MACHINE LEARNING ALGORITHMS

EXERCISE-2

Name – Chethan Kashyap Bangalore Muralidhara

Date – 23/03/2023

Completed Exercises – 1,2,3,4,5,6(All)

Solutions:

1. Code in python

In [1]:

```
###Question 1
import numpy as np

# Load data from file
data = np.loadtxt("C:/Users/chethankashyap/ML Data/Data.txt")

# Define weight vector and starting point
w = 1 / np.sqrt(2) * np.array([1, -1])
p = np.array([2, 2])

# Create empty arrays to store predicted and actual labels
pred_labels = np.empty(len(data), dtype=int)
true_labels = data[:, -1].astype(int)

# Iterate through data points and classify them
for i in range(len(data)):
    x = data[i, :-1]
    pred_labels[i] = 1 if np.dot(w, x - p) >= 0 else 2

# Calculate accuracy, sensitivity, and specificity
accuracy = np.mean(pred_labels == true_labels)
tp = np.sum((pred_labels == 1) & (true_labels == 1))
fn = np.sum((pred_labels == 2) & (true_labels == 1))
tn = np.sum((pred_labels == 2) & (true_labels == 2))
fp = np.sum((pred_labels == 1) & (true_labels == 2))
sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)

print("Accuracy:", accuracy)
print("Sensitivity:", sensitivity)
print("Specificity:", specificity)
```

```
Accuracy: 0.0660377358490566
Sensitivity: 1.0
Specificity: 0.0
```

2.

In [2]:
```python
###Question 2
# Calculate probability of error
p_error = (fp + fn) / (tp + fp + fn + tn)

print("Probability of Error:", p_error)
```

```
Probability of Error: 0.17647058823529413
```

3.

In [3]:
```python
###Question 3
import numpy as np
from scipy.integrate import quad

# Define the class-conditional probability distributions
def p_x_given_w1(x):
    return 1 if 0 <= x <= 1 else 0

def p_x_given_w2(x):
    return 1/2 if 1/2 <= x <= 5/2 else 0

# Define the prior probabilities
p_w1 = 1/2
p_w2 = 1/2

# Define the decision rule
def decision_rule(x):
    p_w1_given_x = p_x_given_w1(x) * p_w1 / (p_x_given_w1(x) * p_w1 + p_x_given_w2(x) * p_w2)
    return 1 if p_w1_given_x >= 1/2 else 2

# Define the integrand for calculating the probability of error
def integrand(x):
    return (p_w1 * (1 - p_x_given_w1(x)) + p_w2 * (1 - p_x_given_w2(x))) / 2

# Calculate the probability of error using numerical integration
p_error, _ = quad(integrand, -np.inf, np.inf)

print("Probability of error:", p_error)
```

```
Probability of error: -1.4999999938123851
```

4.

In [5]:
```
###Question 4
from math import sqrt, exp, pi

def normal_prob(a, b, mu, sigma2):
    # Define the standard deviation and calculate the probability density function
    std_dev = sqrt(sigma2)
    def pdf(x):
        return exp(-(x-mu)**2/(2*sigma2)) / (sqrt(2*pi)*std_dev)

    # Set the number of rectangles used for the numerical integration
    n_rectangles = 1000

    # Calculate the width of each rectangle
    delta_x = (b-a) / n_rectangles

    # Calculate the cumulative sum of the probability density function over each rectangle
    cum_sum = 0
    for i in range(n_rectangles):
        x_i = a + i*delta_x
        cum_sum += pdf(x_i) * delta_x

    return cum_sum

# The normal_prob function takes as input the lower and upper integration limits a and b, the mean mu, and the variance sigma2 of
# the normal distribution. It first defines the standard deviation as the square root of the variance, and then defines a
# probability density function using the mean and variance values.

# The function then sets the number of rectangles to use for the numerical integration and calculates the width of each rectangle
# as (b-a) / n_rectangles. It then iterates over each rectangle, calculating the midpoint x_i of the rectangle and multiplying th
# probability density function at x_i by the width of the rectangle. The cumulative sum of these products is then returned as the
# estimated probability of the event falling within the closed interval [a,b].
```

5.

