In [143…
```python
import numpy as np
import matplotlib.pyplot as plt
import random
import re
import pandas as pd
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.datasets import make_circles
from scipy.cluster.hierarchy import dendrogram
from sklearn.datasets import load_iris, load_sample_image
from sklearn.cluster import AgglomerativeClustering
from PIL import Image
import time
```

What is clustering

Clustering is a technique in machine learning and data analysis that involves grouping similar data points together into clusters or subgroups based on their similarity or distance from each other. Clustering is an unsupervised learning method, which means that it does not require labeled data or a pre-defined set of target classes to be trained on. Instead, clustering algorithms analyze the data to identify patterns and similarities within the data, and then group the data points into clusters based on these patterns.

The goal of clustering is to identify hidden structures or patterns in the data, such as groups of customers with similar purchasing habits, or clusters of genes with similar expression patterns. Clustering can be used for a variety of applications, such as image segmentation, customer segmentation, anomaly detection, and recommendation systems.

There are many clustering algorithms available, each with its own strengths and weaknesses. Some of the most popular clustering algorithms include k-means clustering, hierarchical clustering, DBSCAN, and spectral clustering. These algorithms differ in the way they measure similarity or distance between data points, the way they group data points into clusters, and the computational complexity of the algorithm. The choice of clustering algorithm depends on the nature of the data and the goals of the analysis.

Here we introduce some basic clustering algorithm, kmeans, kmeans ++ and hierarchical clustering

How the kmeans algorithm works

K-means is a clustering algorithm that partitions a dataset into k clusters, where k is a pre-defined number of clusters. The algorithm works by iteratively assigning each data point to the nearest centroid, and then updating the centroids to the mean of the data points in each cluster. The algorithm terminates when the assignment of data points to cluster no longer changes, or when a maximum number of iterations is reached.

The k-means algorithm works as follows:

- Initialize k centroids randomly from the data points.
- Assign each data point to the nearest centroid.
- Recalculate the centroid of each cluster by taking the mean of all the data points assigned to that cluster.
- Repeat steps 2-3 until the assignments of data points to cluster no longer change.

K-means clustering is a popular algorithm for its simplicity and speed. It is widely used in data mining, image processing, and other applications. However, it does have some limitations. One limitation is that the number of clusters k must be specified in advance, and the quality of the clustering can depend on the choice of k. Another limitation is that the algorithm can get stuck in local minima, where the final clustering may not be optimal. To overcome these limitations, variants of the k-means algorithm have been proposed, such as hierarchical k-means and k-means++, which aim to improve the initial centroid selection and convergence properties of the algorithm.

Below is an example of using kmeans clustering on a 1 dimension array called grade

In [105…

```
# set a grade array
grade = [92.65, 93.87, 74.06, 86.94, 92.26, 94.46, 92.94, 80.65, 92.86, 85.9
```

In [162…

```
def kmeans(X:np.ndarray, k:int, centroids=None, max_iter=30, tolerance=1e-2)
    # Randomly initialize centroids if not given
    if centroids is None:
        centroids = X[np.random.choice(X.shape[0], size=k, replace=False), :
        print("initial_centroids\n", centroids)
    for i in range(max_iter):
        # Assign each point to the nearest centroid
        distances = np.sqrt(((X - centroids[:,np.newaxis])**2).sum(axis=2))
        # get the labels of the cluster
        labels = np.argmin(distances, axis=0)

        # Compute new centroids as the mean of points
        new_centroids = np.array([X[labels == j].mean(axis=0) for j in range
        # Check convergence criterion
        if np.allclose(new_centroids, centroids, rtol=0, atol=tolerance):
            break

        centroids = new_centroids

    return centroids, labels
```

In [164…

```
k = 3  # initialize k centroids

grades = np.array(grade).reshape(-1,1)

centroids, labels  = kmeans(grades, k)

print("final centroids: ", centroids.reshape(1,-1))
print("labels for each x: ", labels)
for j in range(k):
    print("vector assignments: ", grades[labels==j].reshape(1,-1))
```

```
initial_centroids
 [[94.46]
 [93.87]
 [87.85]]
final centroids:  [[94.52       92.58833333 84.07428571]]
labels for each x:  [1 0 2 2 1 0 1 2 1 2 1 0 2 2 2 1]
vector assignments:  [[93.87 94.46 95.23]]
vector assignments:  [[92.65 92.26 92.94 92.86 91.79 93.03]]
vector assignments:  [[74.06 86.94 80.65 85.94 85.37 87.85 87.71]]
```

We can see from the above output, 94.46, 93.87 and 87.85 are three centroids that we initialize. Then we repeatedly assign each data point to the nearest centroid, recalculating the centroid of each cluster by taking the mean of all the data points assigned to that cluster. We repeat the process until the data points to cluster no longer change, here we have our final centroids 94.52, 92.58 and 84.07. Here we have three cluster so there are three labels for each x, 0,1 and 2. For each cluster, we assign that labels to the data point, here we have three vector assignments.

