1. Implement the Naïve Bayes Classifier on the below given dataset. Test record for the given dataset is (Rainy, Cool, Normal, True). Also test the same on a large dataset with a sample test record.

In [19]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import LabelEncoder
from sklearn import metrics
from sklearn.model_selection import train_test_split
import random
```

In [20]:
```
df = pd.read_csv("golf.csv")
df
```

Out[20]:

	Outlook	Temp	Humidity	Windy	Play Golf
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny	Mild	High	True	No

In [21]:
```
# adding the test data in dataset to convert into encoded format.. later split it for test dataset.
df.loc[len(df)] = ['Rainy','Cool','Normal',True,'NULL']
df
```

Out[21]:

	Outlook	Temp	Humidity	Windy	Play Golf
0	Rainy	Hot	High	False	No
1	Rainy	Hot	High	True	No
2	Overcast	Hot	High	False	Yes
3	Sunny	Mild	High	False	Yes
4	Sunny	Cool	Normal	False	Yes
5	Sunny	Cool	Normal	True	No
6	Overcast	Cool	Normal	True	Yes
7	Rainy	Mild	High	False	No
8	Rainy	Cool	Normal	False	Yes
9	Sunny	Mild	Normal	False	Yes
10	Rainy	Mild	Normal	True	Yes
11	Overcast	Mild	High	True	Yes
12	Overcast	Hot	Normal	False	Yes
13	Sunny	Mild	High	True	No
14	Rainy	Cool	Normal	True	NULL

In [22]:
```
df_features_X = df.iloc[:,:-1]
df_target_Y = df[['Play Golf']]
```

In [23]:
```
# encoding the values...
LE = LabelEncoder()
for i in range(df_features_X.shape[1]):
    df_features_X.iloc[:,i] = LE.fit_transform(df_features_X.iloc[:,i])

df_features_X
```

Out[23]:

	Outlook	Temp	Humidity	Windy
0	1	1	0	0
1	1	1	0	1
2	0	1	0	0
3	2	2	0	0
4	2	0	1	0
5	2	0	1	1
6	0	0	1	1
7	1	2	0	0
8	1	0	1	0
9	2	2	1	0
10	1	2	1	1
11	0	2	0	1
12	0	1	1	0
13	2	2	0	1
14	1	0	1	1

In [24]:
```
# splitting the test data...
test_X = df_features_X.iloc[-1,:]
test_X
```

Out[24]:
```
Outlook     1
Temp        0
Humidity    1
Windy       1
Name: 14, dtype: int64
```

In [25]:
```
import warnings
warnings.filterwarnings('ignore')

# get train_x and train_y data..
train_X = df_features_X.iloc[:-1,:]
train_Y = df_target_Y.iloc[:-1, :]

classifier = GaussianNB()
classifier.fit(train_X, train_Y)

# predicting
Y_predicted = classifier.predict([np.array(test_X)])
# print("Accuracy = ", metrics.accuracy_score(y_test, Y_predicted))

Y_predicted
```

Out[25]:
```
array(['Yes'], dtype='<U3')
```

In [ ]:
```

```

Testing the data on larger dataset..

In [26]:
```
df = pd.read_csv("lung_cancer.csv")
df
```

Out[26]:

	GENDER	AGE	SMOKING	YELLOW_FINGERS	ANXIETY	PEER_PRESSURE	CHRONIC DISEASE	FATIGUE	ALLERGY	WHEEZING	ALCOHOL CONSUMING	COUGHING	SHORTNESS OF BREATH	SWALLOWING DIFFICULTY	C
0	M	69	1	2	2	1	1	2	1	2	2	2	2	2	
1	M	74	2	1	1	1	2	2	2	1	1	1	2	2	
2	F	59	1	1	1	2	1	2	1	2	1	2	2	1	
3	M	63	2	2	2	1	1	1	1	1	2	1	1	2	
4	F	63	1	2	1	1	1	1	1	2	1	2	2	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
304	F	56	1	1	1	2	2	2	1	1	2	2	2	2	
305	M	70	2	1	1	1	1	2	2	2	2	2	2	1	
306	M	58	2	1	1	1	1	1	2	2	2	2	1	1	
307	M	67	2	1	2	1	1	2	2	1	2	2	2	1	
308	M	62	1	1	1	2	1	2	2	2	2	1	1	2	

309 rows × 16 columns

Pre-processing the data..

1 = NO, 2 = Yes

In [27]:
```
for each in df.columns[2:-1]:
    df[each] = np.where(df[each] == 2, 1, 0)

