Machine learning cookbook

By Kaveh Karimadini

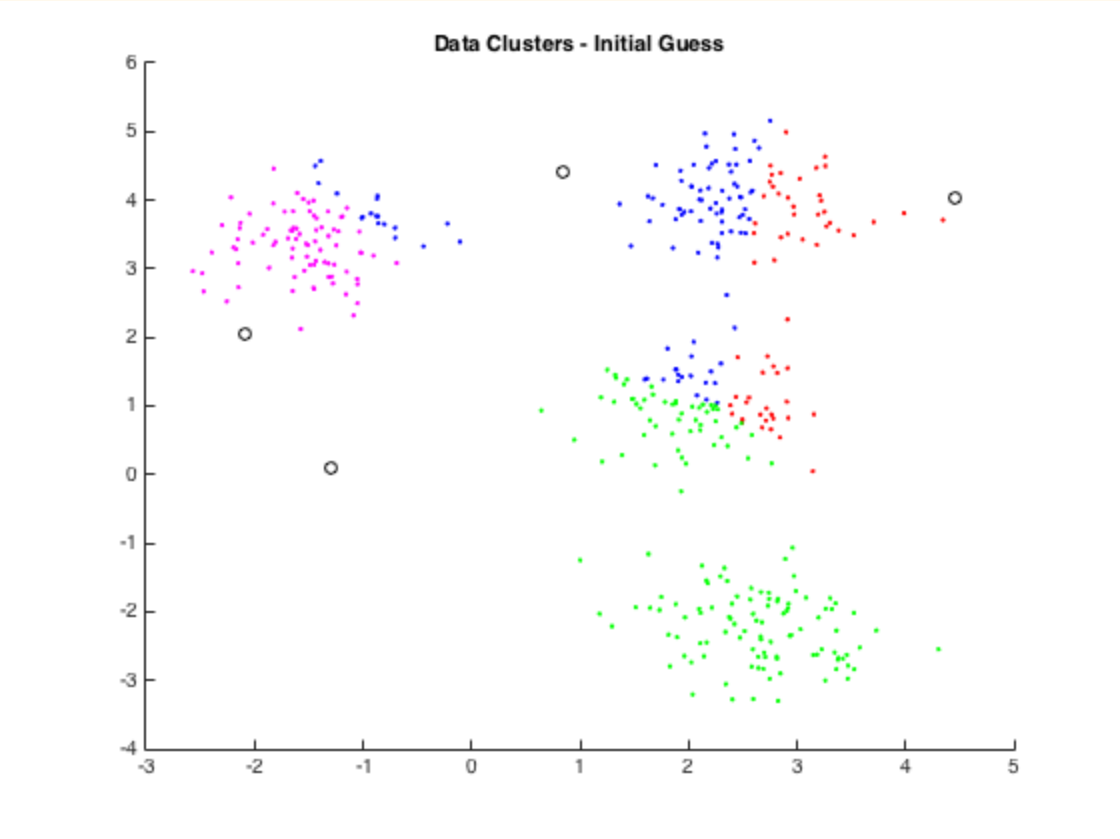
INTRODUCTION

Machine learning is a subset of artificial intelligence that enables computers to learn from data and improve over time. Coined in the 1950s, the term "machine learning" was defined by AI pioneer **Arthur Samuel** as "the field of study that gives computers the ability to learn without being explicitly programmed."

This technology has seen significant growth in recent years, driven by advances in statistics, computer science, and the availability of large datasets. Machine learning is used in various applications, such as automated translation, image recognition, voice search, and self-driving cars. It involves the development of algorithms and statistical models that allow computers to improve their performance through experience, making it a crucial component of many modern technologies.

The goal of machine learning is to **understand the structure of data and fit it into models that can be utilized by people**. Unlike traditional computational approaches, machine learning algorithms allow computers to train on data inputs and **use statistical analysis to automate decision-making** processes based on the data. This has led to the widespread use of machine learning in various technologies, such as facial recognition and predictive systems

Clustering:



★ The goal of clustering algorithms is to identify those latent groupings of observations which, if done well, allows us to predict the class of observations even without a target vector.

Clustering Using K-Means:

Problem:

You want to group observations into k groups.

