Module 6: Data Clustering and Principal Component Analysis (PCA)

Quick Reference Guide

Learning Outcomes:

1. Apply SVD to a specific dataset
2. Analyze the results of PCA in a specific context
3. Plot and interpret singular values (scree plot)
4. Select r for a desired level of variance capture
5. Create the k-means algorithm in Python
6. Apply the k-means function in Python
7. Interpret the results of k-means and PCA given an initial dataset
8. Utilize scikit-learn to conduct k-means clustering
9. Compare the two different ways of initializing centroids (k-means++ vs. random initialization)
10. Compare the results of multiple clustering techniques on a given dataset

Singular Value Decomposition (SVD)

In the context of machine learning and other areas of science and engineering, two algorithms are of particular importance to data scientists:

- The k-means algorithm for clustering
- The principal component analysis (PCA) for dimensionality reduction

Both k-means and PCA are considered 'unsupervised' learning algorithms, in contrast to supervised learning, as they do not make use of the output in the training data. And in both, an assumption is made that the dataset consists of real numbers only. The dataset has N rows, or samples, and D columns, or features. These two algorithms share a common goal—to find patterns in the data:

- PCA finds patterns in the columns
- K-means finds patterns in the rows

Despite their shared goal, k-means and PCA are fundamentally different.

PCA versus K-means

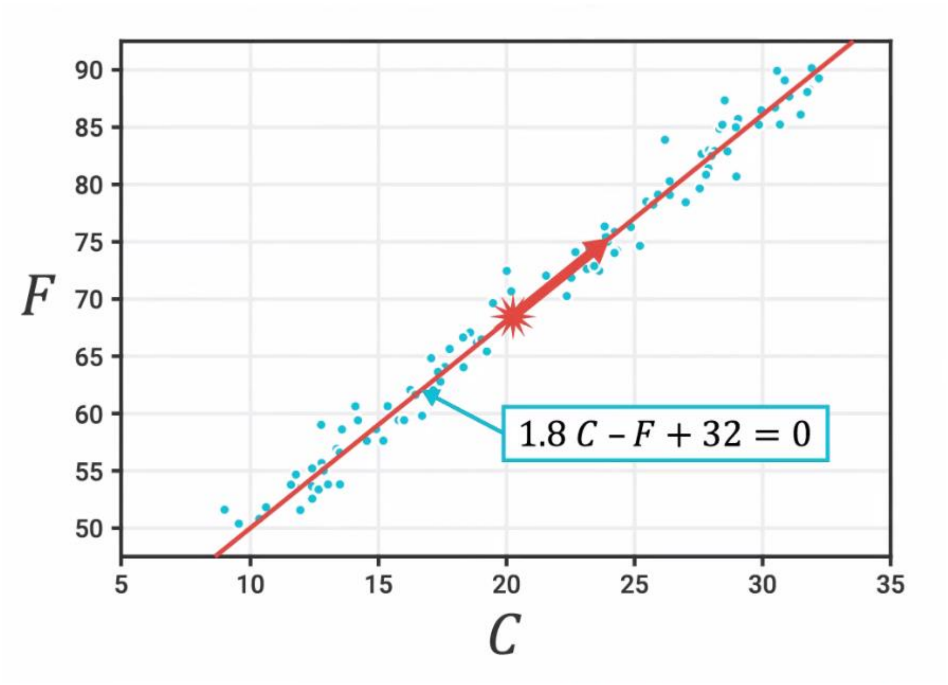
PCA looks for new **columns** that are **linear combinations** of the existing columns, and which capture **the bulk of the variation** in the data. K-means looks for **rows** that are **similar to each** other and **creates groups** out of such rows.

PCA

To help you understand PCA, consider a temperature dataset.

	F	C
0	78.91	25.93
1	82.77	28.13
2	51.70	12.01
3	76.25	23.90
4	82.60	27.70
...	...	...
95	75.75	24.24
96	72.32	20.02
97	85.98	31.52
98	54.94	12.87
99	53.91	13.03

This is a dataset with temperature measurements from 100 different cities. Each temperature is measured with two thermometers, one reporting in Fahrenheit and the other in Centigrade. The temperature measurements, represented in two columns, are highly correlated with a correlation coefficient of 0.99. There is a known, approximately linear, relationship between the two as indicated by the unit conversion formula: $1.8C - F + 32 = 0$.

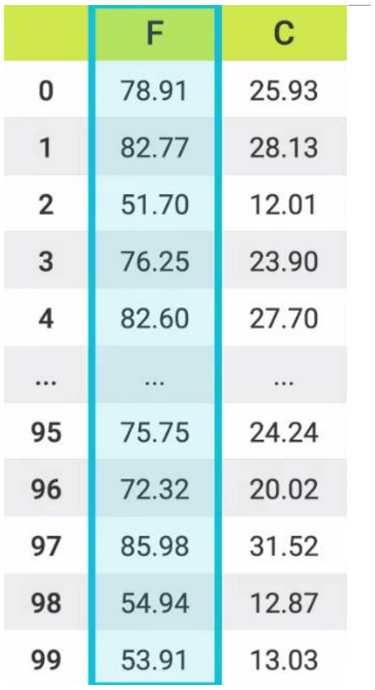


This relationship is not exact, however, because of measurement errors that jitter the data.

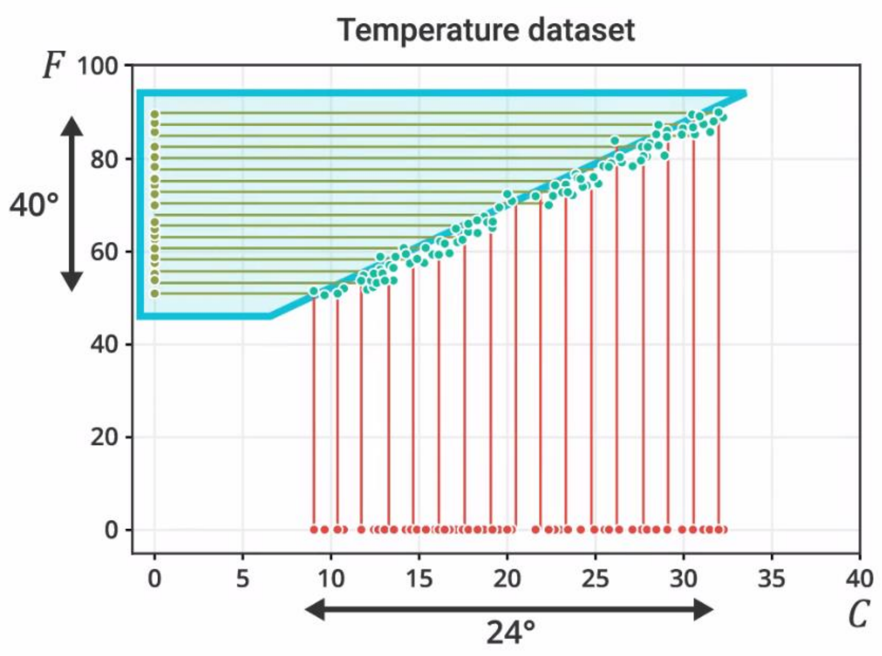
What if your goal is to train a model to predict some other quantity, say, humidity, from these two measurements of temperature? Building a model with two highly-correlated inputs is not a good idea. Rather, you would only use one input because it takes a lot more data to cover a two-dimensional input space properly, than a one-dimensional input space.

