

**NAME : PAVITHRA B**

**REG. NO. : 21BIT0677**

**COURSE : MACHINE LEARNING**

**COURSE CODE : BITE410L**

**DATE : 17/04/24**

DATASET NAME: HEART DISEASE PREDICTION

1. Data Collection: The first step is gathering relevant data related to heart disease. This may include various factors such as age, sex, blood pressure, cholesterol levels, presence of diabetes, family history of heart disease, lifestyle factors (like smoking and exercise habits), and results from various medical tests.

2. Data Preprocessing: Once the data is collected, it needs to be preprocessed to ensure its quality and usability. This can involve handling missing values, removing outliers, and scaling or normalizing numerical features. Categorical variables may also need to be encoded into numerical format.

3. Splitting the Data: The dataset is then typically divided into two subsets: a training set and a testing set. The training set is used to train the Random Tree Classifier, while the testing set is used to evaluate its performance.

4. Model Training: In this step, the Random Tree Classifier algorithm is applied to the training data. A Random Tree Classifier is an ensemble learning method that constructs a multitude of decision trees at training time and outputs the mode of the classes (classification) or mean prediction (regression) of the individual trees. Each tree is constructed using a random subset of the features and a random subset of the training data.

5. Model Evaluation: After the model is trained, it needs to be evaluated to assess its performance. This is typically done using the testing set, where the model's predictions are compared to the actual outcomes. Common evaluation metrics for classification tasks include accuracy, precision, recall, F1-score, and area under the receiver operating characteristic curve (AUC-ROC).

.

Overall, the goal of heart disease prediction using a Random Tree Classifier is to develop a reliable model that can assist healthcare professionals in identifying individuals who are at risk of developing heart disease, allowing for early intervention and prevention strategies to be implemented.

WITH LIBRARIES

CODE:

# Importing necessary libraries

print("PAVITHRA B-21BIT0677")

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.ensemble import RandomForestClassifier

# Load the dataset

data = pd.read\_csv(r'C:\Users\Admin\Desktop\ML\heart.csv')

# Data preprocessing

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

y = data['target']

# Splitting the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Feature scaling

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Train the model

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

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

# Predictions

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

# Performance metrics

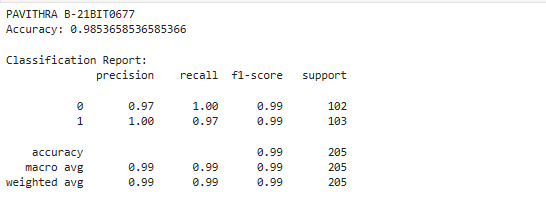
accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred))

OUTPUT:



WITHOUT LIBRARIES

Code:

import csv

import random

import math

print("PAVITHRA B-21BIT0677")

# Load the dataset

def load\_csv(filename):

dataset = []

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

csv\_reader = csv.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())

# Convert string column to integer

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

class\_values = [row[column] for row in dataset]

unique = set(class\_values)

lookup = dict()

for i, value in enumerate(unique):

lookup[value] = i

for row in dataset:

row[column] = lookup[row[column]]

return lookup

# Split dataset into k folds

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

dataset\_split = []

dataset\_copy = list(dataset)

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

for \_ in range(n\_folds):

fold = []

while len(fold) < fold\_size:

index = random.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

# Calculate precision

def precision\_metric(tp, fp):

return tp / (tp + fp)

# Calculate recall

def recall\_metric(tp, fn):

return tp / (tp + fn)

# Calculate F1 score

def f1\_score\_metric(precision, recall):

return 2 \* ((precision \* recall) / (precision + recall))

# Evaluate an algorithm using a cross validation split

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

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

scores = []

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)

return scores

# Split a dataset based on an attribute and an attribute value

def test\_split(index, value, dataset):

left, right = [], []

for row in dataset:

if row[index] < value:

left.append(row)

else:

right.append(row)

return left, right

# Calculate the Gini index for a split dataset

def gini\_index(groups, classes):

n\_instances = float(sum(len(group) for group in groups))

gini = 0.0

for group in groups:

size = float(len(group))

if size == 0:

continue

score = 0.0

for class\_val in classes:

p = [row[-1] for row in group].count(class\_val) / size

score += p \* p

gini += (1.0 - score) \* (size / n\_instances)

return gini

# Select the best split point for a dataset

def get\_split(dataset, n\_features):

class\_values = list(set(row[-1] for row in dataset))

b\_index, b\_value, b\_score, b\_groups = 999, 999, 999, None

features = []

while len(features) < n\_features:

index = random.randrange(len(dataset[0])-1)

if index not in features:

features.append(index)

for index in features:

for row in dataset:

groups = test\_split(index, row[index], dataset)

gini = gini\_index(groups, class\_values)

if gini < b\_score:

b\_index, b\_value, b\_score, b\_groups = index, row[index], gini, groups

return {'index': b\_index, 'value': b\_value, 'groups': b\_groups}

# Create a terminal node value

def to\_terminal(group):

outcomes = [row[-1] for row in group]

return max(set(outcomes), key=outcomes.count)

