**PROJECT: DIABETES DETECTOR**

**COURSE: AI (111700)**

**SUBMITTED TO: SIR AKRAM**

**Group Members:**

**TALHA AHMED SHAIKH || 11377**

**HARIS AHMED || 11269**

**ZAIN AHMED || 10930**

**LOGISTIC REGRESSION CODE:**

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

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

from sklearn.metrics import accuracy\_score

diabetes\_data = pd.read\_csv('kaggle\_diabetes.csv')

class Logistic\_Regression():

  def \_\_init\_\_(self, learning\_rate, no\_of\_iterations):

    self.learning\_rate = learning\_rate

    self.no\_of\_iterations = no\_of\_iterations

  def fit(self, X, Y):

    self.m, self.n = X.shape

    self.w = np.zeros(self.n)

    self.b = 0

    self.X = X

    self.Y = Y

    for i in range(self.no\_of\_iterations):

      self.update\_weights()

  def update\_weights(self):

    Y\_hat = 1 / (1 + np.exp( - (self.X.dot(self.w) + self.b ) ))

    dw = (1/self.m)\*np.dot(self.X.T, (Y\_hat - self.Y))

    db = (1/self.m)\*np.sum(Y\_hat - self.Y)

    self.w = self.w - self.learning\_rate \* dw

    self.b = self.b - self.learning\_rate \* db

  def predict(self, X):

    Y\_pred = 1 / (1 + np.exp(- (X.dot(self.w) + self.b)))

    Y\_pred = np.where( Y\_pred > 0.5, 1, 0)

    return Y\_pred

features = diabetes\_data.drop(columns = 'Outcome', axis=1)

target = diabetes\_data['Outcome']

scaler = StandardScaler()

scaler.fit(features)

standardized\_data = scaler.transform(features)

features = standardized\_data

target = diabetes\_data['Outcome']

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(features,target, test\_size = 0.2, random\_state=2)

print(features.shape, X\_train.shape, X\_test.shape)

classifier = Logistic\_Regression(learning\_rate=0.01, no\_of\_iterations=1000)

classifier.fit(X\_train, Y\_train)

X\_train\_prediction = classifier.predict(X\_train)

training\_data\_accuracy = accuracy\_score( Y\_train, X\_train\_prediction)

print('Accuracy score of the training data : ', training\_data\_accuracy\*100)

input\_data = (4,110,76,20,100,28.4,0.118,27)

#4,110,76,20,100,28.4,0.118,27       nd

#5, 120, 92, 10, 81, 26.1, 0.551, 67 nd

#2, 81, 72, 15, 76, 30.1, 0.547, 25  nd

#1, 168, 88, 29, 0, 35, 0.905, 52    d

input\_data\_as\_numpy\_array = np.asarray(input\_data)

input\_data\_reshaped = input\_data\_as\_numpy\_array.reshape(1,-1)

std\_data = scaler.transform(input\_data\_reshaped)

print(std\_data)

prediction = classifier.predict(std\_data)

print(prediction)

if (prediction[0] == 0):

  print('The person is not diabetic')

else:

  print('The person is diabetic')

**RANDOM FOREST CODE:**

import numpy as np

import pandas as pd

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

diabetes\_data = pd.read\_csv('kaggle\_diabetes.csv')

class Node:

    def \_\_init\_\_(self, feature=None, value=None, result=None):

        self.feature = feature

        self.value = value

        self.result = result

        self.left = None

        self.right = None

class DecisionTreeClassifier:

    def \_\_init\_\_(self, max\_depth=None):

        self.max\_depth = max\_depth

    def calculate\_gini\_index(self, y):

        classes, counts = np.unique(y, return\_counts=True)

        gini = 1

        for count in counts:

            gini -= (count / len(y)) \*\* 2

        return gini

    def split\_dataset(self, X, y, feature, value):

        left\_X = X[X[:, feature] <= value]

        left\_y = y[X[:, feature] <= value]

        right\_X = X[X[:, feature] > value]

        right\_y = y[X[:, feature] > value]

        return left\_X, left\_y, right\_X, right\_y

    def find\_split\_point(self, X, y):

        best\_gini = 1

        best\_feature = None

        best\_value = None

        for feature in range(X.shape[1]):

            unique\_values = np.unique(X[:, feature])

            for value in unique\_values:

                left\_X, left\_y, right\_X, right\_y = self.split\_dataset(X, y, feature, value)

                gini = (len(left\_y) \* self.calculate\_gini\_index(left\_y) + len(right\_y) \* self.calculate\_gini\_index(right\_y)) / len(y)

