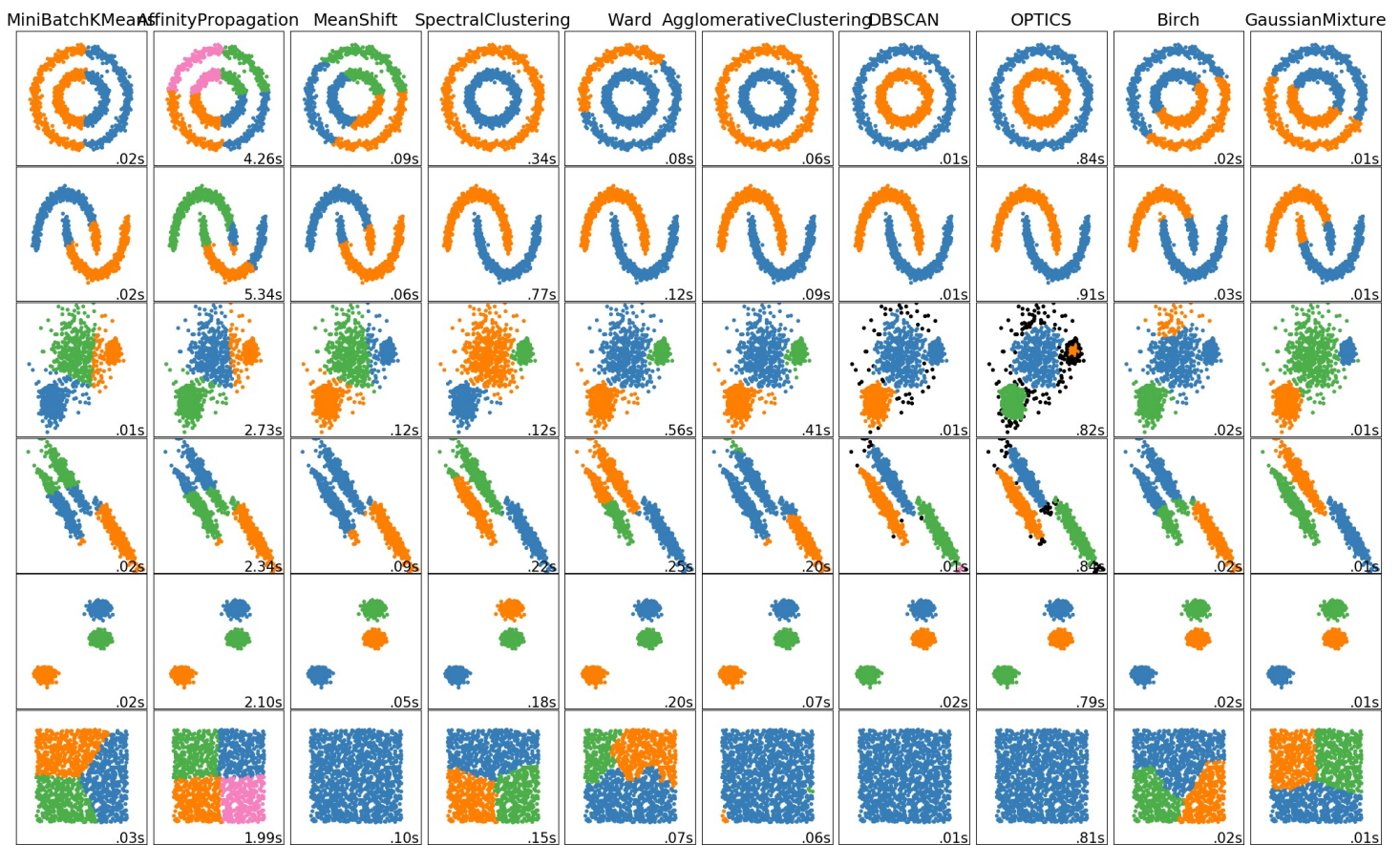# 2.3. Clustering

[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, and 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.

> **Input data**
>
> One important thing to note is that the algorithms implemented in this module can take different kinds of matrix as input. All the methods accept standard data matrices of shape `[n_samples, n_features]`. These can be obtained from the classes in the `sklearn.feature_extraction` module. For `AffinityPropagation`, `SpectralClustering` and `DBSCAN` one can also input similarity matrices of shape `[n_samples, n_samples]`. These can be obtained from the functions in the `sklearn.metrics.pairwise` module.

2.3.1. Overview of clustering methods

A comparison of the clustering algorithms in scikit-learn

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
[K-Means](#)	number of clusters	Very large `n_samples`, medium `n_clusters` with [MiniBatch code](#)	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
[Affinity propagation](#)	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
[Mean-shift](#)	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
[Spectral clustering](#)	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
[Ward hierarchical clustering](#)	number of clusters or distance threshold	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
[Agglomerative clustering](#)	number of clusters or distance threshold, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance

DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
OPTICS	minimum cluster membership	Very large `n_samples`, large `n_clusters`	Non-flat geometry, uneven cluster sizes, variable cluster density	Distances between points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large `n_clusters` and `n_samples`	Large dataset, outlier removal, data reduction.	Euclidean distance between points

Non-flat geometry clustering is useful when the clusters have a specific shape, i.e. a non-flat manifold, and the standard euclidean distance is not the right metric. This case arises in the two top rows of the figure above.

Gaussian mixture models, useful for clustering, are described in [another chapter of the documentation](#) dedicated to mixture models. KMeans can be seen as a special case of Gaussian mixture model with equal covariance per component.

2.3.2. K-means

The `KMeans` algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the *inertia* or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields.

The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean μ_j of the samples in the cluster. The means are commonly called the cluster "centroids"; note that they are not, in general, points from X, although they live in the same space.

The K-means algorithm aims to choose centroids that minimise the **inertia**, or **within-cluster sum-of-squares criterion**:

$$\sum_{i=0}^{n} \min_{\mu_j \in C}(||x_i - \mu_j||^2)$$

Inertia can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks:

- Inertia makes the assumption that clusters are convex and isotropic, which is not always the case. It responds poorly to elongated clusters, or manifolds with irregular shapes.
- Inertia is not a normalized metric: we just know that lower values are better and zero is optimal. But in very high-dimensional spaces, Euclidean distances tend to become inflated (this is an instance of the so-called "curse of dimensionality"). Running a dimensionality reduction algorithm such as [Principal component analysis (PCA)](#) prior to k-means clustering can alleviate this problem and speed up the computations.
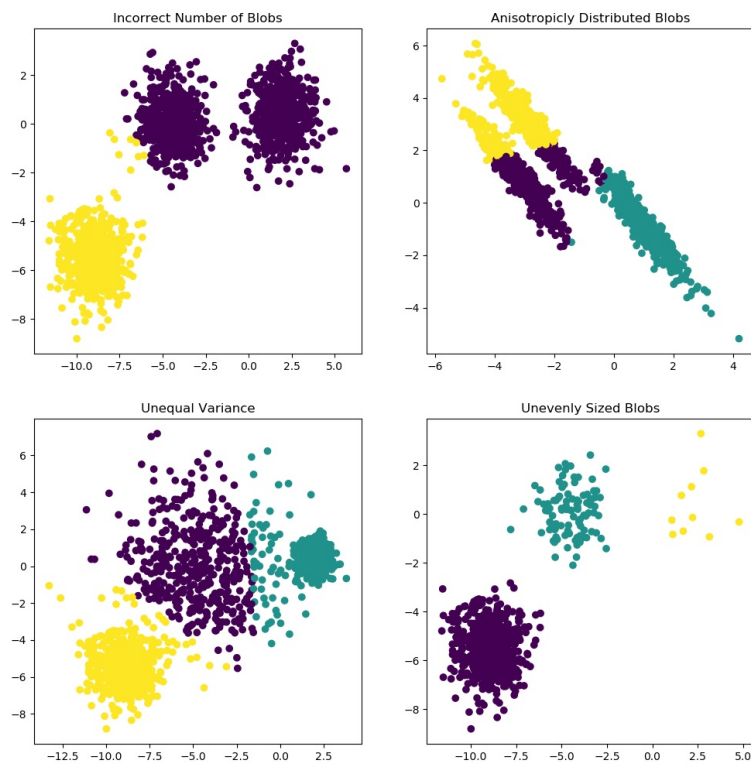
K-means is often referred to as Lloyd's algorithm. In basic terms, the algorithm has three steps. The first step chooses the initial centroids, with the most basic method being to choose k samples from the dataset X. After initialization, K-means consists of looping between the two other steps. The first step assigns each sample to its nearest centroid. The second step creates new centroids by taking the mean value of all of the samples assigned to each previous centroid. The difference between the old and the new centroids are computed and the algorithm repeats these last two steps until this value is less than a threshold. In other words, it repeats until the centroids do not move significantly.

K-means is equivalent to the expectation-maximization algorithm with a small, all-equal, diagonal covariance matrix.

The algorithm can also be understood through the concept of Voronoi diagrams. First the Voronoi diagram of the points is calculated using the current centroids. Each segment in the Voronoi diagram becomes a separate cluster. Secondly, the centroids are updated to the mean of each segment. The algorithm then repeats this until a stopping criterion is fulfilled. Usually, the algorithm stops when the relative decrease in the objective function between iterations is less than the given tolerance value. This is not the case in this implementation: iteration stops when centroids move less than the tolerance.