This is known as the **curse of dimensionality**; it states that the amount of data you need to train a model increases exponentially with the number of inputs. So, it is well worth your time to **reduce the number of inputs**, or the **dimensionality**, of the model. In this particular case, you would look to reduce the number of inputs from two to one.

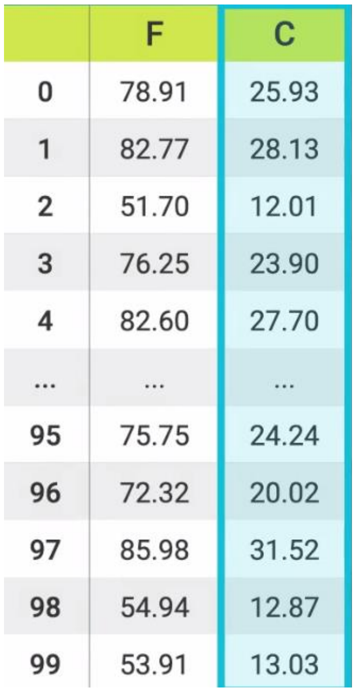
An easy way to achieve this, is to just throw away one of the two columns. After all, they both contain the same information, modulus the noise. If you throw away the column with C-values and keep the column with F-values, you are left with temperatures ranging from about 50 to 90 degrees, a spread of about 40.



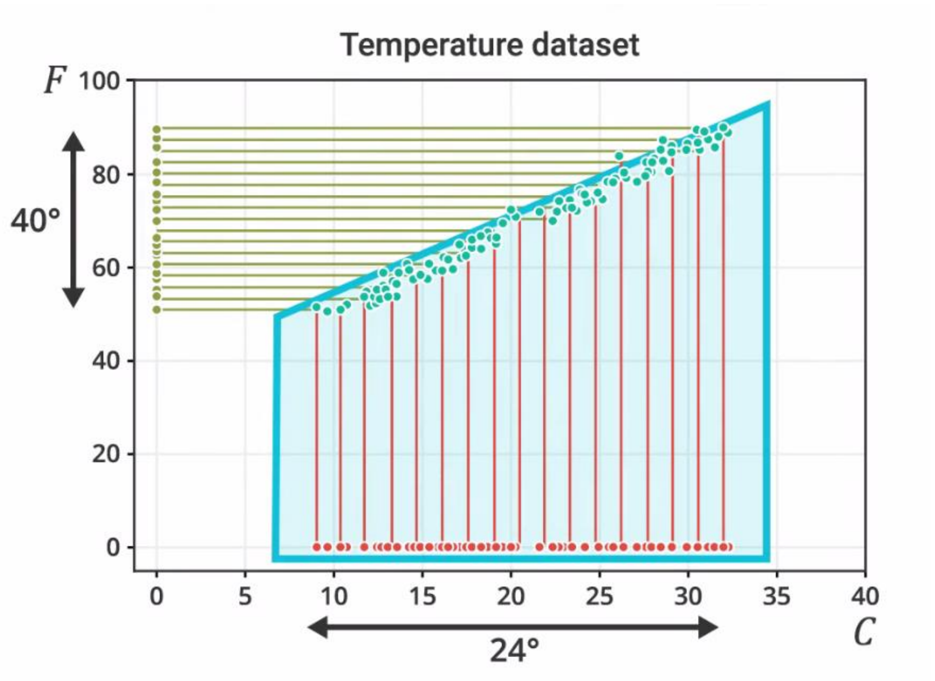
	F	C
0	78.91	25.93
1	82.77	28.13
2	51.70	12.01
3	76.25	23.90
4	82.60	27.70
...	...	...
95	75.75	24.24
96	72.32	20.02
97	85.98	31.52
98	54.94	12.87
99	53.91	13.03



Alternatively, if you throw away the F-values and keep the C-values, the resulting values range from about eight to 32 degrees, a spread of about 24.



	F	C
0	78.91	25.93
1	82.77	28.13
2	51.70	12.01
3	76.25	23.90
4	82.60	27.70
...	...	...
95	75.75	24.24
96	72.32	20.02
97	85.98	31.52
98	54.94	12.87
99	53.91	13.03

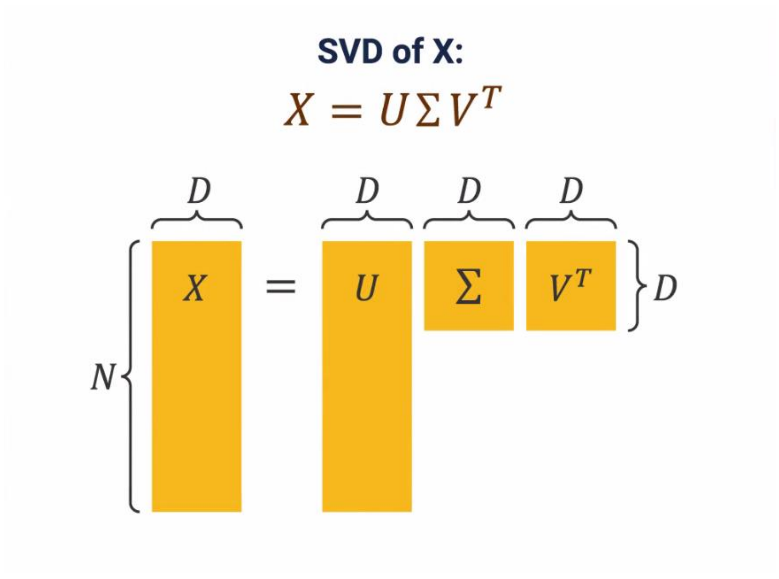


Between these two approaches, the former is preferable because it results in a larger spread, or **variance**, of the data. This leads to a more precise model, assuming the thermometers have equal precision. The idea is that, generally speaking, it is good to have input data spread over a larger area of the input space. In other words, you want to **maximize the variance of the input data**. So your goal is to maximize the variance, which in this instance will give a spread of about 45. And you know from the formula that the inclination is approximately 1.8. But in general, you will not have a handy formula and would need to **compute the variance** from the data alone. This can be done using principal component analysis.