Solution, Use k-means clustering:

```
# Load libraries
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
# Load data
iris = datasets.load_iris()
features = iris.data
# Standardize features
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
# Create k-means object
cluster = KMeans(n_clusters=3, random_state=0, n_init="auto")
# Train model
model = cluster.fit(features_std)
```

Discussion:

1. The algorithm attempts to group observations into k groups, with each group ideally having roughly equal variance.
    a.
2. The number of groups, k, is specified by the user as a hyperparameter. Specifically, in k-means:
    a. k cluster "center" points are created at random locations.
    b. For each observation:
        i. The distance between each observation and the k center points is calculated
        ii. The observation is assigned to the cluster of the nearest center point.
3. The center points are moved to the means (i.e., centers) of their respective clusters.
4. Steps 2 and 3 are repeated until no observation changes in cluster membership.

Note: k-means clustering ideal input:

- ☑ the clusters are convex shaped (e.g., a circle, a sphere).
- ☑ All features are equally scaled. We must standardize the features to meet this assumption.
- ☑ the groups are balanced (i.e., have roughly the same number of observations).

Implementation:

Visualizing K-Means Clustering(Click here)

- ★ K-Means clustering is implemented in the KMeans class. The most important parameter is n_clusters, which sets the number of clusters k.
- ★ We will want to select k based on using some criteria like silhouette coefficients which measure the similarity within clusters compared with the similarity between clusters.
- ★ K-Means clustering is computationally expensive, we might want to take advantage of all the cores(multi threading) on our computer. We can do this by setting `n_jobs=-1` in sklearn.

```
# View predicted class
model.labels_
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 2, 2, 2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2,
1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 2, 2, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 1, 2,
2, 1, 2, 1, 1, 2, 2, 2, 2, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 2, 2, 2,
2, 1, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 1], dtype=int32)
```

💯 If we compare this to the observation's true class, we can see that, despite the difference in class labels (i.e., 0, 1, and 2), k-means did reasonably well:

```
# View true class
iris.target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

The performance of k-means drops considerably, even critically, if we select the wrong number of clusters.

💯 We can use the trained cluster to **predict the value of new observations**:

```
  # Create new observation
    new_observation = [[0.8, 0.8, 0.8, 0.8]]
    # Predict observation's cluster
    model.predict(new_observation)
    array([2], dtype=int32)
```

💯 We can even use cluster_centers_ to see the values those center points:

```
# View cluster centers
model.cluster_centers_
array([[-1.01457897, 0.85326268, -1.30498732, -1.25489349],
[-0.01139555, -0.87600831, 0.37707573, 0.31115341],
[ 1.16743407, 0.14530299, 1.00302557, 1.0300019 ]])
```

Speeding Up K-Means Clustering:

Problem:

You want to group observations into k groups, but **k-means takes too long.**

Solution, Use mini-batch k-means:

```
# Load libraries
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MiniBatchKMeans
# Load data
iris = datasets.load_iris()
features = iris.data
# Standardize features
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
# Create k-mean object
```

