots(ncols=2)
assignments = {}
while to_do:
    item = to_do.pop()
    neighbors = kdtree.query_radius(atleast_2d(data_[item, :]), eps)[0]
    if len(neighbors) < min_samples:
        assignments[item] = 0
        plot_dbscan(plot_data, assignments, axes)
    else:
        new_cluster = run_dbscan(item, set())
        to_do = [t for t in to_do if t not in new_cluster]
        for member in new_cluster:
            assignments.update({member: n_clusters})
        n_clusters += 1
```



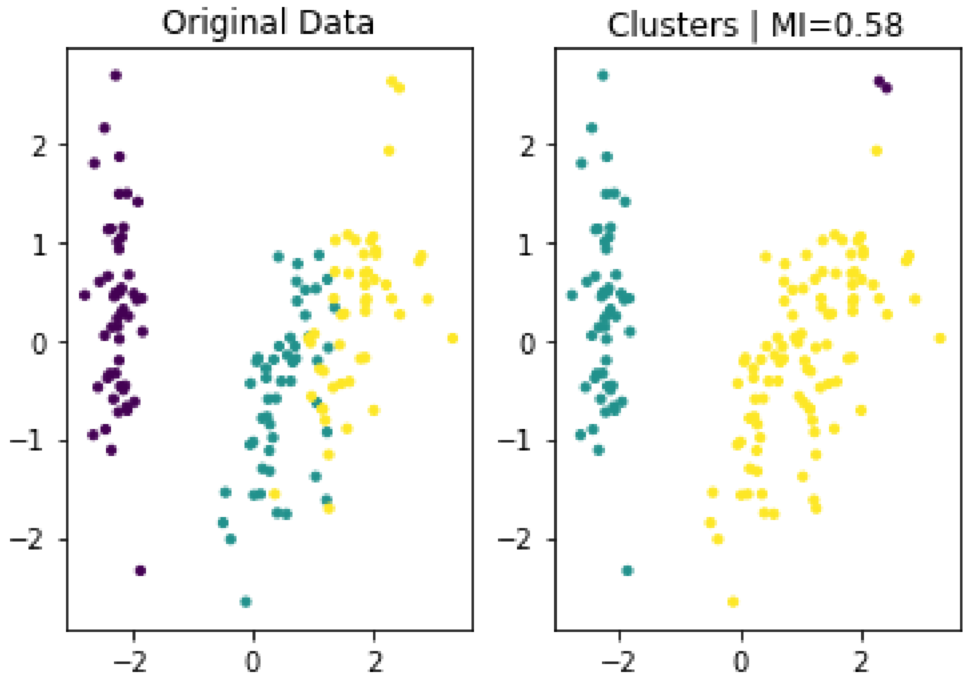
1.4.4 HDBSCAN

Hierarchical DBSCAN is a more recent development that assumes clusters are islands of potentially differing density to overcome the DBSCAN challenges just mentioned. It also aims to identify the core and non-core samples. It uses the parameters min_cluster_ size, and min_samples to select a neighborhood and extend a cluster. The algorithm iterates over multiple eps values and chooses the most stable clustering. In addition to identifying clusters of varying density, it provides insight into the density and hierarchical structure of the data.

The following figures show how DBSCAN and HDBSCAN are able to identify very differently shaped clusters.

