```
%pip install -q otter-grader
```

```
import sys

IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    ! git clone https://github.com/tanish738/CDA-HW2-TESTS.git tests
    import otter
    grader = otter.Notebook()
else:
    print("Not running in Colab")
```

```
import otter
grader = otter.Notebook()
```

Assignment Summary

In this assignment, we explored and implemented several **fundamental machine learning algorithms from scratch**.
Each method approached classification differently, giving us insights into linear, tree-based, instance-based, and anomaly-detection models.

Distribution of Problems Solved

1. **Logistic Regression**
   - Implemented sigmoid, cost function, gradient computation, parameter updates, and prediction.
   - Learned how linear models classify data using probabilities and decision boundaries.
2. **One-Class SVM**
   - Implemented anomaly detection for imbalanced datasets.
   - Learned how to separate "normal" data from potential outliers using a margin-based approach.
3. **Decision Trees**
   - Implemented Gini impurity and best-split search.
   - Understood how recursive partitioning builds interpretable, rule-based models.
4. **K-Nearest Neighbors (KNN)**
   - Implemented distance calculation, neighbor search, and majority voting.
   - Learned how instance-based models adapt to complex, non-linear decision boundaries.

Logistic Regression from Scratch (Binary Classification)

Assignment Overview

In this assignment, you will implement a **binary logistic regression classifier** entirely from scratch using **Python** and the **NumPy** library.

The goal is to **classify data points into one of two distinct classes** (0 or 1) by learning the underlying decision boundary.
This exercise will strengthen your understanding of the **mathematics and mechanics** behind one of the most fundamental machine learning algorithms.

Learning Objectives

By completing this assignment, you will:

- Understand how logistic regression works under the hood.
- Implement essential components step by step:
  1. **Sigmoid function** – to map raw scores into probabilities.
  2. **Cost function** – binary cross-entropy loss.
  3. **Gradient computation** – calculate how to adjust parameters.

4. **Gradient Descent update rule** – iteratively optimize model parameters.
5. **Prediction function** – classify samples based on learned weights.

- Train and evaluate your logistic regression classifier on a dataset.

```
import sys
import numpy as np
import matplotlib.pyplot as plt
!curl -so p0_data.txt "https://dl.dropboxusercontent.com/scl/fi/8rhvgae0al2s9z9oabxoo/p0_data.txt?rlkey=bj4nbz4q013vz0a
data = np.loadtxt('p0_data.txt', delimiter=',')

train_X = data[:, 0:2]
train_y = data[:, 2]

# Get the number of training examples and the number of features
m_samples, n_features = train_X.shape
print ("# of training examples = ", m_samples)
print ("# of features = ", n_features)

pos = np.where(train_y == 1)
neg = np.where(train_y == 0)
plt.scatter(train_X[pos, 0], train_X[pos, 1], marker='o', c='b')
plt.scatter(train_X[neg, 0], train_X[neg, 1], marker='x', c='r')
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
plt.legend(['Admitted', 'Not Admitted'])
plt.show()
```

```
def sigmoid(z: int)-> float:
    """
      Sigmoid function
       Parameters
    ----------
    z : float or numpy.ndarray
        The input value(s) for which the sigmoid is to be calculated. The function is designed to handle both single nu

    Returns
    -------
    float or numpy.ndarray
        The result of the sigmoid calculation. The output will be of the same type as the input `z`, with all values ma
    """
    # ****** ENTER CODE  ******
    s = 1 / (1 + np.exp(-z))
    return s
    # ****** END CODE  ******
```

```
grader.check("q1")
```

Logistic Regression Cost Function

We will implement the **Binary Cross-Entropy Loss** (Log Loss) for logistic regression.

1. **Use the sigmoid function** for predictions:
2. **Clip predicted probabilities** to avoid `log(0)` errors:
3. **Compute the average loss** over all training examples:

```
def cost_function(theta: np.ndarray, X: np.ndarray, y: np.ndarray) -> float:
    """ Logistic regression cost function
    Hint: clip h to avoid log(0).
     Parameters
    ----------
    theta : numpy.ndarray
        The model's parameters (weights and bias) with shape (n_features + 1,).

    X : numpy.ndarray
        The feature matrix with shape (m_samples, n_features + 1).

    y : numpy.ndarray
        The target vector containing the true labels (0s and 1s) with shape (m_samples,).
```

```
    Returns
    -------
    float
        The computed logistic regression cost, a single scalar value.
    """

    m = len(y)
    # ****** ENTER CODE  ******
    h = sigmoid(np.dot(X, theta))
    # Clip predictions to avoid log(0)
    h = np.clip(h, 1e-15, 1 - 1e-15)
    cost = -(1/m) * (np.dot(y, np.log(h)) + np.dot((1-y), np.log(1-h)))
    return cost
    # ****** END CODE  ******
```

```
grader.check("q2")
```

Gradient Update

To compute the gradient of the loss function with respect to the parameters:

1. **Subtract actual labels from predictions:**
2. **Multiply by the feature matrix transpose:**
3. **Divide by the number of examples (m):**

```
def gradient_update(theta: np.ndarray, X: np.ndarray, y: np.ndarray) -> np.ndarray:
    """ Gradient for logistic regression
    Parameters
    ----------
    theta : numpy.ndarray
        The model's current parameters (weights and bias) with shape `(n_features + 1,)`.

    X : numpy.ndarray
        The feature matrix with shape `(m_samples, n_features + 1)`.

    y : numpy.ndarray
        The target vector containing the true labels (0s and 1s) with shape `(m_samples,)`.

    Returns
    -------
    numpy.ndarray
        The computed gradient vector with the same shape as `theta`, `(n_features + 1,)`.

    """
    m = len(y)
    # ****** ENTER CODE  ******
    h = sigmoid(np.dot(X, theta))
    gradient = (1/m) * np.dot(X.T, (h - y))
    return gradient
    # ****** END CODE  ******
```

```
grader.check("q3")
```

Gradient Descent Algorithm

In each iteration of Gradient Descent we do two key updates:

1. **Gradient Update**
    - Compute the gradient of the loss function w.r.t. the parameters.
2. **Parameter Update**
    - Update parameters in the opposite direction of the gradient:

```
def gradient_descent(theta, X, y, alpha, max_iterations, print_iterations):
    """ Batch gradient descent algorithm """
    iteration = 0
```

```
        prev_cost = float('inf')  # Track previous cost

        while iteration < max_iterations:
            iteration += 1
            #  ***** ENTER YOUR CODE *****
            gradient = gradient_update(theta, X, y)
            theta = theta - alpha * gradient
            # ****** END CODE  ******

            # For every print_iterations number of iterations
            if iteration % print_iterations == 0 or iteration == 1:
                cost = cost_function(theta, X, y)
                print("[ Iteration", iteration, "]", "cost =", cost)

                # Visualization
                plt.figure(figsize=(5, 4))
                plt.xlim([20,110])
                plt.ylim([20,110])

                pos = np.where(y == 1)
                neg = np.where(y == 0)

                # Plot original data
                plt.scatter(X[pos, 1], X[pos, 2], marker='o', c='b')
                plt.scatter(X[neg, 1], X[neg, 2], marker='x', c='r')
                plt.xlabel('Exam 1 score')
                plt.ylabel('Exam 2 score')
                plt.legend(['Admitted', 'Not Admitted'])

                # Plot decision boundary
                t = np.linspace(20, 110, 100)
                if abs(theta[2]) > 1e-6:
                    decision_boundary = -(theta[0] + theta[1] * t) / theta[2]
                    mask = (decision_boundary >= 20) & (decision_boundary <= 110)
                    if np.any(mask):
                        plt.plot(t[mask], decision_boundary[mask], c='g', linewidth=2, label='Decision Boundary')

                plt.title(f'Iteration {iteration}')
                plt.show()
            # Early stopping if cost stops improving significantly
            if iteration > 1000 and abs(prev_cost - cost) < 1e-6:
                print(f"Converged at iteration {iteration}")
                break

            prev_cost = cost

        return theta
```

```
grader.check("q4")
```

Predict Function (Logistic Regression)

Steps:

1. Compute probabilities using the **sigmoid function**:
2. Apply **threshold at 0.5**:
   - If (h >= 0.5) → predict **1**
   - If (h < 0.5) → predict **0**

Theory Question 1 - Accuracy of Logistic Regression

Feel Free to change the parameters : [Hint - Use the visualization to get insights of data]

- The accuracy expected is 90 and you might have to change the parameters to get this
- This will be manaully graded we will also check the fit of the line in the graphs
- In the submission for this question you need to present all the graphs basically the output of the cell below after it is executed

```
train_X_with_bias = np.column_stack([np.ones(len(train_X)), train_X])
print(f"Data shape with bias: {train_X_with_bias.shape}")

initial_theta = np.array([0, 0, 0]) # you can change the intialization for better results
alpha_test = 0.01
max_iter = 1000
print_iter = 100

learned_theta = gradient_descent(initial_theta, train_X_with_bias, train_y, alpha_test, max_iter, print_iter)

def predict(theta, X):
    pass
    #  Predict labels using learned parameters
    # ****** ENTER CODE  ******
    probabilities = sigmoid(np.dot(X, theta))
    predictions = (probabilities >= 0.5).astype(int)
    return predictions
    # ****** END CODE  ******


def calculate_accuracy(theta, X, y):
    """ Calculate classification accuracy """
    predictions = predict(theta, X)
    accuracy = np.mean(predictions == y) * 100
    return accuracy

training_accuracy = calculate_accuracy(learned_theta, train_X_with_bias, train_y)
print(f"\nTraining Accuracy: {training_accuracy}%")
```

One-Class SVM

Purpose:
One-Class SVM (Support Vector Machine) is an **unsupervised learning algorithm** used for **anomaly detection** or **outlier detection**.

Idea:
It tries to **learn the boundary of a single class** of "normal" data and identifies points that lie **outside this boundary** as anomalies.

How it Works:

1. Maps input data into a **high-dimensional feature space** using a **kernel function** (commonly RBF).
2. Finds a **hyperplane** (or hypersurface) that best **encloses the normal data**.
3. Points outside this boundary are considered **outliers**.

Key Parameters:

- `nu`: An upper bound on the fraction of outliers (controls sensitivity).
- `kernel`: The function used to map data to higher dimensions.

Credit Card Fraud Detection: 3D PCA Visualization

Dataset

- Kaggle's **Credit Card Fraud dataset** with transactions labeled as `0 = normal` and `1 = fraud`.
- Features include `Time`, `Amount`, and 28 anonymized PCA components (`V1`–`V28`).

Steps in the Code

1. **Resampling:** Created a smaller dataset of 2000 transactions with a **70:30 ratio** of normal to fraud for better visualization.
2. **Feature Preparation:** Separated features (`X`) and labels (`y_true`) and **standardized** features using `StandardScaler`.
3. **Dimensionality Reduction:** Applied **PCA** to reduce features to 3 components for 3D plotting.
4. **Visualization:** Plotted normal (blue) and fraud (red) transactions in 3D PCA space to observe patterns and class separation.

Purpose

- Helps visualize the distribution of normal vs. fraudulent transactions and provides intuition before applying anomaly detection or classification models like **One-Class SVM**.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from mpl_toolkits.mplot3d import Axes3D  # needed for 3D plots
from sklearn.utils import resample
import kagglehub
from kagglehub import KaggleDatasetAdapter
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import numpy as np
import pandas as pd

    # 1. Load dataset from Kaggle
df = kagglehub.dataset_load(
        KaggleDatasetAdapter.PANDAS,
        "mlg-ulb/creditcardfraud",
        "creditcard.csv",
    )

# --- 1. Resample to 70:30 ratio ---
fraud = df[df["Class"] == 1]
normal = df[df["Class"] == 0]

# target total size
target_total = 2000
target_fraud = int(0.3 * target_total)   # 30% fraud
target_normal = int(0.7 * target_total)  # 70% normal

fraud_resampled = resample(fraud, replace=True, n_samples=target_fraud, random_state=42)
normal_resampled = resample(normal, replace=False, n_samples=target_normal, random_state=42)

df_small = pd.concat([fraud_resampled, normal_resampled]).sample(frac=1, random_state=42)

print(f"Reduced dataset size: {len(df_small)}")
print(df_small["Class"].value_counts(normalize=True))

# --- 2. Features and labels ---
X = df_small.drop("Class", axis=1).values
y_true = df_small["Class"].values  # 0 = normal, 1 = fraud

# --- 3. Standardize features ---
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# --- 4. Reduce to 3D with PCA ---
pca = PCA(n_components=3, random_state=42)
X_3d = pca.fit_transform(X_scaled)

# --- 5. Plot in 3D ---
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X_3d[y_true == 0, 0], X_3d[y_true == 0, 1], X_3d[y_true == 0, 2],
           c='blue', alpha=0.5, label="Normal")
ax.scatter(X_3d[y_true == 1, 0], X_3d[y_true == 1, 1], X_3d[y_true == 1, 2],
           c='red', alpha=0.7, label="Fraud")

ax.set_title("3D PCA Projection of Credit Card Transactions (70:30 ratio)")
ax.set_xlabel("PC 1")
ax.set_ylabel("PC 2")
ax.set_zlabel("PC 3")
ax.legend()
plt.show()
```

One-Class SVM: Primal and Dual Form

The goal of a One-Class SVM is to **find a boundary around the normal data (inliers)** in a high-dimensional space. Formally, we want to find a hyperplane that separates most of the data from the origin in feature space $\phi(x)$.

Primal Formulation

$$\min_{w,\rho,\xi} \quad \frac{1}{2}\|w\|^2 - \rho + \frac{1}{\nu n}\sum_{i=1}^{n} \xi_i$$

subject to:

$$w^\top \phi(x_i) \geq \rho - \xi_i, \quad \xi_i \geq 0$$

Where:

- w = normal vector of the hyperplane
- ρ = offset / threshold
- ξ_i = slack variables (allow some points to lie outside the boundary)
- $\nu \in (0, 1]$ = upper bound on the fraction of outliers

Intuition:

- Minimize $\|w\|^2$ → keep the hyperplane “tight” around the data
- Maximize ρ → push the boundary away from the origin
- Slack ξ_i allows some points to be considered outliers

Dual Formulation

Introducing Lagrange multipliers $\mu_i \geq 0$ for the constraints and solving gives the dual problem:

$$\min_{\mu} \quad \frac{1}{2}\sum_{i,j} \mu_i \mu_j K(x_i, x_j)$$

subject to:

$$0 \leq \mu_i \leq \frac{1}{\nu n}, \quad \sum_i \mu_i = 1$$

Where:

- $K(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$ is the **kernel function**
- μ_i = dual coefficients associated with each training point

Intuition:

- The kernel trick allows computation in high-dimensional space without explicitly mapping $\phi(x)$
- Constraints enforce that only a fraction ν of points can be outliers
- $\sum_i \mu_i = 1$ ensures proper scaling of the hyperplane

Programming Assignment: Kernel One-Class SVM

Overview

In this assignment, you will implement parts of a **Kernel-based One-Class SVM**.
The goal is to learn how kernel methods and dual optimization are used to detect anomalies (outliers vs. inliers).

We provide a partially completed class `KernelOneClassSVM` with:

- Pre-written methods for optimization and training.
- Clear function stubs and docstrings.
- Comments showing exactly what you need to implement.

Your task is to **fill in the missing methods**

Class Provided

The `KernelOneClassSVM` class contains the following methods:

Already Implemented (Do NOT modify)

- `__init__` → initializes the model with kernel parameters.
- `fit` → fits the model by solving the dual optimization problem.
- `score_samples` → computes anomaly scores using the decision function.

To Implement (Your Task)

The following functions must be completed by you:

1. `_kernel_function(self, X1, X2)`
   - Compute the kernel matrix between two datasets.
   - Use Gaussian RBF:
2. `_objective(self, mu)`
   - Compute the dual objective function:
3. `_objective_gradient(self, mu)`
   - Compute the gradient of the dual objective:
4. `_compute_rho(self)`
   - Compute the threshold ρ using complementary slackness:
     - For support vectors strictly inside bounds ($0 < \mu_i < 1/(\nu n)$):
     - If none exist, take the average over all support vectors.
5. `decision_function(self, X)`
   - Compute decision values for test samples:
6. `predict(self, X)`
   - Predict labels based on decision values:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from sklearn.metrics import pairwise_distances
from sklearn.metrics import recall_score
import warnings
warnings.filterwarnings('ignore')
```

```

class KernelOneClassSVM:
    def __init__(self, kernel='rbf', gamma=0.1, nu=0.1, tol=1e-6):
        self.kernel = kernel
        self.gamma = gamma
        self.nu = nu
        self.tol = tol

        # These will be set during fitting
        self.X_train = None
        self.mu = None
        self.support_vectors = None
        self.support_indices = None
        self.rho = None
        self.K_train = None
    def _kernel_function(self, X1: np.ndarray, X2: np.ndarray) -> np.ndarray:
        #Compute kernel matrix between X1 and X2.

        #Parameters:
        #-----------
        #X1 : array-like, shape (n_samples_1, n_features)
        #X2 : array-like, shape (n_samples_2, n_features)

        #Returns:
        #--------
        #K : array, shape (n_samples_1, n_samples_2)
        #    Kernel matrix
        #  Gaussian RBF kernel: exp(-γ||x_i - x_j||²)
        # ****** ENTER CODE  ******
        distances = pairwise_distances(X1, X2, metric='sqeuclidean')
        K = np.exp(-self.gamma * distances)
        return K
        # ****** END CODE  ******
    def _objective(self, mu: np.ndarray) -> float:
        #   Parameters
        #     ----------
        #     mu : numpy.ndarray
        #         The vector of dual coefficients or optimization variables, with shape `(n_samples,)`.
        #     Returns
```

```
        #     -------
        #     float
        #         The scalar value of the dual objective function.
        #     Implementation Details
        #     --------------------
        #     The function calculates the objective using the formula:
        #     L(\mu) = \frac{1}{2} \mu^\top K \mu
        # ****** ENTER CODE  ******
        objective = 0.5 * np.dot(mu.T, np.dot(self.K_train, mu))
        return objective
        # ****** END CODE  ******

    def _objective_gradient(self, mu: np.ndarray) -> np.ndarray:
        # Parameters
        # ----------
        # mu : numpy.ndarray
        #     The vector of dual coefficients or optimization variables, with shape `(n_samples,)`.
        # Returns
        # -------
        # numpy.ndarray
        #     The computed gradient vector, with the same shape as `mu`, `(n_samples,)`.
        # # Gradient formula:
        # ∇_μ (1/2 * μ^T K μ) = K μ
        # ****** ENTER CODE  ******
        gradient = np.dot(self.K_train, mu)
        return gradient
        # ****** END CODE  ******

    # No Changes To be Made in the fit  function
    def fit(self, X):
        #  Fit the One-Class SVM model.
        # Parameters
        # ----------
        # X : numpy.ndarray, shape (n_samples, n_features)
        #     The training data to be used for fitting the model.
        # Returns
        # -------
        # OneClassSVM
        #     Returns the instance of the class itself, allowing for method chaining.
        # Intitalize the data
        X = np.array(X)
        self.X_train = X
        n_samples = X.shape[0]
        # Compute kernel matrix
        self.K_train = self._kernel_function(X, X)

        # Set up optimization problem
        # Variables: μ (dual coefficients)
        mu_init = np.ones(n_samples) / n_samples  # Initialize to satisfy equality constraint

        # Bounds: 0 ≤ μᵢ ≤ 1/(ν*n)
        bounds = [(0, 1.0 / (self.nu * n_samples)) for _ in range(n_samples)]

        # Equality constraint: μᵀ1 = 1
        constraint = {
                'type': 'eq',
                'fun': lambda mu: np.sum(mu) - 1.0,
                'jac': lambda mu: np.ones(n_samples)
        }

            # Solve optimization problem
        result = minimize(
            fun=self._objective,
            x0=mu_init,
            method='SLSQP',
            jac=self._objective_gradient,
            bounds=bounds,
            constraints=constraint,
            options={'ftol': self.tol, 'disp': False}
        )
        if not result.success:
            print(f"Optimization warning: {result.message}")
        self.mu = result.x
        self.support_indices = np.where(self.mu > self.tol)[0]
        self.support_vectors = X[self.support_indices]
        mu_support = self.mu[self.support_indices]
        # Compute ρ using complementary slackness
```

```
        # For support vectors with 0 < μᵢ < 1/(ν*n), we have ⟨w,xᵢ⟩ = ρ
        self._compute_rho()

        return self

    def _compute_rho(self):
        # Compute the threshold ρ using complementary slackness conditions.
        # This threshold defines the boundary of the decision function. Data points are
        # classified as inliers or outliers based on their position relative to this threshold.
        # The value of rho is computed based on the support vectors found during optimization.
        # - For support vectors strictly inside bounds (0 < μ_i < 1/(ν*n)):
        #  ρ = Σ μ_j K(x_j, x_i)
        # - If none satisfy, take the average decision over all support vectors.
        #Steps:
        #1. Compute the upper bound for μᵢ, which is 1 / (ν * n).
        #2. Identify support vectors where μᵢ lies strictly between (tol, upper_bound - tol).
        #These are the margin support vectors.
        #3. If such support vectors exist:
            #- Pick one (e.g., the first).
             # - Compute ρ as Σ μⱼ K(xⱼ, xᵢ).
        #4. If none exist:
            #- Fall back to computing ρ as the average decision value over all support vectors.
            # set self.rho no need to return anything
        n_samples = len(self.X_train)
        upper_bound = 1.0 / (self.nu * n_samples)

        # Find support vectors that are strictly between bounds
        mask = (self.mu > self.tol) & (self.mu < upper_bound - self.tol)

        # ****** ENTER CODE  ******
        if np.any(mask):
            # Use margin support vectors
            margin_sv_idx = np.where(mask)[0][0]  # Take the first one
            self.rho = np.sum(self.mu * self.K_train[:, margin_sv_idx])
        else:
            # Fallback: average over all support vectors
            sv_decision_values = np.sum(self.mu[:, np.newaxis] * self.K_train, axis=0)[self.support_indices]
            self.rho = np.mean(sv_decision_values)
        # ****** END CODE  ******


    def decision_function(self, X: np.ndarray) -> np.ndarray:
        # Parameters:
        # ----------
        # X : array-like, shape (n_samples, n_features)
        #     Test samples
        # Returns:
        # -------
        # decision : array, shape (n_samples,)
        #     Decision function values
        # Decision function formula:
        # f(x) = Σ μ_i K(x, x_i) - ρ
        # - Positive → inlier
        # - Negative → outlier
        if self.mu is None:
            raise ValueError("Model has not been fitted yet.")

        X = np.array(X)


        # ****** ENTER CODE  ******
        K_test = self._kernel_function(X, self.X_train)
        decision = np.dot(K_test, self.mu) - self.rho
        return decision
        # ****** END CODE  ******

    def predict(self, X: np.ndarray) -> np.ndarray:
        # Parameters:
        # ----------
        # X : array-like, shape (n_samples, n_features)
        #     Test samples
        # Returns:
        # y_pred : array, shape (n_samples,)
        #     Predicted labels (1 for inliers, -1 for outliers)
        # - Use decision_function(X)
        # - Apply threshold 0:
        #     decision >= 0 → inlier (1)
```

```
        #     decision < 0 → outlier (-1)
        # Predicted labels (1 for inliers, -1 for outliers) is an array
        # ****** ENTER CODE  ******
        decision = self.decision_function(X)
        y_pred = np.where(decision >= 0, 1, -1)
        return y_pred
        # ****** END CODE  ******
    def score_samples(self, X):
        """
        Parameters:
        -----------
        X : array-like, shape (n_samples, n_features)
        Returns:
        --------
        scores : array, shape (n_samples,)
            Anomaly scores
        """
        return self.decision_function(X)



X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y_true, test_size=0.3, random_state=42, stratify=y_true
    )
```

```
grader.check("q5")
```

```
X_train_normal = X_train[y_train == 0]
model = KernelOneClassSVM(kernel="rbf", gamma=0.1, nu=0.05)
model.fit(X_train_normal)


y_pred = model.predict(X_test)
    # Convert One-Class SVM outputs (-1 = anomaly, +1 = inlier) → fraud = 1, normal = 0
y_pred_binary = np.where(y_pred == -1, 1, 0)

    # 9. Evaluation
print("\n=== Evaluation on Test Set ===")
print(confusion_matrix(y_test, y_pred_binary))
report = classification_report(y_test, y_pred_binary, digits=4)
print(report)
recall = recall_score(y_test, y_pred_binary, average='macro')   # or 'weighted', 'micro', None
print("Recall:", recall)
```

```
grader.check("q6")
```

Do not need to change the below function it is used to - demonstrate the use of a custom **Kernel One-Class SVM** for anomaly detection on credit card fraud data, including 3D visualization using PCA and evaluation on a test set.

```
from sklearn.model_selection import train_test_split
def plot_ocsvm_results_3d(X, y_true, model, title="One-Class SVM Results (3D)"):
    """
    Visualize One-Class SVM results in 3D using PCA for dimensionality reduction.
    Decision boundary is approximated via grid sampling.
    """
    # Reduce to 3D
    pca = PCA(n_components=3, random_state=42)
    X_3d = pca.fit_transform(X)

    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(111, projection='3d')

    # Scatter plot of inliers/outliers
    inliers_mask = y_true == 0
    outliers_mask = y_true == 1

    ax.scatter(X_3d[inliers_mask, 0], X_3d[inliers_mask, 1], X_3d[inliers_mask, 2],
               c="blue", alpha=0.5, label="Normal")
    ax.scatter(X_3d[outliers_mask, 0], X_3d[outliers_mask, 1], X_3d[outliers_mask, 2],
```

```
                  c="red", alpha=0.7, label="Fraud")

        # Support vectors (projected in 3D)
        if len(model.support_indices) > 0:
            support_proj = pca.transform(model.X_train[model.support_indices])
            ax.scatter(support_proj[:, 0], support_proj[:, 1], support_proj[:, 2],
                       facecolors="none", edgecolors="black", s=80, linewidth=1.5,
                       label="Support Vectors")

        # Approximate decision boundary using a 3D grid
        grid_size = 20  # reduce if too slow
        x_min, x_max = X_3d[:, 0].min() - 1, X_3d[:, 0].max() + 1
        y_min, y_max = X_3d[:, 1].min() - 1, X_3d[:, 1].max() + 1
        z_min, z_max = X_3d[:, 2].min() - 1, X_3d[:, 2].max() + 1

        xx, yy, zz = np.meshgrid(
            np.linspace(x_min, x_max, grid_size),
            np.linspace(y_min, y_max, grid_size),
            np.linspace(z_min, z_max, grid_size)
        )
        grid_points = np.c_[xx.ravel(), yy.ravel(), zz.ravel()]
        # Project grid back to original space
        grid_original = pca.inverse_transform(grid_points)
        scores = model.decision_function(grid_original)
        scores = scores.reshape(xx.shape)

        # Plot isosurface (decision boundary at 0)
        try:
            from skimage import measure
            verts, faces, _, _ = measure.marching_cubes(scores, level=0)
            verts_transformed = np.c_[xx.ravel()[verts[:, 0].astype(int)],
                                      yy.ravel()[verts[:, 1].astype(int)],
                                      zz.ravel()[verts[:, 2].astype(int)]]
            ax.plot_trisurf(verts_transformed[:, 0], verts_transformed[:, 1],
                            faces, verts_transformed[:, 2], color="cyan", alpha=0.15)
        except Exception as e:
            print("⚠️ Could not render 3D boundary surface:", e)

        ax.set_title(title)
        ax.set_xlabel("PC 1")
        ax.set_ylabel("PC 2")
        ax.set_zlabel("PC 3")
        ax.legend()
        plt.show()

    plot_ocsvm_results_3d(X_test, y_test, model,
    title="One-Class SVM on Credit Card Fraud (3D PCA Projection)")
```

Pima Indians Diabetes Dataset

The **Pima Indians Diabetes dataset** is a well-known medical dataset from the **National Institute of Diabetes**.
It is frequently used for **binary classification tasks** in machine learning.

Dataset Overview

- **Samples (rows):** 768 female patients (age ⩾ 21)
- **Features (columns):** 8 health-related attributes
- **Target (label):** Binary outcome
  - `0` ⟶ No diabetes
  - `1` ⟶ Diabetes

Features

Feature	Description
Pregnancies	Number of times pregnant
Glucose	Plasma glucose concentration
BloodPressure	Diastolic blood pressure (mm Hg)
SkinThickness	Triceps skinfold thickness (mm)
Insulin	2-Hour serum insulin (mu U/ml)

Feature	Description
BMI	Body mass index (weight/height²)
DiabetesPedigreeFunction	Diabetes pedigree function
Age	Age in years

Data Loading Steps

1. Dataset is **downloaded directly** from GitHub.
2. **Column names** are assigned (since the raw file has none).
3. Data is loaded into a **pandas DataFrame**.
4. **Basic info** is printed:
   - Shape of dataset
   - Feature names
   - Class distribution
5. Data is split into:
   - **Features (X)** → NumPy array
   - **Target labels (y)** → NumPy array

Summary

- Dataset contains **768 samples** with **8 input features**.
- Target is **binary (0/1)** for diabetes diagnosis.
- No preprocessing applied yet (raw form).

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import LabelEncoder
import requests
from io import StringIO
from typing import Dict, Any, Union


def load_pima_diabetes_data():
    """Load the Pima Indians Diabetes dataset as-is"""
    # UCI dataset URL
    url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"

    # Column names for Pima Indians Diabetes dataset
    columns = [
        'pregnancies', 'glucose', 'blood_pressure', 'skin_thickness',
        'insulin', 'bmi', 'diabetes_pedigree', 'age', 'outcome'
    ]

    # Download with requests (disable SSL verify to avoid cert issues)
    print("Downloading Pima Indians Diabetes dataset...")
    response = requests.get(url, verify=False)
    if response.status_code != 200:
        raise Exception(f"Failed to download dataset, status code {response.status_code}")

    # Convert response content to DataFrame
    data = pd.read_csv(StringIO(response.text), names=columns)
    print(f"Successfully loaded {len(data)} samples")

    # Display basic info about the dataset
    print(f"\nDataset shape: {data.shape}")
    print(f"Feature columns: {columns[:-1]}")
    print(f"Target column: {columns[-1]}")
    print(f"Class distribution: {data['outcome'].value_counts().sort_index().values}")

    # Separate features and target - NO preprocessing, use raw data
    X = data.drop('outcome', axis=1).values
    y = data['outcome'].values

    return X, y, columns[:-1]
```

Decision Tree from Scratch

In this section, we will implement **two core functions** used in building a Decision Tree:

1. **Gini Impurity**
   - Measures how mixed the classes are in a node.
   - A pure node (all samples same class) has **Gini = 0**.
2. **Best Split**
   - For each feature and threshold, split the data into left/right subsets.
   - Compute **weighted Gini impurity** of the split.
   - Choose the feature and threshold with the **lowest impurity**.

```
from typing import Dict, Any, Union
class DecisionTreeFromScratch:
    def __init__(self, criterion="gini", max_depth=5, min_samples_split=2):
        self.criterion = criterion
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.tree = None

    def gini_impurity(self, y: np.ndarray) -> float:
        # Compute the Gini impurity for labels y.
        # Parameters
        # ----------
        # y : numpy.ndarray, shape (n_samples,)
        #     The array of class labels for which to compute the impurity.

        # Returns
        # -------
        # float
        #     The computed Gini impurity, a value between 0 and 1. A value of 0 indicates
        #     a pure node (all samples belong to the same class), while a higher value
        #     indicates greater impurity.
        # Formula: Gini(y) = 1 - Σ p(c)^2
        if len(y) == 0:
            return 0
        _, counts = np.unique(y, return_counts=True)
        # ****** ENTER CODE  ******
        probs = counts / len(y)
        gini = 1 - np.sum(probs ** 2)
        return gini
        # ****** END CODE  ******

    def _best_split(self, X: np.ndarray, y: np.ndarray) -> tuple[int, float | None]:
        # Find the best split of data based on Gini impurity.
        # Parameters
        # ----------
        # X : numpy.ndarray, shape (n_samples, n_features)
        #     The feature matrix of the dataset.
        # y : numpy.ndarray, shape (n_samples,)
        #     The target labels corresponding to the samples in `X`.

        # Returns
        # -------
        # tuple[int, float | None]
        #     A tuple containing the index of the best feature to split on and the
        #     corresponding threshold value.
        # You have to implement weighted gini impurity
        n_samples, n_features = X.shape
        best_score = float("inf")
        best_feat, best_thresh = None, None

        for feat in range(n_features):
            thresholds = np.unique(X[:, feat])
            for thresh in thresholds:
                left_mask = X[:, feat] <= thresh
                right_mask = ~left_mask

                y_left, y_right = y[left_mask], y[right_mask]

                if len(y_left) == 0 or len(y_right) == 0:
                    continue
```

```
                # Weighted Gini impurity
                # ****** ENTER CODE  ******
                n_left, n_right = len(y_left), len(y_right)
                gini_left = self.gini_impurity(y_left)
                gini_right = self.gini_impurity(y_right)
                score = (n_left/n_samples) * gini_left + (n_right/n_samples) * gini_right
                # ****** END CODE  ******

                if score < best_score:
                    best_score, best_feat, best_thresh = score, feat, thresh

        return best_feat, best_thresh




    def _build(self, X: np.ndarray, y: np.ndarray, depth: int, parent_label: int) -> Dict[str, Any]:
        """Recursively build the decision tree."""
        if len(y) == 0:
            return {"leaf": True, "label": parent_label}

        if (depth >= self.max_depth or len(y) < self.min_samples_split or
            len(np.unique(y)) == 1):
            label = np.bincount(y).argmax()
            return {"leaf": True, "label": label}

        feat, thresh = self._best_split(X, y)
        if feat is None:
            label = np.bincount(y).argmax()
            return {"leaf": True, "label": label}

        left_mask = X[:, feat] <= thresh
        right_mask = ~left_mask
        label = np.bincount(y).argmax()

        return {
            "leaf": False,
            "feature": feat,
            "thresh": thresh,
            "left": self._build(X[left_mask], y[left_mask], depth + 1, label),
            "right": self._build(X[right_mask], y[right_mask], depth + 1, label)
        }

    def fit(self, X: np.ndarray, y: np.ndarray) -> None:
        """Fit the decision tree to data X and labels y."""
        parent_label = np.bincount(y).argmax()
        self.tree = self._build(X, y, 0, parent_label)

    def _predict_one(self, node: Dict[str, Any], x: np.ndarray) -> int:
        """Predict the label for a single input sample."""
        if node["leaf"]:
            return node["label"]

        if x[node["feature"]] <= node["thresh"]:
            return self._predict_one(node["left"], x)
        else:
            return self._predict_one(node["right"], x)

    def predict(self,X: np.ndarray) -> np.ndarray:
        """Predict labels for all samples in dataset X."""
        return np.array([self._predict_one(self.tree, x) for x in X])
```

```
grader.check("q7")
```

Implementing K-Nearest Neighbors (KNN) from Scratch

In this assignment, you will implement the **K-Nearest Neighbors (KNN)** classification

---

Background

KNN is a **lazy learning algorithm** that makes predictions for a new data point by looking at the *k* closest training examples (neighbors) in feature space. The predicted label is decided by **majority vote** among the neighbors.

Steps:

1. Store the training data.
2. For each test sample:
   - Compute the distance to each training sample.
   - Identify the `k` nearest neighbors.
   - Collect their labels and perform majority voting.
3. Return the predicted label.

Your Task

We provide you with a class `KNNFromScratch` containing function stubs.
You must complete the missing implementation of the following function.

- **`predict`**: For each test point, compute distances, pick `k` nearest neighbors, and perform majority voting.

```
class KNNFromScratch:
    def __init__(self, k=3, metric="euclidean"):
        self.k = k
        self.metric = metric

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def _distance(self, a, b):
        if self.metric == "euclidean":
            return np.sqrt(np.sum((a - b) ** 2))
        elif self.metric == "manhattan":
            return np.sum(np.abs(a - b))
        else:
            raise ValueError(f"Unknown metric: {self.metric}")

    def predict(self, X_test):
        """
        Hints:
        1. For each test sample, compute the distance to every training sample.
        2. Sort the distances and pick the indices of the k nearest neighbors.
        3. Collect the labels of these k nearest neighbors.
        4. Use majority voting to decide the final predicted label.
        5. Store the prediction and repeat for all test samples.
        """
        predictions = []
        for test_point in X_test:
            # ****** ENTER CODE  ******
            # Compute distances to all training points
            distances = []
            for train_point in self.X_train:
                dist = self._distance(test_point, train_point)
                distances.append(dist)

            # Get indices of k nearest neighbors
            distances = np.array(distances)
            k_indices = np.argsort(distances)[:self.k]

            # Get labels of k nearest neighbors
            k_labels = self.y_train[k_indices]

            # Majority voting
            unique_labels, counts = np.unique(k_labels, return_counts=True)
            prediction = unique_labels[np.argmax(counts)]
            predictions.append(prediction)
            # ****** END CODE  ******
        return np.array(predictions)
```

```
grader.check("q8")
```

The programming section of this assignment is now complete. From this point onward, you only need to answer the theory questions in the designated Markdown cells.

Do not modify any of the provided code cells beyond this point.

```
# ---------------------------
# Load dataset
# ---------------------------
X, y, feature_names = load_pima_diabetes_data()

# ---------------------------
# PCA visualization (2D)
# ---------------------------
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

plt.figure(figsize=(7, 5))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap="coolwarm", edgecolors="k", alpha=0.7)
plt.title("Pima Indians Diabetes Dataset (PCA projection)")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(label="Outcome")
plt.show()
```

Theory Question 2

Looking at the **visualization of the decision boundaries**,
which model do you think would provide a better fit among all the models we implemented from scratch?

- **Logistic Regression** .
- **Decision Tree**
- **K-Nearest Neighbors (KNN)**
- **One Class SVM (OCSVM)**

Based on the visualization, discuss which model fits the dataset best and **why**.

Explanation :

Based on the visualization, I believe **K-Nearest Neighbors (KNN)** would provide the best fit among the implemented models. The PCA projections show complex, non-linear data distributions with overlapping classes and irregular boundaries. KNN's local decision-making through neighborhood voting naturally adapts to these complex patterns, while logistic regression is constrained by its linear assumption, decision trees may overfit with brittle splits, and One-Class SVM is primarily designed for anomaly detection rather than balanced binary classification. KNN's flexibility makes it most suitable for capturing the local patterns and non-linear decision boundaries evident in this dataset.

```
def gradient_descent(theta, X, y, alpha, max_iterations):
    """ Batch gradient descent algorithm """
    iteration = 0
    prev_cost = float('inf')  # Track previous cost

    while iteration < max_iterations:
        iteration += 1
        # Update step
        gradient = gradient_update(theta, X, y)
        theta = theta - alpha * gradient

    return theta
```

```
if __name__ == "__main__":
    print("Loading Pima Indians Diabetes Dataset...")
    X, y, feature_names = load_pima_diabetes_data()
    print(f"Dataset shape: {X.shape}")
    print(f"Features: {feature_names}")
    print(f"Classes distribution: {np.bincount(y)} (0=No Diabetes, 1=Diabetes)")

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=42, stratify=y
    )

    # Initialize models
    models = {
        'Decision Tree': DecisionTreeFromScratch(max_depth=10),
        'KNN': KNNFromScratch(k=7),
        'Logistic Regression': None,  # We’ll run separately with your gradient descent
    }

    # Train and evaluate Decision Tree & KNN
    for name, model in models.items():
        if model is None:
            continue
        print(f"\nTraining {name}...")
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        acc = accuracy_score(y_test, y_pred)
        print(f"{name} Accuracy: {acc:.4f}")

    # Run Logistic Regression with Gradient Descent
    print("\nRunning Logistic Regression (from scratch)...")
    train_X_with_bias = np.column_stack([np.ones(len(X_train)), X_train])
    test_X_with_bias = np.column_stack([np.ones(len(X_test)), X_test])

    initial_theta = np.zeros(train_X_with_bias.shape[1])
    alpha_test = 0.01
    max_iter = 10000


    learned_theta = gradient_descent(initial_theta, train_X_with_bias, y_train,
                                     alpha_test, max_iter)

    y_pred_logreg = (sigmoid(np.dot(test_X_with_bias, learned_theta)) >= 0.5).astype(int)
    logreg_acc = accuracy_score(y_test, y_pred_logreg)
    print(f"Logistic Regression Accuracy: {logreg_acc:.4f}")

    # One-Class SVM (treat class 0 as "normal", detect class 1 as anomaly)
    print("\nRunning One-Class SVM...")
    X_train_normal = X_train[y_train == 0]  # only healthy patients for training
    ocsvm = KernelOneClassSVM(kernel="rbf", gamma=0.1, nu=0.05)
    ocsvm.fit(X_train_normal)

    y_pred_ocsvm = ocsvm.predict(X_test)
    # Map OCSVM output: +1 = inlier (normal), -1 = outlier (diabetes)
    y_pred_ocsvm = np.where(y_pred_ocsvm == 1, 0, 1)

    ocsvm_acc = accuracy_score(y_test, y_pred_ocsvm)
    print(f"One-Class SVM Accuracy: {ocsvm_acc:.4f}")
```

Theory Question 3

Give the **ascending order of the accuracy** values printed for you,
and explain the results.

Hint: Use the visualizations from the above cells to support your explanation.

Reflection

- Did this **match what you initially predicted** based on the plots?
- If **yes** → briefly explain why the results align with the help of visual boundaries.
- If **no** → identify which model’s performance did not match your expectation,
  and think of possible reasons and explain them

One-Class SVM (35.06%) < Logistic Regression (60.61%) < KNN (74.03%) < Decision Tree (77.06%)

Explanation:

The results show that Decision Tree performed best, followed closely by KNN. This makes sense given the medical nature of the data - features like glucose levels, BMI, and age likely have natural threshold values that decision trees can effectively capture through splits. One-Class SVM performed poorly because it was trained only on non-diabetic patients and asked to detect diabetic cases as anomalies, but the 35% diabetes prevalence is too high for effective anomaly detection.

Reflection:

Yes! This partially matches my initial prediction. I correctly anticipated that KNN would perform well due to the non-linear patterns visible in the PCA visualization, and it indeed ranked second. However, I underestimated Decision Tree's performance. The medical features in this dataset (glucose thresholds, BMI categories, age ranges) are naturally suited to tree-based splits, which explains why Decision Tree achieved the highest accuracy despite the apparent complexity in the PCA plot. My prediction about Logistic Regression being limited by linear assumptions was confirmed, as it struggled with the non-linear relationships. The One-Class SVM result aligned with expectations since it's fundamentally mismatched for this balanced classification problem.

单元格类型不受支持。双击即可检查/修改内容。