Coordinate Descent Methods for Logistic Regression: A Comparative Study on the Wine Dataset

Allen Zou

February 26, 2024

Abstract

Logistic regression is a widely employed method for solving binary classification problems, often necessitating iterative optimization techniques. This study explores the application of coordinate descent methods to logistic regression, with a specific focus on the Memory-Aware Coordinate Descent algorithm. A comparative analysis is conducted with a Random-Feature Coordinate Descent approach, examining convergence properties. Experimental results are presented to showcase the efficiency and behavior of the Memory-Aware Coordinate Descent algorithm in comparison to Random-Feature Coordinate Descent. Additionally, we extend the Memory-Aware Coordinate Descent algorithm into a k-sparse setting and assess its performance across varying k values.

1 Introduction

Logistic regression serves as a foundational technique for binary classification, often requiring iterative optimization strategies for model refinement. This study delves into coordinate descent methods, with a specific focus on the application of the Memory-Aware Coordinate Descent algorithm to logistic regression. As an alternative to traditional optimization techniques, the simplicity of coordinate descent warrants a comprehensive investigation.

2 Coordinate Descent Strategies

Coordinate descent involves updating one variable of the weight vector at a time while holding others fixed. In the context of logistic regression, the goal is to iteratively adjust the coefficients to minimize the cost function. This study focuses on two coordinate descent algorithms: Random-Feature Coordinate Descent and Memory-Aware Coordinate Descent.

Random-Feature Coordinate Descent merely updates a randomly chosen feature in each iteration with the negative gradient. Memory-Aware Coordinate Descent, however, maintains a memory vector of previously computed partial gradient values. It selects a coordinate based on which partial gradient has the highest absolute magnitude and updates that memory vector accordingly.

2.1 Pseudocode

Algorithm 1 Random-Feature Coordinate Descent

```
Require: Initial weights $w$, Dataset $X$, Labels $y$,
    Learning rate $\gamma$, Number of iterations $T$
 1: for $t = 1$ to $T$ do
 2:     $i \leftarrow$ Randomly choose from $[0, d)$
 3:     $dw_i \leftarrow$ Compute $\frac{\partial L}{\partial w_i}$ using $w_i$
 4:     $w_i \leftarrow w_i - \gamma \cdot dw_i$
 5: end for
 6: return $w$
```

Algorithm 2 Memory-Aware Coordinate Descent

```
Require: Initial weights $w$, Dataset $X$, Labels $y$,
    Learning rate $\gamma$, Number of iterations $T$
 1: Initialize $m$ to all 0                ▷ Memory vector
 2: for $t = 1$ to $T$ do
 3:     if $m$ has an empty slot then
 4:         $i \leftarrow$ first empty coordinate
 5:     else
 6:         $i \leftarrow$ argmax(abs($m$))
 7:     end if
 8:     $dw_i \leftarrow$ Compute $\frac{\partial L}{\partial w_i}$
 9:     $w_i \leftarrow w_i - \gamma \cdot dw_i$
10: end for
11: return $w$
```

2.2 Assumptions About Convergence

The coordinate descent algorithms presented here assume the differentiability and convexity of the loss function $L(w)$. The differentiability of the loss function is essential for computing gradients, which are necessary for weight updates in each iteration. Additionally, the convexity property ensures the well-behaved nature of the optimization problem associated with logistic regression on the wine dataset.

In the context of logistic regression, the convex logistic loss function is characterized by a single global minimum. The algorithms, including Random-Feature Coordinate Descent and Memory-Aware Coordinate Descent, rely on the convexity of the loss function to efficiently find the global minimum. Memory-Aware Coordinate Descent, in particular, maintains a memory vector to enhance convergence by selecting coordinates based on the highest absolute magnitude of the partial gradient.

It's important to note that these assumptions may be influenced by the choice of dataset and specific characteristics of the logistic loss function. While these conditions hold for logistic regression on the wine dataset, the convergence behavior could vary in the presence of non-convex loss functions or datasets with different properties.

3 Experimental Setup

For this study, the Wine dataset was employed, initially consisting of 178 data points across 13 dimensions and belonging to 3 classes. To create a binary classification problem, the dataset was narrowed down to include only the first two classes, resulting in 130 data points—59 from the first class and 71 from the second. An 8:2 split was applied for training and testing.

Both Memory-Aware and Random-Feature Coordinate Descent algorithms were executed with 100 runs each, where each run encompassed 50 iterations. The choice of a relatively low number of iterations was motivated by the observed rapid convergence of the algorithms. The resulting plots showcase the average function evaluations over iterations, providing a robust overview of algorithm behavior. The only hyper-parameter considered for coordinate descent was the learning rate, which was set to 0.1 to ensure meaningful comparisons.

