# FIND-S

# Implementation Find-S

# Step-1 : Initiliaze h with most specific hypothesis in H

# h will reject every instance

h=['phi','phi','phi','phi','phi','phi'] # 'phi' indicate no value is accepted. 'any' indicates every value is accepted.

# Training Data

Data=[['Sunny','Warm','Normal','Strong','Warm','Same','Yes'],

['Sunny','Warm','High','Strong','Warm','Same','Yes'],

['Sunny','Warm','Normal','Strong','Warm','Same','No'],

['Sunny','Warm','High','Strong','Cool','Change','Yes']

]

# Step-2 : Iterate over training data and replace constraints in hypothesis with more general constraints

'''

Function: isConsistent(hypothesis,data)

This function check whether given hypothesis is consitent with given data instance.

It compare each attribute's value in hypothesis with respective attribute's value in data.

'''

def isConsistent(h,d):

# Check number of attribute is hypothesis is one less than number of attribute in data.

# Since one attribute in data is class attribute which is not considered in hypothesis.

if len(h)!=len(d)-1:

print('Number of attributes are not same in hypothesis.')

return False

else:

# variable 'matched keeps number of attributes which are consistent in hypothesis.

matched=0

# Iterate over each attribute in hypothesis

for i in range(len(h)):

# Check if attribute in hypothesis is equal to repsective attribute's value in data instance or

# it has 'any' value.

if ( (h[i]==d[i]) | (h[i]=='any') ):

# if condition is satisfied then increase matched

matched=matched+1

# Return true if for all attribute's value in data, hypothesis is consistent.

if matched==len(h):

return True

else:

return False

'''

Function: makeConsistent(hypothesis,data)

This function change hypothesis to make it consistent with given data instance.

'''

def makeConsistent(h,d):

# Iterate over each attribute in hypothesis

for i in range(len(h)):

# if ith attribute in hypothesis reject each value. 'phi' indicates that each value is rejected.

if((h[i] == 'phi')):

# Replace ith value in hypothesis with data instance's ith attribute value.

h[i]=d[i]

# if hypothesis ith value is not 'phi' and it is also not equal to ith value in data instance.

elif(h[i]!=d[i]):

# Replace ith value in hypothesis with 'any'. 'any' accept each value for that attribute.

h[i]='any'

# Return updated hypothesis

return h

print('Begin : Hypothesis :',h)

print('==========================================')

# Iterate over each data instance in given training data

for d in Data:

# Consider only positive instance ( instance with 'Yes' class)

if d[len(d)-1]=='Yes':

# Check whether hypothesis is consistent with current data instance

if ( isConsistent(h,d)):

# Print hypothesis

print ("Hypothesis :",d)

else:

# If hypothesis is not consistent then make it consistent with current data instance

h=makeConsistent(h,d)

# Print current data instance and updated hypothesis

print ('Training data :',d)

print ('Updated Hypothesis :',h)

print()

print('--------------------------------')

print('==========================================')

print('End: Hypothesis :',h)

# Candidate

import csv

with open("enjoysport.csv") as f:

csv\_file=csv.reader(f)

data=list(csv\_file)

for i in data:

if i[-1]=="Yes":

s=i[:-1]

break

print(s)

g=[['?' for i in range(len(s))] for j in range(len(s))]

for i in data:

if i[-1]=="Yes":

for j in range(len(s)):

if i[j]!=s[j]:

s[j]='?'

g[j][j]='?'

elif i[-1]=="No":

for j in range(len(s)):

if i[j]!=s[j]:

g[j][j]=s[j]

else:

g[j][j]="?"

print("\nSteps of Candidate Elimination Algorithm",data.index(i)+1)

print(s)

print(g)

gh=[]

for i in g:

for j in i:

if j!='?':

gh.append(i)

break

print("\nFinal specific hypothesis:\n",s)

print("\nFinal general hypothesis:\n",gh)

