**1 FIND – S**

**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

with open('find-s-training-examples.csv') as csvfile:

data = [line[:-1] for line in csv.reader(csvfile) if line[-1] == 'Y']

print('Positive training examples are: {}'.format(data))

S = ['$'] \* len(data[0])

print('\nOutput at each step is \n{}'.format(S))

for example in data:

i = 0

for feature in example:

**S[i] = feature if S[i] == '$' or S[i] == feature else '?'**

i += 1

print(S)

**2 CANDIDATE ELIMINATION**

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

**def consistent(h1, h2)**

**def candidateElimination()**

import csv

with open('candidate-elimination-training-examples.csv') as file:

data = [**tuple(line)** for line in csv.reader(file)]

D = []

for i in range(len(data[0])):

**D.append(list(set([ele[i] for ele in data])))**

def consistent(h1, h2):

for x, y in zip(h1, h2):

**if not (x == '?' or (x != '$' and (x == y or y == '$'))):**

return False

return True

def candidateElimination():

G = {('?',) \* (len(data[0]) - 1),}

S = ['$'] \* (len(data[0]) - 1)

num = 0

print('G[{0}]:'.format(num), G)

print('S[{0}]:'.format(num), S)

for item in data:

num += 1

inp, res = item[:-1], item[-1]

if res in 'Yy':

G = {g for g in G if consistent(g, inp)}

**i = 0**

for s, x in zip(S, inp):

if s != x:

S[i] = '?' if s != '$' else x

**i += 1**

else:

S = S

Gprev = G.copy()

for g in Gprev:

#if g not in G:

#continue

**for i in range(len(g)):**

**if g[i] == '?':**

**for val in D[i]:**

**if val != inp[i] and S[i] == val:**

g\_new = g[:i] + (val, ) + g[i + 1:]

G.add(g\_new)

else:

G.add(g)

**G.difference\_update([h for h in G if any([consistent(h, g1) for g1 in G if g1 != h])])**

print('G[{0}]:'.format(num), G)

print('S[{0}]:'.format(num), S)

candidateElimination()

**3 ID3**

**Write 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.**

**def entropyOfList(aList):**

**def informationGain(tennis, splitAttributeName, targetAttributeName):**

**def id3(tennis, targetAttributeName, attributeNames, defaultClass = None):**

**def classify(instance, tree, default = None):**

import math

from collections import Counter

from pprint import pprint

from pandas import DataFrame

tennis = DataFrame.from\_csv('id3-training-examples.csv')

print('PlayTennis dataset:', tennis)

def entropyOfList(aList):

cnt = Counter(aList)

probs = [x / len(aList) for x in cnt.values()]

entropy = sum([**-**prob \* math.log(prob, 2) for prob in probs])

return entropy

print('\nEntropy of PlayTennis dataset: %.4f' % entropyOfList(tennis['PlayTennis']))

def informationGain(tennis, splitAttributeName, targetAttributeName):

split = tennis.groupby(splitAttributeName)

**agg\_ent = split.agg({targetAttributeName: [entropyOfList, lambda x: len(x) / len(tennis)]})**

agg\_ent.columns = ['Entropy', 'PropObservations']

newEntropy = sum(agg\_ent['Entropy'] \* agg\_ent['PropObservations'])

oldEntropy = entropyOfList(tennis[targetAttributeName])

return oldEntropy - newEntropy

print('\nInformation gain for Outlook: %.4f' % informationGain(tennis, 'Outlook', 'PlayTennis'))

print('Information gain for Temperature: %.4f' % informationGain(tennis, 'Temperature', 'PlayTennis'))

print('Information gain for Humidity: %.4f' % informationGain(tennis, 'Humidity', 'PlayTennis'))

print('Information gain for Wind: %.4f' % informationGain(tennis, 'Wind', 'PlayTennis'))

def id3(tennis, targetAttributeName, attributeNames, defaultClass = None):

cnt = Counter(tennis[targetAttributeName])

if len(cnt) == 1:

return next(iter(cnt))

elif tennis.empty or (not attributeNames):

return defaultClass

else:

defaultClass = max(cnt.keys())

**gains = [informationGain(tennis, attr, targetAttributeName) for attr in attributeNames]**

**best = attributeNames[gains.index(max(gains))]**

tree = {best: {}}

remainingAttributeNames = [i for i in attributeNames if i != best]

for attr, subset in tennis.groupby(best):

tree[best][attr] = **id3(subset, targetAttributeName, remainingAttributeNames, defaultClass)**

return tree

attributeNames = list(tennis.columns)

print('\nList of attributes:', attributeNames)

attributeNames.remove('PlayTennis')

print('Predicting attributes:', attributeNames)

**tree = id3(tennis, 'PlayTennis', attributeNames)**

print('\nThe resultant decision tree is:')

**pprint**(tree)

attribute = next(iter(tree))

**def classify(instance, tree, default = None):**

**attribute = next(iter(tree))**

**if instance[attribute] in tree[attribute].keys():**

**result = tree[attribute][instance[attribute]]**

**if isinstance(result, dict):**

**return classify(instance, result)**

**else:**

**return result**

**else:**

**return default**

trainData = tennis.iloc[1: -4]

testData = tennis.iloc[-4: ]

trainTree = id3(trainData, 'PlayTennis', attributeNames)

**testData['predicted2'] = testData.apply(classify, axis = 1, args = (trainTree, 'Yes'))**

**print('\nPredicted values for sample data:\n', testData['predicted2'])**

**print('Accuracy: ', sum(testData['predicted2'] == testData['PlayTennis']) / len(testData.index))**

**4 BACK PROPAGATION**

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

**def initializeNetwork(nInputs, nHidden, nOutputs):**

**def activate(weights, inputs):**

**def forwardPropagate(network, row):**

**def backwardPropagateError(network, expected):**

**def updateWeights(network, row, lRate):**

**def trainNetwork(network, dataset, lRate, nIter, nOutputs):**

import math

import random

def initializeNetwork(nInputs, nHidden, nOutputs):

network = []

hiddenLayer = [{'weights': [random.uniform(-.5, .5) for i in range(nInputs + 1)]} for i in range(nHidden)]

network.append(hiddenLayer)

outputLayer = [{'weights': [random.uniform(-.5, .5) for i in range(nInputs + 1)]} for i in range(nOutputs)]

network.append(outputLayer)

print('The initial neural network is')

for i, layer in zip(range(1, len(network) + 1), network):

for j, neuron in zip(range(1, len(layer) + 1), layer):

print('Layer[%d] Node[%d]: ' % (i, j), neuron)

return network

def activate(weights, inputs):

activation = weights[-1]

for i in **range(len(weights) - 1)**:

activation **+=** weights[i] \* inputs[i]

return activation

def forwardPropagate(network, row):

inputs = row

for layer in network:

newInputs = []

for neuron in layer:

activation = activate(neuron['weights'], inputs)

**neuron['output'] = 1 / (1 + math.exp(-activation))**

newInputs.append(neuron['output'])

inputs = newInputs

return inputs

def backwardPropagateError(network, expected):

for i in **range(len(network) - 1, -1, -1):**

layer = network[i]

errors = []

if i != len(network) - 1:

for j in range(len(layer)):

error = 0

**for neuron in network[i + 1]:**

**error += neuron['weights'][j] \* neuron['delta']**

errors.append(error)

else:

for j in range(len(layer)):

**errors.append(expected[j] - layer[j]['output'])**

for j in range(len(layer)):

**neuron = layer[j]**

**neuron['delta'] = errors[j] \* neuron['output'] \* (1 - neuron['output'])**

def updateWeights(network, row, lRate):

for i in **range(len(network))**:

**inputs = row[:-1]**

if i != 0:

inputs = [neuron['output'] for neuron in network[i - 1]]

**for neuron in network[i]:**

**for j in range(len(inputs)):**

**neuron['weights'][j] += inputs[j] \* neuron['delta'] \* lRate**

**neuron['weights'][-1] += neuron['delta'] \* lRate**

def trainNetwork(network, dataset, lRate, nIter, nOutputs):

for iter in range(nIter):

**sumOfErrors = 0**

for row in dataset:

outputs = forwardPropagate(network, row)

expected = [0 for i in range(nOutputs)]

**expected[row[-1]] = 1**

**sumOfErrors += sum([(expected[i] - outputs[i]) \*\* 2 for i in range(len(expected))])**

backwardPropagateError(network, expected)

updateWeights(network, row, lRate)

**print(iter, 'Error = ', sumOfErrors)**

**random.seed()**

dataset = [ [0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]]

#nInputs = len(dataset[0]) - 1

#nOutputs = len(set([row[-1] for row in dataset]))

network = initializeNetwork(2, 2, 2)

trainNetwork(network, dataset, 0.5, 20, 2)

print('The final neural network is')

for i, layer in zip(range(1, len(network) + 1), network):

for j, neuron in zip(range(1, len(layer) + 1), layer):

print('Layer[%d] Node[%d]: ' % (i, j), neuron)

**5 NAÏVE BAYES**

**Write 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.**

**def mean(numbers)**

**def stdev(numbers):**

**variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers) - 1)**

**def summarize(dataset)**

**def calcProb(summary, item)**

import csv

import math

def mean(numbers):

return sum(numbers) / len(numbers)

def stdev(numbers):

avg = mean(numbers)

**variance = sum([pow(x - avg, 2) for x in numbers]) / (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 calcProb(summary, item):

prob = 1

for i in range(len(summary)):

x = item[i]

mean, stdev = summary[i]

**exponent = math.exp(-pow(x - mean, 2) / (2 \* stdev \*\* 2))**

**final = exponent / (math.sqrt(2 \* math.pi) \* stdev)**

prob \*= final

return prob

with open('naive-bayes-training-examples.csv') as file:

data = [line for line in csv.reader(file)]

**for i in range(len(data)):**

**data[i] = [float(x) for x in data[i]]**

split = int(0.90 \* len(data))

train = data[:split]

test = data[split:]

print('\nTotal number of hypotheses:', len(data))

print('Number of hypotheses in training data:', len(train))

print('Number of hypotheses in test data:', len(test))

print("\nThe values assumed for the concept learning attributes are:")

print("OUTLOOK: Sunny = 1, Overcast = 2 and Rain = 3\nTEMPERATURE: Hot = 1, Mild = 2 and Cool = 3\nHUMIDITY: High = 1 and Normal = 2\nWIND: Weak = 1 and Strong = 2")

print("TARGET CONCEPT: PlayTennis where Yes = 10 and No = 5")

print("\nTraining dataset:")

for x in train:

print(x)

print("\nTest dataset:")

for x in test:

print(x)

yes = []

no = []

for i in range(len(train)):

if data[i][-1] == 5.0:

no.append(data[i])

else:

yes.append(data[i])

yes = summarize(yes)

no = summarize(no)

predictions = []

for item in test:

yesProb = calcProb(yes, item)

noProb = calcProb(no, item)

predictions.append(10.0 if(yesProb > noProb) else 5.0)

correct = 0

for i in range(len(test)):

if(test[i][-1] == predictions[i]):

correct += 1

print("\nActual values are:")

for i in range(len(test)):

print(test[i][-1], end=" ")

print("\nPredicted values are:")

for i in range(len(predictions)):

print(predictions[i], end=" ")

print("\nAccuracy is %.1f%%" % ((correct / len(test)) \* 100))

**6 NAÏVE BAYES TEXT CLASSIFIER**

**Assuming a set of documents that need to be classified, use the naïve Bayesian Classifier model to perform this task. Built-in Java classes/API can be used to write the program. Calculate the accuracy, precision, and recall for your data set.**

from sklearn.datasets import fetch\_20newsgroups

from sklearn.metrics import **confusion\_matrix, accuracy\_score, classification\_report**

from sklearn.feature\_extraction.text import CountVectorizer, TfidfTransformer

from sklearn.naive\_bayes import MultinomialNB

train = fetch\_20newsgroups(subset = 'train', shuffle = True)

**print('The categories of 20NewsGroups are:')**

for cat in train.target\_names:

print(cat)

**categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']**

train = fetch\_20newsgroups(subset = 'train', categories = categories, shuffle = True)

test = fetch\_20newsgroups(subset = 'test', categories = categories, shuffle = True)

**countVectorizer = CountVectorizer()**

traintf = countVectorizer.**fit\_transform(train.data)**

print('\ntf train count:', traintf.shape)

testtf = countVectorizer.**transform(test.data)**

print('tf test count:', testtf.shape)

**tfidftransformer = TfidfTransformer()**

traintfidf = tfidftransformer.**fit\_transform(traintf)**

print('\ntf train count:', traintf.shape)

testtfidf = tfidftransformer.**transform(testtf)**

print('tf test count:', testtf.shape)

**model = MultinomialNB()**

**model.fit(traintfidf, train.target)**

**predicted = model.predict(testtfidf)**

**print('Accuracy score:', accuracy\_score(test.target, predicted))**

**print(classification\_report(test.target, predicted, target\_names = test.target\_names))**

**print('Confusion Matrix: ', confusion\_matrix(test.target, predicted))**

**7 BAYESIAN BELIEF NETWORK**

**Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set. You can use Java/Python ML library classes/API.**

import numpy as np

import pandas as pd

import urllib

from urllib.request import urlopen

import pgmpy

**from pgmpy.models import BayesianModel**

**from pgmpy.estimators import MaximumLikelihoodEstimator**

**from pgmpy.inference import VariableElimination**

**url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.hungarian.data'**

**np.set\_printoptions(threshold = np.nan)**

**names = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'heartdisease']**

heartDisease = **pd.read\_csv**(urlopen(url), names = names)

print(heartDisease.head())

del heartDisease['oldpeak']

del heartDisease['slope']

del heartDisease['ca']

del heartDisease['thal']

heartDisease = heartDisease.replace('?', np.nan)

print(heartDisease.dtypes)

**model = BayesianModel([('age', 'trestbps'), ('age', 'fbs'), ('sex', 'trestbps'), ('sex', 'trestbps'), ('exang', 'trestbps'), ('trestbps', 'heartdisease'), ('fbs', 'heartdisease'), ('heartdisease', 'restecg'), ('heartdisease', 'thalach'), ('heartdisease', 'chol')])**

model.fit(heartDisease, estimator = MaximumLikelihoodEstimator)

print(model.get\_cpds('age'))

print(model.get\_cpds('chol'))

print(model.get\_cpds('sex'))

**model.get\_independencies()**

**inference = VariableElimination(model)**

q = inference.query(variables = ['heartdisease'], **evidence = {'age': 28}**)

print(q['heartdisease'])

q = inference.query(variables = ['heartdisease'], **evidence = {'chol': 100}**)

print(q['heartdisease'])

**8 KMEANS AND EM**

**Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using *k*-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.**

import numpy as np

import pandas as pd

**from sklearn.cluster import KMeans**

**from sklearn.mixture import GaussianMixture**

from sklearn.datasets import load\_iris

from sklearn.metrics import accuracy\_score, confusion\_matrix

**from sklearn import preprocessing**

**import matplotlib.pyplot as plt**

l1 = [0, 1, 2]

**def rename(S):**

**l2 = []**

**for i in S:**

**if i not in l2:**

**l2.append(i)**

**for i in S:**

**pos = l2.index(i)**

**i = l1[pos]**

**return S**

iris = load\_iris()

print('Data', iris.data)

print('Target names:', iris.target\_names)

print('Target:', iris.target)

**x = pd.DataFrame(iris.data)**

x.columns = ['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']

**y = pd.DataFrame(iris.target)**

y.columns = ['Targets']

**model = KMeans(n\_clusters = 3)**

**model.fit(x)**

plt.figure(figsize = (14, 7))

colormap = np.array(['red', 'lime', 'black'])

plt.subplot(1, 2, 1)

plt.scatter(x.PetalLength, x.PetalWidth, c = colormap[**y.Targets**], s = 40)

plt.title(**'Real Classification'**)

plt.subplot(1, 2, 2)

plt.scatter(x.PetalLength, x.PetalWidth, c = colormap[**model.labels\_**], s = 40)

plt.title(**'KMeans Classification'**)

plt.show()

**km = rename(model.labels\_)**

print('What KMeans thought:', km)

print('Accuracy score of KMeans:', accuracy\_score(y, km))

print('Confusion matris of KMeans:', confusion\_matrix(y, km))

**scaler = preprocessing.StandardScaler()**

**scaler.fit(x)**

**xsa = scaler.transform(x)**

**xs = pd.DataFrame(xsa, columns = x.columns)**

**print('\n', xs.sample(5))**

**gmm = GaussianMixture(n\_components = 3)**

**gmm.fit(xs)**

**y\_cluster\_gmm = gmm.predict(xs)**

plt.subplot(1, 2, 1)

plt.scatter(x.PetalLength, x.PetalWidth, c = colormap[**y\_cluster\_gmm**], s = 40)

plt.title(**'GMM Classification'**)

plt.show()

**em = rename(y\_cluster\_gmm)**

print('What EM thought:', km)

print('Accuracy score of EM:', accuracy\_score(y, em))

print('Confusion matris of EM:', confusion\_matrix(y, em))

**9 K NEAREST NEIGHBOUR**

**Write a program to implement *k*-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.**

import numpy as np

from sklearn.datasets import load\_iris

**from sklearn.model\_selection import train\_test\_split**

**from sklearn.neighbors import KNeighborsClassifier**

dataset = load\_iris()

print('IRIS FEATURES | TARGET NAMES:', dataset.target\_names)

print('\nData:', dataset["data"])

print('\nTarget', dataset["target"])

**xtrain, xtest, ytrain, ytest = train\_test\_split(dataset["data"], dataset["target"], random\_state = 0)**

print("\nX TRAIN \n", xtrain)

print("\nX TEST \n", xtest)

print("\nY TRAIN \n", ytrain)

print("\nY TEST \n", ytest)

kn = KNeighborsClassifier(n\_neighbors = 1)

kn.fit(xtrain, ytrain)

predictions = kn.predict(xtest)

for i in range(len(xtest)):

print("\nActual: {0} {1} \nPredicted: {2} {3}".format(ytest[i], dataset["target\_names"][ytest[i]], predictions, dataset["target\_names"][predictions]))

print("\nTEST SCORE[ACCURACY]: {:.2f}\n".format(kn.score(xtest, ytest)))

**10 REGRESSION – BOKEH IS LIFE!!!**

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

import numpy as np

from bokeh.layouts import gridplot

from bokeh.plotting import figure, show

def local\_regression(x0, X, Y, tau):

x0 = np.r\_[1, x0]

X = np.c\_[np.ones(len(X)), X]

**xw = X.T \* np.exp(np.sum((X - x0) \*\* 2, axis = 1) / (-2 \* tau \*\* 2))**

**beta = np.linalg.pinv(xw @ X) @ xw @ Y**

return x0 @ beta

n = 1000

**X = np.linspace(-3, 3, num = n)**

print('The **dataset**(10 samples) X is:', X[1: 10])

**Y = np.log(np.abs(X \*\* 2 - 1) + 0.5)**

print('\nThe **fitted curve dataset**(10 samples) Y is:', Y[1: 10])

**X += np.random.normal(scale = 0.1, size = n)**

print('\nThe **normalized dataset**(10 samples) X is:', X[1: 10])

**domain = np.linspace(-3, 3, num = 300)**

print('\nThe domain (10 samples) is:', domain[1: 10])

def plot\_lwr(tau):

prediction = [local\_regression(x0, X, Y, tau) for x0 in domain]

plot = figure(plot\_width = 400, plot\_height = 400)

plot.title.text = 'tau: %g' % tau

plot.scatter(X, Y, alpha = 0.3)

**plot.line(domain, prediction, line\_width = 2, color = 'red')**

return plot

**show(gridplot([[plot\_lwr(10), plot\_lwr(1)], [plot\_lwr(.1), plot\_lwr(.01)]]))**