Next we update the basic k means with some additional function

K-means and k-means++ are both clustering algorithms that are used to partition a dataset into k clusters. However, k-means++ is an improvement over the standard k-means algorithm in terms of the initial selection of the cluster centroids.

In the standard k-means algorithm, the initial centroids are chosen randomly from the data points. This approach can lead to poor clustering results if the initial centroids are chosen poorly, such as if they are initialized close to each other or far away from the true cluster centers.

In contrast, k-means++ is a modification of the k-means algorithm that uses a more sophisticated initialization method for the cluster centroids. The initialization method consists of the following steps:

The first centroid is chosen randomly from the data points. For each data point, the distance to the nearest centroid is computed. The next centroid is chosen from the data points with a probability proportional to the squared distance to the nearest centroid. Steps 2-3 are repeated until k centroids have been chosen.

In [168…

```python
def kmeans(X:np.ndarray, k:int, centroids=None, max_iter=30, tolerance=1e-2)
    # Initialize centroids using k-means++ if not given
    if centroids == "kmeans++":
        centroids = [X[np.random.randint(len(X))]]
        # Choose remaining centroids using K-means++ algorithm
        for _ in range(1, k):
            # Compute distances to the nearest centroid for each point
            distances = np.array([min([np.linalg.norm(x-c)**2 for c in centr
            # Choose new centroid with probability proportional to distance
            new_centroid = X[np.random.choice(len(X), p=distances/sum(distan
            centroids.append(new_centroid)
        centroids = np.array(centroids)

    # if none use normal k-means
    elif centroids is None:
        centroids = X[np.random.choice(X.shape[0], size=k, replace=False), :

    for i in range(max_iter):
        # Assign each point to the nearest centroid
        distances = np.sqrt(((X - centroids[:,np.newaxis])**2).sum(axis=2))
        labels = np.argmin(distances, axis=0)

        # Compute new centroids as the mean of points
        new_centroids = np.array([X[labels == j].mean(axis=0) for j in range

        # Check convergence criterion
        if np.allclose(new_centroids, centroids, atol=tolerance):
            break

        centroids = new_centroids

    return centroids, labels
```

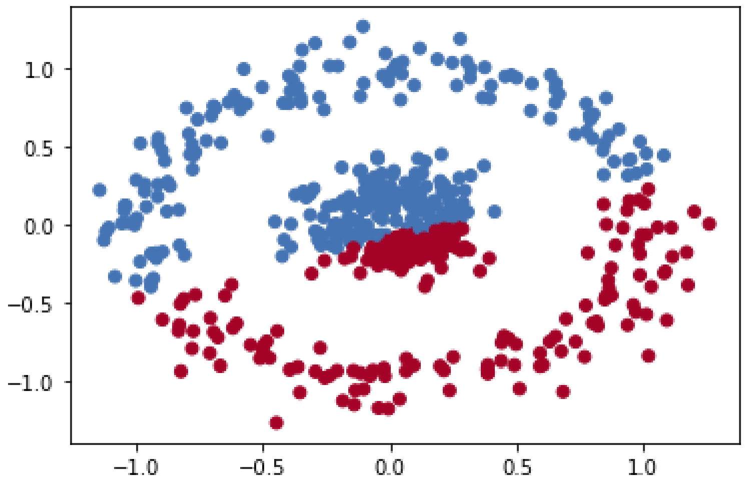
The k-means++ algorithm ensures that the initial centroids are well-spaced and avoids the problem of poor initial centroid selection. This leads to more accurate and stable clustering results, especially when the number of clusters k is large or the data has a complex structure.

Overall, k-means++ is generally preferred over the standard k-means algorithm for clustering tasks, as it often produces better results with faster convergence. However, it may be computationally more expensive due to the additional step of choosing the initial centroids.