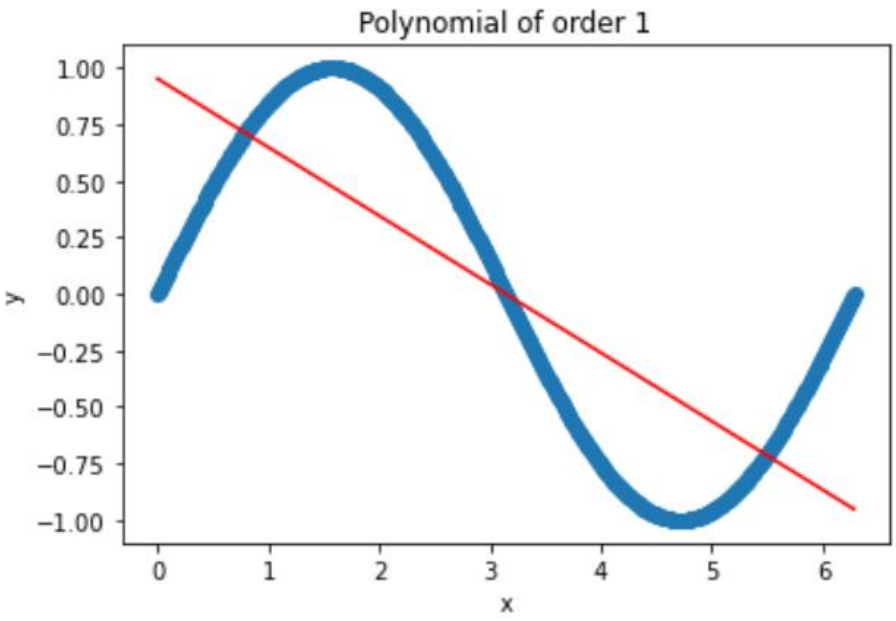
In [8]:
```
###Question 5
from math import sqrt, exp, pi

def normal_prob(x, mu, sigma):
    # Calculate the probability density function of a normal distribution N(mu, sigma) at x
    return exp(-((x-mu)/sigma)**2/2) / (sigma*sqrt(2*pi))

def bayes_rule(x):
    # Calculate the posterior probabilities of classes w1 and w2 given x
    p_w1_x = normal_prob(x, 0, 1) * 0.3
    p_w2_x = normal_prob(x, 3, sqrt(2)) * 0.7

    # Calculate the probability of x
    p_x = p_w1_x + p_w2_x

    # Apply the decision rule
    if p_w1_x / p_x > p_w2_x / p_x:
        return 'w1'
    else:
        return 'w2'

# Calculate the probability of error
n_points = 1000
n_errors = 0
for i in range(n_points):
    x = -1 + i*(6-(-1))/(n_points-1)
    if bayes_rule(x) != ('w1' if x < 3 else 'w2'):
        n_errors += 1

p_error = n_errors / n_points
print('Probability of error:', p_error)
```