Given enough time, K-means will always converge, however this may be to a local minimum. This is highly dependent on the initialization of the centroids. As a result, the computation is often done several times, with different initializations of the centroids. One method to help address this issue is the k-means++ initialization scheme, which has been implemented in scikit-learn (use the `init='k-means++'` parameter). This initializes the centroids to be (generally) distant from each other, leading to provably better results than random initialization, as shown in the reference.

The algorithm supports sample weights, which can be given by a parameter `sample_weight`. This allows to assign more weight to some samples when computing cluster centers and values of inertia. For example, assigning a weight of 2 to a sample is equivalent to adding a duplicate of that sample to the dataset X.

A parameter can be given to allow K-means to be run in parallel, called `n_jobs`. Giving this parameter a positive value uses that many processors (default: 1). A value of -1 uses all available processors, with -2 using one less, and so on. Parallelization generally speeds up computation at the cost of memory (in this case, multiple copies of centroids need to be stored, one for each job).

> **Warning:** The parallel version of K-Means is broken on OS X when `numpy` uses the `Accelerate` Framework. This is expected behavior: `Accelerate` can be called after a fork but you need to execv the subprocess with the Python binary (which multiprocessing does not do under posix).

K-means can be used for vector quantization. This is achieved using the transform method of a trained model of `KMeans`.

References:

- ["k-means++: The advantages of careful seeding"](#) Arthur, David, and Sergei Vassilvitskii, *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (2007)

2.3.2.1. Mini Batch K-Means

The `MiniBatchKMeans` is a variant of the `KMeans` algorithm which uses mini-batches to reduce the computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration. These mini-batches drastically reduce the amount of computation required to converge to a local solution. In contrast to other algorithms that reduce the convergence time of k-means, mini-batch k-means produces results that are generally only slightly worse than the standard algorithm.

The algorithm iterates between two major steps, similar to vanilla k-means. In the first step, b samples are drawn randomly from the dataset, to form a mini-batch. These are then assigned to the nearest centroid. In the second step, the centroids are updated. In contrast to k-means, this is done on a per-sample basis. For each sample in the mini-batch, the assigned centroid is updated by taking the streaming average of the sample and all previous samples assigned to that centroid. This has the effect of decreasing the rate of change for a centroid over time. These steps are performed until convergence or a predetermined number of iterations is reached.

`MiniBatchKMeans` converges faster than `KMeans`, but the quality of the results is reduced. In practice this difference in quality can be quite small, as shown in the example and cited reference.
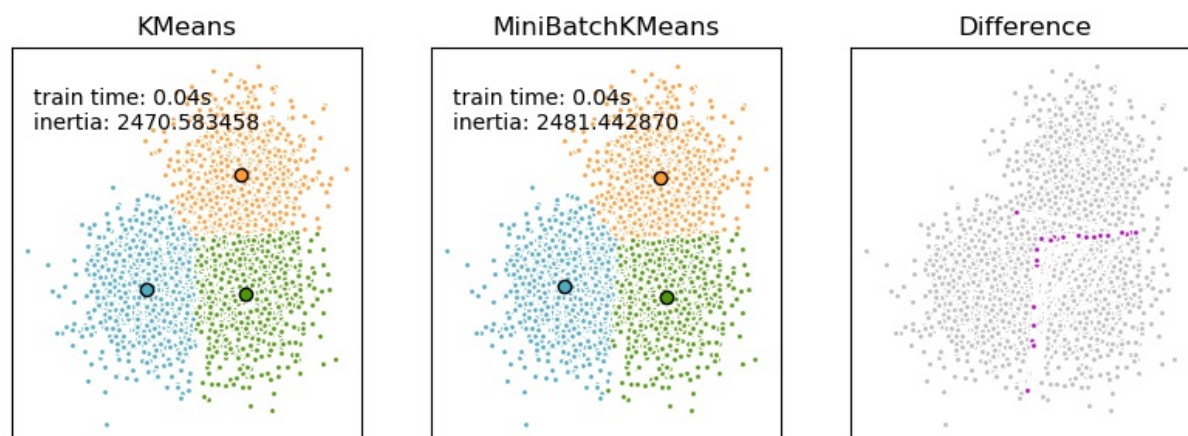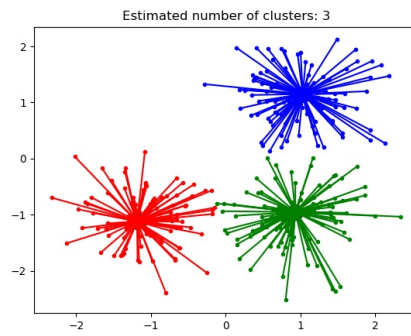
References:

- ["Web Scale K-Means clustering"](#) D. Sculley, *Proceedings of the 19th international conference on World wide web* (2010)

2.3.3. Affinity Propagation

`AffinityPropagation` creates clusters by sending messages between pairs of samples until convergence. A dataset is then described using a small number of exemplars, which are identified as those most representative of other samples. The messages sent between pairs represent the suitability for one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and hence the final clustering is given.

Estimated number of clusters: 3

Affinity Propagation can be interesting as it chooses the number of clusters based on the data provided. For this purpose, the two important parameters are the *preference*, which controls how many exemplars are used, and the *damping factor* which damps the responsibility and availability messages to avoid numerical oscillations when updating these messages.

The main drawback of Affinity Propagation is its complexity. The algorithm has a time complexity of the order $O(N^2 T)$, where N is the number of samples and T is the number of iterations until convergence. Further, the memory complexity is of the order $O(N^2)$ if a dense similarity matrix is used, but reducible if a sparse similarity matrix is used. This makes Affinity Propagation most appropriate for small to medium sized datasets.

> **Examples:**
> - [Demo of affinity propagation clustering algorithm](): Affinity Propagation on a synthetic 2D datasets with 3 classes.
> - [Visualizing the stock market structure]() Affinity Propagation on Financial time series to find groups of companies

Algorithm description: The messages sent between points belong to one of two categories. The first is the responsibility $r(i, k)$, which is the accumulated evidence that sample k should be the exemplar for sample i. The second is the availability $a(i, k)$ which is the accumulated evidence that sample i should choose sample k to be its exemplar, and considers the values for all other samples that k should be an exemplar. In this way, exemplars are chosen by samples if they are (1) similar enough to many samples and (2) chosen by many samples to be representative of themselves.

More formally, the responsibility of a sample k to be the exemplar of sample i is given by:

$$r(i, k) \leftarrow s(i, k) - max[a(i, k') + s(i, k') \forall k' \neq k]$$

Where $s(i, k)$ is the similarity between samples i and k. The availability of sample k to be the exemplar of sample i is given by:

$$a(i, k) \leftarrow min[0, r(k, k) + \sum_{i' \ s.t. \ i' \notin \{i,k\}} r(i', k)]$$

To begin with, all values for r and a are set to zero, and the calculation of each iterates until convergence. As discussed above, in order to avoid numerical oscillations when updating the messages, the damping factor λ is introduced to iteration process:

$$r_{t+1}(i, k) = \lambda \cdot r_t(i, k) + (1 - \lambda) \cdot r_{t+1}(i, k)$$
$$a_{t+1}(i, k) = \lambda \cdot a_t(i, k) + (1 - \lambda) \cdot a_{t+1}(i, k)$$

where t indicates the iteration times.

2.3.4. Mean Shift

`MeanShift` clustering aims to discover *blobs* in a smooth density of samples. It is a centroid based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

Given a candidate centroid x_i for iteration t, the candidate is updated according to the following equation:

$$x_i^{t+1} = m(x_i^t)$$

Where $N(x_i)$ is the neighborhood of samples within a given distance around x_i and m is the *mean shift* vector that is computed for each centroid that points towards a region of the maximum increase in the density of points. This is computed using the following equation, effectively updating a centroid to be the mean of the samples within its neighborhood:

$$m(x_i) = \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i) x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)}$$

The algorithm automatically sets the number of clusters, instead of relying on a parameter `bandwidth`, which dictates the size of the region to search through. This parameter can be set manually, but can be estimated using the provided `estimate_bandwidth` function, which is called if the bandwidth is not set.

The algorithm is not highly scalable, as it requires multiple nearest neighbor searches during the execution of the algorithm. The algorithm is guaranteed to converge, however the algorithm will stop iterating when the change in centroids is small.

Labelling a new sample is performed by finding the nearest centroid for a given sample.

Estimated number of clusters: 3
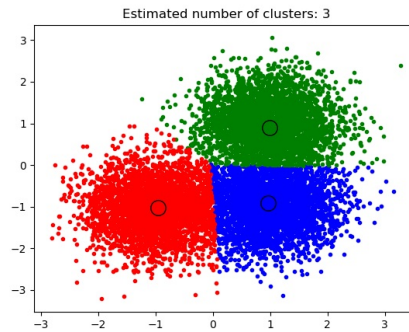
Examples:

- A demo of the mean-shift clustering algorithm: Mean Shift clustering on a synthetic 2D datasets with 3 classes.

References:

- "Mean shift: A robust approach toward feature space analysis." D. Comaniciu and P. Meer, *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2002)

2.3.5. Spectral clustering

`SpectralClustering` performs a low-dimension embedding of the affinity matrix between samples, followed by clustering, e.g., by KMeans, of the components of the eigenvectors in the low dimensional space. It is especially computationally efficient if the affinity matrix is sparse and the `amg` solver is used for the eigenvalue problem (Note, the `amg` solver requires that the pyamg module is installed.)