Let's try to visualize kmeans ++

In [173…

```
X, _ = make_circles(n_samples=500, noise=0.1, factor=.2)
centroids, labels = kmeans(X, 2)
colors = np.array(['#4574B4','#A40227'])
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()
```



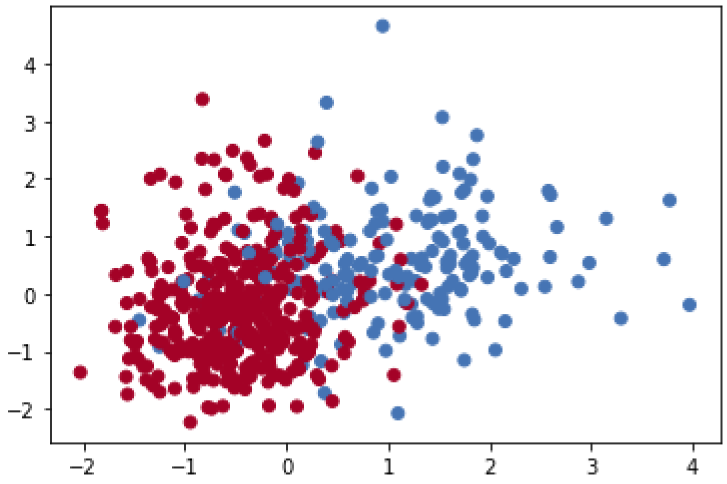
Here we can see the data have been clustered into two different colors, and we can see it does not work well with circular data, as the data at the center and the outside ring should be in two different cluster. We perform some sample on some data sets and visualize the results. As we discussed above, kmeans takes some time to run, so we time and compare experiment with different clustering number

sample run on the cancer date sets

In [183…

```python
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
cancer = load_breast_cancer()
X = cancer.data
y = cancer.target # y = 0 means cancer, y = 1 benign

# scale so that tbe distances mean the same in all dimension
sc = StandardScaler()
X = sc.fit_transform(X)

start_time = time.time()

centroids, labels = kmeans(X, k=2, centroids="kmeans++", tolerance=0.1)
likely_confusion_matrix = confusion_matrix(y, labels)
m = pd.DataFrame(likely_confusion_matrix,columns=['pred_F','pred_T'], index=
print(m)
print('clustering accuracy', accuracy_score(y,labels))
plt.scatter(X[:,0], X[:,1], c=colors[labels])
plt.show()

end_time = time.time()
elapsed_time = end_time - start_time
print(f"Elapsed time: {elapsed_time:.2f} seconds")
```

```
   pred_F  pred_T
F     169      43
T       7     350
clustering accuracy 0.9121265377855887
```



```
Elapsed time: 0.07 seconds
```

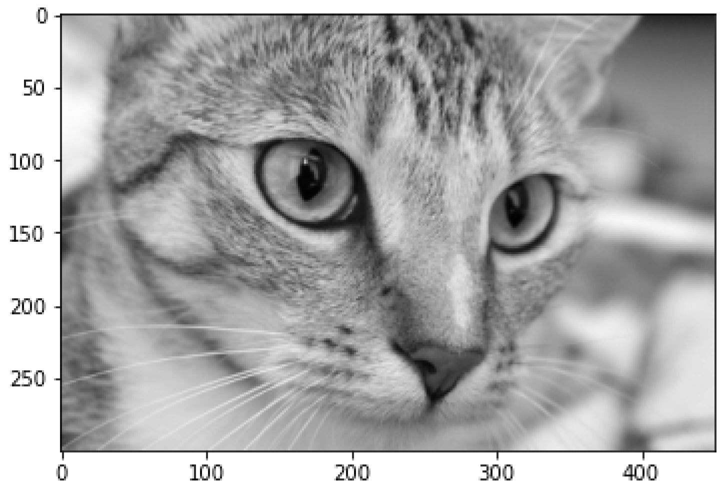
We used kmeans++ to cluster the data points in a cancer data, where y = 0 means cancer, y = 1 mean benign. The red dots From the picture above, we can see the visualization of the sample data sets with two clusters. The confusion matrix shows the numbers of true positive, false positive, true native, false negative. The clustering accuracy is about 0.91 and the time to run the kmeans algorithm is 0.07 seconds

Application to image compression

Now we apply the k-means++ algorithm to compress some image. First we start with a grey-scale image of a cat.

