05.11-K-Means_cak20190330

March 30, 2019

Remark: The original file has been partially chosen for teaching in 2102541 IoT Fundamentals. Try varying the number of clusters to be generated and to be found as well as other parameters (C. Aswakul 30 Mar 2019).

This notebook contains an excerpt from the Python Data Science Handbook by Jake VanderPlas; the content is available on GitHub.

< In-Depth: Manifold Learning | Contents | In Depth: Gaussian Mixture Models >

1 In Depth: k-Means Clustering

In the previous few sections, we have explored one category of unsupervised machine learning models: dimensionality reduction. Here we will move on to another class of unsupervised machine learning models: clustering algorithms. Clustering algorithms seek to learn, from the properties of the data, an optimal division or discrete labeling of groups of points.

Many clustering algorithms are available in Scikit-Learn and elsewhere, but perhaps the simplest to understand is an algorithm known as *k-means clustering*, which is implemented in `sklearn.cluster.KMeans`.

We begin with the standard imports:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn as sns; sns.set()  # for plot styling
        import numpy as np
```

1.1 Introducing k-Means

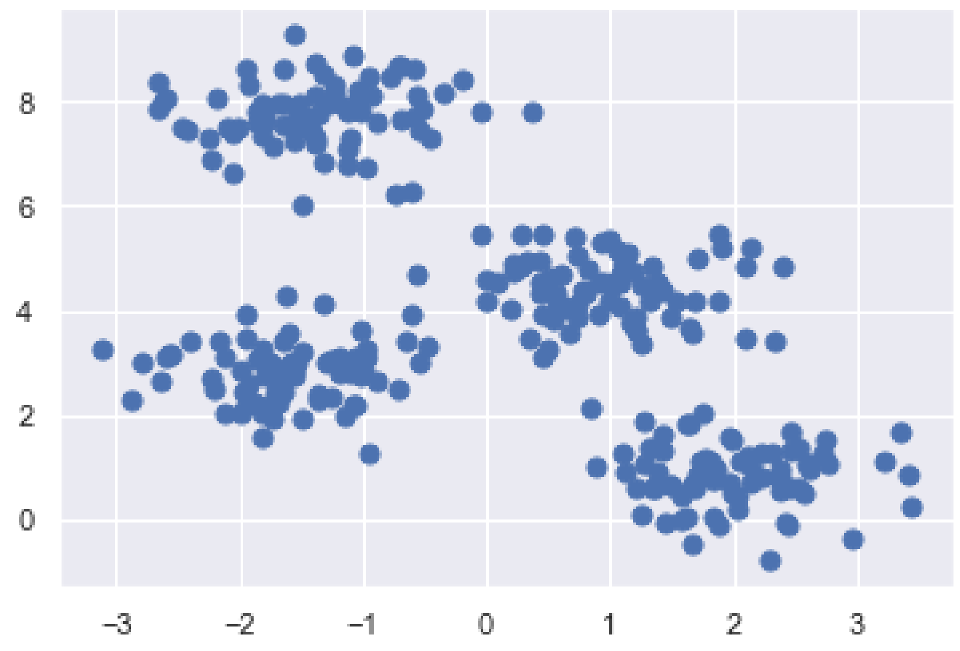
The *k*-means algorithm searches for a pre-determined number of clusters within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like:

- The "cluster center" is the arithmetic mean of all the points belonging to the cluster.
- Each point is closer to its own cluster center than to other cluster centers.

Those two assumptions are the basis of the *k*-means model. We will soon dive into exactly *how* the algorithm reaches this solution, but for now let's take a look at a simple dataset and see the *k*-means result.

First, let's generate a two-dimensional dataset containing four distinct blobs. To emphasize that this is an unsupervised algorithm, we will leave the labels out of the visualization

