BCT CODES

**C1. Write a program in solidity to create Student data. Use the following constructs:**

**• Structures**

**• Arrays**

**• Fallback**

**Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract StudentData {

struct Student {

string name;

uint age;

string course;

uint studentId;

}

Student[] public students;

mapping(uint => uint) public studentIdToIndex;

fallback() external payable {

revert("Fallback function called, no direct ether transfers allowed.");

}

function addStudent(string memory \_name, uint \_age, string memory \_course, uint \_studentId) public {

require(studentIdToIndex[\_studentId] == 0, "Student ID already exists!");

Student memory newStudent = Student({

name: \_name,

age: \_age,

course: \_course,

studentId: \_studentId

});

students.push(newStudent);

studentIdToIndex[\_studentId] = students.length;

}

function getStudentById(uint \_studentId) public view returns (string memory, uint, string memory, uint) {

uint index = studentIdToIndex[\_studentId];

require(index > 0, "Student ID does not exist!");

Student memory s = students[index - 1];

return (s.name, s.age, s.course, s.studentId);

}

function getTotalStudents() public view returns (uint) {

return students.length;

}

receive() external payable {

revert("Direct ether transfer not allowed.");

}

}

**C2. Write a program in solidity to create Employee data. Use the following constructs:**

**• Structures**

**• Arrays**

**• Fallback**

**Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract EmployeeData {

// Structure to hold employee information

struct Employee {

string name;

uint age;

string position;

uint employeeId;

}

// Array to store employees

Employee[] public employees;

// Mapping to track employee ID to index in the employees array

mapping(uint => uint) private employeeIdToIndex;

// Fallback function to prevent direct Ether transfers

fallback() external {

revert("Direct Ether transfers are not allowed!");

}

// Function to add a new employee

function addEmployee(string memory \_name, uint \_age, string memory \_position, uint \_employeeId) public {

require(employeeIdToIndex[\_employeeId] == 0, "Employee ID already exists!");

// Create a new employee and add it to the array

employees.push(Employee(\_name, \_age, \_position, \_employeeId));

// Map employee ID to the index in the array (index + 1 to avoid zero mapping)

employeeIdToIndex[\_employeeId] = employees.length;

}

// Function to retrieve an employee by ID

function getEmployeeById(uint \_employeeId) public view returns (string memory, uint, string memory) {

uint index = employeeIdToIndex[\_employeeId];

require(index > 0, "Employee ID does not exist!");

// Retrieve the employee information

Employee memory e = employees[index - 1]; // Adjust for zero indexing

return (e.name, e.age, e.position);

}

// Function to get the total number of employees

function getTotalEmployees() public view returns (uint) {

return employees.length;

}

}

**C3. Write a smart contract on a test network, for Bank account of a customer for following operations:**

1. **Deposit money 2) Withdraw Money 3) Show balance**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract BankAccount {

// Mapping to store the balances of each customer's account

mapping(address => uint256) private balances;

// Event to log deposits

event Deposited(address indexed customer, uint256 amount);

// Event to log withdrawals

event Withdrawn(address indexed customer, uint256 amount);

// Function to deposit money into the account

function deposit() public payable {

require(msg.value > 0, "Deposit amount must be greater than 0");

// Increase the customer's balance

balances[msg.sender] += msg.value;

// Emit a deposit event

emit Deposited(msg.sender, msg.value);

}

// Function to withdraw money from the account

function withdraw(uint256 amount) public {

require(amount > 0, "Withdrawal amount must be greater than 0");

require(balances[msg.sender] >= amount, "Insufficient balance");

// Decrease the customer's balance

balances[msg.sender] -= amount;

// Transfer the specified amount to the customer

payable(msg.sender).transfer(amount);

// Emit a withdrawal event

emit Withdrawn(msg.sender, amount);

}

// Function to show the balance of the account

function showBalance() public view returns (uint256) {

return balances[msg.sender];

}

}

**C4. Write a smart contract on a test network, for product inventory including following operations:**

1. **Receive Product 2) Sale product 3) Display Stock**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract ProductInventory {

// Structure to represent a product

struct Product {

uint256 productId;

string name;

uint256 price; // price in Wei (smallest unit of Ether)

uint256 quantity;

}

// Mapping to store products with their ID as key

mapping(uint256 => Product) public products;

// Counter for product IDs

uint256 public productCount = 0;