[15]:
```
clusterer = HDBSCAN()
data['clusters'] = clusterer.fit_predict(features_standardized)
fig, axes = plt.subplots(ncols=2)
labels, clusters = data.label, data.clusters
mi = adjusted_mutual_info_score(labels, clusters)
axes[0].scatter(*features_2D.T, c=data.label, s=10)
axes[0].set_title('Original Data')
axes[1].scatter(*features_2D.T, c=data.clusters, s=10)
axes[1].set_title('Clusters | MI={:.2f}'.format(mi))
```

[15]:
```
Text(0.5,1,'Clusters | MI=0.58')
```

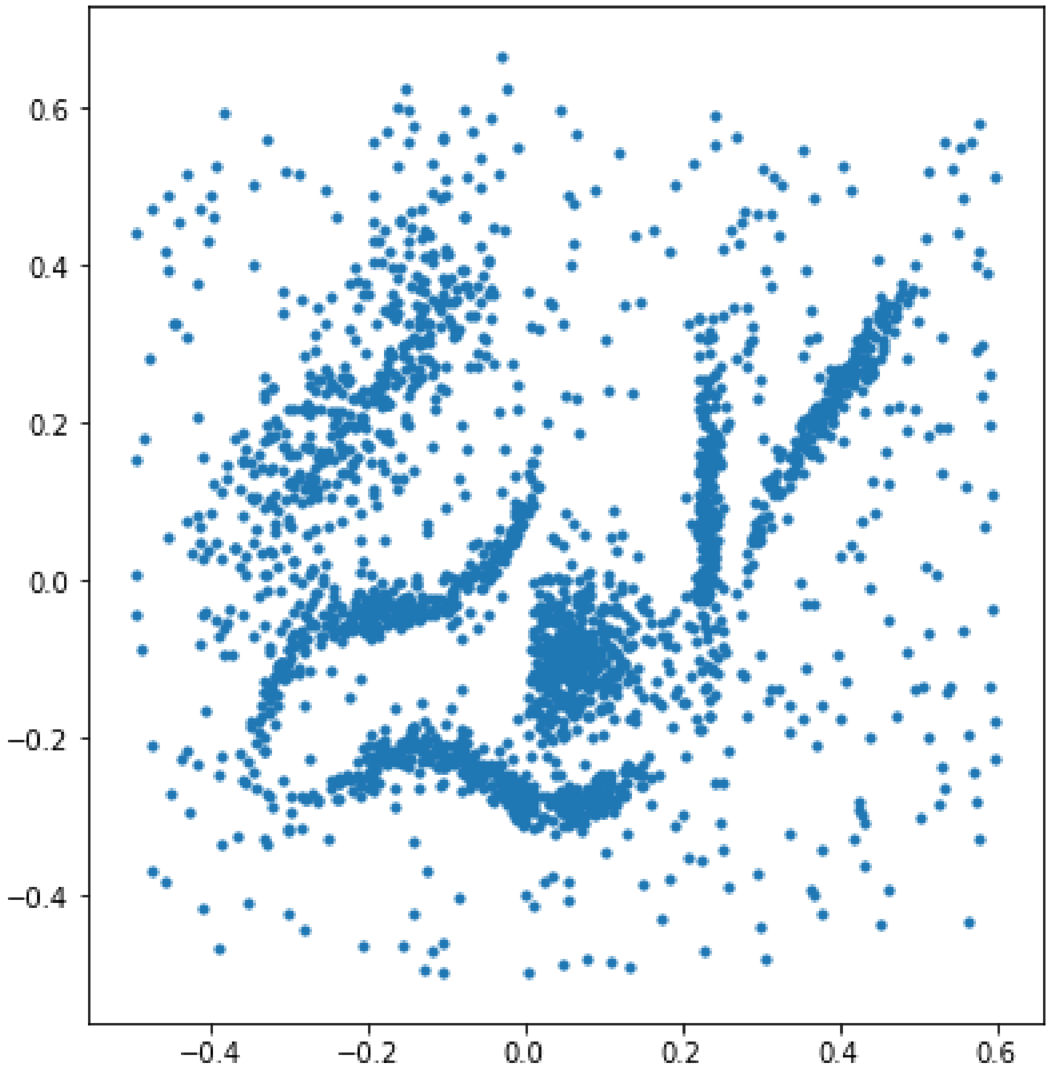

Original Data

Clusters | MI=0.58

1.4.5 Alternative Dataset

[16]:

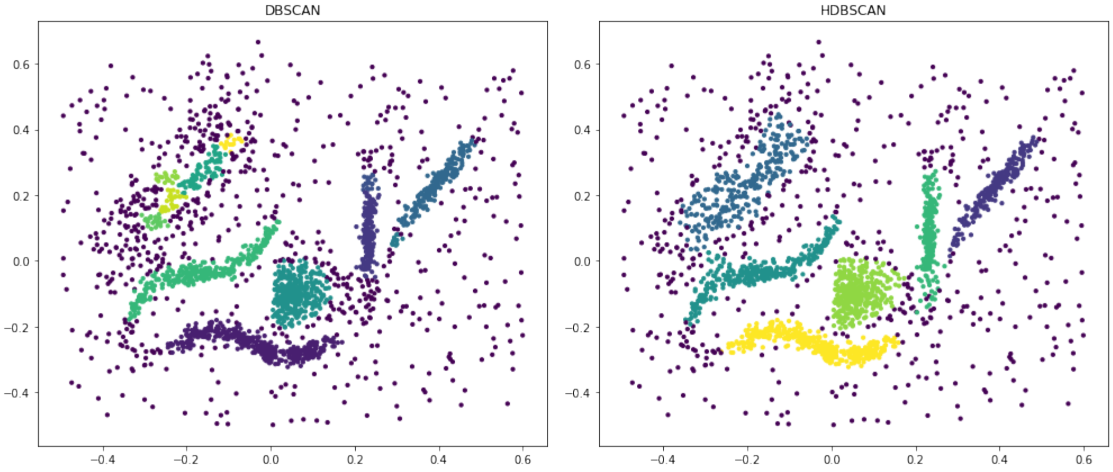
```
