```
//BT - Bank
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
contract Bank {
    uint256 private balance = 0;
    address public accHolder;
    constructor() {
        accHolder = msg.sender;  // msg.sender = deployer's address
    }
    function deposit() public payable {
        require(msg.value > 0, "Deposit amount should be greater than 0.");
        require(msg.sender == accHolder, "You are not the account owner.");
        balance += msg.value;
    }
    function withdraw(uint256 _amount) public {
        require(msg.sender == accHolder, "You are not the account owner.");
        require(_amount > 0, "Withdraw amount must be greater than 0.");
        require(balance >= _amount, "Not enough balance.");

        balance -= _amount;
        payable(msg.sender).transfer(_amount);
    }



    function showBalance() public view returns (uint256) {
        require(msg.sender == accHolder, "You are not the account owner.");
        return balance;
    }
}
```

BT - Student

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract StudentData {
    // Structure to store student details
    struct Student {
        uint rollNo;
        string name;
        uint age;
    }
    // Array to store multiple students
    Student[] public students;
    // Event to log student addition
    event StudentAdded(uint rollNo, string name, uint age);
    // Function to add a student
    function addStudent(uint _rollNo, string memory _name, uint _age) public {
        students.push(Student(_rollNo, _name, _age));
        emit StudentAdded(_rollNo, _name, _age);
    }
    // Function to get total number of students
    function getStudentCount() public view returns (uint) {
        return students.length;
    }

    // Function to get student details by index
    function getStudent(uint index) public view returns (uint, string memory, uint) {
        require(index < students.length, "Invalid index");
        Student memory s = students[index];
        return (s.rollNo, s.name, s.age);
    }
    // Fallback function to handle unexpected transactions
    fallback() external payable {
        // This executes if someone sends Ether or calls a non-existing function
    }
    // Receive function to handle plain Ether transfers
    receive() external payable {}
}
```

//**DAA – Fibonacci**

```
def fibonacci_iter(n):
    if n < 0:
        return -1, 1
    if n == 0 or n == 1:
        return n, 1
    steps = 0
    a = 0
    b = 1
    print("Iterative series:", a, b, end=" ")
    for i in range(2, n + 1):
        c = a + b
        print(c, end=" ")
        a = b
        b = c
        steps += 1
    print()  # new line after series
    return c, steps  # removed +1
def fibonacci_recur(n):
    if n < 0:
        return -1, 1
    if n == 0 or n == 1:
        return n, 1
    fib1, steps1 = fibonacci_recur(n - 1)
    fib2, steps2 = fibonacci_recur(n - 2)
    return fib1 + fib2, steps1 + steps2 + 1
def print_recursive_series(n):
    print("Recursive series:", end=" ")
    for i in range(n + 1):
        print(fibonacci_recur(i)[0], end=" ")
```

```
    print()
if __name__ == '__main__':
    n = int(input("Enter a number: "))
    fib_iter, steps_iter = fibonacci_iter(n)
    print("Iterative:", fib_iter)
    print("Steps:", steps_iter)
    print_recursive_series(n)
    fib_recur, steps_recur = fibonacci_recur(n)
    print("Recursive:", fib_recur)
    print("Steps:", steps_recur)
```

//**DAA – Huffman**

```
import heapq
class Node:
    def __init__(self, ch, freq):
        self.ch = ch
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def print_codes(root, code=""):
    if root is None:
        return
    if not root.left and not root.right:
        print(f"{root.ch} : {code}")
        return
    print_codes(root.left, code + "0")
    print_codes(root.right, code + "1")
def build_huffman(chars, freqs):
    heap = [Node(chars[i], freqs[i]) for i in range(len(chars))]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        new_node = Node('-', left.freq + right.freq)
        new_node.left = left
        new_node.right = right
        heapq.heappush(heap, new_node)
    print("\nHuffman Codes:")
    print_codes(heap[0])
n = int(input("Enter number of characters: "))
chars = input("Enter characters (no spaces): ").strip()
freqs = list(map(int, input("Enter their frequencies: ").split()))
build_huffman(list(chars), freqs)
```

//DAA – Fractional Knapsack