# K-means

import numpy as np

import pandas as pd

from copy import deepcopy

def euclidean(a,b, ax=1):

return np.linalg.norm(a-b, axis=ax)

def main():

k = 3

X = pd.read\_csv('kmeans.csv',index\_col=False)

print(X)

x1 = X['X1'].values

x2 = X['X2'].values

X = np.array(list(zip(x1, x2)))

print(X)

C\_x = [6.2, 6.6 ,6.5]

C\_y = [3.2, 3.7, 3.0]

Centroid = np.array(list(zip(C\_x, C\_y)), dtype=np.float32)

print("Initial Centroids")

print(Centroid.shape)

Centroid\_old = np.zeros(Centroid.shape)

print(Centroid\_old)

# Cluster Lables(0, 1, 2)

clusters = np.zeros(len(X))

print(clusters)

error = euclidean(Centroid, Centroid\_old, None)

print(error)

iterr = 0

# Loop will run till the error becomes zero

while error != 0:

# Assigning each value to its closest cluster

iterr = iterr + 1

for i in range(len(X)):

#print(X[i])

distances = euclidean(X[i], Centroid)

#print(distances)

cluster = np.argmin(distances)

clusters[i] = cluster

Centroid\_old = deepcopy(Centroid)

# Finding the new centroids by taking the Mean

for p in range(k):

points = [X[j] for j in range(len(X)) if clusters[j] == p]

Centroid[p] = np.mean(points, axis=0)

print(" Centre of the clusters after ", iterr," Iteration \n", Centroid)

error = euclidean(Centroid, Centroid\_old, None)

print("Error ... ",error)

if \_\_name\_\_ == "\_\_main\_\_":

main()

# 

# Decision tree

from sklearn.tree import DecisionTreeClassifier

import pandas as pd

import numpy as np

from sklearn import metrics

from sklearn.metrics import confusion\_matrix

from sklearn.tree import export\_graphviz

import graphviz

from sklearn import tree

###########################################################################################################

##########################################################################################################

"""

Import the Zoo Dataset

"""

#Import the dataset

dataset = pd.read\_csv('zoo\_data.csv')

#We drop the animal names since this is not a good feature to split the data on

#dataset=dataset.drop('animal\_name',axis=1)

###########################################################################################################

##########################################################################################################

"""

Split the data into a training and a testing set

"""

train\_features = dataset.iloc[:80,:-1]

test\_features = dataset.iloc[80:,:-1]

train\_targets = dataset.iloc[:80,-1]

test\_targets = dataset.iloc[80:,-1]

###########################################################################################################

##########################################################################################################

"""

Train the model

"""

tree1 = DecisionTreeClassifier(criterion = 'entropy').fit(train\_features,train\_targets)

export\_graphviz(tree1, out\_file="mytree.dot")

with open("mytree.dot") as f:

dot\_graph = f.read()

graphviz.Source(dot\_graph)

tree.plot\_tree(tree1)

###########################################################################################################

##########################################################################################################

"""

Predict the classes of new, unseen data

"""

prediction = tree1.predict(test\_features)

cm = confusion\_matrix(test\_targets, prediction)

print('\n'.join([''.join(['{:4}'.format(item) for item in row]) for row in cm]))

#confusionmatrix = np.matrix(cm)

FP = cm.sum(axis=0) - np.diag(cm)

FN = cm.sum(axis=1) - np.diag(cm)

TP = np.diag(cm)

TN = cm.sum() - (FP + FN + TP)

print('False Positives\n {}'.format(FP))

print('False Negetives\n {}'.format(FN))

print('True Positives\n {}'.format(TP))

print('True Negetives\n {}'.format(TN))

TPR = TP/(TP+FN)

print('Sensitivity \n {}'.format(TPR))

TNR = TN/(TN+FP)

print('Specificity \n {}'.format(TNR))

Precision = TP/(TP+FP)

print('Precision \n {}'.format(Precision))

Recall = TP/(TP+FN)

print('Recall \n {}'.format(Recall))

Acc = (TP+TN)/(TP+TN+FP+FN)

print('Áccuracy \n{}'.format(Acc))

Fscore = 2\*((Precision\*Recall)/(Precision+Recall))

print('FScore \n{}'.format(Fscore))

