MET CS 688
Clustering III

Leila Ghaedi – Fall 2024

Creator: cemagraphics | Credit: Getty Images/iStockphoto

Types of Clustering Algorithms

- K-representative Clustering, Partitioning
- Density Based Clustering
- Hierarchical Clustering
- Probabilistic Clustering

Probability vs Likelihood

- Probability means what is the chance of observing *X* in the given sample dataset. This means that in a given dataset, what is the chance of observing *X*?

- Likelihood means given the observed subset *X* of a dataset, what is the best distribution parameters, that fit the given *X*.

- Randy Gallistel provides a good comparison as follows: ”Probability attaches to possible results; likelihood attaches to hypotheses“.

Probabilistic model-based clustering

- In most of the cluster analysis methods we have discussed so far, each data object can be assigned to only one of the clusters.
- This kind of strict cluster assignments are required in some applications, such as assigning customers to marketing strategies.
- In some other applications, this rigid requirement may not be desirable. In this section, we demonstrate the need for fuzzy or flexible cluster assignment in some applications and introduce a general method to compute probabilistic clusters and assignments.
- In probabilistic clustering the assignment of points to clusters is “soft”, in the sense that the membership of a data point x in a cluster Ck is given as a probability, denoted by pk(x).

Probabilistic model-based clustering Use Cases

Clustering product reviews:

Imagine that an e-commerce company has an online store, where customers not purchase online and create reviews of products.

- Not every product receives reviews.
- Some products may have many reviews.
- A review may involve multiple products.

In this context, a group of products, services, or issues that are highly related could be clustered together.

Assigning a review to one cluster exclusively would not work well for this task.

Suppose there is a cluster for “cameras” and another for “computers.” What if a review talks about the compatibility between a camera and a computer? The review is related to both clusters; however, it does not exclusively belong to either cluster.

Maximum Likelihood Estimation (MLE) Approach

- Maximum Likelihood Estimation is an approach to estimating the distribution of data, and an algorithm that implements it is Expectation Maximization.

- For example, we have a small part of a dataset, and the original dataset has a distribution (e.g. Gaussian distribution). We don't know what are the parameters (mean and standard deviation) of that Gaussian distribution, because there could be infinite numbers of Gaussian distributions. The MLE is a procedure that tries to identify the mean, and standard deviation, by using the sample dataset and constructing a distribution that is the closest match to the original dataset distribution.

- The *goal* of MLE is to find the best distribution parameters that fit the original dataset (population).

Maximum Likelihood Estimation (MLE) Approach

- The maximum likelihood estimation method aims to solve the following optimization problem, which says that we should choose the best weight vector w that maximizes the likelihood of the training set. The intuition is that we want to find the optimal model parameter (i.e., the weight vector w) so that there is the highest "chance" (i.e., the likelihood or the probability) of observing the entire training set.
- It means we calculate the inverse of minimization, which is equal to maximization. In simple words, minimizing the error is equivalent to maximizing the log-likelihood.

Expectation Maximization

- Latent variable problem: Although MLE does not have access to the original dataset, it assumes the dataset is complete or fully observed.

- Nevertheless, part of the dataset could be missing in the observation subset that is used by MLE. Those missing parts could construct parameters that are known as latent variables. Here, latent variables refer to parameters that do not exist in the observed dataset.

- Expectation Maximization (EM) algorithm implements the MLE, when there is a latent variable in the dataset.

- In summary, EM algorithm is recommended to use for MLE implementation when we are dealing with missing data in our observed dataset.

EM Algorithm

- The objective of EM, like the MLE objective, is to find unknown parameters that find the best distribution fit to the original dataset (approximate maximum likelihood).
- E-step makes an initial guess of parameters for the expected distribution.
- M-step starts after the E-step, and when newly observed data will be fed into the model. In this step, the EM algorithm tweaks the estimated parameters (from E-Step) to cover newly observed data as well.
- From M-step, the process will be repeated until the created distribution does not change in E-step or M-step and it reaches a stable state (converged), or a maximum threshold of iteration reaches.

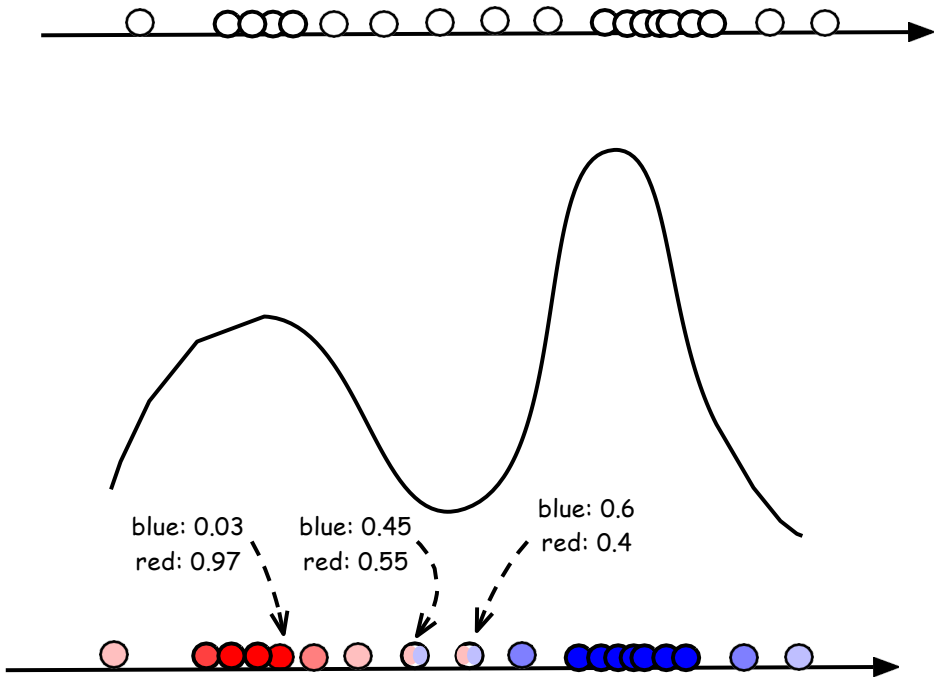
Probabilistic Clustering

Described clustering methods assume each data point will be assigned to one cluster and a single data point cannot participate in more than one cluster.

Probabilistic clustering allows a data point to be a member of more than one cluster. This is called soft clustering or probabilistic clustering.

- This type of clustering assumes the dataset is a mixture of two or more probability distributions. Each distribution presents a cluster, and distribution is described by a density function with a weight (mixing coefficient).

- In other words, mixture model clustering methods assume that a dataset can be divided into probability distributions (clusters) and clusters are following a known distribution, e.g. Gaussian. Therefore, we can say a dataset is composed of a combination of different distributions. A mixture model is a weighted sum of distributions.



On dimensional data distribution on top and its distribution on the bottom.

How to convert a dataset into a combination of different distributions?

Maximum Likelihood Estimation

Problem: We don't have access to the entire dataset, but we have access to a portion of the dataset, and by using this available portion, we would like to estimate a distribution that the entire dataset fits in it.

We use MLE to determine the best distribution that presents the entire dataset, by using the sample dataset.

The MLE method of GMM tries to identify the mean, μ, and standard deviation, δ, to construct a distribution which is the closest match to the dataset.

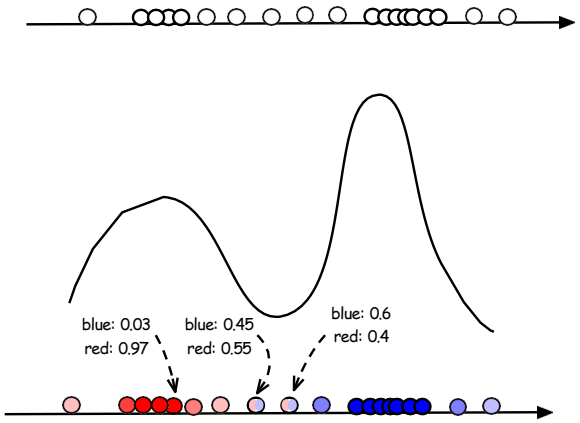
Gaussian Mixture Model

- A Gaussian mixture model (GMM) is a probabilistic model that represents data as a mixture of Gaussian distributions. GMMs are used for clustering, density estimation, and dimensionality reduction. They are a powerful algorithm for discovering underlying patterns in a dataset.
- GMM is an example of a probabilistic model-based clustering. That is, we assume that the probability density function of each cluster follows a Gaussian distribution. Suppose there are k clusters. The two parameters for the probability density function of each cluster are center, μ , and standard deviation.

