Unit IV

Unsupervised Learning Techniques: Clustering, K-Means, Limits of K-Means, Using Clustering for Image Segmentation, Using Clustering for Preprocessing, Using Clustering for Semi-Supervised Learning, DBSCAN, Gaussian Mixtures.

Dimensionality Reduction: The Curse of Dimensionality, Main Approaches for Dimensionality Reduction, PCA, Using Scikit-Learn, Randomized PCA, Kernel PCA

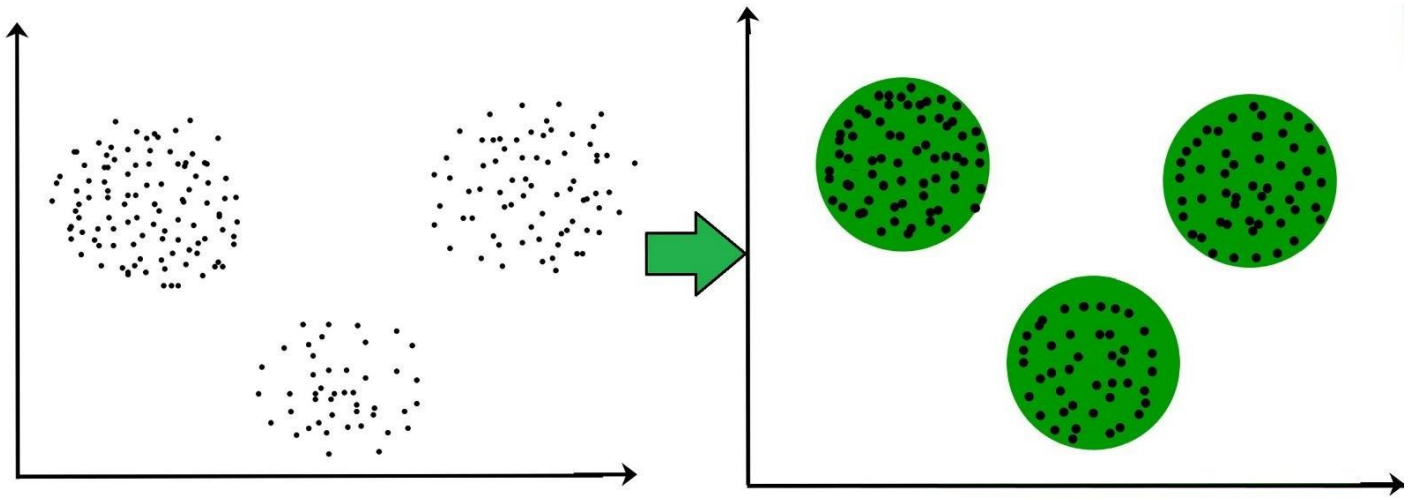
……..

1. Clustering

It is basically a type of *unsupervised learning method*. An unsupervised learning method is a method in which we draw references from datasets consisting of input data without labeled responses. Generally, it is used as a process to find meaningful structure, explanatory underlying processes, generative features, and groupings inherent in a set of examples.

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them.

For ex– The data points in the graph below clustered together can be classified into one single group. We can distinguish the clusters, and we can identify that there are 3 clusters in the below picture.



Clustering is used in a wide variety of applications, including:

a) For customer segmentation: you can cluster your customers based on their purchases, their activity on your website, and so on. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment.

For example, this can be useful in *recommender systems* to suggest content that other users in the same cluster enjoyed.

b) For data analysis: when analyzing a new dataset, it is often useful to first discover clusters of similar instances, as it is often easier to analyze clusters separately

c) As a dimensionality reduction technique: once a dataset has been clustered, it is usually possible to measure each instances *affinity* with each cluster (affinity is any measure of how well an instance fits into a cluster). Each instances feature vector x can then be replaced with the vector of its cluster affinities. If there are *k*-clusters, then this vector is *k*-dimensional. This is typically much lower dimensional than the original feature vector, but it can preserve enough information for further processing.

d) For *anomaly detection* (also called *outlier detection*): any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behaviour; you can detect users with unusual behaviour, such as an unusual number of requests per second, and so on.

Anomaly detection is particularly useful in detecting defects in manufacturing, or for *fraud detection*.

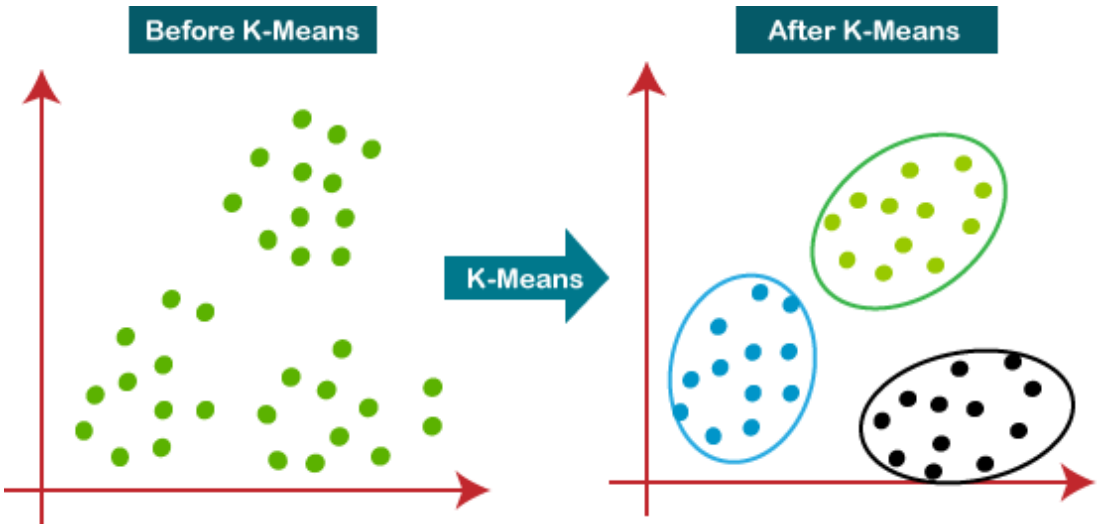
e) For semi-supervised learning: if you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This can greatly increase the amount of labels available for a subsequent supervised learning algorithm, and thus improve its performance.

f) For search engines: for example, some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database: similar images would end up in the same cluster. Then when a user provides a reference image, all you need to do is to find this image's cluster using the trained clustering model, and you can then simply return all the images from this cluster.

g) To segment an image: by clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to reduce the number of different colors in the image considerably. This technique is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

2. k-means clustering

- The k-means clustering algorithm is one of the simplest unsupervised learning algorithms for solving the clustering problem.
- Let it be required to classify a given data set into a certain number of clusters, say, k clusters.
- We start by choosing k points arbitrarily as the "centres" of the clusters, one for each cluster, We then associate each of the given data points with the nearest centre.
- We now take the averages of the data points associated with a centre and replace the centre with the average, and this is done for each of the centres.
- We repeat the process until the centres converge to some fixed points. The data points nearest to the centres form the various clusters in the dataset. Each cluster is represented by the associated centre.



