Netaji Subhas University of Technology, Delhi

Subject : Pattern Processing using AI

- Anureet Kaur
2019UCS2037

Experiment 1

1A)

Aim
Implement Decision Tree.

Theory
A decision tree is one of the supervised machine learning algorithms which can be used for regression and classification problems. A decision tree follows a set of if-else conditions to visualize the data and classify it according to the conditions. The term itself implies that it displays the predictions that come from a sequence of feature-based splits using a flowchart that resembles a tree structure. The decision is made by the leaves at the end, which follows the root node.

Types of Decision Tress :-

1. ID3

The algorithm iteratively (repeatedly) dichotomizes (divides) characteristics into two or more groups at each step, hence the name "ID3" (Iterative Dichotomiser 3).
ID3, developed by Ross Quinlan, constructs a decision tree from the top down in a greedy manner. Simply said, the greedy technique means that we choose the best feature at the time of each iteration to produce a node, whereas the top-down approach indicates that we build the tree from the top down.
ID3 is typically only applied to classification issues involving solely nominal features.
Entropy = Entropy is nothing but the uncertainty in our dataset or measure of disorder

$$E(s) = -p_{(+)} \log(p)_{(+)} - p_{(-)} \log(p)_{(-)}$$

where,
$p_{(+)}$ = $probabilty\ of\ positive\ class$
$p_{(-)}$ = $probabilty\ of\ negative\ class$
Information Gain = Information gain measures the reduction of uncertainty given some feature and it is also a deciding factor for which attribute should be selected as a decision node or root node.

$$Information\ Gain = E(Y) - E(Y|X)$$

2. CART

The threshold value of an attribute is used to divide the nodes in the decision tree into sub-nodes. The CART method accomplishes this by using the Gini Index criterion to find the sub nodes' best homogeneity.
The training set is the root node, which is divided into two by taking the best attribute and threshold value into account. Additionally, the subsets are divided according to the same rationale. This continues until the tree has either produced all of its potential leaves or found its last pure sub-set. Tree pruning is another name for this.

Code and Output

In [1]:
```
from sklearn import tree
```

In [2]:
```
clf = tree.DecisionTreeClassifier()
```

In [3]:
```
# height, hair length, voice pitch
X = [[180, 15, 0],
     [167, 42, 1],
     [136, 35, 1],
     [174, 15, 0],
     [141, 28, 1]]
```

In [4]:
```
Y = ['Man','Woman','Woman','Man','Woman']
```

In [5]:
```
dtclf = clf.fit(X,Y)
```

In [6]:
```
prediction = dtclf.predict([[133,37,1]])
```

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

```
['Woman']
```

In [8]:
```
from matplotlib import pyplot as plt
```

1B)

Aim
Visualise Decision Tree

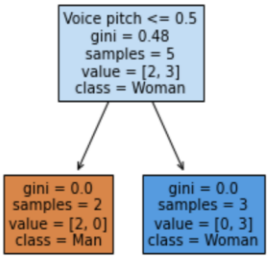
Theory
For regression and classification issues, a decision tree is a widely used non-parametric efficient machine learning modelling tool. A decision tree uses predictor data to make sequential, hierarchical decisions regarding the outcomes variable in order to identify solutions. They require extremely minimal data, can handle both qualitative and quantitative data, and are incredibly simple to grasp due to their visual depiction.
Visualization steps :–
In the decision tree, the nodes are split into subnodes on the basis of a threshold value of an attribute. The CART algorithm does that by searching for the best homogeneity for the subnodes, with the help of the Gini Index criterion.

1. The root node is taken as the training set and is split into two by considering the best attribute and threshold value.
2. Further, the subsets are also split using the same logic.
3. This continues till the last pure sub-set is found in the tree or the maximum number of leaves possible in that growing tree. This is also known as Tree Pruning.

In this example, for Voice Pitch <=0.5, we obtain a gini score of 0.48 which splits the data into two leaf nodes, for YES, it is a Man and for NO, it is a Woman.

Code and Output

```
In [8]: from matplotlib import pyplot as plt

In [9]: fig = plt.figure(figsize=(4,4), facecolor="white")
        tree.plot_tree(dtclf, feature_names=["Height", "Hair length", "Voice pitch"],
                       class_names=["Man","Woman"],filled=True)

Out[9]: [Text(111.6, 163.07999999999998, 'Voice pitch <= 0.5\ngini = 0.48\nsamples = 5\nvalue = [2, 3]\nclass = Woman'),
         Text(55.8, 54.360000000000014, 'gini = 0.0\nsamples = 2\nvalue = [2, 0]\nclass = Man'),
         Text(167.39999999999998, 54.360000000000014, 'gini = 0.0\nsamples = 3\nvalue = [0, 3]\nclass = Woman')]
```



Experiment 2

Aim
Implement SVM.