PCA projects data in a way that maximizes the variance and is therefore optimal for building machine learning models for any arbitrary number of inputs. PCA is built upon a technique from linear algebra called singular value decomposition (SVD). SVD is a matrix factorization with many applications. Implementations of SVD are included in a number of Python packages, such as NumPy, SciPy, and scikit-learn. SciPy is similar to NumPy, except slightly faster. Scikit-learn offers a direct implementation of PCA, which can be integrated into a machine learning pipeline.

SVD

This is the formula for SVD.



X represents the input data. It has N rows and D columns. The columns are sometimes referred to as the **features**, and the rows are the **samples**. There will typically be more samples than features. N is usually larger than D, and X is a tall matrix. Running SVD on X decomposes it into three matrices:

- U: A tall matrix with the same dimensions as X
- Sigma (Σ): A square $D \times D$ diagonal matrix with (singular) values along the diagonal only; zeros everywhere else
- V transpose (V^T): A square $D \times D$ matrix

U and V are known respectively as the **left**- and **right singular vector matrices**. Sigma's values on the diagonal are called the **singular values**.

Principal Component Analysis (PCA)

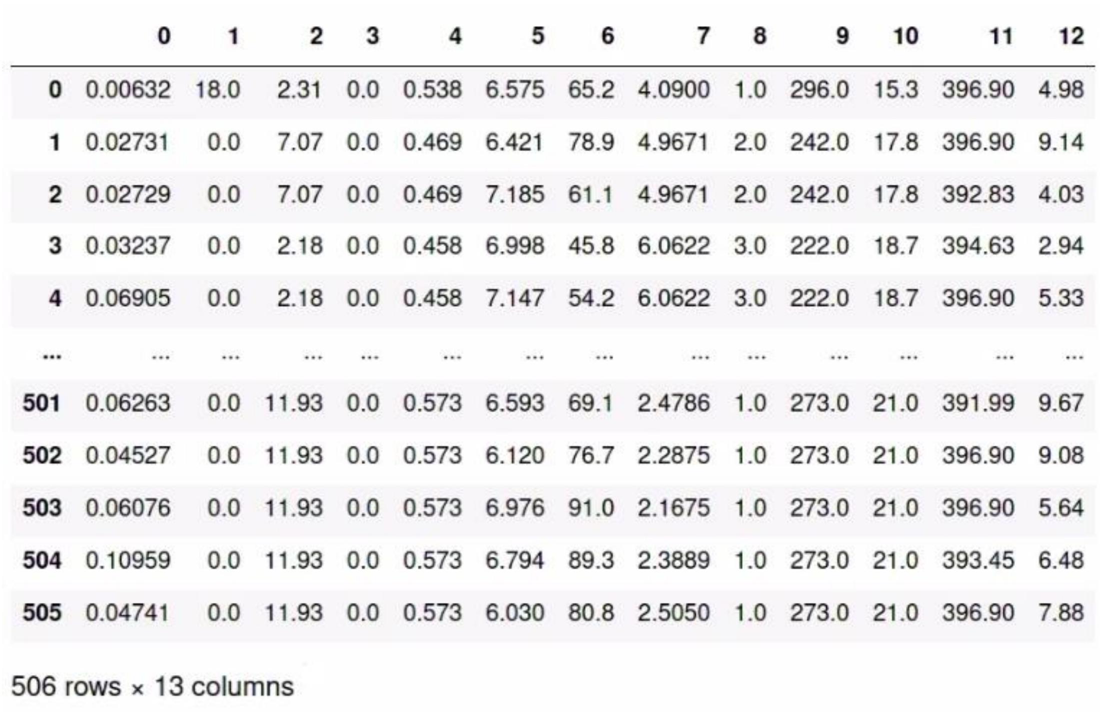
To set the scene for PCA, you import the standard packages for numerical work (NumPy, pandas, and Matplotlib):

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Next, you load the dataset and put it into a dataframe. As an example, the Boston dataset from scikit-learn is used:

```
from sklearn.datasets import load_boston
X, _ = load_boston(return_X_y=True)
X = pd.DataFrame(X)
```

The dataset contains 13 characteristics of 506 different houses in the Boston area. It is useful to predict the price of a house, given 13 anonymous numerical characteristics.



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88

506 rows × 13 columns

PCA involves two main steps:

- Normalize the data
- Perform the singular value decomposition on the normalized data

The function **plot(kind='scatter')** can be used to create a scatterplot of the data. Note that the mean is not necessarily at the origin. For PCA, however, you need the mean to be at 0.

The function **X.describe()** can be used to tell you precisely what the mean is for every column in the dataframe, as a location vector, as well as what the standard deviation is for every column. Major differences between the standard deviation values of your respective columns can have negative effects on the numerical computation of SVD. The formula for normalization of the data should be used to shift the data cloud to the origin and normalize it so that all of the standard deviations are one. The formula to do this is:

$$X_{norm} = \frac{X - \mu}{\sigma}$$

You subtract the mean, μ, from the data cloud, X, to shift it to the origin.

Then you divide it by the standard deviation, σ, in order for the normalized data to have unit standard deviation. When the means all have very small values—essentially zero—and the standard deviations are all ones, the data has been normalized. Once the data has been normalized, you can perform SVD on the normalized data.

Performing SVD on the normalized data

The formula for SVD is: $X_{norm} = U\Sigma V^T$.

To perform SVD on the normalized data, a version of SVD from SciPy linear algebra can be used. So, you can import it:
from scipy.linalg import svd

You can call it svd on the normalized data: **svd(Xnorm)**. When calling that, you can set **full_matrices=False** to avoid matrices that are padded with zeros. You can save the result into U, s, and Vt (V transposed):
U, s, Vt = svd(Xnorm, full_matrices=False)

Ensure Sigma is defined as equal to a diagonal matrix with the singular values, s, along the diagonal: **Sigma = np.diag(s)**. Similarly, V needs to be defined as V transposed: **V = Vt.T**.

The next step is to check that the decomposition is correct. You run the matrix multiplication $(U\Sigma V^T)$ and compare the result to the X_{norm}.