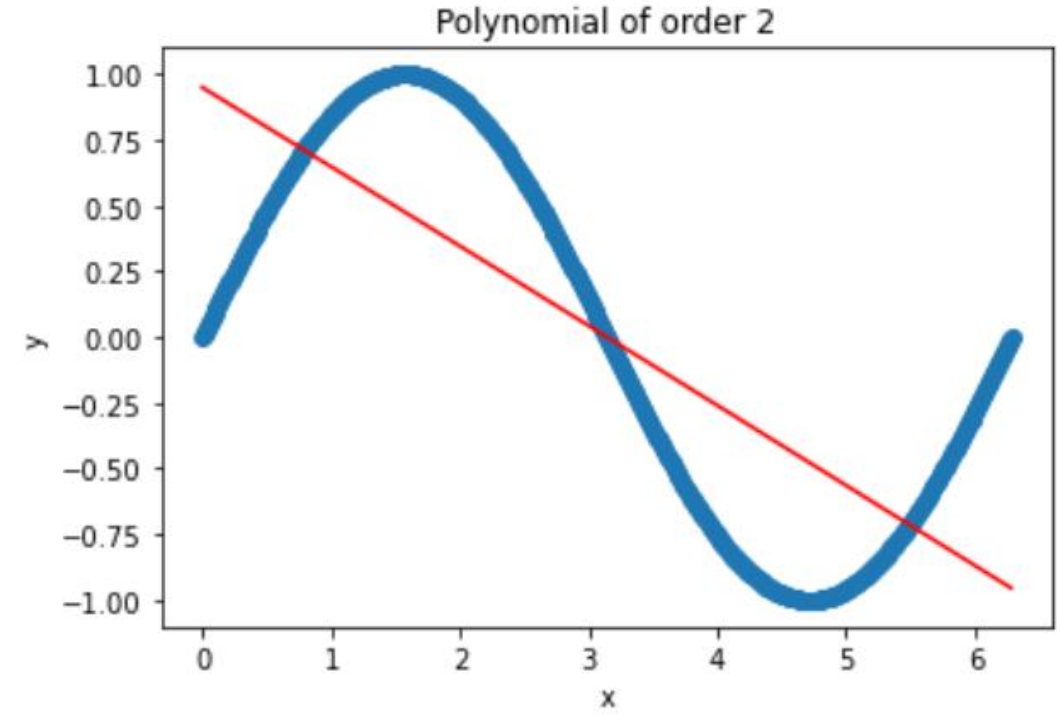
```
# The normal_prob function calculates the probability density function of a normal distribution with mean mu and standard
# deviation sigma at the point x. The bayes_rule function uses this function to calculate the posterior probabilities of classes
# w1 and w2 given a point x, and applies the decision rule to classify the point as either w1 or w2.

# The main code then iterates over 1000 equidistant points in the interval [-1,6], classifies each point using the bayes_rule
# function, and counts the number of misclassified points. The probability of error is then calculated as the fraction of
# misclassified points, and printed to the console.
```