Gaussian Mixture Model

- If we assume all result clusters should have a gaussian distribution, we can use Gaussian Mixture Model (GMM) to cluster the dataset.

- As it has been described each cluster is modeled by a distribution. The Gaussian distribution is defined by mean (correspond to center of the distribution) and standard deviation (how spread are the data around the center).

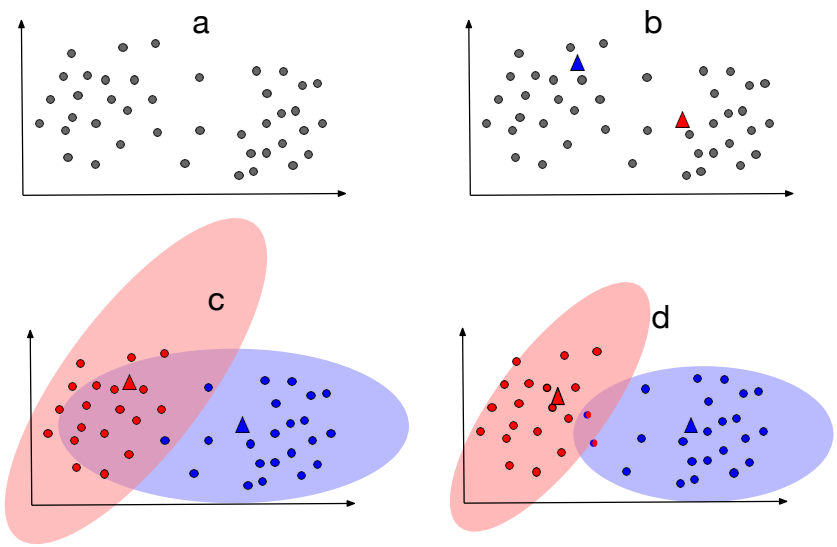


Gaussian Mixture Model

- To identify distributions (clusters) of a dataset which constitute the clustering, the clustering is defined as a parameter estimation problem. As we have described, we are seeking to identify the parameter and GMM uses EM algorithm to accomplish this task.

- In GMM each cluster is characterized by mean vector, covariance matrix and associated probability.

- The goal of GMM is to represent each sub-dataset as its own distribution (or mixture component). The entire dataset could then be represented as a mixture of *n* Gaussian distributions, and *n* is the number of clusters.



EM is the main challenge and its computational complexity is similar to k-mean, O(m.n.i)

Expectation step: This step compares the distances (in probabilities) from all other points to **the two selected points**. Then based on the calculated **probabilities**, each data point will be assigned to one or more clusters.

Maximization step: This step **moves the mean points**, i.e. triangles, into the center (mean) of each cluster. Afterward, again the algorithm reassigns data points based on the new locations of means. This process iteratively continues until there will be no changes in mean points, they stay at their location, and then both clusters were finalized.

In other words, the EM algorithm stops the iteration, either by observing that mean and SD are not moving or by reaching a predefined threshold, which a user can specify this threshold.

GMM Example

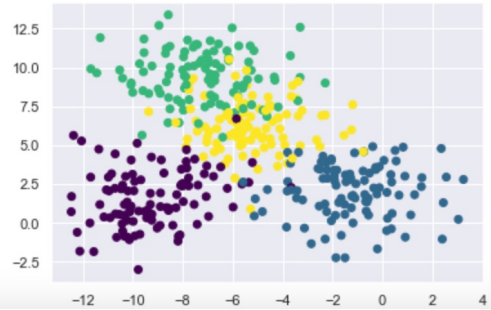
- https://www.kaggle.com/code/vipulgandhi/gaussian-mixture-models-clustering-explained

GMM Example 1

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns; sns.set()
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn import mixture
```

In [2]:
```
#Creating the test data for clustering
# y is the actual label for blobs
# the gmm model is agnostic to y

X, y = make_blobs(n_samples=400, centers=4, n_features=4, cluster_std=1.8, random_state=820)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
```

Out[2]: <matplotlib.collections.PathCollection at 0x7f9671642dc0>



GMM Example 1

In [3]:
```
gmm = mixture.GaussianMixture(n_components=4)
gmm.fit(X)
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=10, cmap='viridis')
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x7f96717092e0>
```

GMM Example 1

Gaussian mixture model contains a probabilistic model under the hood, it is also possible to find probabilistic cluster assignments. In Scikit-Learn this is done using the predict_proba method.

```
In [4]: probs = (gmm.predict_proba(X)).round(3)
        probs[100:120]

Out[4]: array([[1.   , 0.   , 0.   , 0.   ],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 0.   , 0.998, 0.002],
               [0.   , 0.   , 0.   , 1.   ],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 0.   , 0.995, 0.005],
               [0.   , 0.   , 1.   , 0.   ],
               [0.962, 0.   , 0.   , 0.038],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 0.   , 0.996, 0.004],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 1.   , 0.   , 0.   ],
               [0.   , 1.   , 0.   , 0.   ],
               [0.998, 0.   , 0.   , 0.002],
               [0.   , 1.   , 0.   , 0.   ],
               [0.995, 0.   , 0.002, 0.003],
               [0.887, 0.   , 0.001, 0.111]])
```

GMM Example 1

In [5]:
```
# visualize the effect of the probabilities
size = 10 * probs.max(1) ** 2  # square emphasizes differences
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size)
```

Out[5]:
```
<matplotlib.collections.PathCollection at 0x7f96507d84c0>
```

GMM Example 2

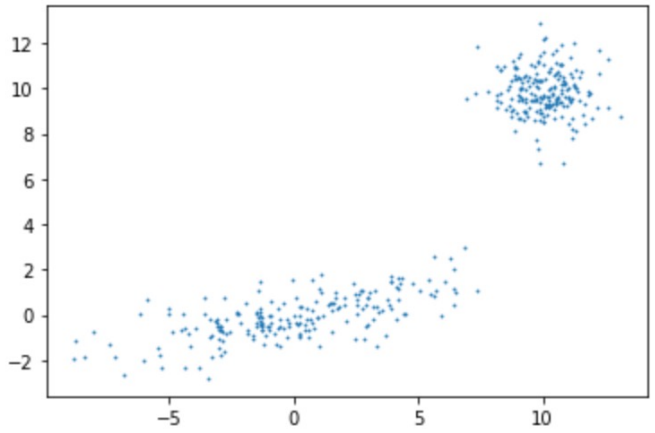
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from sklearn import mixture
```

In [2]:
```
n_samples=200
# generate random sample, two components
np.random.seed(70)
# generate spherical data centered on (20, 20)
shifted_gaussian  =  np.random.randn(n_samples,  2) + np.array([10, 10])
# generate zero centered stretched Gaussian data
C = np.array([[0., -0.7], [3.5, .7]])
stretched_gaussian = np.dot(np.random.randn(n_samples,2), C)

# concatenate the two datasets into the final training set
X_train = np.vstack([shifted_gaussian, stretched_gaussian])
```

GMM Example 2

In [3]:

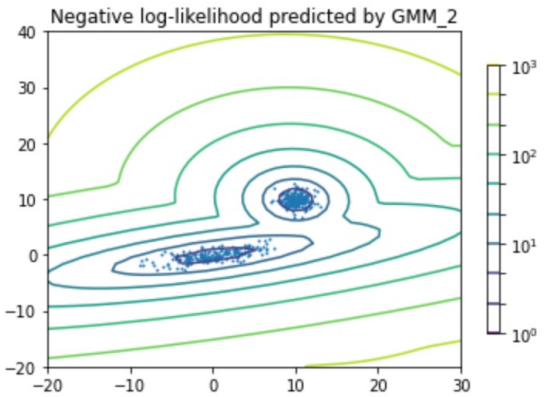
```
# fit a Gaussian Mixture Model with two components
gmm_2 = mixture.GaussianMixture(n_components=2, covariance_type='full')
gmm_2.fit(X_train)

plt.scatter(X_train[:,0], X_train[:,1], 0.8)
plt.show()
```



GMM Example 2

In [4]:
```
#display predicted scores by the model as a contour plot
x=np.linspace(-20, 30)
y=np.linspace(-20, 40)
X,Y=np.meshgrid(x,y)
XX = np.array([X.ravel(), Y.ravel()]).T
Z = -gmm_2.score_samples(XX)
Z = Z.reshape(X.shape)
CS = plt.contour(X,Y,Z, norm=LogNorm(vmin=1.0, vmax=1000.0),levels=np.logspace(0,3,10))
CB = plt.colorbar(CS, shrink=0.8, extend='both')
plt.scatter(X_train[:,0],  X_train[:,1],  .9)
plt.title('Negative log-likelihood predicted by GMM_2')
plt.axis('tight')
```

Out[4]: (-20.0, 30.0, -20.0, 40.0)



Fuzzy C-Mean

- It is very similar to k-mean and starts by defining some random points in the space. This algorithm operates as follows:

1. Similar to k-mean it starts by randomly selecting *c* data points, as centroid of clusters, and *c* is a hyper parameter which defines the number of clusters.