```
class Item:
    def __init__(self, profit, weight):
        self.profit = profit
        self.weight = weight
    # Define a function to calculate profit/weight ratio
    def ratio(self):
        return self.profit / self.weight
def fractionalKnapsack(capacity, items):
    # Sort items by ratio (profit per weight) in descending order
    items.sort(key=Item.ratio, reverse=True)
    totalValue = 0.0
    for item in items:
        if capacity >= item.weight:
            totalValue += item.profit
            capacity -= item.weight
        else:
            totalValue += item.profit * (capacity / item.weight)
            break
    return totalValue
# ---- Main Program ----
if __name__ == "__main__":
    n = int(input("Enter number of items: "))
    items = []
    for i in range(n):
        profit = int(input(f"Enter profit of item {i + 1}: "))
        weight = int(input(f"Enter weight of item {i + 1}: "))
        items.append(Item(profit, weight))
    capacity = int(input("Enter capacity of knapsack: "))
    print("Maximum value in knapsack:", fractionalKnapsack(capacity, items))
```

//DAA - knapsack

```
def knapsack_01(n, values, weights, W):
    dp = [[0] * (W+1) for _ in range(n+1)]
    for i in range(n+1):
        for w in range(W+1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]
    selected_items = []
    i, w = n, W
    while i > 0 and w > 0:
        if dp[i][w] != dp[i-1][w]:
            selected_items.append(i-1)
            w -= weights[i-1]
        i -= 1
    return dp[n][W], selected_items
if __name__ == "__main__":
    n = 3
    values = [60, 100, 120]
    weights = [10, 20, 30]
    W = 50

    max_value, selected_items = knapsack_01(n, values, weights, W)
    print("Maximum value:", max_value)
    print("Selected items:", selected_items)
```

//**DAA – N-queens**

```
N=8
pos=[-1]* N
pos[0]=0

def is_safe(row,col):
    for r in range(row):
        c=pos[r]
        if c==col or abs(c-col) == abs(r-row):
            return False
    return True

def solve(row=1):
    if row==N:
        for r in range(N):
            print(" ".join("Q" if pos[r]==c else "." for c in range(N)))
        return True
    for col in range(N):
        if is_safe(row,col):
            pos[row]=col
            if solve(row+1): return True
            pos[row]=-1
    return False


solve()
```

//ML – Uber

```
[1]import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
[2]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
[3]  df = pd.read_csv("uber.csv")
[4]  df.head()
[5]  print(df.isnull().sum())
[6]  df = df.dropna()
[7]  df["distance"] = (
   ((df['dropoff_latitude'] - df['pickup_latitude'])**2 +
    (df['dropoff_longitude'] - df['pickup_longitude'])**2) * 0.5
)
[8]  df.head()
[9]  sns.boxplot(x=df['fare_amount'])     # 14) Plot boxplot to visualize fare
distribution & potential outliers
plt.title("Boxplot for Fare Amount (Outliers)") # 15) Title
plt.show()
[10]  df = df[(df['fare_amount'] > 0) & (df['fare_amount'] < 100)]
df = df[(df['passenger_count'] > 0) & (df['passenger_count'] <= 6)]
df = df[df['distance'] < 5]
[11]  df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'])
df['hour'] = df['pickup_datetime'].dt.hour
df['day_of_week'] = df['pickup_datetime'].dt.dayofweek
[12]  corr = df[['fare_amount', 'distance', 'passenger_count', 'hour',
'day_of_week']].corr()
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title("Correlation Matrix")
plt.show()
```

```
[13]  X = df[['distance', 'passenger_count', 'hour', 'day_of_week']]  # 21) Feature matrix with 4 predictors
y = df['fare_amount']
[14]  X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
[15]  lr = LinearRegression()
lr.fit(X_train, y_train)
y_pred_lr = lr.predict(X_test)
[16]  rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
[17]  def evaluate(y_true, y_pred, model_name):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    r2 = r2_score(y_true, y_pred)
    print(f"{model_name} Results:")
    print(f"  R² Score: {r2:.4f}")
    print(f"  RMSE: {rmse:.4f}\n")

evaluate(y_test, y_pred_lr, "Linear Regression")
evaluate(y_test, y_pred_rf, "Random Forest Regression")
```

//ML – Email [1]import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
[2] df = pd.read_csv("emails.csv")
[3] df.head()
[4] X = df.drop(columns=['Prediction', 'Email No.'])
y = df['Prediction']
[5] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
[6] knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)
[7] knn_acc = accuracy_score(y_test, knn_pred)
[8] print("----- KNN Model -----")
print("Accuracy:", knn_acc)
print("Confusion Matrix:\n", confusion_matrix(y_test, knn_pred))
print("Classification Report:\n", classification_report(y_test, knn_pred))
[9] svm = SVC(kernel='linear')
svm.fit(X_train, y_train)
svm_pred = svm.predict(X_test)
[10] svm_acc = accuracy_score(y_test, svm_pred)
[11] print("\n----- SVM Model -----")
print("Accuracy:", svm_acc)
print("Confusion Matrix:\n", confusion_matrix(y_test, svm_pred))
print("Classification Report:\n", classification_report(y_test, svm_pred))
[12] print("\n----- Model Comparison -----")
print(f"KNN Accuracy: {knn_acc:.4f}")
print(f"SVM Accuracy: {svm_acc:.4f}")
if svm_acc > knn_acc:
  print("SVM performs better for spam classification.")
else: print("KNN performs better for spam classification.")

//**ML – bankchhurn**

