CAMBRIDGE INSTITUTE OF TECHNOLOGY
KR PURAM,BANGLORE -560036 – KARNATAKA

AI AND ML APPLICATION DEVELOPMENT LABORATORY
(Effective from the academic year 2018 -2019)

SEMESTER – VII

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

LAB MANUAL

SUBJECT- AI AND ML APPLICATION DEVELOPMENT LABORATORY

CODE- 18AIL76

BY:-

Dr. Buddesab
Associate professor,
Dept of AIML

Prof. Syed Hayath
Assistant professor,
Dept of AIML

Program 1

Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions.

Dataset - link

```
import pandas as pd import numpy as np from
sklearn import datasets from sklearn.neighbors
import KNeighborsClassifier from
sklearn.model_selection import train_test_split

# Importing Dataset dataset =
datasets.load_iris() X =
pd.DataFrame(dataset.data)
y = pd.DataFrame(dataset.target)

# Splitting The Dataset Into Training Set And Testing Set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Model Fit And Prediction knn =
KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train.values.ravel())

accuracy_train = knn.score(X_train, y_train)
accuracy_test = knn.score(X_test, y_test)

print("Training Accuracy\n", accuracy_train) print("Testing
Accuracy\n", accuracy_test)

available_class = pd.DataFrame(dataset.target_names)
print("Dataset Classes\n", available_class)

example = np.array([5.7, 3, 4.2, 1.2]) example
= example.reshape(1, -1)
print("Input Sample\n", example)

example_prediction = int(knn.predict(example))
print("Predicted Class\n", available_class[0][example_prediction])
```

OUTPUT –

```
Training Accuracy
 0.9583333333333334
Testing Accuracy
 1.0
Dataset Classes
            0
0      setosa
1      versicolor
2      virginica
Input Sample
 [[5.7 3.  4.2 1.2]]
Predicted Class
 Versicolor
```

Program - 2

Develop a program to apply K-means algorithm to cluster a set of data stored in .CSV file. Use the same data set for clustering using EM algorithm. Compare the results of these two algorithms and comment on the quality of clustering.

Dataset- same as 1st

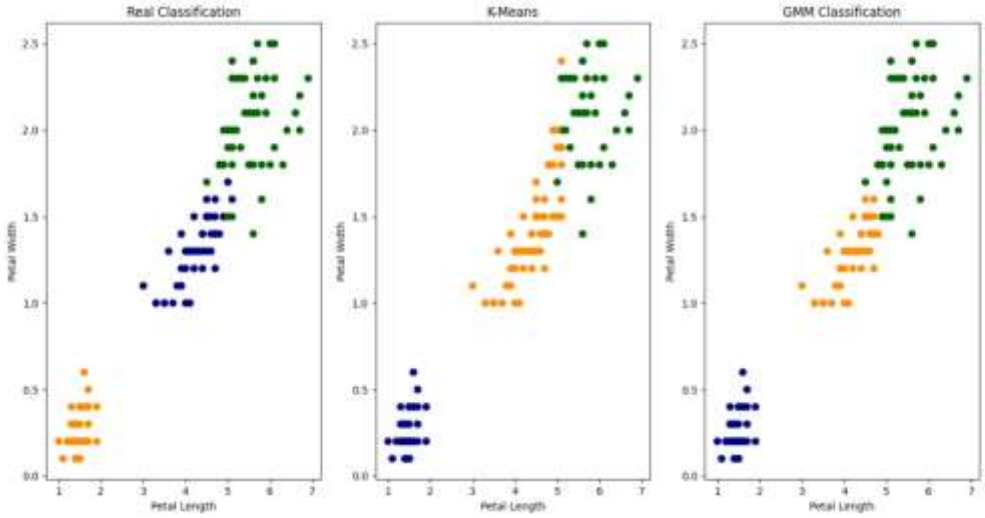
```
import sklearn.metrics as sm import pandas as
pd import numpy as np import
matplotlib.pyplot as plt from sklearn import
datasets from sklearn import preprocessing
from sklearn.cluster import KMeans from
sklearn.mixture import GaussianMixture

# Importing Dataset dataset
= datasets.load_iris()

# Attribute Variable(s)
X = pd.DataFrame(dataset.data)
X.columns = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width'] #
Target Variable
y = pd.DataFrame(dataset.target) y.columns
= ['Targets']
```

```
plt.figure(figsize=(14, 7))
colormap = np.array(['darkorange', 'navy', 'darkgreen'])

# REAL PLOT
plt.subplot(1, 3, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real Classification') plt.xlabel('Petal Length') plt.ylabel('Petal
Width')

# K-PLOT
plt.subplot(1, 3, 2) model =
KMeans(n_clusters=3)
model.fit(X)
predY = np.choose(model.labels_, [0, 1, 2]).astype(np.int64)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[predY], s=40)
plt.title('K-Means') plt.xlabel('Petal Length') plt.ylabel('Petal Width')
print('K-Means Algorithm')
print('The Accuracy Score Of K-Mean Algorithm: ', sm.accuracy_score(y, model.labels_))
print('The Confusion Matrix Of K-Mean Algorithm:\n', sm.confusion_matrix(y,
model.labels_))

# EM PLOT
scaler = preprocessing.StandardScaler()
scaler.fit(X) xsa = scaler.transform(X) xs =
pd.DataFrame(xsa, columns=X.columns)
gmm = GaussianMixture(n_components=3)
gmm.fit(xs) y_gmm =
gmm.predict(xs)
plt.subplot(1, 3, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_gmm], s=40)
plt.title('GMM Classification') plt.xlabel('Petal Length') plt.ylabel('Petal Width')
print('EM Algorithm') print('The Accuracy Score Of EM Algorithm: ',
sm.accuracy_score(y, y_gmm)) print('The Confusion Matrix Of EM Algorithm:\n',
sm.confusion_matrix(y, y_gmm)) plt.show()
```

