```python
# project_2_main.py

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from tqdm import tqdm
from sklearn.metrics import log_loss
from sklearn.linear_model import LogisticRegression

# Load the Wine dataset
wine = load_wine()

# Select only the data and target corresponding to the first two classes
class_indices = (wine.target == 0) | (wine.target == 1)
X_binary = wine.data[class_indices]
y_binary = wine.target[class_indices]

# Split the data into training and testing sets with stratification
X_train, X_test, y_train, y_test = train_test_split(X_binary, y_binary,
test_size=0.2, random_state=42, stratify=y_binary)

# Logistic function (sigmoid)
def logistic_function(x):
    exp_values = np.exp(-np.clip(x, -500, 500))  # Clip values to avoid overflow
    return 1 / (1 + exp_values)

# Number of runs
num_runs = 10000

# Number of iterations
num_iterations = 100

# Common seed for random initialization
common_seed = 42
np.random.seed(common_seed)

# Pre-generate initial weights for all runs
all_initial_weights = np.random.uniform(low=-10, high=10, size=(num_runs,
X_train.shape[1]))

# Memory-Aware Coordinate Descent
def memory_aware_coordinate_descent(X, y, initial_weights, max_iter=1000,
learning_rate=0.1, line_search=True, random_state=None):
    num_runs, num_features = initial_weights.shape
```

```
    X_normalized = (X - X.mean(axis=0)) / X.std(axis=0)
    loss_values = np.zeros((max_iter + 1, num_runs))

    # Initialize memory for gradients
    gradient_memory = np.zeros(num_features)

    for run in tqdm(range(num_runs), desc='Memory-Aware Coordinate Descent
Runs'):
        np.random.seed(random_state + run)  # Set seed for reproducibility
        w = np.copy(initial_weights[run])

        # Compute initial loss
        y_pred_proba = logistic_function(np.dot(X_normalized, w))  # Use
normalized X
        loss_values[0, run] = log_loss(y, y_pred_proba)

        for iteration in tqdm(range(1, max_iter + 1), desc='Iterations',
leave=False):
            # Check if there's an empty slot in the memory vector
            empty_slot = np.any(gradient_memory == 0)

            if empty_slot:
                # Find the first empty slot in the memory vector
                coordinate = np.argmax(gradient_memory == 0)
            else:
                # Choose the coordinate with the largest magnitude gradient in
recent memory
                coordinate = np.argmax(np.abs(gradient_memory))

            # Compute gradient for the selected coordinate
            gradient = -(y - logistic_function(np.dot(X_normalized, w))) @
X_normalized[:, coordinate]

            # Update memory for the selected coordinate
            gradient_memory[coordinate] = gradient

            # Backtracking line search
            if line_search:
                step_size = backtracking_line_search(X_normalized, y, w,
gradient_memory, coordinate)
            else:
                step_size = learning_rate

            # Update the weight for the selected coordinate
            w[coordinate] -= step_size * gradient_memory[coordinate]
```

```
            # Compute and store the loss for the current iteration
            y_pred_proba = logistic_function(np.dot(X_normalized, w))
            loss_values[iteration, run] = log_loss(y, y_pred_proba)

    return np.mean(loss_values, axis=1), np.std(loss_values, axis=1)

# Random-Feature Coordinate Descent
def random_feature_coordinate_descent(X, y, initial_weights, max_iter=1000,
learning_rate=0.1, line_search=True, random_state=None):
    num_runs, num_features = initial_weights.shape
    _, d = X.shape
    X_normalized = (X - X.mean(axis=0)) / X.std(axis=0)
    loss_values = np.zeros((max_iter + 1, num_runs))

    for run in tqdm(range(num_runs), desc="Random-Feature Coordinate Descent
Runs"):
        np.random.seed(random_state + run)  # Set seed for reproducibility
        w = np.copy(initial_weights[run])  # Use specific initial weights for
each run
        loss_values[0, run] = log_loss(y, logistic_function(np.dot(X_normalized,
w)))

        for iteration in tqdm(range(1, max_iter + 1), desc='Iterations',
leave=False):
            coordinate = np.random.randint(0, d)  # Choose a coordinate uniformly
at random
            gradients = np.zeros(num_features)
            gradient_i = -(y - logistic_function(np.dot(X_normalized, w))) @
X_normalized[:, coordinate]
            gradients[coordinate] = gradient_i

            # Backtracking line search
