1) **Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file.**

```
import csv
hypo = ['%','%','%','%','%','%']
with open('finds.csv') as csv_file:
   readcsv = csv.reader(csv_file,delimiter = ',')
   data = []
   print("\nThe given training examples are:")
   for row in readcsv:
      print(row)
      if row[len(row)-1].upper()=="YES":
         data.append(row)
   print("\nThe positive exampes are:")
   for x in data:
      print(x)
      print()
TotalExamples = len(data);
i = j = k = 0
print("\nThe steps of Find-S algorithm are:")
print(hypo)
list = []
p = 0
d = len(data[p])-1
print(d)
```

```
for j in range(d):
    list.append(data[i][j])
    hypo = list
i = 1
for i in range(TotalExamples):
    for k in range(d):
        if hypo[k]!=data[i][k]:
            hypo[k] = '?'
            k = k+1
        else:
            hypo[k]
    print()
    print(hypo)
print("\nThe maximally specific Find-S hypothesis:")
list = []
for i in range(d):
    list.append(hypo[i])
print(list)
```

2) **For a given set of training data examples stored in a .CSV file, implement and demonstrate the CandidateElimination algorithm to output a description of the set of all hypotheses consistent with the training examples.**

```
import numpy as np

import pandas as pd
# Loading Data from a CSV File
data = pd.DataFrame(data=pd.read_csv('candidate.csv'))
concepts = np.array(data.iloc[:,0:-1])
```

```
target = np.array(data.iloc[:,-1])
def learn(concepts, target):
    specific_h = concepts[0].copy()
    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    for i, h in enumerate(concepts):
        if target[i] == "Yes":
            for x in range(len(specific_h)):
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'
        if target[i] == "No":
            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] = '?'
            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 S:", s_final, sep="\n")
print("Final G:", g_final, sep="\n")
data.head()
```

3) Develop a program to demonstrate the prediction of values of a given dataset using Linear regression

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

def estimate_coef(x, y):
    n=np.size(x)
    m_x, m_y = np.mean(x), np.mean(y)
    SS_xy = np.sum(y*x - n*m_y*m_x)
    SS_xx = np.sum(x*x - n*m_x*m_x)
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x
    return(b_0, b_1)
```

```
def plot_regression_line(x, y, b):
    plt.scatter(x, y, color = "m", marker = "o", s=30)
    y_pred = b[0] + b[1]*x
    plt.plot(x, y_pred, color = "g")
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()

x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])
b = estimate_coef(x, y)
print("Estimated coefficients : \nb_0 = {}\nb_1 = {}".format(b[0],b[1]))
plot_regression_line(x, y, b)
```

4) Develop a program to demonstrate the prediction of values of a given dataset using Multiple linear regression

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

def loadCSV(filename):
    with open(filename,"r") as csvfile:
        lines = csv.reader(csvfile)
        dataset = list(lines)
        for i in range(len(dataset)):
            dataset[i] = [float(x) for x in dataset[i]]
    return np.array(dataset)

def normalize(X):
    mins = np.min(X,axis=0)
    maxs = np.max(X,axis=0)
    rng = maxs - mins
    norm_X = 1 - ((maxs - X)/rng)
    return norm_X

def logistic_func(beta, X):
    return 1.0/(1 + np.exp(-np.dot(X, beta.T)))
```

```
def log_gradient(beta, X, y):
    first_calc = logistic_func(beta, X) - y.reshape(X.shape[0], -1)
    final_calc = np.dot(first_calc.T, X)
    return final_calc

def cost_func(beta, X, y):
    log_func_v = logistic_func(beta, X)
    y = np.squeeze(y)
    step1 = y * np.log(log_func_v)
    step2 = (1 - y) * np.log(1 - log_func_v)
    final = -step1 - step2
    return np.mean(final)

def grad_desc(X, y, beta, lr=.01, converge_change=.001):
    cost = cost_func(beta, X, y)
    change_cost = 1
    num_iter = 1
    while(change_cost > converge_change):
        old_cost = cost
        beta = beta - (lr * log_gradient(beta, X, y))
        cost = cost_func(beta, X, y)
        change_cost = old_cost - cost
        num_iter += 1
    return beta, num_iter

def pred_values(beta, X):
    pred_prob = logistic_func(beta, X)
    pred_value = np.where(pred_prob >= .5, 1, 0)
    return np.squeeze(pred_value)

def plot_reg(X, y, beta):
    x_0 = X[np.where(y == 0.0)]
    x_1 = X[np.where(y == 1.0)]
    plt.scatter([x_0[:, 1]], [x_0[:, 2]], c='b', label='y = 0')
    plt.scatter([x_1[:, 1]], [x_1[:, 2]], c='r', label='y = 1')
    x1 = np.arange(0,1,0.1)
    x2 = -(beta[0,0] + beta[0,1]*x1)/beta[0,2]
    plt.plot(x1, x2, c='k', label='reg line')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
    plt.show()


dataset = loadCSV('logistic_data.csv')
X = normalize(dataset[:,:-1])
X = np.hstack((np.matrix(np.ones(X.shape[0])).T,X))
y = dataset[:,-1]
beta = np.matrix(np.zeros(X.shape[1]))
beta,num_iter = grad_desc(X,y,beta)
print("Estimated regression coefficients:",beta)
print("No. of iterations:",num_iter)
y_pred = pred_values(beta,X)
print("Correctly predicted labels:", np.sum(y == y_pred))
plot_reg(X, y, beta)
```

5) Develop a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

Id3.py

```
import numpy as np
import math
from data_loader import read_data
class Node:
    def __init__(self, attribute):
        self.attribute = attribute
        self.children = []
        self.answer = ""
    def __str__(self):
        return self.attribute
def subtables(data, col, delete):
    dict = {}
    items = np.unique(data[:, col])
    count = np.zeros((items.shape[0], 1), dtype=np.int32)
    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                count[x] += 1
    for x in range(items.shape[0]):
        dict[items[x]] = np.empty((int(count[x]), data.shape[1]), dtype="|S32")
        pos = 0
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                dict[items[x]][pos] = data[y]
                pos += 1
        if delete:
            dict[items[x]] = np.delete(dict[items[x]], col, 1)
    return items, dict
def entropy(S):
    items = np.unique(S)
    if items.size == 1:
        return 0
    counts = np.zeros((items.shape[0], 1))
    sums = 0
    for x in range(items.shape[0]):
        counts[x] = sum(S == items[x]) / (S.size * 1.0)
    for count in counts:
        sums += -1 * count * math.log(count, 2)
    return sums

def gain_ratio(data, col):
    items, dict = subtables(data, col, delete=False)
    total_size = data.shape[0]
    entropies = np.zeros((items.shape[0], 1))
    intrinsic = np.zeros((items.shape[0], 1))
    for x in range(items.shape[0]):
        ratio = dict[items[x]].shape[0]/(total_size * 1.0)
        entropies[x] = ratio * entropy(dict[items[x]][:, -1])
        intrinsic[x] = ratio * math.log(ratio, 2)
    total_entropy = entropy(data[:, -1])
    iv = -1 * sum(intrinsic)
    for x in range(entropies.shape[0]):
        total_entropy -= entropies[x]
    return total_entropy / iv
def create_node(data, metadata):
    if (np.unique(data[:, -1])).shape[0] == 1:
        node = Node("")
        node.answer = np.unique(data[:, -1])[0]
        return node
    gains = np.zeros((data.shape[1] - 1, 1))
    for col in range(data.shape[1] - 1):
        gains[col] = gain_ratio(data, col)
    split = np.argmax(gains)
    node = Node(metadata[split])
    metadata = np.delete(metadata, split, 0)
    items, dict = subtables(data, split, delete=True)
    for x in range(items.shape[0]):
        child = create_node(dict[items[x]], metadata)
        node.children.append((items[x], child))
    return node
def empty(size):
    s = ""
    for x in range(size):
        s += "   "
    return s
def print_tree(node, level):
    if node.answer != "":
        print(empty(level), node.answer)
        return
    print(empty(level), node.attribute)
    for value, n in node.children:
        print(empty(level + 1), value)
        print_tree(n, level + 2)
metadata, traindata = read_data("tennis.csv")
data = np.array(traindata)
node = create_node(data, metadata)
print_tree(node, 0)
```

data_loader.py

```
import csv
```

```
def read_data(filename):

    with open(filename, 'r') as csvfile:
        datareader = csv.reader(csvfile, delimiter=',')
        headers = next(datareader)
        metadata = []
        traindata = []
        for name in headers:
            metadata.append(name)

        for row in datareader:
            traindata.append(row)
    return (metadata, traindata)
```

6) **Develop a program to implement the naïve Bayesian Classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.**

```
print("\nNaive Bayes Classifier for concept learning problem")
import csv
import math
def safe_div(x,y):
    if y == 0:
        return 0
    return x / y

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 safe_div(sum(numbers),float(len(numbers)))

def stdev(numbers):
    avg = mean(numbers)
    variance = safe_div(sum([pow(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(-safe_div(math.pow(x-mean,2),(2*math.pow(stdev,2))))
    final = safe_div(1 , (math.sqrt(2*math.pi) * stdev)) * exponent
    return final
```

```
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 = safe_div(correct,float(len(testSet))) * 100.0
    return accuracy

def main():
    filename = 'tennis_naive.csv'
    splitRatio = 0.9
    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\nHUMIDITY=> 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 attributes
#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()
```

7) **Assuming a set of documents that need to be classified, use the naïve Bayesian Classifier model to perform this task. Calculate the accuracy, precision, and recall for your data set.**

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
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)
print("DOCUMENT-TERM-MATRIX",X_train_tf)

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))
```

8) Develop a program to construct Support Vector Machine considering a Sample Dataset.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.style as style
from sklearn import svm

style.use("ggplot")

X = np.array([[1,2],[5,8],[1.5,1.8],[8,8],[1,0.6],[9,11],[10,10],[3,2]])
y = [0,1,0,1,0,1,1,0]

clf = svm.SVC(kernel = 'linear')
clf.fit(X,y)
print("Prediction is:",clf.predict([[7,4.0]]))

w = clf.coef_[0]
print(w)

a = -w[0]/w[1]
xx = np.linspace(0,12)
yy = a * xx - clf.intercept_[0]/w[1]

h0 = plt.plot(xx,yy,'k-',label = "non weighted div")

plt.scatter(X[:,0],X[:,1],c = y)
plt.legend()
plt.show()
```

9) Implement K Means algorithm

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans

X = np.array([[0,-3],
    [-10,15],
    [0,-12],
    [24,-10],
    [30,45],
    [85,70],
    [7,80],
    [60,0],
    [0,52],
    [80,91],])

plt.scatter(X[:,0],X[:,1], label='True Position')

kmeans = KMeans(n_clusters=2)
kmeans.fit(X)

print(kmeans.cluster_centers_)

print(kmeans.labels_)

plt.scatter(X[:,0], X[:,1], c=kmeans.labels_, cmap='rainbow')
plt.scatter(kmeans.cluster_centers_[:,0] ,kmeans.cluster_centers_[:,1], color='black')
plt.show()
```

10) **Develop a program to implement K-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions**

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

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

dataset = pd.read_csv(url, names=names)

dataset.head()

X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 4].values

from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.80)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=7)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)

from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

11) Implement Random Forest algorithm

```
import pandas as pd

from sklearn import datasets

iris = datasets.load_iris()
print(iris.target_names)

print(iris.feature_names)
print(iris.data[0:5])

print(iris.target)

data=pd.DataFrame({
  'sepal length':iris.data[:,0],
  'sepal width':iris.data[:,1],
  'petal length':iris.data[:,2],
  'petal width':iris.data[:,3],
  'species':iris.target
})
data.head()

from sklearn.model_selection import train_test_split

X=data[['sepal length', 'sepal width', 'petal length', 'petal width']]
y=data['species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50)

from sklearn.ensemble import RandomForestClassifier

clf=RandomForestClassifier(n_estimators=50)

clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)

from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

12) Write a program to construct Recommendation System for Music data.

Popularity.py

```
import pandas
from sklearn.model_selection import train_test_split
import numpy as np
import time
import Recommendor as Rec

#Read user_id, song_id, listen_count
triplets = 'https://static.turi.com/datasets/millionsong/10000.txt'
songs_metadata = 'https://static.turi.com/datasets/millionsong/song_data.csv'

song_df_a = pandas.read_table(triplets,header=None)
```

```
song_df_a.columns = ['user_id', 'song_id', 'listen_count']

#Read song  metadata [song_id,title, release,artist_name, year]

song_df_b =  pandas.read_csv(songs_metadata)

#Merge the two dataframes above to create input dataframe for recommender systems
song_df1 = pandas.merge(song_df_a, song_df_b.drop_duplicates(['song_id']),
on="song_id", how="left")
song_df1.head()
print("Total no of songs:",len(song_df1))

song_df1 = song_df1.head(10000)

#Merge song title and artist_name columns to make a new column
song_df1['song'] = song_df1['title'].map(str) + " - " + song_df1['artist_name']

song_gr = song_df1.groupby(['song']).agg({'listen_count': 'count'}).reset_index()
grouped_sum = song_gr['listen_count'].sum()
song_gr['percentage']  = song_gr['listen_count'].div(grouped_sum)*100
song_gr.sort_values(['listen_count', 'song'], ascending = [0,1])
print(song_gr)

u = song_df1['user_id'].unique()
print("The no. of unique users:", len(u))

#['user_id', 'song_id', 'listen_count',title,release,artist_name,year,song]

train, test_data = train_test_split(song_df1, test_size = 0.20, random_state=0)
print("*****Training data*****")
print(train.head(5))

pm = Rec.popularity_recommender()                        #create an instance of the class
pm.create_p(train, 'user_id', 'song')
print("******starting the recommendation****")
user_id1 = u[8]                                          #Recommended songs list for a user
print(pm.recommend_p(user_id1))
```

```
#print("**** starting the recommendation2****")
#user_id2 = u[8]
#print(pm.recommend_p(user_id2))
```

Recommendor.py

```
import numpy as np
import pandas
class popularity_recommender():
    def __init__(self):
        self.t_data = None
        self.u_id = None           #ID of the user
        self.i_id = None           #ID of Song the user is listening to
        self.pop_recommendations = None #getting popularity recommendations

    #Create the system model
    def create_p(self, t_data, u_id, i_id):
        self.t_data = t_data
        self.u_id = u_id
        self.i_id = i_id
        #Get the no. of times each song has been listened as recommendation score
        #Get a count of user_ids for each unique song as recommendation score
        t_data_grouped = t_data.groupby([self.i_id]).agg({self.u_id: 'count'}).reset_index()
        t_data_grouped.rename(columns = {'user_id': 'score'},inplace=True)

        #Sort the songs based upon recommendation score
        t_data_sort = t_data_grouped.sort_values(['score', self.i_id], ascending = [0,1])

        #Generate a recommendation rank based upon score
        t_data_sort['Rank'] = t_data_sort['score'].rank(ascending=0, method='first')

        #Get the top 10 recommendations
        self.pop_recommendations = t_data_sort.head(10)

    #Use the system model to give recommendations
    def recommend_p(self, u_id):
        u_recommendations = self.pop_recommendations
        #Add user_id column for which the recommended songs are generated
        u_recommendations['user_id'] = u_id
        #Bring user_id column to the front
        cols = u_recommendations.columns.tolist()
        #cols = cols[-1:] + cols[:-1]
        cols = cols[:-1]
        u_recommendations = u_recommendations[cols]

        return u_recommendations
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
dataset=pd.read_csv('tennis.csv')
Le = LabelEncoder()

dataset['outlook'] = Le.fit_transform(dataset['outlook'])
dataset['temperature'] = Le.fit_transform(dataset['temperature'])
dataset['humidity'] = Le.fit_transform(dataset['humidity'])
dataset['wind'] = Le.fit_transform(dataset['wind'])

print(dataset.head())
x=dataset.iloc[:,0:-1].values
y=dataset.iloc[:,-1].values
x_train, x_test,y_train,y_test=train_test_split(x,y,test_size=0.30)
dt=DecisionTreeClassifier()
dt.fit(x_train,y_train)
y_pred=dt.predict(x_test)
from sklearn import tree
fig = plt.figure(figsize=(20,20))
viz = tree.plot_tree(dt)
```