TW1

```
import csv
with open('Training_examples.csv','r') as f:
    r = csv.reader(f)
    l = list(r)
print(l)
h = ['%','%','%','%','%','%']
for i in l:
    if i[-1] == 'Yes':
        j = 0
        for x in i:
            if x != 'Yes':
                if x != h[j] and h[j] == '%':
                    h[j] = x
                elif x != h[j] and h[j] != '%':
                    h[j] = '?'
            j = j + 1
print(h)
```

Dataset

['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No'],
['Sunny', 'Warm', 'High', 'Strong', 'Cold', 'Change', 'Yes']

TW2

```
import numpy as np
import pandas as pd

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

concepts = np.array(data.iloc[:,0:-1])

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


def learn(concepts, target):

    print("initialization of specific_h and general_h")

    specific_h = concepts[0].copy()
    print(specific_h)

    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print(general_h)

    # The learning iterations
    for i, h in enumerate(concepts):

        # Checking if the hypothesis has a positive target
        if target[i] == "Yes":
            for x in range(len(specific_h)):

                # Change values in S & G only if values change
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'

        # Checking if the hypothesis has a positive target
        if target[i] == "No":
            for x in range(len(specific_h)):

                # For negative hyposthesis change values only  in G
                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)
    # find indices where we have empty rows, meaning those that are unchanged
    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
        # remove those rows from general_h
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    # Return final values
    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")
```

TW3

```
import math
import csv

def load_csv(filename):
    lines = csv.reader(open(filename,'r'))
    dataset = list(lines)

    header = dataset.pop(0)
    return dataset, header

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

def subtables(data, col, delete):
    dic = {}
    coldata = [row[col] for row in data]
    attr = list(set(coldata))
    for k in attr:
        dic[k] = []
    for y in range(len(data)):
        key = data[y][col]
        if delete:
            del data[y][col]
        dic[key].append(data[y])
    return attr, dic

def entropy(s):
    attr = list(set(s))

    if len(attr) == 1:
        return 0
    count = [0,0]
    for i in range(2):
        count[i] = sum([1 for x in s if attr[i] == x])/(len(s) * 1.0)
    sums = 0
    for cnt in count:
        sums += -1 * cnt * math.log(cnt,2)
    return sums

def compute_gain(data, col):
    attValues, dic = subtables(data, col, delete = False)
    total_entropy = entropy([row[-1] for row in data])

    for x in range(len(attValues)):
        ratio = len(dic[attValues[x]])/(len(data) * 1.0)
        entro = entropy([row[-1] for row in dic[attValues[x]]])
        total_entropy -= ratio * entro
    return total_entropy
```

```
def build_tree(data,attributes):
    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=[compute_gain(data,col)for col in range(n)]
    split=gains.index(max(gains))
    node=Node(attributes[split])
    fea=attributes[:split]+attributes[split+1:]
    attr,dic=subtables(data,split,delete=True)
    print('attr value is',attr)

    for x in range(len(attr)):
        child=build_tree(dic[attr[x]],fea)
        node.childern.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.childern:
        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.childern:
        if x_test[pos]==value:
            classify(n,x_test,features)

dataset,features=load_csv("tennis2.csv")
node=build_tree(dataset,features)

print("the decision tree for the dataset using ID3 algorithm is")
print_tree(node,0)

testdata,features=load_csv("tennis2.csv")
for xtest in testdata:
    print("the test instance:",xtest)
    print("the predicted label:")
    classify(node,xtest,features)
```

TW4

```
import numpy as np

X = np.array(([0,0],[0,1],[1,0],[1,1]),dtype = float)
y = np.array(([0],[0],[0],[1]), dtype = float)

def sigmoid(x):
   return 1/(1 + np.exp(-x))
def derivatives_sigmoid(x):
   return x * (1-x)

epoch = 700
lr = 0.9
inputlayer_neuron = 2
hiddenlayer_neuron = 3
output_neuron = 1

wh = np.random.uniform(size = (inputlayer_neuron, hiddenlayer_neuron))
bh = np.random.uniform(size = (1, hiddenlayer_neuron))
wout = np.random.uniform(size=(hiddenlayer_neuron, output_neuron))
bout = np.random.uniform(size = (1, output_neuron))

for i in range(epoch):
   hinp1 = np.dot(X, wh)
   hinp = hinp1 + bh
   hlayer_act = sigmoid(hinp)

   outinp1 = np.dot(hlayer_act, wout)
   outinp = outinp1 + bout
   output_act = sigmoid(outinp)

   EO = y - output_act
   outgrad = derivatives_sigmoid(output_act)
   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("Actual Output: \n", str(y))
print("Predicted Output: \n", output)
```