```
In [2]: from sklearn.datasets.samples_generator import make_blobs
        X, y_true = make_blobs(n_samples=300, centers=4,
                               cluster_std=0.60, random_state=0)
        plt.scatter(X[:, 0], X[:, 1], s=50);
```

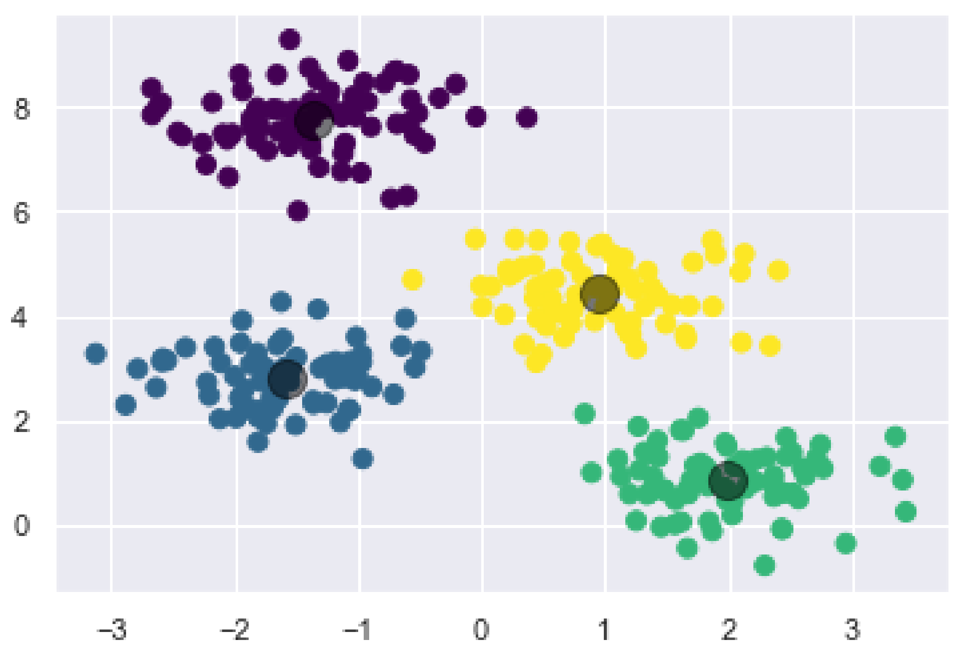


By eye, it is relatively easy to pick out the four clusters. The *k*-means algorithm does this automatically, and in Scikit-Learn uses the typical estimator API:

```
In [3]: from sklearn.cluster import KMeans
        kmeans = KMeans(n_clusters=4)
        kmeans.fit(X)
        y_kmeans = kmeans.predict(X)
```

Let's visualize the results by plotting the data colored by these labels. We will also plot the cluster centers as determined by the *k*-means estimator:

```
In [4]: plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

        centers = kmeans.cluster_centers_
        plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
```



The good news is that the k-means algorithm (at least in this simple case) assigns the points to clusters very similarly to how we might assign them by eye. But you might wonder how this algorithm finds these clusters so quickly! After all, the number of possible combinations of cluster assignments is exponential in the number of data points—an exhaustive search would be very, very costly. Fortunately for us, such an exhaustive search is not necessary: instead, the typical approach to k-means involves an intuitive iterative approach known as *expectation–maximization*.

1.2 k-Means Algorithm: Expectation–Maximization

Expectation–maximization (E–M) is a powerful algorithm that comes up in a variety of contexts within data science. k-means is a particularly simple and easy-to-understand application of the algorithm, and we will walk through it briefly here. In short, the expectation–maximization approach here consists of the following procedure:

1. Guess some cluster centers
2. Repeat until converged
3. *E-Step*: assign points to the nearest cluster center
4. *M-Step*: set the cluster centers to the mean

Here the "E-step" or "Expectation step" is so-named because it involves updating our expectation of which cluster each point belongs to. The "M-step" or "Maximization step" is so-named because it involves maximizing some fitness function that defines the location of the cluster centers—in this case, that maximization is accomplished by taking a simple mean of the data in each cluster.

The literature about this algorithm is vast, but can be summarized as follows: under typical circumstances, each repetition of the E-step and M-step will always result in a better estimate of the cluster characteristics.

We can visualize the algorithm as shown in the following figure. For the particular initialization shown here, the clusters converge in just three iterations. For an interactive version of this figure, refer to the code in the Appendix.

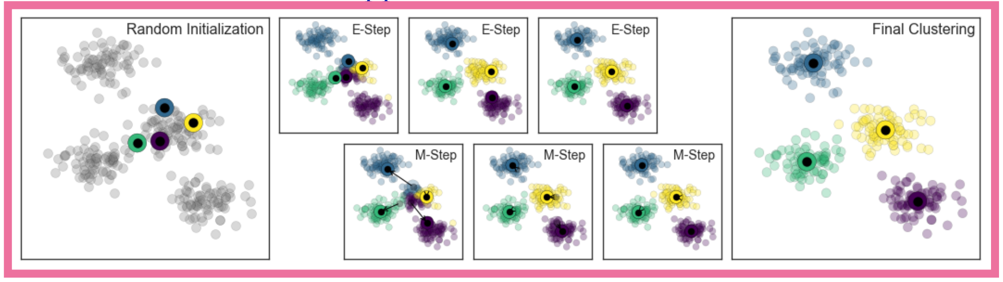
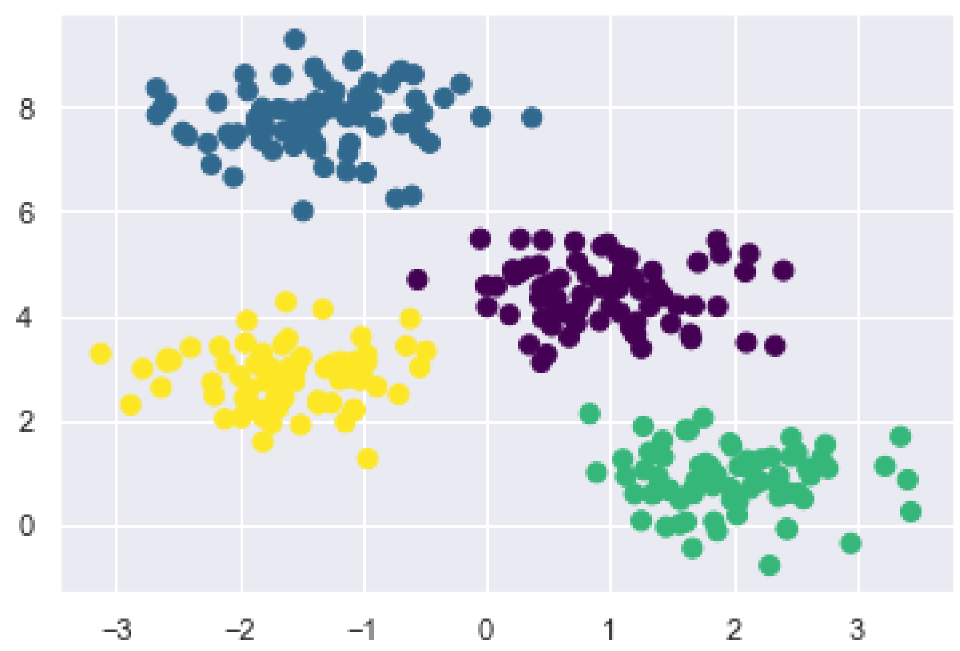


figure source in Appendix

The *k*-Means algorithm is simple enough that we can write it in a few lines of code. The following is a very basic implementation:

```
In [5]: from sklearn.metrics import pairwise_distances_argmin

        def find_clusters(X, n_clusters, rseed=2):
            # 1. Randomly choose clusters
            rng = np.random.RandomState(rseed)
            i = rng.permutation(X.shape[0])[:n_clusters]
            centers = X[i]

            while True:
                # 2a. Assign labels based on closest center
                labels = pairwise_distances_argmin(X, centers)

                # 2b. Find new centers from means of points
                new_centers = np.array([X[labels == i].mean(0)
                                        for i in range(n_clusters)])

                # 2c. Check for convergence
                if np.all(centers == new_centers):
                    break
                centers = new_centers

            return centers, labels

In [6]: centers, labels = find_clusters(X, 4)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```