Output –

```
K-Means Algorithm
The Accuracy Score Of K-Mean Algorithm:  0.24 The
Confusion Matrix Of K-Mean Algorithm:
 [[ 0 50  0]
 [48  0  2]
 [14  0 36]]
EM Algorithm
The Accuracy Score Of EM Algorithm:  0.3333333333333333 The
Confusion Matrix Of EM Algorithm:
 [[ 0 50  0]
 [45  0  5]
```

[0 0 50]]



Program – 3

Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

Dataset - [link]()

```
import numpy as np import
pandas as pd
import matplotlib.pyplot as plt

data_frame = pd.read_csv('data/tips.csv')

feature_array = np.array(data_frame.total_bill) label_array
= np.array(data_frame.tip)

total_cases = feature_array.shape[0]

y_matrix = np.matrix(label_array).T
x_matrix = np.hstack((np.ones((total_cases, 1)), np.matrix(feature_array).T))

y_predicted = np.zeros(total_cases)

# get weight matrix def
get_weight_matrix(current_x, x_matrix, tau):
weight_matrix = np.mat(np.eye(total_cases))
for j in range(total_cases):        distance_vector
= current_x - x_matrix[j]
      weight_matrix[j, j] = np.exp(-1 * distance_vector*distance_vector.T / (2.0 * tau ** 2))
return weight_matrix
```

get theta matrix def get_theta_matrix(current_x,
x_matrix, y_matrix, tau):
   weight_matrix = get_weight_matrix(current_x, x_matrix, tau)
   theta_matrix = np.linalg.inv(x_matrix.T * (weight_matrix * x_matrix)) * x_matrix.T *
weight_matrix * y_matrix
   return theta_matrix

get final y predicted def get_y_for_all_x(current_x, x_matrix,
y_matrix, tau):   theta_matrix = get_theta_matrix(current_x,
x_matrix, y_matrix, tau)   y_predicted_result = current_x *
theta_matrix   return y_predicted_result