13.2.2 Example

We illustrate the algorithm in the case where there are only two variables so that the data points and cluster centres can be geometrically represented by points in a coordinate plane. The distance between the points (x_1, x_2) and (y_1, y_2) will be calculated using the familiar distance formula of elementary analytical geometry:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}.$$

Problem

Use k-means clustering algorithm to divide the following data into two clusters and also compute the the representative data points for the clusters.

x_1	1	2	2	3	4	5
x_2	1	1	3	2	3	5

Table 13.1: Data for k-means algorithm example

Solution

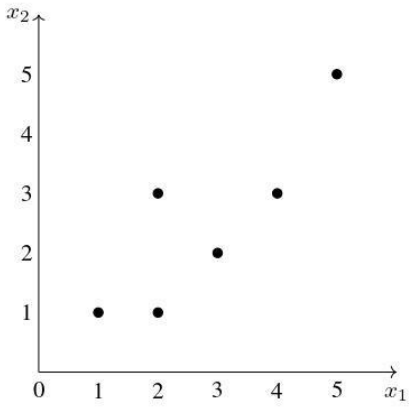


Figure 13.1: Scatter diagram of data in Table 13.1

1. In the problem, the required number of clusters is 2 and we take $k = 2$.
2. We choose two points arbitrarily as the initial cluster centres. Let us choose arbitrarily (see Figure 13.2)
$$\vec{v}_1 = (2, 1), \quad \vec{v}_2 = (2, 3).$$
3. We compute the distances of the given data points from the cluster centers.

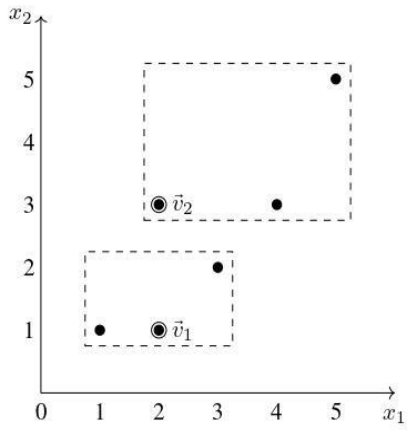


Figure 13.2: Initial choice of cluster centres and the resulting clusters

$\vec{x}_i$	Data point	Distance from $\vec{v}_1 = (2,1)$	Distance from $\vec{v}_2 = (2,3)$	Minimum distance	Assigned center
$\vec{x}_1$	$(1,1)$	1	2.24	1	$\vec{v}_1$
$\vec{x}_2$	$(2,1)$	0	2	0	$\vec{v}_1$
$\vec{x}_3$	$(2,3)$	2	0	0	$\vec{v}_2$
$\vec{x}_4$	$(3,2)$	1.41	1.41	0	$\vec{v}_1$
$\vec{x}_5$	$(4,3)$	2.82	2	2	$\vec{v}_2$
$\vec{x}_6$	$(5,5)$	5	3.61	3.61	$\vec{v}_2$

(The distances of $\vec{x}_4$ from $\vec{v}_1$ and $\vec{v}_2$ are equal. We have assigned $\vec{v}_1$ to $\vec{x}_4$ arbitrarily.)

This divides the data into two clusters as follows (see Figure 13.2):

Cluster 1: $\{\vec{x}_1, \vec{x}_2, \vec{x}_4\}$ represented by $\vec{v}_1$

Number of data points in Cluster 1: $c_1 = 3$.

Cluster 2 : $\{\vec{x}_3, \vec{x}_5, \vec{x}_6\}$ represented by $\vec{v}_2$

Number of data points in Cluster 2: $c_2 = 3$.

4. The cluster centres are recalculated as follows:

$$\begin{aligned}\vec{v}_1 &= \frac{1}{c_1}(\vec{x}_1 + \vec{x}_2 + \vec{x}_4) \\ &= \frac{1}{3}(\vec{x}_1 + \vec{x}_2 + \vec{x}_4) \\ &= (2.00, 1.33) \\ \vec{v}_2 &= \frac{1}{c_2}(\vec{x}_3 + \vec{x}_5 + \vec{x}_6) \\ &= \frac{1}{3}(\vec{x}_3 + \vec{x}_5 + \vec{x}_6) \\ &= (3.67, 3.67)\end{aligned}$$

5. We compute the distances of the given data points from the new cluster centers.

$\vec{x}_i$	Data point	Distance from $\vec{v}_1 = (2,1)$	Distance from $\vec{v}_2 = (2,3)$	Minimum distance	Assigned center
$\vec{x}_1$	$(1,1)$	1.05	3.77	1.05	$\vec{v}_1$
$\vec{x}_2$	$(2,1)$	0.33	3.14	0.33	$\vec{v}_1$
$\vec{x}_3$	$(2,3)$	1.67	1.80	1.67	$\vec{v}_1$
$\vec{x}_4$	$(3,2)$	1.20	1.80	1.20	$\vec{v}_1$
$\vec{x}_5$	$(4,3)$	2.60	0.75	0.75	$\vec{v}_2$
$\vec{x}_6$	$(5,5)$	4.74	1.89	1.89	$\vec{v}_2$

This divides the data into two clusters as follows (see Figure 13.4):

Cluster 1 : $\{\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4\}$ represented by $\vec{v}_1$

Number of data points in Cluster 1: $c_1 = 4$.

Cluster 2 : $\{\vec{x}_5, \vec{x}_6\}$ represented by $\vec{v}_2$

Number of data points in Cluster 1: $c_2 = 2$.

6. The cluster centres are recalculated as follows:

$$\begin{aligned}\vec{v}_1 &== \frac{1}{c_1}(\vec{x}_1 + \vec{x}_2 + \vec{x}_3 + \vec{x}_4) \\ &= \frac{1}{4}(\vec{x}_1 + \vec{x}_2 + \vec{x}_3 + \vec{x}_4) \\ &= (2.00, 1.33) \\ \vec{v}_2 &= \frac{1}{2}(\vec{x}_5 + \vec{x}_6) = (3.67, 3.67)\end{aligned}$$

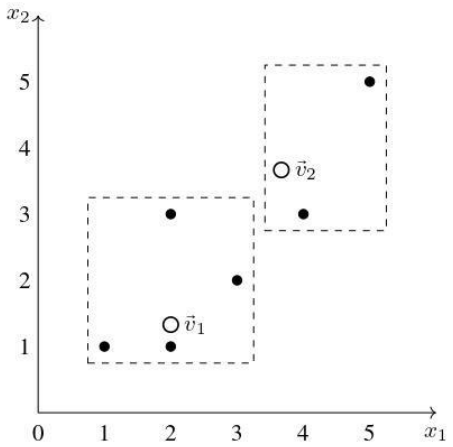


Figure 13.3: Cluster centres after first iteration and the corresponding clusters

7. We compute the distances of the given data points from the new cluster centers.
   4.609772 3.905125 2.692582 2.500000 1.118034 1.118034

$\vec{x}_i$	Data point	Distance from $\vec{v}_1 = (2,1)$	Distance from $\vec{v}_2 = (2,3)$	Minimum distance	Assigned center
$\vec{x}_1$	$(1,1)$	1.25	4.61	1.25	$\vec{v}_1$
$\vec{x}_2$	$(2,1)$	0.75	3.91	0.75	$\vec{v}_1$
$\vec{x}_3$	$(2,3)$	1.25	2.69	1.25	$\vec{v}_1$
$\vec{x}_4$	$(3,2)$	1.03	2.50	1.03	$\vec{v}_1$
$\vec{x}_5$	$(4,3)$	2.36	1.12	1.12	$\vec{v}_2$
$\vec{x}_6$	$(5,5)$	4.42	1.12	1.12	$\vec{v}_2$

This divides the data into two clusters as follows (see Figure **??**):

Cluster 1 : $\{\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4\}$ represented by $\vec{v}_1$

Number of data points in Cluster 1: $c_1 = 4$.

