Project 2 Write-up

Haotian Xu
hax030@ucsd.edu

Abstract

In this project, we explore how well different coordinate descent methods work for solving binary logistic regression, a common type of problem in machine learning for classifying data into two categories. We compare three methods: Random-feature Coordinate Descent (RCD), Greedy Coordinate Descent (GCD), and Cyclic Coordinate Descent (CCD). Our goal is to see how each method chooses coordinates and updates them, and then test to see which one performs best through experiments.

1 Algorithm Description

The three coordinate descent methods we selected all choose one coordinate per iteration. We will explain how the weight is updated and how our methods select coordinates in the following parts.

1.1 Weight Initializing and Updating

We use the same weight initializing and updating strategy in all of the three coordinate descent methods. We initialize the weight as a vector of zeros that has the same size as the number of features. In each iteration, we update the new weight w_{t+1} for the chosen coordinates j using equations below:

$$w_{t+1} = w_t - \eta \nabla L_j(w) \tag{1}$$

where the loss function is

$$L(w) = \sum_{i=1}^{n} \ln(1 + e^{-y_i \cdot (wx_i)}) \tag{2}$$

and the gradient of the loss function for the coordinates j is

$$\nabla L_j(w) = \sum_{i=1}^{n} \left(-\frac{y_i \cdot x_{i,j}}{1 + e^{y_i \cdot (wx_i)}} \right) \tag{3}$$

and η is the learning rate or step size. In our project, we will set it as a fixed value for all methods in the experiments.

Our methods rely on using the gradient to update weights, which requires the loss function to be differentiable. Additionally, we need the loss function to be convex to ensure that our updates lead us to a global minimum. The loss function we selected is both differentiable and convex, making it suitable for our approach. However, this method may not be effective for cost functions that are either non-differentiable or non-convex, as these characteristics could prevent us from finding the global minimum.

1.2 Random-feature Coordinate Descent

In the Random-feature Coordinate Descent method, we randomly choose one coordinate and update the weight using it. This method will be our baseline method. Below is the pseudocode corresponding to this method:

Algorithm 1 Random-feature Coordinate Descent

```
1: Input: $X_{\text{train}}, y_{\text{train}}, \eta$
2: Output: Trained weights $w$, Losses per iteration
3: Initialize weight vector $w = \mathbf{0}$
4: Initialize an empty list for losses $L = []$
5: while not converged or iteration limit not reached do
6:     Select a coordinate $j$ randomly
7:     Compute gradient $\nabla_j L(w) = \frac{\partial L}{\partial w_j}$
8:     Update $w_j$ by $w_j = w_j - \eta \nabla_j L(w)$
9:     Compute current loss $L_{\text{current}}$ over $X_{\text{train}}$
10:    Append $L_{\text{current}}$ to $L$
11: end while
12: return $w, L$
```

1.3 Greedy Coordinate Descent

In this method (Schmidt, 2022), we choose the coordinate with the maximum gradient value and use it to update weights. We use this method with the assumption that choosing the gradient that updates the weights most will lead to global optimum quicker and closer. In order to avoid the cost of computing gradients for every coordinate in each iteration, we employ a heap to maintain the order of gradient values. Below is the pseudocode for this method:

Algorithm 2 Greedy Coordinate Descent

```
1: Input: $X_{\text{train}}, y_{\text{train}}, \eta$
2: Output: Trained weights $w$, Losses per iteration
3: Initialize weight vector $w = \mathbf{0}$
4: Initialize an empty list for losses $L = []$
5: Compute full gradient $\nabla L(w)$ for all features
6: Create a max heap $H$ of $(-|\nabla_j L(w)|, j)$ for all features $j$
7: while not converged or iteration limit not reached do
8:     Pop $(\_, j)$ from $H$ with the largest $|\nabla_j L(w)|$
9:     Compute gradient $\nabla_j L(w)$ for coordinate $j$
10:    Update $w_j$ by $w_j = w_j - \eta \nabla_j L(w)$
11:    Compute new gradient $\nabla_j L(w)$ for updated $w_j$
12:    Push $(-|\nabla_j L(w)|, j)$ back into $H$
13:    Compute current loss $L_{\text{current}}$ over $X_{\text{train}}$
14:    Append $L_{\text{current}}$ to $L$
15: end while
16: return $w, L$
```

1.4 Cyclic Coordinate Descent

In the Cyclic Coordinate Descent (Schmidt, 2022) (Wang and Chen, 1991) method, we systematically update each coordinate in sequence. Starting with the first coordinate, we move to the next one with each iteration. Once we reach the last coordinate, we loop back to the first and repeat the process. This cycle continues until the optimization criteria are met. This method ensures that every coordinate has an equal chance of being selected. We assume this characteristic can help avoid potential information lost in the random choice method.