2. It calculates the fuzzy membership of each node in a cluster and assign the probabilities of membership to each node.

3. Next, it computes the centroid of each clusters and move the centroid points toward the center of each cluster.

4. Step (2) and (3) repeats iteratively until centroid points do not change or the maximum number of iterations (given by the user) reached.

Fuzzy C-Mean Example

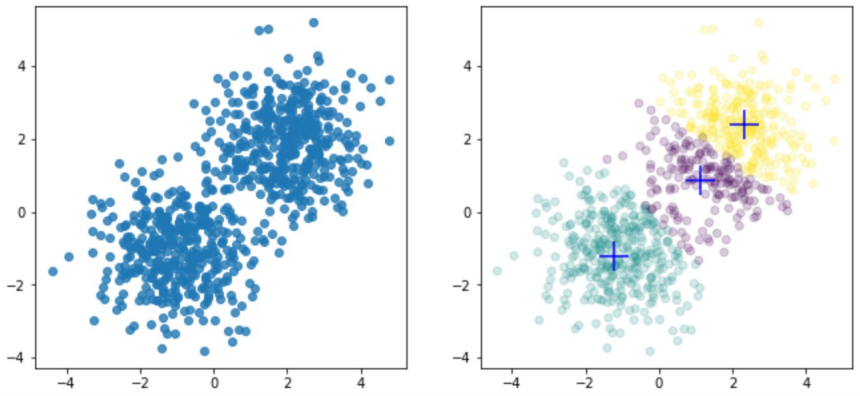
- pip install fuzzy-c-means

In [1]:
```
%matplotlib  inline
import  numpy  as  np
from fcmeans import FCM
from matplotlib import pyplot as plt
```

In [2]:
```
n_samples = 400
X = np.concatenate((np.random.normal((-1,-1), size=(n_samples,2)),
                    np.random.normal((2,2), size=(n_samples,2))   ))
fcm = FCM(n_clusters=3)
fcm.fit(X)

# outputs
fcm_centers = fcm.centers
fcm_labels = fcm.predict(X)
```

Fuzzy C-Mean Example

In [3]:
```
# outputs
fcm_centers = fcm.centers
fcm_labels = fcm.predict(X)
# plot  result
f, axes = plt.subplots(1, 2, figsize=(11,5))
axes[0].scatter(X[:,0], X[:,1], alpha=.8)
axes[1].scatter(X[:,0], X[:,1], c=fcm_labels, alpha=.2)
axes[1].scatter(fcm_centers[:,0],fcm_centers[:,1],
                marker="+", s=500,  c='b')

plt.savefig('fuzzy-output.jpg')
plt.show()
```



Fuzzy C-Mean Example

In [4]:
```
# look at soft (fuzzy) predictions
soft_prediction_labels = fcm.soft_predict(X)
soft_prediction_labels
```

Out[4]:
```
array([[0.01565674, 0.97882972, 0.00551354],
       [0.09677115, 0.85863035, 0.0445985 ],
       [0.14714758, 0.78989339, 0.06295903],
       ...,
       [0.18521577, 0.03695805, 0.77782618],
       [0.03300345, 0.00566055, 0.961336  ],
       [0.18697768, 0.05196247, 0.76105985]])
```

Category	Algorithm Name	Input Parameters	Descriptions
Partition based	k-means	1 number of clusters 2 threshold for iteration (optional)	the most used clustering algorithm due to its simplicity
	k-medoids	1 number of clusters 2 threshold for iteration (optional)	very similar to k-mean but more tolerant to outliers
Density based	DBScan	1 epsilon 2 minimum points	can clusters data points with convex shape
	OPTICS	1 minimum points 2 epsilon (search distance)	OPTICS reduces sensitivity of the epsilon parameter (in comparison to DBScan). It can also handles clusters with different densities.
Hierarchical	SLINK (agglomerative)	No parameter required	Both of these methods are computationally in-efficient, but they are baseline algorithms to hierarchical clusterings. For efficient hierarchical clustering check BIRCH and CURE.
	DIANA (divisive)	No parameter required	
Hierarchical for Large Dataset	BIRCH	1 CF-Tree branching factor 2 CF-Tree threshold 3 CF-Tree max. number of entries in leaf node	very scalable and be able to undo what has been done in its previous step. Sometimes CF-Tree can not fit into the memory and we get out of memory error
	CURE	1- number of clusters	Useful to handle large datasets with non-convex shapes.
Probabilistic	GMM	1- number of clusters	enables soft clustering, operates based on the two steps of expectation and maximization
	Fuzzy C-Mean	1- number of clusters	enables soft clustering, operates very similar k-mean

A summary of described clustering algorithms.

+ topic modeling (LSA and LDA)

MET CS 688
Dimensionality Reduction & Data Decomposition

Leila Ghaedi – Fall 2024

Creator: cemagraphics | Credit: Getty Images/iStockphoto

Dimensionality Reduction and Data Decomposition

The Curse of Dimensionality

- The curse of dimensionality is a phenomenon that occurs when analyzing and organizing data in high-dimensional spaces. This can lead to difficulties for machine learning algorithms.
- In machine learning problems that involve learning from a finite number of data samples in a high-dimensional feature space with each feature having a range of possible values, typically an enormous amount of training data is required to ensure that there are several samples with each combination of values.
- In other word, as the number of features or dimensions grows, the amount of data we need to generalize accurately grows exponentially.

Dimensionality Reduction

- Usually, our real-world dataset is too large or too noisy. However, we would like to keep information that can help the machine learning algorithm, but on the other hand, our dataset is too large.

- In these scenarios, a method that can reduce data, while keeping its semantic is very helpful. These methods called data reduction methods.

- Data reduction is the process of transforming the dataset into a smaller dataset while maintaining its integrity to the original dataset. In other words, it is projecting the data from its original dimensional space into a lower dimensional space.

- Data compression methods, or some of the feature engineering methods, could also considered as data reduction techniques.

Dimensionality Reduction

- Dimensionality Reduction, takes a high-dimensional representation of the data, consisting of many features, and finds a new way to represent this data that summarizes the essential characteristics with fewer features.

- A common application for dimensionality reduction is reduction to two dimensions for visualization purposes.

- Transforming data using unsupervised learning can have many motivations. The most common motivations are visualization, compressing the data, and finding a representation that is more informative for further processing.

Principal Component Analysis (PCA)

- One of the simplest and most widely used algorithms for dimensionality reduction is principal component analysis.
- Principal component analysis is a method that rotates the dataset in a way such that the rotated features are statistically uncorrelated.
- This rotation is often followed by selecting only a subset of the new features, according to how important they are for explaining the data.

Principal Component Analysis (PCA)

- Principal component analysis transforms the multidimensional data into a lower dimensional data.
- PCA tries to reduce the number of highly correlated attributes in the dataset.
- It has many application especially in image recognition, such as face recognition, optical character recognition, missing data reconstruction, anomaly detection, and many more applications.