Theory
Support Vector Machine or SVM is one of the most popular supervised learning algorithms, which is used for classification as well as regression problems. However, primarily, it is used for classification problems in machine learning.
The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.
SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine.
SVM algorithm can be used for face detection, image classification, text categorization, etc.

Types of SVM
SVM can be of two types :–

1. Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier used is called as linear SVM classifier.
2. Non-linear SVM: Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm
Hyperplane
There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.
The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features then hyperplane will be a straight line. And if there are 3 features, then hyperplane will be a 2-dimension plane.
Support Vectors
The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as support vectors. Since these vectors support the hyperplane, hence called support vectors.

Code and Output

In [1]:
```
from sklearn.svm import SVC
import matplotlib.pyplot as plt
```

In [2]:
```
clf = SVC()
```

In [3]:
```
# height, hair length, voice pitch
X = [[180, 15, 0],
     [167, 42, 1],
     [136, 35, 1],
     [174, 15, 0],
     [141, 28, 1]]
```

In [4]:
```
Y = ['Man','Woman','Woman','Man','Woman']
```

In [5]:
```
svcClassifier = clf.fit(X,Y)
```

In [6]:
```
prediction = svcClassifier.predict([[133,37,1]])
```

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

```
['Woman']
```

MNIST Dataset

In [8]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import seaborn as sns
from sklearn.metrics import classification_report,confusion_matrix
```

In [9]:
```
x, y = load_digits(return_X_y=True)
```

In [10]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, train_size=0.8)
```

In [11]:
```
steps = [('scaler', StandardScaler()), ('SVM', SVC())]
```

In [12]:
```
pipeline = Pipeline(steps)
```

In [13]:
```
svcClassifier = pipeline.fit(x_train, y_train)
```

In [14]:
```
print(classification_report(y_test,svcClassifier.predict(x_test)))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        31
           1       1.00      1.00      1.00        38
           2       1.00      0.97      0.99        39
           3       0.97      1.00      0.99        35
           4       0.91      0.97      0.94        33
           5       1.00      0.97      0.99        37
           6       1.00      1.00      1.00        35
           7       0.95      0.98      0.96        41
           8       1.00      0.97      0.98        29
           9       1.00      0.98      0.99        42

    accuracy                           0.98       360
   macro avg       0.98      0.98      0.98       360
weighted avg       0.98      0.98      0.98       360
```

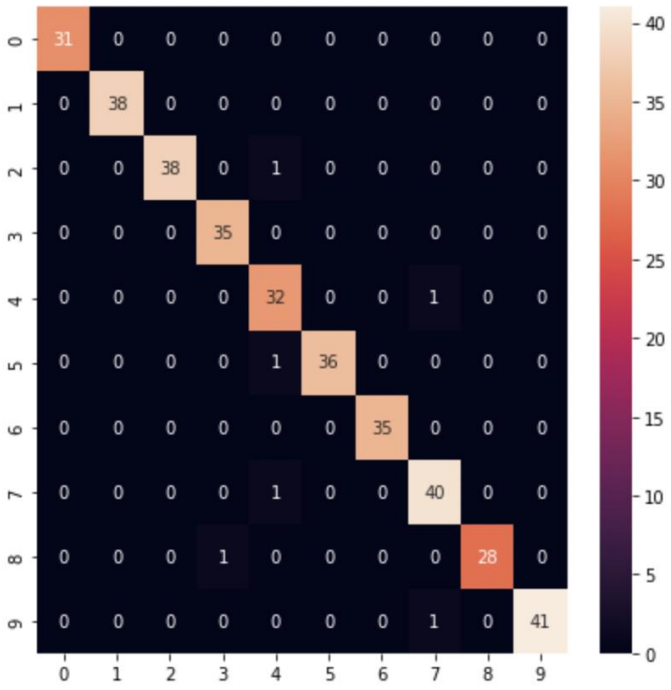
In [15]:
```
train_acc = float(svcClassifier.score(x_train, y_train)*100)
print("train accuracy score:", train_acc)

test_acc = float(svcClassifier.score(x_test, y_test)*100)
print("test accuracy score:", test_acc)
```

```
train accuracy score: 99.58246346555325
test accuracy score: 98.33333333333333
```