Algorithm 3 Cyclic Coordinate Descent

```
1: Input: $X_{\text{train}}, y_{\text{train}}, \eta$
2: Output: Trained weights $w$, Losses per iteration
3: Initialize weight vector $w = \mathbf{0}$
4: Initialize an empty list for losses $L = []$
5: Initialize coordinate index $j = 0$
6: while not converged or iteration limit not reached do
7:     Compute gradient $\nabla_j L(w)$ for current coordinate $j$
8:     Update $w_j$ by $w_j = w_j - \eta \nabla_j L(w)$
9:     Compute current loss $L_{\text{current}}$ over $X_{\text{train}}$
10:    Append $L_{\text{current}}$ to $L$
11:    $j = (j + 1) \mod$ number of features
12: end while
13: return $w, L$
```

2 Convergence

For our methods to successfully converge to the optimal loss, a few key conditions must be satisfied. Firstly, our loss function should be differentiable. We need this condition to ensure that we can compute gradients, which guide the update to the weights during the optimization process. Additionally, the loss function should also be convex. The convexity of the loss function makes sure that we can find the global optimum using the gradient method in logistic regression. Finally, the learning rate and the iteration limit should be chosen carefully. If the learning rate is too large, we may jump across the global optimum. If the number of iterations is not sufficient, our methods may not be able to reach the global minimum.

3 Experiment and Results

3.1 Data Preprocessing

We evaluate the coordinate descent methods on the UCI wine dataset. This dataset originally includes three classes, but we exclude the third class to turn it into a dataset for binary classification. We adjust the labels to 1 and -1 to fit our chosen loss function in (2). Additionally, we normalize the dataset to ensure all features are on the same scale.

3.2 Unregularized Logistic Regression

Before applying our coordinate descent methods, we start by training a basic logistic regression model without any regularization, using the scikit-learn library. We then retrieve the weights from this model and calculate the loss with our specified loss function. The calculated final loss, L^*, comes out to be 2.793×10^{-4}. This value serves as a benchmark to gauge the effectiveness of our coordinate descent methods.

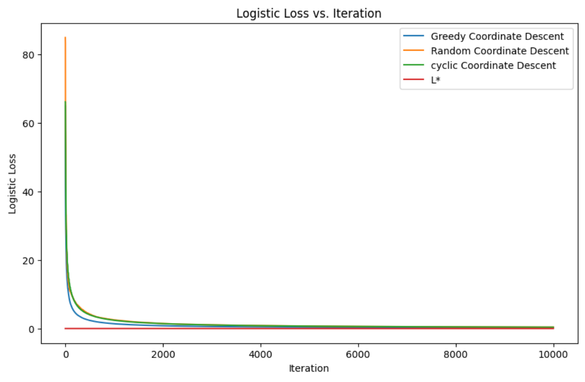


Figure 1: Logistic Loss vs. Iteration (t) with 10,000 Iterations

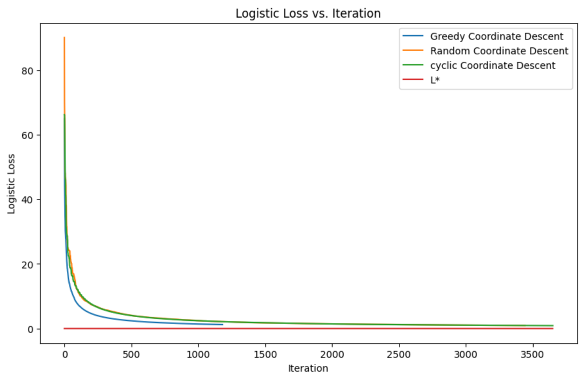


Figure 2: Logistic Loss vs. Iteration (t) with Early Stopping

3.3 Results

We conducted two experiments. First, we ran all three methods for 10,000 iterations with a learning rate of 0.01. Since each method updates one coordinate at a time, comparing their losses over the same number of iterations is fair. Figure 1 displays how the loss changes over time for each method.

