# sci-kit learn K means

## • k means 介紹

k means 是一個聚類 (cluster) 的演算法，目的即是要一組資料將其分為 k 類。主要演算法如下：

1. 隨機選取資料組中的 k 筆資料當做初始的群集中心
2. 計算每個資料對應到最短距離的群中心
3. 以目前的分類重新計算群中心
4. 重複步驟 2 與 3，直到收斂

以下將介紹 sci-kit learn K means 幾個比較重要的 function

```
class KMeans(...):
```

```
    def __init__(self, n_clusters=8, init='k-means++', n_init=10,
                 max_iter=300, tol=1e-4, precompute_distances='auto',
                 verbose=0, random_state=None, copy_x=True, n_jobs=1):
```

```
    def fit(self, X, y=None):
        self.cluster_centers_, self.labels_, self.inertia_, self.n_iter_ = \
            k_means(
                X, n_clusters=self.n_clusters, init=self.init,
                n_init=self.n_init, max_iter=self.max_iter,
                verbose=self.verbose, return_n_iter=True,
                precompute_distances=self.precompute_distances,
                tol=self.tol, random_state=random_state, copy_x=self.copy_x,
                n_jobs=self.n_jobs)
        return self
```

## • __init__

__init__ 的功用即是要細部初始化 k means 裡的各項數值。

| |
|---|
| n-cluster：決定此演算法要分成幾類 |
| max_iter：每次執行演算法是最多執行幾輪（即是前面 k means 介紹裡的步驟 4） |
| n_init：隨機找尋群集中心會做幾次（即是前面介紹裡的步驟 1） |
| tol：每一輪找完新的群集中心後，然後前次中心與後一次中心移動的距離差不超過 tol，則我們認為 kmeans 已收斂 |
| n_jobs：決定可以有多少個 jobs 來跑 k means |

- **fit**

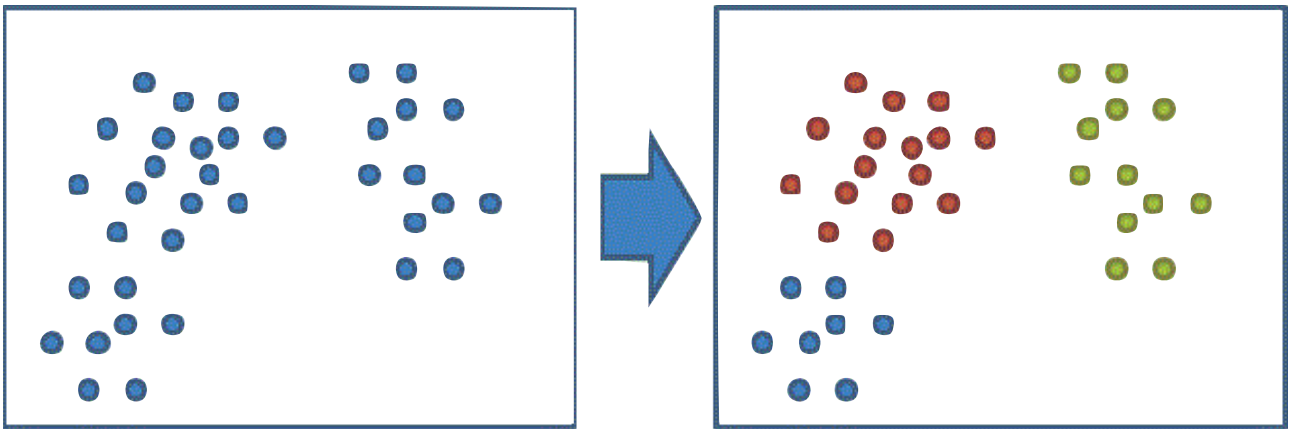
fit 的功用是找尋群集中心。他最主要是呼叫 k_means 這個 function 執行 k means 演算法。

**fit -> k_means**

```
def k_means(...)
    best_inertia = np.infty

    x_squared_norms = row_norms(X, squared=True)

    best_labels, best_inertia, best_centers = None, None, None

        for it in range(n_init):
            labels, inertia, centers, n_iter_ =
                _kmeans_single(
                X, n_clusters, max_iter=max_iter, init=init,
                verbose=verbose,
                precompute_distances=precompute_distances,
                tol=tol,x_squared_norms=x_squared_norms,
                random_state=random_state)

            # determine if these results are the best so far
            if best_inertia is None or inertia < best_inertia:
                best_labels = labels.copy()
                best_centers = centers.copy()
                best_inertia = inertia
                best_n_iter = n_iter_
    return best_centers, best_labels, best_inertia, best_n_iter
```

```
for it in range(n_init):
    ....
```