// Event to log when a product is received

event ProductReceived(uint256 productId, string name, uint256 quantity, uint256 price);

// Event to log when a product is sold

event ProductSold(uint256 productId, string name, uint256 quantity, uint256 totalPrice);

// Function to receive new products or add stock to an existing product

function receiveProduct(string memory \_name, uint256 \_price, uint256 \_quantity) public {

require(\_quantity > 0, "Quantity must be greater than zero");

require(\_price > 0, "Price must be greater than zero");

// Create new product or update existing product

productCount++;

products[productCount] = Product(productCount, \_name, \_price, \_quantity);

// Emit an event for product received

emit ProductReceived(productCount, \_name, \_quantity, \_price);

}

// Function to sell a product

function sellProduct(uint256 \_productId, uint256 \_quantity) public payable {

require(\_quantity > 0, "Quantity must be greater than zero");

Product storage product = products[\_productId];

require(product.quantity >= \_quantity, "Not enough stock available");

require(msg.value >= \_quantity \* product.price, "Insufficient payment");

// Deduct the quantity sold

product.quantity -= \_quantity;

// Emit an event for product sold

emit ProductSold(\_productId, product.name, \_quantity, msg.value);

}

// Function to display stock of a specific product

function displayStock(uint256 \_productId) public view returns (string memory name, uint256 price, uint256 quantity) {

Product memory product = products[\_productId];

return (product.name, product.price, product.quantity);

}

// Function to display all available products and their stock

function displayAllProducts() public view returns (Product[] memory) {

Product[] memory allProducts = new Product[](productCount);

for (uint256 i = 1; i <= productCount; i++) {

allProducts[i - 1] = products[i];

}

return allProducts;

}

}

**C5. Write a program in solidity to create Customer data. Use the following constructs:**

**• Structures**

**• Arrays**

**• Fallback**

**Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract CustomerData {

// Defining a structure to store customer information

struct Customer {

uint256 customerId;

string name;

uint256 age;

string email;

}

// Array to store multiple customers

Customer[] public customers;

// Mapping to keep track of the customer's existence by customerId

mapping(uint256 => bool) public customerExists;

// Event to emit when a new customer is added

event CustomerAdded(uint256 customerId, string name, uint256 age, string email);

// Function to add a new customer to the system

function addCustomer(uint256 \_customerId, string memory \_name, uint256 \_age, string memory \_email) public {

require(!customerExists[\_customerId], "Customer ID already exists!");

Customer memory newCustomer = Customer({

customerId: \_customerId,

name: \_name,

age: \_age,

email: \_email

});

customers.push(newCustomer);

customerExists[\_customerId] = true;

emit CustomerAdded(\_customerId, \_name, \_age, \_email);

}

// Function to retrieve customer details by index in the array

function getCustomer(uint256 index) public view returns (uint256, string memory, uint256, string memory) {

require(index < customers.length, "Invalid index!");

Customer memory customer = customers[index];

return (customer.customerId, customer.name, customer.age, customer.email);

}

// Fallback function to handle any Ether sent to the contract

fallback() external payable {

// You can add logic here, e.g., reverting transactions if not meant to send Ether

}

// Function to check contract balance

function getContractBalance() public view returns (uint256) {

return address(this).balance;

}

}

**DAA 2**

#include <iostream>

#include <vector>

#include <map>

#include <ctime>

#include <algorithm>

#include <cmath>

using namespace std;

class Node {

public:

char symbol;

int freq;

Node\* left;

Node\* right;

Node(char symbol, int freq, Node\* left = nullptr, Node\* right = nullptr)

: symbol(symbol), freq(freq), left(left), right(right) {}

};

// Helper function to recursively calculate Huffman codes

void calculateHuffmanCodes(Node\* node, const string& code, map<char, string>& huffmanCodes) {

if (node) {

// If it's a leaf node, add code to the map

if (!node->left && !node->right) {

huffmanCodes[node->symbol] = code;

}

// Recur for left and right children

calculateHuffmanCodes(node->left, code + "0", huffmanCodes);

calculateHuffmanCodes(node->right, code + "1", huffmanCodes);

}

}

int main() {

int n;

cout << "Enter the number of characters: ";

cin >> n;

vector<char> chars(n);

vector<int> freq(n);

cout << "Enter characters and their frequencies:\n";

for (int i = 0; i < n; ++i) {

cout << "Character " << i + 1 << ": ";

cin >> chars[i];

cout << "Frequency of " << chars[i] << ": ";

cin >> freq[i];