In [16]:
```
cf_matrix = confusion_matrix(y_test,svcClassifier.predict(x_test))
print(cf_matrix)
```

```
[[31  0  0  0  0  0  0  0  0  0]
 [ 0 38  0  0  0  0  0  0  0  0]
 [ 0  0 38  0  1  0  0  0  0  0]
 [ 0  0  0 35  0  0  0  0  0  0]
 [ 0  0  0  0 32  0  0  1  0  0]
 [ 0  0  0  0  1 36  0  0  0  0]
 [ 0  0  0  0  0  0 35  0  0  0]
 [ 0  0  0  0  1  0  0 40  0  0]
 [ 0  0  0  1  0  0  0  0 28  0]
 [ 0  0  0  0  0  0  0  1  0 41]]
```

In [17]:
```
fig = plt.figure(figsize=(7,7), facecolor="white")
cf_matrix = confusion_matrix(y_test,svcClassifier.predict(x_test))
sns.heatmap(cf_matrix, annot=True)
plt.show()
```



Experiment 3

Aim

Implement Agglomerative Hierarchical Clustering.

Theory

Hierarchical clustering is an unsupervised machine learning algorithm, which is used to group the unlabelled datasets into a cluster and also known as hierarchical cluster analysis or HCA.

In this algorithm, we develop the hierarchy of clusters in the form of a tree, and this tree-shaped structure is known as the dendrogram.

The hierarchical clustering technique has two approaches

1. Agglomerative: Agglomerative is a bottom-up approach, in which the algorithm starts with taking all data points as single clusters and merging them until one cluster is left.
2. Divisive: Divisive algorithm is the reverse of the agglomerative algorithm as it is a top-down approach.

Agglomerative Hierarchical Clustering

The agglomerative hierarchical clustering algorithm is a popular example of HCA. To group the datasets into clusters, it follows the bottom-up approach. It means, this algorithm considers each dataset as a single cluster at the beginning, and then start combining the closest pair of clusters together. It does this until all the clusters are merged into a single cluster that contains all the datasets.

This hierarchy of clusters is represented in the form of the dendrogram. The dendrogram is a tree-like structure that is mainly used to store each step as a memory that the HC algorithm performs. In the dendrogram plot, the Y-axis shows the Euclidean distances between the data points, and the x-axis shows all the data points of the given dataset.

Measure for the distance between two clusters

The closest distance between the two clusters is crucial for the hierarchical clustering. There are various ways to calculate the distance between two clusters, and these ways decide the rule for clustering. These measures are called Linkage methods. Some of the popular linkage methods are given below:

1. Single Linkage: It is the Shortest Distance between the closest points of the clusters.
2. Complete Linkage: It is the farthest distance between the two points of two different clusters. It is one of the popular linkage methods as it forms tighter clusters than single-linkage.
3. Average Linkage: It is the linkage method in which the distance between each pair of datasets is added up and then divided by the total number of datasets to calculate the average distance between two clusters. It is also one of the most popular linkage methods.

4. Centroid Linkage: It is the linkage method in which the distance between the centroid of the clusters is calculated.

From the above-given approaches, we can apply any of them according to the type of problem or business requirement.

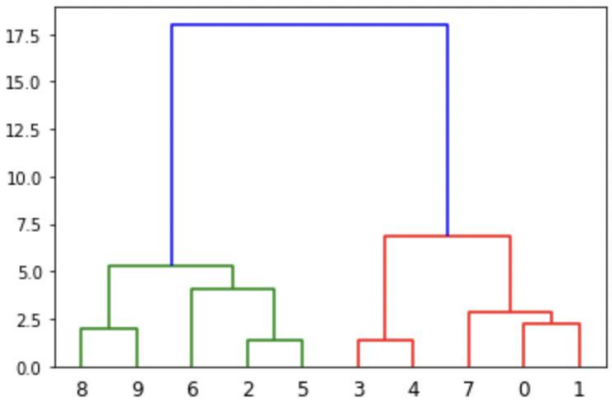
Code and Output

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering
```

In [2]:
```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
```

In [3]:
```
data = list(zip(x, y))
```

In [4]:
```
linkage_data = linkage(data, method='ward', metric='euclidean')
dendrogram(linkage_data)

plt.show()
```

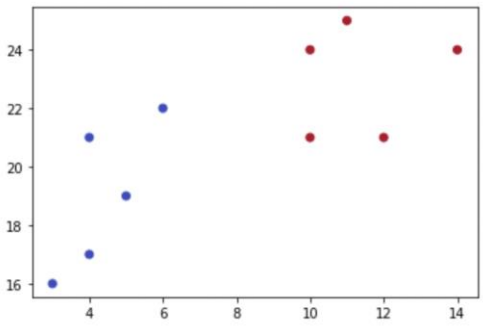


In [5]:
```
hierarchical_cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')
labels = hierarchical_cluster.fit_predict(data)
```

In [6]:
```
print(labels)
```