這個迴圈每跑一次便是從頭隨機找尋群集中心來做 k means 演算法。（即是上面介紹當中的步驟 1）而我們會在 n_init 次的 k means 演算法當中找尋最佳的群集中心點來做為我們的這次分類的結果。
此外，這個迴圈會呼叫 _kmeans_single 來做每一次的 k means 演算法。（即是上面介紹的步驟 2 到 4）



跑完 fit 函式後的結果圖例

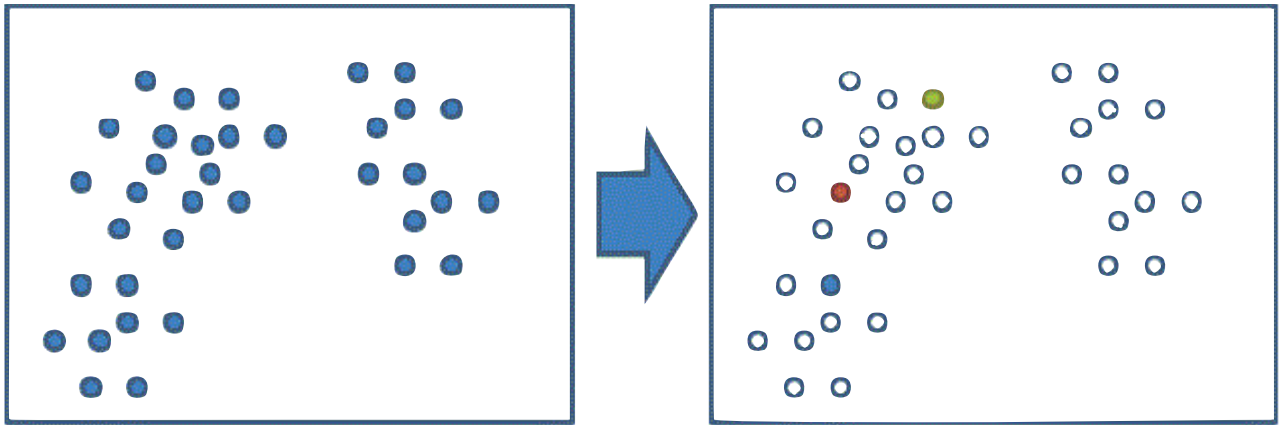
**fit -> k_means -> _kmeans_single**

```
def _kmeans_single(...):

    centers = _init_centroids(X, n_clusters, init,
                              random_state=random_state,
                              x_squared_norms=x_squared_norms)

    distances = np.zeros(shape=(X.shape[0],), dtype=np.float64)

    for i in range(max_iter):
        centers_old = centers.copy()
            labels, inertia = \
            _labels_inertia(X, x_squared_norms, centers,
                            precompute_distances=precompute_distances,
                            distances=distances)

            centers = _k_means._centers_dense(X, labels, n_clusters,
                                              distances)

        if best_inertia is None or inertia < best_inertia:
            best_labels = labels.copy()
            best_centers = centers.copy()
            best_inertia = inertia

        if squared_norm(centers_old - centers) <= tol:
            break

    return best_labels, best_inertia, best_centers, i + 1
```

`_init_centroids(…)`：這個 function 主要是初始化最剛開始的群集中心

`_labels_inertia(…)`：這個 function 將會決定每個點距離自己最近的中心，並將距離加總

`_center_dense(…)`：這個 function 主要是重新去找尋新的群集中心



跑完 _init_centroids 函式後的結果圖例

**fit -> k_means -> _kmeans_single -> _labels_inertia**

```
def _labels_inertia(...):
    return labels_inertia_precompute_dense(X,
           x_squared_norms,centers,distances)
```