Principal Component Analysis (PCA)

- Suppose that the data to be reduced consist of vectors described by d attributes or dimensions. Principal components analysis (PCA) searches for k orthonormal vectors that can best be used to represent the data, where $k \leq d$.

- Unlike attribute subset selection, which reduces the attribute set size by retaining a subset of the initial set of attributes, PCA “combines” the essence of attributes by creating an alternative, smaller set of variables.

PCA Steps:

1. Standardize the range of continuous variables:

   This involves scaling the variables to have a mean of 0 and a standard deviation of 1. This step is essential to give all variables equal importance in the analysis, as variables with larger scales might dominate the computation.

2. Compute the covariance matrix to identify correlations:

   The covariance matrix is a square matrix that shows how much pairs of variables vary together. A positive covariance indicates a positive relationship, while a negative covariance indicates a negative relationship.

3. Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components:

   The eigenvectors and eigenvalues are derived from the covariance matrix. The eigenvectors represent the directions of maximum variance in the data, and the eigenvalues indicate the magnitude of the variance in those directions. These are crucial for determining the principal components.

4. Create a feature vector to decide which principal components to keep:

   The feature vector is composed of the eigenvectors corresponding to the largest eigenvalues. By sorting the eigenvalues in descending order, you can choose the top-k eigenvectors to form the feature vector.

5. Recast the data along the principal components' axes:

   Finally, you transform the original data by projecting it onto the selected principal components. This results in a new set of variables (principal components) that capture most of the variance in the original data.

PCA Steps:
Covariance Matrix

- Covariance Matrix: $\text{cov}(x, y) = \frac{\sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y)}{n}$
- Covariance is calculated between two-dimensional data.
- When we have more than two dimensions, we need to calculate the covariance between each two. For example, for a three -dimensional data X, Y, and Z will be calculated as follows, which is a symmetric matrix.

$$C = \begin{bmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{bmatrix}$$

PCA Steps:
Eigen Vector and Eigen Values

- As covariance has been calculated the PCA should breaks down the covariance matrix into vectors which describes the direction and magnitude.

- These vectors are called “principal component”. Each dimension is associated with a principal component, i.e. PC1 spans the direction of most variation, PC2 spans the direction of the second most variation and so forth.

- To calculate these principal components the PCA algorithm calculates “eigenvectors” and their corresponding “eigenvalues” from the covariance matrix.

PCA Steps:
Eigen Vector and Eigen Values

Assuming *A* is a matrix $A = \begin{bmatrix} 3 & 2 \\ 3 & -2 \end{bmatrix}$

Its **eigenvector** is *v* $v = \begin{matrix} 2 \\ 1 \end{matrix}$

$$A \,.\, v = \lambda \,.\, v$$

Its **eigenvalue** is lambda. $\lambda = 4$

PCA Example 1

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
```

In [2]:
```
# Load Iris dataset
iris = load_iris()
data = iris.data
target = iris.target
feature_names = iris.feature_names
feature_names
```

Out[2]:
```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

In [3]:
```
print(data)
```

```
[7.9 3.8 6.4 2. ]
[6.4 2.8 5.6 2.2]
[6.3 2.8 5.1 1.5]
[6.1 2.6 5.6 1.4]
[7.7 3.  6.1 2.3]
[6.3 3.4 5.6 2.4]
```

PCA Example 1

In [4]:
```
# Step 1: Standardize the data
standardized_data = (data - np.mean(data, axis=0)) / np.std(data, axis=0)

# Step 2: Apply PCA
pca = PCA()
principal_components = pca.fit_transform(standardized_data)
```

In [5]:
```
# Step 3: Explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = np.cumsum(explained_variance_ratio)

print(f"explained_variance_ratio: {explained_variance_ratio}")
print(f"cumulative_variance_ratio: {cumulative_variance_ratio}")
```

```
explained_variance_ratio: [0.72962445 0.22850762 0.03668922 0.00517871]
cumulative_variance_ratio: [0.72962445 0.95813207 0.99482129 1.        ]
```

PCA Example 1

Setting the number of principal components based on cumulative variance described

In [6]:
```
# Step 4: Determine the number of components
num_components = np.argmax(cumulative_variance_ratio >= 0.95) + 1
print(f"Number of components to retain 99% variance: {num_components}")

# Step 5: Select the top components
selected_components = principal_components[:, :num_components]
```

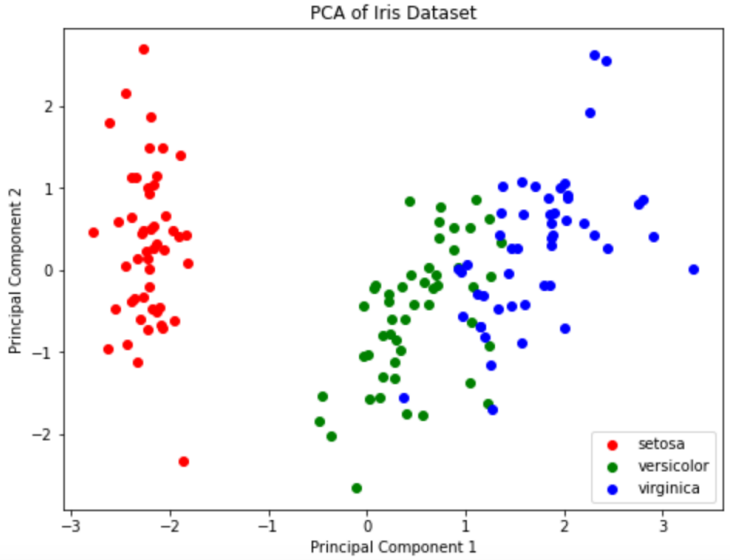
```
Number of components to retain 99% variance: 2
```

In [7]:
```
print(selected_components)
```

```
[-2.6331007  -0.96150673]
[-2.1987406   1.86005711]
[-2.26221453  2.68628449]
[-2.2075877   1.48360936]
[-2.19034951  0.48883832]
[-1.898572    1.40501879]
[-2.34336905  1.12784938]
[-1.914323    0.40885571]
[-2.20701284  0.92412143]
```

PCA Example 1

In [8]:
```
# Visualize the data in 2D using the first two principal components
plt.figure(figsize=(8, 6))
for i, c in zip(range(3), ['red', 'green', 'blue']):
    plt.scatter(selected_components[target == i, 0], selected_components[target == i, 1],
                c=c, label=iris.target_names[i])
plt.title('PCA of Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()
```



PCA Example 2

In [1]:
```
from sklearn.decomposition import PCA
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
target=cancer.target
data= cancer.data
feature_names = cancer.feature_names
feature_names
#print(cancer.data)
```

Out[1]:
```
array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error',
       'fractal dimension error', 'worst radius', 'worst texture',
       'worst perimeter', 'worst area', 'worst smoothness',
       'worst compactness', 'worst concavity', 'worst concave points',
       'worst symmetry', 'worst fractal dimension'], dtype='<U23')
```

PCA Example 2

In [2]:
```
print(data)
```

```
[[1.799e+01 1.038e+01 1.228e+02 ... 2.654e-01 4.601e-01 1.189e-01]
 [2.057e+01 1.777e+01 1.329e+02 ... 1.860e-01 2.750e-01 8.902e-02]
 [1.969e+01 2.125e+01 1.300e+02 ... 2.430e-01 3.613e-01 8.758e-02]
 ...
 [1.660e+01 2.808e+01 1.083e+02 ... 1.418e-01 2.218e-01 7.820e-02]
 [2.060e+01 2.933e+01 1.401e+02 ... 2.650e-01 4.087e-01 1.240e-01]
 [7.760e+00 2.454e+01 4.792e+01 ... 0.000e+00 2.871e-01 7.039e-02]]
```

In [3]:
```
scaler = StandardScaler()
scaler.fit(data)
X_scaled = scaler.transform(data)
```

In [4]:
```
print(X_scaled)
```

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

PCA Example 2

In [5]:
```
# Find out what number of principal components
# describe the data
pca = PCA()
principal_components = pca.fit_transform(X_scaled)

explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = np.cumsum(explained_variance_ratio)

print(f"explained_variance_ratio: {explained_variance_ratio}")
print(f"cumulative_variance_ratio: {cumulative_variance_ratio}")
```

```
explained_variance_ratio: [4.42720256e-01 1.89711820e-01 9.39316326e-02 6.60213492e-02
 5.49576849e-02 4.02452204e-02 2.25073371e-02 1.58872380e-02
 1.38964937e-02 1.16897819e-02 9.79718988e-03 8.70537901e-03
 8.04524987e-03 5.23365745e-03 3.13783217e-03 2.66209337e-03
 1.97996793e-03 1.75395945e-03 1.64925306e-03 1.03864675e-03
 9.99096464e-04 9.14646751e-04 8.11361259e-04 6.01833567e-04
 5.16042379e-04 2.72587995e-04 2.30015463e-04 5.29779290e-05
 2.49601032e-05 4.43482743e-06]
cumulative_variance_ratio: [0.44272026 0.63243208 0.72636371 0.79238506 0.84734274 0.88758796
 0.9100953  0.92598254 0.93987903 0.95156881 0.961366   0.97007138
 0.97811663 0.98335029 0.98648812 0.98915022 0.99113018 0.99288414
 0.9945334  0.99557204 0.99657114 0.99748579 0.99829715 0.99889898
 0.99941502 0.99968761 0.99991763 0.99997061 0.99999557 1.        ]
```

PCA Example 2

In [6]:
```
# Determine the number of components for over 95% variance
num_components = np.argmax(cumulative_variance_ratio >= 0.95) + 1
print(f"Number of components to retain 99% variance: {num_components}")
```

```
Number of components to retain 99% variance: 10
```

In [7]:
```
pca = PCA(n_components=num_components)
# fit PCA model to breast cancer data
pca.fit(X_scaled)
```

Out[7]:
```
PCA
PCA(n_components=10)
```

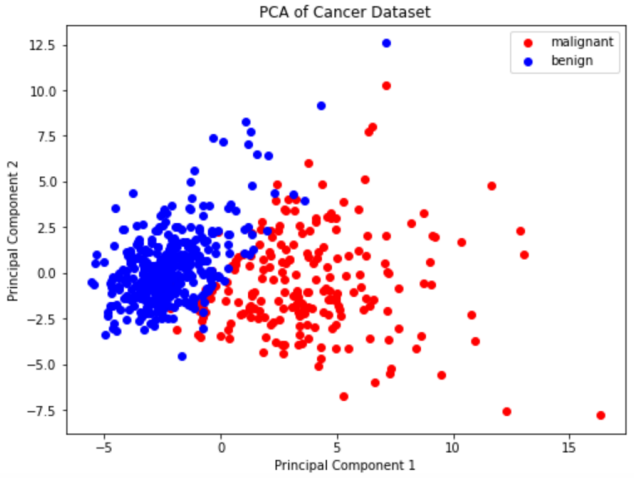
In [8]:
```
selected_components = principal_components[:, :num_components]
```

In [9]:
```
X_pca = pca.transform(X_scaled)
print("Original shape: {}".format(str(X_scaled.shape)))
print("Reduced shape: {}".format(str(X_pca.shape)))
```

```
Original shape: (569, 30)
Reduced shape: (569, 10)
```

PCA Example 2

In [10]:
```
# Visualize the data in 2D using the first two principal components
plt.figure(figsize=(8, 6))
for i, c in zip(range(3), ['red',  'blue']):
    plt.scatter(X_pca[target == i, 0], X_pca[target == i, 1],
                c=c, label=cancer.target_names[i])
plt.title('PCA of Cancer Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()
```

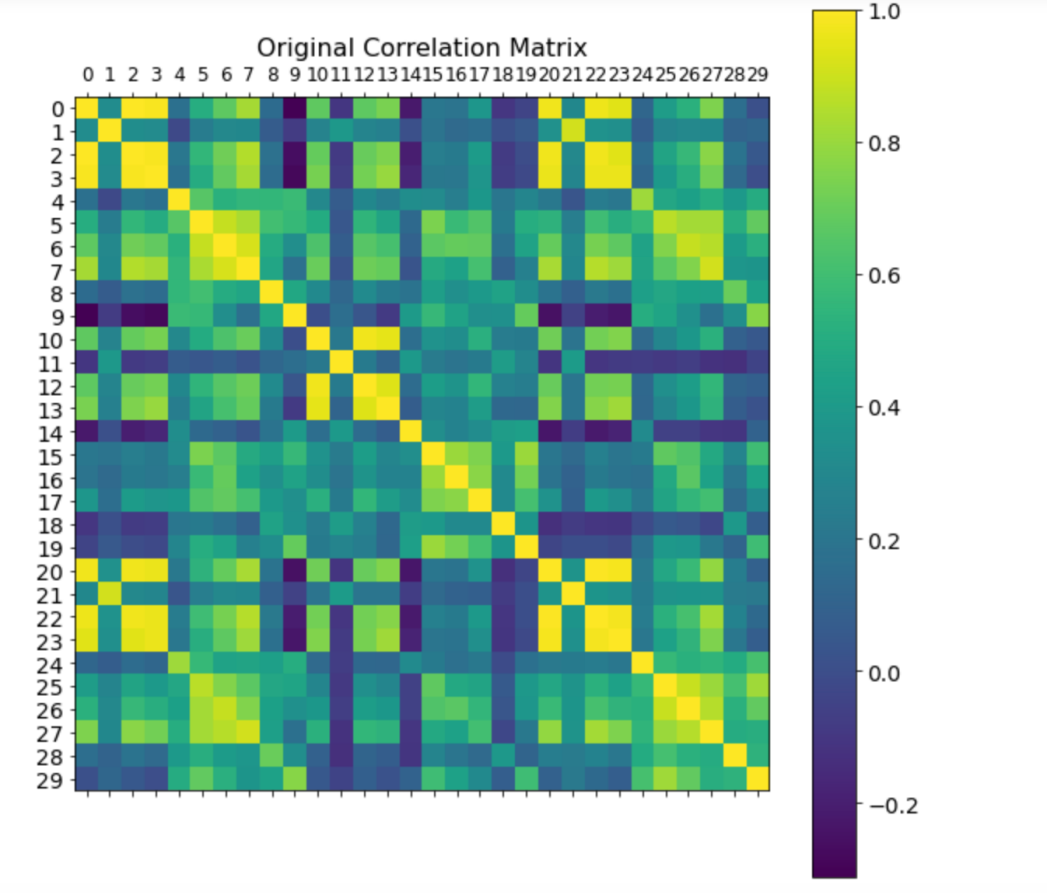


PCA Example 2

In [11]:
```
# create data frames
df = pd.DataFrame(data=data)
PCA_df= pd.DataFrame(data=X_pca)
```

In [12]:
```
f = plt.figure(figsize=(10, 10))
plt.matshow(df.corr(), fignum=f.number)
plt.xticks(range(df.select_dtypes(['number']).shape[1]), df.select_dtypes(['number']).columns, fontsize=12)
plt.yticks(range(df.select_dtypes(['number']).shape[1]), df.select_dtypes(['number']).columns, fontsize=14)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
plt.title('Original Correlation Matrix', fontsize=16);
```

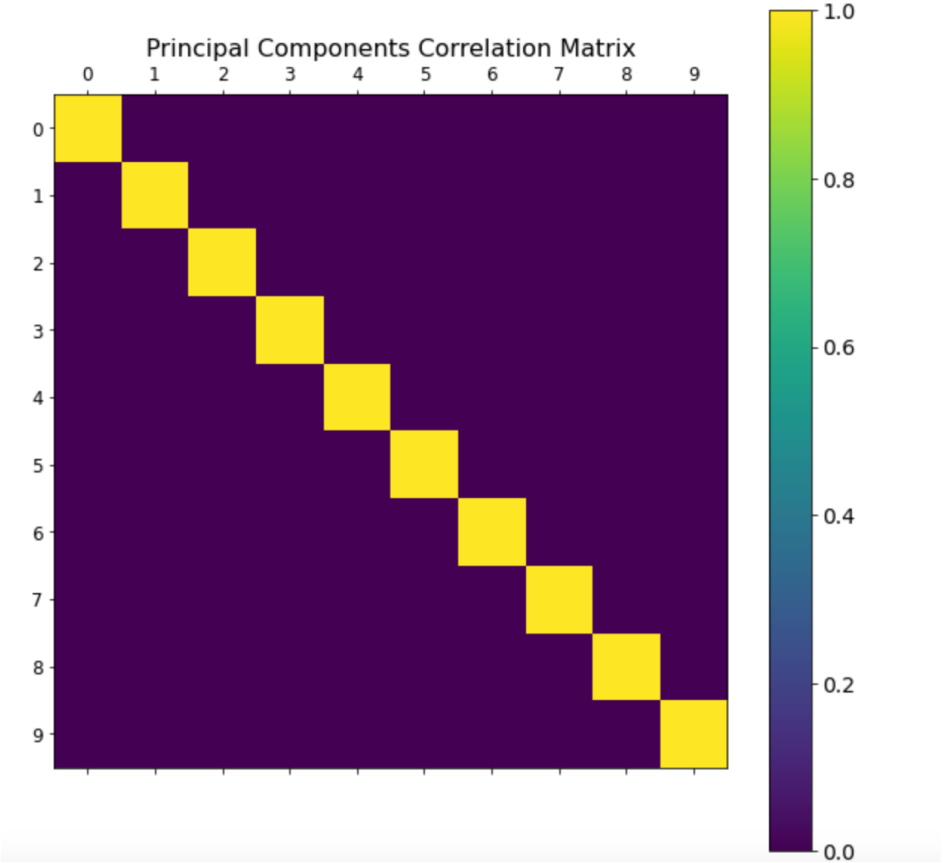
PCA Example 2



PCA Example 2

In [13]:
```
f = plt.figure(figsize=(10, 10))
plt.matshow(PCA_df.corr(), fignum=f.number)
plt.xticks(range(PCA_df.select_dtypes(['number']).shape[1]), PCA_df.select_dtypes(['number']).columns, fontsize=12)
plt.yticks(range(PCA_df.select_dtypes(['number']).shape[1]), PCA_df.select_dtypes(['number']).columns, fontsize=12)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
plt.title('Principal Components Correlation Matrix', fontsize=16);
```

PCA Example 2



Linear Discriminant Analysis (LDA)

- Linear Discriminant Analysis (LDA) is a dimensionality reduction technique.
- Unlike PCA, which focuses on capturing the maximum variance in the data, LDA aims to find the linear combinations of features that best separate the classes in a supervised learning context.
- Linear discriminant analysis (LDA), is a generalization of Fisher's linear discriminant, a method used in statistics and other fields, to find a linear combination of features that characterizes or separates two or more classes of objects or events.

LDA Steps

1. LDA requires a labeled dataset. It is a supervised dimensionality reduction technique.

2. Calculate the mean vectors for each class.

3. Compute two types of scatter matrices: the within-class scatter matrix (Sw) and the between-class scatter matrix (Sb).

4. Compute Eigenvalues and Eigenvectors: Solve the generalized eigenvalue problem to obtain the eigenvalues and corresponding eigenvectors of the matrix $Sw^{-1}Sb$.

5. Sort and select components: Sort the eigenvalues in descending order and choose the top-k eigenvectors as the transformation matrix. These eigenvectors become the new features, known as discriminant functions.

6. Project the original data onto the selected discriminant functions (eigenvectors) to obtain a lower-dimensional representation of the data.

Linear Discriminant Analysis (LDA)

- LDA maximizes the ratio of the between-class scatter matrix to the within-class scatter matrix. This means that LDA aims to maximize the separation between classes while minimizing the spread of data within each class.

- LDA is particularly useful when the goal is to find a projection that maximizes class separability, making it a powerful tool for tasks like classification.

LDA vs ANOVA

- LDA is closely related to analysis of variance (ANOVA) and regression analysis, which also attempt to express one dependent variable as a linear combination of other features or measurements.

- ANOVA uses categorical independent variables and a continuous dependent variable, whereas discriminant analysis has continuous independent variables and a categorical dependent variable (i.e. the class label).

Linear Discriminant Analysis Example

In [1]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
```

In [2]:
```
# load iris dataset
iris = load_iris()
data = iris.data

# convert dataset to pandas DataFrame
df = pd.DataFrame(data = np.c_[iris['data'], iris['target']],
                  columns = iris['feature_names'] + ['target'])
df['species'] = pd.Categorical.from_codes(iris.target, iris.target_names)
df.columns = ['s_length', 's_width', 'p_length', 'p_width', 'target', 'species']

df.head()
```

Out[2]:

	s_length	s_width	p_length	p_width	target	species
0	5.1	3.5	1.4	0.2	0.0	setosa
1	4.9	3.0	1.4	0.2	0.0	setosa
2	4.7	3.2	1.3	0.2	0.0	setosa
3	4.6	3.1	1.5	0.2	0.0	setosa
4	5.0	3.6	1.4	0.2	0.0	setosa

Linear Discriminant Analysis Example

In [3]:
```
#define predictor and response variables
X = df[['s_length', 's_width', 'p_length', 'p_width']]
y = df['species']

#Fit the LDA model
lda_model = LinearDiscriminantAnalysis()
lda_model.fit(X, y)

#Transform the features into the LDA space
X_lda = lda_model.transform(X)
```

In [4]:
```
#Transform the features into the LDA space
X_lda = lda_model.transform(X)

# Print the transformed features
print("Transformed Features (LDA Space):\n", X_lda)

```

```
Transformed Features (LDA Space):
 [[ 8.06179978e+00  3.00420621e-01]
 [ 7.12868772e+00 -7.86660426e-01]
 [ 7.48982797e+00 -2.65384488e-01]
 [ 6.81320057e+00 -6.70631068e-01]
 [ 8.13230933e+00  5.14462530e-01]
 [ 7.70194674e+00  1.46172097e+00]
```

Linear Discriminant Analysis Example

In [5]:
```
1 print("Shape of Original Features:", X.shape)
2 print("Shape of Transformed Features:", X_lda.shape)
```

```
Shape of Original Features: (150, 4)
Shape of Transformed Features: (150, 2)
```

In [6]:
```
1 lda_model.coef_
```

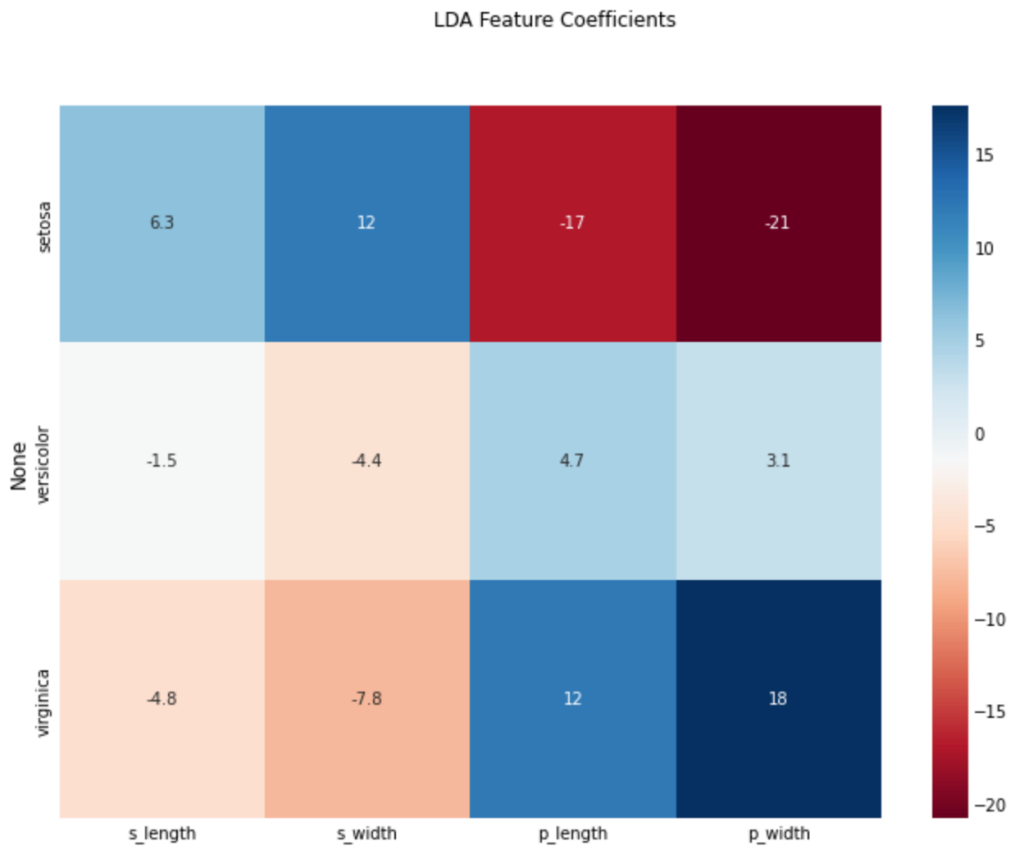
Out[6]:
```
array([[  6.31475846,  12.13931718, -16.94642465, -20.77005459],
       [ -1.53119919,  -4.37604348,   4.69566531,   3.06258539],
       [ -4.78355927,  -7.7632737 ,  12.25075935,  17.7074692 ]])
```

Linear Discriminant Analysis Example

In [7]:

```
plt.style.use('fivethirtyeight')
fig, ax = plt.subplots(1, 1, figsize=(10, 8))

sns.heatmap(pd.DataFrame(lda_model.coef_,
                         columns=df.columns[:-2],
                         index=[lda_model.classes_]),
            ax=ax, cmap='RdBu', annot=True)

plt.suptitle('LDA Feature Coefficients')
pass
```

Linear Discriminant Analysis Example



Linear Discriminant Analysis Example

In [8]:
```
# adding up abosolute values of coefficients
# reading: setosa is more separatable
pd.Series(np.abs(lda_model.coef_).sum(axis=1), index=lda_model.classes_).sort_values().plot.bar(
    figsize=(12, 6), title="LDA Class Coefficient Sums"
)
```

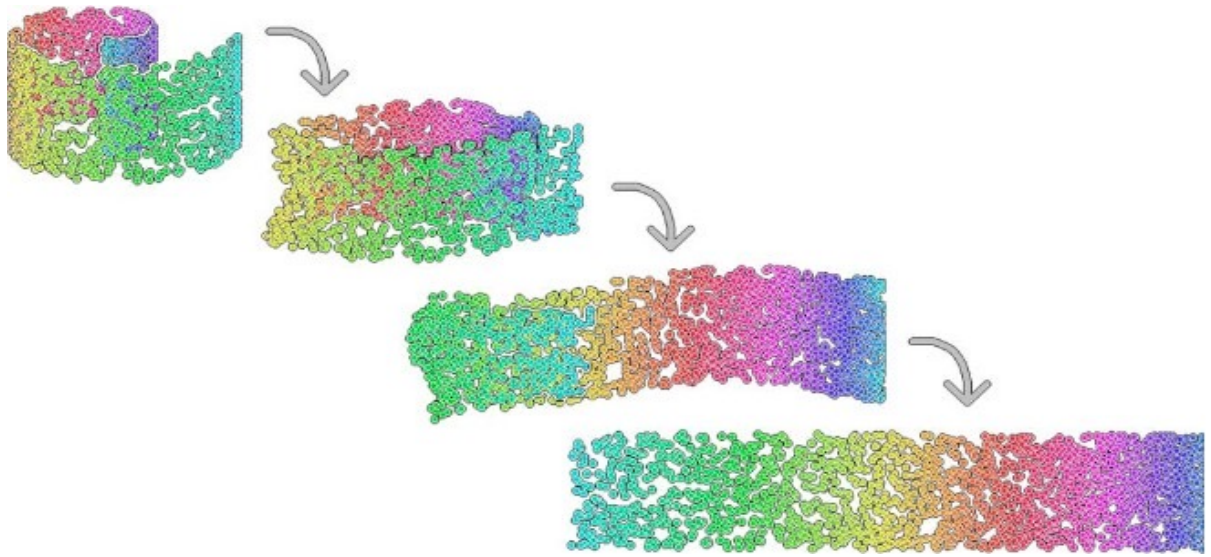
Out[8]: <Axes: title={'center': 'LDA Class Coefficient Sums'}>

Linear vs Nonlinear Dimensionality Reduction

- Linear:
  - Principal Component Analysis
  - Linear Discriminant Analysis
- Nonlinear:
  - Manifold Dimensionality Reduction
    - t-distributed Stochastic Neighbor Embedding (t-SNE)
    - Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP)

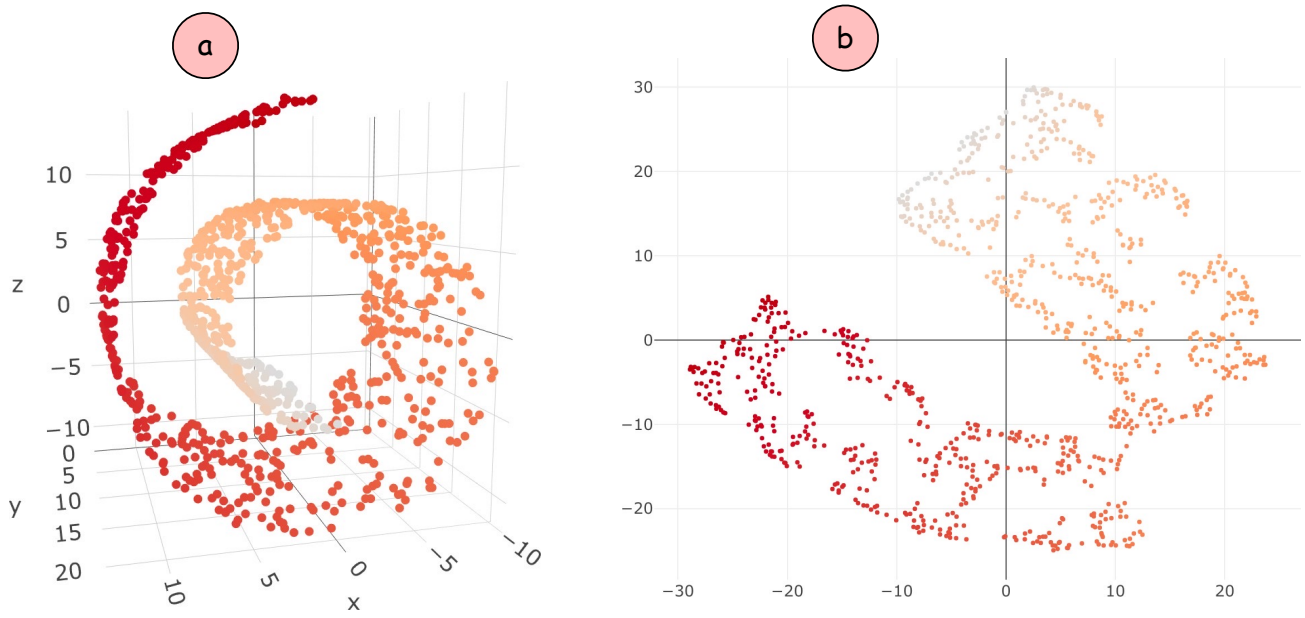
Manifold Function

A manifold is a topological space that locally resembles Euclidean space near each point. Manifolds are important in geometry and mathematical physics because they allow complex structures to be described in terms of simpler spaces.



source: https://blog.datascienceheroes.com/playing-with-dimensions-from-clustering-pca-t-sne-to-carl-sagan/

Manifold function example



(a) Swissroll function in three dimensions (b) swissroll function is twisted and plotted into a two-dimensional space and one dimension has been reduced.

Manifold Function for Dimensionality Reduction

- Manifold learning algorithms are mainly aimed at visualization, and so are rarely used to generate more than two new features.

t_SNE

- t-Distributed Stochastic Neighbor Embedding (t-SNE) is an unsupervised, nonlinear dimensionality reduction technique.

- t_SNE primarily used for data exploration and visualizing high-dimensional data. t-SNE gives an intuition of how the data is arranged in a high-dimensional space.

- The idea behind t-SNE is to find a two-dimensional representation of the data that preserves the distances between points as best as possible. t-SNE starts with a random two-dimensional representation for each data point, and then tries to make points that are close in the original feature space closer, and points that are far apart in the original feature space farther apart.

- t-SNE puts more emphasis on points that are close by, rather than preserving distances between far-apart points. In other words, it tries to preserve the information indicating which points are neighbors to each other.

t-SNE Python Example

- We will apply the t-SNE manifold learning algorithm on a dataset of handwritten digits that is included in scikit-learn. Each data point in this dataset is an 8×8 grayscale image of a handwritten digit between 0 and 9.

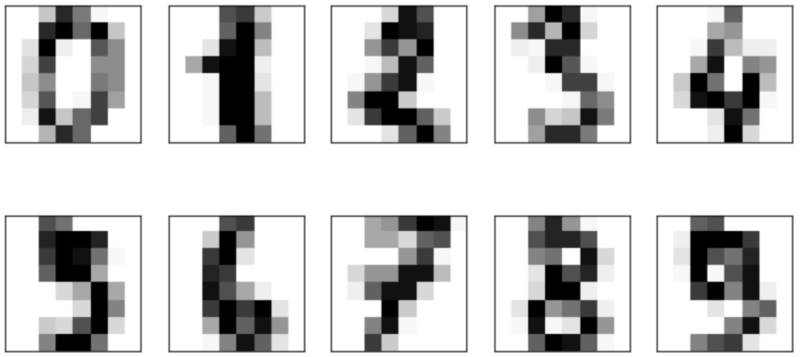
t-SNE Python Example

In [1]:

```
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

In [2]:

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

fig, axes = plt.subplots(2, 5, figsize=(10,5),
subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img, cmap='Greys')
```



t-SNE Python Example

- Let's use PCA to visualize the data reduced to two dimensions. We plot the first two principal components and represent each sample with a digit corresponding to its class.

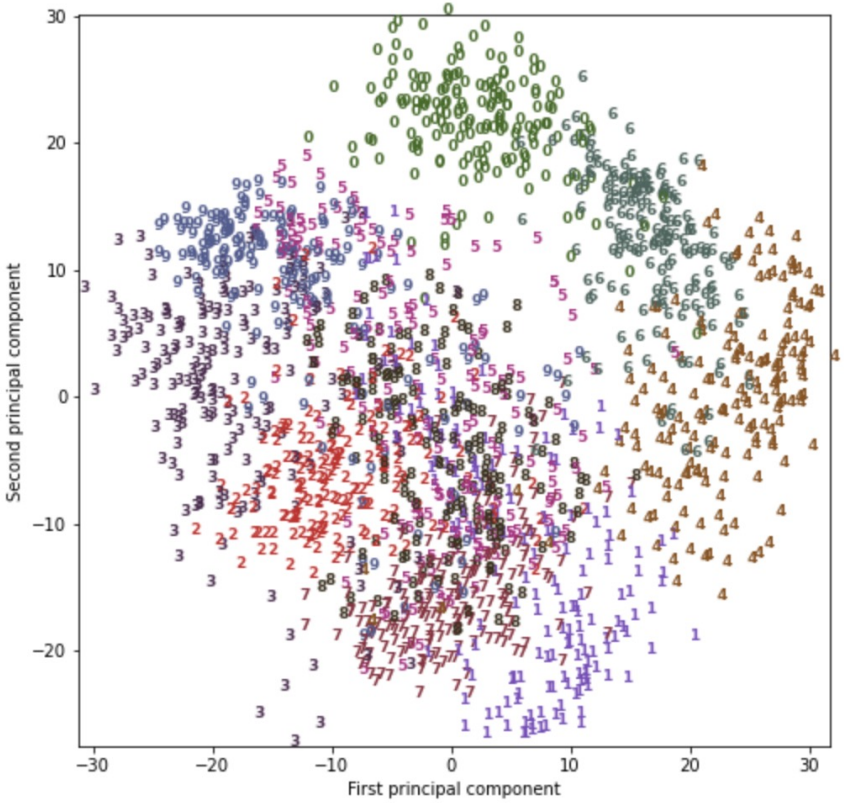
In [3]:

```
# build a PCA model
pca = PCA(n_components=2)
pca.fit(digits.data)
# transform the digits data onto the first two principal components
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(8, 8))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

Out[3]:

```
Text(0, 0.5, 'Second principal component')
```

t-SNE Python Example



t-SNE Python Example

- Let's apply t-SNE to the same dataset and compare the results.

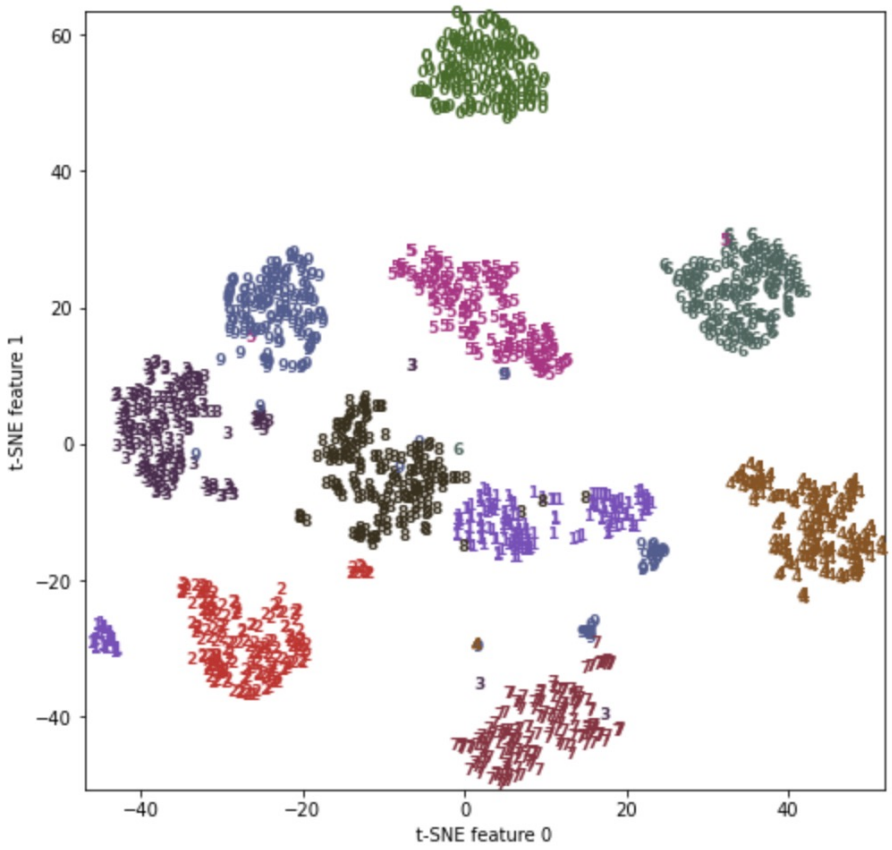
In [4]:

```
tsne = TSNE(random_state=42)
# use fit_transform instead of fit, as TSNE has no transform method
digits_tsne = tsne.fit_transform(digits.data)
plt.figure(figsize=(8, 8))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(),

digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
             color = colors[digits.target[i]],
             fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE feature 0")
plt.ylabel("t-SNE feature 1")
```

Out[4]:

```
Text(0, 0.5, 't-SNE feature 1')
```

t-SNE Python Example



t-SNE Python Example

- The result of t-SNE is quite remarkable. All the classes are quite clearly separated. The ones and nines are somewhat split up, but most of the classes form a single dense group.
- Keep in mind that this method has no knowledge of the class labels: it is completely unsupervised.

References

- Galli, Soledad. Python Feature Engineering Cookbook: Over 70 recipes for creating, engineering, and transforming features to build machine learning models. Packt Publishing. Kindle Edition.
- Dey, Sandipan. Image Processing Masterclass with Python : 50+ Solutions and Techniques Solving Complex Digital Image Processing Challenges Using Numpy, Scipy, Pytorch and Keras (English Edition). BPB Publications. Kindle Edition.
- Han, Jiawei,Pei, Jian,Tong, Hanghang. Data Mining (The Morgan Kaufmann Series in Data Management Systems). Elsevier Science. Kindle Edition.
- Müller, Andreas C.; Guido, Sarah. Introduction to Machine Learning with Python. O'Reilly Media. Kindle Edition.
- S. Guha, R. Rastogi, K. Shim, CURE: an efficient clustering algorithm for large databases, Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98) . Seattle, WA, USA . June 1998:73–84.