Cluster 2 : $\{\vec{x}_5, \vec{x}_6\}$ represented by $\vec{v}_2$

Number of data points in Cluster 1: $c_1 = 2$.

8. The cluster centres are recalculated as follows:

$$\begin{aligned}\vec{v}_1 &= \frac{1}{c_1}(\vec{x}_1 + \vec{x}_2 + \vec{x}_3 + \vec{x}_4)\\ &= \frac{1}{4}(\vec{x}_1 + \vec{x}_2 + \vec{x}_3 + \vec{x}_4)\\ &= (2.00, 1.75)\\ \vec{v}_2 &= \frac{1}{c_2}(\vec{x}_5 + \vec{x}_6)\\ &= \frac{1}{2}(\vec{x}_5 + \vec{x}_6)\\ &= (4.00, 4.50)\end{aligned}$$

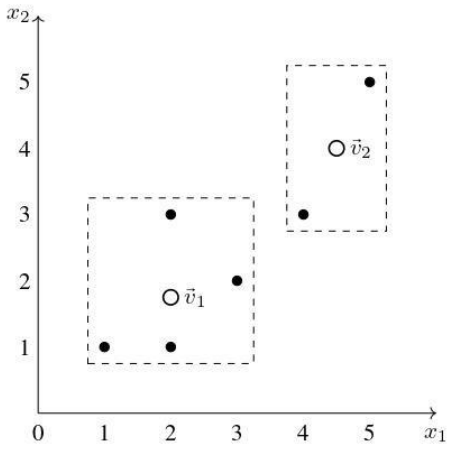


Figure 13.4: New cluster centres and the corresponding clusters

9. This divides the data into two clusters as follows (see Figure **??**):

Cluster 1 : $\{\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4\}$ represented by $\vec{v}_1$

Cluster 2 : $\{\vec{x}_5, \vec{x}_6\}$ represented by $\vec{v}_2$

10. The cluster centres are recalculated as follows:

$$\vec{v}_1 = \frac{1}{4}(\vec{x}_1 + \vec{x}_2 + \vec{x}_3 + \vec{x}_4) = (2.00, 1.75)$$
$$\vec{v}_2 = \frac{1}{2}(\vec{x}_5 + \vec{x}_6) = (4.00, 4.50)$$

We note that these are identical to the cluster centres calculated in Step 8. So there will be no reassignment of data points to different clusters and hence the computations are stopped here.

11. Conclusion: The k means clustering algorithm with $k = 2$ applied to the dataset in Table 13.1 yields the following clusters and the associated cluster centres:

Cluster 1 : $\{\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4\}$ represented by $\vec{v}_1 = (2.00, 1.75)$

Cluster 2 : $\{\vec{x}_5, \vec{x}_6\}$ represented by $\vec{v}_2 = (2.00, 4.75)$

13.2.3 The algorithm

Notations

We assume that each data point is a n-dimensional vector:

$$\vec{x} = (x_1, x_2, \ldots, x_n).$$

The distance between two data points

$$\vec{x} = (x_1, x_2, \ldots, x_n)$$

and

$$\vec{y} = (y_1, y_2, \ldots, x_n)$$

is defined as

$$\|\vec{x} - \vec{y}\| = \sqrt{(x_1 - y_1)^2 + \cdots (x_n - y_n)^2}.$$

Let $X = \{\vec{x}_1, \ldots, \vec{x}_N\}$ be the set of data points, $V = \{\vec{v}_1, \ldots, \vec{v}_k\}$ be the set of centres and c_i for $i = 1, \ldots, k$ be the number of data points in the i-th cluster

Basic idea

What the algorithm aims to achieve is to find a partition the set X into k mutually disjoint subsets $S = \{S_1, S_2, \ldots, S_k\}$ and a set of data points V which minimizes the following within-cluster sum of errors:

$$\sum_{i=1}^{k} \sum_{\vec{x} \epsilon S_i} \|\vec{x} - \vec{v}_i\|^2$$

Algorithm

Step 1. Randomly select k cluster centers $\vec{v}_1, \ldots, \vec{v}_k$.

Step 2. Calculate the distance between each data point $\vec{x}_i$ and each cluster center $\vec{v}_j$.

Step 3. For each $j = 1, 2, \ldots, N$, assign the data point $\vec{x}_j$ to the cluster center $\vec{v}_i$ for which the distance $\|\vec{x}_j - \vec{v}_i\|$ is minimum. Let $\vec{x}_{i1}, \vec{x}_{i2}, \ldots, \vec{x}_{ic_i}$ be the data points assigned to $\vec{v}_i$.

Step 4. Recalculate the cluster centres using

$$\vec{v}_i = \frac{1}{c_i}(\vec{x}_{i1} + \cdots + \vec{x}_{ic_i}), \quad i = 1, 2, \ldots, k.$$

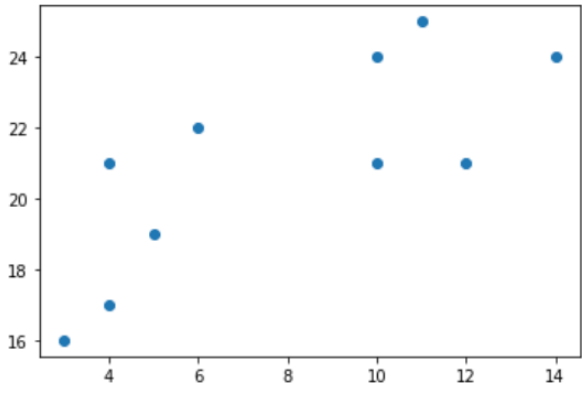
Step 5. Recalculate the distance between each data point and newly obtained cluster centers.

Step 6. If no data point was reassigned then stop. Otherwise repeat from Step 3.

Program:

#1.Start by visualizing some data points:

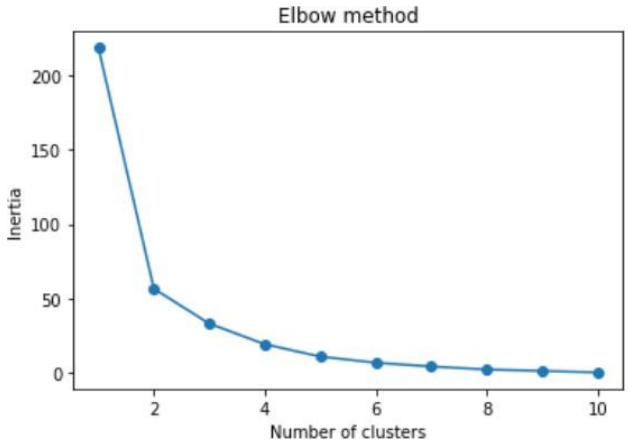
```
import matplotlib.pyplot as plt
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
plt.scatter(x, y)
plt.show()
```



Output:-

#2.Now we utilize the elbow method to visualize the intertia for different values of K:

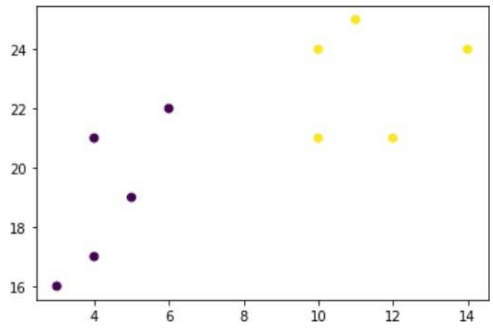
```
from sklearn.cluster import KMeans
data = list(zip(x, y))
inertias = []
for i in range(1,11):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)
plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```



#3.The elbow method shows that 2 is a good value for K, so we retrain and visualize the result:

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)
plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

Output:-



Disadvantages of k-means

1. Choosing k manually

Use the "Loss vs. Clusters" plot to find the optimal (k), as discussed in Interpret Results.

2. Being dependent on initial values.

For a low , you can mitigate this dependence by running k-means several times with different initial values and picking the best result. As increases, you need advanced versions of k-means to pick better values of the initial centroids (called k-means seeding).

3. Clustering data of varying sizes and density.

k-means has trouble clustering data where clusters are of varying sizes and density. To cluster such data, you need to generalize k-means as described in the Advantages section.

4. Clustering outliers.

Centroids can be dragged by outliers, or outliers might get their own cluster instead of being ignored. Consider removing or clipping outliers before clustering.

5. Scaling with number of dimensions.

As the number of dimensions increases, a distance-based similarity measure converges to a constant value between any given examples. Reduce dimensionality either by using PCA on the feature data, or by using "spectral clustering" to modify the clustering algorithm as explained below.

3. Using Clustering for Image Segmentation

Image segmentation is the task of partitioning an image into multiple segments.

In *semantic segmentation*, all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would just be one segment containing all the pedestrians).

In *instance segmentation*, all pixels that are part of the same individual object are assigned to the same segment.

K-means clustering is a very popular clustering algorithm which applied when we have a dataset with labels unknown. The goal is to find certain groups based on some kind of similarity in the data with the number of groups represented by K. This algorithm is generally used in areas like market segmentation, customer segmentation, etc. But, it can also be used to segment different objects in the images on the basis of the pixel values.

The algorithm for image segmentation works as follows:

1. First, we need to select the value of K in K-means clustering.
2. Select a feature vector for every pixel (color values such as RGB value, texture etc.).
3. Define a similarity measure b/w feature vectors such as Euclidean distance to measure the similarity b/w any two points/pixel.
4. Apply K-means algorithm to the cluster centers
5. Apply connected component's algorithm.
6. Combine any component of size less than the threshold to an adjacent component that is similar to it until you can't combine more.

Example:

```
from scipy import ndimage
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
import random
#PARAMS
n_clusters=10
#fake train data
original_train = np.random.random((100, 100, 100, 3)) #100 images of each 100 px,py and RGB
n,x,y,c = original_train.shape
flat_train = original_train.reshape((n,x*y*c))
kmeans = KMeans(n_clusters, random_state=0)
clusters = kmeans.fit_predict(flat_train)
centers = kmeans.cluster_centers_
#visualize centers:
for ci in centers:
    plt.imshow(ci.reshape(x,y,c))
    plt.show()
#visualize other members
for cluster in np.arange(n_clusters):
    cluster_member_indices = np.where(clusters == cluster)[0]
    print("There are %s members in cluster %s" % (len(cluster_member_indices), cluster))
    #pick a random member
    random_member = random.choice(cluster_member_indices)
    plt.imshow(original_train[random_member,:,:,:])
    plt.show()
```

5. Using Clustering for Preprocessing

Clustering can be an efficient approach to dimensionality reduction, in particular as a preprocessing step before a supervised learning algorithm. For example, let's tackle the *digits dataset* which is a simple MNIST-like dataset containing 1,797 grayscale 8×8 images representing digits 0 to 9. First, let us load the dataset

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([

("kmeans", KMeans(n_clusters=50)),

("log_reg", LogisticRegression()),

])

pipeline.fit(X_train, y_train)

pipeline.score(X_test, y_test)
```

Output:-

0.9644444444444444

How about that? We almost divided the error rate by a factor of 2! But we chose the number of clusters *k* completely arbitrarily, we can surely do better. Since K-Means is just a preprocessing step in a classification pipeline, finding a good value for *k* is much simpler than earlier: there's no need to perform silhouette analysis or minimize the inertia, the best value of *k* is simply the one that results in the best classification performance during cross-validation.

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([

("kmeans", KMeans(n_clusters=90)),

("log_reg", LogisticRegression()),

])

pipeline.fit(X_train, y_train)

pipeline.score(X_test, y_test)
```

Output:-

0.9688888888888889

6. Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances.

Let us train a logistic regression model on a sample of 50 labeled instances from the digits dataset:

```
n_labeled = 50

log_reg = LogisticRegression()

log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

What is the performance of this model on the test set?

```
>>> log_reg.score(X_test, y_test)

0.8266666666666667
```

The accuracy is just 82.7%: it should come as no surprise that this is much lower than earlier, when we trained the model on the full training set. Let us see how we can do better. First, let us cluster the training set into 50 clusters, then for each cluster let us find the image closest to the centroid. We will call these images the representative images

k = 50

kmeans = KMeans(n_clusters=k)

X_digits_dist = kmeans.fit_transform(X_train)

representative_digit_idx = np.argmin(X_digits_dist, axis=0)

X_representative_digits = X_train[representative_digit_idx]

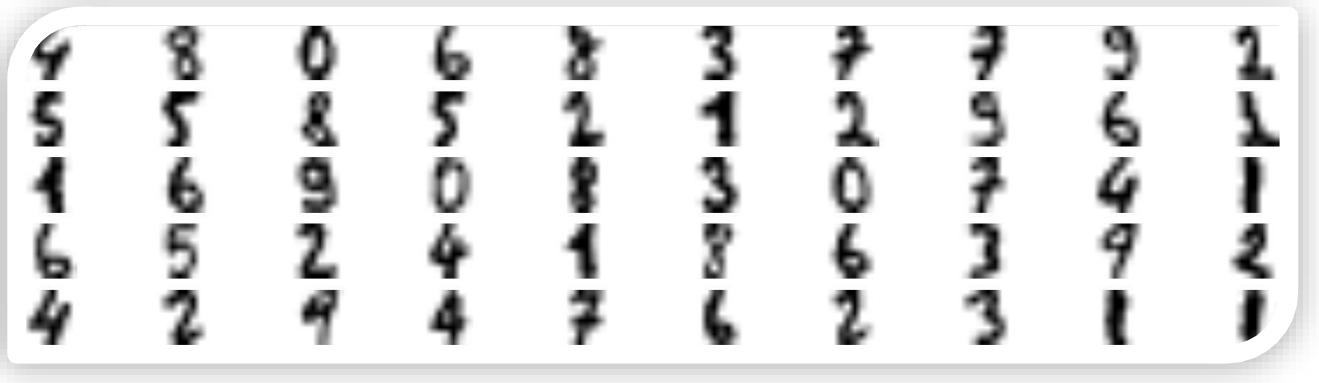


fig: Fifty representative digit images (one per cluster)

Now let's look at each image and manually label it:

y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])

Now we have a dataset with just 50 labeled instances, but instead of being completely

random instances, each of them is a representative image of its cluster. Let's see if the

performance is any better:

>>> log_reg = LogisticRegression()

>>> log_reg.fit(X_representative_digits, y_representative_digits)

>>> log_reg.score(X_test, y_test)

0.9244444444444444

With this approach We jumped from 82.7% accuracy to 92.4%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called label propagation.

7. DBSCAN

This algorithm defines clusters as continuous regions of high density. It is actually quite simple:

For each instance, the algorithm counts how many instances are located within a small distance ε(epsilon) from it. This region is called the instances *ε-neighborhood*.

If an instance has at least min_samples instances in its ε-neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.

All instances in the neighborhood of a core instance belong to the same cluster. This may include other core instances, therefore a long sequence of neighboring core instances forms a single cluster.

Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

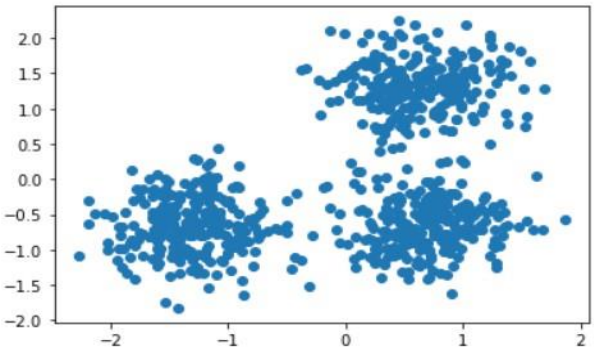
This algorithm works well if all the clusters are dense enough, and they are well separated by low-density regions. The DBSCAN class in Scikit-Learn is as simple to use as you might expect.

Let's test it on the moons dataset

Step-1:Data generation

In [11]:
```
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(
    n_samples=750, centers=centers, cluster_std=0.4, random_state=0
)

X = StandardScaler().fit_transform(X)
```

In [12]:
```
import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1])
plt.show()
```



Step-2: Compute DBSCAN

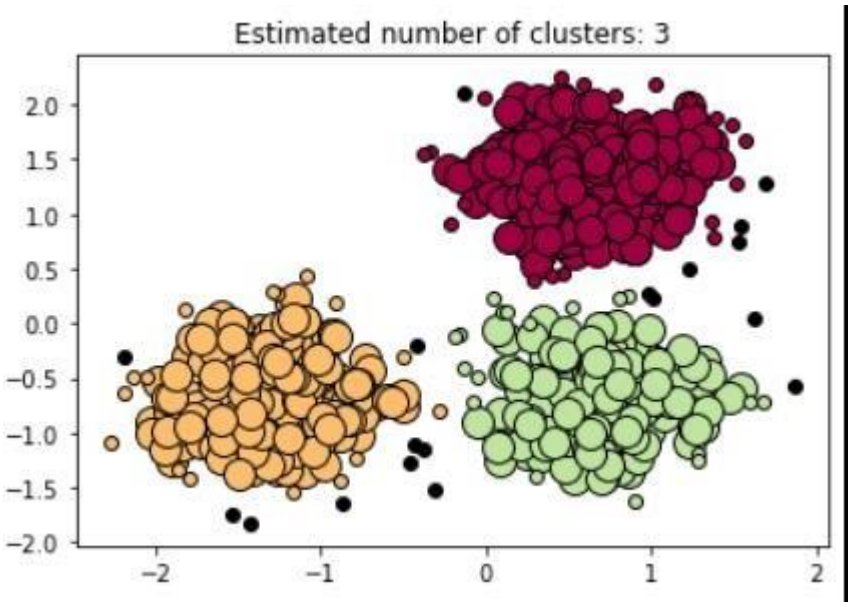
One can access the labels assigned by DBSCAN using the labels_ attribute. Noisy samples are given the label math:-1.

In [13]:
```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn import metrics

db = DBSCAN(eps=0.3, min_samples=10).fit(X)
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print("Estimated number of clusters: %d" % n_clusters_)
print("Estimated number of noise points: %d" % n_noise_)
```

```
Estimated number of clusters: 3
Estimated number of noise points: 18
```

In [14]:

```
unique_labels = set(labels)
core_samples_mask = np.zeros_like(labels, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True

colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = labels == k

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(
        xy[:, 0],
        xy[:, 1],
        "o",
        markerfacecolor=tuple(col),
        markeredgecolor="k",
        markersize=14,
    )

    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(
        xy[:, 0],
        xy[:, 1],
        "o",
        markerfacecolor=tuple(col),
        markeredgecolor="k",
        markersize=6,
    )

plt.title(f"Estimated number of clusters: {n_clusters_}")
plt.show()
```

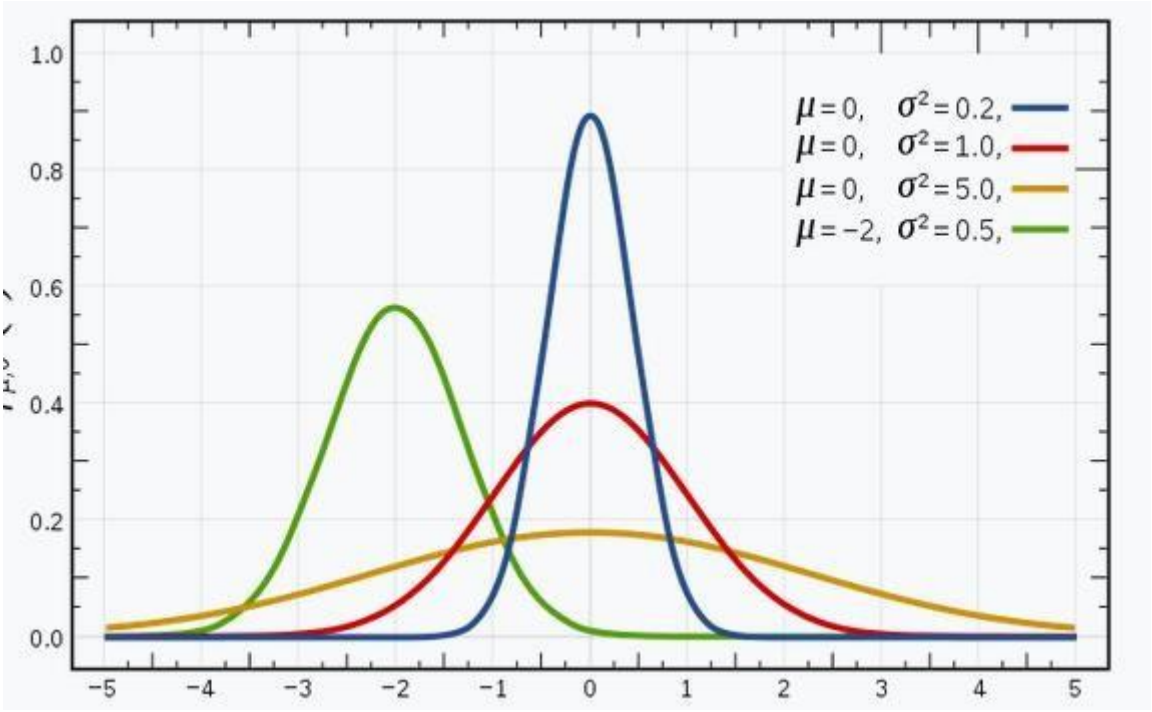


8. Gaussian Mixtures

A *Gaussian mixture model* (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density and orientation.

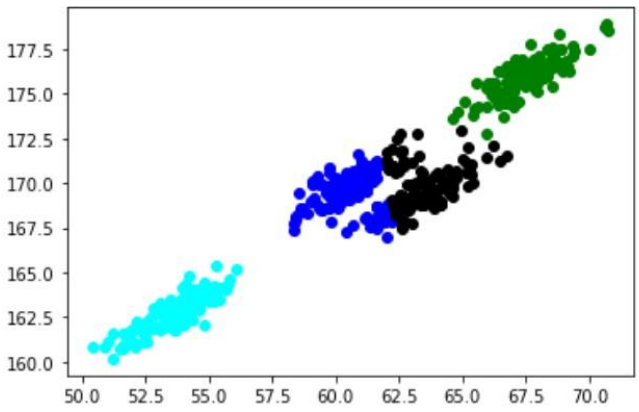
When you observe an instance, you know it was generated from one of the Gaussian distributions, but you are not told which one, and you do not know what the parameters of these distributions are.