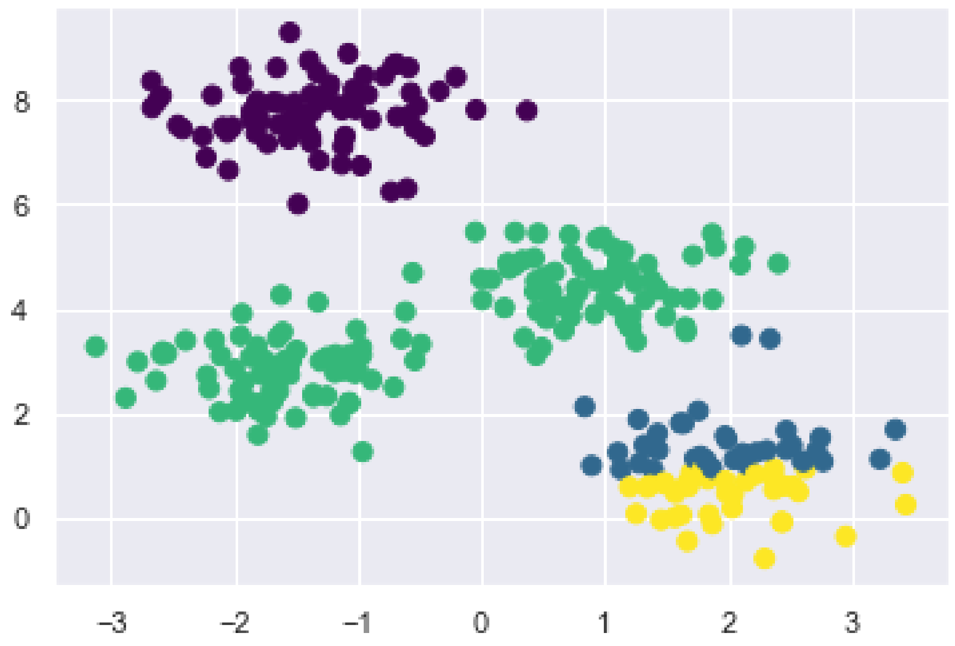
Most well-tested implementations will do a bit more than this under the hood, but the preceding function gives the gist of the expectation–maximization approach.

1.2.1 Caveats of expectation–maximization

There are a few issues to be aware of when using the expectation–maximization algorithm.

The globally optimal result may not be achieved First, although the E–M procedure is guaranteed to improve the result in each step, there is no assurance that it will lead to the *global* best solution. For example, if we use a different random seed in our simple procedure, the particular starting guesses lead to poor results:

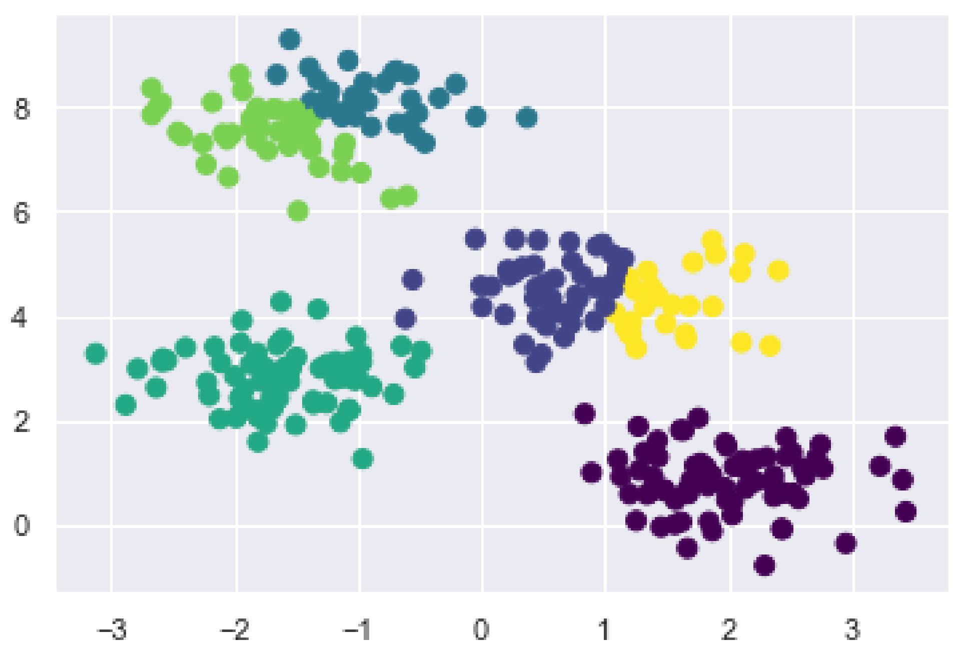
```
In [7]: centers, labels = find_clusters(X, 4, rseed=0)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```



Here the E–M approach has converged, but has not converged to a globally optimal configuration. For this reason, it is common for the algorithm to be run for multiple starting guesses, as indeed Scikit-Learn does by default (set by the `n_init` parameter, which defaults to 10).

The number of clusters must be selected beforehand Another common challenge with *k*-means is that you must tell it how many clusters you expect: it cannot learn the number of clusters from the data. For example, if we ask the algorithm to identify six clusters, it will happily proceed and find the best six clusters:

```
In [8]: labels = KMeans(6, random_state=0).fit_predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```



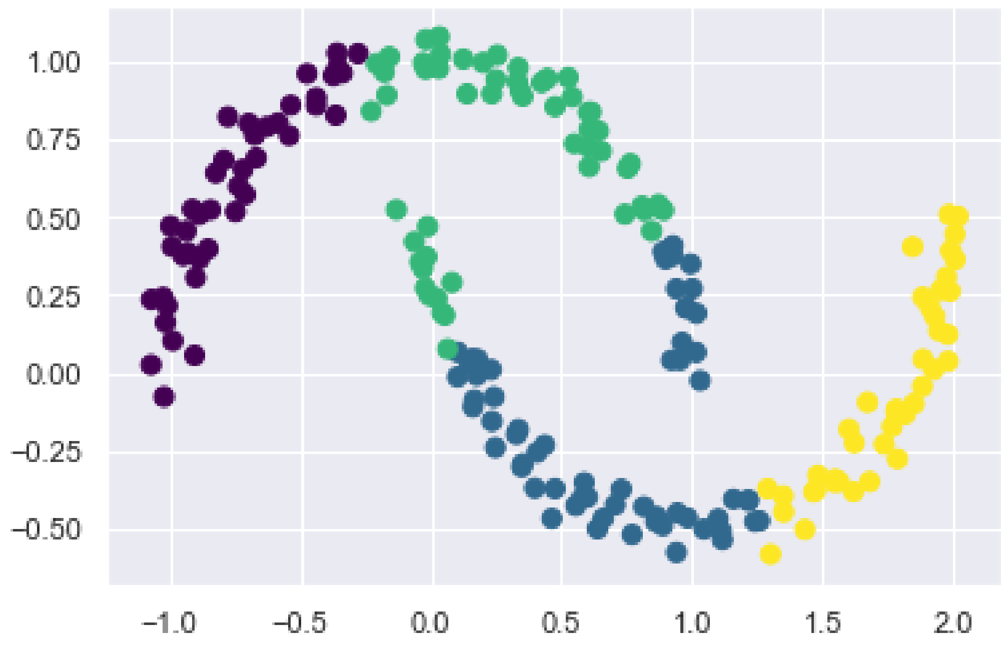
Whether the result is meaningful is a question that is difficult to answer definitively; one approach that is rather intuitive, but that we won't discuss further here, is called silhouette analysis.

Alternatively, you might use a more complicated clustering algorithm which has a better quantitative measure of the fitness per number of clusters (e.g., Gaussian mixture models; see In Depth: Gaussian Mixture Models) or which *can* choose a suitable number of clusters (e.g., DBSCAN, mean-shift, or affinity propagation, all available in the `sklearn.cluster` submodule)

k-means is limited to linear cluster boundaries The fundamental model assumptions of *k*-means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters have complicated geometries.