The graph shows that Random-feature Coordinate Descent and Cyclic Coordinate Descent perform similarly, but Greedy Coordinate Descent reduces the loss much quicker in the first 500 iterations. The final losses of all methods are close to the benchmark loss L^* from standard logistic regression. The graph also proves that our methods converge to the optimal loss. To minimize the effect of randomness, we repeated the experiments and calculated the average run time and final loss for each method, as shown in Table 1.

Greedy Coordinate Descent achieves a better final loss but takes longer to run. Cyclic Coordinate Descent and Random-feature Coordinate Descent have similar final losses, but Cyclic runs quicker.

In our second experiment, we introduced early stopping based on a tolerance λ, stopping the iteration if the loss change is less than λ for over 100 consecutive iterations.

$$|L_{old} - L_{new}| < \lambda \tag{4}$$

Using the same tolerance $\lambda = 10^{-3}$ for all methods, we found that Greedy Coordinate Descent stops earlier but with a higher loss as shown in Figure 2 and Table 2. Cyclic Coordinate Descent shows a slight improvement in both run time and loss compared to the Random method. All three methods save iteration counts dramatically.

3.4 Analysis

The Greedy Coordinate Descent's quick loss reduction confirms our expectations, and its longer runtime is seen as reasonable. This method effectively identifies impactful updates, trading off slightly higher computational time for faster convergence.

Additionally, the similar performance of Cyclic and Random-feature Coordinate Descent in terms of final loss, with Cyclic Coordinate Descent being faster, suggests systematic cycling through coordinates can be as effective as random selection but more efficient.

In the second experiment, Greedy Coordinate Descent's early stopping with higher loss indicates it may converge too quickly, risking premature optimization. Meanwhile, Cyclic and Random methods provide a balanced approach, potentially avoiding such early convergence. This analysis sets the stage for further evaluation of each method's trade-offs and effectiveness.

4 Critical Evaluation

In our evaluation, we've identified areas for enhancement within our approach. A notable issue is Greedy Coordinate Descent's premature stopping, leading to larger losses. This suggests the potential for refining our early-stopping criteria to prevent too-early termination. Additionally, exploring alternative or dynamic adjustments to the learning rate could offer improvements, as our current setup relies on a constant learning rate throughout all tests.

Moreover, our current methods are limited to differentiable and convex loss functions, which may not encompass a wide range of real-world scenarios that involve non-differentiable or non-

Table 1: Average Run Times and Losses for Coordinate Descent Methods

Method	Average Run Time (Secs)	Average Loss	Iterations Count
Random-feature Coordinate Descent	1.043	0.401	10000
Greedy Coordinate Descent	1.462	0.211	10000
Cyclic Coordinate Descent	0.848	0.397	10000

Table 2: Average Run Times and Losses for Coordinate Descent Methods with Early Stopping

Method	Average Run Time (Secs)	Average Loss	Iterations Count
Random-feature Coordinate Descent	0.204	0.923	3441
Greedy Coordinate Descent	0.063	1.232	1181
Cyclic Coordinate Descent	0.136	0.899	3649

convex objectives. To address more complex problems, investigating more sophisticated coordinate descent techniques, like Block Coordinate Descent and Adaptive Coordinate Descent, might be beneficial. These advanced methods could provide the flexibility and robustness needed to tackle a broader spectrum of optimization challenges.

References

Mark Schmidt. 2022. Coordinate optimization and stochastic gradient descent. `https://www.cs.ubc.ca/~schmidtm/Courses/5XX-S22/S3.pdf`. Accessed 26 Feb. 2024.