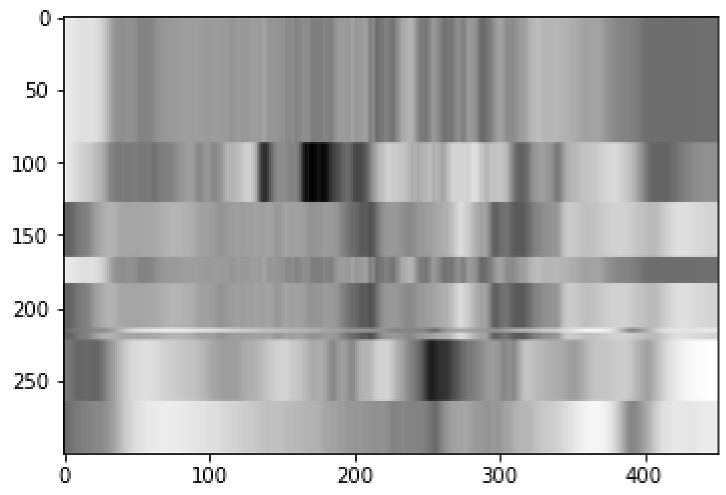
In [111…
```
from skimage import data
from skimage.color import rgb2gray
from skimage import img_as_ubyte,img_as_float
gray_images = {
        "cat":rgb2gray(img_as_float(data.chelsea())),
        "im1":rgb2gray(img_as_float(data.astronaut())),
        "im2":rgb2gray(img_as_float(data.stereo_motorcycle()[0]))
}
```

In [112…
```
plt.imshow(gray_images["cat"], cmap='gray')
plt.show()
```



Above is the original picture before compressing with k-means.

In [134…
```
start_time = time.time()
arr = np.array(gray_images["cat"])
centroids, labels = kmeans(arr, 5, "kmeans++")
compressed = centroids[labels] #reassign all points
plt.imshow(compressed, cmap='gray')
plt.show()


end_time = time.time()
# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Elapsed time: {elapsed_time:.2f} seconds")
```

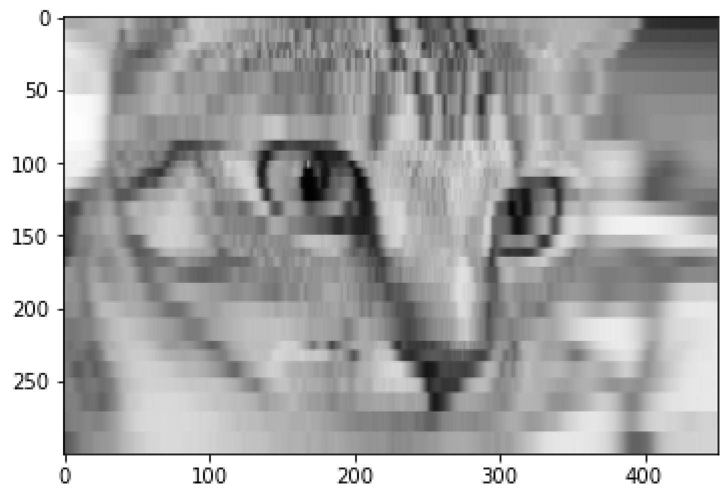


```
Elapsed time: 0.08 seconds
```

After compressing the image with 5 clusters, we cannot recognize the pattern anymore!

In [135…

```python
start_time = time.time()

arr = np.array(gray_images["cat"])
centroids, labels = kmeans(arr, 30, "kmeans++")
compressed = centroids[labels]
plt.imshow(compressed, cmap='gray')
plt.show()

end_time = time.time()
# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Elapsed time: {elapsed_time:.2f} seconds")
```



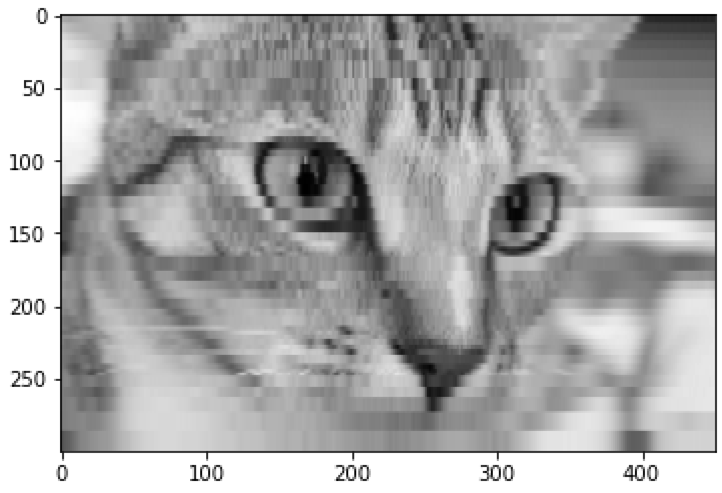
```
Elapsed time: 0.74 seconds
```

We increase the number of cluster to 30, now we can see the cat in the image, but it takes a longer time to process

