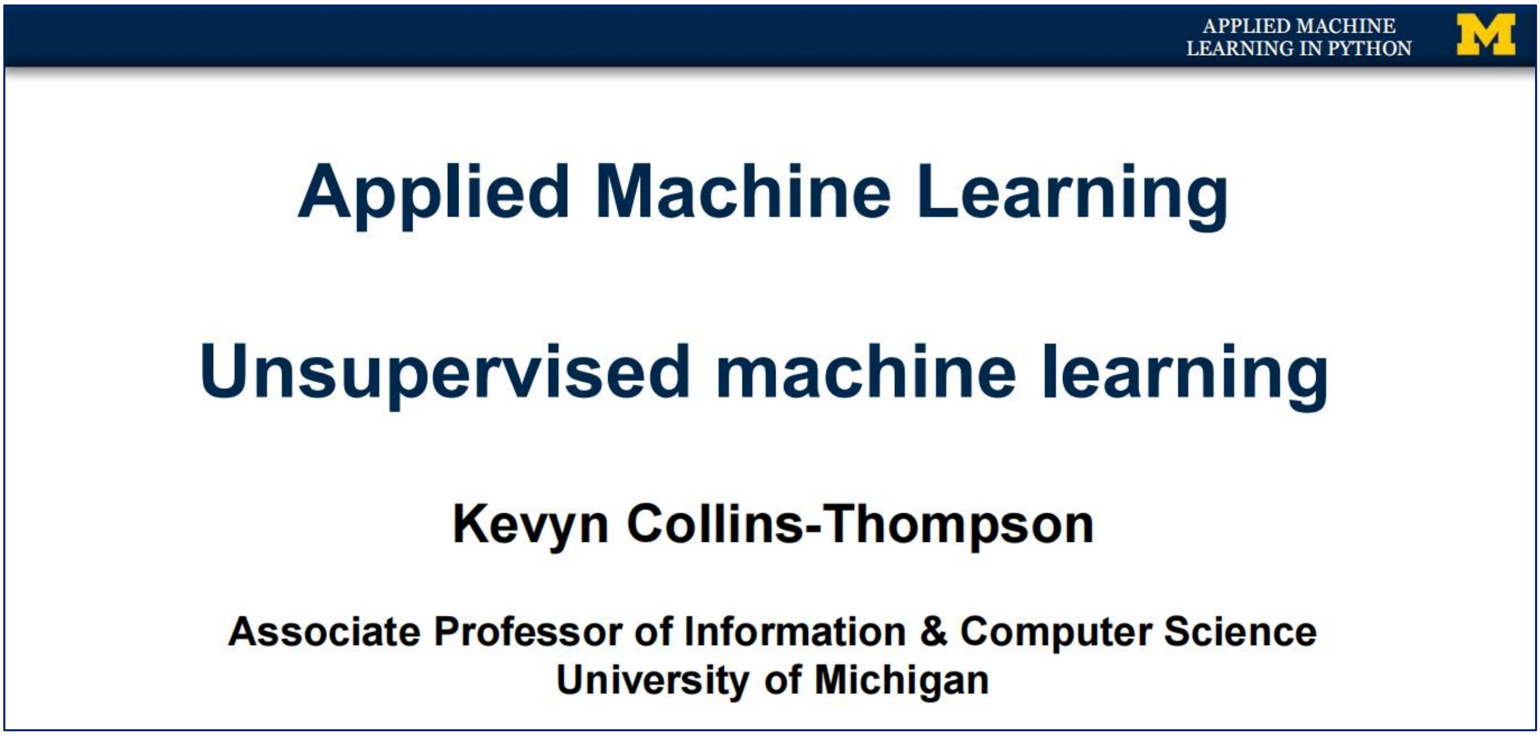
Module 5 - Unsupervised Machine Learning （optional content）



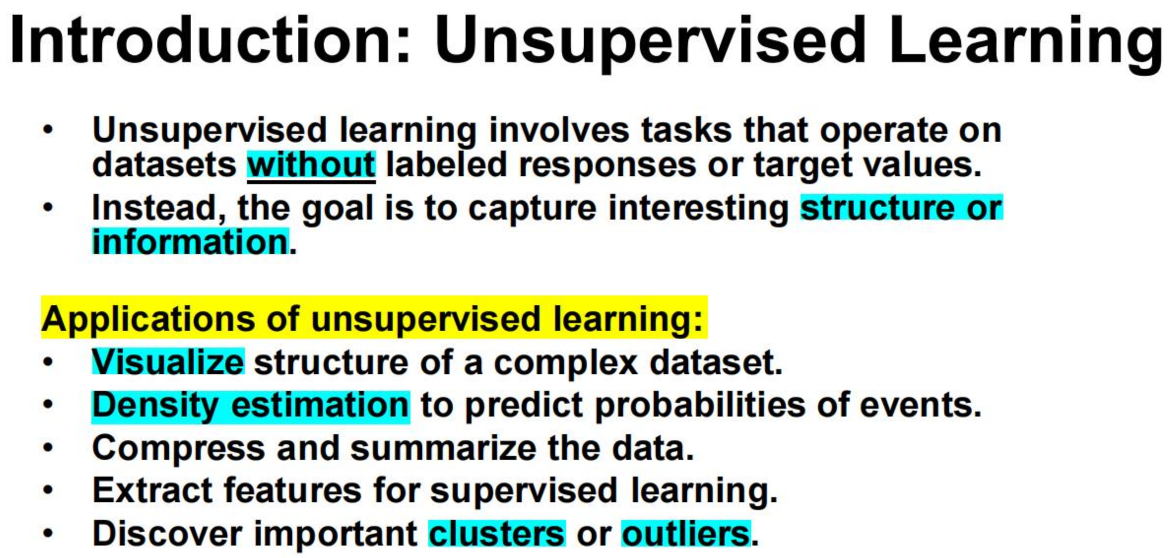
APPLIED MACHINE LEARNING IN PYTHON

Applied Machine Learning

Unsupervised machine learning

Kevyn Collins-Thompson

Associate Professor of Information & Computer Science
University of Michigan



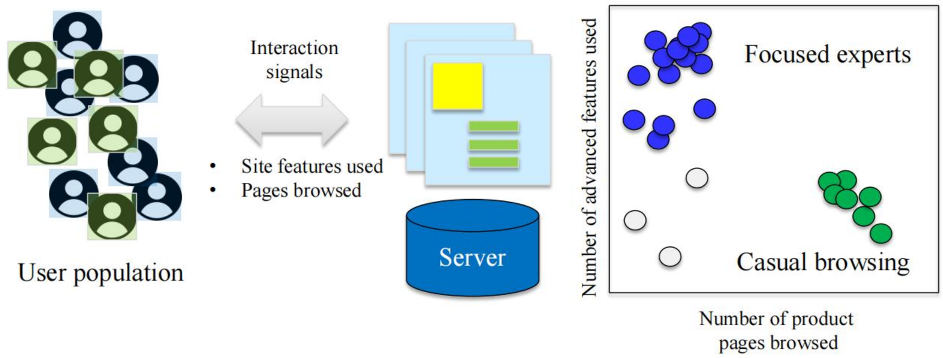
Introduction: Unsupervised Learning

- Unsupervised learning involves tasks that operate on datasets without labeled responses or target values.
- Instead, the goal is to capture interesting structure or information.

Applications of unsupervised learning:

- Visualize structure of a complex dataset.
- Density estimation to predict probabilities of events.
- Compress and summarize the data.
- Extract features for supervised learning.
- Discover important clusters or outliers.