df.head()
```

Out[27]:

	GENDER	AGE	SMOKING	YELLOW_FINGERS	ANXIETY	PEER_PRESSURE	CHRONIC DISEASE	FATIGUE	ALLERGY	WHEEZING	ALCOHOL CONSUMING	COUGHING	SHORTNESS OF BREATH	SWALLOWING DIFFICULTY	CHI P/
0	M	69	0	1	1	0	0	1	0	1	1	1	1	1	
1	M	74	1	0	0	0	1	1	1	0	0	0	1	1	
2	F	59	0	0	0	1	0	1	0	1	0	1	1	0	
3	M	63	1	1	1	0	0	0	0	0	1	0	0	1	
4	F	63	0	1	0	0	0	0	0	1	0	1	1	0	

In [28]:
```
train_X = df.iloc[:,:-1]
train_Y = df.iloc[:,-1]
train_Y
```

Out[28]:
```
0      YES
1      YES
2       NO
3       NO
4       NO
      ...
304    YES
305    YES
306    YES
307    YES
308    YES
Name: LUNG_CANCER, Length: 309, dtype: object
```

In [29]:
```
# encoding
from sklearn.preprocessing import OrdinalEncoder
OE = OrdinalEncoder()

for i in range(train_X.shape[1]):
    train_X.iloc[:,i] = LE.fit_transform(train_X.iloc[:,i])

train_X
```

Out[29]:

	GENDER	AGE	SMOKING	YELLOW_FINGERS	ANXIETY	PEER_PRESSURE	CHRONIC DISEASE	FATIGUE	ALLERGY	WHEEZING	ALCOHOL CONSUMING	COUGHING	SHORTNESS OF BREATH	SWALLOWING DIFFICULTY	C
0	1	26	0	1	1	0	0	1	0	1	1	1	1	1	
1	1	31	1	0	0	0	1	1	1	0	0	0	1	1	
2	0	16	0	0	0	1	0	1	0	1	0	1	1	0	
3	1	20	1	1	1	0	0	0	0	0	1	0	0	1	
4	0	20	0	1	0	0	0	0	0	1	0	1	1	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
304	0	13	0	0	0	1	1	1	0	0	1	1	1	1	
305	1	27	1	0	0	0	0	1	1	1	1	1	1	0	
306	1	15	1	0	0	0	0	0	1	1	1	1	0	0	
307	1	24	1	0	1	0	0	1	1	0	1	1	1	0	
308	1	19	0	0	0	1	0	1	1	1	1	0	0	1	

309 rows × 15 columns

In [30]:
```
# splitting the data...
train_x, test_x, train_y, test_y = train_test_split(train_X, train_Y, test_size=0.2)

classifier = GaussianNB()
classifier.fit(train_x, train_y)

# predicting
Y_predicted = classifier.predict(test_x)
print("Accuracy = ", metrics.accuracy_score(test_y, Y_predicted))
```

```
Accuracy =  0.9193548387096774
```

Conclusion

Classifier gave prediction with accuracy of 0.8870967741935484

In [ ]:

2. Implement a Neural Network Classifier (Back Propagation Network(BPN)) involving at least one hidden layer and trace the BPN algorithm assuming a target class for one epoch. Compare the evaluation of the network on a test set by varying the number of hidden nodes/layers.

In [31]:

```python
import numpy as np


class Neuron:
    def __init__(self, n_inputs=None, weights=None):
        """ Creates a neuron with numbers of inputs as specified by either parameters ``n_inputs`` or ``weights``

        Parameters
        ----------
        n_inputs : int (default None, which implies infer from parameter ``weights``)
        Number of inputs to neuron
        weights : :obj:`numpy.ndarray` of numbers
            Initial weights (to inputs and bias) of neuron
        """

        if (n_inputs is None) and (weights is None):
            raise Exception("Either parameter 'n_inputs' or 'weights' must be specified")

        if n_inputs is not None:
            if n_inputs <= 0:
                raise Exception("Number of inputs to a layer must be positive")
            elif (weights is not None) and (n_inputs != len(weights)-1):
                raise Exception(f"Number of inputs specified in parameter 'n_inputs' ({n_inputs}) and 'weights' ({len(weights)}) don't match")

        if weights is not None:
            self.weights = weights
            self.n_inputs = len(self.weights)-1

        elif n_inputs:
            self.n_inputs = n_inputs
            self.weights = np.random.rand(n_inputs+1)


class Layer:
    """ Class representing a layer in a Multi-Layer Neural Network """

    def __init__(self, n_neurons, n_inputs):
        """ Create a fully-connected layer with ``n_neurons`` neurons each connected to ``n_inputs`` inputs