###########################################################################################################

##########################################################################################################

# Linear Regression

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

def estimate\_coef(x, y):

# number of observations/points

n = np.size(x)

# mean of x and y vector

m\_x, m\_y = np.mean(x), np.mean(y)

# calculating cross-deviation and deviation about x

SS\_xy = np.sum(y\*x) - n\*m\_y\*m\_x

SS\_xx = np.sum(x\*x) - n\*m\_x\*m\_x

# calculating regression coefficients

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):

# plotting the actual points as scatter plot

plt.scatter(x, y, color = "m",

marker = "o", s = 30)

# predicted response vector

y\_pred = b[0] + b[1]\*x

# plotting the regression line

plt.plot(x, y\_pred, color = "g")

# putting labels

plt.xlabel('x')

plt.ylabel('y')

# function to show plot

plt.show()

def main():

# observations

dataset = pd.read\_csv('Food-Truck-LineReg.csv')

x = dataset.iloc[:97,0]

x = np.array(x)

y = dataset.iloc[:97,1]

y=np.array(y)

# estimating coefficients

b = estimate\_coef(x, y)

print("Estimated coefficients: b\_0 = {} b\_1 = {}".format(b[0], b[1]))

# plotting regression line

plot\_regression\_line(x, y, b)

if \_\_name\_\_ == "\_\_main\_\_":

main()

# Logistic

with library

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from sklearn import metrics

from sklearn.metrics import confusion\_matrix

data = pd.read\_csv('heart.csv')

x=data.drop('target',axis=1)

y=data.target

x\_train,x\_test,y\_train,y\_test = train\_test\_split(x,y,test\_size=0.3,random\_state=109)

# Training the Logistic Regression model on the Training set

from sklearn.linear\_model import LogisticRegression

classifier = LogisticRegression(random\_state = 0)

classifier.fit(x\_train, y\_train)

y\_pred=classifier.predict(x\_test)

cm = confusion\_matrix(y\_test,y\_pred)

print(cm)

fp=cm.sum(axis=0)-np.diag(cm)

fn=cm.sum(axis=1)-np.diag(cm)

tp=np.diag(cm)

tn=cm.sum()-(fp+fn+tp)

print("false positives:{}".format(fp))

print("false negatives:{}".format(fn))

print("true positives:{}".format(tp))

print("true negatives:{}".format(tn))

tnr = tn/(tn+fp)

print("tnr:{}".format(tnr))

tpr = tp/(tp+fn)

print("tpr:{}".format(tpr))

acc = (tp+tn)/(tp+tn+fp+fn)

print("acc: {}".format(acc))

recall = (tp/(tp+fp))

print("recall:{}".format(recall))

precision = (tp/(tp+fn))

print("precision:{}".format(precision))

Fscore = 2\*(precision\*recall)/(precision+recall)

print("Fscore: {}".format(Fscore))

without library

# Logistic Regression on Diabetes Dataset

from random import seed

from random import randrange

from csv import reader

from math import exp

# Load a CSV file

def load\_csv(filename):

dataset = list()

with open(filename, 'r') as file:

csv\_reader = reader(file)

for row in csv\_reader:

if not row:

continue

dataset.append(row)

return dataset

# Convert string column to float

def str\_column\_to\_float(dataset, column):

for row in dataset:

row[column] = float(row[column].strip())

# Find the min and max values for each column

def dataset\_minmax(dataset):

minmax = list()

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

col\_values = [row[i] for row in dataset]

value\_min = min(col\_values)

value\_max = max(col\_values)

minmax.append([value\_min, value\_max])

return minmax

# Rescale dataset columns to the range 0-1

def normalize\_dataset(dataset, minmax):

for row in dataset:

for i in range(len(row)):

row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds

def cross\_validation\_split(dataset, n\_folds):

dataset\_split = list()

dataset\_copy = list(dataset)

fold\_size = int(len(dataset) / n\_folds)

for i in range(n\_folds):

fold = list()

while len(fold) < fold\_size:

index = randrange(len(dataset\_copy))