4 Experimental Results

We present experimental results for the Memory-Aware and Random-Feature Coordinate Descent algorithms. The analysis focuses on the first two classes of the Wine dataset, where the mean function evaluation and associated variability for each iteration are depicted in the figure below. The shaded regions around each line represent one standard deviation from the mean function evaluation. For comparison, the loss from scikit-learn's standard logistic regression solver is plotted as a dotted line on the same graph, providing a benchmark for evaluating the performance of the coordinate descent algorithms.

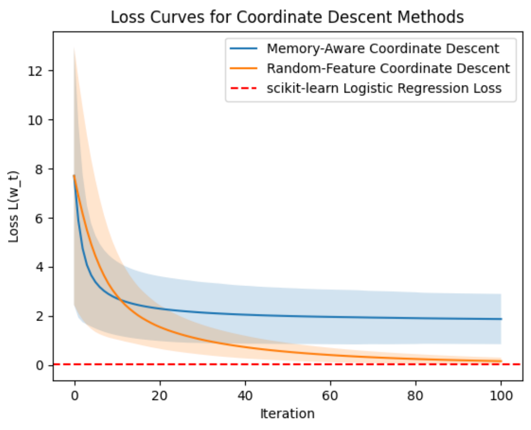

Figure 1: Comparison Between Memory-Aware Coordinate Descent and Random-Feature Coordinate Descent

5 Discussion

In this section, we analyze and interpret the experimental results obtained from the Memory-Aware and Random-Feature Coordinate Descent algorithms applied to logistic regression on the first two classes of the Wine dataset.

Contrary to our initial expectations, the Memory-Aware Coordinate Descent strategy did not outperform Random-Feature Coordinate Descent in terms of the final loss value. A small thing to note is that the Memory-Aware Coordinate Descent method exhibited slightly faster convergence in the initial iterations compared to Random-Feature Coordinate Descent. However, the latter eventually caught up, suggesting that the advantage in early convergence is not sustained.

Interestingly, the variance of the Memory-Aware Coordinate Descent method was found to be larger than that of Random-Feature Coordinate Descent. Additionally, around the points of convergence, the variance almost disappears for the Random-Feature method, indicating a more consistent convergence to a specific location. In contrast, the Memory-Aware Coordinate Descent method retained a large level of variance even after convergence, suggesting that it does not always converge to the same location.

These unexpected findings have implications for use of the Memory-Aware Coordinate Descent strategy in logistic regression. While it may exhibit faster convergence in the early stages, its tendency to have larger variance and not consistently converge to the same location is unusual. The presence of variance in the Memory-Aware Coordinate Descent, even after convergence, seems counterintuitive to the assumption of convexity.

Convex optimization problems are generally expected to converge to a unique minimum due to the well-behaved nature of convex functions. Further investigation into the behavior of the Memory-Aware Coordinate Descent, considering the convexity assumption, may be necessary to reconcile this discrepancy. Fine-tuning of hyperparameters and a more in-depth exploration of the optimization landscape could shed light on the observed behavior.

6 K-sparse Coordinate Descent

In this section, we extend our analysis to explore the performance of the Memory-Aware Coordinate Descent algorithm under sparsity constraints. The algorithm is modified to enforce a k-sparse solution, where the weight vector has at most k nonzero entries. We investigate the impact of varying sparsity levels on convergence behavior and final loss values.

6.1 Experimental Setup

For the k-sparse experiments, we utilized the same Wine dataset with binary classification between the first two classes as in the previous section. The learning rate (0.1) remained consistent with our earlier experiments.

The key modifications include a decrease in the number of iterations to 15 per run, adapting to the faster convergence behavior observed in the k-sparse setting. Unlike the previous experiments, where a single coordinate was updated in each iteration, the k-sparse experiments involved selecting the k coordinates with the largest partial derivative magnitudes recorded so far and updating them with the Memory-Aware Coordinate Descent algorithm.

Additionally, the method for evaluating and plotting values entailed the following steps: after each iteration of updating the weight vector, we extracted its k-sparse version by choosing the k largest partial derivatives and setting the rest of the entries to 0. The loss was subsequently computed based on this k-sparse variant of the current weight vector. This approach allowed us to gain insights into the model's behavior throughout the optimization process.

We conducted experiments for various sparsity levels, specifically testing for k values of 1, 3, 7, 10, and 13, to explore the effects of different degrees of sparsity on the convergence behavior.

6.2 Experimental Results

In this section, we present experimental results for the Memory-Aware and Random-Feature Coordinate Descent algorithms applied to logistic regression under various k-sparsity values (k = 1, 3, 7, 10, and 13). As in the previous experiments, the figure below illustrates the mean function evaluation along with the associated variability for each iteration. The shaded regions surrounding each line indicate one standard deviation from the mean function evaluation.