In [136…

```python
start_time = time.time()

arr = np.array(gray_images["cat"])
centroids, labels = kmeans(arr, 50, "kmeans++")
compressed = centroids[labels]
plt.imshow(compressed, cmap='gray')
plt.show()
```

```
end_time = time.time()
# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Elapsed time: {elapsed_time:.2f} seconds")
```



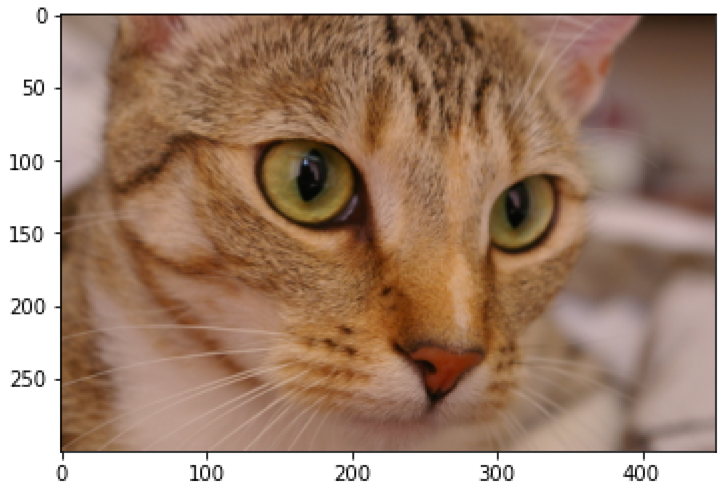
```
Elapsed time: 1.73 seconds
```

By increasing the clusters to 50, we can see a better picture of the cat, but it takes 1.73 secs to run

Next, let's take a look at the color version of the image and see how k-means compress the image.

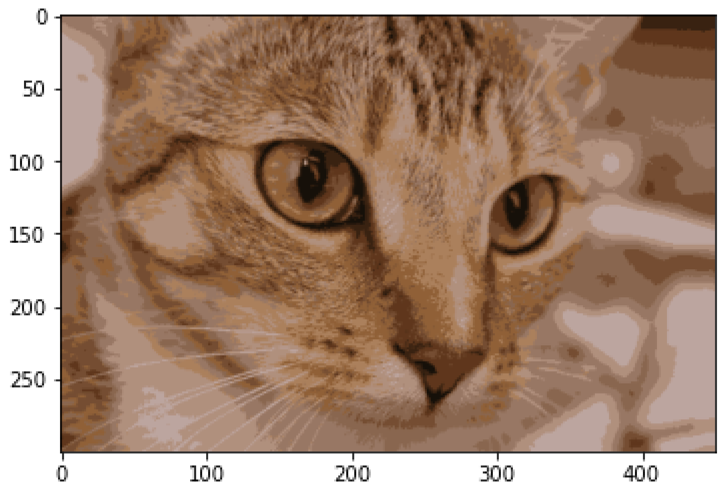
In [116…
```
color_images = {
    "cat":img_as_float(data.chelsea()),
    "im1":img_as_float(data.astronaut()),
    "im2":img_as_float(data.stereo_motorcycle()[0])
}
```

In [125…
```
plt.imshow(color_images['cat'])
plt.show()
```



In [137…
```
start_time = time.time()

arr = np.array(color_images['cat'])
# reshape the array to a 2D matrix of pixels and channels
pixels = arr.reshape((-1,3))
centroids, labels = kmeans(pixels, 10, "kmeans++")
```

```
compressed = centroids[labels] #reassign all points
# convert the shape of the image back to the original image
compressed_arr = compressed.reshape(arr.shape)
plt.imshow(compressed_arr)
plt.show()

end_time = time.time()
# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Elapsed time: {elapsed_time:.2f} seconds")
```

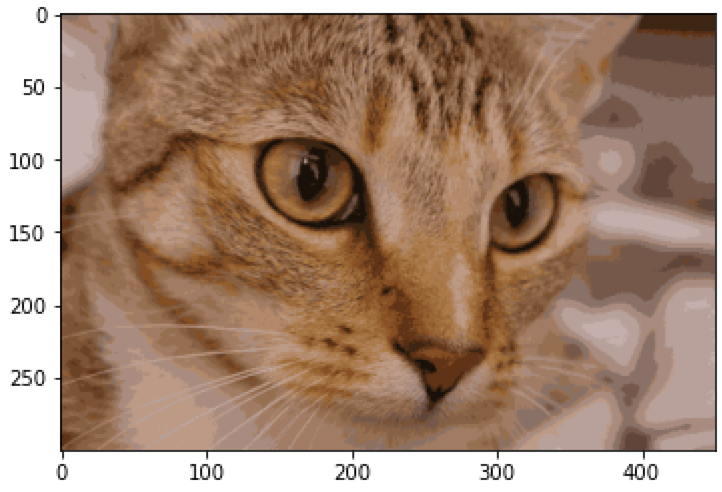


```
Elapsed time: 23.74 seconds
```

It takes a much longer time to run because we need to rechape the 3D matrix to 2D, then run the kmeans and convert it back to the original image