```
cluster = MiniBatchKMeans(n_clusters=3, random_state=0,
batch_size=100,n_init="auto")
# Train model
model = cluster.fit(features_std)
```

Discussion:

- ★ The algorithm **takes small randomly chosen batches of the dataset for each iteration.** It then **updates the locations of cluster centroids based on** the new points from the **batch**. The update is a **gradient descent update**, faster than a **normal Batch K-Means update**.
- ★ This approach can significantly reduce the time required for the algorithm to find convergence (i.e., fit the data) with a **small cost in quality.**
- ★ batch_size controls the number of randomly selected observations in each batch. The larger the size of the batch, the more computationally costly the training process.

Clustering Using Mean Shift:

Problem:

You want to group observations without assuming the number of clusters or their shape.

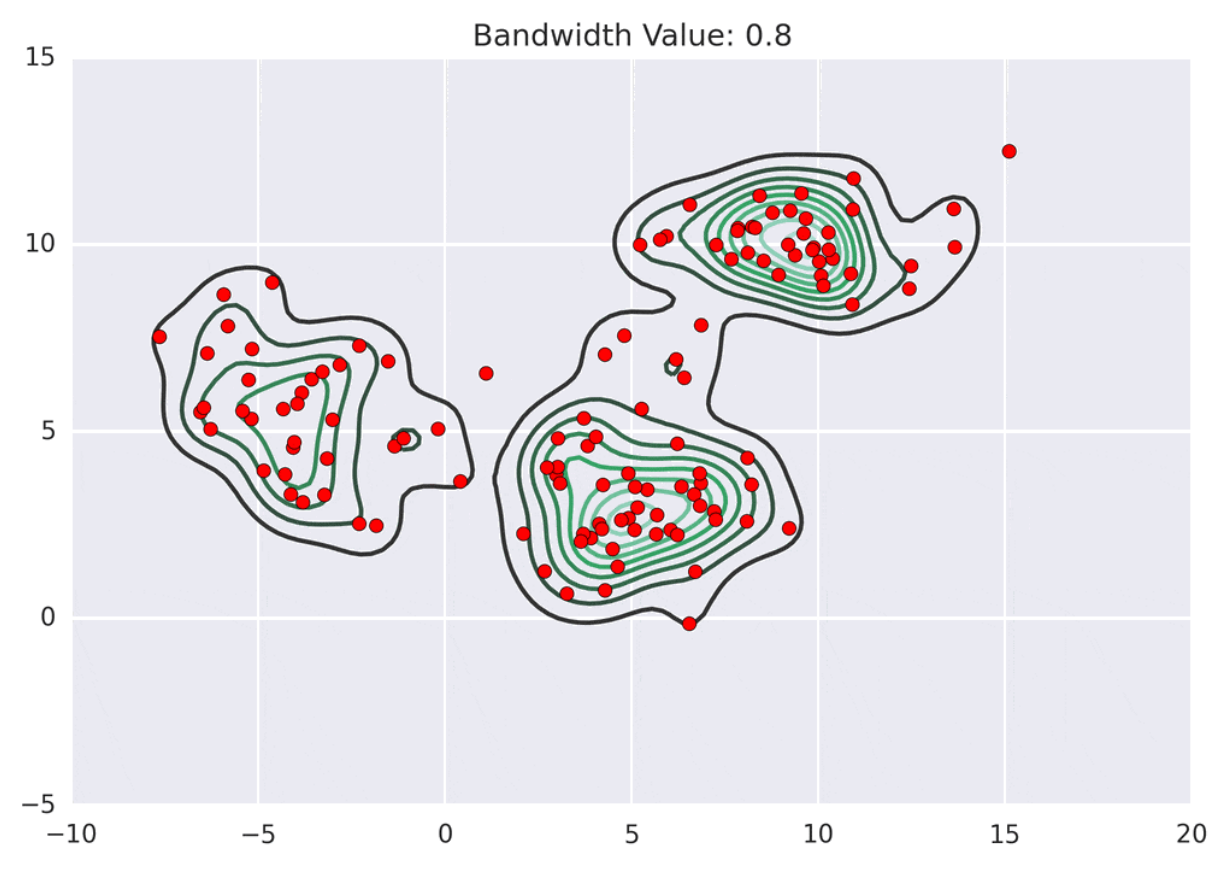
Solution, Use mean shift clustering:

```
# Load libraries
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MeanShift
# Load data
iris = datasets.load_iris()
features = iris.data
# Standardize features
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
# Create mean shift object
cluster = MeanShift(n_jobs=-1)
```