The present version of SpectralClustering requires the number of clusters to be specified in advance. It works well for a small number of clusters, but is not advised for many clusters.

For two clusters, SpectralClustering solves a convex relaxation of the normalised cuts problem on the similarity graph: cutting the graph in two so that the weight of the edges cut is small compared to the weights of the edges inside each cluster. This criteria is especially interesting when working on images, where graph vertices are pixels, and weights of the edges of the similarity graph are computed using a function of a gradient of the image.
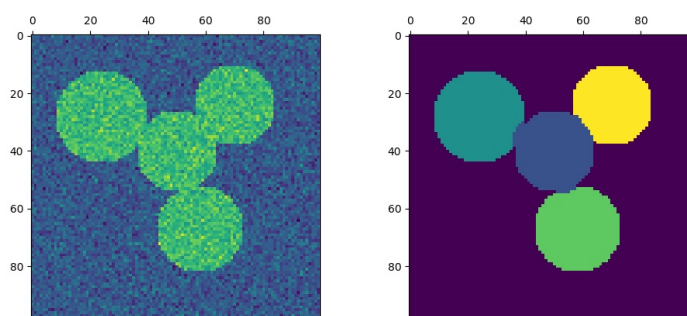
Warning: Transforming distance to well-behaved similarities

Note that if the values of your similarity matrix are not well distributed, e.g. with negative values or with a distance matrix rather than a similarity, the spectral problem will be singular and the problem not solvable. In which case it is advised to apply a transformation to the entries of the matrix. For instance, in the case of a signed distance matrix, is common to apply a heat kernel:

```
similarity = np.exp(-beta * distance / distance.std())
```

See the examples for such an application.

Examples:
- Spectral clustering for image segmentation: Segmenting objects from a noisy background using spectral clustering.
- Segmenting the picture of greek coins in regions: Spectral clustering to split the image of coins in regions.

2.3.5.1. Different label assignment strategies

Different label assignment strategies can be used, corresponding to the `assign_labels` parameter of `SpectralClustering`. `"kmeans"` strategy can match finer details, but can be unstable. In particular, unless you control the `random_state`, it may not be reproducible from run-to-run, as it depends on random initialization. The alternative `"discretize"` strategy is 100% reproducible, but tends to create parcels of fairly even and geometrical shape.
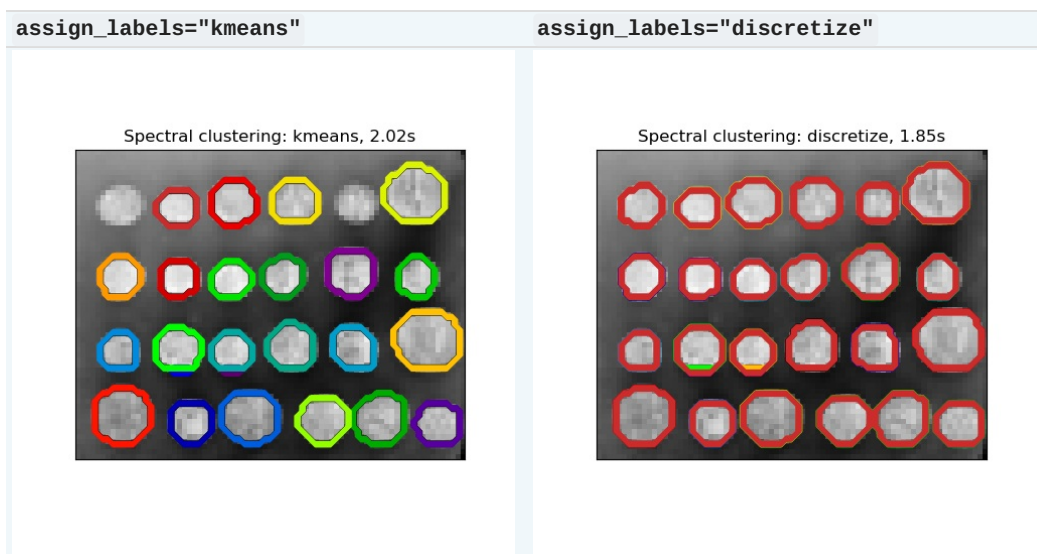
2.3.5.2. Spectral Clustering Graphs

Spectral Clustering can also be used to partition graphs via their spectral embeddings. In this case, the affinity matrix is the adjacency matrix of the graph, and SpectralClustering is initialized with `affinity='precomputed'`:

```
>>> from sklearn.cluster import SpectralClustering
>>> sc = SpectralClustering(3, affinity='precomputed', n_init=100,
...                         assign_labels='discretize')
>>> sc.fit_predict(adjacency_matrix)
```

References:
- "A Tutorial on Spectral Clustering" Ulrike von Luxburg, 2007
- "Normalized cuts and image segmentation" Jianbo Shi, Jitendra Malik, 2000
- "A Random Walks View of Spectral Segmentation" Marina Meila, Jianbo Shi, 2001
- "On Spectral Clustering: Analysis and an algorithm" Andrew Y. Ng, Michael I. Jordan, Yair Weiss, 2001
- "Preconditioned Spectral Clustering for Stochastic Block Partition Streaming Graph Challenge" David Zhuzhunashvili, Andrew Knyazev

2.3.6. Hierarchical clustering

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample. See the Wikipedia page for more details.

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.
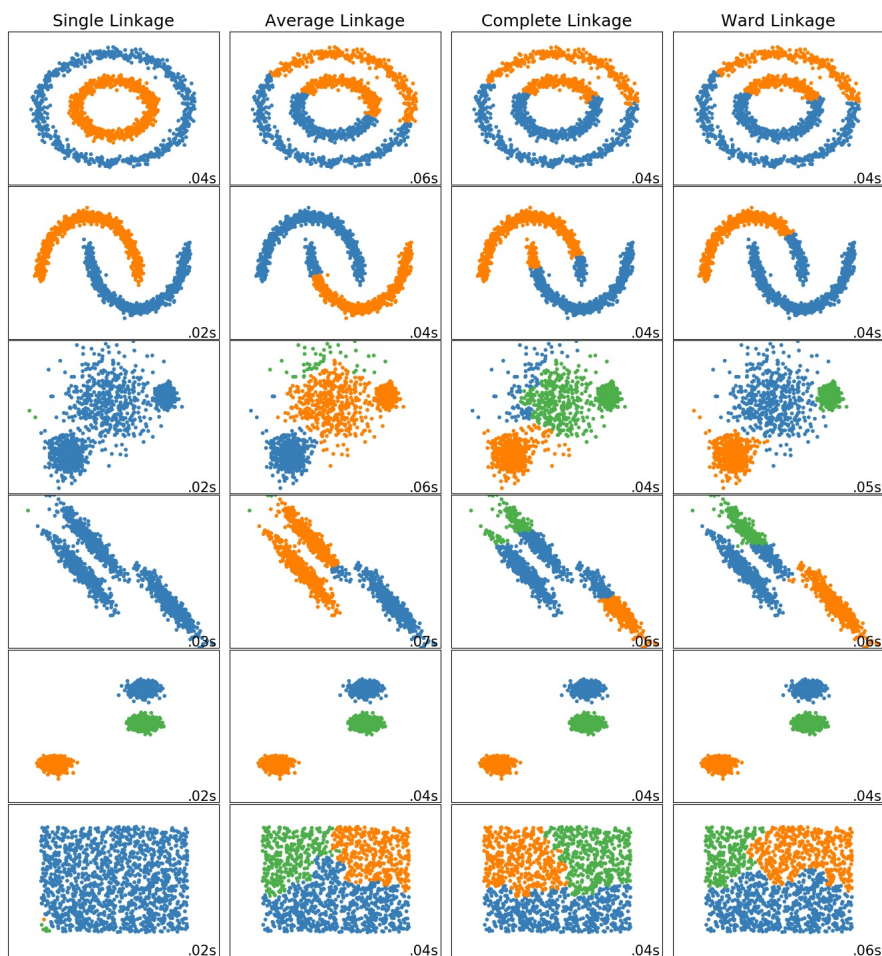
`AgglomerativeClustering` can also scale to large number of samples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all the possible merges.

`FeatureAgglomeration`

The **`FeatureAgglomeration`** uses agglomerative clustering to group together features that look very similar, thus decreasing the number of features. It is a dimensionality reduction tool, see Unsupervised dimensionality reduction.

2.3.6.1. Different linkage type: Ward, complete, average, and single linkage

`AgglomerativeClustering` supports Ward, single, average, and complete linkage strategies.

Agglomerative cluster has a "rich get richer" behavior that leads to uneven cluster sizes. In this regard, single linkage is the worst strategy, and Ward gives the most regular sizes. However, the affinity (or distance used in clustering) cannot be varied with Ward, thus for non Euclidean metrics, average linkage is a good alternative. Single linkage, while not robust to noisy data, can be computed very efficiently and can therefore be useful to provide hierarchical clustering of larger datasets. Single linkage can also perform well on non-globular data.

Examples:
- Various Agglomerative Clustering on a 2D embedding of digits: exploration of the different linkage strategies in a real dataset.

