Comparing different clustering algorithms on toy datasets

https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html (https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html)

- A general picture on the performance of various clustering algorithms is given below.
- While these 2D examples give some intuition about the algorithms, this intuition might not apply to very high dimensional data.
- With the exception of the last dataset, the parameters of each of these dataset-algorithm pairs has been tuned to produce good clustering results. Some algorithms are more sensitive to parameter values than others.
- The last dataset is an example of a homogeneous data, and there is no good clustering. For this example, the null dataset uses the same parameters as the dataset in the row above it, which represents a mismatch in the parameter values and the data structure.

In [16]:

```
import time
import warnings
import numpy as np
import matplotlib.pyplot as plt
from sklearn import cluster, datasets, mixture
from sklearn.neighbors import kneighbors_graph
from sklearn.preprocessing import StandardScaler
from itertools import cycle, islice

np.random.seed(0)

# ============
# Generate datasets. We choose the size big enough to see the scalability
# of the algorithms, but not too big to avoid too long running times
# ============
n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5,
                                      noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None

# Anisotropicly distributed data
random_state = 170
X, y = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

# blobs with varied variances
varied = datasets.make_blobs(n_samples=n_samples,
                             cluster_std=[1.0, 2.5, 0.5],
                             random_state=random_state)

# ============
# Set up cluster parameters
# ============
plt.figure(figsize=(9 * 2 + 3, 12.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1
```

```
default_base = {'quantile': .3,
                'eps': .3,
                'damping': .9,
                'preference': -200,
                'n_neighbors': 10,
                'n_clusters': 3}

datasets = [
    (noisy_circles, {'damping': .77, 'preference': -240,
                     'quantile': .2, 'n_clusters': 2}),
    (noisy_moons, {'damping': .75, 'preference': -220, 'n_clusters': 2}),
    (varied, {'eps': .18, 'n_neighbors': 2}),
    (aniso, {'eps': .15, 'n_neighbors': 2}),
    (blobs, {}),
    (no_structure, {})]

for i_dataset, (dataset, algo_params) in enumerate(datasets):
    # update parameters with dataset-specific values
    params = default_base.copy()
    params.update(algo_params)

    X, y = dataset

    # normalize dataset for easier parameter selection
    X = StandardScaler().fit_transform(X)

    # estimate bandwidth for mean shift
    bandwidth = cluster.estimate_bandwidth(X, quantile=params['quantile'])

    # connectivity matrix for structured Ward
    connectivity = kneighbors_graph(
        X, n_neighbors=params['n_neighbors'], include_self=False)
    # make connectivity symmetric
    connectivity = 0.5 * (connectivity + connectivity.T)

    # ============
    # Create cluster objects
    # ============
    ms = cluster.MeanShift(bandwidth=bandwidth, bin_seeding=True)
    two_means = cluster.MiniBatchKMeans(n_clusters=params['n_clusters'])
    ward = cluster.AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='ward',
```

```
    connectivity=connectivity)
spectral = cluster.SpectralClustering(
    n_clusters=params['n_clusters'], eigen_solver='arpack',
    affinity="nearest_neighbors")
dbscan = cluster.DBSCAN(eps=params['eps'])
affinity_propagation = cluster.AffinityPropagation(
    damping=params['damping'], preference=params['preference'])
average_linkage = cluster.AgglomerativeClustering(
    linkage="average", affinity="cityblock",
    n_clusters=params['n_clusters'], connectivity=connectivity)
birch = cluster.Birch(n_clusters=params['n_clusters'])
gmm = mixture.GaussianMixture(
    n_components=params['n_clusters'], covariance_type='full')

clustering_algorithms = (
    ('MiniBatchKMeans', two_means),
    ('AffinityPropagation', affinity_propagation),
    ('MeanShift', ms),
    ('SpectralClustering', spectral),
    ('Ward', ward),
    ('AgglomerativeClustering', average_linkage),
    ('DBSCAN', dbscan),
    ('Birch', birch),
    ('GaussianMixture', gmm)
)

for name, algorithm in clustering_algorithms:
    t0 = time.time()

    # catch warnings related to kneighbors_graph
    with warnings.catch_warnings():
        warnings.filterwarnings(
            "ignore",
            message="the number of connected components of the " +
            "connectivity matrix is [0-9]{1,2}" +
            " > 1. Completing it to avoid stopping the tree early.",
            category=UserWarning)
        warnings.filterwarnings(
            "ignore",
            message="Graph is not fully connected, spectral embedding" +
            " may not work as expected.",
            category=UserWarning)
        algorithm.fit(X)
```

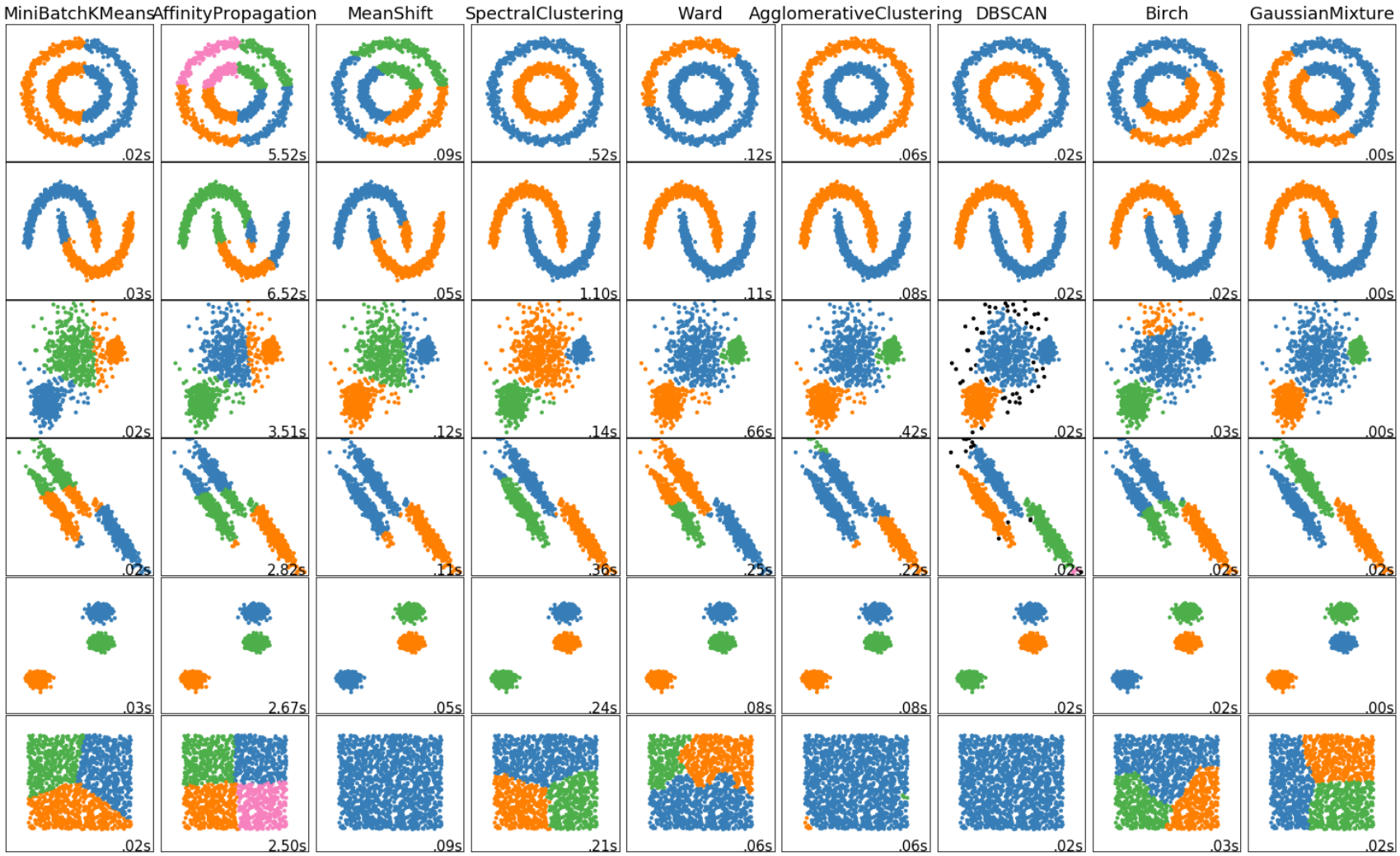
```
        t1 = time.time()
        if hasattr(algorithm, 'labels_'):
            y_pred = algorithm.labels_.astype(np.int)
        else:
            y_pred = algorithm.predict(X)

        plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        colors = np.array(list(islice(cycle(['#377eb8', '#ff7f00', '#4daf4a',
                                             '#f781bf', '#a65628', '#984ea3',
                                             '#999999', '#e41a1c', '#dede00']),
                                      int(max(y_pred) + 1))))
        # add black color for outliers (if any)
        colors = np.append(colors, ["#000000"])
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[y_pred])

        plt.xlim(-2.5, 2.5)
        plt.ylim(-2.5, 2.5)
        plt.xticks(())
        plt.yticks(())
        plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
                 transform=plt.gca().transAxes, size=15,
                 horizontalalignment='right')
        plot_num += 1

plt.show()
```



Clustering algorithms

https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68
(https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68)

Kmeans

- Pros:
  - K-Means has the advantage of fast calculation. It thus has a linear complexity O(n).
- Cons:
  - One has to select how many groups/classes there are.
  - K-means also starts with a random choice of cluster centers and therefore it may yield different clustering results on different runs of the algorithm.
  - Using means as cluster centers is also an disadvantage (see details later)
- K-Medians is another clustering algorithm related to K-Means, except instead of recomputing the group center points using the mean we use the median vector of the group. This method is less sensitive to outliers (because of using the Median) but is much slower for larger datasets as sorting is required on each iteration when computing the Median vector.

Expectation–Maximization (EM) Clustering using Gaussian Mixture Models (GMM)

- The major drawbacks of K-Means is its use of the mean value for the cluster center. This is not the best way of doing things as different clusters can have similar means. For example, two circular clusters with different radii centered at the same means. K-Means cannot handle this types of problems. Not just for these same-mean problems, K-Means also fails in cases where the clusters are not circular, e.g., the two-moon or swirl pattern.
- By assuming that the data points are Gaussian distributed as $N(\mu, \sigma)$, Gaussian Mixture Models (GMMs) give more flexibility than K-Means. First this is a less restrictive assumption than saying they are circular by using the mean. Second, an extra parameter covariance matrix σ in additional to mean vector μ indicates, e.g. in 2D case, that the clusters can take any kind of elliptical shape. In this sense, K-Means is actually a special case of GMM in which each cluster's covariance along all dimensions approaches 0.
- Since GMMs use probabilities, they can have multiple clusters per data point. We can define a data point's class by saying it belongs X-percent to class 1 and Y-percent to class 2. That is, GMMs support mixed membership.
- GMMs and Kmeans belong to the same group of EM algorithms. They both have guaranteed convergence in numerical calculation but also share the same local minimum problem. Their difference mainly lies in:
  - Kmeans hard-assigns probability while GMMs soft-assign probability.
  - Kmeans assumes symmetric distribution of variables (not skewed), same means and same variance, and covariance along all dimenstions approaches 0. GMMs assums non-zero covariance matrix, and data can be non-symmetric. Thus GMMs have more resolving power, e.g. to similar means but different clusters. GMMs can resolve very well the elliptical shape clusters as shown in the figures earlier. **However, for concentric circular clusters, GMMs seems perform not better than Kmeans**.

Mixture of naïve Bayes

Naïve Bayes model can be used in either supervised learning such as spam checking, or unsupervised learning such as news clustering. The model is covered in class but not appeared in notes. It is similar to the supervised case except some small changes.

Factor analysis: mixture of special models

It seems conceptually same as mixtures of Gaussian or naïve Bayes, except using Expectation Maximization (EM) on a special model where latent variable is usually with lower dimension than the data dimensions.

The Factor analysis is not so popular as mixture of the other two models.

Agglomerative Hierarchical Clustering

https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68 (https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68) First check the link above to understand the mechanisms of the algorithm.

- Hierarchical clustering algorithms actually fall into 2 categories: top-down or bottom-up. Bottom-up algorithms treat each data point as a single cluster at the outset and then successively merge (or agglomerate) pairs of clusters until all clusters have been merged into a single cluster that contains all data points. Bottom-up hierarchical clustering is therefore called **hierarchical agglomerative clustering or HAC**. 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. Check out the graphic below for an illustration before moving on to the algorithm steps
- We begin by treating each data point as a single cluster. We then select a distance metric that measures the distance between two clusters. On each iteration we combine two clusters into one. The two clusters to be combined are selected as those with the smallest average linkage.
- Repeat the above step until we reach the root of the tree i.e we only have one cluster which contains all data points. In this way we can select how many clusters we want in the end, simply by choosing when to stop combining the clusters i.e when we stop building the tree!
- Hierarchical clustering **does not require us to specify the number of clusters** and we can even select which number of clusters looks best since we are building a tree. Additionally, the algorithm is not sensitive to the choice of distance metric; all of them tend to work equally well whereas with other clustering algorithms, the choice of distance metric is critical. A particularly good use case of hierarchical clustering methods is when the underlying data has a hierarchical structure and you want to recover the hierarchy; other clustering algorithms can’t do this. These advantages of hierarchical clustering come at the cost of lower efficiency, as it has a time complexity of $O(n^3)$, unlike the linear complexity of K-Means and GMM.

Mean-Shift Clustering

- Check the link for an animation clearly demonstrating the algorithm.
- Mean shift clustering is a sliding-window-based algorithm that attempts to find dense areas of data points. It is a centroid-based algorithm meaning that the goal is to locate the center points of each group/class, which works by updating candidates for center points to be the mean of the points within the sliding-window.
- These candidate windows are then filtered in a post-processing stage to eliminate near-duplicates, forming the final set of center points and their corresponding groups.
- In contrast to K-means clustering there is **no need to select the number of clusters** as mean-shift automatically discovers this. That's a massive advantage.
- The drawback is that the selection of the window size/radius can be non-trivial.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

- DBSCAN is a density based clustered algorithm similar to mean-shift, but with a couple of notable advantages. Check out the animation in the link.
- DBSCAN does not require a pe-set number of clusters at all.
- It also identifies outliers as noises unlike mean-shift which simply throws them into a cluster even if the data point is very different.
- It is able to find arbitrarily sized and arbitrarily shaped clusters quite well.
- The main drawback of DBSCAN is that it doesn't perform as well as others when the clusters are of **varying density**. This is because the setting of the distance threshold ε and minPoints for identifying the neighborhood points will vary from cluster to cluster when the density varies. This drawback also occurs with very high-dimensional data since again the distance threshold ε becomes challenging to estimate.