fold.append(dataset\_copy.pop(index))

dataset\_split.append(fold)

return dataset\_split

# Calculate accuracy percentage

def accuracy\_metric(actual, predicted):

correct = 0

for i in range(len(actual)):

if actual[i] == predicted[i]:

correct += 1

return correct / float(len(actual)) \* 100.0

# Evaluate an algorithm using a cross validation split

def evaluate\_algorithm(dataset, algorithm, n\_folds, \*args):

folds = cross\_validation\_split(dataset, n\_folds)

scores = list()

for fold in folds:

train\_set = list(folds)

train\_set.remove(fold)

train\_set = sum(train\_set, [])

test\_set = list()

for row in fold:

row\_copy = list(row)

test\_set.append(row\_copy)

row\_copy[-1] = None

predicted = algorithm(train\_set, test\_set, \*args)

actual = [row[-1] for row in fold]

accuracy = accuracy\_metric(actual, predicted)

scores.append(accuracy)

print(scores)

return scores

def sigmoid(yhat):

value = 1.0 / (1.0 + exp(-yhat))

return value

# Make a prediction with coefficients

def predict(row, coefficients):

yhat = coefficients[0]

for i in range(len(row)-1):

yhat += coefficients[i + 1] \* row[i]

value = sigmoid(yhat)

return value

# Estimate logistic regression coefficients using stochastic gradient descent

def coefficients\_sgd(train, l\_rate, n\_epoch):

coef = [0.0 for i in range(len(train[0]))]

for epoch in range(n\_epoch):

for row in train:

yhat = predict(row, coef)

error = row[-1] - yhat

coef[0] = coef[0] + l\_rate \* error \* yhat \* (1.0 - yhat)

for i in range(len(row)-1):

coef[i + 1] = coef[i + 1] + l\_rate \* error \* yhat \* (1.0 - yhat) \* row[i]

return coef

# Linear Regression Algorithm With Stochastic Gradient Descent

def logistic\_regression(train, test, l\_rate, n\_epoch):

predictions = list()

coef = coefficients\_sgd(train, l\_rate, n\_epoch)

for row in test:

yhat = predict(row, coef)

yhat = round(yhat)

predictions.append(yhat)

return(predictions)

# Test the logistic regression algorithm on the diabetes dataset

seed(1)

# load and prepare data

filename = 'Student-University.csv'

dataset = load\_csv(filename)

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

str\_column\_to\_float(dataset, i)

# normalize

minmax = dataset\_minmax(dataset)

normalize\_dataset(dataset, minmax)

# evaluate algorithm

n\_folds = 5

l\_rate = 0.1

n\_epoch = 100

scores = evaluate\_algorithm(dataset, logistic\_regression, n\_folds, l\_rate, n\_epoch)

# Naive Bayes

with library

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from sklearn import metrics

from sklearn.metrics import confusion\_matrix

data = pd.read\_csv('heart.csv')

x=data.drop('target',axis=1)

y=data.target

x\_train,x\_test,y\_train,y\_test = train\_test\_split(x,y,test\_size=0.3,random\_state=109)

from sklearn.naive\_bayes import GaussianNB

#Create a Gaussian Classifier

ml = GaussianNB()

# Train the model using the training sets

ml.fit(x\_train,y\_train)

y\_pred=ml.predict(x\_test)

cm = confusion\_matrix(y\_test,y\_pred)

print('\n'.join([''.join(['{:4}'.format(item) for item in row]) for row in cm]))

FP = cm.sum(axis=0) - np.diag(cm)

FN = cm.sum(axis=1) - np.diag(cm)

TP = np.diag(cm)

TN = cm.sum() - (FP + FN + TP)

print('False Positives\n {}'.format(FP))

print('False Negetives\n {}'.format(FN))

print('True Positives\n {}'.format(TP))

print('True Negetives\n {}'.format(TN))

TPR = TP/(TP+FN)

print('Sensitivity \n {}'.format(TPR))

TNR = TN/(TN+FP)

print('Specificity \n {}'.format(TNR))

Precision = TP/(TP+FP)

print('Precision \n {}'.format(Precision))

