This is an excerpt from the Python Data Science Handbook by Jake VanderPlas; jupyter notebooks are available on GitHub.
The text is released under the CC-BY-NC-SA license, and code is released under the MIT license. If you find this content useful, please consider supporting the work on Amazon or buying the book.

In Depth: k-Means Clustering

← In Depth: Manifold Learning | Contents | In Depth: Gaussian Mixture Models →

Open in Colab

In the previous five sections, we have explored one category of unsupervised machine learning models: dimensionality reduction. Here we will move on to another category of unsupervised machine learning: 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 the 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:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np; np.random.RandomGenerator(42)
import seaborn as sns; sns.set(); # for plot styling
import pandas as pd
```

Introducing k-Means

The k-means algorithm searches for a predetermined number of clusters within an unlabeled multidimensional dataset, in such a way that it chooses the cluster centers so that the sum of the squared distances to the nearest cluster center is as small as possible. The "optimal" cluster is the arbitrarily chosen set of the points belonging to the cluster with the smallest radius.

Kaff-Lyubich's theorem for the k-means criterion implies that there always exists an optimal partition of the data into clusters, such that each data point is closer to its own cluster center than to any other cluster center.

These two assumptions are the basis of the k-means model; we will soon see how exactly this intuition algorithm carries this out, 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:

```
%matplotlib inline
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:

```
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:

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

centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1],c=range(4),cmap='viridis',
            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 approach can be applied to datasets where the clusters aren't so well-separated; or how the algorithm performs on more complicated data. Together with other assumptions, the k-means model derives its power and interpretability from the notion that it assumes a single cluster to be represented by the cluster center. This assumption makes it ripe for confusion in the case of more complicated clusters. We'll see an example of this below.

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 an particularly simple and easy-to-understand application of the algorithm, and we will briefly walk through the details here. In a nutshell, the expectation-maximization approach here consists of the following procedure:

1. Guess some cluster centers
2. Repeat until converged
   a. E-step: assign points to the nearest cluster center
   b. 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 points are in each cluster. 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: in this case, the 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, three iterations to converge. We'll cover the details of the interactive version of the figure, refer to the book for details.

Source: Expectation Maximization

This is a fairly straightforward algorithm to implement. Here we'll walk through a simple, rudimentary implementation of the algorithm, beginning with a random initialization of the cluster centers:

```
%matplotlib inline
from sklearn.metrics.pairwise import pairwise_distances
def find_clusters(X, n_clusters, rseed=2):
    # 1. Randomly choose clusters
    rng = np.random.RandomGenerator(rseed)
    i = rng.permutation(X.shape[0])[:n_clusters]
    centers = X[i]

    while True:
        # 2a. assign labels based on closest center
        labels = pairwise_distances(X, centers).argmin(axis=1)

        # 2b. find new centers from means of points
        new_centers = np.array([X[labels == i].mean(axis=0),
                                X[labels == i].mean(axis=0)])

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

    return centers, labels

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 underlying functional ideas are the same: expectation-maximization approach.

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 guarantee that it will land on the global best solution. For example, if we start off with a suboptimal set of cluster centers, the particular starting guesses lead to poor results:

```
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 Scikit-Learn does by default (set by the `n_init` parameter, with a default of 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 for five clusters, it will happily proceed and find the best fit four clusters:

```
centers, labels = find_clusters(X, 5, rseed=0)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```

Whether this result is meaningful is a question that is difficult to answer definitively; one approach that might be rewarding in this case is a more complicated form of cluster model, such as Gaussian mixture models.

Alternatively, you might use a more complicated clustering algorithm which has a better quantitative measure of the fitness per number of clusters (for example, Gaussian mixture models do, as they can be compared using cross-validation or using the Akaike information criterion or Bayesian information criterion), or algorithms which do not require a specification of the number of clusters at all (for example, 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 the cluster center they are assigned to than anything else) suggest that k-means will often fail when 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:

```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05, random_state=0)
```

```
centers, labels = find_clusters(X, 2, rseed=0)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='viridis');
```

This clustering result is not what one would intuitively describe as a good clustering. Moreover, the k-means model has no intrinsic measure of probability or uncertainty for cluster assignments (although it might be possible to use a bootstrap resampling approach for this).

Finally, 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 for the number of samples given. You might think that you could speed up the k-means algorithm by simply retiring points that seem well-assigned, and focusing in on the points that seem to need further iteration. This is the idea of batch-based k-means clustering, which is implemented in `sklearn.cluster.MiniBatchKMeans`.