            if line_search:
                step_size = backtracking_line_search(X_normalized, y, w,
gradients, coordinate)
            else:
                step_size = learning_rate

            w[coordinate] -= step_size * gradients[coordinate]

            y_pred_proba = logistic_function(np.dot(X_normalized, w))
            loss_values[iteration, run] = log_loss(y, y_pred_proba)

    return np.mean(loss_values, axis=1), np.std(loss_values, axis=1)
```

```
# Backtracking Line Search
def backtracking_line_search(X, y, w, gradients, coordinate, beta=0.8):
    step_size = 1.0
    c = 1e-4  # A small constant

    while True:
        new_w = np.copy(w)
        new_w[coordinate] -= step_size * gradients[coordinate]
        y_pred_proba = logistic_function(np.dot(X, new_w))
        new_loss = log_loss(y, y_pred_proba)
        expected_reduction = c * step_size * np.dot(gradients, gradients)

        if new_loss <= log_loss(y, logistic_function(np.dot(X, w))) -
expected_reduction:
            break

        step_size *= beta

    return step_size

# Run scikit-learn Logistic Regression
sklearn_lr = LogisticRegression(max_iter=100000)
sklearn_lr.fit(X_train, y_train)
y_pred_proba_sklearn = sklearn_lr.predict_proba(X_train)[:, 1]
log_loss_sklearn = log_loss(y_train, y_pred_proba_sklearn)

# Run Greedy Coordinate Descent with Line Search
greedy_mean_loss, greedy_std_loss = memory_aware_coordinate_descent(X_train,
y_train, all_initial_weights, max_iter=num_iterations, learning_rate=0.1,
line_search=False, random_state=common_seed)

# Run Random-Feature Coordinate Descent
random_mean_loss, random_std_loss = random_feature_coordinate_descent(X_train,
y_train, all_initial_weights, max_iter=num_iterations, learning_rate=0.1,
line_search=False, random_state=common_seed)

# Plot Loss Curves with Mean and Standard Deviation
iterations = range(num_iterations + 1)  # Including the first point

# Greedy Coordinate Descent
plt.plot(iterations, greedy_mean_loss, label='Memory-Aware Coordinate Descent')
plt.fill_between(iterations, greedy_mean_loss - greedy_std_loss, greedy_mean_loss
+ greedy_std_loss, alpha=0.2)
```

```python
# Random-Feature Coordinate Descent
plt.plot(iterations, random_mean_loss, label='Random-Feature Coordinate Descent')
plt.fill_between(iterations, random_mean_loss - random_std_loss, random_mean_loss
+ random_std_loss, alpha=0.2)

# Plot horizontal dotted line for scikit-learn Logistic Regression minimum loss
plt.axhline(log_loss_sklearn, color='r', linestyle='--', label='scikit-learn
Logistic Regression Loss')

plt.xlabel('Iteration')
plt.ylabel('Loss L(w_t)')
plt.title('Loss Curves for Coordinate Descent Methods')
plt.legend()
plt.savefig('project_2_main.png')
plt.show()
```

```python
# project_2_sparse.py

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from tqdm import tqdm
from sklearn.metrics import log_loss
from sklearn.linear_model import LogisticRegression

# Load the Wine dataset
wine = load_wine()

# Select only the data and target corresponding to the first two classes
class_indices = (wine.target == 0) | (wine.target == 1)
X_binary = wine.data[class_indices]
y_binary = wine.target[class_indices]

# Split the data into training and testing sets with stratification
X_train, X_test, y_train, y_test = train_test_split(X_binary, y_binary,
test_size=0.2, random_state=42, stratify=y_binary)

# Logistic function (sigmoid)
def logistic_function(x):
    exp_values = np.exp(-np.clip(x, -500, 500))  # Clip values to avoid overflow
    return 1 / (1 + exp_values)

# Number of runs
num_runs = 10000

# Number of iterations
num_iterations = 15

# Common seed for random initialization
common_seed = 42
np.random.seed(common_seed)

# Pre-generate initial weights for all runs
all_initial_weights = np.random.uniform(low=-0.1, high=0.1, size=(num_runs,
X_train.shape[1]))

# Memory-Aware Coordinate Descent
def memory_aware_coordinate_descent(X, y, initial_weights, k, max_iter=1000,
learning_rate=0.1, line_search=True, random_state=None):
    num_runs, num_features = initial_weights.shape
```

```
    X_normalized = (X - X.mean(axis=0)) / X.std(axis=0)
    loss_values = np.zeros((max_iter + 1, num_runs))

    # Initialize memory for gradients
    gradient_memory = np.zeros(num_features)

    for run in tqdm(range(num_runs), desc=f'Memory-Aware Coordinate Descent Runs
k={k}'):
        np.random.seed(random_state + run)  # Set seed for reproducibility