TW5

```
import csv
import random
import math
import operator

def loadCsv(filename):
	lines = csv.reader(open(filename))
	dataset = list(lines)
	for i in range(len(dataset)):
		dataset[i] = [float(x) for x in dataset[i]]
	return dataset

def splitDataset(dataset, splitRatio):
	trainSize = int(len(dataset) * splitRatio)
	trainSet = []
	copy = list(dataset)
	i=0
	while len(trainSet) < trainSize:
		#index = random.randrange(len(copy))

		trainSet.append(copy.pop(i))
	return [trainSet, copy]

def separateByClass(dataset):
	separated = {}
	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([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
	return math.sqrt(variance)

def summarize(dataset):
	summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
	del summaries[-1]
	return summaries

def summarizeByClass(dataset):
	separated = separateByClass(dataset)
	summaries = {}
	for classValue, instances in separated.items():
```

```
                summaries[classValue] = summarize(instances)
        return summaries

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

def calculateClassProbabilities(summaries, inputVector):
        probabilities = {}
        for classValue, classSummaries in summaries.items():
                probabilities[classValue] = 1
                for i in range(len(classSummaries)):
                        mean, stdev = classSummaries[i]
                        x = inputVector[i]
                        probabilities[classValue] *= calculateProbability(x, mean, stdev)
        return probabilities

def predict(summaries, inputVector):
        probabilities = calculateClassProbabilities(summaries, inputVector)
        bestLabel, bestProb = None, -1
        for classValue, probability in probabilities.items():
                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
        accuracy = correct / float(len(testSet)) * 100.0
        return accuracy

def main():
        filename = 'ConceptLearning.csv'
        splitRatio = 0.75
        dataset = loadCsv(filename)
        trainingSet, testSet = splitDataset(dataset, splitRatio)
        print('Split {0} rows into'.format(len(dataset)))
        print('Number of Training data: ' + (repr(len(trainingSet))))
        print('Number of Test Data: ' + (repr(len(testSet))))
        print("\nThe values assumed for the concept learning attributes are\n")
```

```
        print("OUTLOOK=> Sunny=1 Overcast=2 Rain=3\nTEMPERATURE=> Hot=1 Mild=2 Cool=3\n
HUMIDITY=> High=1 Normal=2\nWIND=> Weak=1 Strong=2")
        print("TARGET CONCEPT:PLAY TENNIS=> Yes=10 No=5")
        print("\nThe Training set are:")
        for x in trainingSet:
                print(x)
        print("\nThe Test data set are:")
        for x in testSet:
                print(x)
        print("\n")
        # prepare model
        summaries = summarizeByClass(trainingSet)
        # test model
        predictions = getPredictions(summaries, testSet)
        actual = []
        for i in range(len(testSet)):
                vector = testSet[i]
                actual.append(vector[-1])
        # Since there are five attribute values, each attribute constitutes to 20% accuracy. So if all a
ttributes match with predictions then 100% accuracy
        print('Actual values: {0}%'.format(actual))
        print('Predictions: {0}%'.format(predictions))
        accuracy = getAccuracy(testSet, predictions)
        print('Accuracy: {0}%'.format(accuracy))

main()
```

Dataset

1	1	1	1	5
1	1	1	2	5
2	1	1	2	10
3	2	1	1	10
3	3	2	1	10
3	3	2	2	5
2	3	2	2	10
1	2	1	1	5
1	3	2	1	10
3	2	2	2	10
1	2	2	2	10
2	2	1	2	10
2	1	2	1	10
3	2	1	2	5
1	2	1	2	10
1	2	1	2	5