```
[1]import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import confusion_matrix, accuracy_score
[2]  # pip install tensorflow ignore if already installed
[3]  from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
[4]  df = pd.read_csv("Churn_Modelling.csv")
[5]  df.head()
[6]  # 2. Distinguish the feature and target set
X = df.iloc[:, 3:13]   # Features (from CreditScore to EstimatedSalary)
y = df.iloc[:, 13]     # Target (Exited column)
[7]  labelencoder_gender = LabelEncoder()
X['Gender'] = labelencoder_gender.fit_transform(X['Gender'])
# One-hot encode Geography
X = pd.get_dummies(X, columns=['Geography'], drop_first=True)
[8]  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
[9]  # Normalize the data
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
[10]  X_train_df = pd.DataFrame(X_train, columns=X.columns)
print("Sample of normalized training data:")
display(X_train_df.head())
print("\nMean of features after scaling:\n", X_train_df.mean())
print("\nStandard deviation of features after scaling:\n", X_train_df.std())
[11]  model = Sequential()
model.add(Dense(units=6, activation='relu', input_dim=X_train.shape[1]))
model.add(Dropout(0.3))
model.add(Dense(units=6, activation='relu'))
model.add(Dense(units=1, activation='sigmoid'))
```

```
[12] model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
[13] history = model.fit(X_train, y_train, epochs=20, batch_size=32,
validation_data=(X_test, y_test))
[14] train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
print("Accuracy per Epoch:\n")
for i in range(len(train_acc)):
   print(f"Epoch {i+1}: Training Accuracy = {train_acc[i]*100:.2f}%, Validation
Accuracy = {val_acc[i]*100:.2f}%")
[15] import matplotlib.pyplot as plt
[16] plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy per Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
[17] y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5)
[18] cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
[19] acc = accuracy_score(y_test, y_pred)
print("Final Accuracy: {:.2f}%".format(acc * 100))
```

//ML -diabetes

```
[1]import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,
recall_score
[2]  df = pd.read_csv("diabetes.csv")
[3]  df.head()
[4]  X = df.drop("Outcome", axis=1)
y = df["Outcome"]
[5]  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
[6]  scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
[7]  knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
[8]  y_pred = knn.predict(X_test)
[9]  cm = confusion_matrix(y_test, y_pred)      # Confusion Matrix
accuracy = accuracy_score(y_test, y_pred)  # Accuracy
error_rate = 1 - accuracy                # Error Rate
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
[10]  print("\nConfusion Matrix:\n", cm)
print("Accuracy:", round(accuracy, 4))
print("Error Rate:", round(error_rate, 4))
print("Precision:", round(precision, 4))
print("Recall:", round(recall, 4))
```

//ML _ Sales [1]import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
[2] df = pd.read_csv("sales_data_sample.csv", encoding='latin1')
[3] numeric_cols = df.select_dtypes(include=['int64', 'float64']).columns
X = df[numeric_cols]
[4] scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
[5] wcss = [] # Within Cluster Sum of Squares
for i in range(1, 11):
  kmeans = KMeans(n_clusters=i, random_state=42)
  kmeans.fit(X_scaled)
  wcss.append(kmeans.inertia_)
[6] plt.figure(figsize=(8,5))
plt.plot(range(1, 11), wcss, marker='o')
plt.title("Elbow Method Graph")
plt.xlabel("Number of Clusters (K)")
plt.ylabel("WCSS")
plt.show()
[7] k = 4 # choose based on elbow plot
kmeans = KMeans(n_clusters=k, random_state=42)
df['KMeans_Cluster'] = kmeans.fit_predict(X_scaled)
print("\nK-Means Clustering result:")
df.head()
[8] linked = linkage(X_scaled, method='ward')
plt.figure(figsize=(10, 7))
dendrogram(linked, orientation='top', distance_sort='descending',
show_leaf_counts=True)
plt.title('Hierarchical Clustering Dendrogram')
plt.show()
df['Hier_Cluster'] = fcluster(linked, k, criterion='maxclust')
print("\nHierarchical Clustering result:") print(df.head())