Recall = TP/(TP+FN)

print('Recall \n {}'.format(Recall))

Acc = (TP+TN)/(TP+TN+FP+FN)

print('Áccuracy \n{}'.format(Acc))

Fscore = 2\*(Precision\*Recall)/(Precision+Recall)

print('FScore \n{}'.format(Fscore))

without library

import csv

import random

import math

import numpy as np

from sklearn.metrics import confusion\_matrix

#from pandas\_ml import ConfusionMatrix

def loadCsv(filename):

lines = csv.reader(open(filename, 'r'))

dataset = list(lines)

for i in range(len(dataset)):

dataset[i] = [float(x) for x in dataset[i]]

return dataset

def splitData(dataset, sRatio):

trainSize = int(len(dataset) \* sRatio)

trainSet = []

copy = list(dataset)

while len(trainSet) < trainSize:

index = random.randrange(len(copy))

trainSet.append(copy.pop(index))

return [trainSet, copy]

def ClassData(dataset):

classdivision = {}

for i in range(len(dataset)):

vector = dataset[i]

if (vector[-1] not in classdivision):

classdivision[vector[-1]] = []

classdivision[vector[-1]].append(vector)

print(classdivision)

return classdivision

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 process(dataset):

foreveryclass=[]

for attribute in zip(\*dataset):

x = mean(attribute)

y = stdev(attribute)

foreveryclass.append([x,y])

del foreveryclass[-1]

return foreveryclass

def summarizeByClass(dataset):

divided = ClassData(dataset)

#print(separated)

ProcessValues = {} # a dictionary to store mean stdev of all attributes classwise

for classValue, instances in divided.items(): #returns a list of key, value pairs for tuples

ProcessValues[classValue] = process(instances)

#print(ProcessValues)

return ProcessValues

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(ProcessValues, inputVector):

probabilities = {}

for classValue, classSummaries in ProcessValues.items():

probabilities[classValue] = 1

for i in range(len(classSummaries)):

mean, stdev = classSum

maries[i]

x = inputVector[i]

probabilities[classValue] \*= calculateProbability(x, mean, stdev)

#print(probabilities)

return probabilities

def predict(ProcessValues, inputVector):

probabilities = calculateClassProbabilities(ProcessValues, 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(ProcessValues, testSet):

predictions = []

y\_true = []

for i in range(len(testSet)):

result = predict(ProcessValues, testSet[i])

predictions.append(result)

#print(predictions)

for i in range(len(testSet)):

vector=testSet[i]

y\_true.append(vector[-1])

#print(y\_true)

return [y\_true, 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 = 'data.csv'

file = 'Databalancedtest.csv'

sRatio = 0.80

dataset = loadCsv(filename)

trainingSet, testSet = splitData(dataset, sRatio)

#print('Split {} rows into train={} and test={} rows'.format(len(dataset), len(trainingSet), len(testSet)))

# prepare model

ProcessValues = summarizeByClass(trainingSet)

# test model

y\_true, predictions = getPredictions(ProcessValues, testSet)

#print('True Classes of test dataset: {}\n'.format(y\_true))

#print('\nPredicted Classes : {}\n'.format(y\_true))

cm = confusion\_matrix(y\_true, predictions)

#for i in range(6):

#for j in range(6):

#print('{:4}'.format(cm[i][j])),

#print

print('\n\n Confusion Matrix \n')

print('\n'.join([''.join(['{:4}'.format(item) for item in row]) for row in cm]))

#confusionmatrix = np.matrix(cm)

FP = cm.sum(axis=0) - np.diag(cm)

FN = cm.sum(axis=1) - np.diag(cm)

TP = np.diag(cm)

TN = cm.sum() - (FP + FN + TP)

print('False Positives\n {}'.format(FP))

print('False Negetives\n {}'.format(FN))

print('True Positives\n {}'.format(TP))

print('True Negetives\n {}'.format(TN))

TPR = TP/(TP+FN)

print('Sensitivity \n {}'.format(TPR))

