# 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)

# 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()

# 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]

# 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()