```
# Train model
model = cluster.fit(features_std)
```

Discussion:

- ★ No need to set the number of clusters, k, prior to training.It is a non-parametric, **density-based clustering algorithm** used to identify clusters in a dataset, particularly those with arbitrary shapes and **not well-separated.**
- ★ The algorithm works by shifting each data point **towards the mode** (i.e., the highest density) of the distribution of points ***within a certain radius***. It iteratively performs these shifts until the points **converge to a local maximum**, it is the point in the dataset where the density of data points is the highest. It can be **computationally expensive $O(n^2)$**.
- ★ **Analogy**: Imagine a very foggy football field (i.e., a two-dimensional feature space) with 100 people standing on it (i.e., our observations). Because it is foggy, a person can see only a short distance. **Every minute each person looks around and takes a step in the direction of the most people they can see. As time goes on, people start to group together as they repeatedly take steps toward larger and larger crowds.** The end result is clusters of people around the field.
- ★ MeanShift has two important parameters we should be aware of:
    - bandwidth sets the radius of the area (i.e., kernel) an observation uses to determine the direction to shift. In our analogy, bandwidth is how far a person can see through the fog. We can set this parameter manually, but by default a reasonable bandwidth is estimated automatically (with a significant increase in computational cost).
    - Sometimes in mean shifts there are no other observations within an observation's kernel. That is, a person on our football field cannot see a single other person. By default, MeanShift assigns all these "orphan" observations to the kernel of the nearest observation. However, if we want to leave out these orphans, we can set `cluster_all=False`, wherein orphan observations are given the label of -1.



Clustering Using DBSCAN

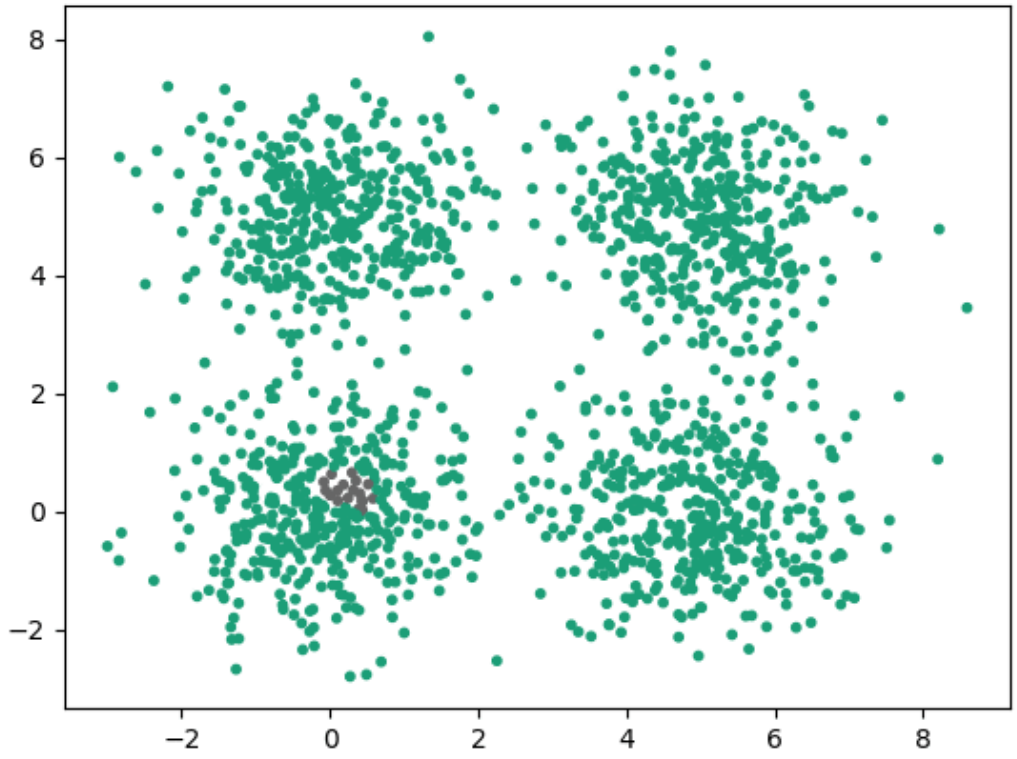
Problem:

You want to group observations into clusters of high density.

Solution, Use DBSCAN clustering:

```
# Load libraries
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
# Load data
iris = datasets.load_iris()
features = iris.data
# Standardize features
```

```
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
# Create DBSCAN object
cluster = DBSCAN(n_jobs=-1)
# Train model
model = cluster.fit(features_std)
```



Discussion:

Visualizing DBSCAN Clustering(click here)

- ★ DBSCAN is motivated by the idea that clusters will be areas where many observations are densely packed together and makes no assumptions of cluster shape.
- ★ In DBSCAN:

```
1. A random observation, xi, is chosen.
2. If xi has a minimum number of close neighbors, we consider it to be part
of a cluster.
3. Step 2 is repeated recursively for all of xi's neighbors, then
```

```
neighbor's neighbor and so on. These are the cluster's core observations.
4. Once step 3 runs out of nearby observations, a new random point is
chosen (i.e.,4.
restart at step 1).
```

★ Outlier detection: Any observation close to a cluster but not a core sample is considered part of a cluster, while any observation not close to the cluster is labeled an outlier.

★ DBSCAN has three main parameters to set:

```
eps:
    The maximum distance from an observation for another observation to be
considered its neighbor.

min_samples:
    The minimum number of observations less than eps distance from an
observation for it to be considered a core observation.