# main loop for i in range(total_cases):   y_predicted[i] =
get_y_for_all_x(x_matrix[i], x_matrix, y_matrix, 1)

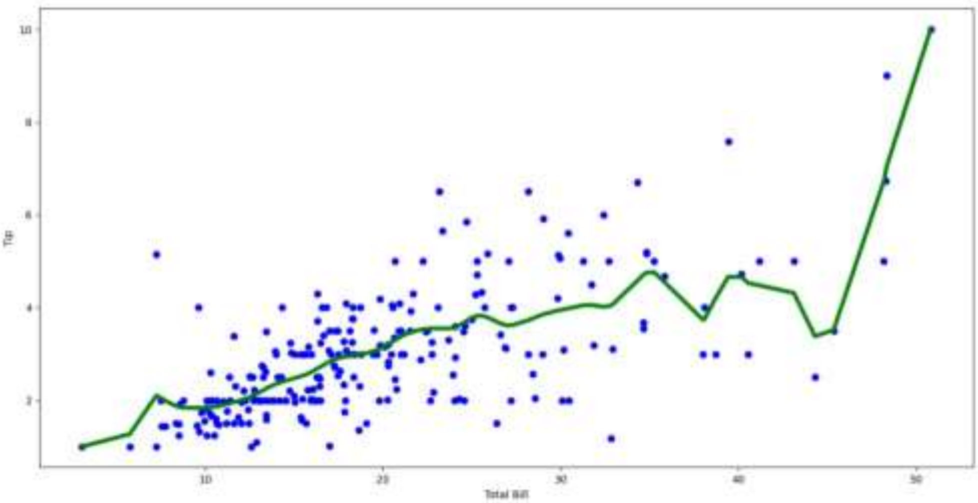
Plotting the results sorted_indexes =
x_matrix[:, 1].argsort(0)
sorted_x = x_matrix[sorted_indexes][:, 0]

figure = plt.figure()
axes = figure.add_subplot(1, 1, 1)
axes.scatter(feature_array, label_array, color='blue')

axes.plot(sorted_x[:, 1], y_predicted[sorted_indexes], color='green', linewidth=4)

plt.xlabel('Total Bill') plt.ylabel('Tip')
plt.show()