alternative_data = np.load('clusterable_data.npy')
fig, ax = plt.subplots(figsize=(7,7))
ax.set_aspect('equal')
ax.scatter(*alternative_data.T, s=10);
```



1.4.6 Compare DBSCAN & HDBSCAN

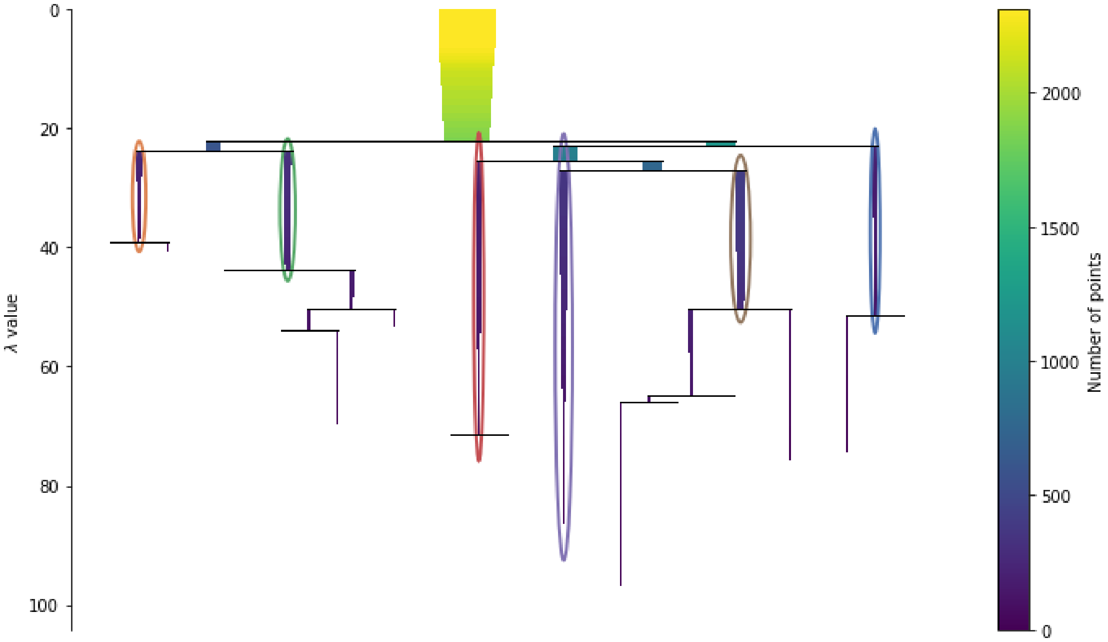
[17]:

```
dbscan = DBSCAN(eps=.02, min_samples=10)
hdbscan = HDBSCAN(min_cluster_size=15, gen_min_span_tree=True)
db_clusters = dbscan.fit_predict(alternative_data)
hdb_clusters = hdbscan.fit_predict(alternative_data)
fig, axes = plt.subplots(ncols=2, figsize=(14,6))
axes[0].scatter(*alternative_data.T, c=db_clusters, s=10, cmap=cmap)
axes[0].set_title('DBSCAN')
axes[1].scatter(*alternative_data.T, c=hdb_clusters, s=10, cmap=cmap)
