CODE 1
To implement the **K-Nearest Neighbors (KNN)** algorithm on the `diabetes.csv` dataset and compute various evaluation metrics such as **confusion matrix**, **accuracy**, **error rate**, **precision**, and **recall**, follow the steps below.

First, download the dataset from Kaggle (you can load it into your environment and specify the correct path to the dataset file). Below is a step-by-step approach to implementing KNN and evaluating the performance.

Steps to Implement KNN Algorithm on the Diabetes Dataset:

1. **Import Libraries and Load the Dataset**:
   We need libraries such as `pandas`, `numpy`, `matplotlib`, `scikit-learn` for the machine learning model, and `seaborn` for visualization.
2. **Preprocess the Data**:
   Load the dataset, clean any missing values, and split it into features and labels. Normalize the features for better KNN performance.
3. **Train and Test Split**:
   Divide the dataset into training and testing sets.
4. **Implement KNN Algorithm**:
   Create and train a KNN model on the training data.
5. **Evaluate the Model**:
   Calculate the confusion matrix, accuracy, error rate, precision, and recall.

```
import pandas as pd
import numpy as np
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
import seaborn as sns
import matplotlib.pyplot as plt

# Step 1: Load the dataset
url = "path_to_your_downloaded/diabetes.csv"  # Update with the correct path
df = pd.read_csv(url)

# Step 2: Explore the dataset (optional)
print(df.head())

# Step 3: Preprocess the dataset
# Check for missing values and handle them if necessary
df.isnull().sum()  # Find any missing values in the dataset
```

```
# We can choose to fill missing values or drop rows/columns with missing data.
# For simplicity, let's assume there are no missing values in the dataset for now.

# Define features and target variable
X = df.drop('Outcome', axis=1)  # Features (all columns except 'Outcome')
y = df['Outcome']  # Target variable (Outcome)

# Step 4: 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 5: Normalize the features using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 6: Implement K-Nearest Neighbors (KNN)
knn = KNeighborsClassifier(n_neighbors=5)  # You can adjust the number of neighbors (k)
knn.fit(X_train_scaled, y_train)

# Step 7: Make predictions
y_pred = knn.predict(X_test_scaled)

# Step 8: Evaluate the model
# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Error Rate (1 - accuracy)
error_rate = 1 - accuracy
print(f"Error Rate: {error_rate}")

# Precision
precision = precision_score(y_test, y_pred)
print(f"Precision: {precision}")

# Recall
recall = recall_score(y_test, y_pred)
print(f"Recall: {recall}")
```

```
# Step 9: Plot the Confusion Matrix
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Not Diabetic", "Diabetic"],
yticklabels=["Not Diabetic", "Diabetic"])
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

Explanation:

- **Dataset Loading and Preprocessing:**
  - We load the dataset using `pd.read_csv()`.
  - `df.isnull().sum()` helps identify if there are any missing values.
  - We split the dataset into features (`X`) and the target variable (`y`). The target variable in this dataset is `Outcome`, which indicates whether a patient is diabetic (1) or not (0).
- **Data Splitting:**
  - We use `train_test_split` to divide the dataset into a training set (80%) and a testing set (20%).
- **Normalization:**
  - We normalize the features using `StandardScaler`, which scales the features to have zero mean and unit variance. This is essential for distance-based algorithms like KNN.
- **KNN Classifier:**
  - We initialize a KNN model with 5 neighbors (`n_neighbors=5`). You can experiment with other values of `k` to optimize performance.
- **Model Evaluation:**
  - **Confusion Matrix:** Shows the true positives, false positives, true negatives, and false negatives.
  - **Accuracy:** Measures the percentage of correct predictions.
  - **Error Rate:** The inverse of accuracy, i.e., 1−Accuracy1 - \text{Accuracy}1−Accuracy.
  - **Precision:** The proportion of positive predictions that are actually correct (useful in the case of imbalanced classes).
  - **Recall:** The proportion of actual positives that are correctly identified by the model.
- **Visualization:**
  - A confusion matrix heatmap is plotted using `seaborn.heatmap()` to visualize the performance.

Tuning `k` in KNN: The performance can vary depending on the value of `k`. You can experiment with different values of `k` (e.g., 3, 7, 9) to find the optimal value.

Cross-validation: You can use `cross_val_score` or `GridSearchCV` to tune hyperparameters and evaluate the model more robustly.

CODE 2

```
#!/usr/bin/env python
# coding: utf-8
# Input from user for number of queens
N = int(input("Enter the number of queens: "))
print(f"Entered number of queens: {N}\n")
# Chessboard initialization (NxN matrix with all elements set to 0)
board = [[0] * N for _ in range(N)]
# Function to check if a position (i, j) is under attack by any other queen
def is_attack(i, j):
# Check if there is a queen in the same row or column
for k in range(N):
if board[i][k] == 1 or board[k][j] == 1:
return True
# Check diagonals
for k in range(N):
for l in range(N):
if (k + l == i + j) or (k - l == i - j): # Checking if in diagonal
if board[k][l] == 1:
return True
return False
# Recursive function to solve the N-Queens problem
def N_queen(n):
# If n is 0, all queens are placed, return True (solution found)
if n == 0:
return True
# Try placing a queen in every position on the board
for i in range(N):
for j in range(N):
# Check if we can place a queen here
if not is_attack(i, j) and board[i][j] != 1:
board[i][j] = 1 # Place the queen
# Recursively try to place the remaining queens
if N_queen(n - 1):
return True # If a valid arrangement is found, return True
# If placing the queen here does not lead to a solution, backtrack
board[i][j] = 0
return False
# Solve the N-Queens problem
```

```
if N_queen(N):
# Output the solution
print(f"Solution for {N}-Queens Problem:")
for row in board:
print(" ".join(str(x) for x in row))
else:
print(f"No solution exists for {N}-Queens problem.")
```