        Parameters
        ----------
        n_neurons : int
            Number of neurons in the layer
        n_inputs : int
            Number of inputs to each neuron in the layer (excluding bias term)
        """

        if n_neurons <= 0:
            raise Exception("Number of neurons in a layer must be positive")
```

```
        if n_inputs <= 0:
            raise Exception("Number of inputs to a layer must be positive")

        self.n_neurons = n_neurons
        self.n_inputs = n_inputs

        self.weights = np.random.rand(n_neurons, n_inputs+1)

        # Alternative way of accessing and interacting with neuron weights
        self.neurons = []
        for i_neuron in range(n_neurons):
            neuron = Neuron(weights=self.weights[i_neuron])
            self.neurons.append(neuron)


    def activate(self, input):
        """ Activates the layer with given input ``input`` and computes output

        Parameters
        ----------
        input : 1-D array_like
            Inputs to the neurons in the layer

        Returns
        -------
        1-D :obj:`numpy.ndarray` of numbers
            Outputs from the neurons in the layer
        """

        if len(input) != self.n_inputs:
            raise Exception(f"Number of inputs given ({len(input)}) doesn't match specified value ({self.n_inputs}) for layer")

        input = np.append(input, 1)
        net_input = self.weights.dot(input)  # weigth*input
        output = 1/(1 + np.exp(-net_input)) # Apply logisitic sigmoid on 'net_inputs'

        # store the output for the most recent activation of the layer
        self.output = output

        return output


class NeuralNetwork:

    def __init__(self, *n_nodes):
        """ Creates a neural network with neuron counts in each layer as specified by parameter ``n_nodes``

        Parameters
        ----------
        n_nodes : :obj:`list` of :obj:`int`
            Specifies the number of nodes in input layer, hidden layers and output layer in sequence
```

```
        Specifies the number of nodes in input layer, hidden layers and output layer in sequence
        """

        if len(n_nodes) < 3:
            raise Exception("Neural network must have minimum 3 layers: Input, Hidden and Output")

        if any(n_node <= 0 for n_node in n_nodes):
            raise Exception("Number of nodes in any layer must be positive")

        # Input Layer
        self.n_inputs = n_nodes[0]
        self.n_outputs = n_nodes[-1]

        # Neural Network Layers
        self.layers = []
        for n_inputs, n_neurons in zip(n_nodes, n_nodes[1:]):
            layer = Layer(n_neurons, n_inputs)
            self.layers.append(layer)


    def activate(self, input):
        """ Activates the neural network with given input ``input`` and compute output

        Parameters
        ----------
        input : 1-D array_like
            Inputs to the neural network

        Returns
        -------
        :obj:`numpy.ndarray` of numbers
            Outputs from the neural network
        """

        if len(input) != self.n_inputs:
            raise Exception(f"Number of inputs given ({len(input)}) doesn't match specified value ({self.n_inputs}) for network")

        layer_input = input

        for layer in self.layers:
            layer_output = layer.activate(layer_input)
            # output of current layer is input for next layer
            layer_input = layer_output

        # store the input, output for the most recent activation of the network
        self.input = np.array(input)
        self.output = layer_output

        return layer_output
```

```
    def propagate(self, learning_rate, target):
        """ Backpropogate errors and adjust weights for most recent activation of neural network

        Parameters
        ----------
        learning_rate : float
            Learning rate for back propogation algorithm

        target : 1-D array_like
            Target value for most recent input to neural network
        """

        if not hasattr(self, 'output'):
            raise Exception("Activate neural network with a training sample before calling 'propogate'")

        if len(target) != self.n_outputs:
            raise Exception(f"Number of outputs given ({len(target)}) doesn't match specified value ({self.n_outputs}) for network")

        # Compute the error for the output layer
        output_layer = self.layers[-1]
        output_layer.error = self.output * (1 - self.output) * (target - self.output)