The below image has a few Gaussian distributions with a difference in mean (μ) and variance (σ^2). Remember that the higher the σ value more would be the spread:



Gaussian Mixtures Model

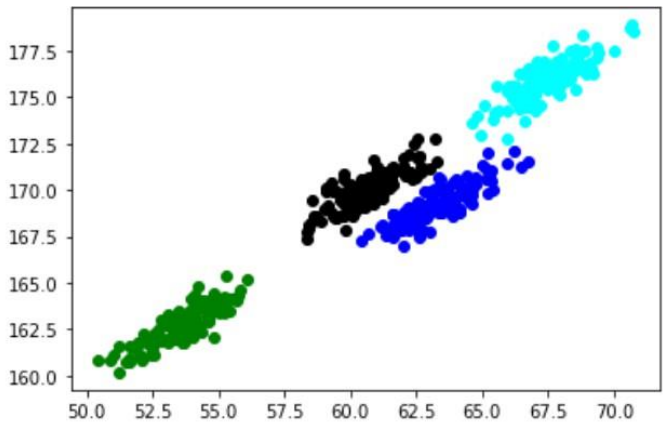
In [4]:
```
#training k-means model
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
data =pd.read_csv('Clustering_gmm.csv')
kmeans = KMeans(n_clusters=4)
kmeans.fit(data)

#predictions from kmeans
pred = kmeans.predict(data)
frame = pd.DataFrame(data)
frame['cluster'] = pred
frame.columns = ['Weight', 'Height', 'cluster']

#plotting results
color=['blue','green','cyan', 'black']
for k in range(0,4):
    data = frame[frame["cluster"]==k]
    plt.scatter(data["Weight"],data["Height"],c=color[k])
plt.show()
```



That's not quite right. The k-means model failed to identify the right clusters. Look closely at the clusters in the center – k-means has tried to build a circular cluster even though the data distribution is elliptical (remember the drawbacks we discussed earlier?).

Let's now build a Gaussian Mixture Model on the same data and see if we can improve on k-means:

```
import pandas as pd
data = pd.read_csv('Clustering_gmm.csv')

# training gaussian mixture model
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=4)
gmm.fit(data)

#predictions from gmm
labels = gmm.predict(data)
frame = pd.DataFrame(data)
frame['cluster'] = labels
frame.columns = ['Weight', 'Height', 'cluster']

color=['blue','green','cyan', 'black']
for k in range(0,4):
    data = frame[frame["cluster"]==k]
    plt.scatter(data["Weight"],data["Height"],c=color[k])
plt.show()
```



Excellent! Those are exactly the clusters we were hoping for. Gaussian Mixture Models have blown k-means out of the water here.

Chapter-II

Dimensionality Reduction: The Curse of Dimensionality, Main Approaches for Dimensionality Reduction, PCA, Using Scikit-Learn, Randomized PCA, Kernel PCA

……..

1. The Curse of Dimensionality

Curse of Dimensionality refers to a set of problems that arise when working with high-dimensional data.

The dimension of a dataset corresponds to the number of attributes/features that exist in a dataset.

A dataset with a large number of attributes, generally of the order of a hundred or more, is referred to as high dimensional data.

Some of the difficulties that come with high dimensional data manifest during analyzing or visualizing the data to identify patterns, and some manifest while training machine learning models.

The difficulties related to training machine learning models due to high dimensional data is referred to as 'Curse of Dimensionality'.

The popular aspects of the curse of dimensionality; 'data sparsity' and 'distance concentration' are discussed in the following sections.

Solutions to Curse of Dimensionality:

One of the ways to reduce the impact of high dimensions is to use a different measure of distance in a space vector. One could explore the use of *cosine similarity* to replace Euclidean distance. Cosine similarity can have a lesser impact on data with higher dimensions. However, use of such method could also be specific to the required solution of the problem.

Other methods:

Other methods could involve the use of reduction in dimensions. Some of the techniques that can be used are:

1. Forward-feature selection: This method involves picking the most useful subset of features from all given features.

2. PCA/t-SNE: Though these methods help in reduction of number of features, but it does not necessarily preserve the class labels and thus can make the interpretation of results a tough task.

……

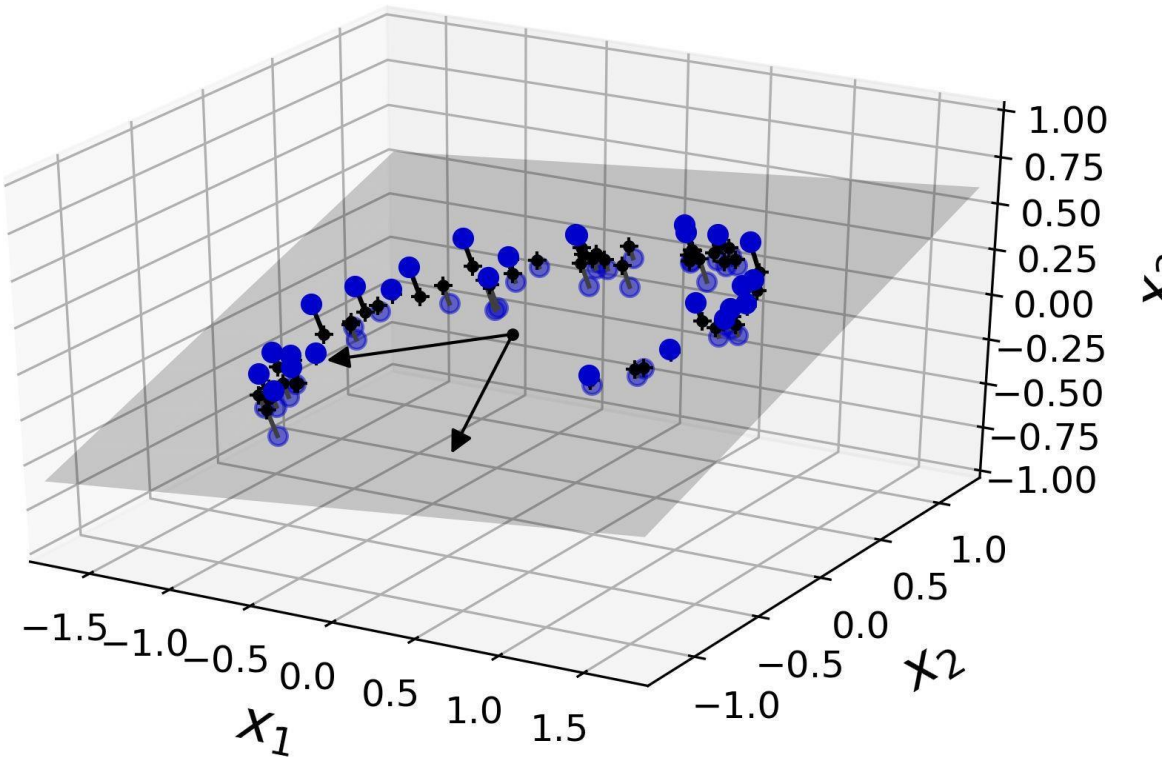
2. Main Approaches for Dimensionality Reduction

The two main approaches to reducing dimensionality: projection and Manifold Learning