labels_inertia_precompute_dense(…)：這個 function 會回傳每個點距自己最近的中心與距離和

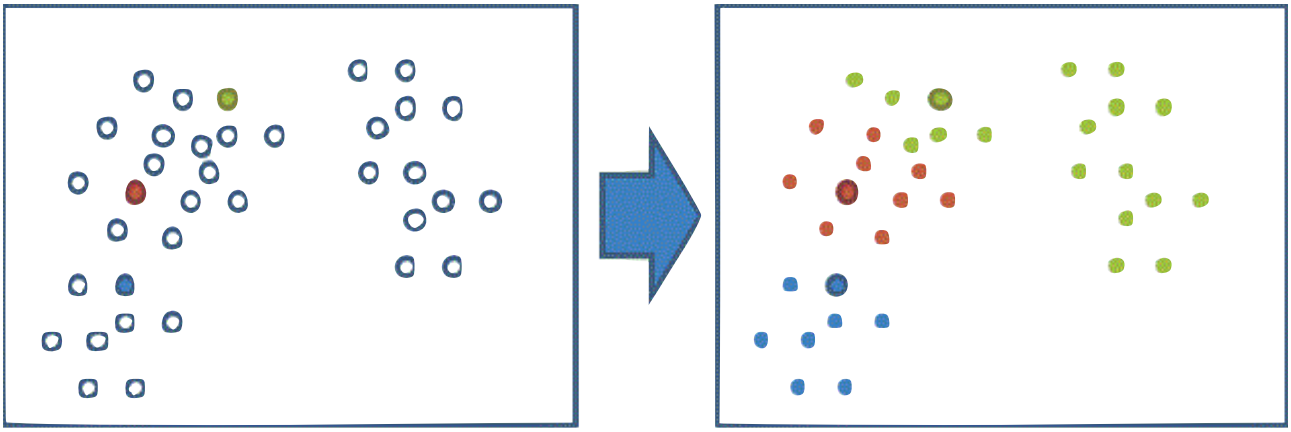
**fit -> k_means -> _kmeans_single -> _labels_inertia -> _labels_inertia_precompute_dense**

```
def _labels_inertia_precompute_dense(...):
    n_samples = X.shape[0]
    k = centers.shape[0]

    all_distances = euclidean_distances(centers, X, x_squared_norms,
                                        squared=True)

    labels = np.empty(n_samples, dtype=np.int32)
    labels.fill(-1)

    mindist = np.empty(n_samples)
    mindist.fill(np.infty)

    for center_id in range(k):
        dist = all_distances[center_id]
        labels[dist < mindist] = center_id
        mindist = np.minimum(dist, mindist)

    if n_samples == distances.shape[0]:
        distances[:] = mindist

    inertia = mindist.sum()

    return labels, inertia
```

euclidean_distances(...)：計算點與點間的距離

```
for center_id in range(k):
    ...
```

這個迴圈將會找詢每個點對於自己而言最近的中心點



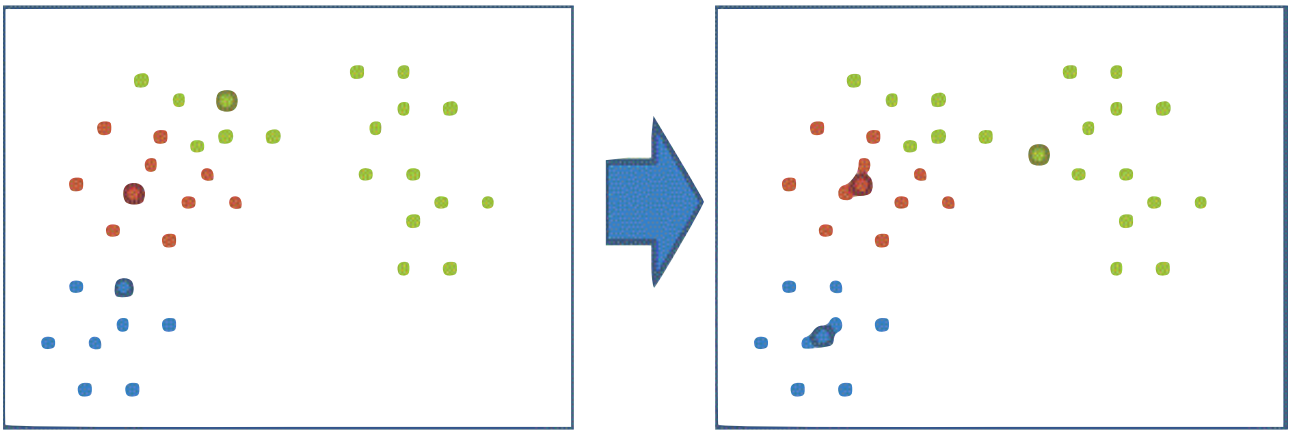
跑完 _labels_inertia_precompute_dense 函式後的結果圖例

**fit -> k_means -> _kmeans_single -> _centers_dense**

```
def _centers_dense(...):
    cdef int n_samples, n_features

    n_samples = X.shape[0]
    n_features = X.shape[1]

    cdef int i, j, c

    cdef np.ndarray[DOUBLE, ndim=2] centers = np.zeros((n_clusters,
                                                        n_features))
    n_samples_in_cluster = np.bincount(labels, minlength=n_clusters)
    empty_clusters = np.where(n_samples_in_cluster == 0)[0]

    if len(empty_clusters):
       far_from_centers = distances.argsort()[::-1]

    for i, cluster_id in enumerate(empty_clusters):
        new_center = X[far_from_centers[i]]
        centers[cluster_id] = new_center
        n_samples_in_cluster[cluster_id] = 1

    for i in range(n_samples):
        for j in range(n_features):
            centers[labels[i], j] += X[i, j]

    centers /= n_samples_in_cluster[:, np.newaxis]

    return centers
```

```
for i in range(n_samples):
    for j in range(n_features):
        ....
centers /= n_samples_in_cluster[:, np.newaxis]
```

這個雙重迴圈將會把各個群集裡的點集合起來，並在下一行的程式碼算出平均，取得新的群集中心



跑完 _centers_dense 函式後的結果圖例