}

// Start measuring time for Huffman tree construction

clock\_t start\_time = clock();

// Initialize nodes list with individual character nodes

vector<Node\*> nodes;

for (int i = 0; i < n; ++i) {

nodes.push\_back(new Node(chars[i], freq[i]));

}

// Build the Huffman Tree by repeatedly combining the two lowest-frequency nodes

while (nodes.size() > 1) {

// Sort nodes by frequency in ascending order

sort(nodes.begin(), nodes.end(), [](Node\* a, Node\* b) {

return a->freq < b->freq;

});

// Take the two nodes with the lowest frequencies

Node\* left = nodes[0];

Node\* right = nodes[1];

// Create a new node combining these two

Node\* newNode = new Node('\0', left->freq + right->freq, left, right);

// Remove the two nodes and add the new node to the list

nodes.erase(nodes.begin());

nodes.erase(nodes.begin());

nodes.push\_back(newNode);

}

// End time for Huffman tree construction

clock\_t end\_time = clock();

double tree\_duration = double(end\_time - start\_time) / CLOCKS\_PER\_SEC;

// Start measuring time for Huffman code calculation

clock\_t code\_start\_time = clock();

map<char, string> huffmanCodes;

calculateHuffmanCodes(nodes[0], "", huffmanCodes); // Root of the Huffman tree is in nodes[0]

clock\_t code\_end\_time = clock();

double code\_duration = double(code\_end\_time - code\_start\_time) / CLOCKS\_PER\_SEC;

// Calculate estimated space in bytes for storing Huffman codes

double spaceUsed = 0;

for (const auto& kv : huffmanCodes) {

int index = find(chars.begin(), chars.end(), kv.first) - chars.begin();

spaceUsed += kv.second.length() \* freq[index];

}

spaceUsed = ceil(spaceUsed / 8);

// Output results

cout << "Huffman Tree Construction Time: " << tree\_duration << " seconds" << endl;

cout << "Huffman Code Calculation Time: " << code\_duration << " seconds" << endl;

cout << "Estimated Space Used for Huffman Codes: " << spaceUsed << " bytes" << endl;

cout << "Huffman Codes:\n";

for (const auto& kv : huffmanCodes) {

cout << kv.first << " -> " << kv.second << endl;

}

return 0;

}

**DAA 1**

#include <iostream>

#include <ctime> // For measuring time

using namespace std;

// Iterative Fibonacci

int fib\_iterative(int n) {

if (n <= 1) return n;

int a = 0, b = 1, c;

for (int i = 2; i <= n; i++) {

c = a + b;

a = b;

b = c;

}

return b;

}

// Recursive Fibonacci

int fib\_recursive(int n) {

if (n <= 1) return n;

return fib\_recursive(n - 1) + fib\_recursive(n - 2);

}

int main() {

int n;

cout << "Enter a number: ";

cin >> n;

// Measure time for iterative Fibonacci

clock\_t start\_iter = clock();

int result\_iter = fib\_iterative(n);

clock\_t end\_iter = clock();

double duration\_iter = double(end\_iter - start\_iter) / CLOCKS\_PER\_SEC;

// Calculate space required for iterative approach

int space\_iter = sizeof(n) + sizeof(int) \* 3; // n, a, b, c variables

cout << "Iterative Fibonacci of " << n << " is: " << result\_iter << endl;

cout << "Time required (iterative): " << duration\_iter << " seconds" << endl;

cout << "Space required (iterative): " << space\_iter << " bytes" << endl;

// Measure time for recursive Fibonacci

clock\_t start\_rec = clock();

int result\_rec = fib\_recursive(n);

clock\_t end\_rec = clock();

double duration\_rec = double(end\_rec - start\_rec) / CLOCKS\_PER\_SEC;

// Calculate space required for recursive approach

// Recursion uses a stack frame for each call, with `sizeof(int)` for each level

int space\_rec = sizeof(n) \* (n > 1 ? n : 1);

cout << "Recursive Fibonacci of " << n << " is: " << result\_rec << endl;

cout << "Time required (recursive): " << duration\_rec << " seconds" << endl;

cout << "Space required (recursive): " << space\_rec << " bytes" << endl;

return 0;

}

**DAA 3**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent an item with weight and value

struct Item {

int value, weight;