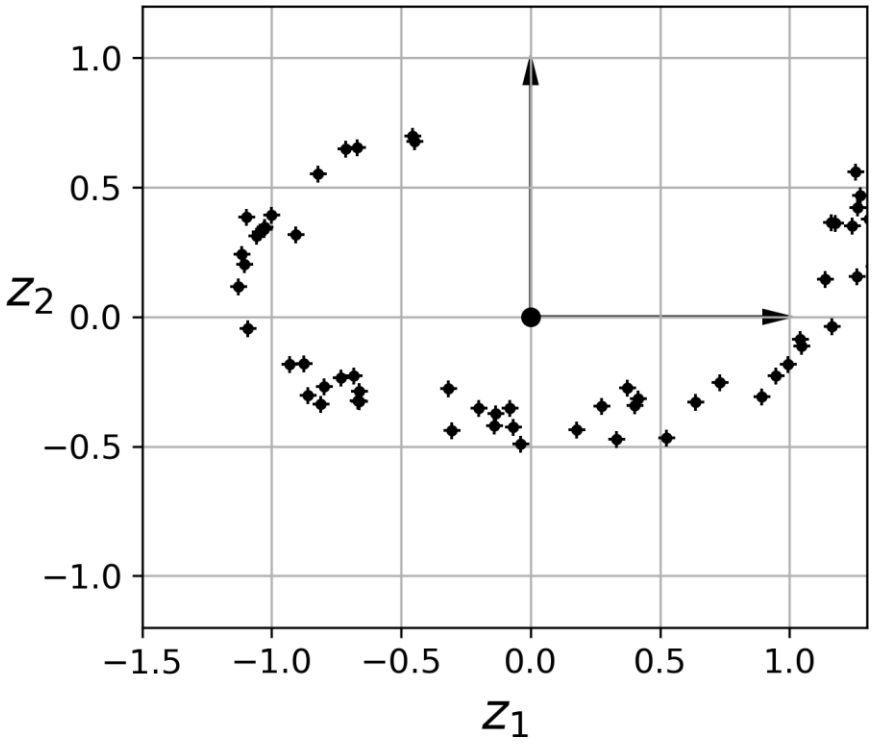
a) Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, all training instances actually lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds very abstract, so let us look at an example.

In Below Figure you can see a 3D dataset represented by the circles.



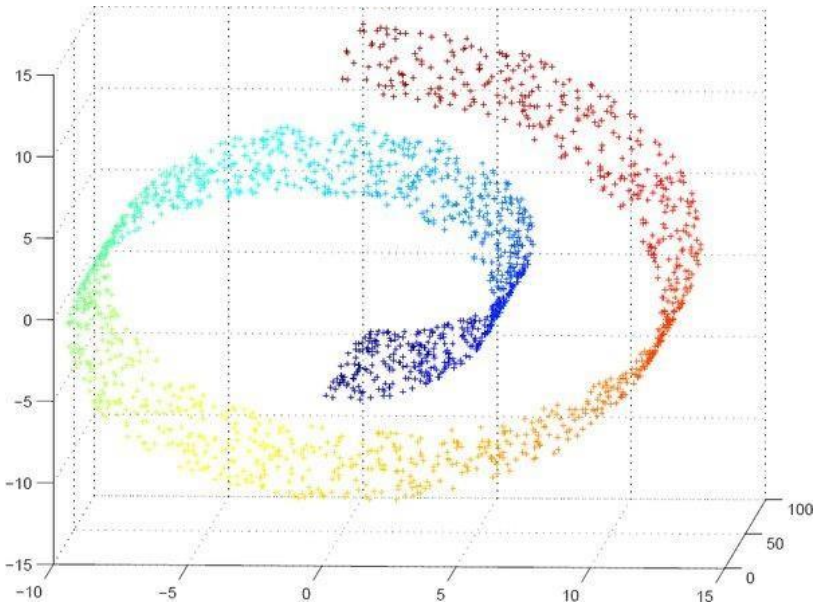
Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space. Now if we project every training instance perpendicularly onto this subspace (as represented by the short lines connecting the instances to the plane), we get the new 2D dataset shown in below



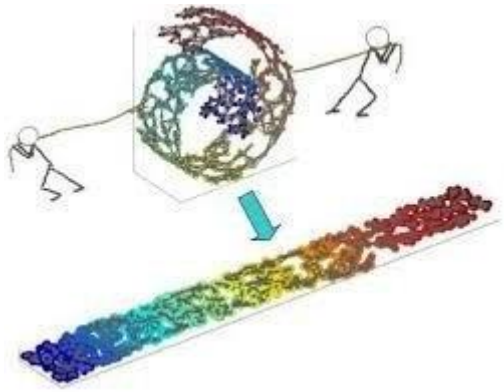
However, projection is not always the best approach to dimensionality reduction.

b) Manifold Learning

- Manifold learning is a type of non-linear dimensionality reduction process.
- It is believed that many datasets has an artificially high dimensionality.
- Suppose your data is 3-dimensional. When visualizing the data, this is what you get:



- It is clear that the data here is simply a 2d plane that is twisted into a 3d space.
- Manifold learning tries to unwrap these folds.

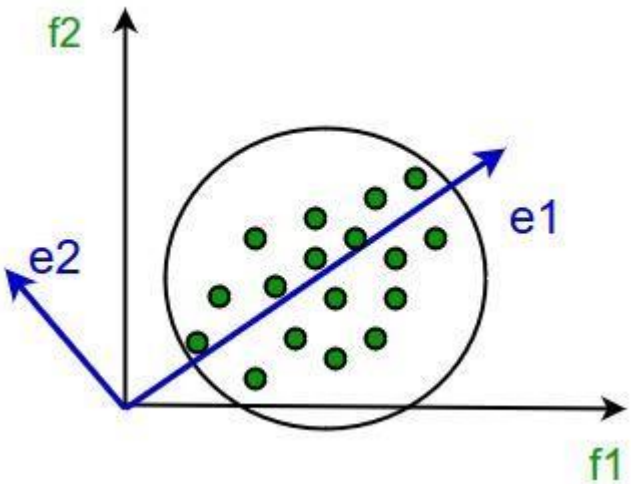


Higher dimensionality data is usually harder to learn on, so manifold learning is just a process to make that kind of data easier to use.

3. Principal Components Analysis

This method was introduced by Karl Pearson. It works on a condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the. lower dimensional space should be maximum. It involves the following steps:

- Construct the covariance matrix of the data.
- Compute the eigenvectors of this matrix.
- Eigenvectors corresponding to the largest eigenvalues are used to reconstruct a large fraction of variance of the original data.



Hence, we are left with a lesser number of eigenvectors, and there might have been some data loss in the process. But, the most important variances should be retained by the remaining eigenvectors.

Advantages of Dimensionality Reduction

- It helps in data compression, and hence reduced storage space.
- It reduces computation time.
- It also helps remove redundant features, if any.

Disadvantages of Dimensionality Reduction

- It may lead to some amount of data loss.
- PCA tends to find linear correlations between variables, which is sometimes undesirable.
- PCA fails in cases where mean and covariance are not enough to define datasets.
- We may not know how many principal components to keep- in practice, some thumb rules are applied.