So you run these two functions and compare the results:
U @ Sigma @ V.T
Xnorm

The results should be exactly the same. Alternatively, to check the whole matrix, you can use the **allclose()** function to help ensure every entry in Xnorm is close to the product. If the function outputs **True**, they are the same.

You can also recover the original data from the normalized data with no loss in precision. The function for that is: $X = \mu + \sigma X_{norm}$.

Interpretation of Principal Component Analysis (PCA)

When interpreting U, Σ, and V^T you should keep the goal of maximizing variance in mind.

$$X = \overbrace{\begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_D \\ | & | & & | \end{bmatrix}}^{U} \overbrace{\begin{bmatrix} \sigma_1 & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ 0 & 0 & \cdots & \sigma_{D-1} & 0 \\ 0 & 0 & \cdots & 0 & \sigma_D \end{bmatrix}}^{\Sigma} \overbrace{\begin{bmatrix} — & v_1^T & — \\ — & v_2^T & — \\ & \vdots & \\ — & v_D^T & — \end{bmatrix}}^{V^T}$$

The columns of V are laying horizontally because they're transposed. The Sigma matrix is diagonal and its entries, σ_1 throughσ_D, are the singular values.

This matrix multiplication can be expressed as a summation of D principal components, σ_i, u_i, v_i^T.

$$= \sum_{i=1}^{D} \sigma_i u_i v_i^T \qquad \sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq \sigma_D$$

Each principal component is a rank 1, $N \times D$ matrix, u_i, v_i^T, weighted by the singular values σ_i. All of the D principal components combine to form the matrix X. The optimal directions for projecting the data are represented by

v_i through v_D. Each one is a direction in RD, the input space, and there are D of them. Being mutually orthogonal, as a group, they form a basis for RD.

The singular values, σ_i through σ_D, are organized from largest to smallest. The value of σ_i conveys the importance of the i'th principal component for the dataset. The first principal component captures the largest amount of the total variance and each subsequent component captures a smaller share.

Approximations for dataset X

This useful feature allows you to construct a sequence of approximations to the dataset X, starting from a very coarse approximation considering only the first principal component, up to the exact dataset considering all D principal components.

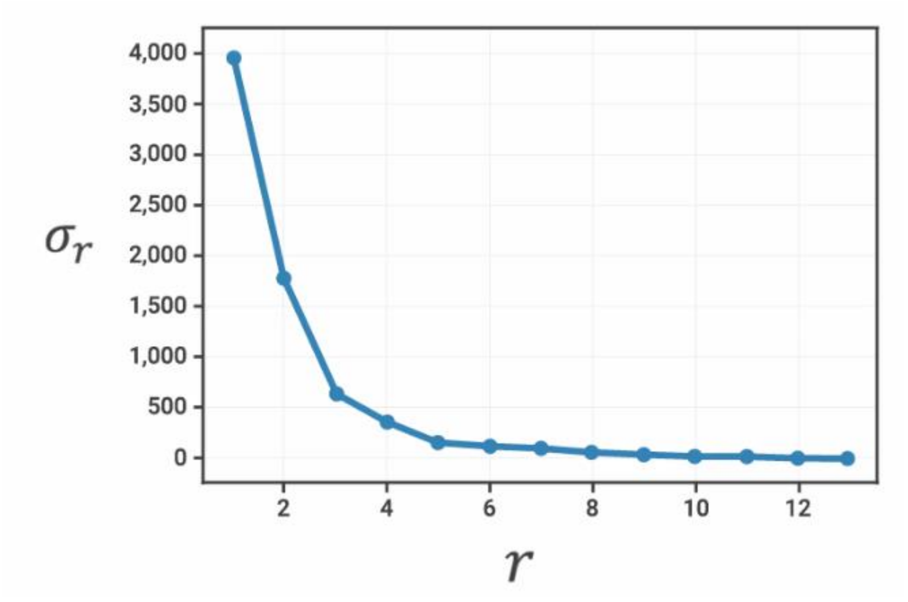
$$1 \leq r \leq D$$

$$\tilde{X}_r^D = \sum_{i=1}^{r} \sigma_i u_i \, v_i^T$$
$$= U_r \Sigma_r v_r^T$$

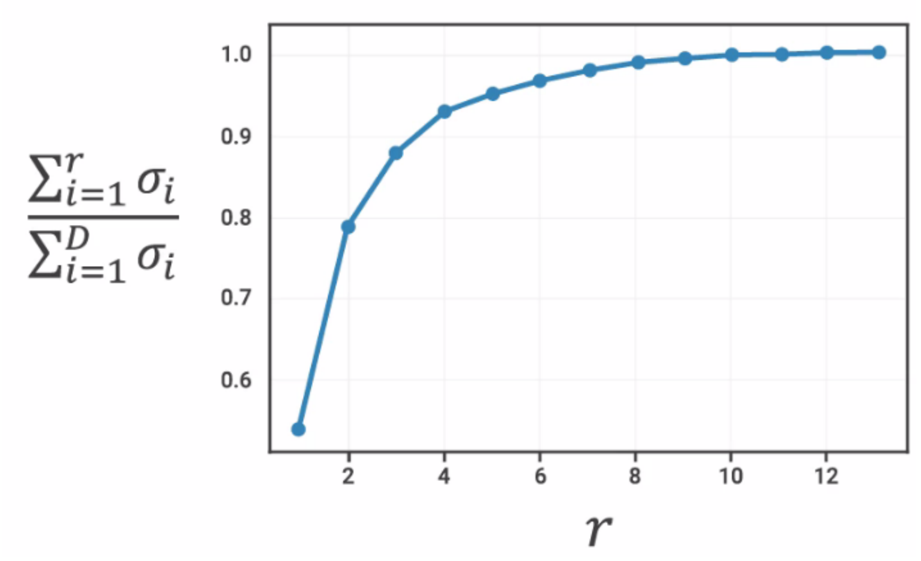
In between there are approximations $\tilde{X}_r^D$, with r varying from 1 to D.

As a matrix multiplication, $Xr, \tilde{r}$ is obtained by keeping the r leftmost columns of $U$and v, and the first are singular values.

Plotting the singular values of a dataset—such as the Boston dataset—in order from the first and largest, to the last and smallest provides a sense of the relative importance of the principal components, with the last few being practically negligible in relation to the first.



Another plot shows the cumulative sum of the singular values divided by their total sum.



It also allows for the quantification of the quality, or total variance captured, by the r'th approximation. This plot can aid decisions on an appropriate level r for the truncation.

