LP 3 Practicals

DAA

Practical 1:- Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

```
def recur(n):
   if n <= 1:
      return n
   else:
      return(recur(n-1) + recur(n-2))

def iterative(n):
   a = 0
   b = 1
   print(a)
   print(b)
   for i in range(2, n):
      print(a +  b)
      a, b = b, a + b

if __name___== "__main__":
   num = int(input("Enter the nth number for series: "))
   if num <= 0:
      print("Please enter a positive integer")
   else:
      print("Fibonacci sequence with recursion:")
      for i in range(num):
         print(recur(i))
   print("Fibonacci series with Iteration:")
   iterative(num)
```

Practical 2:- Write a program to solve a fractional Knapsack problem using a greedy method.

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
def fractional_knapsack(capacity, items):
    items.sort(key=lambda item: item.value/item.weight, reverse=True)

    total_value = 0
    for item in items:
        if capacity >= item.weight:
            capacity -= item.weight
            total_value += item.value
        else:
            fraction = capacity / item.weight
            total_value += item.value * fraction
            break

    return total_value

if __name__ == "__main__":
    n = int(input("Enter the number of items: "))
    items = []

    for i in range(n):
        value = float(input(f"Enter value of item {i+1}: "))
        weight = float(input(f"Enter weight of item {i+1}: "))
        items.append(Item(value, weight))

    capacity = float(input("Enter the capacity of the knapsack: "))
    max_value = fractional_knapsack(capacity, items)
    print(f"Maximum value we can obtain = {max_value}")
```

Practical 3:- Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

1. Knapsack problem using dynamic

```
def knapsack_01(weights, values, capacity):
    n = len(values)
    dp = [[0 for x in range(capacity + 1)] for y in range(n + 1)]
    for i in range(n + 1):
        for w in range(capacity + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i - 1] <= w:
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]

if __name___== "__main__":
    n = int(input("Enter the number of items: "))
    values = []
    weights = []
    for i in range(n):
        value = int(input(f"Enter value of item {i+1}: "))
        weight = int(input(f"Enter weight of item {i+1}: "))
        values.append(value)
        weights.append(weight)
    capacity = int(input("Enter the capacity of the knapsack: "))
    max_value = knapsack_01(weights, values, capacity)
    print(f"Maximum value we can obtain = {max_value}")
```

2. Branch and bound strategy

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
        self.ratio = value / weight

class Node:
    def __init_(self, level, value, weight, bound):
        self.level = level
        self.value = value
        self.weight = weight
        self.bound = bound

def bound(node, capacity, n, items):
    if node.weight >= capacity:
        return 0
    profit_bound = node.value
    j = node.level + 1
    total_weight = node.weight
    while j < n and total_weight + items[j].weight <= capacity:
        total_weight += items[j].weight
```

```
            profit_bound += items[j].value
            j += 1
        if j < n:
            profit_bound += (capacity - total_weight) * items[j].ratio
        return profit_bound

def knapsack_branch_and_bound(capacity, items):
    items.sort(key=lambda x: x.ratio, reverse=True)
    n = len(items)
    Q = []
    u = Node(-1, 0, 0, 0)
    v = Node(-1, 0, 0, 0)
    u.bound = bound(u, capacity, n, items)
    Q.append(u)
    max_profit = 0
    while Q:
        u = Q.pop(0)
        if u.level == -1:
            v.level = 0
        if u.level == n - 1:
            continue
        v.level = u.level + 1
        v.weight = u.weight + items[v.level].weight
        v.value = u.value + items[v.level].value
        if v.weight <= capacity and v.value > max_profit:
            max_profit = v.value
        v.bound = bound(v, capacity, n, items)
        if v.bound > max_profit:
            Q.append(v)
        v.weight = u.weight
        v.value = u.value
        v.bound = bound(v, capacity, n, items)
        if v.bound > max_profit:
            Q.append(v)
    return max_profit

if __name__ == "__main__":
    n = int(input("Enter the number of items: "))
    items = []
    for i in range(n):
        value = int(input(f"Enter value of item {i+1}: "))
        weight = int(input(f"Enter weight of item {i+1}: "))
        items.append(Item(value, weight))
    capacity = int(input("Enter the capacity of the knapsack: "))
    max_value = knapsack_branch_and_bound(capacity, items)
    print(f"Maximum value we can obtain = {max_value}")
```