metric:
    The distance metric used by eps--for example, minkowski or euclidean
(note
    that if Minkowski distance is used, the parameter p can be used to set
the power
    of the Minkowski metric).
```

★ If we look at the clusters in our training data we can see two clusters have been identified, 0 and 1, while outlier observations are labeled -1:

```
# Show cluster membership
model.labels_
array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -1,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -1,
        0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,  0,  1,
        1,  1,  1,  1,  1, -1, -1,  1, -1, -1,  1, -1,  1,  1,  1,  1,  1,
       -1,  1,  1,  1, -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
       -1,  1, -1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1, -1,  1, -1,  1,
        1,  1,  1, -1, -1, -1, -1, -1,  1,  1,  1,  1, -1,  1,  1, -1, -1,
       -1,  1,  1, -1,  1,  1, -1,  1,  1,  1, -1, -1, -1,  1,  1,  1, -1,
       -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1])
```

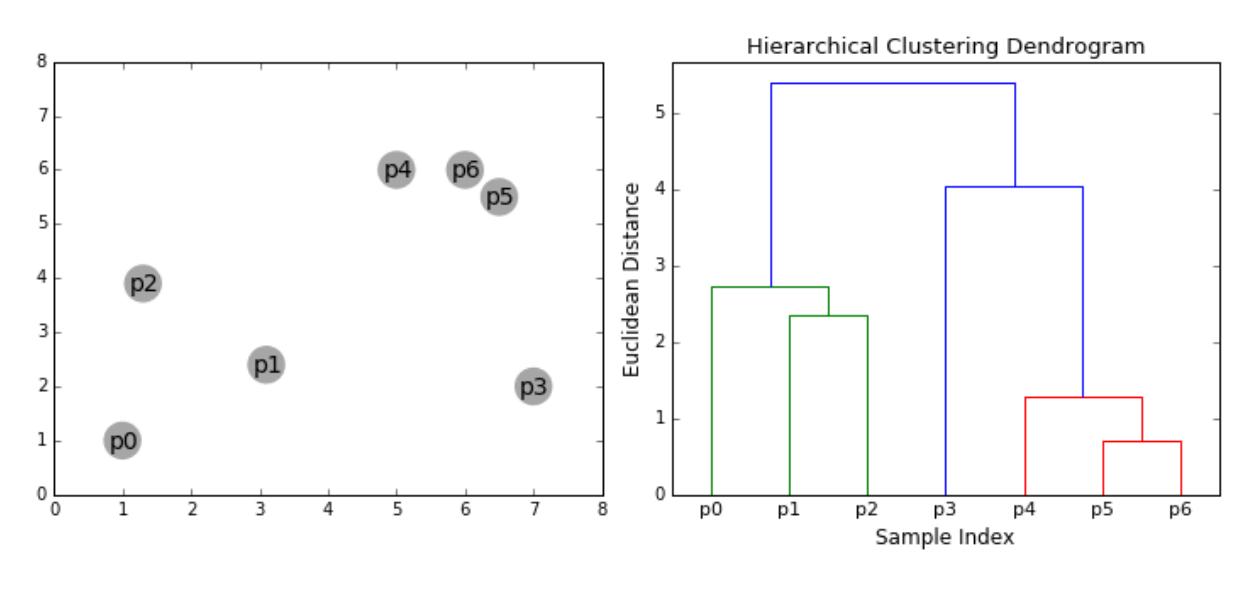
Clustering Using Hierarchical Merging:

Problem:

You want to group observations using a hierarchy of clusters.

Solution, Use agglomerative clustering:

```
# Load libraries
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering
# Load data
iris = datasets.load_iris()
features = iris.data
# Standardize features
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
# Create agglomerative clustering object
cluster = AgglomerativeClustering(n_clusters=3)
# Train model
model = cluster.fit(features_std)
```



Discussion:

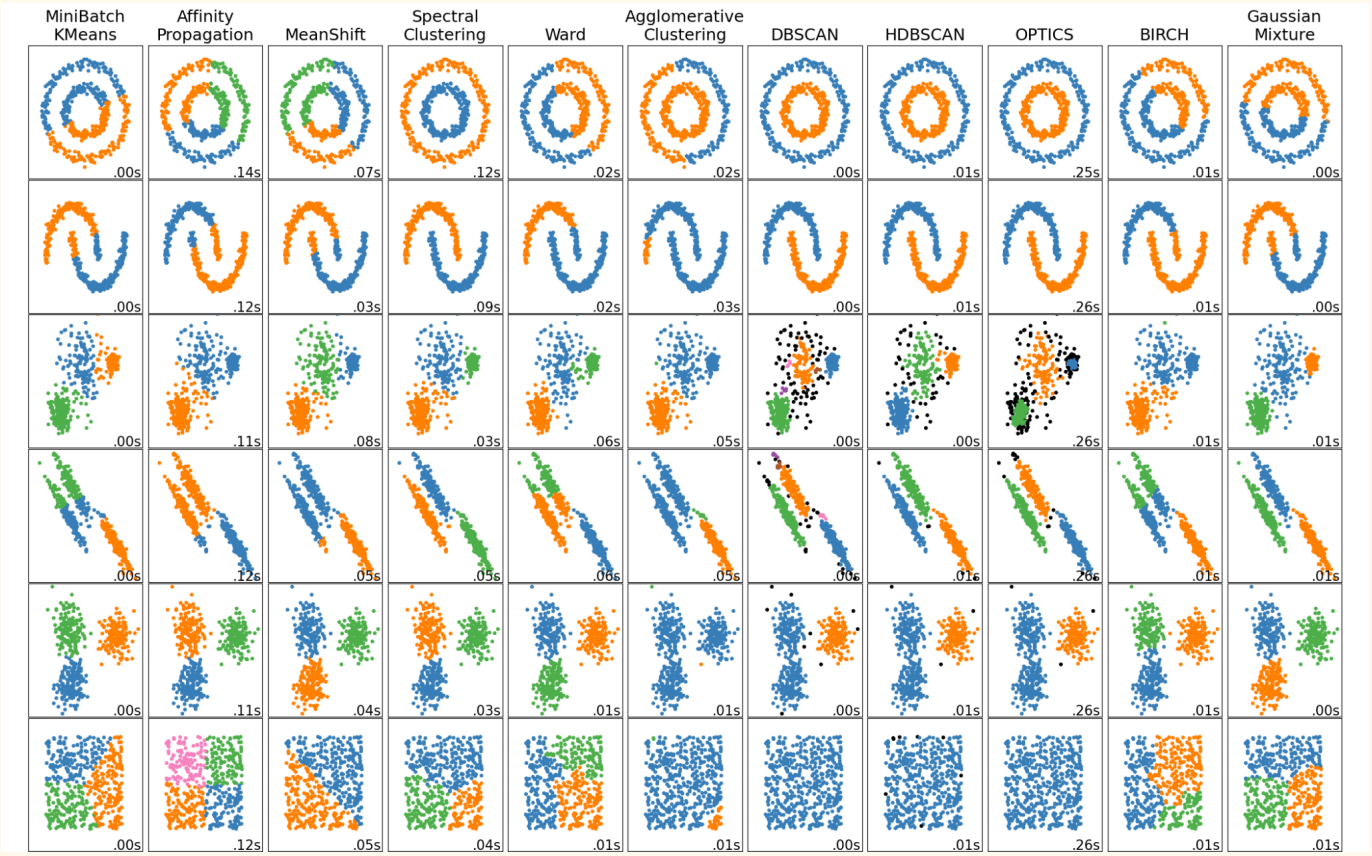
★ **Agglomerative clustering** is a powerful, flexible hierarchical clustering algorithm. At first all observations start as their own clusters. Next, clusters meeting some criteria are merged. This process is repeated, growing clusters until some end point is reached.

★ In scikit-learn, AgglomerativeClustering uses the linkage parameter to determine the merging strategy to minimize:

- Variance of merged clusters (ward)
- Average distance between observations from pairs of clusters (average)
- Maximum distance between observations from pairs of clusters (complete)

★ Two other parameters are useful to know.

1. The **affinity** parameter determines the distance metric used for linkage (minkowski, euclidean, etc.)
2. **n_clusters** sets the number of clusters the clustering algorithm will attempt to find. That is, clusters are successively merged until only n_clusters remain.


MiniBatch KMeans
Affinity Propagation
MeanShift
Spectral Clustering
Ward
Agglomerative Clustering
DBSCAN
HDBSCAN
OPTICS
BIRCH
Gaussian Mixture