L.-C.T. Wang and C.C. Chen. 1991. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *IEEE Transactions on Robotics and Automation*, 7(4):489–499.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import time
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_wine
from sklearn.metrics import log_loss
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
import heapq
```

In [2]:

```
X, y = load_wine(return_X_y=True)
indices = y < 2
X, y = X[indices], y[indices]
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
y[y==0] = -1
```

In [3]:

```
lr = LogisticRegression(penalty=None, fit_intercept=False)
lr.fit(X_scaled, y)
```

Out[3]:

```
▾                    LogisticRegression
LogisticRegression(fit_intercept=False, penalty=None)
```

In [4]:

```
def compute_loss(X, y, w):
  z = np.dot(w.T, X.T)
  loss = np.sum(np.log(1 + np.exp(-y * z)))
  return loss
```

In [5]:

```
w_lr = lr.coef_[0]
L_star = compute_loss(X_scaled, y, w_lr)
print(L_star)
```

```
0.0002793446179289648
```

In [6]:

```
def compute_gradient(X, y, w, j):
  z = np.dot(w.T, X.T)
  gradients = y * X[:, j] / (1 + np.exp(y * z))
  gradient = np.sum(gradients)
  return gradient

def random_coordinate_descent(X, y, learn_rate):
  d = X.shape[1]
  w = np.zeros(d)
  losses = []
  max_iter = 10000
  iter = 0
  while iter < max_iter:
    coord = np.random.choice(X.shape[1])
    grad = compute_gradient(X, y, w, coord)
    w[coord] += learn_rate * grad
    loss = compute_loss(X, y, w)
    losses.append(loss)
    iter += 1
  return w, losses, iter

def random_coordinate_descent_auto_stop(X, y, learn_rate, tol = 1e-3):
  d = X.shape[1]
  w = np.zeros(d)
  losses = []
  max_iter = 10000
  iter = 0
  prev_loss = 1e8
  tol_count = 0
  while iter < max_iter:
    coord = np.random.choice(X.shape[1])
    grad = compute_gradient(X, y, w, coord)
    w[coord] += learn_rate * grad
    loss = compute_loss(X, y, w)
    losses.append(loss)
    if abs(prev_loss - loss) < tol:
      tol_count += 1
      if tol_count > 100:
        break
    else:
      tol_count = 0
    iter += 1
    prev_loss = losses[-1]
  return w, losses, iter

def compute_full_gradient(X, y, w):
  z = np.dot(w.T, X.T)
  gradients = y[:, np.newaxis] * X / (1 + np.exp(y * z)[:, np.newaxis])
  return np.sum(gradients, axis=0)

def greedy_coordinate_descent(X, y, learn_rate):
  d = X.shape[1]
  w = np.zeros(d)
  losses = []
  max_iter = 10000
  iter = 0
  gradients = compute_full_gradient(X, y, w)
  heap = [(-abs(grad), j) for j, grad in enumerate(gradients)]
  heapq.heapify(heap)
  while iter < max_iter:
    _, coord = heapq.heappop(heap)
    grad = compute_gradient(X, y, w, coord)
    w[coord] += learn_rate * grad
    new_grad = compute_gradient(X, y, w, coord)
    heapq.heappush(heap, (-abs(new_grad), coord))
    loss = compute_loss(X, y, w)
    losses.append(loss)
    iter += 1
  return w, losses, iter

def greedy_coordinate_descent_auto_stop(X, y, learn_rate, tol = 1e-3):
  d = X.shape[1]
  w = np.zeros(d)
  losses = []
  max_iter = 10000
  iter = 0
  gradients = compute_full_gradient(X, y, w)
  heap = [(-abs(grad), j) for j, grad in enumerate(gradients)]
  heapq.heapify(heap)
  prev_loss = 1e8
  tol_count = 0
  while iter < max_iter:
    _, coord = heapq.heappop(heap)
    grad = compute_gradient(X, y, w, coord)
    w[coord] += learn_rate * grad
    new_grad = compute_gradient(X, y, w, coord)
    heapq.heappush(heap, (-abs(new_grad), coord))

    loss = compute_loss(X, y, w)
    losses.append(loss)
    if abs(prev_loss - loss) < tol:
      tol_count += 1
      if tol_count > 100:
        break
    else:
      tol_count = 0
    iter += 1
    prev_loss = losses[-1]
  return w, losses, iter

def cyclic_coordinate_descent(X, y, learn_rate):
  d = X.shape[1]
  w = np.zeros(d)
  losses = []
  max_iter = 10000
  iter = 0
  coord = 0
  while iter < max_iter:
    grad = compute_gradient(X, y, w, coord)
    w[coord] += learn_rate * grad