// Constructor

Item(int v, int w) : value(v), weight(w) {}

};

// Function to compare items based on value/weight ratio

bool compare(Item a, Item b) {

double r1 = (double)a.value / a.weight;

double r2 = (double)b.value / b.weight;

return r1 > r2; // Sort in descending order of value/weight ratio

}

// Function to solve the fractional knapsack problem

double fractionalKnapsack(int capacity, vector<Item>& items) {

// Sort items by their value/weight ratio

sort(items.begin(), items.end(), compare);

double totalValue = 0.0; // Variable to store the total value we can carry

for (auto& item : items) {

if (capacity >= item.weight) {

// If the item can be completely taken, take it

capacity -= item.weight;

totalValue += item.value;

} else {

// If the item cannot be completely taken, take a fraction of it

totalValue += item.value \* ((double)capacity / item.weight);

break; // Knapsack is full

}

}

return totalValue;

}

int main() {

int n; // Number of items

int capacity; // Knapsack capacity

// User input

cout << "Enter the number of items: ";

cin >> n;

vector<Item> items;

int value, weight;

// Input values and weights of items

for (int i = 0; i < n; i++) {

cout << "Enter value and weight of item " << i + 1 << ": ";

cin >> value >> weight;

items.push\_back(Item(value, weight));

}

cout << "Enter the capacity of the knapsack: ";

cin >> capacity;

// Solve the fractional knapsack problem

double maxValue = fractionalKnapsack(capacity, items);

// Print the result

cout << "Maximum value in knapsack = " << maxValue << endl;

return 0;

}

**DAA 5**

#include <iostream>

#include <vector>

#include <ctime>

using namespace std;

bool isSafe(const vector<int>& board, int row, int col) {

// Check previous rows for conflicts with the new queen at (row, col)

for (int i = 0; i < row; i++) {

int placedCol = board[i];

// Check column conflict and both diagonals

if (placedCol == col || abs(placedCol - col) == abs(i - row))

return false;

}

return true;

}

bool solveNQueensUtil(vector<int>& board, int row, in3t n) {

if (row == n) return true; // All queens are placed successfully

for (int col = 0; col < n; col++) {

if (isSafe(board, row, col)) {

board[row] = col; // Place queen at (row, col)

if (solveNQueensUtil(board, row + 1, n))

return true; // Recursive call

board[row] = -1; // Backtrack

}

}

return false;

}

void printSolution(const vector<int>& board, int n) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++)

cout << (board[i] == j ? "Q " : ". ");

cout << endl;

}

}

int main() {

int n;

cout << "Enter the value of N: ";

cin >> n;

if (n < 1) {

cout << "Invalid board size." << endl;

return 1;

}

vector<int> board(n, -1); // Initialize board with -1 to indicate no queens placed

clock\_t start = clock();

if (solveNQueensUtil(board, 0, n)) {

printSolution(board, n);

} else {

cout << "No solution exists for N = " << n << endl;

}

clock\_t end = clock();

double time\_taken = double(end - start) / CLOCKS\_PER\_SEC;

int memory\_usage = sizeof(int) \* (n + n); // Rough memory estimation

cout << "Execution time: " << time\_taken << " seconds" << endl;

cout << "Memory used: " << memory\_usage << " bytes" << endl;

return 0;

}

**ML 1**

import pandas as pd

df = pd.read\_csv('uber.csv')

df

print(df.isnull().sum())

# # Drop rows with missing values

df = df.dropna()

import seaborn as sns

import matplotlib.pyplot as plt

sns.boxplot(x=df['fare\_amount'])

plt.show()

sns.boxplot(x=df['passenger\_count'])

plt.show()

Q1 = df['fare\_amount'].quantile(0.25)

Q3 = df['fare\_amount'].quantile(0.75)

IQR = Q3 - Q1

df = df[~((df['fare\_amount'] < (Q1 - 1.5 \* IQR)) | (df['fare\_amount'] > (Q3 + 1.5 \* IQR)))]

sns.boxplot(x=df['fare\_amount'])

plt.show()

import numpy as np

def haversine(lon1, lat1, lon2, lat2):