axes[1].set_title('HDBSCAN');
fig.tight_layout()
```



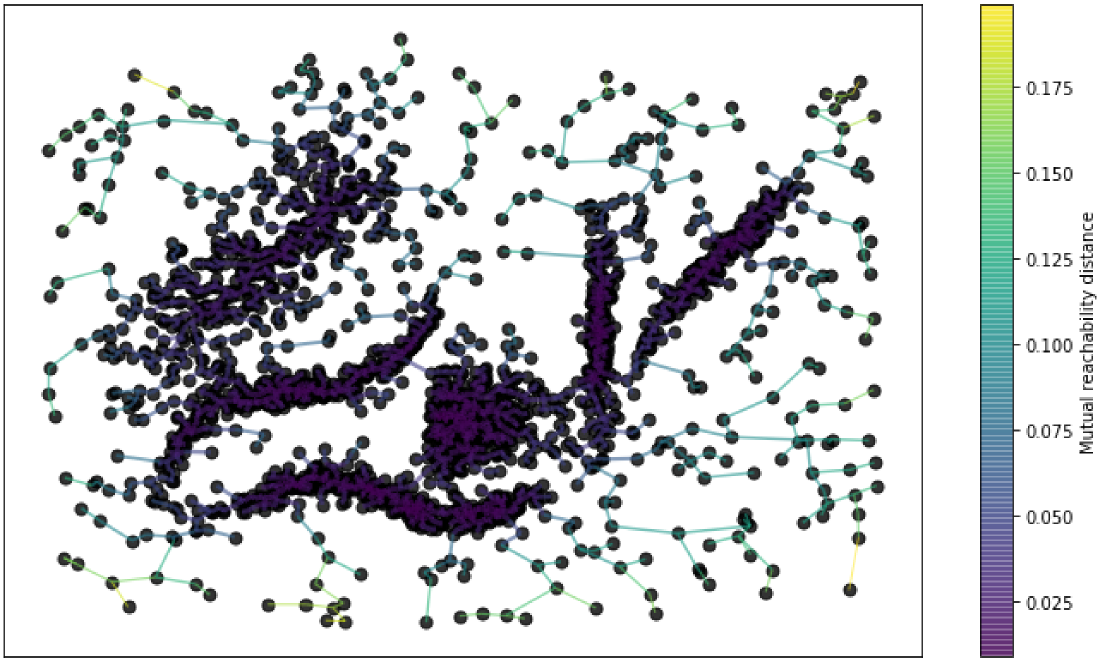
1.4.7 HDBSCAN: Density-based Dendrogram

```
[18]: fig, ax = plt.subplots(figsize=(12,7))
      hdbscan.condensed_tree_.plot(select_clusters=True,
                                   selection_palette=sns.color_palette('deep', 8));
```



1.4.8 Minimum Spanning Tree

[19]:
```
fig, ax = plt.subplots(figsize=(12,7))
hdbscan.minimum_spanning_tree_.plot(edge_cmap='viridis',
                                      edge_alpha=0.6,
                                      node_size=50,
                                      edge_linewidth=1);
```


Mutual reachability distance