    current_loss = compute_loss(X, y, w)
    losses.append(current_loss)
    iter += 1
    coord += 1
    coord = coord % d
  return w, losses, iter

def cyclic_coordinate_descent_shuff(X, y, learn_rate, tol = 1e-3): # Tried but does not see much effect
  d = X.shape[1]
  w = np.zeros(d)
  losses = []
  max_iter = 10000
  iter = 0
  j = 0
  shuff = [i for i in range(0,d)]
  np.random.shuffle(shuff)
  prev_loss = 1e8
  while iter < max_iter:
    coord = shuff[j]
    grad = compute_gradient(X, y, w, coord)
    w[coord] += learn_rate * grad
    current_loss = compute_loss(X, y, w)
    losses.append(current_loss)
    if abs(prev_loss - current_loss) < tol:
      tol_count += 1
      if tol_count > 100:
        break
    else:
      tol_count = 0
    iter += 1
    prev_loss = losses[-1]
    j += 1
    if j >= d:
      j = 0
      np.random.shuffle(shuff)
  return w, losses, iter

def cyclic_coordinate_descent_auto_stop(X, y, learn_rate, tol = 1e-3):
  d = X.shape[1]
  w = np.zeros(d)
  losses = []
  max_iter = 10000
  iter = 0
  coord = 0
  prev_loss = 1e8
  while iter < max_iter:
    grad = compute_gradient(X, y, w, coord)
    w[coord] += learn_rate * grad
    current_loss = compute_loss(X, y, w)
    losses.append(current_loss)
    if abs(prev_loss - current_loss) < tol:
      tol_count += 1
      if tol_count > 100:
        break
    else:
      tol_count = 0
    iter += 1
    coord += 1
    coord = coord % d
    prev_loss = losses[-1]
  return w, losses, iter
```

In [7]:

```
w_random, losses_random, iter_random = random_coordinate_descent(X_scaled, y, 0.01)
print("Final loss for RCD", compute_loss(X_scaled, y, w_random))
```

```
Final loss for RCD 0.40034762651348754
```

In [8]:

```
w_greedy, losses_greedy, iter_greedy = greedy_coordinate_descent(X_scaled, y, 0.01)
print("Final loss for GCD", compute_loss(X_scaled, y, w_greedy))
```