        # Compute the error for hidden layers, from last to first
        for hidden_layer, next_higher_layer in zip(self.layers[-2::-1], self.layers[-1:0:-1]):
            hidden_layer.error = hidden_layer.output * (1 - hidden_layer.output) * (next_higher_layer.weights.T[:-1].dot(next_higher_layer.error))

        # Update the weights for first hidden layer
        first_hidden_layer = self.layers[0]
        first_hidden_layer_delta_weights = learning_rate * first_hidden_layer.error.reshape((-1,1)).dot(self.input.reshape(1, -1))
        first_hidden_layer.weights[:, :-1] += first_hidden_layer_delta_weights

        # Update the weights for all the other hidden layers
        for hidden_layer, previous_lower_layer in zip(self.layers[1:], self.layers):
            hidden_layer_delta_weights = learning_rate * hidden_layer.error.reshape((-1,1)).dot(previous_lower_layer.output.reshape(1, -1))
            hidden_layer.weights[:, :-1] += hidden_layer_delta_weights

        # Update the weights for all layers in the network
        for layer in self.layers:
            layer_delta_bias = learning_rate*layer.error
            layer.weights[:, -1] += layer_delta_bias
```

In [32]:

```
learning_rate = 0.3
n_iterations = 30000

neural_network_OR = NeuralNetwork(2, 3, 2, 1)
neural_network_AND = NeuralNetwork(2, 3, 2, 1)
neural_network_XOR = NeuralNetwork(2, 3, 2, 1)

# training neural network...
for i_iteration in range(n_iterations):
```

```
        # XOR(0,0) = 0
        neural_network_XOR.activate([0,0])
        neural_network_XOR.propagate(learning_rate, [0])
        # XOR(0,1) = 1
        neural_network_XOR.activate([0,1])
        neural_network_XOR.propagate(learning_rate, [1])
        # XOR(1,0) = 1
        neural_network_XOR.activate([1,0])
        neural_network_XOR.propagate(learning_rate, [1])
        # XOR(1,1) = 1
        neural_network_XOR.activate([1,1])
        neural_network_XOR.propagate(learning_rate, [0])

        # OR(0,0) = 0
        neural_network_OR.activate([0,0])
        neural_network_OR.propagate(learning_rate, [0])
        # OR(0,1) = 1
        neural_network_OR.activate([0,1])
        neural_network_OR.propagate(learning_rate, [1])
        # OR(1,0) = 1
        neural_network_OR.activate([1,0])
        neural_network_OR.propagate(learning_rate, [1])
        # OR(1,1) = 1
        neural_network_OR.activate([1,1])
        neural_network_OR.propagate(learning_rate, [1])

        # AND(0,0) = 0
        neural_network_AND.activate([0,0])
        neural_network_AND.propagate(learning_rate, [0])
        # AND(0,1) = 0
        neural_network_AND.activate([0,1])
        neural_network_AND.propagate(learning_rate, [0])
        # AND(1,0) = 0
        neural_network_AND.activate([1,0])
        neural_network_AND.propagate(learning_rate, [0])
        # AND(1,1) = 1
        neural_network_AND.activate([1,1])
        neural_network_AND.propagate(learning_rate, [1])


# test the XOR neural network
print("XOR GATE")
input = [0, 0]
print(f"input: {input}, output:{neural_network_XOR.activate(input)}")
input = [0, 1]
print(f"input: {input}, output:{neural_network_XOR.activate(input)}")
input = [1, 0]
print(f"input: {input}, output:{neural_network_XOR.activate(input)}")
input = [1, 1]
print(f"input: {input}, output:{neural_network_XOR.activate(input)}")
```

In [ ]:

```
# test the OR neural network
```

In [ ]:

```
input = [0, 0]
```

In [ ]:

```
input = [0, 1]
print(f"input: {input}, output:{neural_network_OR.activate(input)}")
```

In [ ]:

```
print(f"input: {input}, output:{neural_network_OR.activate(input)}")
```

In [ ]:

```
print(f"input: {input}, output:{neural_network_OR.activate(input)}")
```

In [ ]:

```
# test the AND neural network
print("\nAND GATE")
```

In [ ]:

```
print(f"input: {input}, output:{neural_network_AND.activate(input)}")
```

In [ ]:

```
print(f"input: {input}, output:{neural_network_AND.activate(input)}")
```

In [ ]:

```
print(f"input: {input}, output:{neural_network_AND.activate(input)}")
input = [1, 1]
```

In [ ]: