Machine Learning Lab Programs

Program 1

python

```
import pandas as pd
data = pd.read_csv('Dataset10.csv')
data = data.drop(['Day'], axis = 1)

attribute=np.array(data)[:,:-1]
print(attribute)
target=np.array(data)[:,-1]
print(target)

def train(att,tar):
  for i,val in enumerate(tar):
    if val=='Yes':
      specific_h=att[i].copy()
      break
  for i,val in enumerate(att):
    if tar[i]=='Yes':
      for x in range(len(specific_h)):
        if val[x]!=specific_h[x]:
          specific_h[x]='?'
        else:
          pass
  return specific_h

print(train(attribute,target))
```

Program 2

python

```
import numpy as np
import pandas as pd

data = pd.DataFrame(pd.read_csv('ENJOYSPORT.csv'))

concepts = np.array(data.iloc[:, :-1])
```

```
print(concepts)
target = np.array(data.iloc[:, -1])
print(target)

def learn(concepts, target):
    specific_h = concepts[9].copy()
    print("initialization of specific_h and general_h")
    print(specific_h)
    general_h = [["?" for i in range(len(specific_h))] for _ in
range(len(specific_h))]
    print(general_h)

    for i, h in enumerate(concepts):
        if target[i] == 1:  # Check if this condition is correct
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'
        elif target[i] == 0:  # Check if this condition is correct
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'
        print("steps of Candidate Elimination Algorithm", i+1)
        print(specific_h)
        print(general_h)

    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?',
'?', '?']]
    for i in indices:
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    return specific_h, general_h

s_final, g_final = learn(concepts, target)
print("Final Specific_h:", s_final, sep="\n")
print("Final General_h:", g_final, sep="\n")
```

Program 3

python

```
import csv
def load_csv(filename):
    lines=csv.reader(open(filename,"r"));
    dataset = list(lines)
    headers = dataset.pop(0)
    return dataset,headers
dataset, features = load_csv('PlayTennis.csv')

class Node:
    def __init__(self,attribute):
        self.attribute=attribute
        self.children=[]
        self.answer=""

def subtables(data,col,delete):
    dic={}
    coldata=[row[col] for row in data]
    attr=list(set(coldata))

    counts=[0]*len(attr)
    r=len(data)
    c=len(data[0])
    for x in range(len(attr)):
        for y in range(r):
            if data[y][col]==attr[x]:
                counts[x]+=1
    for x in range(len(attr)):
        dic[attr[x]]=[[0 for i in range(c)] for j in range(counts[x])]
        pos=0
        for y in range(r):
            if data[y][col]==attr[x]:
                if delete:
                    del data[y][col]
                dic[attr[x]][pos]=data[y]
                pos+=1
    return attr,dic

import math
def entropy(S):
    attr=list(set(S))
```

```
    if len(attr)==1:
        return 0

    counts=[0,0]
    for i in range(2):
        counts[i]=sum([1 for x in S if attr[i]==x])/(len(S)*1.0)

    sums=0
     for cnt in counts:
        sums+=-1*cnt*math.log(cnt,2)
    return sums

def compute_gain(data,col):
    attr,dic = subtables(data,col,delete=False)

    total_size=len(data)
    entropies=[0]*len(attr)
    ratio=[0]*len(attr)

    total_entropy=entropy([row[-1] for row in data])
    for x in range(len(attr)):
        ratio[x]=len(dic[attr[x]])/(total_size*1.0)
        entropies[x]=entropy([row[-1] for row in dic[attr[x]]])
        total_entropy-=ratio[x]*entropies[x]
    return total_entropy

def build_tree(data,features):
    lastcol=[row[-1] for row in data]
    if(len(set(lastcol)))==1:
        node=Node("")
        node.answer=lastcol[0]
        return node

    n=len(data[0])-1
    gains=[0]*n
    for col in range(n):
        gains[col]=compute_gain(data,col)
    split=gains.index(max(gains))
    node=Node(features[split])
    fea = features[:split]+features[split+1:]

    attr,dic=subtables(data,split,delete=True)
```

```
    for x in range(len(attr)):
        child=build_tree(dic[attr[x]],fea)
        node.children.append((attr[x],child))
    return node

def print_tree(node,level):
    if node.answer!="":
        print(" "*level,node.answer)
        return
     print(" "*level,node.attribute)
    for value,n in node.children:
        print(" "*(level+1),value)
        print_tree(n,level+2)

def classify(node,x_test,features):
    if node.answer!="":
        print(node.answer)
        return
    pos=features.index(node.attribute)
    for value, n in node.children:
        if x_test[pos]==value:
            classify(n,x_test,features)

'''Main program'''
dataset,features=load_csv("PlayTennis.csv")
features = features[1:]
dataset = [ele[1:] for ele in dataset]
print("\n Features: ", features)
print("\n Dataset: ", dataset)
node1=build_tree(dataset,features)

print("\n The decision tree for the dataset using ID3 algorithm is")
print_tree(node1,0)
testdata,features=load_csv("testdata.csv")
for xtest in testdata:
    print("\n The test instance:",xtest)
    print("\n The label for test instance:",end="   ")
    classify(node1,xtest,features)
```

Program 4

python

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) # maximum of X array longitudinally
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=500 # Setting training iterations
lr=0.1 # Setting learning rate
inputlayer_neurons = 2 # number of features in data set
hiddenlayer_neurons = 3 # number of hidden layers neurons
output_neurons = 1 # number of neurons at output layer

# weight and bias initialization
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))

# draws a random range of numbers uniformly of dim x*y
for i in range(epoch):
    # Forward Propogation
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+ bout
    output = sigmoid(outinp)

    #Backpropagation
    EO = y-output
    outgrad = derivatives_sigmoid(output)
```

```
    d_output = EO* outgrad
    EH = d_output.dot(wout.T)

    #how much hidden layer wts contributed to error
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad
    # dot product of next layer error and current layer output
    wout += hlayer_act.T.dot(d_output) *lr
    wh += X.T.dot(d_hiddenlayer) *lr

print("\n Input: \n" + str(X))
print("\n Actual Output: \n" + str(y))
print("\n Predicted Output: \n" ,output)
```

Program 5

python

```
import csv
import random
import math

def loadcsv(filename):
    lines = csv.reader(open(filename, "r"));
    dataset = list(lines)
    for i in range(len(dataset)):
    #converting strings into numbers for processing
        dataset[i] = [float(x) for x in dataset[i]]
    return dataset

def splitdataset(dataset, splitratio):
    #67% training size
    trainsize = int(len(dataset) * splitratio);
    trainset = []
    copy = list(dataset);
    while len(trainset) < trainsize:
    #generate indices for the dataset list randomly to pick ele for
    # training data
        index = random.randrange(len(copy));
        trainset.append(copy.pop(index))
    return [trainset, copy]
```

```
def separatebyclass(dataset):
    separated = {} #dictionary of classes 1 and 0
    #creates a dictionary of classes 1 and 0 where the values are
    #the instances belonging to each class
    for i in range(len(dataset)):
        vector = dataset[i]
        if (vector[-1] not in separated):
            separated[vector[-1]] = []
        separated[vector[-1]].append(vector)
    return separated

def mean(numbers):
    return sum(numbers)/float(len(numbers))

def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)

def summarize(dataset): #creates a dictionary of classes
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)];
    del summaries[-1] #excluding labels +ve or -ve
    return summaries

def summarizebyclass(dataset):
    separated = separatebyclass(dataset);
    #print(separated)
    summaries = {}
    for classvalue, instances in separated.items():
    #for key,value in dic.items()
    #summaries is a dic of tuples(mean,std) for each class value
        summaries[classvalue] = summarize(instances)
    #summarize is used to cal to mean and std
    return summaries

def calculateprobability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x-mean,2)/ (2*math.pow(stdev,2))))
    return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent

def calculateclassprobabilities(summaries, inputvector):
    # probabilities contains the all prob of all class of test data
```

```
    probabilities = {}
    for classvalue, classsummaries in summaries.items():
    #class and attribute information as mean and sd
        probabilities[classvalue] = 1
        for i in range(len(classsummaries)):
            mean, stdev = classsummaries[i] #take mean and sd of every attribute for
class 0 and 1 seperaely
            x = inputvector[i] #testvector's first attribute
            probabilities[classvalue] *= calculateprobability(x, mean, stdev);#use
normal dist
    return probabilities

def predict(summaries, inputvector): #training and test data is passed
    probabilities = calculateclassprobabilities(summaries, inputvector)
    bestLabel, bestProb = None, -1
    for classvalue, probability in probabilities.items():
        #assigns that class which has the highest prob
        if bestLabel is None or probability > bestProb:
            bestProb = probability
            bestLabel = classvalue
    return bestLabel

def getpredictions(summaries, testset):
    predictions = []
    for i in range(len(testset)):
        result = predict(summaries, testset[i])
        predictions.append(result)
    return predictions

def getaccuracy(testset, predictions):
    correct = 0
    for i in range(len(testset)):
        if testset[i][-1] == predictions[i]:
            correct += 1
    return (correct/float(len(testset))) * 100.0
def main():
    filename = 'diabetes.csv'
    splitratio = 0.67
    dataset = loadcsv(filename);
    trainingset, testset = splitdataset(dataset, splitratio)
print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),
len(trainingset), len(testset)))
```

```
    # prepare model
    summaries = summarizebyclass(trainingset);
    #print(summaries)
    # test model
    predictions = getpredictions(summaries, testset) #find the predictions of test
data with the training data
    accuracy = getaccuracy(testset, predictions)
    print('Accuracy of the classifier is : {0}%'.format(accuracy))

main()
```

Program 6

python

```
import numpy as np
import pandas as pd
import csv
from pgmpy.estimators import MaximumLikelihoodEstimator
from pgmpy.models import BayesianNetwork
from pgmpy.inference import VariableElimination
heartDisease = pd.read_csv("heart.csv")

heartDisease = heartDisease.replace('?',np.nan)
print('Sample instances from the dataset are given below')
print(heartDisease.head())

print('\n Attributes and datatypes')
print(heartDisease.dtypes)

print('\nLearning CPD using Maximum likelihood estimators')
model.fit(heartDisease,estimator=MaximumLikelihoodEstimator)

print('\n Inferencing with Bayesian Network:')
HeartDiseasetest_infer = VariableElimination(model)

print('\n 1. Probability of HeartDisease given evidence= restecg')
q1=HeartDiseasetest_infer.query(variables=['target'],evidence={'restecg':1})
print(q1)

print('\n 2. Probability of HeartDisease given evidence= cp ')
q2=HeartDiseasetest_infer.query(variables=['target'],evidence={'cp':2})
print(q2)
```

Program 7

python

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

dataset=load_iris()
X=pd.DataFrame(dataset.data)
X.columns=['Sepal_Length','Sepal_Width','Petal_Length','Petal_Width']

y=pd.DataFrame(dataset.target)
y.columns=['Targets']

plt.figure(figsize=(14,7))
colormap=np.array(['red','lime','black'])

#Real PLot
plt.subplot(1,3,1)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y.Targets],s=40)
plt.title('Real')

#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('KMeans')

#GMM 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_cluster_gmm=gmm.predict(xs)
```

```
plt.subplot(1,3,3)
plt.scatter(X.Petal_Length,X.Petal_Width,c=colormap[y_cluster_gmm],s=40)
plt.title('GMM Classification')

kmeans_labels = model.labels_
gmm_labels = gmm.fit_predict(X)

from sklearn.metrics import adjusted_rand_score, silhouette_score
#Silhouette Score
silhouette_kmeans = silhouette_score(y, kmeans_labels)
silhouette_gmm = silhouette_score(y, gmm_labels)

print(f'Silhouette Score for k-Means: {silhouette_kmeans}')
print(f'Silhouette Score for GMM: {silhouette_gmm}')
```