Web Clustering Example

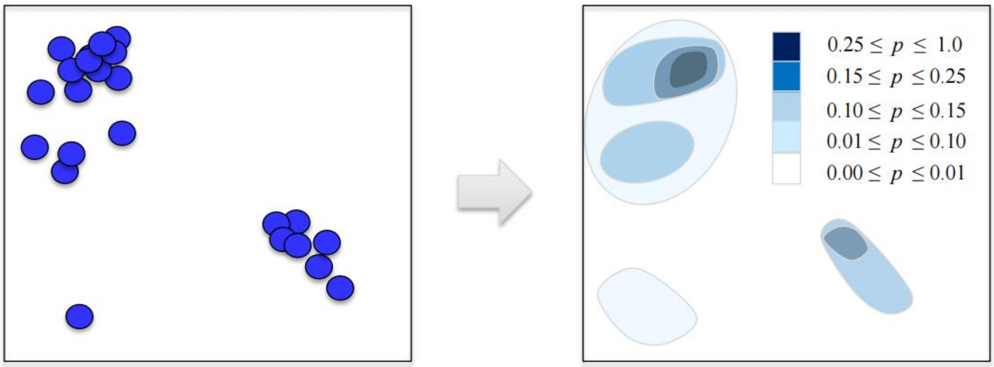


总结：

Two major types of unsupervised learning methods

- **Transformations**
  - *Processes that extract or compute information*
- **Clustering**
  - *Find groups in the data*
  - *Assign every point in the dataset to one of the groups*

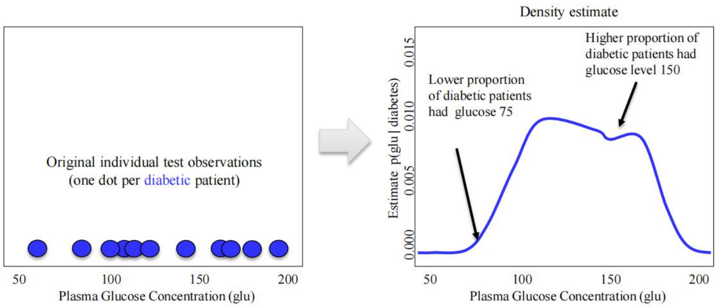
Transformations: Density Estimation



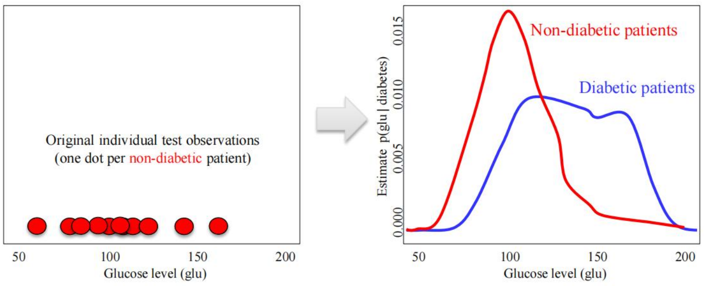
Individual measurements

Density estimate
(Estimated probability *p* of observing a measurement at a given location)

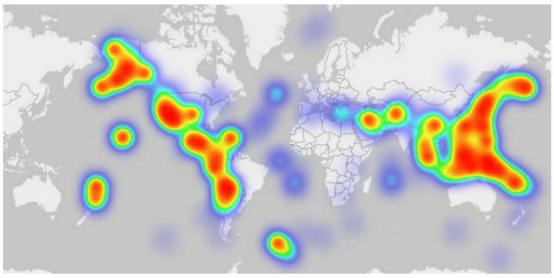
Density Estimation Example



Density Estimation Example



Kernel Density Example



Recent global earthquake activity (U.S. Geological Survey data)

Source: http://www.digital-geography.com/csv-heatmap-leaflet/

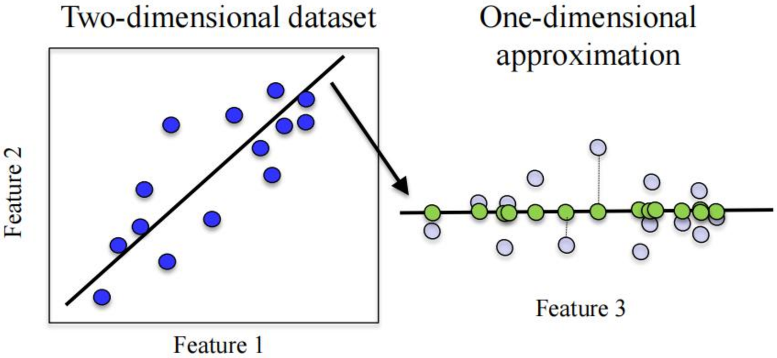
In Scikit-Learn, you can use the **kernel density class** in the sklearn.neighbors module to perform one widely used form of density estimation called kernel density estimation.

Kernel density's especially popular for use in creating heat maps with geospatial data like this one.

总结：

Dimensionality Reduction

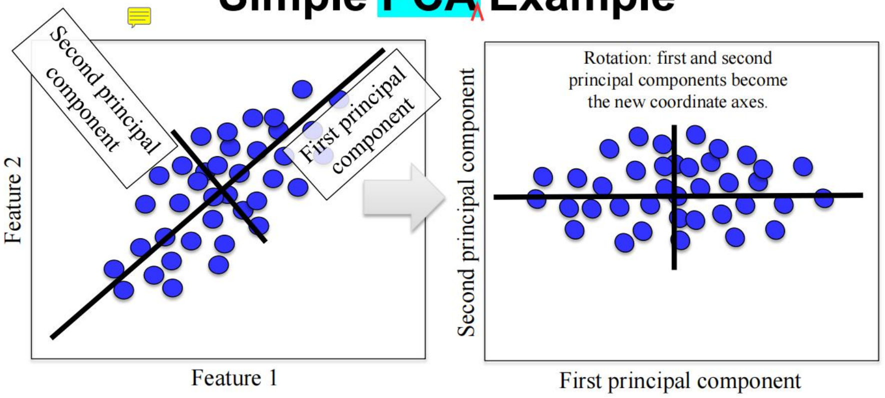
- Finds an approximate version of your dataset using fewer features
- Used for exploring and visualizing a dataset to understand grouping or relationships
- Often visualized using a 2-dimensional scatterplot
- Also used for compression, finding features for supervised learning



The one-dimensional approximation is obtained by projecting the original points onto the diagonal line and using their position on that line as the new single feature.

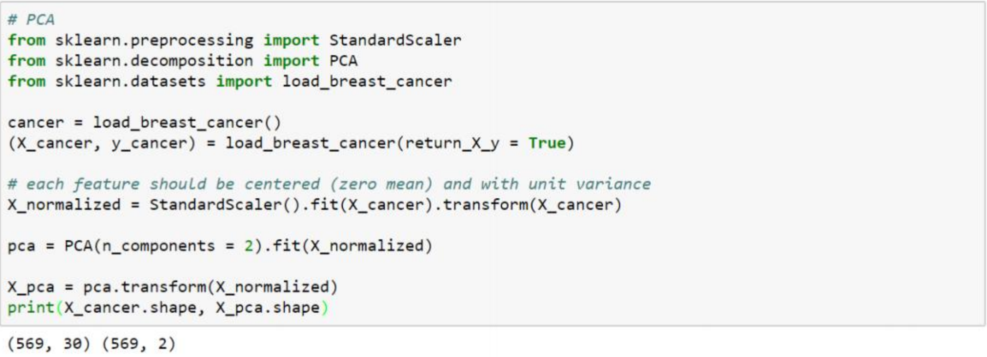
As the name suggests, this kind of transform takes your original dataset that might contain say, 200 features and finds an approximate version of dataset that uses, say, only 10 dimensions. One very common need for dimensionality reduction arises when first exploring a dataset, to understand how the samples may be grouped or related to each other by visualizing it using a two-dimensional scatterplot.

Simple PCA Example


Second principal component
First principal component
Feature 2
Feature 1
Rotation: first and second principal components become the new coordinate axes.
Second principal component
First principal component

In two dimensions, there's only one possible such direction at right angles of the first principal component, but for higher dimensions, there would be infinitely many. With more than two dimensions, the process of finding successive principal components at right angles to the previous ones would continue until the desired number of principal components is reached.

Dimensionality Reduction with PCA in scikit-learn

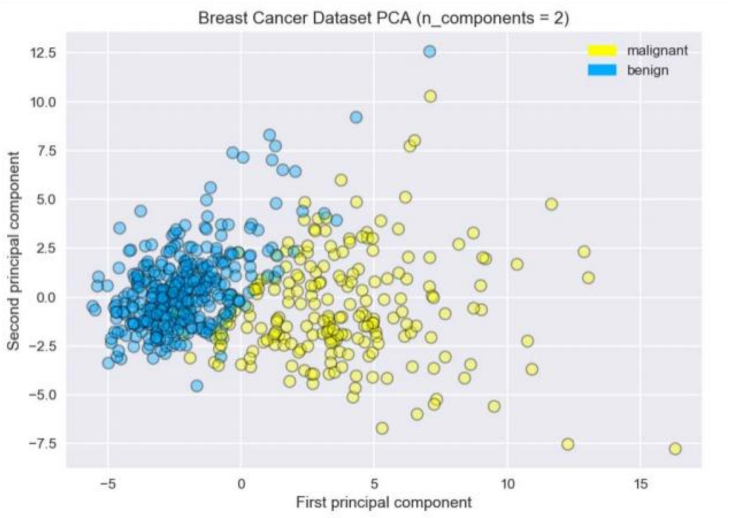


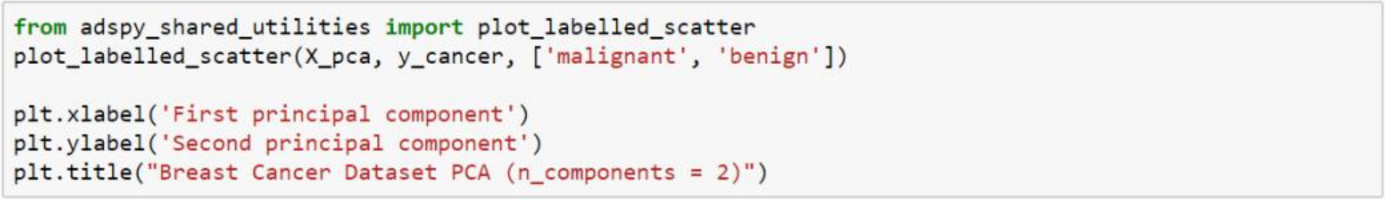
```
# PCA
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
(X_cancer, y_cancer) = load_breast_cancer(return_X_y = True)

# each feature should be centered (zero mean) and with unit variance
X_normalized = StandardScaler().fit(X_cancer).transform(X_cancer)

pca = PCA(n_components = 2).fit(X_normalized)

X_pca = pca.transform(X_normalized)
print(X_cancer.shape, X_pca.shape)
```

```
(569, 30) (569, 2)
```

Notice here since we're not doing supervised learning in evaluating a model against a test set, we don't have to split our dataset into training and test sets.

