Clustering with Python

Methods in Scikit-learn

Nicolás García-Pedrajas

Computational Intelligence and Bioinformatics Research Group

November 13, 2019

Table of contents

Introduction

Avaliable methods:

- *k*-Means.
- Affinity propagation.
- Mean-shift.
- Spectral clustering.
- Ward hierarchical clustering.
- Agglomerative clustering.
- DBSCAN.
- Optics.
- Gaussian mixtures.
- Birch.

Introduction

Clustering of unlabeled data can be performed with the module sklearn.cluster.
Each clustering algorithm comes in two variants:

- a class, that implements the fit method to learn the clusters on train data
- a function, that, given train data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the labels_ attribute.

k-Means

Class: kMeans

Methods:

fit(self, X[, y, sample_weight])	Compute k-means clustering.
fit_predict(self, X[, y, sample_weight])	Compute cluster centers and predict cluster index for each sample.
fit_transform(self, X[, y, sample_weight])	Compute clustering and transform X to cluster-distance space.
get_params(self[, deep])	Get parameters for this estimator.
predict(self, X[, sample_weight])	Predict the closest cluster each sample in X belongs to.
score(self, X[, y, sample_weight])	Opposite of the value of X on the K-means objective.
set_params(self, params)	Set the parameters of this estimator.
transform(self, X)	Transform X to a cluster-distance space.

k-Means

Example

k-Means example

```
>>> from sklearn.cluster import KMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...                 [10, 2], [10, 4], [10, 0]])
>>> kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
>>> kmeans.labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> kmeans.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32)
>>> kmeans.cluster_centers_
array([[10.,  2.],
[ 1.,  2.]])
```

Alternatively you can use `fit_predict()` with the same effect than `fit()` and then `predict()`.

k-Means I

Example

```
#!        /usr/bin/python

"""
==================================================
Demo of affinity propagation clustering algorithm
==================================================

Reference:
Brendan J. Frey and Delbert Dueck, "Clustering by Passing Messages
Between Data Points", Science Feb. 2007

"""
print(__doc__)

from sklearn.cluster import AffinityPropagation
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs

# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=300, centers=centers,
↪  cluster_std=0.5,
```

k-Means II

Example

```python
                                     random_state=0)

# Compute Affinity Propagation
af = AffinityPropagation(preference=-50).fit(X)
cluster_centers_indices = af.cluster_centers_indices_
labels = af.labels_

n_clusters_ = len(cluster_centers_indices)

print('Estimated number of clusters: %d' % n_clusters_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true,
↪  labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true,
↪  labels))
print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"
      % metrics.adjusted_rand_score(labels_true, labels))
print("Adjusted Mutual Information: %0.3f"
      % metrics.adjusted_mutual_info_score(labels_true, labels,
                                           average_method='arithmetic'))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, labels, metric='sqeuclidean'))
```

k-Means III

Example

```
# Plot result
import matplotlib.pyplot as plt
from itertools import cycle

plt.close('all')
plt.figure(1)
plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    class_members = labels == k
    cluster_center = X[cluster_centers_indices[k]]
    plt.plot(X[class_members, 0], X[class_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o',
    ↪  markerfacecolor=col,
             markeredgecolor='k', markersize=14)
    for x in X[class_members]:
        plt.plot([cluster_center[0], x[0]], [cluster_center[1], x[1]],
        ↪  col)
```

k-Means IV

Example

```
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

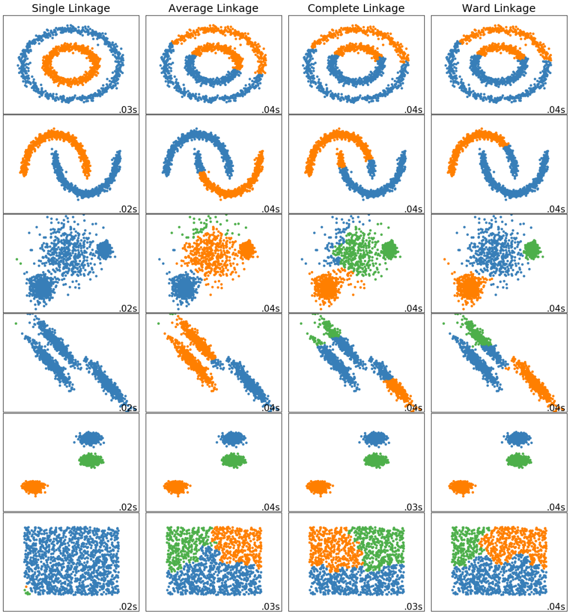
Hierarchical clustering

Agglomerative clustering

The `AgglomerativeClustering` object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:

- Ward minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.
- Maximum or complete linkage minimizes the maximum distance between observations of pairs of clusters.
- Average linkage minimizes the average of the distances between all observations of pairs of clusters.
- Single linkage minimizes the distance between the closest observations of pairs of clusters.

Dendrograms can also be plotted.


Single Linkage
Average Linkage
Complete Linkage
Ward Linkage
.03s
.04s
.04s
.04s
.02s
.04s
.04s
.04s
.02s
.04s
.04s
.04s
.02s
.04s
.04s
.04s
.02s
.04s
.03s
.04s
.02s
.03s
.03s
.04s

DBSCAN

The DBSCAN algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped.
The central component to the DBSCAN is the concept of core samples, which are samples that are in areas of high density.
A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples).
There are two parameters to the algorithm, `min_samples` and `eps`, which define formally what we mean when we say dense. Higher min_samples or lower eps indicate higher density necessary to form a cluster.

Example of DBSCAN I

```
#!        /usr/bin/python
# -*- coding: utf-8 -*-
"""
===================================
Demo of DBSCAN clustering algorithm
===================================

Finds core samples of high density and expands clusters from them.

"""
print(__doc__)

import numpy as np

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler

# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=750, centers=centers,
↪  cluster_std=0.4,
```

Example of DBSCAN II

```
                                      random_state=0)

X = StandardScaler().fit_transform(X)

# Compute DBSCAN
db = DBSCAN(eps=0.3, min_samples=10).fit(X)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true,
↪   labels))
print("Completeness: %0.3f" % metrics.completeness_score(labels_true,
↪   labels))
print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
print("Adjusted Rand Index: %0.3f"
      % metrics.adjusted_rand_score(labels_true, labels))
```

Example of DBSCAN III

```
print("Adjusted Mutual Information: %0.3f"
      % metrics.adjusted_mutual_info_score(labels_true, labels,
                                           average_method='arithmetic'))
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(X, labels))

# Plot result
import matplotlib.pyplot as plt

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)
```

Example of DBSCAN IV

```
    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

There are several metrics that can only be used when the “ground truth” (aka the class labels) is known.
These metrics are not appropriate to the unsupervised case.
For unsupervised case scikit-learn implements several metrics:

- Silhouette Coefficient
- Calinski-Harabasz Index
- Davies-Bouldin Index

Evaluation

Example of use of Silhouette index

In normal usage, the Silhouette Coefficient is applied to the results of a cluster analysis:

Metric example

```python
from sklearn import metrics
from sklearn.metrics import pairwise_distances
from sklearn import datasets
import numpy as np
from sklearn.cluster import KMeans

dataset = datasets.load_iris()
X = dataset.data
y = dataset.target
kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
labels = kmeans_model.labels_
ss = metrics.silhouette_score(X, labels, metric='euclidean')
```