Practical 4:- Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix

```
def is_safe(board, row, col, n):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solve_n_queens(board, col, n):
    if col >= n:
        return True
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
            if solve_n_queens(board, col + 1, n):
                return True
            board[i][col] = 0
    return False

def n_queens(n, start_row, start_col):
    board = [[0 for _ in range(n)] for _ in range(n)]
    board[start_row][start_col] = 1
    if solve_n_queens(board, 1, n): # Start from the next column
        return board
    else:
        return None

def print_board(board):
    if board:
        for row in board:
            print(" ".join("Q" if x == 1 else "." for x in row))
    else:
        print("No solution found")

n = int(input("Enter the value of n (size of board): "))
start_row = 0
start_col = 0
final_board = n_queens(n, start_row, start_col)
print_board(final_board)
```

Practical 5:- Write a program for analysis of quick sort by using deterministic and randomized variant

```
import time
def quick_sort_deterministic(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort_deterministic(arr, low, pi - 1)
        quick_sort_deterministic(arr, pi + 1, high)

def quick_sort_randomized(arr, low, high):
    if low < high:
        random_index = low + (hash(str(arr)) % (high - low + 1)) # Simple random index generation
        arr[low], arr[random_index] = arr[random_index], arr[low]
        pi = partition(arr, low, high)
        quick_sort_randomized(arr, low, pi - 1)
        quick_sort_randomized(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def analyze_sorting_algorithm(sort_function, arr):
    start_time = time.time()
    sort_function(arr, 0, len(arr) - 1)
    end_time = time.time()
    return end_time - start_time

def generate_random_array(size):
    return [i for i in range(size, 0, -1)]  # Simple reverse sorted array for worst-case

if __name__ == "__main__":
    array_size = int(input("Enter the size of the array: "))
    arr = generate_random_array(array_size)
    arr_deterministic = arr.copy()
    arr_randomized = arr.copy()
    print("Analyzing deterministic Quick Sort...")
    deterministic_time = analyze_sorting_algorithm(quick_sort_deterministic, arr_deterministic)
    print("Analyzing randomized Quick Sort...")
    randomized_time = analyze_sorting_algorithm(quick_sort_randomized, arr_randomized)
    print(f"Deterministic Quick Sort Time: {deterministic_time:.6f} seconds")
    print(f"Randomized Quick Sort Time: {randomized_time:.6f} seconds")
```

ML

Practical 1:- Predict the price of the Uber ride from a given pickup point to the agreed drop-off location. Perform following tasks:

1. Preprocess the dataset.

2. Identify outliers.

3. Check the correlation.

4. Implement linear regression and random forest regression models.

5. Evaluate the models and compare their respective scores like R2, RMSE, etc.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

url = "uber.csv"
df = pd.read_csv(url)
print(df.head())
```

1. Data Preprocessing

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

df.dropna(inplace=True)
df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'])
df['hour'] = df['pickup_datetime'].dt.hour
df['day_of_week'] = df['pickup_datetime'].dt.dayofweekdf = df[['passenger_count',
'pickup_longitude', 'pickup_latitude','dropoff_longitude', 'dropoff_latitude', 'hour', 'fare_amount']]
```

2. Identify Outliers

```
z_scores = np.abs(stats.zscore(df['fare_amount']))
df.loc[:, 'outlier'] = z_scores > 3
outliers = df[df['outlier']]
print("Number of outliers detected:", len(outliers))
```

3. Check Correlation

```
correlation_matrix = df.corr()
print(correlation_matrix)

plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Matrix')
plt.show()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

4. Implement Random Forest Regression

```
linear_model = LinearRegression()
```

```
linear_model.fit(X_train, y_train)
y_pred_linear = linear_model.predict(X_test)
```

5. Evaluate Models

```
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
```

6. Evaluate Models

```
r2_linear = r2_score(y_test, y_pred_linear)
rmse_linear = np.sqrt(mean_squared_error(y_test, y_pred_linear))
r2_rf = r2_score(y_test, y_pred_rf)
rmse_rf  = np.sqrt(mean_squared_error(y_test, y_pred_rf))
print(f'Linear Regression: R² = {r2_linear:.4f}, RMSE = {rmse_linear:.4f}')
print(f'Random Forest Regression: R² = {r2_rf:.4f}, RMSE = {rmse_rf:.4f}')
```

Practical 2:- Classify the email using the binary classification method. Email Spam detection has two states:
a) Normal State – Not Spam
b) Abnormal State – Spam. Use K-Nearest Neighbors and Support Vector Machine for classification. Analyze their performance.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

Step 1: Load the dataset
url = "emails.csv"
df = pd.read_csv('emails.csv', encoding='latin-1')
print(df.head())

Step 2: Data Preprocessing
X = df.drop(columns=['Email No.', 'Prediction'])
y = df['Prediction']
X = X.apply(pd.to_numeric, errors='coerce')
X = X.dropna()
y = y[X.index]

Step 3: Split 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)

Step 4: K-Nearest Neighbors Model
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train, y_train)
y_pred_knn = knn_model.predict(X_test)

Step 5: Support Vector Machine Model
svm_model = SVC(kernel='linear')
svm_model.fit(X_train, y_train)
y_pred_svm = svm_model.predict(X_test)

Step 6: Evaluate Models
print("K-Nearest Neighbors Classification Report:")
print(classification_report(y_test, y_pred_knn, target_names=['Normal State - Not Spam (0)',
'Abnormal State - Spam (1)']))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_knn))
print("Accuracy:", accuracy_score(y_test, y_pred_knn))
print("\nSupport Vector Machine Classification Report:")
print(classification_report(y_test, y_pred_svm, target_names=['Normal State - Not Spam (0)',
'Abnormal State - Spam (1)']))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_svm))
print("Accuracy:", accuracy_score(y_test, y_pred_svm))
```

Practical 3:- Given a bank customer, build a neural network-based classifier that can determine whether they will leave or not in the next 6 months

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

# Step 1: Read the dataset
url = 'Churn_Modelling.csv'
df = pd.read_csv(url)

# Step 2: Distinguish feature and target set
X = df.drop(columns=['RowNumber', 'CustomerId', 'Surname', 'Exited'])
y = df['Exited']

X = pd.get_dummies(X, drop_first=True)

# Step 3: Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 4: Initialize and build the model
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(X_train.shape[1],))) # First hidden layer
model.add(Dense(16, activation='relu'))  # Second hidden layer
model.add(Dense(1, activation='sigmoid')) # Output layer for binary classification
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2)

# Step 5: Evaluate the model
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
print(f'Accuracy: {accuracy:.4f}')
print('Confusion Matrix:')
print(conf_matrix)
```

Practical 4:- Implement Gradient Descent Algorithm to find the local minima of a function. For example, find the local minima of the function $y=(x+3)^2$ starting from the point x=2.

```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Define the function and its derivative
def function(x):
    return (x + 3)**2
def derivative(x):
    return 2 * (x + 3)

# Step 2: Implement the Gradient Descent Algorithm
def gradient_descent(starting_point, learning_rate, num_iterations):
    x = starting_point
    x_history = [x]
    for _ in range(num_iterations):
        x -= learning_rate * derivative(x)
        x_history.append(x)
    return x, x_history

# Step 3: Parameters for Gradient Descent
starting_point = 2
learning_rate = 0.1
num_iterations = 50
local_minima, history = gradient_descent(starting_point, learning_rate, num_iterations)
print(f'Local minima found at x = {local_minima:.4f}, y = {function(local_minima):.4f}')

# Step 4: Visualizing the results
x_vals = np.linspace(-6, 0, 100)
y_vals = function(x_vals)
plt.plot(x_vals, y_vals, label='y = (x + 3)²')
plt.scatter(history, function(np.array(history)), color='red', label='Gradient Descent Steps')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Gradient Descent to Find Local Minima')
plt.axhline(0, color='black', linewidth=0.5, ls='--')
plt.axvline(0, color='black', linewidth=0.5, ls='--')
plt.legend()
plt.grid()
plt.show()
```

Practical 5:- Implement K-Means clustering/ hierarchical clustering on sales_data_sample.csv dataset. Determine the number of clusters using the elbow method.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as  plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import dendrogram, linkage
import seaborn as sns

# Step 1: Load the dataset
url = "sales_data_sample.csv"
df = pd.read_csv(url, encoding='ISO-8859-1')

# Step 2: Preprocess the data
features = df[['QUANTITYORDERED', 'PRICEEACH']].copy()
features.dropna(inplace=True)
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)

# Step 3: Determine the number of clusters using the elbow method
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit(features_scaled)
    wcss.append(kmeans.inertia_)
plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), wcss, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of clusters (k)')
plt.ylabel('WCSS')
plt.xticks(range(1, 11))
plt.grid()
plt.show()

# Step 4: Apply K-Means clustering
optimal_k = 3
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
df['Cluster'] = kmeans.fit_predict(features_scaled)
plt.figure(figsize=(10, 6))
plt.scatter(df['QUANTITYORDERED'], df['PRICEEACH'], c=df['Cluster'], cmap='viridis')
plt.title('K-Means Clustering Results')
plt.xlabel('QUANTITYORDERED')
plt.ylabel('PRICEEACH')
plt.grid()
plt.colorbar(label='Cluster')
plt.show()

# Step 5: (Optional) Hierarchical Clustering
linkage_matrix = linkage(features_scaled, method='ward')
plt.figure(figsize=(12, 8))
```

```
dendrogram(linkage_matrix)
plt.title('Dendrogram for Hierarchical Clustering')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.grid()
plt.show()
```

BT

Practical 1:- Write a smart contract on a test network, for Bank account of a customer for following operations:

- **Deposit money**
- **Withdraw Money**
- **Show balance**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract BankAccount {
   mapping(address => uint256) private balances;
   event Deposit(address indexed account, uint256 amountInEther);
   event Withdraw(address indexed account, uint256 amountInEther);
   function deposit() public payable {
      uint256 amountInEther = msg.value / 1 ether;
      require(amountInEther > 0, "Deposit amount should be greater than 0");
      balances[msg.sender] += amountInEther;
      emit Deposit(msg.sender, amountInEther);
   }
   function withdraw(uint256 amountInEther) public {
      require(amountInEther <= balances[msg.sender], "Insufficient balance");
      balances[msg.sender] -= amountInEther;
      uint256 amountInWei = amountInEther * 1 ether;
      payable(msg.sender).transfer(amountInWei);

      emit Withdraw(msg.sender, amountInEther);
   }
   function showBalance() public view returns (uint256) {
      return balances[msg.sender];
   }
}
```

Practical 2:- 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
//https://betterprogramming.pub/developing-a-smart-contract-by-using-re mix-ide-81ff6f44ba2f
pragma solidity ^0.5.0;
contract Crud {
    struct User {
        uint id;
        string name;
    }
    User[] public users;
    uint public nextId = 0;
    function Create(string memory name) public {
        users.push(User(nextId, name));
        nextId++;
    }
    function Read(uint id) view public returns(uint, string memory) {
        for(uint i=0; i<users.length; i++) {
            if(users[i].id == id) {
                return(users[i].id, users[i].name);
            }
        }
    }

    function Update(uint id, string memory name) public {
        for(uint i=0; i<users.length; i++) {
            if(users[i].id == id) {
                users[i].name =name;
            }
        }
    }

    function Delete(uint id) public {
        delete users[id];
    }
    function find(uint id) view internal returns(uint) {
        for(uint i=0; i< users.length; i++) {
            if(users[i].id == id) {
                return i;
            }
        }
        // if user does not exist then revert back
        revert("User does not exist");
    }
}
```

Practical No :3

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Crud {
    struct User {
        uint id;
        string name;
    }
    User[] public users;
    uint public nextId = 0;
    function Create(string memory name) public {
        users.push(User(nextId, name));
        nextId++;
    }
    function Read(uint id) view public returns(uint, string memory) {
        for(uint i=0; i<users.length; i++) {
            if(users[i].id == id) {
                return(users[i].id, users[i].name);
            }
        }
    }
    function Update(uint id, string memory name) public {
        for(uint i=0; i<users.length; i++) {
            if(users[i].id == id) {
                users[i].name =name;
            }
        }
    }
    function Delete(uint id) public {
        delete users[id];
    }
    function find(uint id) view internal returns(uint) {
        for(uint i=0; i< users.length; i++) {
            if(users[i].id == id) {
                return i;
            }
        }
        // if user does not exist then revert back
        revert("User does not exist");
    }
}
```

Practical No : 4

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Crud {
    struct User {
        uint id;
        string name;
    }
    User[] public users;
    uint public nextId = 0;
    function Create(string memory name) public {
        users.push(User(nextId, name));
        nextId++;
    }
    function Read(uint id) view public returns(uint, string memory) {
        for(uint i=0; i<users.length; i++) {
            if(users[i].id == id) {
                return(users[i].id, users[i].name);
            }
        }
    }
    function Update(uint id, string memory name) public {
        for(uint i=0; i<users.length; i++) {
            if(users[i].id == id) {
                users[i].name =name;
            }
        }
    }
    function Delete(uint id) public {
        delete users[id];
    }
    function find(uint id) view internal returns(uint) {
        for(uint i=0; i< users.length; i++) {
            if(users[i].id == id) {
                return i;
            }
        }
        // if user does not exist then revert back
        revert("User does not exist");
    }
}
```