Steps in PCA

Step 1. Data

We consider a dataset having n features or variables denoted by $X_1, X_2, \ldots, X_n$. Let there be N examples. Let the values of the i-th feature X_i be $X_{i1}, X_{i2}, \ldots, X_{iN}$

Features	Example 1	Example 2	$\cdots$	Example N
X_1	X_{11}	X_{12}	$\cdots$	X_{1N}
X_2	X_{21}	X_{22}	$\cdots$	X_{2N}
$\vdots$				
X_i	X_{i1}	X_{i2}	$\cdots$	X_{iN}
$\vdots$				
X_n	X_{n1}	X_{n2}	$\cdots$	X_{nN}

Data for PCA

Step 2. Compute the means of the variables

We compute the mean $\bar{X}_i$ of the variable X_i:

$$\bar{X}_i = \frac{1}{N}(X_{i1} + X_{i2} + \cdots + X_{iN}).$$

Step 3. Calculate the covariance matrix

Consider the variables X_i and X_j (i and j need not be different). The covariance of the ordered pair (X_i, X_j) is defined as[1]

$$\text{Cov}\,(X_i, X_j) = \frac{1}{N-1}\sum_{k=1}^{N}(X_{ik} - \bar{X}_i)(X_{jk} - \bar{X}_j).$$

We calculate the following $n \times n$ matrix S called the covariance matrix of the data. The element in the i-th row j-th column is the covariance $\text{Cov}\,(X_i, X_j)$:

$$S = \begin{bmatrix} \text{Cov}\,(X_1, X_1) & \text{Cov}\,(X_1, X_2) & \cdots & \text{Cov}\,(X_1, X_n) \\ \text{Cov}\,(X_2, X_1) & \text{Cov}\,(X_2, X_2) & \cdots & \text{Cov}\,(X_2, X_n) \\ \vdots & & & \\ \text{Cov}\,(X_n, X_1) & \text{Cov}\,(X_n, X_2) & \cdots & \text{Cov}\,(X_n, X_n) \end{bmatrix}$$

PCA will provide a mechanism to recognize this geometric similarity through algebraic means.

The covariance matrix S is a symmetric matrix and According to Spectral Theorem(Spectral Decomposition)

If A is symmetric (meaning $A^T = A$), then A is orthogonally diagonalizable and has only real eigenvalues. In other words, there exist real numbers $\lambda_1, \ldots, \lambda_n$ (the eigenvalues) and orthogonal, non-zero real vectors $\vec{v}_1, \ldots, \vec{v}_n$ (the eigenvectors) such that for each $i = 1, 2, \ldots, n$:

$$A\vec{v}_i = \lambda_i \vec{v}_i.$$

Here we call $\vec{v}_i$ as Eigen Vector and λ_i as the corresponding Eigen Value and A as the covariance matrix.

Step 4 : Inferring the Principal components from Eigen Values of the Co Variance Matrix

From Spectral theorem we infer

let $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_m \geq 0$ be the eigenvalues of S (in decreasing order) with corresponding orthonormal eigenvectors $\vec{u}_1, \ldots, \vec{u}_m$. These eigenvectors are called the *principal components* of the data set.

The Most Significant Principal Component is the Eigen vector corresponding to the largest Eigen Value.

the trace of S is the sum of the diagonal entries of S, which is the sum of the variances of all m variables. Let's call this the *total variance*, T of the data. On the other hand, the trace of a matrix is equal the sum of its eigenvalues, so $T = \lambda_1 + \ldots + \lambda_m$.

The following interpretation is fundamental to PCA:

- The direction in $\mathbb{R}^m$ given by $\vec{u}_1$ (the first principal direction) "explains" or "accounts for" an amount λ_1 of the total variance, T. What fraction of the total variance? It's $\frac{\lambda_1}{T}$. And similarly, the second principal direction $\vec{u}_2$ accounts for the fraction $\frac{\lambda_2}{T}$ of the total variance, and so on.
- Thus, the vector $\vec{u}_1 \in \mathbb{R}^m$ points in the most "significant" direction of the data set.
- Among directions that are orthogonal to $\vec{u}_1$, $\vec{u}_2$ points in the most "significant" direction of the data set.
- Among directions orthogonal to both $\vec{u}_1$ and $\vec{u}_2$, the vector $\vec{u}_3$ points in the most significant direction, and so on.

Step 5: - Projecting the data using the Principal Components

The projection matrix is obtained by selected Eigen vectors(k<d) numbers. The original dataset is transformed via the projection matrix to obtain a reduced k dimension subspace of original dataset. (below k is denoted as p)

Additional: If your interested in learning numerical problem check the following links

Link-1: https://www.gatevidyalay.com/tag/principal-component-analysis-numerical-example/

Link-2: https://www.vtupulse.com/machine-learning/principal-component-analysis-solved-example/

Link-3 Video: https://www.youtube.com/watch?v=MLaJbA82nzk (Recommended)

Link-4 Vide: https://www.youtube.com/watch?v=n7npKX5zIWI

4. Using Scikit-Learn

Scikit-Learn's PCA class implements PCA using SVD decomposition just like we did before. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
import numpy as np
from sklearn.decomposition import PCA
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
pca = PCA(n_components=2)
pca.fit(X)

print(pca.explained_variance_ratio_)

print(pca.singular_values_)
```

```
[0.99244289 0.00755711]
[6.30061232 0.54980396]
```

5. Randomized PCA

If you set the svd_solver hyperparameter to "randomized", Scikit-Learn uses a stochastic algorithm called *Randomized PCA* that quickly finds an approximation of the first *d* principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when *d* is much smaller than *n*:

```
pca = PCA(n_components=2, svd_solver='randomized')
pca.fit(X)

print(pca.explained_variance_ratio_)

print(pca.singular_values_)
```

```
[0.99244289 0.00755711]
[6.30061232 0.54980396]
```

By default, svd_solver is actually set to "auto": Scikit-Learn automatically uses the randomized PCA algorithm if *m* or *n* is greater than 500 and *d* is less than 80% of *m* or *n*, or else it uses the full SVD approach. If you want to force Scikit-Learn to use full SVD, you can set the svd_solver hyperparameter to "full"

```
In [7]: pca = PCA(n_components=2, svd_solver='full')
        pca.fit(X)

        print(pca.explained_variance_ratio_)

        print(pca.singular_values_)
```

```
[0.99244289 0.00755711]
[6.30061232 0.54980396]
```

Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko et al. 2009, depending on the shape of the input data and the number of components to extract.

It can also use the scipy.sparse.linalg ARPACK implementation of the truncated SVD.

svd_solver : *{'auto', 'full', 'arpack', 'randomized'}, default='auto'*

If auto :

The solver is selected by a default policy based on `X.shape` and `n_components`: if the input data is larger than 500x500 and the number of components to extract is lower than 80% of the smallest dimension of the data, then the more efficient 'randomized' method is enabled. Otherwise the exact full SVD is computed and optionally truncated afterwards.

If full :

run exact full SVD calling the standard LAPACK solver via `scipy.linalg.svd` and select the components by postprocessing

If arpack :

run SVD truncated to n_components calling ARPACK solver via `scipy.sparse.linalg.svds`. It requires strictly 0 < n_components < min(X.shape)

If randomized :

run randomized SVD by the method of Halko et al.

New in version 0.18.0.

6. Kernel PCA

Kernel PCA a mathematical technique that implicitly maps instances into a very high-dimensional space (called the *feature space*), enabling nonlinear classification and regression with Support Vector Machines.

A linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the *original space*. It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called *Kernel PCA (kPCA).6* It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

For example, the following code uses Scikit-Learn's KernelPCA class to perform kPCA with an RBF kernel

```
In [9]: from sklearn.decomposition import KernelPCA
        rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
        X_reduced = rbf_pca.fit_transform(X)
        X_reduced
```

```
Out[9]: array([[ 0.35376229, -0.21816667],
               [ 0.51755714, -0.05228295],
               [ 0.65529111,  0.27044962],
               [-0.35376229, -0.21816667],
               [-0.51755714, -0.05228295],
               [-0.65529111,  0.27044962]])
```