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)

1. **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()

1. **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)

1. **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)

1. **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)

1. **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()

1. **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))

1. **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()

1. **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()

1. **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))

1. **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))

1. **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)