        w = np.copy(initial_weights[run])

        # Compute initial loss
        k_largest_indices = np.argpartition(np.abs(w), -k)[-k:]
        k_sparse_w = np.zeros_like(w)
        k_sparse_w[k_largest_indices] = w[k_largest_indices]

        y_pred_proba = logistic_function(np.dot(X_normalized, k_sparse_w))  # Use
normalized X
        loss_values[0, run] = log_loss(y, y_pred_proba)

        for iteration in tqdm(range(1, max_iter + 1), desc='Iterations',
leave=False):
            # Check if there's an empty slot in the memory vector
            empty_slots = np.where(gradient_memory == 0)[0]

            if len(empty_slots) >= k:
                # Choose k coordinates randomly from empty slots
                coordinates = np.random.choice(empty_slots, size=k,
replace=False)
            else:
                # Choose the top k coordinates with the largest magnitude
gradient in recent memory
                coordinates = np.argsort(np.abs(gradient_memory))[::-1][:k]

            # Compute gradients for the selected coordinates
            gradients = -(y - logistic_function(np.dot(X_normalized, w))) @
X_normalized[:, coordinates]

            # Update memory for the selected coordinates
            gradient_memory[coordinates] = gradients

            # Backtracking line search
            if line_search:
                step_size = backtracking_line_search(X_normalized, y, w,
gradient_memory, coordinates)
```

```
            else:
                step_size = learning_rate

            # Update the weights for the selected coordinates
            w[coordinates] -= step_size * gradient_memory[coordinates]

            # Compute and store the loss for the current iteration on the k-
sparse vector
            k_largest_indices = np.argpartition(np.abs(w), -k)[-k:]
            k_sparse_w = np.zeros_like(w)
            k_sparse_w[k_largest_indices] = w[k_largest_indices]

            y_pred_proba = logistic_function(np.dot(X_normalized, k_sparse_w))
            loss_values[iteration, run] = log_loss(y, y_pred_proba)

    return np.mean(loss_values, axis=1), np.std(loss_values, axis=1)

# Backtracking Line Search
def backtracking_line_search(X, y, w, gradients, coordinate, beta=0.8):
    step_size = 1.0
    c = 1e-4  # A small constant

    while True:
        new_w = np.copy(w)
        new_w[coordinate] -= step_size * gradients[coordinate]
        y_pred_proba = logistic_function(np.dot(X, new_w))
        new_loss = log_loss(y, y_pred_proba)
        expected_reduction = c * step_size * np.dot(gradients, gradients)

        if new_loss <= log_loss(y, logistic_function(np.dot(X, w))) -
expected_reduction:
            break

        step_size *= beta

    return step_size

# Run scikit-learn Logistic Regression
sklearn_lr = LogisticRegression(max_iter=100000)
sklearn_lr.fit(X_train, y_train)
y_pred_proba_sklearn = sklearn_lr.predict_proba(X_train)[:, 1]
log_loss_sklearn = log_loss(y_train, y_pred_proba_sklearn)

# Values of k to test
k_values = [1, 3, 7, 10, 13]
```

```python
# Results storage
k_sparse_losses = {}

# Run k-sparse Momentum-based Coordinate Descent for each k value
final_losses = []
for k in k_values:
    mean_loss, std_loss = memory_aware_coordinate_descent(X_train, y_train,
all_initial_weights, k, max_iter=num_iterations, learning_rate=0.1,
line_search=False, random_state=common_seed)
    k_sparse_losses[k] = (mean_loss, std_loss)

    final_loss = mean_loss[-1]
    final_losses.append(final_loss)

# Print the final losses for each k
for k, final_loss in zip(k_values, final_losses):
    print(f"Final loss for k={k}: {final_loss}")

# Plot Loss Curves with Mean and Standard Deviation for different k values
iterations = range(num_iterations + 1)  # Including the first point

plt.figure(figsize=(10, 6))
for k in k_values:
    if k == 13:
        label = 'No Sparsity'
    else:
        label = f'k = {k}'
    plt.plot(iterations, k_sparse_losses[k][0], label=label)
    plt.fill_between(iterations, k_sparse_losses[k][0] - k_sparse_losses[k][1],
k_sparse_losses[k][0] + k_sparse_losses[k][1], alpha=0.2)

# Plot horizontal dotted line for scikit-learn Logistic Regression minimum loss
plt.axhline(log_loss_sklearn, color='r', linestyle='--', label='scikit-learn
Logistic Regression Loss')

plt.xlabel('Iteration')
plt.ylabel('Loss L(w_t)')
plt.title('Loss Curves for K-sparse Momentum-based Coordinate Descent')
plt.legend()
plt.savefig('project_2_k_sparse.png')
plt.show()
```