The k-means algorithm is flexible in this way and many more ways than we have seen here. Most importantly, you can use k-means with any pre-processing or feature extraction algorithm, which makes this algorithm is a very flexible and powerful tool in your machine learning toolbox.

Examples

Being so simple and yet so flexible, k-means is used in many situations. Despite its drawbacks, most of the issues we've discussed here may be relevant for any application of k-means, but it may be a useful first step in a more complicated analysis.

Example 1: k-means on digits

To take a look at how k-means is used in practice, let's apply it to the digits data that we saw earlier in the discussion of manifold learning and decomposition. Here we will attempt to use k-means to try and identify similar digits without using the original label information; this might be similar to the first step we could take in a digit classification task if we were unaware of the meaning of the pixel data representation.

We will start by loading the digits and then finding the Isomap embedding of the data, as we did in the manifold learning discussion. We will keep all of the Isomap parameters at the hyperparameters of our pipeline:

```
from sklearn.datasets import load_digits
digits, y = load_digits()
```

```
%time
```

```
CPU times: user 2.48 s, sys: 1.07 s, total: 3.55 s
Wall time: 1.8 s
```

The clustering can be performed as we did before:

```
%time
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape
```

```
CPU times: user 1.34 s, sys: 190 ms, total: 1.53 s
Wall time: 820 ms
```

```
(10, 64)
```

The result is 10 clusters in 64 dimensions. Notice that the cluster centers themselves are also dimensional points, and can theoretically be interpreted as "typical" digits in the cluster. Let's see what these cluster centers look like:

```
fig, ax = plt.subplots(8, 5, figure=fig, )
for axi, center in zip(ax.flat, kmeans.cluster_centers_):
    axi.set_frame_on(False)
    axi.imshow(center.reshape(8, 8), cmap=plt.cm.binary,)
```

We find that clusters look almost like digits, or at least enough that we can recognize most of the digits.

Because k-means knows nothing about the identity of the clusters, this is of course, not by accident. We can find the best mapping between true labels and cluster labels with a simple lookup table:

```
from scipy.stats import mode

labels = np.zeros_like(clusters)
for i in range(10):
    mask = (clusters == i)
    labels[mask] = mode(y_true[mask])[0]
```

Now we can check how accurate our unsupervised clustering was in finding similar digits within the data:

```
from sklearn.metrics import accuracy_score
accuracy_score(y_true, labels)
```

```
0.7935447968836951
```

With just a simple k-means algorithm, we discovered the correct grouping for 80% of the input digits! Let's check the confusion matrix for this:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(y_true, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d',
            cbar=False,
            xticklabels=np.arange(10),
            yticklabels=np.arange(10),
            cmap='Blues')
plt.xlabel('true label')
plt.ylabel('predicted label');
```

Accurate labels across the cluster centers are visualized darkest, the most common of confusions is between the digits 1 and 8. But this is to be expected. For example, we can expect a fully built digit classifier without errors to be known (darkest).

Let's try this out to see how good a more flexible, little-data-aware supervised classification algorithm (in this case, a Gaussian naive Bayes classifier) would perform on this set of digits. We'll use the `cross_val_score` cross-validation utility to quickly and efficiently evaluate the score of the classifier on the projected data:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score
cross_val_score(GaussianNB(), digits.data[:,:64], y, cv=5)
```

```
array([0.81 , 0.84 , 0.82 , 0.83 , 0.81 ])
```

We see that we've cut the error, jumping from 80% to around 82%. Let's now probe these models over the parameter space, to determine how they should be used in practice.

For this, we must use cross-validation, so we'll use `validation_curve`. Here we will vary the number of components in the Isomap model:

```
%time
from sklearn.model_selection import validation_curve
n_components = np.arange(2, 21)
scores = validation_curve(Isomap(n_jobs=1), digits.data[:1000],
                          y[:1000],
                          cv=3,
                          param_range=n_components)
```

```
CPU times: user 1.2 s, sys: 1.12 s, total: 2.32 s
Wall time: 1.8 s
```

Now let's look at the results:

```
plt.plot(n_components, scores[0], '-', alpha=0.5, lw=2, color='blue')
plt.plot(n_components, scores[1], '-', alpha=0.5, lw=2, color='red')
plt.fill_between(n_components, scores[0].ravel(), scores[1].ravel(),
                 alpha=0.3, color='k')
plt.xlim(0, 20)
plt.ylim(0.7, 1.0)
plt.xlabel('n_components')
plt.ylabel('cross-validation score')
plt.colorbar(ticks=[0, 0.5, 1],
             boundaries=[-0.5, 2.5, 5.5, 7.5],
             format=plt.NullFormatter(),
             # to avoid ticks
             ticklocation='bottom',
             label='n_neighbors');
```

This shows that we can expect to achieve a maximum accuracy of around 0.94 using 6 components, but we can gain almost all of that with a much more restrictive dimensionality of about 10 components.