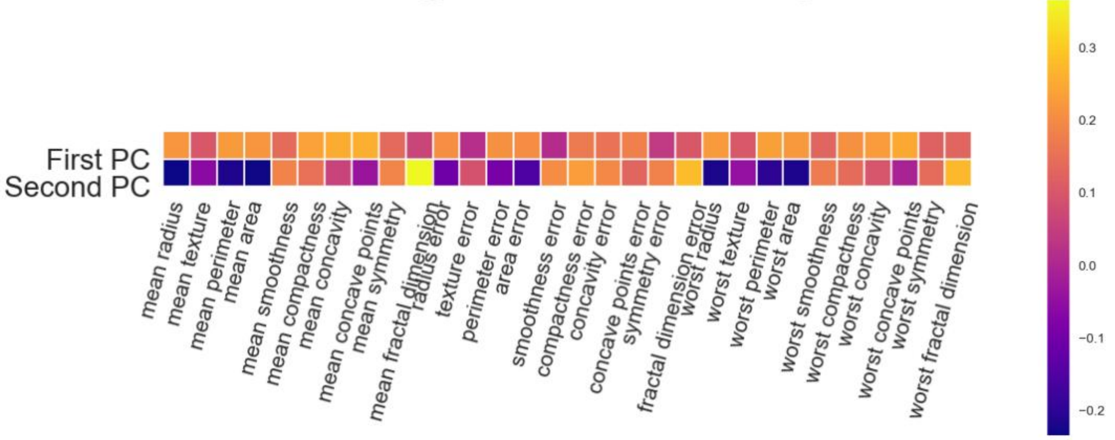
总结：





```
from adspy_shared_utilities import plot_labelled_scatter
plot_labelled_scatter(X_pca, y_cancer, ['malignant', 'benign'])

plt.xlabel('First principal component')
plt.ylabel('Second principal component')
plt.title("Breast Cancer Dataset PCA (n_components = 2)")
```

Visualizing PCA Components

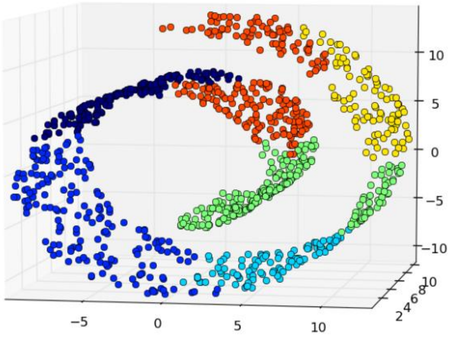


We can create a heat map that visualizes the first two principal components of the breast cancer dataset to get an idea of what feature groupings each component is associated with. Note that we can get the arrays representing the two principal component axes that define the PCA space using the **PCA.components_attribute** that's filled in after the PCA fit method is used on the data.

We can see that the first principle component is all positive, showing a general correlation between all 30 features. In other words, they tend to vary up and down together.
The second principle component has a mixture of positive and negative signs; but in particular, we can see a cluster of negatively signed features that co-vary together and in the opposite direction of the remaining features.
Looking at the names, it makes sense the subset wold co-vary together. We see the pair mean texture and worst texture and the pair mean radius and worst radius varying together and so on.

总结：

The ”Swiss Roll" Dataset

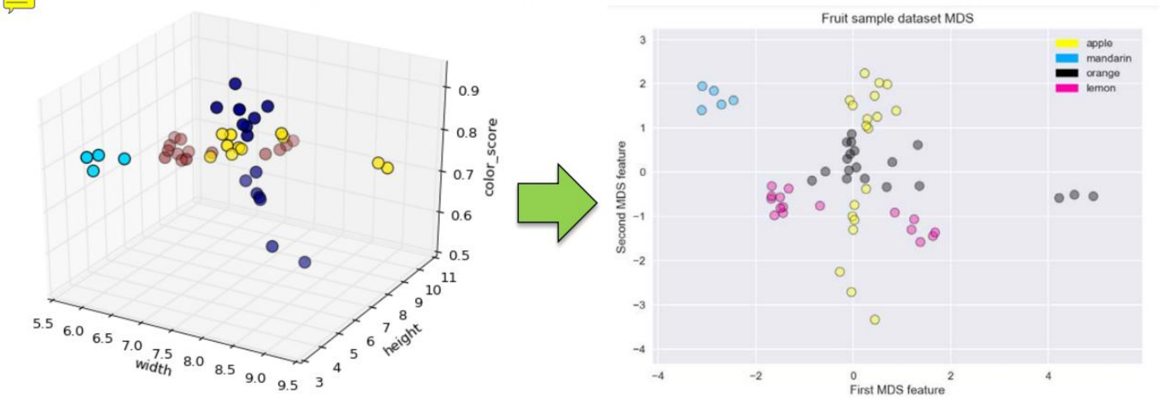


```
sklearn.datasets.make_swiss_roll(n_samples=1500, noise=0.05)
See: http://scikit-learn.org/stable/modules/clustering.html#hierarchical-clustering
```

PCA gives a good initial tool for exploring a dataset, but may not be able to find more subtle groupings that produce better visualizations for more complex datasets.

There is a family of unsupervised algorithms called **Manifold Learning Algorithms** that are very good at finding low dimensional structure in high dimensional data and are very useful for visualizations.

Multidimensional scaling (MDS) attempts to find a distance-preserving low-dimensional projection

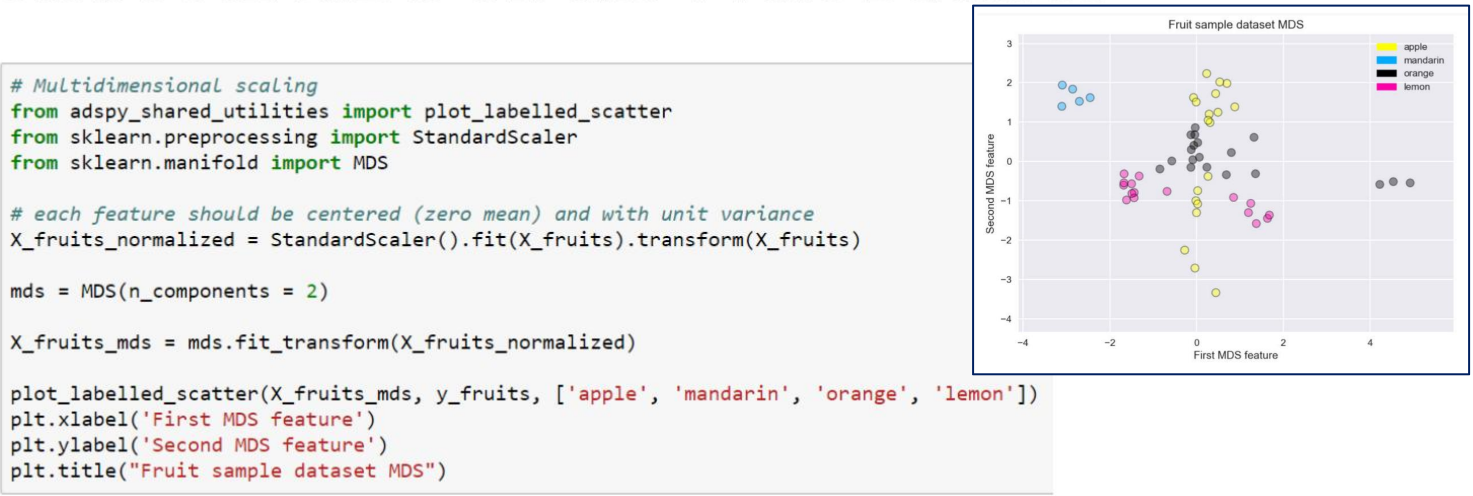


High-dimensional dataset

Two-dimensional MDS projection

One widely used manifold learning method is called **multi-dimensional scaling**, or **MDS**. There are many flavors of MDS, but they all have the same general goal; to visualize a high dimensional dataset and project it onto a lower dimensional space - in most cases, a two-dimensional page - in a way that preserves information about how the points in the original data space are close to each other.