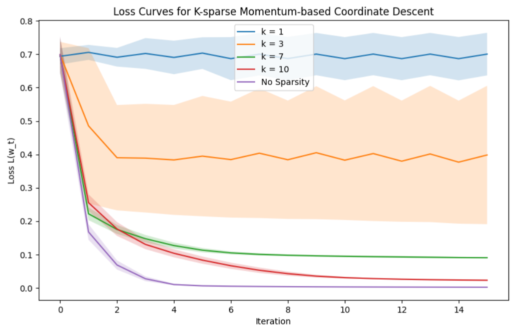

Figure 2: K-sparse Memory-Aware Coordinate Descent for Different k Values

k Value	Final Loss
1	0.046272
3	0.038545
7	0.064082
10	0.106584
13	0.107748

Table 1: Final Loss for Different k Values in k-sparse Coordinate Descent

6.3 Discussion

In the k-sparse setting, it is evident that all runs converge significantly faster, with the method for extracting the k-sparse gradient involving choosing the k largest partial derivatives and setting the rest of the entries to 0.

There is also an intricate relationship between the sparsity level (k), convergence speed, and final loss values. Contrary to the initial assumption, higher values of k do not consistently lead to faster convergence. Notably, the value of k = 10 seems to converge noticeably slower than the rest, challenging expectations. Nevertheless, a higher value of k does lead to a lower final loss value, aligning with the anticipated trend.

Interestingly, k = 1 and k = 3 struggle with convergence, with these cases never converging to a single loss value but oscillating around a value. Strangely, k = 3 exhibits significant variance compared to all the other cases, even k = 1. However, as soon as we hit k = 7, the loss improves dramatically, and there is not a substantial difference between the k = 7, k = 10, and k = 13 cases. It is intriguing to note that, for values of k = 10 and k = 13, they strangely lead to lower loss values than the LogisticRegression model in scikit-learn, introducing a noteworthy observation that warrants further investigation into the algorithm's behavior. Specifically, we should scrutinize the implementation details of the code to ensure the observed trends are not influenced by implementation artifacts.

7 Limitations

While our study focuses on a binary classification setting, one avenue for future research involves extending our methodology to scenarios with a higher number of classes. The decision to restrict our analysis to classes 1 and 2 was made to simplify the problem for coordinate descent on logistic regression, resulting in 59 and 71 points, respectively. Recognizing the potential impact of training on a limited number of points, it is essential to acknowledge that generalization may be constrained. Future work should explore datasets with more extensive class distributions to assess the scalability and robustness of the proposed Memory-Aware Coordinate Descent method. These include investigating the algorithm's performance under diverse datasets, addressing potential challenges associated with larger class distributions, and exploring extensions of the methodology to multi-class classification settings.

8 Conclusion

In this study, we delved into the application of coordinate descent methods, with a specific emphasis on the Memory-Aware Coordinate Descent algorithm in logistic regression optimization. Our comparative analysis included a Random-Feature Coordinate Descent approach, providing insights into the convergence properties of these methods.

The experimental results revealed nuanced dynamics in the performance of the Memory-Aware Coordinate Descent algorithm. While it exhibited slightly faster convergence in the initial iterations compared to Random-Feature Coordinate Descent, its overall performance, in terms of the final loss value, did not surpass that of Random-Feature Coordinate Descent. The presence of variance in the Memory-Aware method, even after convergence, raises interesting questions about its behavior in relation to the convexity assumption.

Furthermore, our exploration extended to the k-sparse setting, where the Memory-Aware Coordinate Descent algorithm was modified to enforce sparsity constraints. The results revealed intriguing dynamics in the relationship between sparsity levels (k) and optimization outcomes. A higher value of k did contribute to achieving a lower final loss value, aligning with the anticipated trade-off. However, higher values of k did not consistently lead to faster convergence. The unexpected trends observed in k-sparse experiments prompt further investigation into the algorithm's behavior under sparsity constraints.

As we conclude, it's essential to acknowledge the limitations of our study, including the binary classification setting and the relatively limited number of data points. Future research could extend our methodology to scenarios with a higher number of classes and explore datasets with more extensive class distributions to enhance generalizability. Additionally, further investigation into the implementation details is warranted to scrutinize potential factors contributing to the observed dynamics and unexpected behaviors.

Looking ahead, I anticipate further exploration across diverse datasets, conducting more parameter tuning, and personally delving into alternative selection strategies. These efforts are intended to refine and broaden the applicability of coordinate descent methods, addressing the limitations identified in this study.

```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)
```

```python
# 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()
```

```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)
```

```python
            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()
```