## 2.3.6.2. Visualization of cluster hierarchy

It's possible to visualize the tree representing the hierarchical merging of clusters as a dendrogram. Visual inspection can often be useful for understanding the structure of the data, though more so in the case of small sample sizes.

Hierarchical Clustering Dendrogram

(7) (8) 41 (5)(10)(7) (4) (8) (9)(15)(5) (7) (4)(22)(15)(23)
Number of points in node (or index of point if no parenthesis).
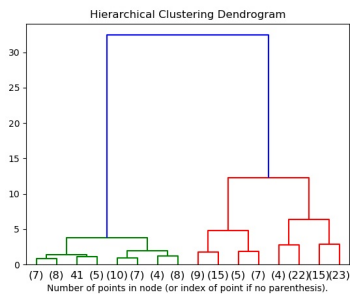
2.3.6.3. Adding connectivity constraints

An interesting aspect of `AgglomerativeClustering` is that connectivity constraints can be added to this algorithm (only adjacent clusters can be merged together), through a connectivity matrix that defines for each sample the neighboring samples following a given structure of the data. For instance, in the swiss-roll example below, the connectivity constraints forbid the merging of points that are not adjacent on the swiss roll, and thus avoid forming clusters that extend across overlapping folds of the roll.
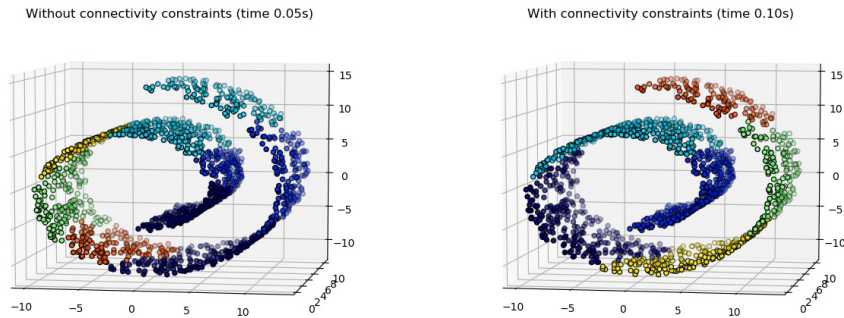
These constraint are useful to impose a certain local structure, but they also make the algorithm faster, especially when the number of the samples is high.

The connectivity constraints are imposed via an connectivity matrix: a scipy sparse matrix that has elements only at the intersection of a row and a column with indices of the dataset that should be connected. This matrix can be constructed from a-priori information: for instance, you may wish to cluster web pages by only merging pages with a link pointing from one to another. It can also be learned from the data, for instance using `sklearn.neighbors.kneighbors_graph` to restrict merging to nearest neighbors as in [this example](#), or using `sklearn.feature_extraction.image.grid_to_graph` to enable only merging of neighboring pixels on an image, as in the [coin](#) example.

Examples:
- [A demo of structured Ward hierarchical clustering on an image of coins](#): Ward clustering to split the image of coins in regions.
- [Hierarchical clustering: structured vs unstructured ward](#): Example of Ward algorithm on a swiss-roll, comparison of structured approaches versus unstructured approaches.
- [Feature agglomeration vs. univariate selection](#): Example of dimensionality reduction with feature agglomeration based on Ward hierarchical clustering.
- [Agglomerative clustering with and without structure](#)

Warning: Connectivity constraints with single, average and complete linkage
Connectivity constraints and single, complete or average linkage can enhance the 'rich getting richer' aspect of agglomerative clustering, particularly so if they are built with `sklearn.neighbors.kneighbors_graph`. In the limit of a small number of clusters, they tend to give a few macroscopically occupied clusters and almost empty ones. (see the discussion in [Agglomerative clustering with and without structure](#)). Single linkage is the most brittle linkage option with regard to this issue.
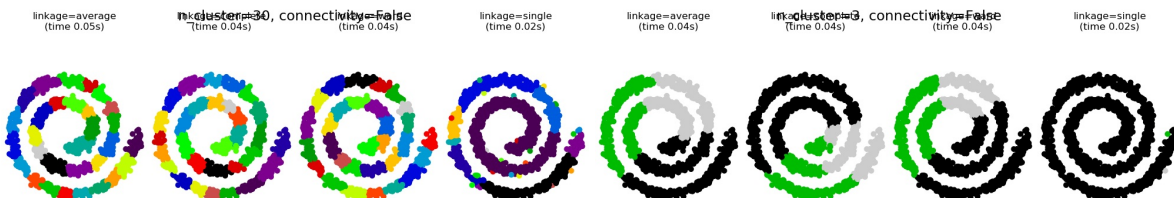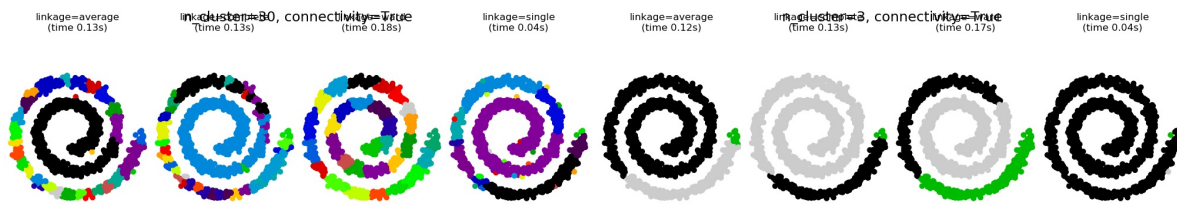
2.3.6.4. Varying the metric

Single, average and complete linkage can be used with a variety of distances (or affinities), in particular Euclidean distance (*l2*), Manhattan distance (or Cityblock, or *l1*), cosine distance, or any precomputed affinity matrix.
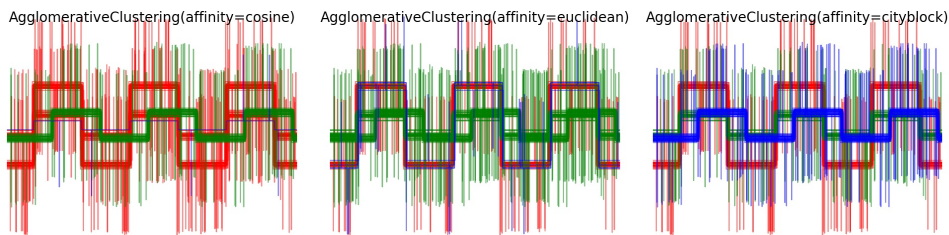
- *l1* distance is often good for sparse features, or sparse noise: i.e. many of the features are zero, as in text mining using occurrences of rare words.
- *cosine* distance is interesting because it is invariant to global scalings of the signal.

The guidelines for choosing a metric is to use one that maximizes the distance between samples in different classes, and minimizes that within each class.

AgglomerativeClustering(affinity=cosine)   AgglomerativeClustering(affinity=euclidean)   AgglomerativeClustering(affinity=cityblock)

Examples:

- [Agglomerative clustering with different metrics](#)

2.3.7. 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.

More formally, we define a core sample as being a sample in the dataset such that there exist `min_samples` other samples within a distance of `eps`, which are defined as *neighbors* of the core sample. This tells us that the core sample is in a dense area of the vector space. A cluster is a set of core samples that can be built by recursively taking a core sample, finding all of its neighbors that are core samples, finding all of *their* neighbors that are core samples, and so on. A cluster also has a set of non-core samples, which are samples that are neighbors of a core sample in the cluster but are not themselves core samples. Intuitively, these samples are on the fringes of a cluster.

Any core sample is part of a cluster, by definition. Any sample that is not a core sample, and is at least `eps` in distance from any core sample, is considered an outlier by the algorithm.

While the parameter `min_samples` primarily controls how tolerant the algorithm is towards noise (on noisy and large data sets it may be desirable to increase this parameter), the parameter `eps` is *crucial to choose appropriately* for the data set and distance function and usually cannot be left at the default value. It controls the local neighborhood of the points. When chosen too small, most data will not be clustered at all (and labeled as `-1` for "noise"). When chosen too large, it causes close clusters to be merged into one cluster, and eventually the entire data set to be returned as a single cluster. Some heuristics for choosing this parameter have been discussed in the literature, for example based on a knee in the nearest neighbor distances plot (as discussed in the references below).

In the figure below, the color indicates cluster membership, with large circles indicating core samples found by the algorithm. Smaller circles are non-core samples that are still part of a cluster. Moreover, the outliers are indicated by black points below.

Estimated number of clusters: 3
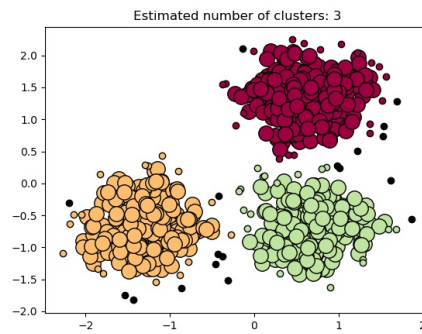
Implementation

The DBSCAN algorithm is deterministic, always generating the same clusters when given the same data in the same order. However, the results can differ when data is provided in a different order. First, even though the core samples will always be assigned to the same clusters, the labels of those clusters will depend on the order in which those samples are encountered in the data. Second and more importantly, the clusters to which non-core samples are assigned can differ depending on the data order. This would happen when a non-core sample has a distance lower than `eps` to two core samples in different clusters. By the triangular inequality, those two core samples must be more distant than `eps` from each other, or they would be in the same cluster. The non-core sample is assigned to whichever cluster is generated first in a pass through the data, and so the results will depend on the data ordering.