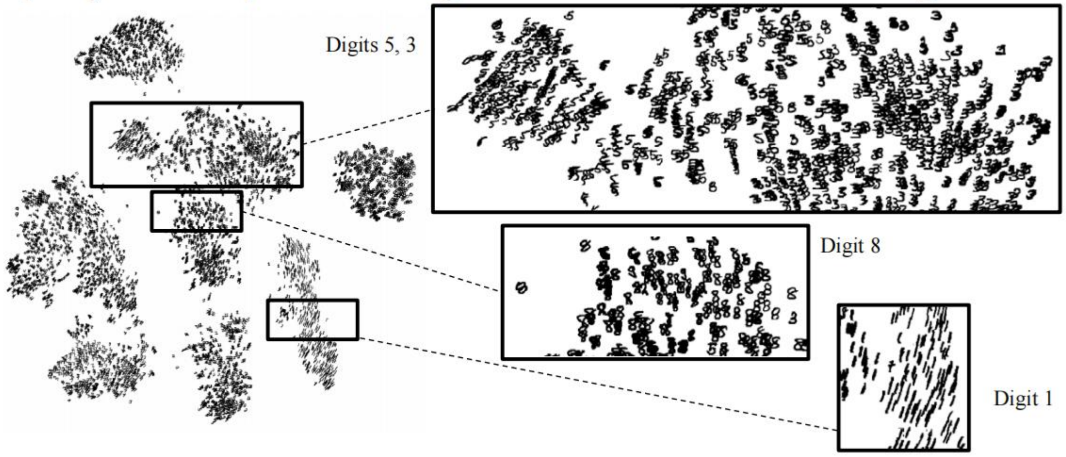
Notebook: MDS on the Fruit Dataset



```
# Multidimensional scaling
from adspy_shared_utilities import plot_labelled_scatter
from sklearn.preprocessing import StandardScaler
from sklearn.manifold import MDS

# each feature should be centered (zero mean) and with unit variance
X_fruits_normalized = StandardScaler().fit(X_fruits).transform(X_fruits)

mds = MDS(n_components = 2)

X_fruits_mds = mds.fit_transform(X_fruits_normalized)

plot_labelled_scatter(X_fruits_mds, y_fruits, ['apple', 'mandarin', 'orange', 'lemon'])
plt.xlabel('First MDS feature')
plt.ylabel('Second MDS feature')
plt.title("Fruit sample dataset MDS")
```

总结：

t-SNE: a powerful manifold learning method that finds a 2D projection preserving information about neighbors



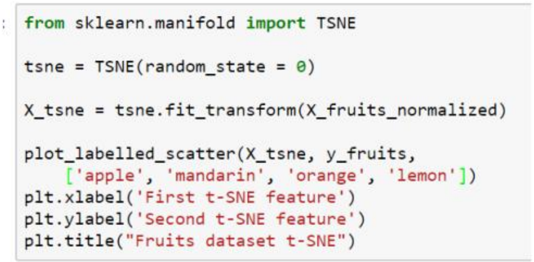
Source: van der Maaten & Hinton https://lvdmaaten.github.io/tsne/

An especially powerful manifold learning algorithm for visualizing your data is called **t-SNE**. t-SNE finds a two-dimensional representation of your data, such that the distances between points in the 2D scatterplot match as closely as possible the distances between the same points in the original high dimensional dataset.

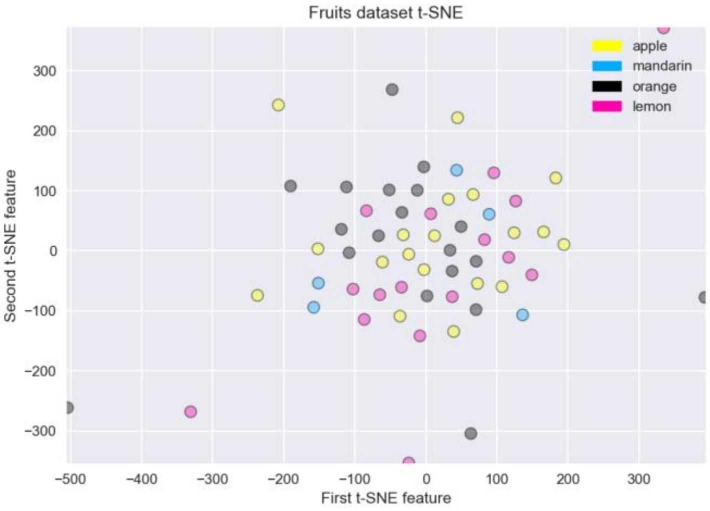
In particular, t-SNE gives much more weight to preserving information about distances between points that are neighbors.

Here's an example of t-SNE applied to the images in the handwritten digits dataset. You can see that this two-dimensional plot preserves the neighbor relationships between images that are similar in terms of their pixels. For example, the cluster for most of the digit eight samples is closer to the cluster for the digits three and five, in which handwriting can appear more similar than to say the digit one, whose cluster is much farther away.

Notebook: t-SNE on the Fruit Dataset



```
from sklearn.manifold import TSNE

tsne = TSNE(random_state = 0)

X_tsne = tsne.fit_transform(X_fruits_normalized)

plot_labelled_scatter(X_tsne, y_fruits,
    ['apple', 'mandarin', 'orange', 'lemon'])
plt.xlabel('First t-SNE feature')
plt.ylabel('Second t-SNE feature')
plt.title("Fruits dataset t-SNE")
```



And here's an example of applying t-SNE on the fruit dataset. The code is very similar to applying MDS and essentially just replaces MDS with t-SNE.

The interesting thing here is that t-SNE does a poor job of finding structure in this rather small and simple fruit dataset,
which reminds us that we should try at least a few different approaches when visualizing data using manifold learning to see which works best for a particular dataset.

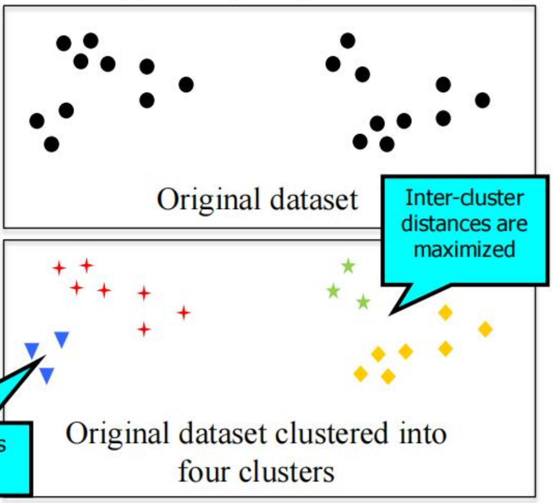
t-SNE tends to work better on datasets that have more well-defined local structure; in other words, more clearly defined patterns of neighbors.

总结:

Clustering:

Finding a way to divide a dataset into groups ('clusters')

- Data points within the same cluster should be 'close' or 'similar' in some way.
- Data points in different clusters should be 'far apart' or 'different'
- Clustering algorithms output a cluster membership index for each data point:
  - *Hard clustering: each data point belongs to exactly one cluster*
  - *Soft (or fuzzy) clustering: each data po is assigned a weight, score of membership for each clu*


Original dataset
Inter-cluster distances are maximized
Intra-cluster distances are minimized
Original dataset clustered into four clusters

If new data points were being added over time, some clustering algorithms could also predict which cluster a new data instance should be assigned to. Similar to classification, but without being able to train the clustering model using label examples in advanced.

K-means Clustering

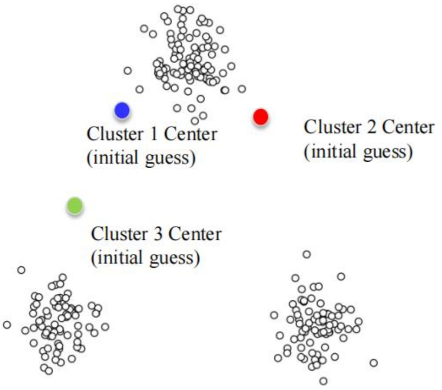
The k-means algorithm

Initialization Pick the number of clusters *k* you want to find. Then pick *k random* points to serve as an initial guess for the cluster centers.

Step A Assign each data point to the nearest cluster center.

Step B Update each cluster center by replacing it with the mean of all points assigned to that cluster (in step A).