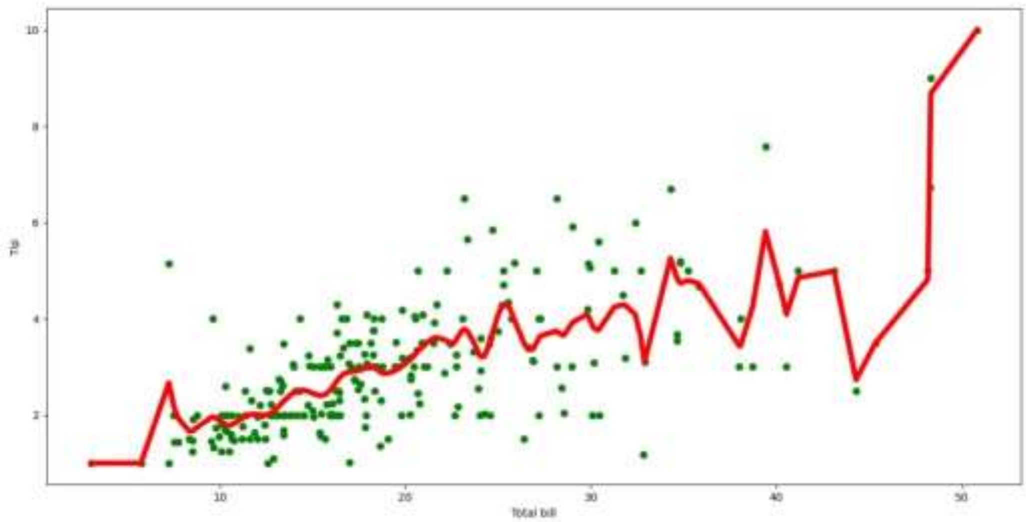
Output-



Or

```
import matplotlib.pyplot as plt
import pandas as pd import
numpy as np def kernel(point,
xmat, k):     m,n =
np.shape(xmat)     weights =
np.mat(np.eye((m)))     for j in
range(m):         diff = point - X[j]
        weights[j,j] = np.exp(diff*diff.T/(-2.0*k**2))
return weights def localWeight(point, xmat, ymat,
k):
    wei = kernel(point,xmat,k)
    W = (X.T*(wei*X)).I*(X.T*(wei*ymat.T))
    return W

def localWeightRegression(xmat, ymat, k):     m,n =
np.shape(xmat)     ypred = np.zeros(m)     for i in
range(m):         ypred[i] =
xmat[i]*localWeight(xmat[i],xmat,ymat,k)     return ypred

# load data points data =
pd.read_csv('tips.csv') bill =
np.array(data.total_bill)
tip = np.array(data.tip)

#preparing and add 1 in bill
mbill = np.mat(bill) mtip =
np.mat(tip) m=
np.shape(mbill)[1] one =
np.mat(np.ones(m)) X =
np.hstack((one.T,mbill.T))
#set k here
ypred = localWeightRegression(X,mtip,0.5)
SortIndex = X[:,1].argsort(0)
xsort = X[SortIndex][:,0] fig =
plt.figure() ax =
fig.add_subplot(1,1,1)
ax.scatter(bill,tip,
color='green')
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5)
plt.xlabel('Total bill') plt.ylabel('Tip')
plt.show()
```

Output -



Program - 4

Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float) y =
y/100

#Sigmoid Function def sigmoid
(x):     return 1/(1 + np.exp(-x))
#Derivative of Sigmoid Function
def derivatives_sigmoid(x):
   return  x  *  (1  -  x)
#Variable    initialization
epoch=10000        lr=0.1
inputlayer_neurons  =  2
hiddenlayer_neurons = 3
output_neurons     =     1
#weight initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons)) for
i in range(epoch):
#Forward Propogation
```

```
    hinp=np.dot(X,wh)
hlayer_act = sigmoid(hinp)
outinp=np.dot(hlayer_act,wout)
    output = sigmoid(outinp)

#Backpropagation

    EO = y-output
    outgrad = derivatives_sigmoid(output)
d_output = EO* outgrad     EH =
d_output.dot(wout.T)     hiddengrad =
derivatives_sigmoid(hlayer_act
d_hiddenlayer = EH * hiddengrad     wout +=
hlayer_act.T.dot(d_output) *lr     wh +=
X.T.dot(d_hiddenlayer) *lr     print("Input: \n"
+ str(X)) y=y*100 output=output*100
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

Output

```
 Input:
[[2. 9.]
 [1. 5.]
 [3. 6.]]
Actual Output:
[[92.]
 [86.]
 [89.]]
Predicted Output:
[[89.77293616]
 [87.35314244]
 [89.77030837]]
```

Program – 5

Demonstrate Genetic algorithm by taking a suitable data for any simple application.

```
import random

# Define the knapsack problem parameters
max_weight = 50 items
= [
   {'name': 'item1', 'weight': 10, 'value': 60},
   {'name': 'item2', 'weight': 20, 'value': 100},
   {'name': 'item3', 'weight': 30, 'value': 120},
   {'name': 'item4', 'weight': 15, 'value': 50},
   {'name': 'item5', 'weight': 25, 'value': 60},
]

# Genetic algorithm parameters
population_size = 100 generations
= 50
mutation_rate = 0.1

# Initialize a population of random solutions def
initialize_population(pop_size, item_count):
   population = []     for _ in range(pop_size):         solution =
[random.randint(0, 1) for _ in range(item_count)]
population.append(solution)     return population

# Calculate the fitness of a solution def fitness(solution):     total_weight =
sum(item['weight'] for item, bit in zip(items, solution) if bit)     total_value =
sum(item['value'] for item, bit in zip(items, solution) if bit)     return
total_value if total_weight <= max_weight else 0

# Perform single-point crossover def
crossover(parent1, parent2):
   crossover_point = random.randint(1, len(parent1) - 1)     child1
= parent1[:crossover_point] + parent2[crossover_point:]
child2 = parent2[:crossover_point] + parent1[crossover_point:]
return child1, child2

# Mutate a solution def mutate(solution,
mutation_rate):     mutated_solution = []
for bit in solution:         if
random.random() < mutation_rate:
mutated_solution.append(1 - bit)  # Flip
the bit with a probability of mutation_rate
else:
         mutated_solution.append(bit)
return mutated_solution

# Main genetic algorithm loop population =
initialize_population(population_size, len(items)) for generation in
range(generations):     population = sorted(population, key=lambda x:
```

```
fitness(x), reverse=True)     new_population = population[:population_size
// 2]

    for _ in range(population_size // 2):         parent1,
parent2 = random.choices(population, k=2)         child1,
child2 = crossover(parent1, parent2)         child1 =
mutate(child1, mutation_rate)         child2 =
mutate(child2, mutation_rate)
        new_population.extend([child1, child2])

    population = new_population

best_solution = max(population, key=fitness) best_value = sum(item['value']
for item, bit in zip(items, best_solution) if bit) best_weight =
sum(item['weight'] for item, bit in zip(items, best_solution) if bit)

print("Best solution:", best_solution) print("Total
value:", best_value)
print("Total weight:", best_weight)
```