As the next step, you would project the data onto the lower dimensional subspace. To do this, you multiply the approximate data matrix in D dimensions, $\tilde{X}_r^D$, by the matrix of principal vectors V_r. This is the formula:

$$\tilde{X}_r^r = \tilde{X}_r^D V_r = U_r \Sigma_r$$

From the formula for the SVD, and the fact that V is orthonormal, this equals U_r times Σ_r.

Principal Component Analysis (Continued)

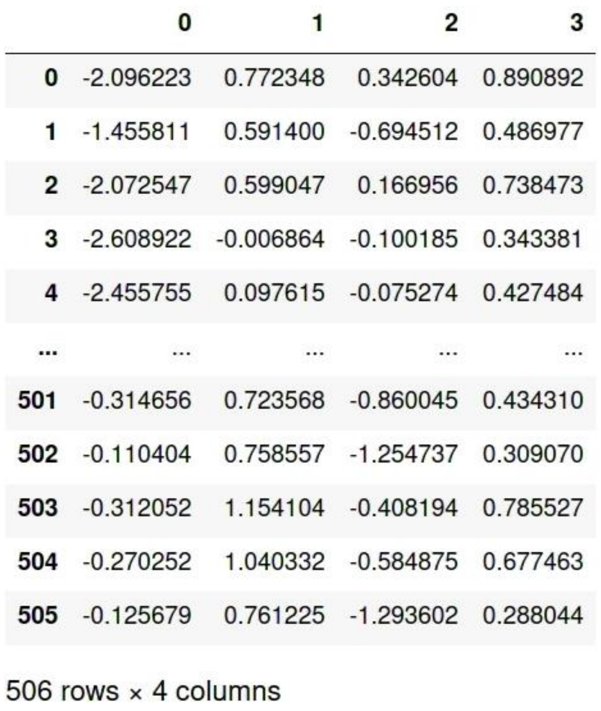
The final component of PCA involves using the SVD of the data and projecting it down to a lower dimensional space.

The formula that you use for finding the lower dimensional data is:

$$\tilde{X}_r^r = U_r \Sigma_r$$

If you decide to project the data onto four dimensions, for example, that means r is equal to 4.

You can use a pandas dataframe with the **pd.DataFrame()** function to view the data:
Xrr = pd.DataFrame(Ur @ Sigmar)



	0	1	2	3
0	-2.096223	0.772348	0.342604	0.890892
1	-1.455811	0.591400	-0.694512	0.486977
2	-2.072547	0.599047	0.166956	0.738473
3	-2.608922	-0.006864	-0.100185	0.343381
4	-2.455755	0.097615	-0.075274	0.427484
...	...	...	...	...
501	-0.314656	0.723568	-0.860045	0.434310
502	-0.110404	0.758557	-1.254737	0.309070
503	-0.312052	1.154104	-0.408194	0.785527
504	-0.270252	1.040332	-0.584875	0.677463
505	-0.125679	0.761225	-1.293602	0.288044

506 rows × 4 columns

Incorporating new data

If new data is introduced at this point, you have two options:

- Add the new data into the original dataset and rerun the whole PCA
- Project the new data into the existing principal components that have already been computed

If the new data is sufficiently different from the data in the original dataset and you expect the principal components to be significantly different, it may be best to compute everything again. However, if the new data is similar to the original dataset, you can project it into the existing space with this formula:

$$\tilde{X}_r^r = \tilde{X}_r^D V_r$$

The projected data is the original normalized data times the first r columns of the V matrix. You can append this to the end of the existing dataframe using **loc()** to produce an updated PCA, if you wish.

SVD and PCA Summary

PCA is used to reduce high-dimensional data to a lower-dimensional space in a way that preserves variability.

The computation of the principal components has two steps:

1. Center the data
2. Call SVD

This produces matrices U, Sigma, and V.

$$X \ = U\Sigma V_T \approx U_r \Sigma_r V_r$$

The V matrix contains the projection directions in the form of linear combinations of the original input columns, which are ordered from most to least significant, with the significance given by the value of the corresponding singular value from the Σ matrix.

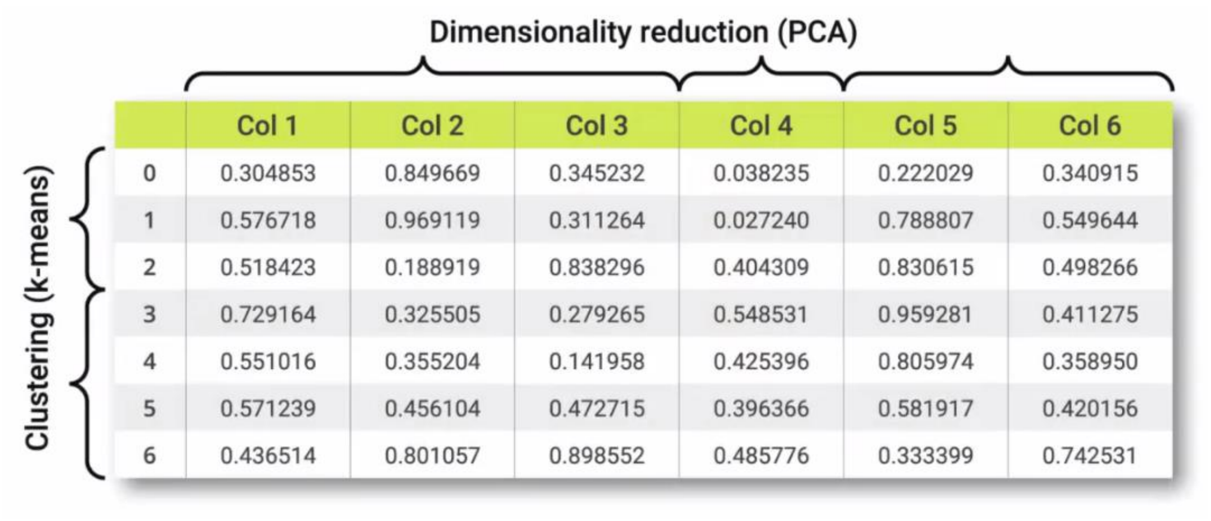
$$U\Sigma = \text{Projected data}$$

You choose the number of components by plotting the cumulative singular values and finding the smallest number that captures the variance you wish to preserve.

The code can be written using the SciPy implementation of SVD. New data can be integrated by first centering it and then projecting it onto the smaller subspace.