TW6

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
import numpy as np
categories = ['alt.atheism', 'soc.religion.christian','comp.graphics', 'sci.med']
twenty_train = fetch_20newsgroups(subset='train',categories=categories,shuffle=True)
twenty_test = fetch_20newsgroups(subset='test',categories=categories,shuffle=True)

print(len(twenty_train.data))
print(len(twenty_test.data))
print(twenty_train.target_names)
print("\n".join(twenty_train.data[0].split("\n")))
print(twenty_train.target[0])

count_vect = CountVectorizer()
X_train_tf = count_vect.fit_transform(twenty_train.data)

tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_tf)
X_train_tfidf.shape

mod = MultinomialNB()
mod.fit(X_train_tfidf, twenty_train.target)
X_test_tf = count_vect.transform(twenty_test.data)
X_test_tfidf = tfidf_transformer.transform(X_test_tf)
predicted = mod.predict(X_test_tfidf)
print("Accuracy:", accuracy_score(twenty_test.target, predicted))
print(classification_report(twenty_test.target,predicted,target_names=twenty_test.target_names))
print("confusion matrix is \n",metrics.confusion_matrix(twenty_test.target, predicted))
```

TW7

```
import bayespy as bp
import numpy as np
import csv
from colorama import init, Fore, Back, Style
init()

ageEnum = {'SuperSeniorCitizen':0, 'SeniorCitizen':1, 'MiddleAged':2, 'Youth':3, 'Teen':4}
genderEnum = {'Male':0, 'Female':1}
familyHistoryEnum = {'Yes':0, 'No':1}
dietEnum = {'High':0, 'Medium':1, 'Low':2}
lifeStyleEnum = {'Athlete':0, 'Active':1, 'Moderate':2, 'Sedetary':3}
cholesterolEnum = {'High':0, 'BorderLine':1, 'Normal':2}
heartDiseaseEnum = {'Yes':0, 'No':1}

with open('heart_disease_data.csv') as csvfile:
    lines = csv.reader(csvfile)
    dataset = list(lines)
    data = []
    for x in dataset:

data.append([ageEnum[x[0]],genderEnum[x[1]],familyHistoryEnum[x[2]],dietEnum[x[3]],lifeStyleEnu
m[x[4]],cholesterolEnum[x[5]],heartDiseaseEnum[x[6]]])

data = np.array(data)
N = len(data)

p_age = bp.nodes.Dirichlet(1.0*np.ones(5))
age = bp.nodes.Categorical(p_age, plates=(N,))
age.observe(data[:,0])

p_gender = bp.nodes.Dirichlet(1.0*np.ones(2))
gender = bp.nodes.Categorical(p_gender, plates=(N,))
```

```
gender.observe(data[:,1])

p_familyhistory = bp.nodes.Dirichlet(1.0*np.ones(2))
familyhistory = bp.nodes.Categorical(p_familyhistory, plates=(N,))
familyhistory.observe(data[:,2])

p_diet = bp.nodes.Dirichlet(1.0*np.ones(3))
diet = bp.nodes.Categorical(p_diet, plates=(N,))
diet.observe(data[:,3])

p_lifestyle = bp.nodes.Dirichlet(1.0*np.ones(4))
lifestyle = bp.nodes.Categorical(p_lifestyle, plates=(N,))
lifestyle.observe(data[:,4])

p_cholesterol = bp.nodes.Dirichlet(1.0*np.ones(3))
cholesterol = bp.nodes.Categorical(p_cholesterol, plates=(N,))
cholesterol.observe(data[:,5])

p_heartdisease = bp.nodes.Dirichlet(np.ones(2), plates=(5, 2, 2, 3, 4, 3))
heartdisease = bp.nodes.MultiMixture([age, gender, familyhistory, diet, lifestyle, cholesterol], bp.nodes.Categorical, p_heartdisease)
heartdisease.observe(data[:,6])
p_heartdisease.update()

m = 0
while m == 0:
   print("\n")
   res = bp.nodes.MultiMixture([int(input('Enter Age: ' + str(ageEnum))), int(input('Enter Gender: ' + str(genderEnum))), int(input('Enter FamilyHistory: ' + str(familyHistoryEnum))), int(input('Enter dietEnum: ' + str(dietEnum))), int(input('Enter LifeStyle: ' + str(lifeStyleEnum))), int(input('Enter Cholesterol: ' + str(cholesterolEnum)))], bp.nodes.Categorical, p_heartdisease).get_moments()[0][heartDiseaseEnum['Yes']]
   print("Probability(HeartDisease) = " +  str(res))
   m = int(input("Enter for Continue:0, Exit :1  "))
```

TW8