The current implementation uses ball trees and kd-trees to determine the neighborhood of points, which avoids calculating the full distance matrix (as was done in scikit-learn versions before 0.14). The possibility to use custom metrics is retained; for details, see `NearestNeighbors`.

Memory consumption for large sample sizes

This implementation is by default not memory efficient because it constructs a full pairwise similarity matrix in the case where kd-trees or ball-trees cannot be used (e.g., with sparse matrices). This matrix will consume n^2 floats. A couple of mechanisms for getting around this are:

- Use [OPTICS](#) clustering in conjunction with the `extract_dbscan` method. OPTICS clustering also calculates the full pairwise matrix, but only keeps one row in memory at a time (memory complexity n).
- A sparse radius neighborhood graph (where missing entries are presumed to be out of eps) can be precomputed in a memory-efficient way and dbscan can be run over this with `metric='precomputed'`. See [**sklearn.neighbors.NearestNeighbors.radius_neighbors_graph**](#).
- The dataset can be compressed, either by removing exact duplicates if these occur in your data, or by using BIRCH. Then you only have a relatively small number of representatives for a large number of points. You can then provide a `sample_weight` when fitting DBSCAN.

References:

- "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise" Ester, M., H. P. Kriegel, J. Sander, and X. Xu, In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, pp. 226–231. 1996
- "DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). In ACM Transactions on Database Systems (TODS), 42(3), 19.

2.3.8. OPTICS

The **OPTICS** algorithm shares many similarities with the **DBSCAN** algorithm, and can be considered a generalization of DBSCAN that relaxes the `eps` requirement from a single value to a value range. The key difference between DBSCAN and OPTICS is that the OPTICS algorithm builds a *reachability* graph, which assigns each sample both a `reachability_` distance, and a spot within the cluster `ordering_` attribute; these two attributes are assigned when the model is fitted, and are used to determine cluster membership. If OPTICS is run with the default value of *inf* set for `max_eps`, then DBSCAN style cluster extraction can be performed repeatedly in linear time for any given `eps` value using the `cluster_optics_dbscan` method. Setting `max_eps` to a lower value will result in shorter run times, and can be thought of as the maximum neighborhood radius from each point to find other potential reachable points.
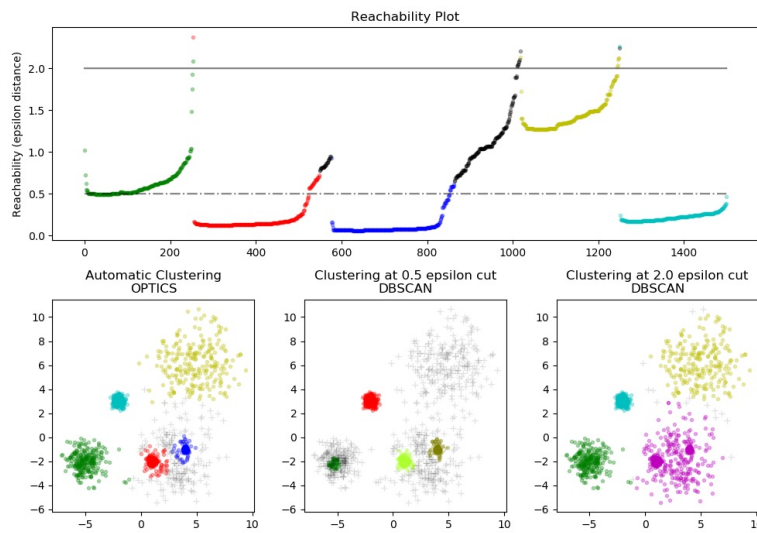
The *reachability* distances generated by OPTICS allow for variable density extraction of clusters within a single data set. As shown in the above plot, combining *reachability* distances and data set `ordering_` produces a *reachability plot*, where point density is represented on the Y-axis, and points are ordered such that nearby points are adjacent. 'Cutting' the reachability plot at a single value produces DBSCAN like results; all points above the 'cut' are classified as noise, and each time that there is a break when reading from left to right signifies a new cluster. The default cluster extraction with OPTICS looks at the steep slopes within the graph to find clusters, and the user can define what counts as a steep slope using the parameter `xi`. There are also other possibilities for analysis on the graph itself, such as generating hierarchical representations of the data through reachability-plot dendrograms, and the hierarchy of clusters detected by the algorithm can be accessed through the `cluster_hierarchy_` parameter. The plot above has been color-coded so that cluster colors in planar space match the linear segment clusters of the reachability plot. Note that the blue and red clusters are adjacent in the reachability plot, and can be hierarchically represented as children of a larger parent cluster.

Examples:

- Demo of OPTICS clustering algorithm

Comparison with DBSCAN

The results from OPTICS `cluster_optics_dbscan` method and DBSCAN are very similar, but not always identical; specifically, labeling of periphery and noise points. This is in part because the first samples of each dense area processed by OPTICS have a large reachability value while being close to other points in their area, and will thus sometimes be marked as noise rather than periphery. This affects adjacent points when they are considered as candidates for being marked as either periphery or noise.

Note that for any single value of `eps`, DBSCAN will tend to have a shorter run time than OPTICS; however, for repeated runs at varying `eps` values, a single run of OPTICS may require less cumulative runtime than DBSCAN. It is also important to note that OPTICS' output is close to DBSCAN's only if `eps` and `max_eps` are close.

Computational Complexity

Spatial indexing trees are used to avoid calculating the full distance matrix, and allow for efficient memory usage on large sets of samples. Different distance metrics can be supplied via the `metric` keyword.

For large datasets, similar (but not identical) results can be obtained via HDBSCAN. The HDBSCAN implementation is multithreaded, and has better algorithmic runtime complexity than OPTICS, at the cost of worse memory scaling. For extremely large datasets that exhaust system memory using HDBSCAN, OPTICS will maintain *n* (as opposed to *n^2*) memory scaling; however, tuning of the `max_eps` parameter will likely need to be used to give a solution in a reasonable amount of wall time.

References:

- "OPTICS: ordering points to identify the clustering structure." Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. In ACM Sigmod Record, vol. 28, no. 2, pp. 49-60. ACM, 1999.

2.3.9. Birch

The `Birch` builds a tree called the Clustering Feature Tree (CFT) for the given data. The data is essentially lossy compressed to a set of Clustering Feature nodes (CF Nodes). The CF Nodes have a number of subclusters called Clustering Feature subclusters (CF Subclusters) and these CF Subclusters located in the non-terminal CF Nodes can have CF Nodes as children.

The CF Subclusters hold the necessary information for clustering which prevents the need to hold the entire input data in memory. This information includes:

- Number of samples in a subcluster.
- Linear Sum - A n-dimensional vector holding the sum of all samples
- Squared Sum - Sum of the squared L2 norm of all samples.
- Centroids - To avoid recalculation linear sum / n_samples.
- Squared norm of the centroids.

The Birch algorithm has two parameters, the threshold and the branching factor. The branching factor limits the number of subclusters in a node and the threshold limits the distance between the entering sample and the existing subclusters.

This algorithm can be viewed as an instance or data reduction method, since it reduces the input data to a set of subclusters which are obtained directly from the leaves of the CFT. This reduced data can be further processed by feeding it into a global clusterer. This global clusterer can be set by `n_clusters`. If `n_clusters` is set to None, the subclusters from the leaves are directly read off, otherwise a global clustering step labels these subclusters into global clusters (labels) and the samples are mapped to the global label of the nearest subcluster.

Algorithm description:

- A new sample is inserted into the root of the CF Tree which is a CF Node. It is then merged with the subcluster of the root, that has the smallest radius after merging, constrained by the threshold and branching factor conditions. If the subcluster has any child node, then this is done repeatedly till it reaches a leaf. After finding the nearest subcluster in the leaf, the properties of this subcluster and the parent subclusters are recursively updated.
- If the radius of the subcluster obtained by merging the new sample and the nearest subcluster is greater than the square of the threshold and if the number of subclusters is greater than the branching factor, then a space is temporarily allocated to this new sample. The two farthest subclusters are taken and the subclusters are divided into two groups on the basis of the distance between these subclusters.
- If this split node has a parent subcluster and there is room for a new subcluster, then the parent is split into two. If there is no room, then this node is again split into two and the process is continued recursively, till it reaches the root.

Birch or MiniBatchKMeans?

- Birch does not scale very well to high dimensional data. As a rule of thumb if `n_features` is greater than twenty, it is generally better to use MiniBatchKMeans.
- If the number of instances of data needs to be reduced, or if one wants a large number of subclusters either as a preprocessing step or otherwise, Birch is more useful than MiniBatchKMeans.

How to use partial_fit?