Clustering and K-Means

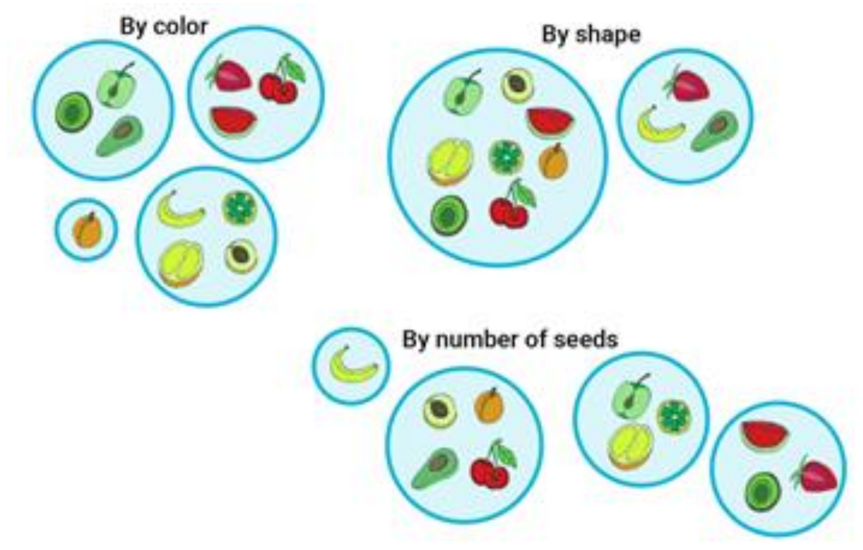
Clustering is a method for creating groups out of the rows in a dataset.


Dimensionality reduction (PCA)
Clustering (k-means)

	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
0	0.304853	0.849669	0.345232	0.038235	0.222029	0.340915
1	0.576718	0.969119	0.311264	0.027240	0.788807	0.549644
2	0.518423	0.188919	0.838296	0.404309	0.830615	0.498266
3	0.729164	0.325505	0.279265	0.548531	0.959281	0.411275
4	0.551016	0.355204	0.141958	0.425396	0.805974	0.358950
5	0.571239	0.456104	0.472715	0.396366	0.581917	0.420156
6	0.436514	0.801057	0.898552	0.485776	0.333399	0.742531

Each row is a sample, or a point in a cloud, whereas each column is a dimension of the space that the cloud lives in.

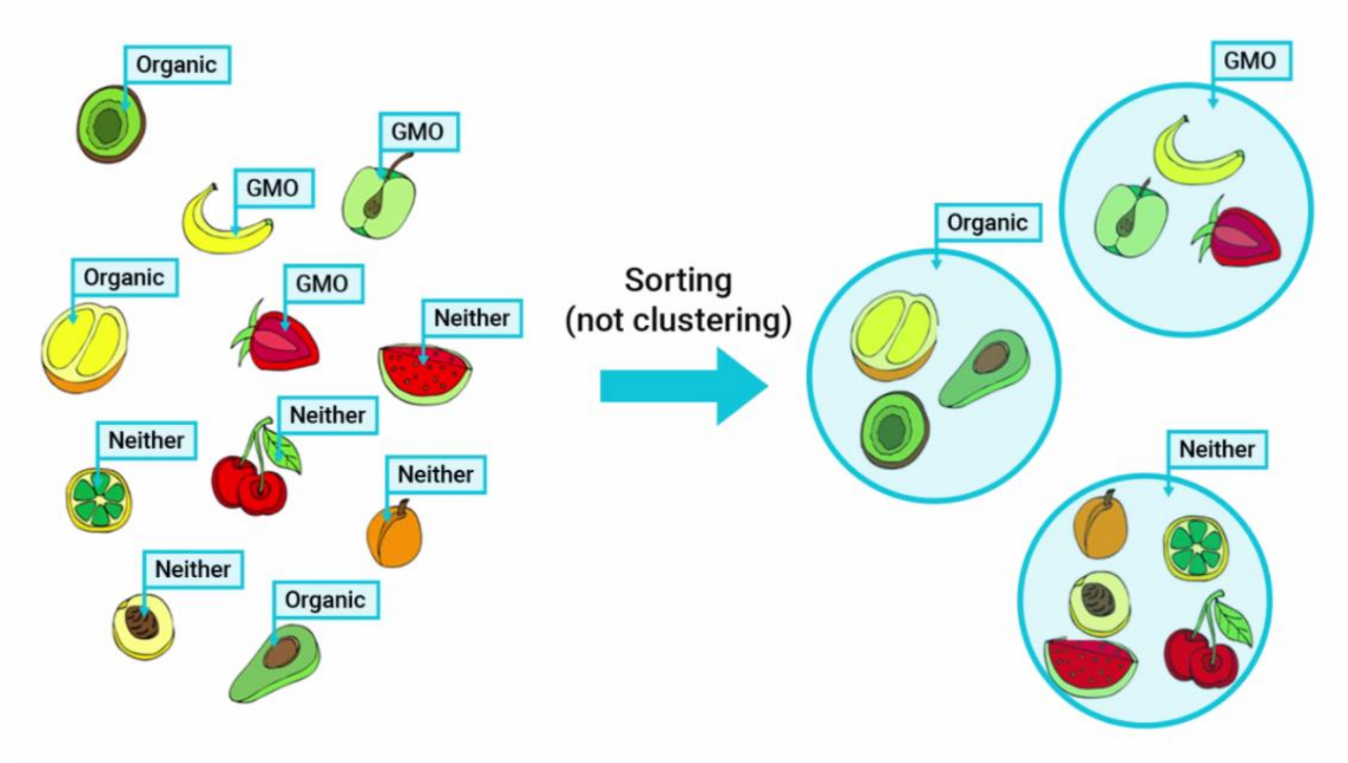
When asked to create groups out of an assortment of items, such as a bunch of different fruit, there may be multiple ways to solve the problem.