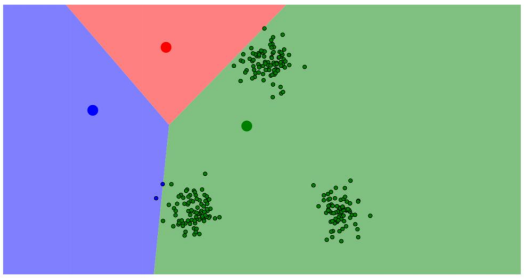
Repeat steps A and B until the centers converge to a stable solution.


Cluster 1 Center (initial guess)
Cluster 2 Center (initial guess)
Cluster 3 Center (initial guess)

Now one aspect of k means is that different random starting points for the cluster centers often result in very different clustering solutions. So typically, the k-means algorithm is run in scikit-learn with ten different random initializations. And the solution occurring the most number of times is chosen.

Demo: https://www.naftaliharris.com/blog/visualizing-k-means-clustering/

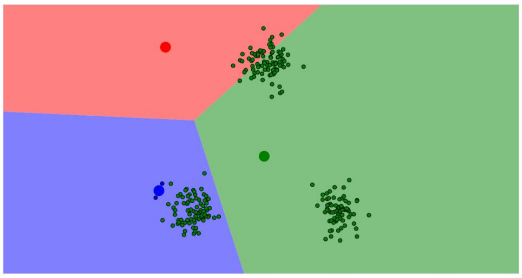
K-means Example: Step 1A



We want three clusters, so three centers are chosen randomly.

Data points are colored according to the closest center.

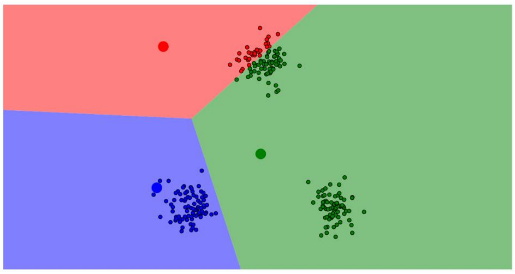
K-means Example: Step 1B



Each center is then updated…

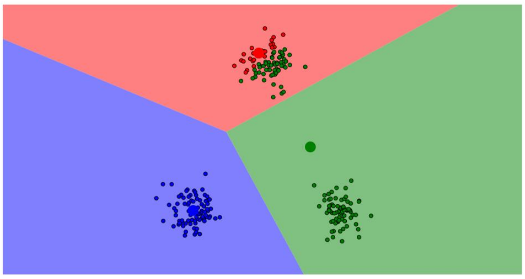
… using the mean of all points assigned to that cluster.

K-means Example: Step 2A



Data points are colored (again) according to the closest center.

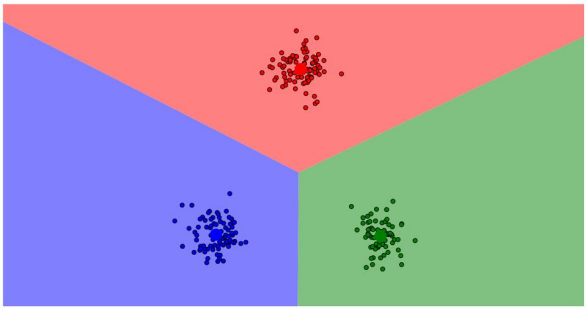
K-means Example: Step 2B



Re-calculate all cluster centers.

总结：

K-means Example: Converged

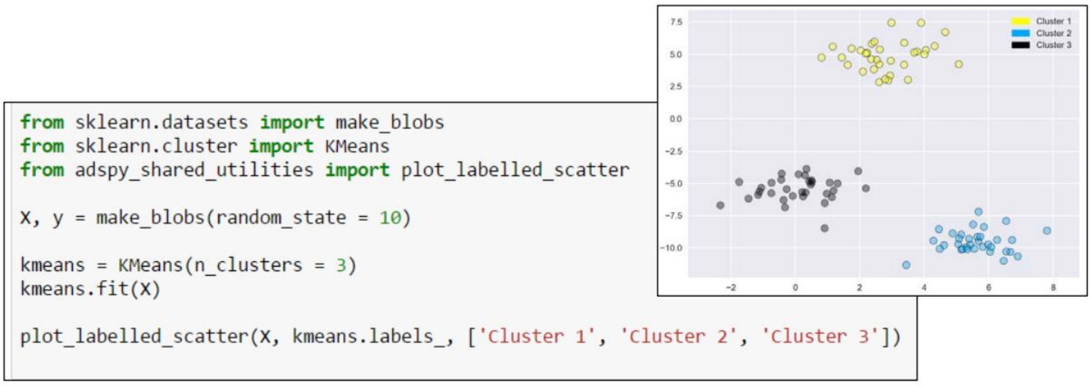


After repeating these steps for several more iterations…

The centers converge to a stable solution!

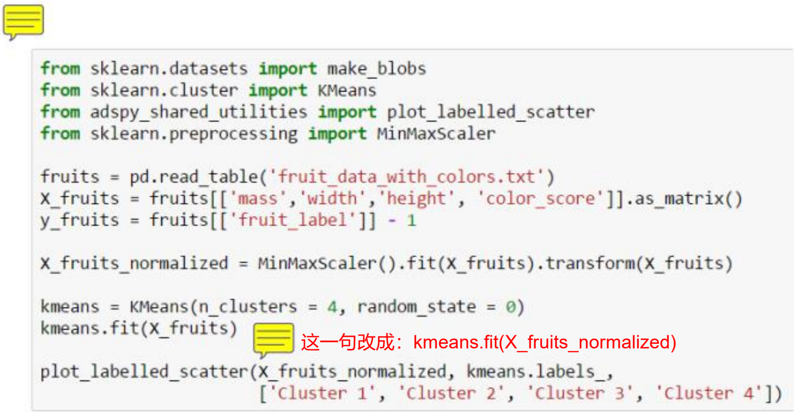
These centers define the final clusters.

k-means Example in Scikit-Learn



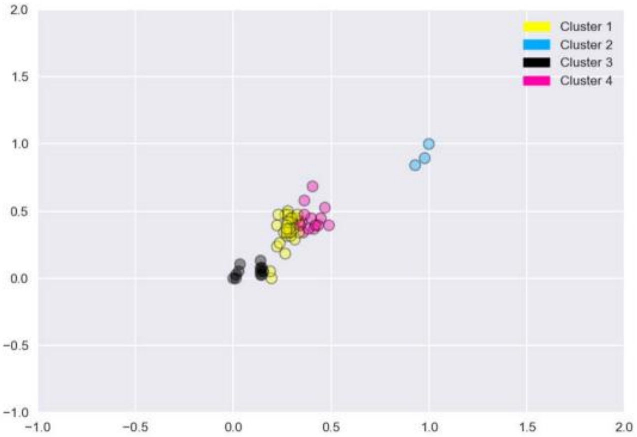
```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from adspy_shared_utilities import plot_labelled_scatter

X, y = make_blobs(random_state = 10)

kmeans = KMeans(n_clusters = 3)
kmeans.fit(X)

plot_labelled_scatter(X, kmeans.labels_, ['Cluster 1', 'Cluster 2', 'Cluster 3'])
```

k-means Output on the Fruits Dataset



```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from adspy_shared_utilities import plot_labelled_scatter
from sklearn.preprocessing import MinMaxScaler

fruits = pd.read_table('fruit_data_with_colors.txt')
X_fruits = fruits[['mass','width','height', 'color_score']].as_matrix()
y_fruits = fruits[['fruit_label']] - 1

X_fruits_normalized = MinMaxScaler().fit(X_fruits).transform(X_fruits)

kmeans = KMeans(n_clusters = 4, random_state = 0)
kmeans.fit(X_fruits)

plot_labelled_scatter(X_fruits_normalized, kmeans.labels_,
                      ['Cluster 1', 'Cluster 2', 'Cluster 3', 'Cluster 4'])
```