To avoid the computation of global clustering, for every call of `partial_fit` the user is advised

1. To set `n_clusters=None` initially
2. Train all data by multiple calls to partial_fit.
3. Set `n_clusters` to a required value using `brc.set_params(n_clusters=n_clusters)`.
4. Call `partial_fit` finally with no arguments, i.e. `brc.partial_fit()` which performs the global clustering.

| Birch without global clustering | Birch with global clustering | MiniBatchKMeans |
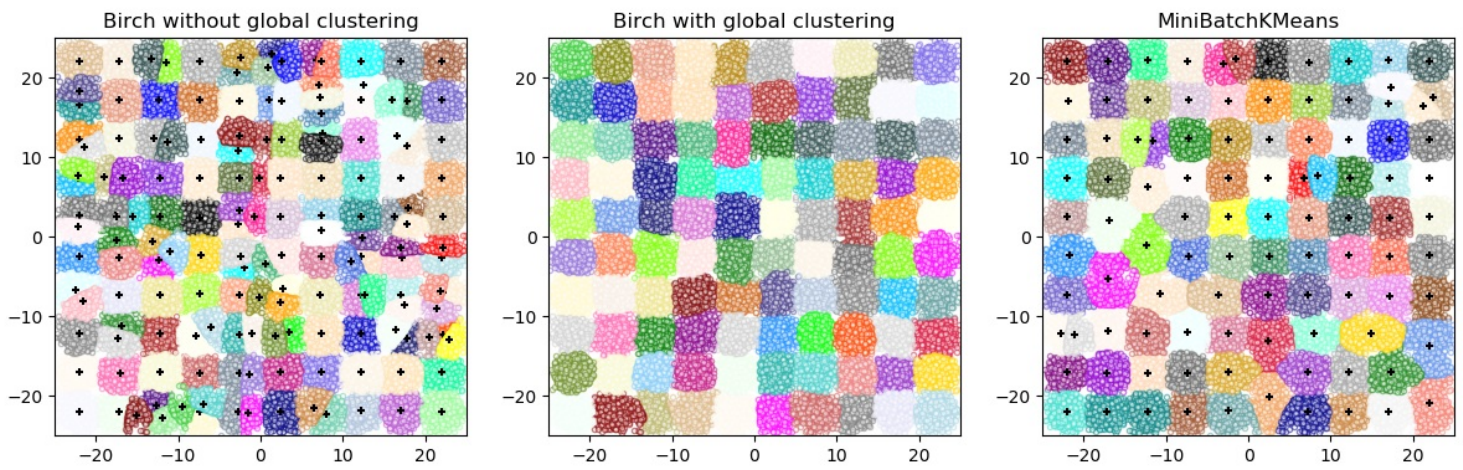
References:

- Tian Zhang, Raghu Ramakrishnan, Maron Livny BIRCH: An efficient data clustering method for large databases.
  https://www.cs.sfu.ca/CourseCentral/459/han/papers/zhang96.pdf
- Roberto Perdisci JBirch - Java implementation of BIRCH clustering algorithm https://code.google.com/archive/p/jbirch

2.3.10. Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm. In particular any evaluation metric should not take the absolute values of the cluster labels into account but rather if this clustering define separations of the data similar to some ground truth set of classes or satisfying some assumption such that members belong to the same class are more similar than members of different classes according to some similarity metric.

2.3.10.1. Adjusted Rand index

Given the knowledge of the ground truth class assignments `labels_true` and our clustering algorithm assignments of the same samples `labels_pred`, the **adjusted Rand index** is a function that measures the **similarity** of the two assignments, ignoring permutations and **with chance normalization**:

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3, and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
0.24...
```

Furthermore, **adjusted_rand_score** is **symmetric**: swapping the argument does not change the score. It can thus be used as a **consensus measure**:

```
>>> metrics.adjusted_rand_score(labels_pred, labels_true)
0.24...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
1.0
```

Bad (e.g. independent labelings) have negative or close to 0.0 scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_rand_score(labels_true, labels_pred)
-0.12...
```

2.3.10.1.1. Advantages

- **Random (uniform) label assignments have a ARI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Rand index or the V-measure for instance).
- **Bounded range [-1, 1]**: negative values are bad (independent labelings), similar clusterings have a positive ARI, 1.0 is the perfect match score.
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with "folded" shapes.

2.3.10.1.2. Drawbacks

- Contrary to inertia, **ARI requires knowledge of the ground truth classes** while is almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

    However ARI can also be useful in a purely unsupervised setting as a building block for a Consensus Index that can be used for clustering model selection (TODO).

> **Examples:**
> - [Adjustment for chance in clustering performance evaluation](#): Analysis of the impact of the dataset size on the value of clustering measures for random assignments.

2.3.10.1.3. Mathematical formulation

If C is a ground truth class assignment and K the clustering, let us define a and b as:

- a, the number of pairs of elements that are in the same set in C and in the same set in K
- b, the number of pairs of elements that are in different sets in C and in different sets in K

The raw (unadjusted) Rand index is then given by:

$$\mathrm{RI} = \frac{a + b}{C_2^{n_{samples}}}$$

Where $C_2^{n_{samples}}$ is the total number of possible pairs in the dataset (without ordering).

However the RI score does not guarantee that random label assignments will get a value close to zero (esp. if the number of clusters is in the same order of magnitude as the number of samples).

To counter this effect we can discount the expected RI $E[\mathrm{RI}]$ of random labelings by defining the adjusted Rand index as follows:

$$\mathrm{ARI} = \frac{\mathrm{RI} - E[\mathrm{RI}]}{\max(\mathrm{RI}) - E[\mathrm{RI}]}$$

> **References**
> - [Comparing Partitions](#) L. Hubert and P. Arabie, Journal of Classification 1985
> - [Wikipedia entry for the adjusted Rand index](#)

2.3.10.2. Mutual Information based scores

Given the knowledge of the ground truth class assignments `labels_true` and our clustering algorithm assignments of the same samples `labels_pred`, the **Mutual Information** is a function that measures the **agreement** of the two assignments, ignoring permutations. Two different normalized versions of this measure are available, **Normalized Mutual Information (NMI)** and **Adjusted Mutual Information (AMI)**. NMI is often used in the literature, while AMI was proposed more recently and is **normalized against chance**:

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3 and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
0.22504...
```

All, `mutual_info_score`, `adjusted_mutual_info_score` and `normalized_mutual_info_score` are symmetric: swapping the argument does not change the score. Thus they can be used as a **consensus measure**:

```
>>> metrics.adjusted_mutual_info_score(labels_pred, labels_true)
0.22504...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
1.0

>>> metrics.normalized_mutual_info_score(labels_true, labels_pred)
1.0
```

This is not true for `mutual_info_score`, which is therefore harder to judge:

```
>>> metrics.mutual_info_score(labels_true, labels_pred)
0.69...
```

Bad (e.g. independent labelings) have non-positive scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.adjusted_mutual_info_score(labels_true, labels_pred)
-0.10526...
```

2.3.10.2.1. Advantages

- **Random (uniform) label assignments have a AMI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Mutual Information or the V-measure for instance).
- **Upper bound of 1**: Values close to zero indicate two label assignments that are largely independent, while values close to one indicate significant agreement. Further, an AMI of exactly 1 indicates that the two label assignments are equal (with or without permutation).

2.3.10.2.2. Drawbacks

- Contrary to inertia, **MI-based measures require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

  However MI-based measures can also be useful in purely unsupervised setting as a building block for a Consensus Index that can be used for clustering model selection.

- NMI and MI are not adjusted against chance.

> **Examples:**
> - Adjustment for chance in clustering performance evaluation: Analysis of the impact of the dataset size on the value of clustering measures for random assignments. This example also includes the Adjusted Rand Index.

2.3.10.2.3. Mathematical formulation

Assume two label assignments (of the same N objects), U and V. Their entropy is the amount of uncertainty for a partition set, defined by:

$$H(U) = - \sum_{i=1}^{|U|} P(i) \log(P(i))$$

where $P(i) = |U_i|/N$ is the probability that an object picked at random from U falls into class U_i. Likewise for V:

$$H(V) = - \sum_{j=1}^{|V|} P'(j) \log(P'(j))$$

With $P'(j) = |V_j|/N$. The mutual information (MI) between U and V is calculated by:

$$\text{MI}(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P(i,j) \log\left(\frac{P(i,j)}{P(i)P'(j)} \right)$$

where $P(i, j) = |U_i \cap V_j|/N$ is the probability that an object picked at random falls into both classes U_i and V_j.

It also can be expressed in set cardinality formulation:

$$\text{MI}(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \frac{|U_i \cap V_j|}{N} \log\left(\frac{N|U_i \cap V_j|}{|U_i||V_j|} \right)$$

The normalized mutual information is defined as

$$\text{NMI}(U, V) = \frac{\text{MI}(U, V)}{\text{mean}(H(U), H(V))}$$

This value of the mutual information and also the normalized variant is not adjusted for chance and will tend to increase as the number of different labels (clusters) increases, regardless of the actual amount of "mutual information" between the label assignments.

The expected value for the mutual information can be calculated using the following equation [VEB2009]. In this equation, $a_i = |U_i|$ (the number of elements in U_i) and $b_j = |V_j|$ (the number of elements in V_j).

$$E[\text{MI}(U, V)] = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} \sum_{n_{ij}=(a_i+b_j-N)^+}^{\min(a_i, b_j)} \frac{n_{ij}}{N} \log\left(\frac{N \cdot n_{ij}}{a_i b_j} \right) \frac{a_i! b_j! (N - a_i)! (N - b_j)!}{N! n_{ij}! (a_i - n_{ij})! (b_j - n_{ij})! (N - a_i - b_j + n_{ij})!}$$

Using the expected value, the adjusted mutual information can then be calculated using a similar form to that of the adjusted Rand index:

$$\text{AMI} = \frac{\text{MI} - E[\text{MI}]}{\text{mean}(H(U), H(V)) - E[\text{MI}]}$$

For normalized mutual information and adjusted mutual information, the normalizing value is typically some *generalized* mean of the entropies of each clustering. Various generalized means exist, and no firm rules exist for preferring one over the others. The decision is largely a field-by-field basis; for instance, in community detection, the arithmetic mean is most common. Each normalizing method provides "qualitatively similar behaviours" [YAT2016]. In our implementation, this is controlled by the `average_method` parameter.