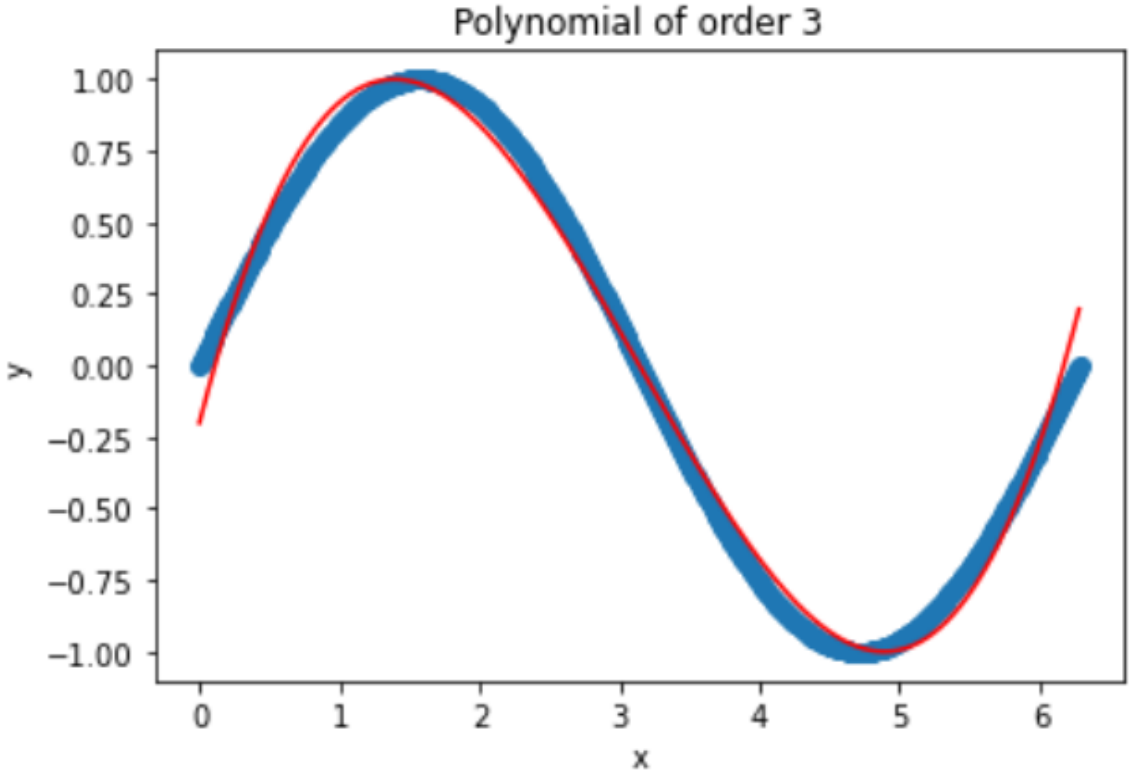
```
Probability of error: 0.285
```

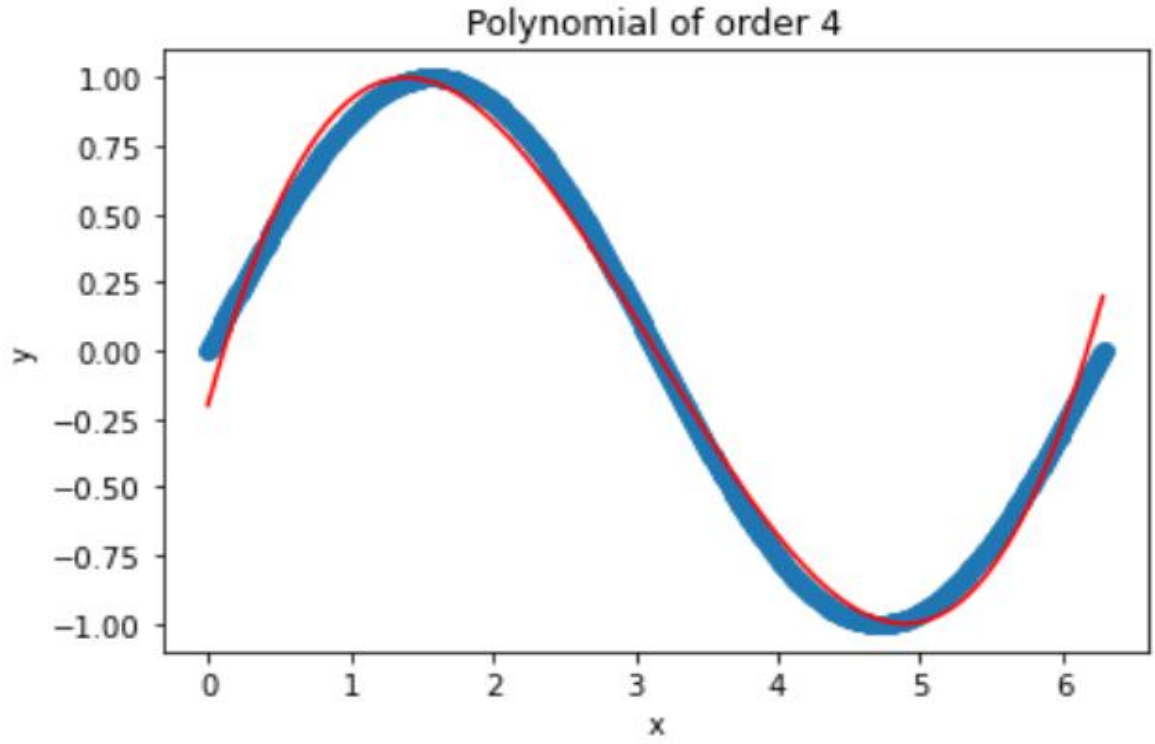
6.