```
from sklearn import datasets, metrics
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
import matplotlib.pyplot as plt
import pandas as pd

dataset = datasets.load_iris()
X = pd.DataFrame(dataset.data)
y = pd.DataFrame(dataset.target)
model = KMeans(n_clusters=3)
model.fit(X)
plt.figure()
plt.subplot(3, 2, 1)
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y.iloc[:, 0], s=10, cmap='viridis')
plt.title("Actual Classification - Sepal")
plt.subplot(3, 2, 2)
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=model.labels_, s=10, cmap='viridis')
plt.title("K-Means Classification - Sepal")

plt.subplot(3, 2, 3)
plt.scatter(X.iloc[:, 2], X.iloc[:, 3], c=y.iloc[:, 0], s=10, cmap='viridis')
plt.title("Actual Classification - Petal")

plt.subplot(3, 2, 4)
plt.scatter(X.iloc[:, 2], X.iloc[:, 3], c=model.labels_, s=10, cmap='viridis')
plt.title("K-Means Classification - Petal")

print('K-Means Accuracy : ', metrics.accuracy_score(y, model.labels_))

# EM Algorithm
gmm = GaussianMixture(n_components=3)
```

```
gmm.fit(X)
y_predict = gmm.predict(X)
plt.subplot(3, 2, 5)
plt.scatter(X.iloc[:, 2], X.iloc[:, 3], c=y.iloc[:, 0], s=10, cmap='viridis')
plt.title("Actual Classification - Petal")
plt.subplot(3, 2, 6)
plt.scatter(X.iloc[:, 2], X.iloc[:, 3], c=y_predict, s=10, cmap='viridis')
plt.title("GMM Classification - Petal")
print('GMM Accuracy : ', metrics.accuracy_score(y, y_predict))
print("Confusion Matrix : \n", metrics.confusion_matrix(y, y_predict))
plt.show()
```

TW9

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn import datasets

iris = datasets.load_iris()
iris_data = iris.data
iris_labels = iris.target

x_train, x_test, y_train, y_test = train_test_split(iris_data, iris_labels)
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(x_train,y_train)
y_pred = classifier.predict(x_test)

correct = 0
for i in range(len(y_test)):
    if y_pred[i] == y_test[i]:
        correct += 1
print('Accuracy: ', (correct/float(len(y_test))) * 100, '%')

print('Confusion Matrix is: \n', confusion_matrix(y_test, y_pred))
print('Accuracy Metrics: \n', classification_report(y_test, y_pred))
```

TW10

```
from math import ceil
import numpy as np
from scipy import linalg
def lowess(x, y, f=2./3., iter=3):
    n = len(x)
    r = int(ceil(f*n))
    h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]
    w = np.clip(np.abs((x[:,None] - x[None,:]) / h), 0.0, 1.0)
    w = (1 - w**3)**3
    yest = np.zeros(n)
    delta = np.ones(n)
    for iteration in range(iter):
        for i in range(n):
            weights = delta * w[:,i]
            b = np.array([np.sum(weights*y), np.sum(weights*y*x)])
            A = np.array([[np.sum(weights), np.sum(weights*x)],
                 [np.sum(weights*x), np.sum(weights*x*x)]])
            beta = linalg.solve(A, b)
            yest[i] = beta[0] + beta[1]*x[i]
        residuals = y - yest
        s = np.median(np.abs(residuals))
        delta = np.clip(residuals / (6.0 * s), -1, 1)
        delta = (1 - delta**2)**2
    return yest
if __name__ == '__main__':
    import math
    n = 100
    x = np.linspace(0, 2 * math.pi, n)
    print("==========================values of x====================")
    print(x)
    y = np.sin(x) + 0.3*np.random.randn(n)
```

```
print("================================Values of y==================")
print(y)
f = 0.25
yest = lowess(x, y, f=f, iter=3)

import pylab as pl
pl.clf()
pl.plot(x, y, label='y noisy')
pl.plot(x, yest, label='y pred')
pl.legend()
pl.show()
```