R = 6371 # Radius of the Earth in km

lon1, lon2 = np.radians(lon1), np.radians(lon2)

lat1, lat2 = np.radians(lat1), np.radians(lat2)

dlon = lon2 - lon1

dlat = lat2 - lat1

a = np.sin(dlat/2) \*\* 2 + np.cos(lat1) \* np.cos(lat2) \* np.sin(dlon/2) \*\* 2

c = 2 \* np.arctan2(np.sqrt(a), np.sqrt(1 - a))

return R \* c

df['distance'] = haversine(df['pickup\_longitude'], df['pickup\_latitude'],

df['dropoff\_longitude'], df['dropoff\_latitude'])

df

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

# Features

X = df[['distance', 'passenger\_count']]

# Target

y = df['fare\_amount']

# Split the data

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

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

import matplotlib.pyplot as plt

import seaborn as sns

# Initialize the model

lr = LinearRegression()

# Train the model

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

# Predict

y\_pred\_lr = lr.predict(X\_test)

# Evaluate the model

rmse\_lr = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_lr))

r2\_lr = r2\_score(y\_test, y\_pred\_lr)

print(f"Linear Regression - RMSE: {rmse\_lr}, R2: {r2\_lr}")

# 3. Check the correlation

# Exclude the 'pickup\_datetime' column from correlation calculation

correlation\_matrix = df.select\_dtypes(include=np.number).corr() # Select only numeric columns for correlation

sns.heatmap(correlation\_matrix, annot=True)

plt.show()

from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(n\_estimators=100, random\_state=42)

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

y\_pred\_rf = rf.predict(X\_test)

# Evaluation of model

rmse\_rf = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_rf))

r2\_rf = r2\_score(y\_test, y\_pred\_rf)

print(f"Random Forest Regression - RMSE: {rmse\_rf}, R2: {r2\_rf}")

print(f"Linear Regression - RMSE: {rmse\_lr}, R2: {r2\_lr}")

print(f"Random Forest Regression - RMSE: {rmse\_rf}, R2: {r2\_rf}")

**ML2**

import pandas as pd

import numpy as np

import seaborn as sns

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, confusion\_matrix, classification\_report

from sklearn.metrics import accuracy\_score, r2\_score

from sklearn.preprocessing import StandardScaler

df = pd.read\_csv('emails.csv')

df

X = df.iloc[:, 1:-1]  # 1st column to 2999th column: the word counts

y = df.iloc[:, -1]    # Last column: the labels (spam or not-spam)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Standardize

scaler = StandardScaler()

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

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

# K-Nearest Neighbors Classifier

knn = KNeighborsClassifier(n\_neighbors=5)

knn.fit(X\_train\_scaled, y\_train)

y\_pred\_knn = knn.predict(X\_test\_scaled)

# Support Vector Machine Classifier

svm = SVC(kernel='linear')  # You can experiment with different kernels

svm.fit(X\_train\_scaled, y\_train)

y\_pred\_svm = svm.predict(X\_test\_scaled)

 Metrics Calculation

def calculate\_metrics(y\_true, y\_pred):

    mse = mean\_squared\_error(y\_true, y\_pred)

    rmse = np.sqrt(mse)

    mae = mean\_absolute\_error(y\_true, y\_pred)

    r2 = r2\_score(y\_true, y\_pred)

    print(f"MSE: {mse}")

    print(f"RMSE: {rmse}")

    print(f"MAE: {mae}")

    print(f"R²: {r2}")

    return mse, rmse, mae, r2

# Evaluate KNN

print("K-Nearest Neighbors Performance:")

mse\_knn, rmse\_knn, mae\_knn, r2\_knn = calculate\_metrics(y\_test, y\_pred\_knn)

print("Confusion Matrix (KNN):\n", confusion\_matrix(y\_test, y\_pred\_knn))

print("Classification Report (KNN):\n", classification\_report(y\_test, y\_pred\_knn))

# Evaluate SVM

print("\nSupport Vector Machine Performance:")

mse\_svm, rmse\_svm, mae\_svm, r2\_svm = calculate\_metrics(y\_test, y\_pred\_svm)

print("Confusion Matrix (SVM):\n", confusion\_matrix(y\_test, y\_pred\_svm))

print("Classification Report (SVM):\n", classification\_report(y\_test, y\_pred\_svm))

sns.heatmap(confusion\_matrix(y\_test, y\_pred\_svm),annot= True)

sns.heatmap(confusion\_matrix(y\_test, y\_pred\_knn),annot= True)