# Create child splits for a node or make terminal

def split(node, max\_depth, min\_size, n\_features, depth):

left, right = node['groups']

del(node['groups'])

if not left or not right:

node['left'] = node['right'] = to\_terminal(left + right)

return

if depth >= max\_depth:

node['left'], node['right'] = to\_terminal(left), to\_terminal(right)

return

if len(left) <= min\_size:

node['left'] = to\_terminal(left)

else:

node['left'] = get\_split(left, n\_features)

split(node['left'], max\_depth, min\_size, n\_features, depth+1)

if len(right) <= min\_size:

node['right'] = to\_terminal(right)

else:

node['right'] = get\_split(right, n\_features)

split(node['right'], max\_depth, min\_size, n\_features, depth+1)

# Build a decision tree

def build\_tree(train, max\_depth, min\_size, n\_features):

root = get\_split(train, n\_features)

split(root, max\_depth, min\_size, n\_features, 1)

return root

# Make a prediction with a decision tree

def predict(node, row):

if row[node['index']] < node['value']:

if isinstance(node['left'], dict):

return predict(node['left'], row)

else:

return node['left']

else:

if isinstance(node['right'], dict):

return predict(node['right'], row)

else:

return node['right']

# Create a random subsample from the dataset with replacement

def subsample(dataset, ratio):

sample = []

n\_sample = round(len(dataset) \* ratio)

while len(sample) < n\_sample:

index = random.randrange(len(dataset))

sample.append(dataset[index])

return sample

# Make a prediction with a list of bagged trees

def bagging\_predict(trees, row):

predictions = [predict(tree, row) for tree in trees]

return max(set(predictions), key=predictions.count)

# Random Forest Algorithm

def random\_forest(train, test, max\_depth, min\_size, sample\_size, n\_trees, n\_features):

trees = []

for \_ in range(n\_trees):

sample = subsample(train, sample\_size)

tree = build\_tree(sample, max\_depth, min\_size, n\_features)

trees.append(tree)

predictions = [bagging\_predict(trees, row) for row in test]

return predictions

# Load dataset

filename = r"C:\Users\Admin\Desktop\ML\heart.csv"

dataset = load\_csv(filename)

# Skip the header row

dataset = dataset[1:]

# Convert string attributes to integers

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

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

# Convert class column to integers

str\_column\_to\_int(dataset, len(dataset[0])-1)

# Evaluate algorithm

n\_folds = 5

max\_depth = 10

min\_size = 1

sample\_size = 1.0

n\_features = int(math.sqrt(len(dataset[0])-1))

for n\_trees in [1, 5, 10]:

scores = evaluate\_algorithm(dataset, random\_forest, n\_folds, max\_depth, min\_size, sample\_size, n\_trees, n\_features)

print('Trees: %d' % n\_trees)

print('Scores: %s' % scores)

print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))

# Train the final model using all data

final\_model = random\_forest(dataset, dataset, max\_depth, min\_size, sample\_size, n\_trees, n\_features)

predictions = final\_model

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

# Calculate TP, FP, TN, FN

tp = sum(1 for p, a in zip(predictions, actual) if p == 1 and a == 1)

fp = sum(1 for p, a in zip(predictions, actual) if p == 1 and a == 0)

tn = sum(1 for p, a in zip(predictions, actual) if p == 0 and a == 0)

fn = sum(1 for p, a in zip(predictions, actual) if p == 0 and a == 1)

# Calculate precision

precision = precision\_metric(tp, fp)

# Calculate recall

recall = recall\_metric(tp, fn)

# Calculate F1 score

f1\_score = f1\_score\_metric(precision, recall)

# Calculate accuracy

accuracy = accuracy\_metric(actual, predictions)

print("\nPerformance Metrics:")

print(f"Precision: {precision}")

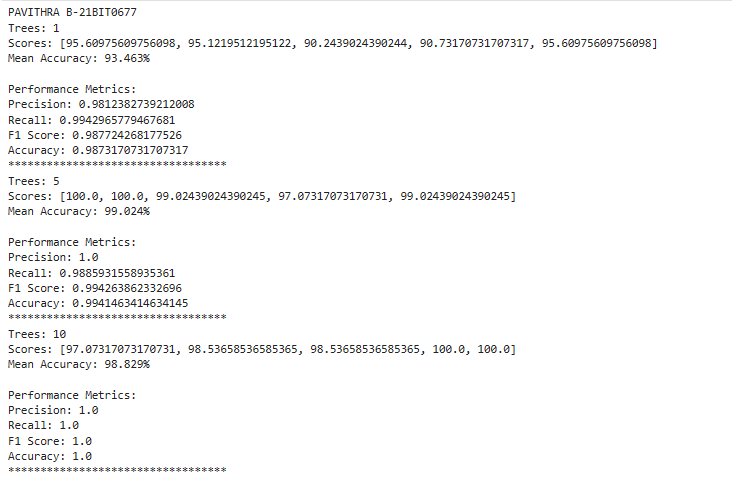
print(f"Recall: {recall}")

print(f"F1 Score: {f1\_score}")

print(f"Accuracy: {accuracy/100}")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

OUTPUT:



THANK YOU…