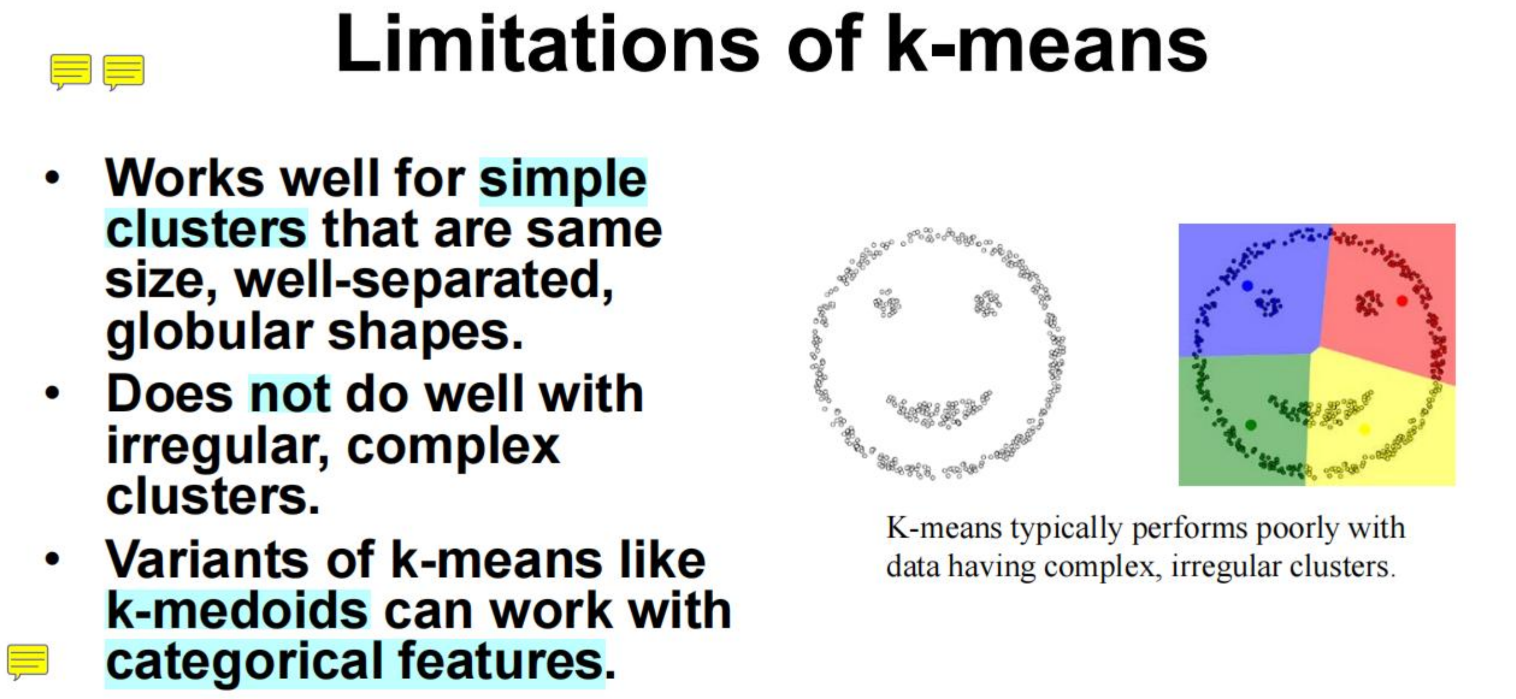
这一句改成：kmeans.fit(X_fruits_normalized)



Can you interpret how these clusters correspond with the true fruit labels?

Note that kmeans is very sensitive to the range of future values. So if your data has features with very different ranges, it's important to normalize using min-max scaling, as we did for some supervised learning methods.

总结：



Limitations of k-means

- Works well for simple clusters that are same size, well-separated, globular shapes.
- Does not do well with irregular, complex clusters.
- Variants of k-means like k-medoids can work with categorical features.

K-means typically performs poorly with data having complex, irregular clusters.

One distinction should be made here between clustering algorithms that can predict which center new data points should be assigned to, and those that cannot make such predictions. K-means supports the predict method, and so we can call the fit and predict methods separately.

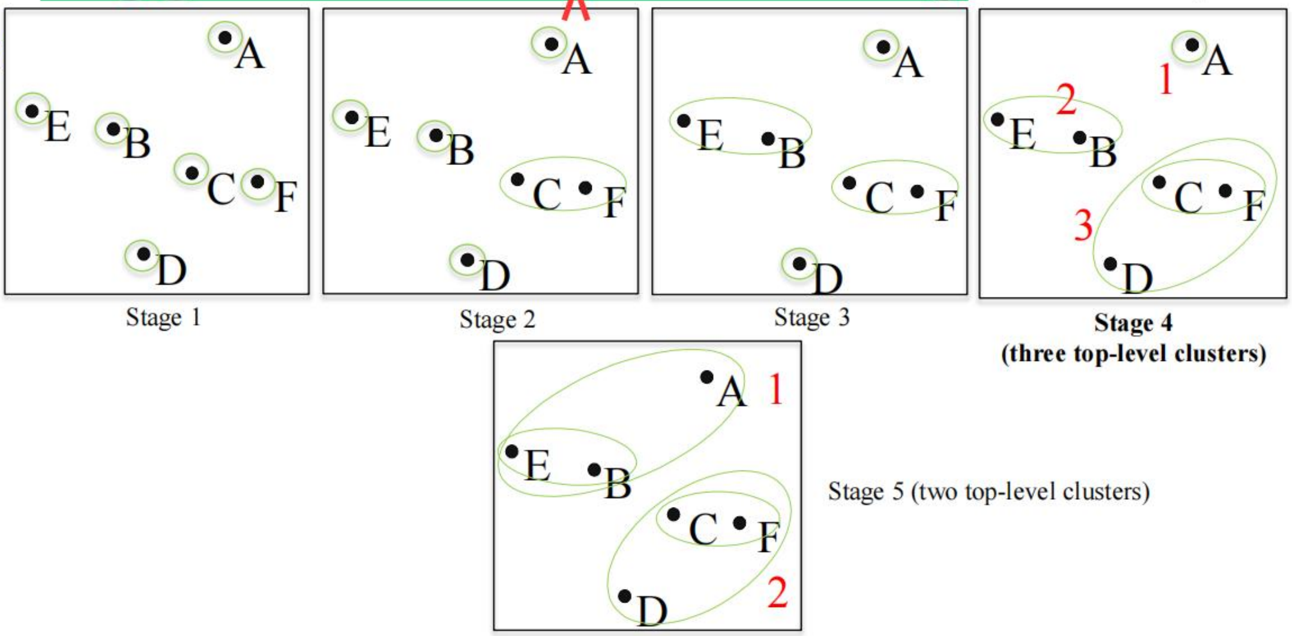
Later methods we'll look at like agglomerative clustering do not and must perform the fit and predict in a single step, as we'll see.

Note that kmeans is very sensitive to the range of future values. So if your data has features with very different ranges, it's important to normalize using min-max scaling, as we did for some supervised learning methods.

Also the version k-means we saw here assumed that the data features were continuous values. However, in some cases we may have categorical features, where taking the mean doesn't make sense.
In that case, there are variants of k-means that can use a more general definition of distance. Such as the **k-medoids algorithm** that can work with categorical features.

总结：

Agglomerative Clustering Example



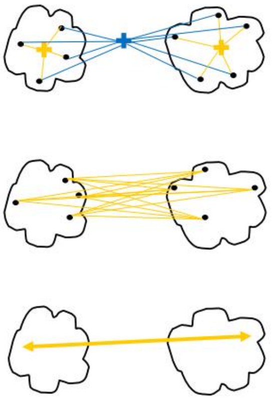
Agglomerative clustering refers to a family of clustering methods that work by doing an iterative bottom up approach.

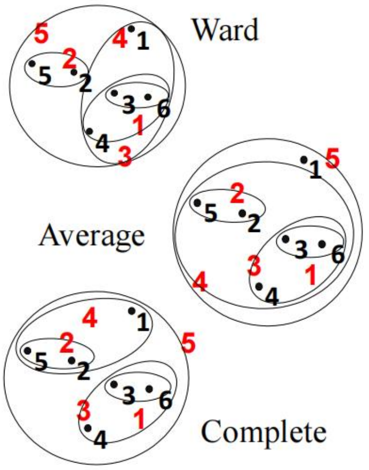
- First, each data point is put into its own cluster of one item.
- Then, a sequence of clusterings are done where the most similar two clusters at each stage are merged into a new cluster. Then, this process is repeated until some stopping condition is met. In scikit-learn, the stopping condition is the number of clusters.

In Stage 1, each data point is in its own cluster, shown by the circles around the points.
In Stage 2, the two most similar clusters, which at this stage amounts to defining the closest points are merged. And this process is continued, as denoted by the expanding and closed regions that denote each cluster.

Linkage Criteria for Agglomerative Clustering

- **Ward's method**
  - *Least increase in total variance (around cluster centroids)*
- **Average linkage**
  - *Average distance between clusters*
- **Complete linkage**
  - *Max distance between clusters*





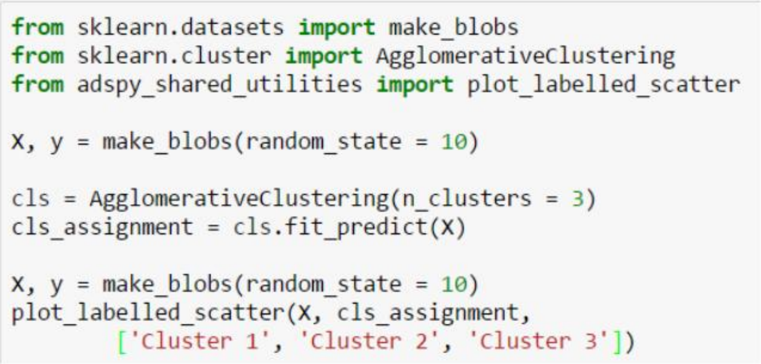
You can choose how the agglomerative clustering algorithm determines the most similar cluster by specifying one of several possible linkage criteria.