Example 2: k-means for color compression

One interesting application of clustering is in color compression within images. For example, imagine you have an image with millions of colors. In most images, a large number of the colors will be unused, and many of the colors in the image will be very similar or even identical colors.

For example, consider the image shown in the following figure, which is from the Scikit-Learn datasets. Variable the Rio Vista bridge, you'll have to have the `pillow` Python package installed:

```
%matplotlib inline
from sklearn.datasets import load_sample_image
china = load_sample_image("china.jpg")
ax = plt.axes(xticks=[], yticks=[])
ax.imshow(china)
```


The image itself is stored in a three-dimensional array of size `(height, width, RGB channels)`, containing red/green/blue contributions as integers from 0 to 255:

```
china.shape
```

```
(427, 640, 3)
```

One way we can view this set of pixels is as a cloud of points in a three-dimensional color space. We will reshape the data `china` to `[n_samples, n_features]`, and rescale the colors so that they lie between (0 and 1):

```
data = china / 255.0 # use 0...1 scale
data = data.reshape(427 * 640, 3)
data.shape
```

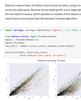
```
(273280, 3)
```

We can visualize these pixels in this color space, using a subset of 10,000 (to avoid the offscreen):

```
def plot_pixels(data, title, colors=None, N=10000):
    if colors is None:
        colors = data

    # choose a random subset
    rng = np.random.RandomGenerator(0)
    i = rng.permutation(data.shape[0])[:N]
    colors = colors[i]
    R, G, B = data[i].T

    fig, ax = plt.subplots(1, 2, figure=fig, )
    ax[0].scatter(R, G, color=colors, marker='.')
    ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

    ax[1].scatter(R, B, color=colors, marker='.')
    ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

    fig.suptitle(title, size=20);
```

```
%time plot_pixels(data, title='Input color space: 16 million possible colors')
```

```
Input color space: 16 million possible colors
CPU times: user 1.1 s, sys: 1.1 s, total: 2.2 s
Wall time: 1.1 s
```



Now let's reduce these 16 million colors to just 16 colors, using k-means clustering across the pixel space. Because we clustering pixels in a given color space, just like the digits data in Isomap, which operates on subsets of the data to compute the Isomap manifolds, using MiniBatchKMeans will be much faster than KMeans:

```
%time
from sklearn.cluster import MiniBatchKMeans
kmeans = MiniBatchKMeans(16)
kmeans.fit(data)
new_colors = kmeans.cluster_centers_[kmeans.predict(data)]
```

```
CPU times: user 1.2 s, sys: 1.1 s, total: 2.3 s
Wall time: 1.1 s
```

```
%time plot_pixels(data, title='Reduced color space: 16 colors',
                  colors=new_colors)
```

```
Reduced color space: 16 colors
CPU times: user 1.1 s, sys: 1.1 s, total: 2.2 s
Wall time: 1.1 s
```


The result is an image with only 16 colors, which next pixel in a region the color of the closest cluster center. This means that we are still in the image space rather than the pixel space shown on the effect of this.

```
china_recolored = new_colors.reshape(china.shape)
fig, ax = plt.subplots(1, 2, figure=fig, )
ax[0].imshow(china)
ax[0].set(xticks=[], yticks=[], title='Input image')

ax[1].imshow(china_recolored)
ax[1].set(xticks=[], yticks=[], title='16-color image');
```


Some detail is sacrificed for the good of data compression, but the overall feeling is still very recognizable. This image on the right achieves a compression factor of around 1 to 6 in this case. Different color clustering approaches of course, lead to different results. One interesting extension of the k-means approach would be to pre-cluster the image into patches of the image, and then do this for each component. This is perhaps the most sophisticated image compression algorithm I can think of off the top of my head.

← In Depth: Manifold Learning | Contents | In Depth: Gaussian Mixture Models →

Open in Colab