In particular, the boundaries between *k*-means clusters will always be linear, which means that it will fail for more complicated boundaries. Consider the following data, along with the cluster labels found by the typical *k*-means approach:

```
In [9]: from sklearn.datasets import make_moons
        X, y = make_moons(200, noise=.05, random_state=0)

In [10]: labels = KMeans(4, random_state=0).fit_predict(X)
         plt.scatter(X[:, 0], X[:, 1], c=labels,
                     s=50, cmap='viridis');
```

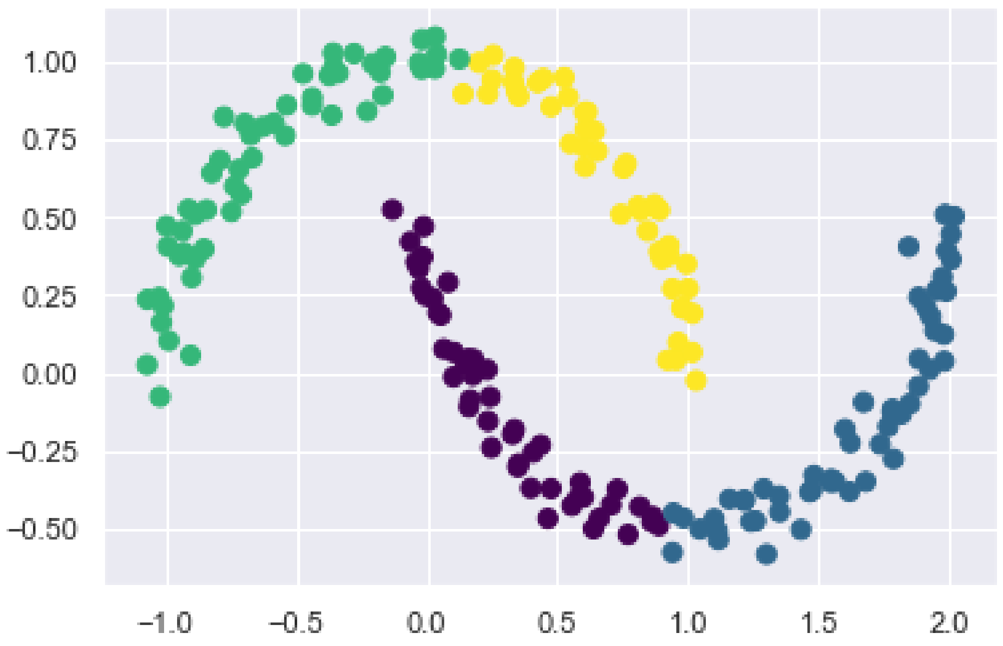


This situation is reminiscent of the discussion in In-Depth: Support Vector Machines, where we used a kernel transformation to project the data into a higher dimension where a linear separation is possible. We might imagine using the same trick to allow *k*-means to discover non-linear boundaries.

One version of this kernelized *k*-means is implemented in Scikit-Learn within the `SpectralClustering` estimator. It uses the graph of nearest neighbors to compute a higher-dimensional representation of the data, and then assigns labels using a *k*-means algorithm:

```
In [11]: from sklearn.cluster import SpectralClustering
         model = SpectralClustering(n_clusters=4, affinity='nearest_neighbors',
                                    assign_labels='kmeans')
         labels = model.fit_predict(X)
         plt.scatter(X[:, 0], X[:, 1], c=labels,
                     s=50, cmap='viridis');
```

```
/Users/chaodit/anaconda3/lib/python3.7/site-packages/sklearn/manifold/spectral_embedding_.py:237
  warnings.warn("Graph is not fully connected, spectral embedding"
```



We see that with this kernel transform approach, the kernelized *k*-means is able to find the more complicated nonlinear boundaries between clusters.

k-means can be slow for large numbers of samples Because each iteration of *k*-means must access every point in the dataset, the algorithm can be relatively slow as the number of samples grows. You might wonder if this requirement to use all data at each iteration can be relaxed; for example, you might just use a subset of the data to update the cluster centers at each step. This is the idea behind batch-based *k*-means algorithms, one form of which is implemented in `sklearn.cluster.MiniBatchKMeans`. The interface for this is the same as for standard `KMeans`; we will see an example of its use as we continue our discussion.

< In-Depth: Manifold Learning | Contents | In Depth: Gaussian Mixture Models >