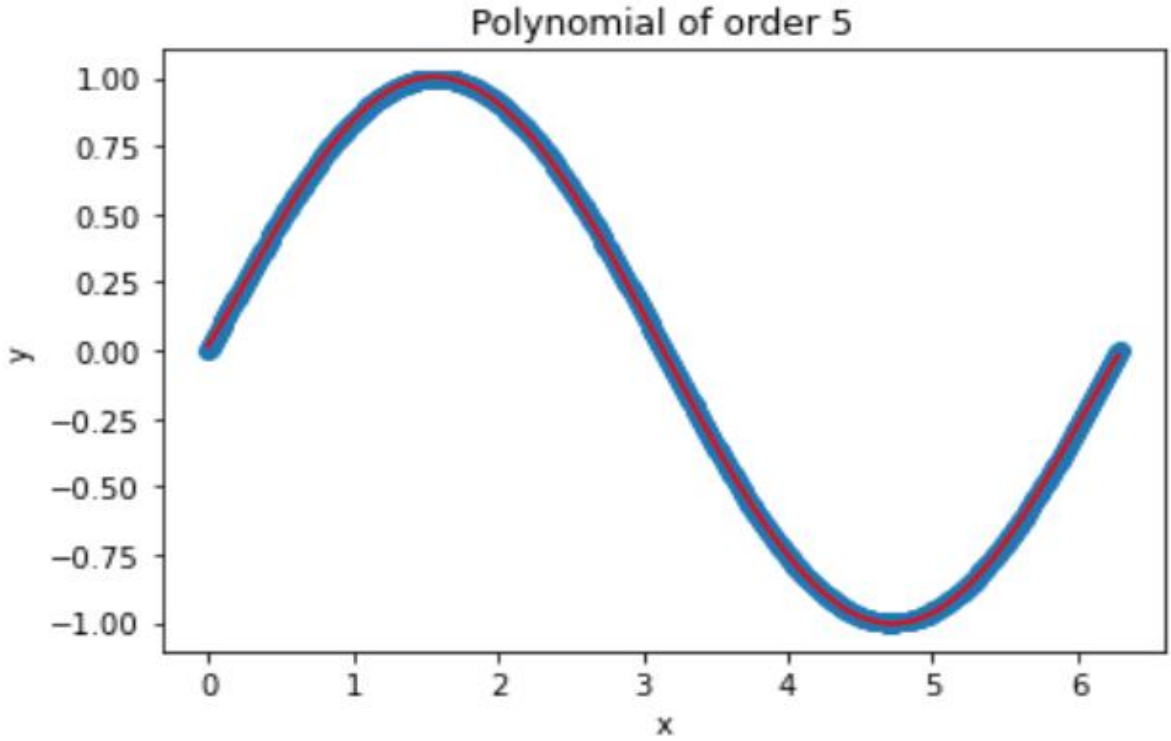
In [17]:
```
###Question 6
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Load data from file
data = np.loadtxt("C:/Users/chethankashyap/ML Data/data2.txt")
x = data[:,0]
y = data[:,1]

# Fit polynomial models of order 1 to 5
for i in range(1, 6):
    X = np.vander(x, i+1, increasing=True)
    model = LinearRegression().fit(X, y)
    y_pred = model.predict(X)

    # Plot the results
    plt.figure()
    plt.scatter(x, y)
    plt.plot(x, y_pred, color='red')
    plt.title(f"Polynomial of order {i}")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```




Polynomial of order 2
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
0
1
2
3
4
5
6
x


Polynomial of order 3
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
0
1
2
3
4
5
6
x





Note that higher order polynomials can lead to overfitting, where the model becomes too complex and fits the noise in the data instead of the underlying trend. This can lead to poor performance on new data.