**ML3**

**import** pandas **as** pd

**import** numpy **as** np

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

**from** sklearn.preprocessing **import** StandardScaler, LabelEncoder

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

**from** tensorflow.keras.models **import** Sequential

**from** tensorflow.keras.layers **import** Dense

**import** matplotlib.pyplot **as** plt

In [ ]:

**!**pip install seaborn

**import** seaborn **as** sns

In [ ]:

df **=** pd**.**read\_csv('Churn\_Modelling.csv')

In [ ]:

df

*#step 2*

X **=** df**.**drop(['RowNumber', 'CustomerId', 'Surname', 'Exited'], axis**=**1) *# Remove unnecessary columns*

y **=** df['Exited']

In [ ]:

X

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

In [ ]:

scaler **=** StandardScaler()

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

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

In [ ]:

model **=** Sequential()

In [ ]:

model**.**add(Dense(64, input\_dim**=**X\_train**.**shape[1], activation**=**'relu')) *# Using ReLU activation*

*# Hidden Layer*

model**.**add(Dense(32, activation**=**'relu')) *# You can experiment with other activations here (sigmoid, tanh, etc.)*

*# Output Layer*

model**.**add(Dense(1, activation**=**'sigmoid')) *# Sigmoid for binary classification*

*# Compile the model*

model**.**compile(optimizer**=**'adam', loss**=**'binary\_crossentropy', metrics**=**['accuracy'])

*# Train the model*

history **=** model**.**fit(X\_train, y\_train, epochs**=**30, batch\_size**=**32, validation\_data**=**(X\_test, y\_test))

*# Step 5: Evaluate the model and print accuracy score and confusion matrix*

y\_pred **=** (model**.**predict(X\_test) **>** 0.5)**.**astype(int)

le\_gender **=** LabelEncoder()

X['Gender'] **=** le\_gender**.**fit\_transform(X['Gender'])

le\_geo **=** LabelEncoder()

X['Geography'] **=** le\_geo**.**fit\_transform(X['Geography'])

In [ ]:

X

*# Accuracy score*

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

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

*# Confusion Matrix*

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

print("Confusion Matrix:")

print(cm)

model\_tanh **=** Sequential()

model\_tanh**.**add(Dense(64, input\_dim**=**X\_train**.**shape[1], activation**=**'tanh'))

model\_tanh**.**add(Dense(32, activation**=**'tanh'))

model\_tanh**.**add(Dense(1, activation**=**'sigmoid'))

model\_tanh**.**compile(optimizer**=**'adam', loss**=**'binary\_crossentropy', metrics**=**['accuracy'])

model\_tanh**.**fit(X\_train, y\_train, epochs**=**20, batch\_size**=**32, validation\_data**=**(X\_test, y\_test))

*# Evaluating model with 'tanh'*

y\_pred\_tanh **=** (model\_tanh**.**predict(X\_test) **>** 0.5)**.**astype(int)

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

print(f"Accuracy with Tanh activation: {accuracy\_tanh **\*** 100:.2f}%")

model\_sigmoid **=** Sequential()

model\_sigmoid**.**add(Dense(64, input\_dim**=**X\_train**.**shape[1], activation**=**'sigmoid'))

model\_sigmoid**.**add(Dense(32, activation**=**'sigmoid'))

model\_sigmoid**.**add(Dense(1, activation**=**'sigmoid'))

model\_sigmoid**.**compile(optimizer**=**'adam', loss**=**'binary\_crossentropy', metrics**=**['accuracy'])

history\_sigmoid **=** model\_sigmoid**.**fit(X\_train\_scaled, y\_train, epochs**=**20, batch\_size**=**32, validation\_data**=**(X\_test\_scaled, y\_test))

y\_pred\_sigmoid **=** (model\_sigmoid**.**predict(X\_test\_scaled) **>** 0.5)**.**astype(int)

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

print(f"Accuracy with Sigmoid in all layers: {accuracy\_sigmoid **\*** 100:.2f}%")

print("Confusion Matrix with Sigmoid in all layers:")

print(confusion\_matrix(y\_test, y\_pred\_sigmoid))

**ML5**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

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

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import accuracy\_score

from sklearn.metrics import classification\_report

from sklearn.metrics import precision\_score

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import recall\_score

df=pd.read\_csv("C:\\Users\\admin\\Downloads\\archive (1)\\diabetes.csv")

df

df.describe()

df.head(5)

df.tail(5)

df.isnull().sum()

df.isna().sum()

df\_x=df

df\_y=df['Outcome']

df\_y

def RemoveOutliers(df,var):