                if gini < best\_gini:

                    best\_gini = gini

                    best\_feature = feature

                    best\_value = value

        return best\_feature, best\_value

    def build\_tree(self, X, y, depth=0):

        if depth == self.max\_depth or len(np.unique(y)) == 1:

            return Node(result=np.argmax(np.bincount(y)))

        feature, value = self.find\_split\_point(X, y)

        if feature is None or value is None:

            return Node(result=np.argmax(np.bincount(y)))

        left\_X, left\_y, right\_X, right\_y = self.split\_dataset(X, y, feature, value)

        left\_node = self.build\_tree(left\_X, left\_y, depth + 1)

        right\_node = self.build\_tree(right\_X, right\_y, depth + 1)

        node = Node(feature=feature, value=value)

        node.left = left\_node

        node.right = right\_node

        return node

    def fit(self, X, y):

        self.tree = self.build\_tree(X, y)

    def predict(self, X):

        predictions = []

        for instance in X:

            node = self.tree

            while node.result is None:

                if instance[node.feature] <= node.value:

                    node = node.left

                else:

                    node = node.right

            predictions.append(node.result)

        return np.array(predictions)

class RandomForestClassifier:

    def \_\_init\_\_(self, n\_estimators=100, max\_depth=None):

        self.n\_estimators = n\_estimators

        self.max\_depth = max\_depth

        self.estimators = []

    def fit(self, X, y):

        for \_ in range(self.n\_estimators):

            indices = np.random.choice(len(X), size=len(X), replace=True)

            X\_subset = X[indices]

            y\_subset = y[indices]

            dt = DecisionTreeClassifier(max\_depth=self.max\_depth)

            dt.fit(X\_subset, y\_subset)

            self.estimators.append(dt)

    def predict(self, X):

        predictions = []

        for instance in X:

            ensemble\_predictions = [estimator.predict([instance])[0] for estimator in self.estimators]

            prediction = np.argmax(np.bincount(ensemble\_predictions))

            predictions.append(prediction)

        return np.array(predictions)

X = diabetes\_data.drop(columns='Outcome').values

y = diabetes\_data['Outcome'].values

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

random\_forest = RandomForestClassifier(n\_estimators=100, max\_depth=5)

random\_forest.fit(X\_train, y\_train)

predictions = random\_forest.predict(X\_test)

accuracy = np.sum(predictions == y\_test) / len(y\_test)

print("Accuracy:", accuracy\*100)

input\_data = np.array([1, 168, 88, 29, 0, 35, 0.905, 52]).reshape(1, -1)

#4,110,76,20,100,28.4,0.118,27       nd

#5, 120, 92, 10, 81, 26.1, 0.551, 67 nd

#2, 81, 72, 15, 76, 30.1, 0.547, 25  nd

#1, 168, 88, 29, 0, 35, 0.905, 52    d

prediction = random\_forest.predict(input\_data)

if prediction == 0:

    print('The person is not diabetic')

else:

    print('The person is diabetic')

**DECISION-TREE CODE:**

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

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

from sklearn.metrics import accuracy\_score

diabetes\_data = pd.read\_csv('kaggle\_diabetes.csv')

class Node:

    def \_\_init\_\_(self, feature=None, value=None, result=None):

        self.feature = feature

        self.value = value

        self.result = result

        self.left = None

        self.right = None

class DecisionTreeClassifier:

    def \_\_init\_\_(self, max\_depth=None):

        self.max\_depth = max\_depth

    def calculate\_gini\_index(self, y):

        classes, counts = np.unique(y, return\_counts=True)

        gini = 1

        for count in counts:

            gini -= (count / len(y)) \*\* 2

        return gini

    def split\_dataset(self, X, y, feature, value):

        left\_X = X[X[:, feature] <= value]

        left\_y = y[X[:, feature] <= value]

        right\_X = X[X[:, feature] > value]

        right\_y = y[X[:, feature] > value]

        return left\_X, left\_y, right\_X, right\_y

    def find\_split\_point(self, X, y):

        best\_gini = 1

        best\_feature = None

        best\_value = None

        for feature in range(X.shape[1]):

            unique\_values = np.unique(X[:, feature])

            for value in unique\_values:

                left\_X, left\_y, right\_X, right\_y = self.split\_dataset(X, y, feature, value)

                gini = (len(left\_y) \* self.calculate\_gini\_index(left\_y) + len(right\_y) \* self.calculate\_gini\_index(right\_y)) / len(y)