In [138…

```
start_time = time.time()

arr = np.array(color_images['cat'])
# reshape the array to a 2D matix of pixels and channels
pixels = arr.reshape((-1,3))
centroids, labels = kmeans(pixels, 20, "kmeans++")
compressed = centroids[labels] #reassign all points
# convert the shape of the image back to the original image
compressed_arr = compressed.reshape(arr.shape)
plt.imshow(compressed_arr)
plt.show()

end_time = time.time()
# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"Elapsed time: {elapsed_time:.2f} seconds")
```



```
Elapsed time: 98.11 seconds
```

Significantly longer running time with just 20 clusters

so why does kmeans run so slow?

The time complexity of k-means algorithm is O(I *n* k * d), where I is the number of iterations required for convergence, n is the number of data points, k is the number of clusters, and d is the number of dimensions in the data.

In each iteration, the algorithm assigns each data point to its nearest cluster center, and then updates the position of each cluster center based on the mean of its assigned points. The algorithm continues iterating until the cluster assignments no longer change or a maximum number of iterations is reached.

The time complexity of the k-means algorithm is dominated by the step of assigning each data point to its nearest cluster center, which requires computing the distance between the data point and each cluster center. This step takes O(n *k* d) time. The step of updating the position of each cluster center takes O(n *k* d) time as well. The number of iterations required for convergence varies depending on the initial cluster assignments and the convergence criteria, but it typically ranges from a few to several hundred.

Overall, the time complexity of k-means algorithm can be relatively high for large datasets or high-dimensional data, but it can be improved with techniques such as parallelization, initialization strategies, and early stopping criteria.

Agglomerative Clustering

Instead of picking number of cluster in k-means, agglomerative clustering is a type hierarchical clustering that works from the bottom-up. All observations in the data sets X start in their own cluter, and them the clusters are joined together, two cluster at a time until there is only one cluster left.

In [140…
```
def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram
    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack(
        [model.children_, model.distances_, counts]
    ).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)
```

In [144…

```
# Load the iris dataset
iris = load_iris()
X, y = iris.data[0:20,], iris.target
```

In [187…

```
X, y
```

```
Out[187]: (array([[ 1.09706398, -2.07333501,  1.26993369, ...,  2.29607613,
                     2.75062224,  1.93701461],
                   [ 1.82982061, -0.35363241,  1.68595471, ...,  1.0870843 ,
                    -0.24388967,  0.28118999],
                   [ 1.57988811,  0.45618695,  1.56650313, ...,  1.95500035,
                     1.152255  ,  0.20139121],
                   ...,
                   [ 0.70228425,  2.0455738 ,  0.67267578, ...,  0.41406869,
                    -1.10454895, -0.31840916],
                   [ 1.83834103,  2.33645719,  1.98252415, ...,  2.28998549,
                     1.91908301,  2.21963528],
                   [-1.80840125,  1.22179204, -1.81438851, ..., -1.74506282,
                    -0.04813821, -0.75120669]]),
            array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                   0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0,
                   1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
                   1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1,
                   1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0,
                   0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
                   1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1,
                   1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0,
                   0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0,
                   1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1,
                   1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                   0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1,
                   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1,
                   1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0,
                   0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0,
                   0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0,
                   1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1,
                   1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0,
                   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1,
                   1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
                   1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
                   1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1,
                   1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
                   1, 1, 1, 1, 1, 0, 1, 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, 0, 0, 0, 0, 0, 0, 1]))
```

Here we loaded the sample data from iris. We start with 20 clusters, with indexes 0-19, for each new cluster that is formed by merging two clusters, we add an index. 'model.children_' tells you which cluster indexes are merged together. If there is a number that is larger than 19, then these are new clusters that were formed by merging other clusters.

```
In [145…  # setting distance_threshold=0 ensures we compute the full tree.
          model = AgglomerativeClustering(distance_threshold=0, n_clusters=None)

          model = model.fit(X)
```