```
[0 0 1 0 0 1 1 0 1 1]
```

In [7]:
```
plt.scatter(x, y, c=labels, cmap='coolwarm')
plt.show()
```



Experiment 4

Aim

Write a program to implement k-means clustering from scratch.

Theory

K-Means Clustering is an unsupervised learning algorithm that is used to solve the clustering problems in machine learning or data science.
It groups the unlabelled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on.
It is an iterative algorithm that divides the unlabelled dataset into k different clusters in such a way that each dataset belongs only one group that has similar properties.

It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabelled dataset on its own without the need for any training. It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.

K-means follows Expectation-Maximization approach to solve the problem. The Expectation-step is used for assigning the data points to the closest cluster and the Maximization-step is used for computing the centroid of each cluster.

We can understand the working of K-Means clustering algorithm with the help of following steps –

1. First, we need to specify the number of clusters, K, need to be generated by this algorithm.
2. Next, randomly select K data points and assign each data point to a cluster. In simple words, classify the data based on the number of data points.
3. Now it will compute the cluster centroids.
4. Next, keep iterating the following until we find optimal centroid which is the assignment of data points to the clusters that are not changing any more –
   a. First, the sum of squared distance between data points and centroids would be computed.
   b. Now, we have to assign each data point to the cluster that is closer than other cluster (centroid).
   c. At last compute the centroids for the clusters by taking the average of all data points of that cluster.

While working with K-means algorithm we need to take care of the following things –

- While working with clustering algorithms including K-Means, it is recommended to standardize the data because such algorithms use distance-based measurement to determine the similarity between data points.

- Due to the iterative nature of K-Means and random initialization of centroids, K-Means may stick in a local optimum and may not converge to global optimum. That is why it is recommended to use different initializations of centroids.

Code and Output

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

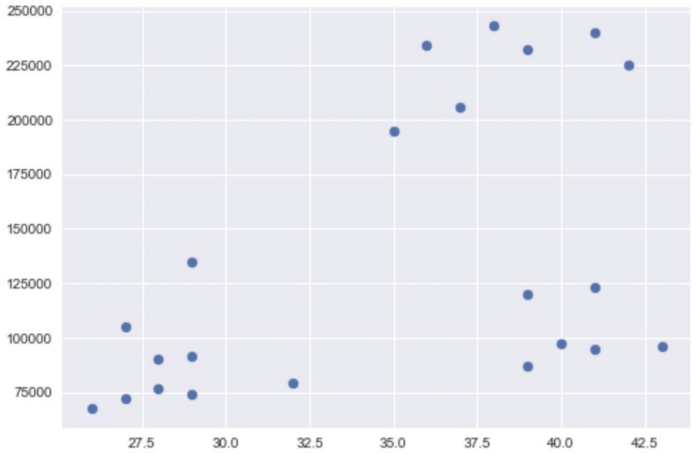
In [2]:
```
data = pd.read_excel("K Means Clustering.xlsx")
```

In [3]:
```
data.head()
```

Out[3]:

	Name	Age	Income
0	N1	27	105000
1	N2	29	135000
2	N3	29	91500
3	N4	28	90000
4	N5	42	225000

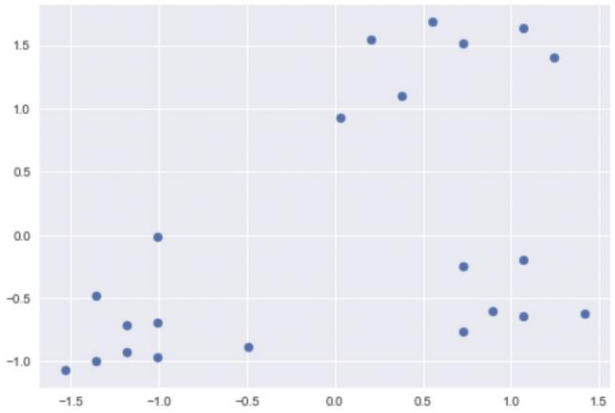
In [4]:
```
plt.style.use("seaborn")
plt.figure(0)

plt.scatter(data['Age'],data['Income'])
plt.show()
```



In [5]:
```
X1 = data["Age"].values
X2 = data["Income"].values

X1 = (X1-np.mean(X1))/np.std(X1)
X2 = (X2-np.mean(X2))/np.std(X2)