                if gini < best\_gini:

                    best\_gini = gini

                    best\_feature = feature

                    best\_value = value

        return best\_feature, best\_value

    def build\_tree(self, X, y, depth=0):

        if depth == self.max\_depth or len(np.unique(y)) == 1:

            return Node(result=np.argmax(np.bincount(y)))

        feature, value = self.find\_split\_point(X, y)

        if feature is None or value is None:

            return Node(result=np.argmax(np.bincount(y)))

        left\_X, left\_y, right\_X, right\_y = self.split\_dataset(X, y, feature, value)

        left\_node = self.build\_tree(left\_X, left\_y, depth + 1)

        right\_node = self.build\_tree(right\_X, right\_y, depth + 1)

        node = Node(feature=feature, value=value)

        node.left = left\_node

        node.right = right\_node

        return node

    def fit(self, X, y):

        self.tree = self.build\_tree(X, y)

    def predict(self, X):

        predictions = []

        for instance in X:

            node = self.tree

            while node.result is None:

                if instance[node.feature] <= node.value:

                    node = node.left

                else:

                    node = node.right

            predictions.append(node.result)

        return np.array(predictions)

X = diabetes\_data.drop(columns='Outcome').values

y = diabetes\_data['Outcome'].values

print(X.shape[1])

print(X)

print(y)

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

y\_pred = decision\_tree.predict(X\_test)

accuracy = np.sum(y\_pred == y\_test) / len(y\_test)

print(f'Accuracy: {accuracy\*100}')

input\_data = np.array([4,110,76,20,100,28.4,0.118,27]).reshape(1, -1)

#4,110,76,20,100,28.4,0.118,27       nd

#5, 120, 92, 10, 81, 26.1, 0.551, 67 nd

#2, 81, 72, 15, 76, 30.1, 0.547, 25  nd

#1, 168, 88, 29, 0, 35, 0.905, 52    d

prediction = decision\_tree.predict(input\_data)

if prediction == 0:

    print('The person is not diabetic')

else:

    print('The person is diabetic')

**KNN-CLASSIFIER CODE:**

import pandas as pd

from sklearn.preprocessing import StandardScaler

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

from sklearn.metrics import accuracy\_score

import numpy as np

import statistics

diabetes\_data = pd.read\_csv('kaggle\_diabetes.csv')

features = diabetes\_data.drop(columns='Outcome', axis=1)

target = diabetes\_data['Outcome']

import numpy as np

import statistics

class KNN\_Classifier():

    def \_\_init\_\_(self):

        self.distance\_metric = distance\_metric

    def get\_distance\_metric(self, training\_data\_point, test\_data\_point):

        dist = 0

        for i in range(len(training\_data\_point) - 1):

            dist = dist + (training\_data\_point[i] - test\_data\_point[i]) \*\* 2

        euclidean\_dist = np.sqrt(dist)

        return euclidean\_dist

    def nearest\_neighbors(self, X\_train, test\_data, k):

        distance\_list = []

        for training\_data in X\_train:

            distance = self.get\_distance\_metric(training\_data, test\_data)

            distance\_list.append((training\_data, distance))

        distance\_list.sort(key=lambda x: x[1])

        neighbors\_list = []

        for j in range(k):

            neighbors\_list.append(distance\_list[j][0])

        return neighbors\_list

    def predict(self, X\_train, test\_data, k):

        neighbors = self.nearest\_neighbors(X\_train, test\_data, k)

        label = []

        for data in neighbors:

            label.append(data[-1])

        predicted\_class = statistics.mode(label)

        return predicted\_class

scaler = StandardScaler()

scaler.fit(features)

standardized\_data = scaler.transform(features)

features = standardized\_data

target = diabetes\_data['Outcome']

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(features, target, test\_size=0.2, random\_state = 2)

X\_train = np.insert(X\_train, 8, Y\_train, axis=1)

classifierr = KNN\_Classifier()

prediction = classifierr.predict(X\_train, X\_test[2], k=5)

input\_data = [1, 168, 88, 29, 0, 35, 0.905, 52]

input\_data\_as\_numpy\_array = np.asarray(input\_data)

input\_data\_reshaped = input\_data\_as\_numpy\_array.reshape(1, -1)

std\_data = scaler.transform(input\_data\_reshaped)

print(std\_data)

prediction = classifierr.predict(X\_train, std\_data[0], k=3)

print(prediction)

if prediction == 0:

    print('The person is not diabetic')

else:

    print('The person is diabetic')