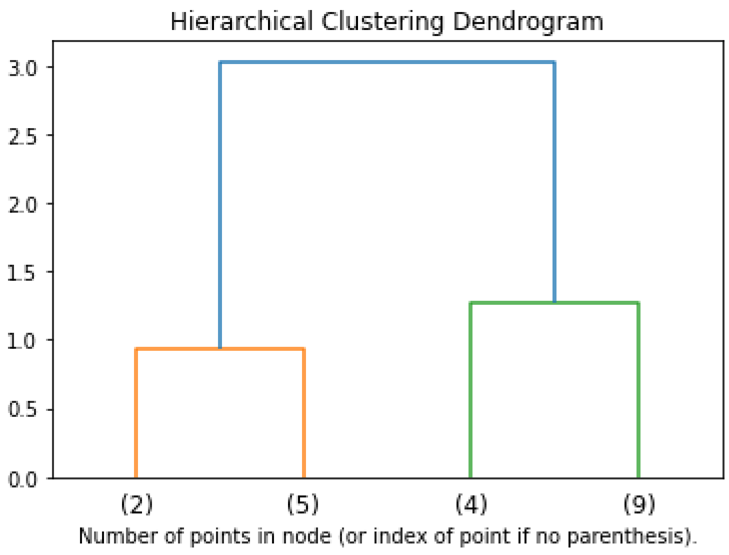
```
In [146…  model.children_
```

```
Out[146]: array([[ 0, 17],
                 [ 1, 12],
                 [ 4, 20],
                 [ 9, 21],
                 [ 7, 22],
                 [ 2,  3],
                 [ 6, 11],
                 [ 5, 18],
                 [10, 19],
                 [ 8, 13],
                 [16, 28],
                 [25, 26],
                 [14, 15],
                 [27, 30],
                 [23, 31],
                 [29, 34],
                 [32, 33],
                 [24, 35],
                 [36, 37]])
```

Here we can see the initial 20 cluster from the data

The process of agglomerative clustering starts by calculating the distances between all pairs of data points. The distance between two clusters is then defined as the minimum distance between any two points in the two clusters. At each step, the two closest clusters are merged into a single cluster, and the distances between this new cluster and the remaining clusters are updated.

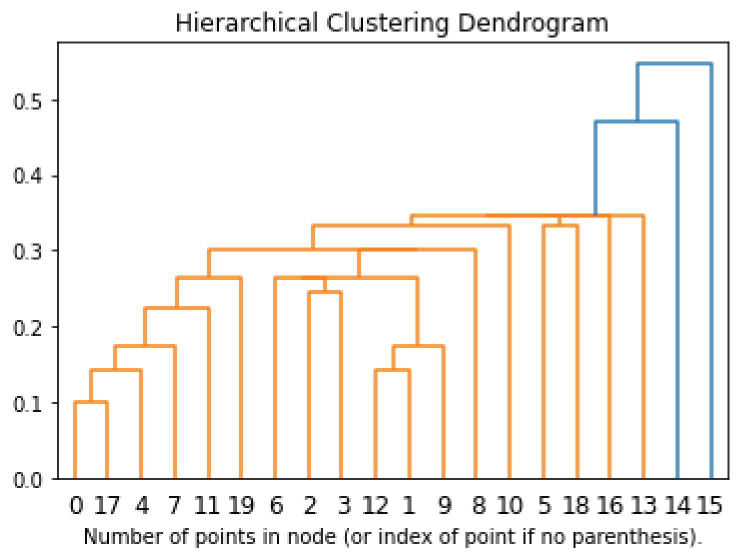
```
In [147…  plt.title("Hierarchical Clustering Dendrogram")
          plot_dendrogram(model, truncate_mode="level", p=1)
          plt.xlabel("Number of points in node (or index of point if no parenthesis)."
          plt.show()
```



The x-axis show the number of points in node, cluster with the smallest distance between any two points in the two clusters are joined, so '(2)' and '(5)' are joined first, then '(4)' and '(9)', then these two big cluster are joined together.

Let's see what happens when we change the linkage to 'singe', which means we join cluster based on which two clusters have the smallest minimum distance between any pair of points in the two clusters.

```
In [148…  model = AgglomerativeClustering(distance_threshold=0, n_clusters=None, linka

          model = model.fit(X)
          plt.title("Hierarchical Clustering Dendrogram")
          plot_dendrogram(model, truncate_mode="level", p=100)
          plt.xlabel("Number of points in node (or index of point if no parenthesis)."
          plt.show()
```



In the above example, we used the default linkage ward linkage and single linkage to calculate the distance between data points and clusters.