X = np.column_stack((X1,X2))
print(X.shape)
```

```
(22, 2)
```

In [6]:
```
plt.figure(0)

plt.scatter(X1,X2)
plt.show()
```



K Means Clustering Algorithm

1. Random Initialisation

In [7]:
```
k = 3                                                    # specifying how many clusters we want to have
colour = ["green","red","purple"]                        # specifying colour for each of the cluster
clusters = {}                                            # dictionary of all the clusters

for i in range(k):

    # initialising one random center for each cluster in the range (-1.5,1.5)
    center = 1.5*(2*np.random.random((X.shape[1],))-1)    # generating one data point having 2 features

    # we also need an assignment list (so with each cluster center, we associate a list)
    points = []                                  # contains points assigned to / associated with that cluster

    # for every cluster we create one dictionary (this is the ith cluster)
    cluster = {
        'center': center,
        'points': points,
        'colour': colour[i]
    }
    clusters[i] = cluster
```

In [8]:
```
clusters
```

Out[8]:
```
{0: {'center': array([ 0.49841014, -1.34623009]),
  'points': [],
  'colour': 'green'},
 1: {'center': array([0.76164261, 1.01725974]), 'points': [], 'colour': 'red'},
 2: {'center': array([-1.4920082 ,  0.66873958]),
  'points': [],
  'colour': 'purple'}}
```

2. Implementing E Step

In [9]:
```
# helper function to compute distance between 2 vectors or points (we use euclidean distance)

def distance(v1,v2):
    return np.sqrt(np.sum((v1-v2)**2))
```

In [10]:
```
def assignPointToClusters(clusters):

    # iterating over all the training data points
    for ix in range(X.shape[0]):
        dist = []
        curr_x = X[ix]        # current data point

        # computing the distance of current data point from the center of the kth cluster
        for kx in range(k):
            d = distance(curr_x, clusters[kx]['center'])
            dist.append(d)

        # choosing the cluster which has the minimum distance
        current_cluster = np.argmin(dist)

        clusters[current_cluster]['points'].append(curr_x)

    return
```

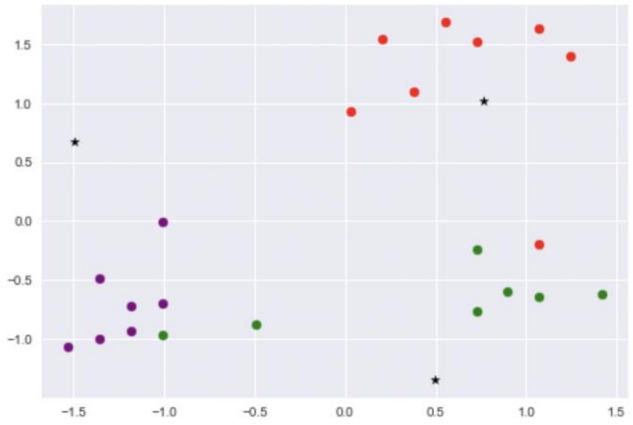
In [11]:
```
assignPointToClusters(clusters)
```

In [12]:
```
clusters
```

Out[12]:
```
{0: {'center': array([ 0.49841014, -1.34623009]),
  'points': [array([-1.00915747, -0.9629726 ]),
   array([-0.48881065, -0.88063069]),
   array([ 0.89878087, -0.59831557]),
   array([ 1.07222982, -0.64536809]),
   array([ 1.4191277 , -0.62184183]),
   array([ 0.72533193, -0.24542167]),
   array([ 0.72533193, -0.76299939])],
  'colour': 'green'},
 1: {'center': array([0.76164261, 1.01725974]),
  'points': [array([1.24567876, 1.40141653]),
   array([0.72533193, 1.51904783]),
   array([1.07222982, 1.63667914]),
   array([0.55188299, 1.68373166]),
   array([0.20498511, 1.54257409]),
   array([0.03153617, 0.93089133]),
   array([0.37843405, 1.09557515]),
   array([ 1.07222982, -0.19836915])],
  'colour': 'red'},
 2: {'center': array([-1.4920082 ,  0.66873958]),
  'points': [array([-1.35605536, -0.48068427]),
   array([-1.00915747, -0.01015907]),
   array([-1.00915747, -0.69242061]),
   array([-1.18260641, -0.71594687]),
   array([-1.5295043 , -1.06884077]),
   array([-1.35605536, -0.99826199]),
   array([-1.18260641, -0.92768321])],
  'colour': 'purple'}}
```

In [13]:
```
# function to see if we are able to plot the cluster centers and the points associated with them

def plotCluster(clusters):
    # iterate over every cluster
    for kx in range(k):