- Ward's method chooses to merge the two clusters that give the smallest increase in total variance within all clusters.
- Average linkage merges the two clusters that have the smallest average distance between points.
- Complete linkage, which is also known as maximum linkage, merges the two clusters that have the smallest maximum distance between their points.

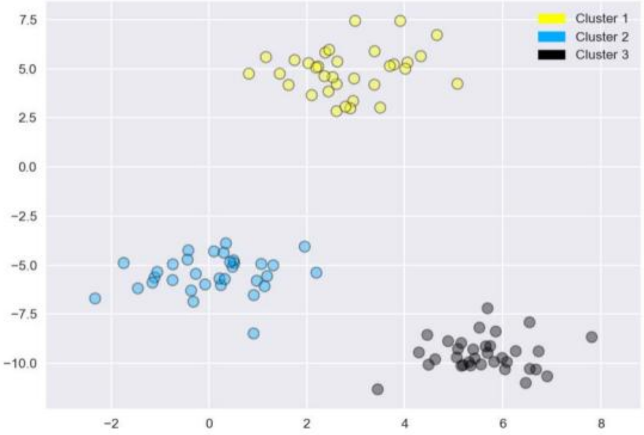
In general, Ward's method works well on most data sets, and that's our usual method of choice. In some cases, if you expect the sizes of the clusters to be very different, for example, that one cluster is much larger than the rest. It's worth trying average and complete linkage criteria as well.

总结:

Agglomerative Clustering in Scikit-Learn

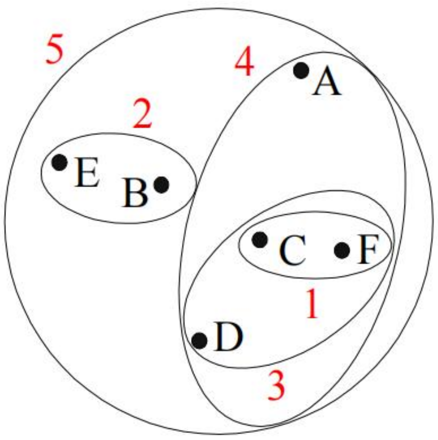


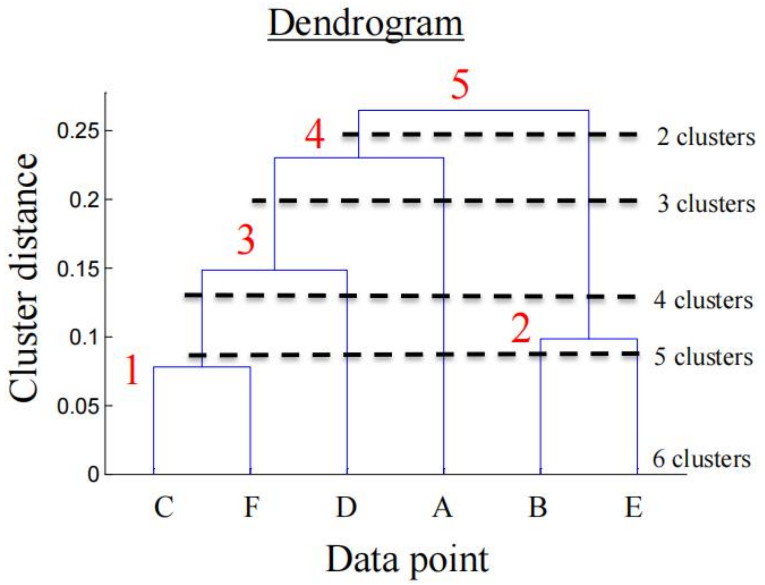
```
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering
from adspy_shared_utilities import plot_labelled_scatter

X, y = make_blobs(random_state = 10)

cls = AgglomerativeClustering(n_clusters = 3)
cls_assignment = cls.fit_predict(X)

X, y = make_blobs(random_state = 10)
plot_labelled_scatter(X, cls_assignment,
        ['Cluster 1', 'Cluster 2', 'Cluster 3'])
```



总结：

Hierarchical Clustering



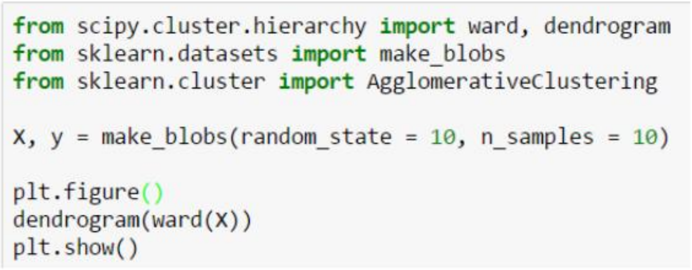


One of the nice things about agglomerative clustering is that it automatically arranges the data into a **hierarchy** as an effect of the algorithm, reflecting the order and cluster distance at which each data point is assigned to successive clusters.
This hierarchy can be useful to visualize using what's called a **dendrogram**, which can be used even with higher dimensional data.

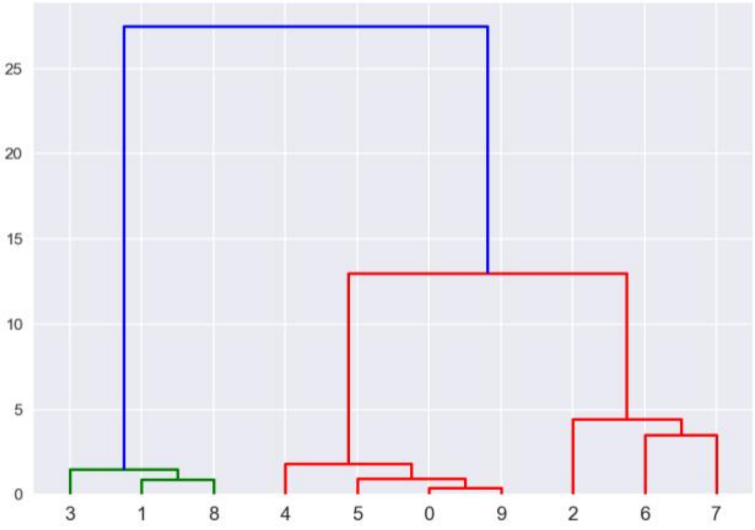
The data points are at the bottom and are numbered. The y axis represents cluster distance, namely, the distance that two clusters are apart in the data space.
The data points form the leaves of the tree at the bottom, and the new node parent in the tree is added as each pair of successive clusters is merged.
The height of the node parent along the y axis captures how far apart the two clusters were when they merged, with the branch going up representing the new merged cluster.

Note that you can tell how far apart the merged clusters are by the length of each branch of the tree.

Dendrogram Example



```
from scipy.cluster.hierarchy import ward, dendrogram
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering

X, y = make_blobs(random_state = 10, n_samples = 10)

plt.figure()
dendrogram(ward(X))
plt.show()
```

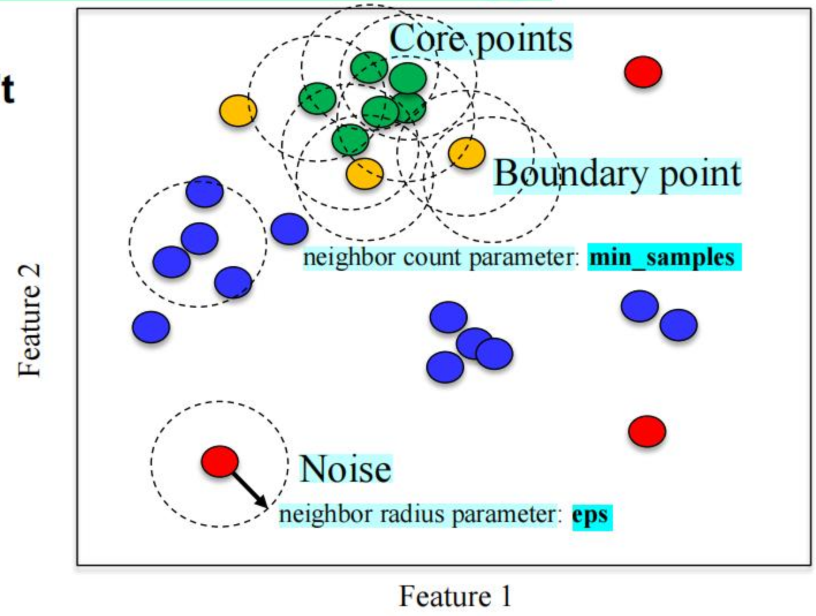


Scikit-learn doesn't provide the ability to plot dendrograms, but SciPy does. SciPy handles clustering a little differently than scikit-learn, but here is an example.
dendrogram /'dendrəgræm/ n. [生物] 系统树图（表示亲缘关系的树状图解）

总结：

DBSCAN Clustering

- Unlike k-means, you don't need to specify # of clusters
- Relatively efficient – can be used with large datasets
- Identifies likely noise points


Core points
Boundary point
neighbor count parameter: min_samples
Noise
neighbor radius parameter: eps
Feature 2
Feature 1

The main idea behind DBSCAN is that clusters represent areas in the dataspace that are more dense with data points, while being separated by regions that are empty or at least much less densely populated.