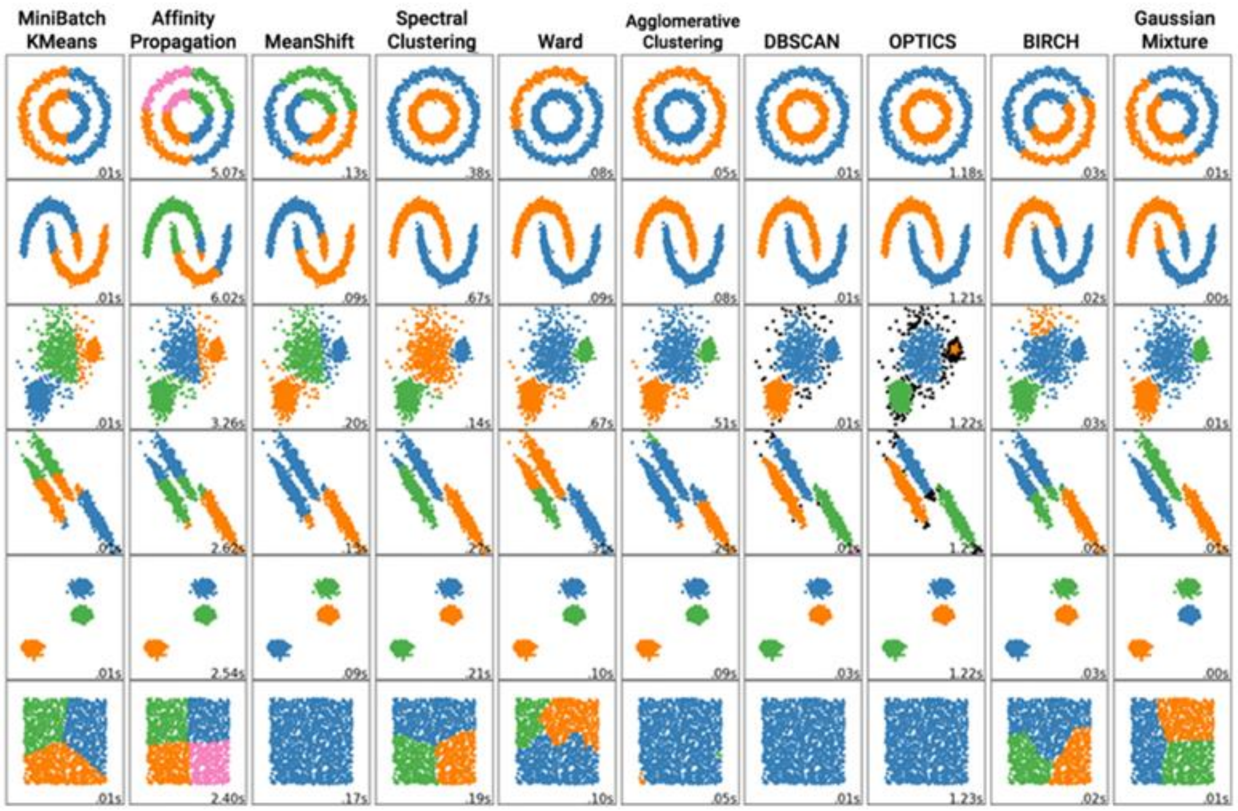
A few options include grouping the fruit by color, shape, or number of seeds. These are good choices that have some characteristics in common.

First, they group fruit according to their **proximity in space**, such as the color space, shape space, or number of seeds space. The characteristics you use and your definition of proximity comprises a crucial design decision when applying a clustering algorithm. Second, they form a moderate number of groups or clusters.

Since the goal of clustering is to provide insight into the data, creating either a single group for all fruit, or a separate group for every individual fruit, would be uninformative and useless.



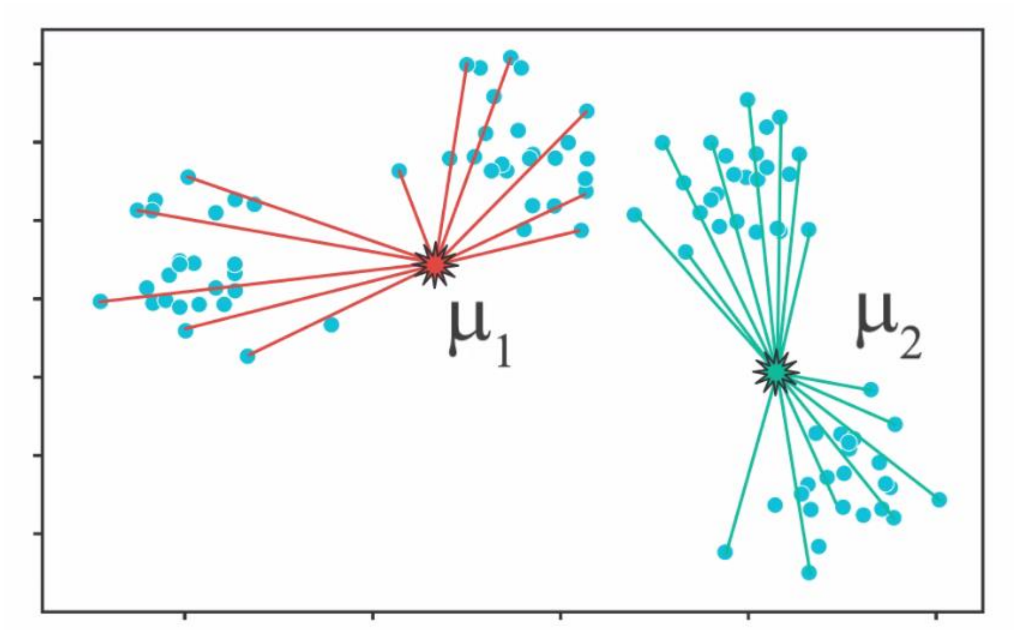
As an unsupervised machine learning method, **clustering algorithms** create group labels where there were none. This approach to grouping **differs from sorting**, which is primarily guided by predetermined labels.



This figure from scikit-learn's documentation on cluster analysis shows ten different clustering methods applied to six different problems on 2D data. It demonstrates some of the strengths and weaknesses of the different algorithms. K-means offers a good entry point for understanding how the algorithms work and it is very popular, as it scales well to large datasets.

Understanding k-means

K-means is a **centroid-based method**. That means that each cluster is characterized by a point, μ_k, called the centroid of the cluster. Datapoints are assigned to their nearest cluster. So, in this case, each point evaluates its distance to both μ_1 and μ_2, and then joins the cluster it is closest to.



The problem that k-means tries to solve is where to place the centroids such that the sum of all of the squared distance is minimized. Summing squared distances to a point is also the formula in mechanics for finding the rotational inertia about that point. For clustering purposes, the **inertia** is the sum of the squared distances from cluster points to their centroids. Hence, **the goal of k-means** is to **minimize the inertia of the clustering** assignment. This is a difficult problem to solve and there are no known algorithms that can reliably find a best solution for an arbitrary dataset in a reasonable amount of time. K-means is a good algorithm for quickly finding suboptimal solutions to this problem.

The basic k-means algorithm iterates between two steps:

- Update step
- Assignment step

The **update step** involves choosing and adjusting the positions of the centroids. In the **assignment step**, datapoints are classified or clustered with their nearest centroid. Subsequent iterations of the update step place each centroid at the mean of the datapoints in its cluster. Iterating this way is guaranteed to converge to a solution. The solution, however, depends uniquely on the initial placement of the centroids. To obtain a good solution with k-means, it is not sufficient to do a single run of the algorithm. You should execute an ensemble of runs, each with a different, randomly-chosen initial condition. You keep only the solution with the lowest inertia.

To select K, which is the number of clusters, you run an ensemble of k-means over a range of different values of K. When K = 1, everything is classified into one large cluster and the result is trivial. The opposite extreme is to set K equal to the number of datapoints. Then the total inertia is zero because every point can be assigned to its own cluster. Between these two extremes, the inertia will decrease monotonically with increasing K.

The **elbow method** involves creating an inertia plot and identifying optimal K. This is the point at which K is on the limit between large and small differential benefit, represented as a sharp change in the slope of inertia.

Clustering in Scikit-Learn

One method of clustering data in Python, is using scikit-learn.

Consider the example of a store owner. They would like to learn more about their customers. As such, they set up a rewards program and collect data from 200 customers. This data includes customers' gender, age, annual income, and spending score. Next, they want to see if there are patterns in this data that can be used to better target those customers.

Clusters can be identified using one of the following:

- K-means
- K-means++
- Density-based spatial clustering of applications with noise (DBSCAN)

K-means uses random initialization and sets the centroids to existing datapoints at random. An advantage of k-means is that it is able to predict, or assign a class to, a new customer.

K-means++ is an improved initialization strategy that distributes the centroids more intelligently throughout the dataset. K-means++ avoids clustering initial centroids, which could lead to numerical problems. Since it works much better than random initialization, especially for large datasets, it is the default in k-means initialization.

DBSCAN is among the better-performing and cutting-edge clustering algorithms. DBSCAN selects its own number of parameters. This clustering algorithm also does not classify all of the points but marks some as outliers. A disadvantage of DBSCAN is that it does not have a predict method.

Here are some functions you may need:

- **pd.read_csv():** Instructs pandas to read the imported CSV file and to load the data into the dataframe.
- **data.set_index():** Lets you specify the dataframe index using existing columns.
- **X.plot(kind='scatter',x= '*Column name*', y='*Column name*'):** Instructs pandas to create a scatterplot using certain columns in the dataframe.

- **Cluster.KMeans(n_clusters=*number of clusters*, init='*initialization strategy*'):** Creates a k-means object with the number of clusters and the initialization strategy that you specify.
- **Kmeans.fit(X):** Runs the algorithm on the k-means object to create the clusters.

DBSCAN

The DBSCAN method has several advantages over k-means.

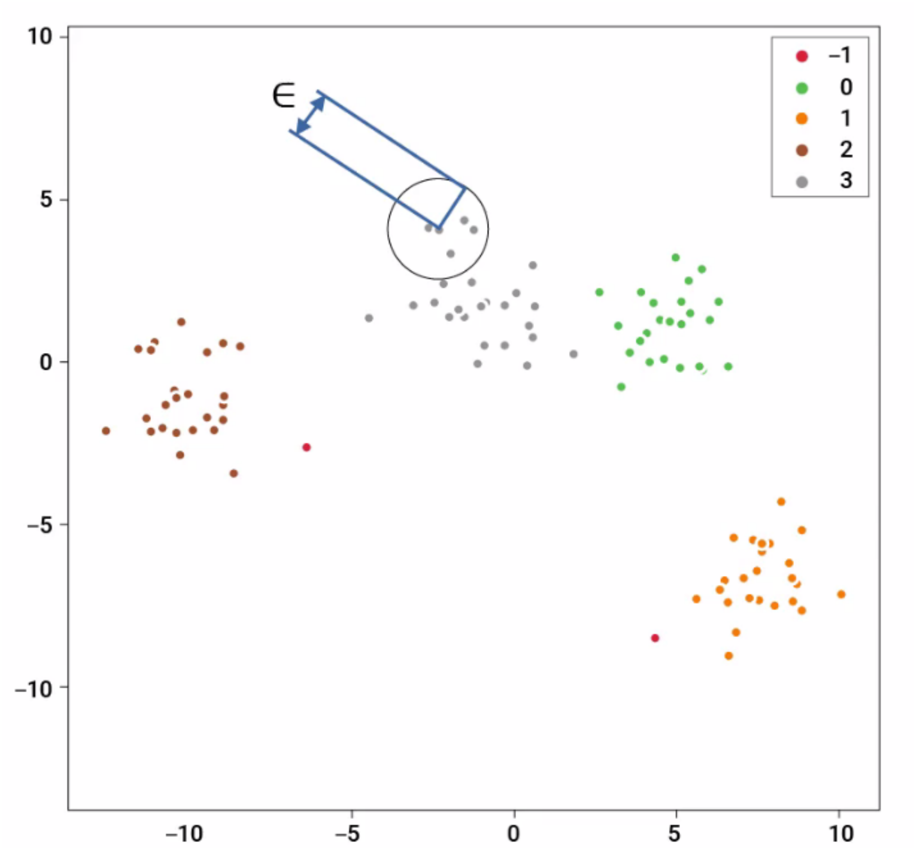
Advantages of DBSCAN

First, DBSCAN is a centroidless algorithm, which means it does not require the initial step of randomly placing the centroids. This eliminates randomness from the algorithm almost completely, and—as a consequence—there is no need to run ensembles of DBSCAN. The lack of centroids also means the number of clusters arises naturally from the algorithm, eliminating the need to run algorithms for various values of K.

A second clear advantage of DBSCAN over k-means is its ability to create curved boundaries between the clusters. In k-means, the area that defines a cluster is always a convex polygon. K-means is incapable of identifying clusters that are not linearly separable.

In addition, DBSCAN has a built-in **outlier detection** feature. Points that are sufficiently removed from the bulk of the data cloud are designated by DBSCAN as outliers and are not classified at all.

To detect outliers, DBSCAN takes two parameters, a radius epsilon (∈) and an integer, min_samples. It begins by drawing a ball of radius ∈ around every point in the dataset.



If the ball captures at least min_sample points, then that point is designated a **core point**. Core points are then joined to other core points in their ∈ neighborhood to form **clusters**. Points that are not core points, but that contain at least one other core point in their ∈ neighborhood are called **boundary points**. These are included in the cluster of their neighboring core points. When a boundary point is adjacent to two different clusters, it is randomly assigned to one of the options. Points that have no core or boundary points in their ∈ ball are the outliers and DBSCAN will assign these a cluster of −1.

DBSCAN is sensitive to its two input parameters: The ∈ radius, and the minimum number of neighbors of core points.