        # converting the list to numpy array so that we can give it to matplotlib function
        pts = np.array(clusters[kx]['points'])

        # PLOTTING THE POINTS
        try:
            plt.scatter(pts[:,0],pts[:,1],color=clusters[kx]['colour'])
        except:
            pass

        # PLOTTING THE CLUSTER CENTERS
        uk = clusters[kx]['center']
        plt.scatter(uk[0],uk[1],color='black',marker='*')
```

In [14]:
```
plotCluster(clusters)
```

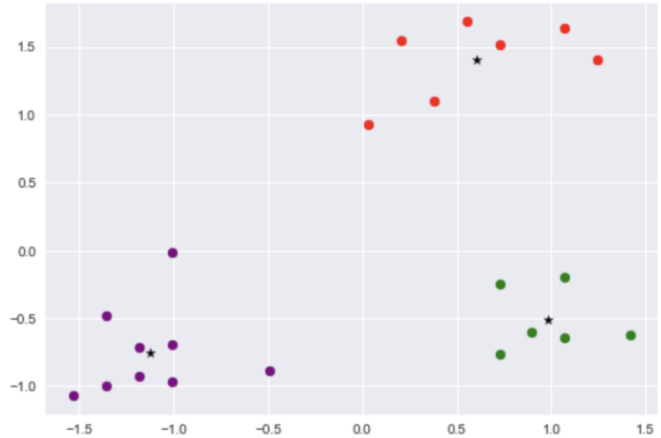


3. Implementing M Step

In [15]:
```
def updateClusters(clusters):
    for kx in range(k):
        pts = np.array(clusters[kx]['points'])

        # if a cluster has non-zero points, we will take the mean and update that cluster
        if pts.shape[0]>0:
            new_uk = pts.mean(axis=0)          # new cluster center
            clusters[kx]['center'] = new_uk    # updating the center

            # emptying the list of points of that cluster as will repeat E-Step to get the points
            clusters[kx]['points'] = []
```

4. Repeating E and M steps again and again until there is no change in the value of cluster centers

In [16]:
```
while(1):
  cnt = 0
  prev_center = []

  for ix in range(k):
    prev_center.append(clusters[ix]['center'])

  assignPointToClusters(clusters)
  updateClusters(clusters)

  for ix in range(k):
    if(all(prev_center[ix]==clusters[ix]['center'])):
      cnt+=1

  if(cnt==k):
    break

assignPointToClusters(clusters)
plotCluster(clusters)
```



Experiment 5

Aim

Implement Maximum-Likelihood estimation.

Theory

Maximum likelihood estimation (MLE) is a technique used for estimating the parameters of a given distribution, using some observed data. The parameter values are found such that they maximise the likelihood that the process described by the model produced the data that were actually observed.
For example, if a population is known to follow a normal distribution but the mean and variance are unknown, MLE can be used to estimate them using a limited sample of the population, by finding particular values of the mean and variance so that the observation is the most likely result to have occurred.

Let $x_1, x_2, \ldots, x_n$ be observations from n independent and identically distributed random variables drawn from a Probability Distribution f_0, where f_0 is known to be from a family of distributions f that depend on some parameters.
For example, f_0 could be known to be from the family of normal distributions f, which depend on parameters σ (standard deviation) and μ (mean), and $x_1, x_2, \ldots, x_n$ would be observations from f_0.

The goal of MLE is to maximize the likelihood function:

$$L = f(x_1, x_2, \ldots, x_n|\theta) = f(x_1|\theta) \times f(x_2|\theta) \times \ldots \times f(x_n|\theta)$$

Often, the average log-likelihood function is easier to work with:

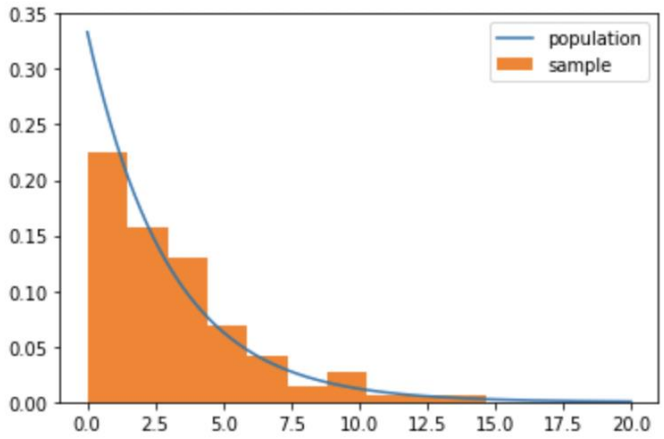
$$\hat{\ell} = \frac{1}{n} \log L = \frac{1}{n} \sum_{i=1}^{n} \log f(x_i|\theta)$$

Code and Output

In [1]:
```
import numpy as np
from scipy.stats import expon
import matplotlib.pyplot as plt
```

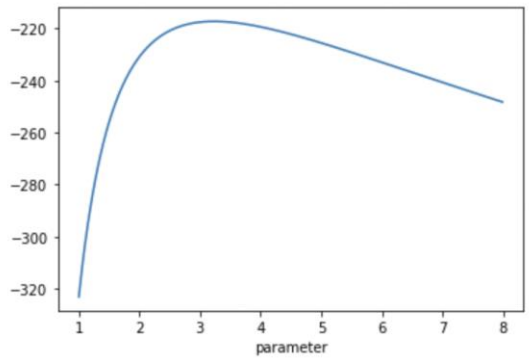
In [2]:
```
# population with exponential distribution