- One advantage of DBSCAN is that you don't need to specify the number of clusters in advance.
- Another advantage is that it works well with datasets that have more complex cluster shapes.
- It can also find points that are outliers that shouldn't reasonably be assigned to any cluster.

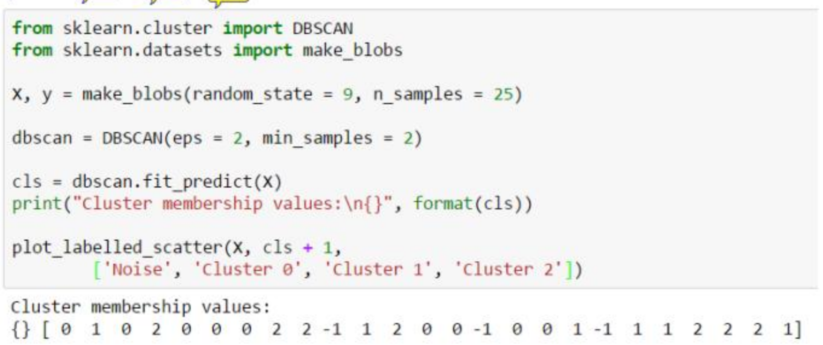
DBSCAN is relatively efficient and can be used for large datasets.

All points that lie in a more dense region are called **core samples**. For a given data point, if there are min sample of other data points that lie within a distance of eps, that given data points is labeled as a core sample. all core samples that are with a distance of eps units apart are put into the same cluster.

Points that are within a distance of eps units from core points, but not core points themselves, are **termed boundary points**.

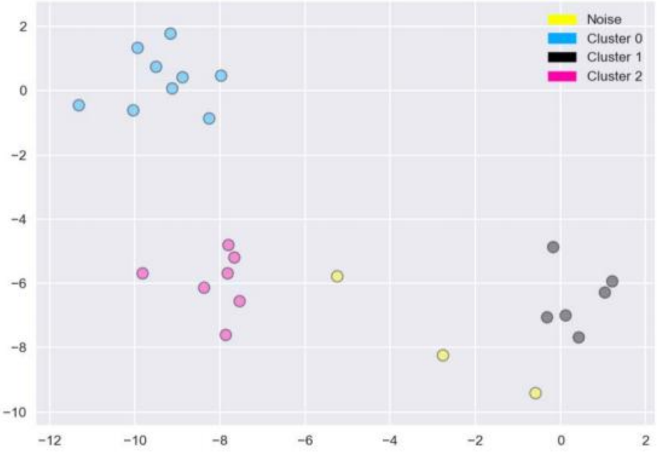
In addition to points being categorized as core samples, points that don't end up belonging to any cluster are considered as **noise**.

DBSCAN Example in Scikit-Learn



```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state = 9, n_samples = 25)

dbscan = DBSCAN(eps = 2, min_samples = 2)

cls = dbscan.fit_predict(X)
print("Cluster membership values:\n{}", format(cls))

plot_labelled_scatter(X, cls + 1,
        ['Noise', 'Cluster 0', 'Cluster 1', 'Cluster 2'])
```

```
Cluster membership values:
{} [ 0  1  0  2  0  0  0  2  2 -1  1  2  0  0 -1  0  0  1 -1  1  1  2  2  2  1]
```



Just like with a agglomerative clustering, DBSCAN doesn't make cluster assignments from new data. So we use the fit predict method to cluster and get the cluster assignments back in one step.

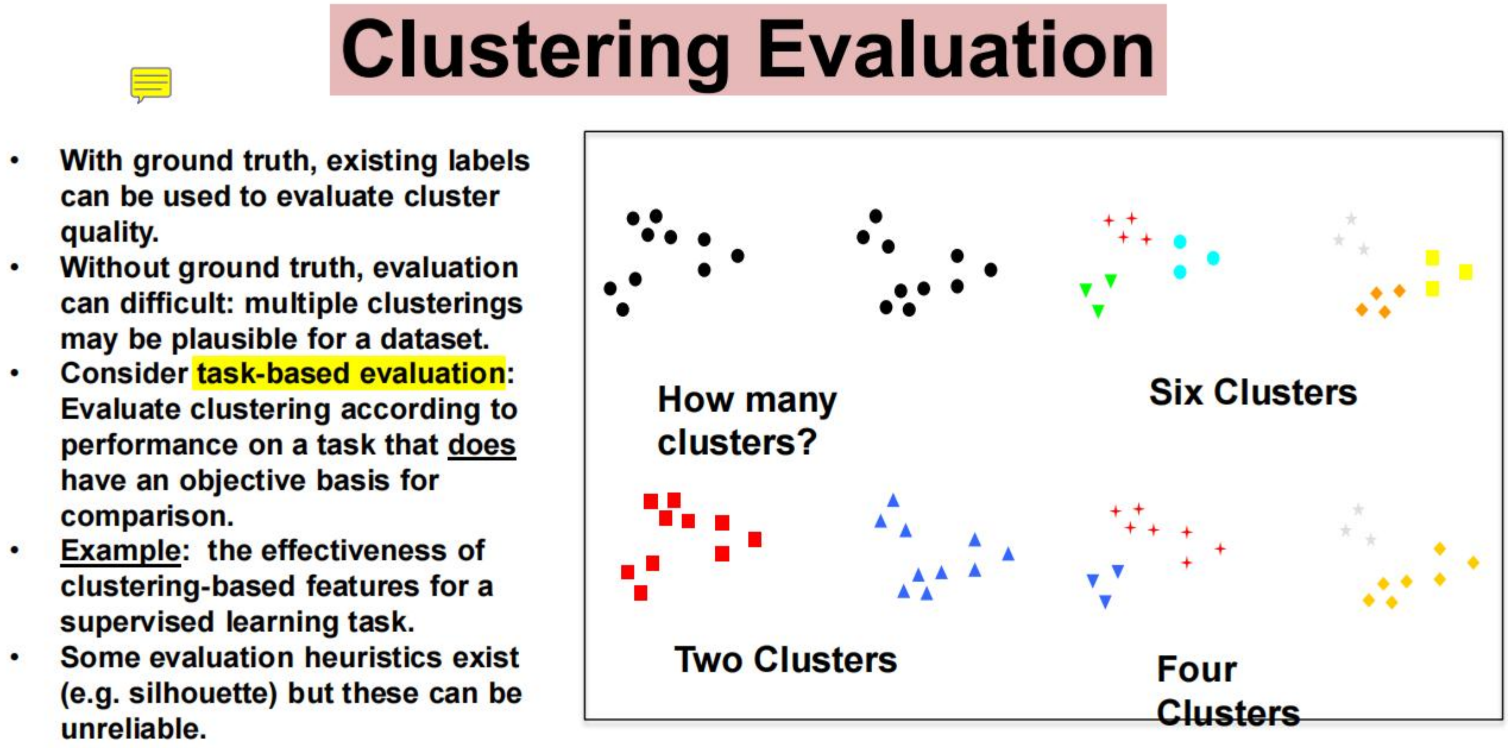
One consequence of not having the right settings of eps and min samples for your particular dataset might be that the cluster memberships returned by DBSCAN may all be assigned the label -1, which indicates noise.

总结：

Basically, the EPS setting does implicitly control the number of clusters that are found.

With DBSCAN, if you've scaled your data using a standard scalar or min-max scalar to make sure the feature values have comparable ranges, finding an appropriate value for eps is a bit easer to do.

One final note, make sure that when you use the cluster assignments from DBSCAN, you check for and handle the -1 noise value appropriately. Since this negative value might cause problems, for example, if the cluster assignment is used as an index into another array later on.



Clustering Evaluation

- With ground truth, existing labels can be used to evaluate cluster quality.
- Without ground truth, evaluation can difficult: multiple clusterings may be plausible for a dataset.
- Consider task-based evaluation: Evaluate clustering according to performance on a task that does have an objective basis for comparison.
- Example: the effectiveness of clustering-based features for a supervised learning task.
- Some evaluation heuristics exist (e.g. silhouette) but these can be unreliable.

Another issue with evaluating clustering algorithms is that it can be hard to automatically interpret or label the meaning of the clusters that are found. And this is still a step that requires human expertise to judge.

总结：