Vinh et al. (2010) named variants of NMI and AMI by their averaging method [VEB2010]. Their 'sqrt' and 'sum' averages are the geometric and arithmetic means; we use these more broadly common names.

References

- Strehl, Alexander, and Joydeep Ghosh (2002). "Cluster ensembles – a knowledge reuse framework for combining multiple partitions". Journal of Machine Learning Research 3: 583–617. doi:10.1162/153244303321897735.
- Wikipedia entry for the (normalized) Mutual Information
- Wikipedia entry for the Adjusted Mutual Information

[VEB2009] Vinh, Epps, and Bailey, (2009). "Information theoretic measures for clusterings comparison". Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09. doi:10.1145/1553374.1553511. ISBN 9781605585161.

[VEB2010] Vinh, Epps, and Bailey, (2010). "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance". JMLR <http://jmlr.csail.mit.edu/papers/volume11/vinh10a/vinh10a.pdf>

[YAT2016] Yang, Algesheimer, and Tessone, (2016). "A comparative analysis of community detection algorithms on artificial networks". Scientific Reports 6: 30750. doi:10.1038/srep30750.

2.3.10.3. Homogeneity, completeness and V-measure

Given the knowledge of the ground truth class assignments of the samples, it is possible to define some intuitive metric using conditional entropy analysis.

In particular Rosenberg and Hirschberg (2007) define the following two desirable objectives for any cluster assignment:

- **homogeneity**: each cluster contains only members of a single class.
- **completeness**: all members of a given class are assigned to the same cluster.

We can turn those concept as scores `homogeneity_score` and `completeness_score`. Both are bounded below by 0.0 and above by 1.0 (higher is better):

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]

>>> metrics.homogeneity_score(labels_true, labels_pred)
0.66...

>>> metrics.completeness_score(labels_true, labels_pred)
0.42...
```

Their harmonic mean called **V-measure** is computed by **v_measure_score**:

```
>>> metrics.v_measure_score(labels_true, labels_pred)
0.51...
```

This function's formula is as follows:

$$v = \frac{(1 + \beta) \times \text{homogeneity} \times \text{completeness}}{(\beta \times \text{homogeneity} + \text{completeness})}$$

`beta` defaults to a value of 1.0, but for using a value less than 1 for beta:

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=0.6)
0.54...
```

more weight will be attributed to homogeneity, and using a value greater than 1:

```
>>> metrics.v_measure_score(labels_true, labels_pred, beta=1.8)
0.48...
```

more weight will be attributed to completeness.

The V-measure is actually equivalent to the mutual information (NMI) discussed above, with the aggregation function being the arithmetic mean [B2011].

Homogeneity, completeness and V-measure can be computed at once using **homogeneity_completeness_v_measure** as follows:

```
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
(0.66..., 0.42..., 0.51...)
```

The following clustering assignment is slightly better, since it is homogeneous but not complete:

```
>>> labels_pred = [0, 0, 0, 1, 2, 2]
>>> metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)
(1.0, 0.68..., 0.81...)
```

> **Note:** **v_measure_score** is **symmetric**: it can be used to evaluate the **agreement** of two independent assignments on the same dataset.
> This is not the case for **completeness_score** and **homogeneity_score**: both are bound by the relationship:
> ```
> homogeneity_score(a, b) == completeness_score(b, a)
> ```

2.3.10.3.1. Advantages

- **Bounded scores**: 0.0 is as bad as it can be, 1.0 is a perfect score.
- Intuitive interpretation: clustering with bad V-measure can be **qualitatively analyzed in terms of homogeneity and completeness** to better feel what 'kind' of mistakes is done by the assignment.
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with "folded" shapes.

2.3.10.3.2. Drawbacks

- The previously introduced metrics are **not normalized with regards to random labeling**: this means that depending on the number of samples, clusters and ground truth classes, a completely random labeling will not always yield the same values for homogeneity, completeness and hence v-measure. In particular **random labeling won't yield zero scores especially when the number of clusters is large**.

This problem can safely be ignored when the number of samples is more than a thousand and the number of clusters is less than 10. **For smaller sample sizes or larger number of clusters it is safer to use an adjusted index such as the Adjusted Rand Index (ARI)**.

Clustering measures for 2 random uniform labelings with equal number of clusters
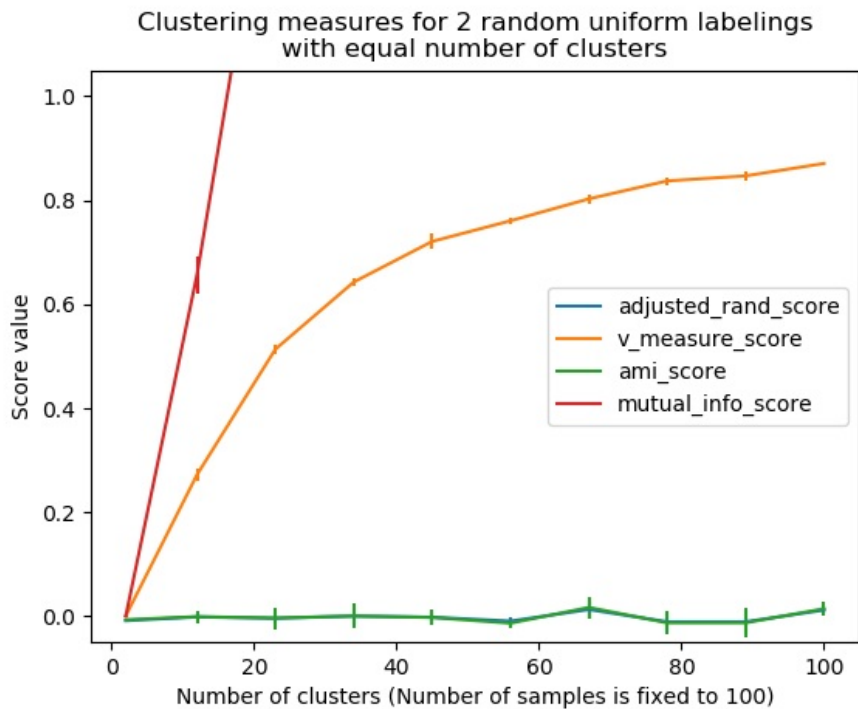
- These metrics **require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

> **Examples:**
> - [Adjustment for chance in clustering performance evaluation](#): Analysis of the impact of the dataset size on the value of clustering measures for random assignments.

2.3.10.3.3. Mathematical formulation

Homogeneity and completeness scores are formally given by:

$$h = 1 - \frac{H(C|K)}{H(C)}$$

$$c = 1 - \frac{H(K|C)}{H(K)}$$

where $H(C|K)$ is the **conditional entropy of the classes given the cluster assignments** and is given by:

$$H(C|K) = -\sum_{c=1}^{|C|} \sum_{k=1}^{|K|} \frac{n_{c,k}}{n} \cdot \log\left(\frac{n_{c,k}}{n_k}\right)$$

and $H(C)$ is the **entropy of the classes** and is given by:

$$H(C) = -\sum_{c=1}^{|C|} \frac{n_c}{n} \cdot \log\left(\frac{n_c}{n}\right)$$

with n the total number of samples, n_c and n_k the number of samples respectively belonging to class c and cluster k, and finally $n_{c,k}$ the number of samples from class c assigned to cluster k.

The **conditional entropy of clusters given class** $H(K|C)$ and the **entropy of clusters** $H(K)$ are defined in a symmetric manner.

Rosenberg and Hirschberg further define **V-measure** as the **harmonic mean of homogeneity and completeness**:

$$v = 2 \cdot \frac{h \cdot c}{h + c}$$

References

- [V-Measure: A conditional entropy-based external cluster evaluation measure](#) Andrew Rosenberg and Julia Hirschberg, 2007
- **[B2011]** [Identication and Characterization of Events in Social Media](#), Hila Becker, PhD Thesis.

2.3.10.4. Fowlkes-Mallows scores

The Fowlkes-Mallows index (`sklearn.metrics.fowlkes_mallows_score`) can be used when the ground truth class assignments of the samples is known. The Fowlkes-Mallows score FMI is defined as the geometric mean of the pairwise precision and recall:

$$\mathrm{FMI} = \frac{\mathrm{TP}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})}}$$

Where `TP` is the number of **True Positive** (i.e. the number of pair of points that belong to the same clusters in both the true labels and the predicted labels), `FP` is the number of **False Positive** (i.e. the number of pair of points that belong to the same clusters in the true labels and not in the predicted labels) and `FN` is the number of **False Negative** (i.e the number of pair of points that belongs in the same clusters in the predicted labels and not in the true labels).

The score ranges from 0 to 1. A high value indicates a good similarity between two clusters.

```
>>> from sklearn import metrics
>>> labels_true = [0, 0, 0, 1, 1, 1]
>>> labels_pred = [0, 0, 1, 1, 2, 2]
```

```
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

One can permute 0 and 1 in the predicted labels, rename 2 to 3 and get the same score:

```
>>> labels_pred = [1, 1, 0, 0, 3, 3]
```

```
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.47140...
```

Perfect labeling is scored 1.0:

```
>>> labels_pred = labels_true[:]
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
1.0
```

Bad (e.g. independent labelings) have zero scores:

```
>>> labels_true = [0, 1, 2, 0, 3, 4, 5, 1]
>>> labels_pred = [1, 1, 0, 0, 2, 2, 2, 2]
>>> metrics.fowlkes_mallows_score(labels_true, labels_pred)
0.0
```

2.3.10.4.1. Advantages

- **Random (uniform) label assignments have a FMI score close to 0.0** for any value of `n_clusters` and `n_samples` (which is not the case for raw Mutual Information or the V-measure for instance).
- **Upper-bounded at 1**: Values close to zero indicate two label assignments that are largely independent, while values close to one indicate significant agreement. Further, values of exactly 0 indicate **purely** independent label assignments and a FMI of exactly 1 indicates that the two label assignments are equal (with or without permutation).
- **No assumption is made on the cluster structure**: can be used to compare clustering algorithms such as k-means which assumes isotropic blob shapes with results of spectral clustering algorithms which can find cluster with "folded" shapes.

2.3.10.4.2. Drawbacks

- Contrary to inertia, **FMI-based measures require the knowledge of the ground truth classes** while almost never available in practice or requires manual assignment by human annotators (as in the supervised learning setting).

References

- E. B. Fowlkes and C. L. Mallows, 1983. "A method for comparing two hierarchical clusterings". Journal of the American Statistical Association. http://wildfire.stat.ucla.edu/pdflibrary/fowlkes.pdf
- [Wikipedia entry for the Fowlkes-Mallows Index](#)

2.3.10.5. Silhouette Coefficient

If the ground truth labels are not known, evaluation must be performed using the model itself. The Silhouette Coefficient (`sklearn.metrics.silhouette_score`) is an example of such an evaluation, where a higher Silhouette Coefficient score relates to a model with better defined clusters. The Silhouette Coefficient is defined for each sample and is composed of two scores:

- **a**: The mean distance between a sample and all other points in the same class.
- **b**: The mean distance between a sample and all other points in the *next nearest cluster*.

The Silhouette Coefficient *s* for a single sample is then given as:

$$s = \frac{b - a}{max(a, b)}$$

The Silhouette Coefficient for a set of samples is given as the mean of the Silhouette Coefficient for each sample.

```
>>> from sklearn import metrics
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn import datasets
>>> X, y = datasets.load_iris(return_X_y=True)
```

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

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans_model.labels_
>>> metrics.silhouette_score(X, labels, metric='euclidean')
0.55...
```

> **References**
>
> - Peter J. Rousseeuw (1987). "Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis". Computational and Applied Mathematics 20: 53–65. doi:10.1016/0377-0427(87)90125-7.

2.3.10.5.1. Advantages

- The score is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero indicate overlapping clusters.
- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.

2.3.10.5.2. Drawbacks

- The Silhouette Coefficient is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

> **Examples:**
>
> - Selecting the number of clusters with silhouette analysis on KMeans clustering : In this example the silhouette analysis is used to choose an optimal value for n_clusters.

2.3.10.6. Calinski-Harabasz Index

If the ground truth labels are not known, the Calinski-Harabasz index (`sklearn.metrics.calinski_harabasz_score`) - also known as the Variance Ratio Criterion - can be used to evaluate the model, where a higher Calinski-Harabasz score relates to a model with better defined clusters.

The index is the ratio of the sum of between-clusters dispersion and of inter-cluster dispersion for all clusters (where dispersion is defined as the sum of distances squared):

```
>>> from sklearn import metrics
>>> from sklearn.metrics import pairwise_distances
>>> from sklearn import datasets
>>> X, y = datasets.load_iris(return_X_y=True)
```

In normal usage, the Calinski-Harabasz index is applied to the results of a cluster analysis:

```
>>> import numpy as np
>>> from sklearn.cluster import KMeans
>>> kmeans_model = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans_model.labels_
>>> metrics.calinski_harabasz_score(X, labels)
561.62...
```

2.3.10.6.1. Advantages

- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.
- The score is fast to compute.

2.3.10.6.2. Drawbacks

- The Calinski-Harabasz index is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

2.3.10.6.3. Mathematical formulation

For a set of data E of size n_E which has been clustered into k clusters, the Calinski-Harabasz score s is defined as the ratio of the between-clusters dispersion mean and the within-cluster dispersion:

$$s = \frac{\mathrm{tr}(B_k)}{\mathrm{tr}(W_k)} \times \frac{n_E - k}{k - 1}$$

where $\mathrm{tr}(B_k)$ is trace of the between group dispersion matrix and $\mathrm{tr}(W_k)$ is the trace of the within-cluster dispersion matrix defined by:

$$W_k = \sum_{q=1}^{k} \sum_{x \in C_q} (x - c_q)(x - c_q)^T$$

$$B_k = \sum_{q=1}^{k} n_q (c_q - c_E)(c_q - c_E)^T$$

with C_q the set of points in cluster q, c_q the center of cluster q, c_E the center of E, and n_q the number of points in cluster q.

> **References**
> - Caliński, T., & Harabasz, J. (1974). "A Dendrite Method for Cluster Analysis". Communications in Statistics-theory and Methods 3: 1-27. doi:10.1080/03610927408827101.

2.3.10.7. Davies-Bouldin Index

If the ground truth labels are not known, the Davies-Bouldin index (`sklearn.metrics.davies_bouldin_score`) can be used to evaluate the model, where a lower Davies-Bouldin index relates to a model with better separation between the clusters.

This index signifies the average 'similarity' between clusters, where the similarity is a measure that compares the distance between clusters with the size of the clusters themselves.

Zero is the lowest possible score. Values closer to zero indicate a better partition.

In normal usage, the Davies-Bouldin index is applied to the results of a cluster analysis as follows:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> from sklearn.cluster import KMeans
>>> from sklearn.metrics import davies_bouldin_score
>>> kmeans = KMeans(n_clusters=3, random_state=1).fit(X)
>>> labels = kmeans.labels_
>>> davies_bouldin_score(X, labels)
0.6619...
```

2.3.10.7.1. Advantages

- The computation of Davies-Bouldin is simpler than that of Silhouette scores.
- The index is computed only quantities and features inherent to the dataset.

2.3.10.7.2. Drawbacks

- The Davies-Boulding index is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained from DBSCAN.
- The usage of centroid distance limits the distance metric to Euclidean space.

2.3.10.7.3. Mathematical formulation

The index is defined as the average similarity between each cluster C_i for $i = 1, \ldots, k$ and its most similar one C_j. In the context of this index, similarity is defined as a measure R_{ij} that trades off:

- s_i, the average distance between each point of cluster i and the centroid of that cluster – also know as cluster diameter.
- d_{ij}, the distance between cluster centroids i and j.

A simple choice to construct R_ij so that it is nonnegative and symmetric is:

$$R_{ij} = \frac{s_i + s_j}{d_{ij}}$$

Then the Davies-Bouldin index is defined as:

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{i \neq j} R_{ij}$$

References

- Davies, David L.; Bouldin, Donald W. (1979). "A Cluster Separation Measure" IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-1 (2): 224-227. doi:10.1109/TPAMI.1979.4766909.
- Halkidi, Maria; Batistakis, Yannis; Vazirgiannis, Michalis (2001). "On Clustering Validation Techniques" Journal of Intelligent Information Systems, 17(2-3), 107-145. doi:10.1023/A:1012801612483.
- Wikipedia entry for Davies-Bouldin index.

2.3.10.8. Contingency Matrix

Contingency matrix (`sklearn.metrics.cluster.contingency_matrix`) reports the intersection cardinality for every true/predicted cluster pair. The contingency matrix provides sufficient statistics for all clustering metrics where the samples are independent and identically distributed and one doesn't need to account for some instances not being clustered.

Here is an example:

```
>>> from sklearn.metrics.cluster import contingency_matrix
>>> x = ["a", "a", "a", "b", "b", "b"]
>>> y = [0, 0, 1, 1, 2, 2]
>>> contingency_matrix(x, y)
array([[2, 1, 0],
       [0, 1, 2]])
```

The first row of output array indicates that there are three samples whose true cluster is "a". Of them, two are in predicted cluster 0, one is in 1, and none is in 2. And the second row indicates that there are three samples whose true cluster is "b". Of them, none is in predicted cluster 0, one is in 1 and two are in 2.

A confusion matrix for classification is a square contingency matrix where the order of rows and columns correspond to a list of classes.

2.3.10.8.1. Advantages

- Allows to examine the spread of each true cluster across predicted clusters and vice versa.
- The contingency table calculated is typically utilized in the calculation of a similarity statistic (like the others listed in this document) between the two clusterings.

2.3.10.8.2. Drawbacks

- Contingency matrix is easy to interpret for a small number of clusters, but becomes very hard to interpret for a large number of clusters.
- It doesn't give a single metric to use as an objective for clustering optimisation.

References

- Wikipedia entry for contingency matrix

Toggle Menu