population_rate = 3
sample_size = 100

get_sample = lambda n: np.random.exponential(population_rate, n)
xs = np.arange(0, 20, 0.001)
ys = expon.pdf(xs, scale=population_rate)
```

In [3]:
```
plt.plot(xs, ys, label='population')

sample = get_sample(sample_size)
plt.hist(sample, density=True, label='sample')

plt.legend()
plt.show()
```



In [4]:
```
# estimate λ(rate) parameter of the actual population by having a sample from this population

log_likelihood = lambda rate: sum([np.log(expon.pdf(v, scale=rate)) for v in sample])

rates = np.arange(1, 8, 0.01)
estimates = [log_likelihood(r) for r in rates]

plt.xlabel('parameter')
plt.plot(rates, estimates)
print('parameter value: ', rates[estimates.index(max(estimates))])
```

```
parameter value:  3.230000000000002
```



Experiment 6

Aim

Implement Principal Component Analysis and use it for unsupervised learning.

Theory

Principal Component Analysis is an unsupervised learning algorithm that is used for the dimensionality reduction in machine learning. It is a statistical process that converts the observations of correlated features into a set of linearly uncorrelated features with the help of orthogonal transformation. It is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set. These new transformed features are called the Principal Components.

PCA generally tries to find the lower-dimensional surface to project the high-dimensional data. It works by considering the variance of each attribute because the high attribute shows the good split between the classes, and hence it reduces the dimensionality.

Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Because smaller data sets are easier to explore and visualize and make analysing data much easier and faster for machine learning algorithms without extraneous variables to process.
So, to sum up, the idea of PCA is simple - reduce the number of variables of a data set, while preserving as much information as possible.

Steps to perform PCA

1. Standardize the range of continuous initial variables.
2. Compute the covariance matrix to identify correlations.
3. Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components.
4. Create a feature vector to decide which principal components to keep.
5. Recast the data along the principal components axes.

It is one of the popular tools that is used for exploratory data analysis and predictive modelling. It is a technique to draw strong patterns from the given dataset by reducing the variances. Some real-world applications of PCA are image processing, movie recommendation system, optimizing the power allocation in various communication channels. It is a feature extraction technique, so it contains the important variables and drops the least important variable.

Code and Output

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

In [2]:
```
cancer = load_breast_cancer()
df = pd.DataFrame(cancer['data'], columns = cancer['feature_names'])
df.head()
```

Out[2]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809

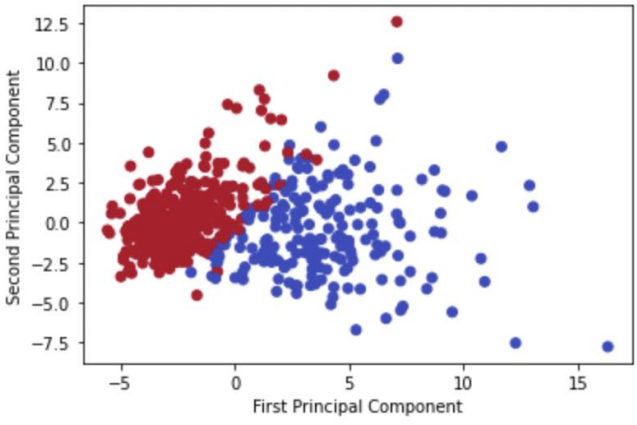
5 rows × 30 columns

In [3]:
```
scalar = StandardScaler()
scalar.fit(df)
scaled_data = scalar.transform(df)
```

In [4]:
```
pca = PCA(n_components = 2)
pca.fit(scaled_data)
x_pca = pca.transform(scaled_data)