Output –

Best solution: [0, 1, 1, 0, 0]
Total value: 220
Total weight: 50

Program – 6

Demonstrate Q learning algorithm with suitable assumption for a problem statement.

```
import numpy as np

# Estado terminal
terminal = 5

# Possiveis acoes
actions = ['UP','DW','LF','RG']

# Recompensas rws =
np.array([-1]*6)
rws[5] = 10
```

```
# Duas trajetorias
paths = [(0, ['UP','UP','UP','RG']), (4, ['RG','RG','LF','UP'])]

# Constantes alpha
= 0.5 gamma = 0.8

def print_value(value):
    print('[' + str(value[2]) + ' ' + str(value[5]))
print(str(value[1]) + ' ' + str(value[4]))     print(str(value[0])
+ ' ' + str(value[3]) + ']\n')

def update_value(value, state, action):
    index = actions.index(action)
next_state = state     rw = 0

    if action == 'UP':         if
state == 2 or state == 5:
rw = -10         else:
            next_state = state + 1

    elif action == 'DW':
        if state == 0 or state == 3:
            rw = -10
else:
            next_state = state - 1

    elif action == 'LF':         if state == 0 or
state == 1  or state == 2:
            rw = -10
else:
            next_state = state - 3

    elif action == 'RG':         if state == 3 or
state == 4 or state == 5:            rw = -10
else:
            next_state = state + 3

    if rw == 0:         rw =
rws[next_state]

    value[index][state] = value[index][state] + alpha * (rw + gamma *
max(value[i][next_state] for i in range(4)) - value[index][state])

    return value, next_state
```

```
def return_policy(value):
policy = np.array(['   ']*6)
policy[5] = '+10'

   for     state     in     range(5):                               policy[state]     =
actions[np.argmax([value[action][state] for action in range(4)])]

   print(policy[2] + ' ' + policy[5])
print(policy[1] + ' ' + policy[4])
   print(policy[0] + ' ' + policy[3]+ '\n')

def main():
   # Inicializar matriz Q com valores 0, considerando as quatro acoes
value = [np.zeros(6),np.zeros(6),np.zeros(6),np.zeros(6)]

   for i in range(len(paths)):
state = paths[i][0]
      actions = paths[i][1]

      for action in actions:            value, state =
update_value(value, state, action)

         if state == terminal:
            break

      # Acao UP
      print_value(value[0])

      # Acao DW
print_value(value[1])        # Acao LF
print_value(value[2])

      # Acao RG
print_value(value[3])

      # Politica
return_policy(value)
 if __name__ ==
'__main__':          main()
```

Output-

[-5.0 0.0
-0.5 0.0
-0.5 0.0]

[0.0 0.0
0.0 0.0
0.0 0.0]

[0.0 0.0
0.0 0.0
0.0 0.0]

[5.0 0.0
0.0 0.0
0.0 0.0]

RG +10 DW
UP
DW UP

[-5.0 0.0
1.25 0.0
-0.5 0.0]

[0.0 0.0
0.0 0.0
0.0 0.0]

[0.0 0.0
0.0 -0.5
0.0 0.0]

[5.0 0.0
0.0 -7.5
0.0 0.0]

RG +10
UP UP
DW UP