Program 8

python

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

iris = load_iris()
data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
data['target'] = iris.target

X = data.drop('target', axis=1)
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

k = 3  # Number of neighbors
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix: \n", cm)

correct_predictions = []
wrong_predictions = []

for i in range(len(y_test)):
    if y_test.iloc[i] == y_pred[i]:
        correct_predictions.append((X_test.iloc[i].tolist(), y_test.iloc[i],
y_pred[i]))
    else:
        wrong_predictions.append((X_test.iloc[i].tolist(), y_test.iloc[i], y_pred[i]))


print("\nCorrect Predictions:")
for cp in correct_predictions:
    print(f"Features: {cp[0]}, True Label: {cp[1]}, Predicted Label: {cp[2]}")

print("\nWrong Predictions:")
for wp in wrong_predictions:
    print(f"Features: {wp[0]}, True Label: {wp[1]}, Predicted Label: {wp[2]}")
```

Program 9

python

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

# Generate synthetic data
np.random.seed(0)
X = np.linspace(0, 10, 100)
y = np.sin(X) + np.random.normal(scale=0.5, size=X.shape)

plt.scatter(X, y, label='Data points')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Synthetic Dataset')
plt.legend()
```

```
plt.show()

def locally_weighted_regression(X, y, x_query, tau):
    m = X.shape[0]
    y_pred = np.zeros_like(x_query)
    for i, x_q in enumerate(x_query):
        # Calculate weights
        weights = np.exp(- (X - x_q) ** 2 / (2 * tau ** 2))

        # Form the weighted design matrix
        W = np.diag(weights)

        # Fit the linear model
        X_ = np.vstack([np.ones(m), X]).T
        theta = np.linalg.inv(X_.T @ W @ X_) @ X_.T @ W @ y

        # Predict the value at x_query
        y_pred[i] = np.array([1, x_q]) @ theta

    return y_pred

# Parameters
tau = 0.5 # Bandwidth parameter
# Predict values
x_query = np.linspace(0, 10, 100)
y_pred = locally_weighted_regression(X, y, x_query, tau)

# Plot results
plt.scatter(X, y, label='Data points')
plt.plot(x_query, y_pred, color='red', label='LWR fit')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Locally Weighted Regression')
plt.legend()
plt.show()
```

Program 10

python

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

iris = datasets.load_iris()
X = iris.data
y = iris.target
print("Features:\n", iris.feature_names)
print("Classes:\n", iris.target_names)

# Convert to DataFrame for better visualization
df = pd.DataFrame(data=np.c_[iris['data'], iris['target']],
columns=iris['feature_names'] + ['target'])
print("\nFirst 5 rows of the dataset:\n", df.head())

# Plotting (optional)
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', edgecolor='k', s=50)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Iris Data Sepal Length vs Sepal Width')
plt.show()

# Make predictions
y_pred = svm_classifier.predict(X_test)
# Evaluate the model
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nAccuracy Score:\n", accuracy_score(y_test, y_pred))

df.info()
df['target'].value_counts()

svm_classifier_rbf = SVC(random_state = 42)
svm_classifier_rbf.fit(X_train, y_train)
```

```
# Make predictions
y_pred = svm_classifier_rbf.predict(X_test)
# Evaluate the model
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nAccuracy Score:\n", accuracy_score(y_test, y_pred))

svm_classifier_poly = SVC(kernel = 'poly', random_state = 42)
svm_classifier_poly.fit(X_train, y_train)

# Make predictions
y_pred = svm_classifier_poly.predict(X_test)
# Evaluate the model
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nAccuracy Score:\n", accuracy_score(y_test, y_pred))
```