Q1 = df[var].quantile(0.25)

Q3 = df[var].quantile(0.75)

IQR = Q3 - Q1

high, low = Q3+1.5\*IQR, Q1-1.5\*IQR

print("Highest allowed in variable:", var, high)

print("lowest allowed in variable:", var, low)

count = df[(df[var] > high) | (df[var] < low)][var].count()

print('Total outliers in:',var,':',count)

df = df[((df[var] >= low) & (df[var] <= high))]

return df

fig,axes=plt.subplots(1,3)

sns.boxplot(data=df,x='Insulin',ax=axes[0])

sns.boxplot(data=df,x='Pedigree',ax=axes[1])

sns.boxplot(data=df,x='Glucose',ax=axes[2])

plt.show()

df=RemoveOutliers(df,'Insulin')

df=RemoveOutliers(df,'Pedigree')

fig,axes=plt.subplots(1,3)

sns.boxplot(data=df,x='Insulin',ax=axes[0])

sns.boxplot(data=df,x='Pedigree',ax=axes[1])

sns.boxplot(data=df,x='Glucose',ax=axes[2])

plt.show()

xtrain,xtest,ytrain,ytest=train\_test\_split(df\_x,df\_y,test\_size=0.20,random\_state=13)

from sklearn.preprocessing import StandardScaler

sc=StandardScaler()

xtrain=sc.fit\_transform(xtrain)

xtest=sc.transform(xtest)

knn = KNeighborsClassifier(n\_neighbors=5)

knn.fit(xtrain,ytrain)

ypred=knn.predict(xtest)

cm=confusion\_matrix(ytest,ypred)

sns.heatmap(cm,annot=True)

plt.show()

print(classification\_report(ytest,ypred))

accuracy = accuracy\_score(ytest, ypred)

precision = precision\_score(ytest, ypred)

recall = recall\_score(ytest, ypred)

error\_rate = 1 - accuracy

print(f"Accuracy: {accuracy \* 100:.2f}%")

print(f"Error Rate: {error\_rate \* 100:.2f}%")

print(f"Precision: {precision \* 100:.2f}%")

print(f"Recall: {recall \* 100:.2f}%")

#Loop through different values of K to check the impact of k on accuracy

print("\nEvaluating different values of k for better performance:")

for k in range(1, 11):

knn = KNeighborsClassifier(n\_neighbors=k)

knn.fit(xtrain, ytrain)

ypred = knn.predict(xtest)

accuracy = accuracy\_score(ytest, ypred)

print(f"K = {k}, Accuracy = {accuracy \* 100:.2f}%")

**ML 6**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

df = pd.read\_csv('sales\_data\_sample.csv', encoding='latin')

df

df.describe()

df.head()

df.tail()

#Relevant numeric features for clustering

df\_cluster = df[['SALES', 'QUANTITYORDERED', 'PRICEEACH']]

scaler = StandardScaler()

df\_scaled = scaler.fit\_transform(df\_cluster)

#Finding optimal number of clusters

wcss = []

k\_values = range(1, 11)

for k in k\_values:

kmeans = KMeans(n\_clusters=k, init='k-means++', random\_state=42)

kmeans.fit(df\_scaled)

wcss.append(kmeans.inertia\_)

plt.figure(figsize=(8, 5))

plt.plot(k\_values, wcss, marker='o', linestyle='--')

plt.title('Elbow Method to Determine Optimal Number of Clusters')

plt.xlabel('Number of Clusters (K)')

plt.ylabel('WCSS')

plt.show()

optimal\_k = 3

kmeans = KMeans(n\_clusters=optimal\_k, init='k-means++', random\_state=42)

y\_kmeans = kmeans.fit\_predict(df\_scaled)

df['Cluster'] = y\_kmeans

print(df[['SALES', 'QUANTITYORDERED', 'PRICEEACH', 'Cluster']].head())

plt.figure(figsize=(8, 5))

plt.scatter(df\_scaled[:, 0], df\_scaled[:, 1], c=y\_kmeans, cmap='viridis')

plt.title(f'K-Means Clustering with {optimal\_k} Clusters')

plt.xlabel('Feature 1: SALES')

plt.ylabel('Feature 2: QUANTITYORDERED')

plt.show()