x_pca.shape
```

Out[4]:
```
(569, 2)
```

In [5]:
```
plt.scatter(x_pca[:, 0], x_pca[:, 1], c = cancer['target'], cmap ='coolwarm')

plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.show()
```



In [6]:
```
pca.components_
```

Out[6]:
```
array([[ 0.21890244,  0.10372458,  0.22753729,  0.22099499,  0.14258969,
         0.23928535,  0.25840048,  0.26085376,  0.13816696,  0.06436335,
         0.20597878,  0.01742803,  0.21132592,  0.20286964,  0.01453145,
         0.17039345,  0.15358979,  0.1834174 ,  0.04249842,  0.10256832,
         0.22799663,  0.10446933,  0.23663968,  0.22487053,  0.12795256,
         0.21009588,  0.22876753,  0.25088597,  0.12290456,  0.13178394],
       [-0.23385713, -0.05970609, -0.21518136, -0.23107671,  0.18611302,
         0.15189161,  0.06016536, -0.0347675 ,  0.19034877,  0.36657547,
        -0.10555215,  0.08997968, -0.08945723, -0.15229263,  0.20443045,
         0.2327159 ,  0.19720728,  0.13032156,  0.183848  ,  0.28009203,
        -0.21986638, -0.0454673 , -0.19987843, -0.21935186,  0.17230435,
         0.14359317,  0.09796411, -0.00825724,  0.14188335,  0.27533947]])
```

Experiment 7

Aim

Write a python program to implement simple Chatbot.

Theory

Bots are specially built software that interacts with internet users automatically. Bots are made up of algorithms that assist them in completing jobs. By auto-designed, we mean that they run on their own, following instructions, and therefore begin the conservation process without the need for human intervention.

Types of Chatbots

Chatbots deliver instantly by understanding the user requests with pre-defined rules and AI based chatbots. There are two types of chatbots.

- Rule Based Chatbots: This type of chatbots answer the customer queries using the pre-defined rules. These bots answer common queries such as hours of operation of business, addresses, phone numbers and tracking status.
- Conversational AI Chatbots: This type of chatbots using Natural language Processing(NLP) to understand the context and intent of a user input before providing the response. These Bots train themselves as per the user inputs and more they learn, more they become user interactive.

ChatterBot is a Python library to create a chat bot that is developed to provide automated responses to user inputs. It makes utilization of a combination of Machine Learning algorithms in order to generate multiple types of responses. This feature enables developers to construct chatbots using Python that can communicate with humans and provide relevant and appropriate responses. Moreover, the ML algorithms support the bot to improve its performance with experience.

Another amazing feature of the ChatterBot library is its language independence. The library is developed in such a manner that makes it possible to train the bot in more than one programming language.

Working of the ChatterBot library

When a user inserts a particular input in the chatbot (designed on ChatterBot), the bot saves the input and the response for any future usage. This information (of gathered experiences) allows the chatbot to generate automated responses every time a new input is fed into it.

The program picks the most appropriate response from the nearest statement that matches the input and then delivers a response from the already known choice of statements and responses. Over time, as the chatbot indulges in more communications, the precision of reply progresses.

Code and Output

In [1]:
```
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
```

In [2]:
```
import ssl

try:
    _create_unverified_https_context = ssl._create_unverified_context
except AttributeError:
    pass
else:
    ssl._create_default_https_context = _create_unverified_https_context
```

In [3]:
```
bot = ChatBot('Bot')
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /Users/nupur/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package punkt to /Users/nupur/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /Users/nupur/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [4]:
```
trainer = ListTrainer(bot)

trainer.train([
'Hi',
'Hello',
'Please take my order.',
'What would you like to have?',
'I would like to have pasta, pizza and french fries.',
'Anything to drink?',
'One large cold coffee.',
'Is that all?',
'Yes, thanks!',
'No Problem! Your order will be ready in 15 min.'
])
```

```
List Trainer: [####################] 100%
```

In [5]:
```
while True:
    request=input('You: ')
    if request == 'Ok':
        print('Bot: Bye!')
        break
    else:
        response = bot.get_response(request)
        print('Bot:', response)
```

```
You: Hi
Bot: Hello
You: How are you?
Bot: I am doing well.
You: Please take my order.
Bot: What would you like to have?
You: I would like to have pasta, pizza and french fries.
Bot: Anything to drink?
You: One large cold coffee.
Bot: Is that all?
You: Yes, thanks!
Bot: No Problem! Have a Good Day!
You: Ok
Bot: Bye!
```