TNR = TN/(TN+FP)

print('Specificity \n {}'.format(TNR))

Precision = TP/(TP+FP)

print('Precision \n {}'.format(Precision))

Recall = TP/(TP+FN)

print('Recall \n {}'.format(Recall))

Acc = (TP+TN)/(TP+TN+FP+FN)

print('Áccuracy \n{}'.format(Acc))

Fscore = 2\*(Precision\*Recall)/(Precision+Recall)

print('FScore \n{}'.format(Fscore))

#accuracy = getAccuracy(testSet, predictions)

#print('Accuracy: {}%'.format(accuracy))

main()

# SVM

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split # Import train\_test\_split function

from sklearn import svm #Import svm model

from sklearn import metrics #Import scikit-learn metrics module for accuracy calculation

from sklearn.metrics import confusion\_matrix

data = pd.read\_csv("heart.csv")

## The data looks like this

## age sex cp trestbps chol fbs restecg thalach exang oldpeak slope ca thal target

##0 63 1 3 145 233 1 0 150 0 2.3 0 0 1 1

##1 37 1 2 130 250 0 1 187 0 3.5 0 0 2 1

##2 41 0 1 130 204 0 0 172 0 1.4 2 0 2 1

##3 56 1 1 120 236 0 1 178 0 0.8 2 0 2 1

##4 57 0 0 120 354 0 1 163 1 0.6 2 0 2 1

#Separate the data -- last column 'target' is removed from the input feature set x

x = data.drop('target',axis = 1)

y = data.target

#split the test set and train set

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.3,random\_state=109) # 70% training and 30% test

#Create a svm Classifier

ml = svm.SVC(kernel='linear') # Linear Kernel

#Train the model using the training sets

ml.fit(x\_train, y\_train)

#Predict the response for test dataset

y\_pred = ml.predict(x\_test)

# Model Accuracy: how often is the classifier correct?

#print(ml.score(x\_test,y\_test))

cm = confusion\_matrix(y\_test,y\_pred)

print('\n'.join([''.join(['{:4}'.format(item) for item in row]) for row in cm]))

FP = cm.sum(axis=0) - np.diag(cm)

FN = cm.sum(axis=1) - np.diag(cm)

TP = np.diag(cm)

TN = cm.sum() - (FP + FN + TP)

print('False Positives\n {}'.format(FP))

print('False Negetives\n {}'.format(FN))

print('True Positives\n {}'.format(TP))

print('True Negetives\n {}'.format(TN))

TPR = TP/(TP+FN)

print('Sensitivity \n {}'.format(TPR))

TNR = TN/(TN+FP)

print('Specificity \n {}'.format(TNR))

Precision = TP/(TP+FP)

print('Precision \n {}'.format(Precision))

Recall = TP/(TP+FN)

print('Recall \n {}'.format(Recall))

Acc = (TP+TN)/(TP+TN+FP+FN)

print('Áccuracy \n{}'.format(Acc))

Fscore = 2\*(Precision\*Recall)/(Precision+Recall)

print('FScore \n{}'.format(Fscore))

# 

# Random Forest

#Import scikit-learn dataset library

from sklearn import datasets

#Load dataset

iris = datasets.load\_iris()

# print the label species(setosa, versicolor,virginica)

print(iris.target\_names)

# print the names of the four features

print(iris.feature\_names)

# print the iris data (top 5 records)

print(iris.data[0:5])

# print the iris labels (0:setosa, 1:versicolor, 2:virginica)

print(iris.target)

# Creating a DataFrame of given iris dataset.

import pandas as pd

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()

# Import train\_test\_split function

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

X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features

y=data['species'] # Labels

# Split dataset into training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3) # 70% training and 30% test

#Import Random Forest Model

from sklearn.ensemble import RandomForestClassifier

#Create a Gaussian Classifier

clf=RandomForestClassifier(n\_estimators=100)

#Train the model using the training sets y\_pred=clf.predict(X\_test)

clf.fit(X\_train,y\_train)

y\_pred=clf.predict(X\_test)

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

# Model Accuracy, how often is the classifier correct?

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))