Here is the summary of different linkage methods we can use

Single linkage: This method computes the distance between the closest pair of points in each cluster and uses that as the distance between clusters. It tends to produce long, thin clusters and is sensitive to noise and outliers.

Complete linkage: This method computes the distance between the farthest pair of points in each cluster and uses that as the distance between clusters. It tends to produce compact, spherical clusters and is less sensitive to noise and outliers.

Average linkage: This method computes the average distance between all pairs of points in each cluster and uses that as the distance between clusters. It can produce clusters of varying shapes and sizes and is less sensitive to noise and outliers than single linkage.

Ward linkage: This method minimizes the variance of the distances between points within each cluster when merging them. It tends to produce clusters of similar sizes and is sensitive to the shape and variance of the data.

The choice of linkage method depends on the specific characteristics of the data and the desired clustering solution. For example, complete linkage may be preferred for datasets with compact, spherical clusters, while single linkage may be preferred for datasets with long, thin clusters or hierarchical structures. Average linkage is a good general-purpose linkage method that can work well for many types of datasets.

How about time complexity of agglomerative clustering?

The time complexity of agglomerative clustering depends on the number of data points, the distance metric used, and the linkage criterion used to merge clusters.

For a dataset with n data points, the time complexity of agglomerative clustering can be computed as $O(n^3)$ for the complete linkage criterion and $O(n^2)$ for single linkage criterion. The average and ward linkage criteria fall between these two extremes, with a time complexity of $O(n^2 logn)$ in practice.

The reason for the high time complexity is that, in each iteration of agglomerative clustering, all pairwise distances between clusters need to be computed. This requires computing the distance between each pair of points in the merged clusters, resulting in a total of O(n^2) distance computations per iteration. The algorithm then performs $O(n)$ iterations to cluster all the points.

Therefore, agglomerative clustering can be computationally expensive for large datasets, making it less practical for large-scale applications. However, there are optimization techniques such as hierarchical tree reduction and fast nearest neighbor search that can reduce the time complexity and improve the scalability of agglomerative clustering algorithms.

What are the use cases of agglomerative clustering?

Agglomerative clustering is a widely used technique in various fields for clustering and exploring data. Here are some common use cases for agglomerative clustering:

Image Segmentation: Agglomerative clustering can be used for segmenting images into regions or objects with similar color, texture or shape characteristics.

Biological Data Analysis: Agglomerative clustering is used to analyze genomic data, such as gene expression data, to identify clusters of genes with similar expression patterns that may indicate common biological processes.

Marketing Segmentation: Agglomerative clustering is used to segment customers into groups based on demographic, behavioral, and transactional data, allowing businesses to better understand their customers and target them more effectively.

Document Clustering: Agglomerative clustering can be used to cluster text documents based on their content, allowing for more efficient retrieval and classification of documents.

Anomaly Detection: Agglomerative clustering can be used to detect anomalies or outliers in datasets by identifying clusters that deviate significantly from the norm.

Overall, agglomerative clustering is a powerful technique that can be used for a wide range of clustering applications, providing insights into complex data structures and uncovering hidden patterns in the data.

K-means vs agglomerative clustering

In conclusion, K-means and agglomerative clustering are both popular unsupervised machine learning algorithms used for clustering. Here are some of the key differences between these two algorithms:

Initialization: In k-means, the algorithm starts by randomly initializing k cluster centroids and then iteratively updates them to minimize the sum of squared distances between data points and their nearest centroids. In contrast, agglomerative clustering starts with each data point as a separate cluster and then iteratively merges the closest pairs of clusters until a stopping criterion is met.

Number of clusters: In k-means, the number of clusters k is specified as a hyperparameter and must be known in advance. In contrast, agglomerative clustering does not require the number of clusters to be specified in advance and can automatically determine the number of clusters based on the structure of the data.

Cluster shape and size: K-means tends to produce clusters that are spherical and of similar sizes, while agglomerative clustering can produce clusters of varying shapes and sizes.

Time complexity: The time complexity of k-means is generally lower than that of agglomerative clustering, particularly for large datasets. K-means has a time complexity of $O(nki)$, where n is the number of data points, k is the number of clusters, and i is the number of iterations. Agglomerative clustering has a time complexity of $O(n^3)$ for the "complete" and "average" linkage methods, and $O(n^2 logn)$ for the "ward" linkage method.

In general, k-means may be preferred for datasets with well-defined spherical clusters and when the number of clusters is known in advance, while agglomerative clustering may be preferred for datasets with irregular shapes and when the number of clusters is not known in advance. However, the choice of algorithm depends on the specific characteristics of the data and the desired clustering solution.