```
Final loss for GCD 0.2111636319553393
```

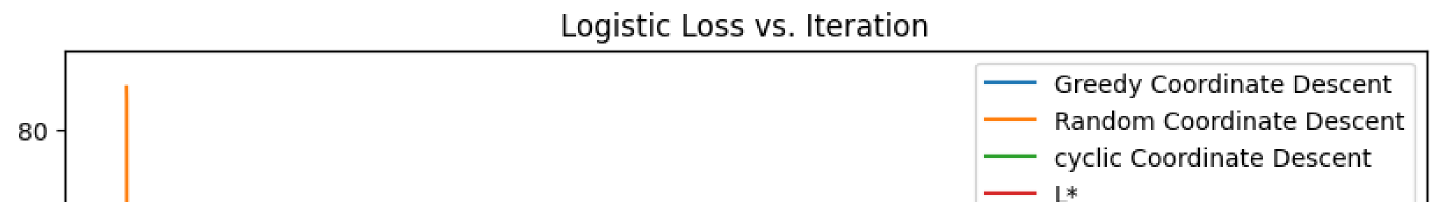
In [9]:

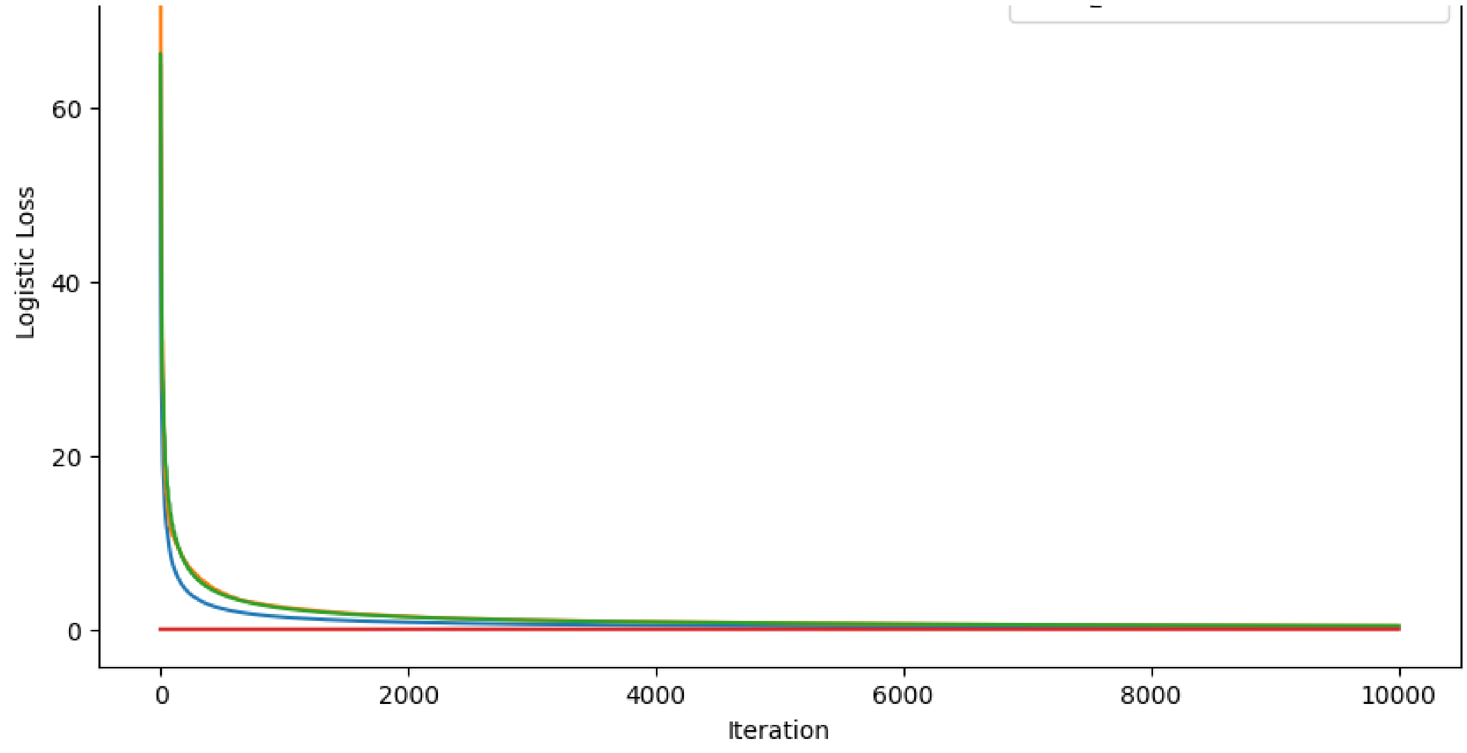
```
w_cyclic, losses_cyclic, iter_cyclic = cyclic_coordinate_descent(X_scaled, y, 0.01)
print("Final loss for CCD", compute_loss(X_scaled, y, w_cyclic))
```

```
Final loss for CCD 0.3970475342467698
```

In [10]:

```
plt.figure(figsize=(10, 6))
plt.plot(losses_greedy, label='Greedy Coordinate Descent')
plt.plot(losses_random, label='Random Coordinate Descent')
plt.plot(losses_cyclic, label='cyclic Coordinate Descent')
plt.plot([L_star]*iter_random, label='L*')
plt.xlabel('Iteration')
plt.ylabel('Logistic Loss')
plt.title('Logistic Loss vs. Iteration')
plt.legend()
plt.show()
```





In [11]:

```
w_random1, losses_random1, iter_random1 = random_coordinate_descent_auto_stop(X_scaled, y,
0.01)
print("Number of iterations taken:", iter_random1)
print("Final loss for RCD", compute_loss(X_scaled, y, w_random1))
```

```
Number of iterations taken: 3441
Final loss for RCD 0.9496744170178879
```

In [12]:

```
w_greedy1, losses_greedy1, iter_greedy1 = greedy_coordinate_descent_auto_stop(X_scaled, y,
0.01)
print("Number of iterations taken:", iter_greedy1)
print("Final loss for GCD", compute_loss(X_scaled, y, w_greedy1))
```

```
Number of iterations taken: 1181
Final loss for GCD 1.2321161148832704
```

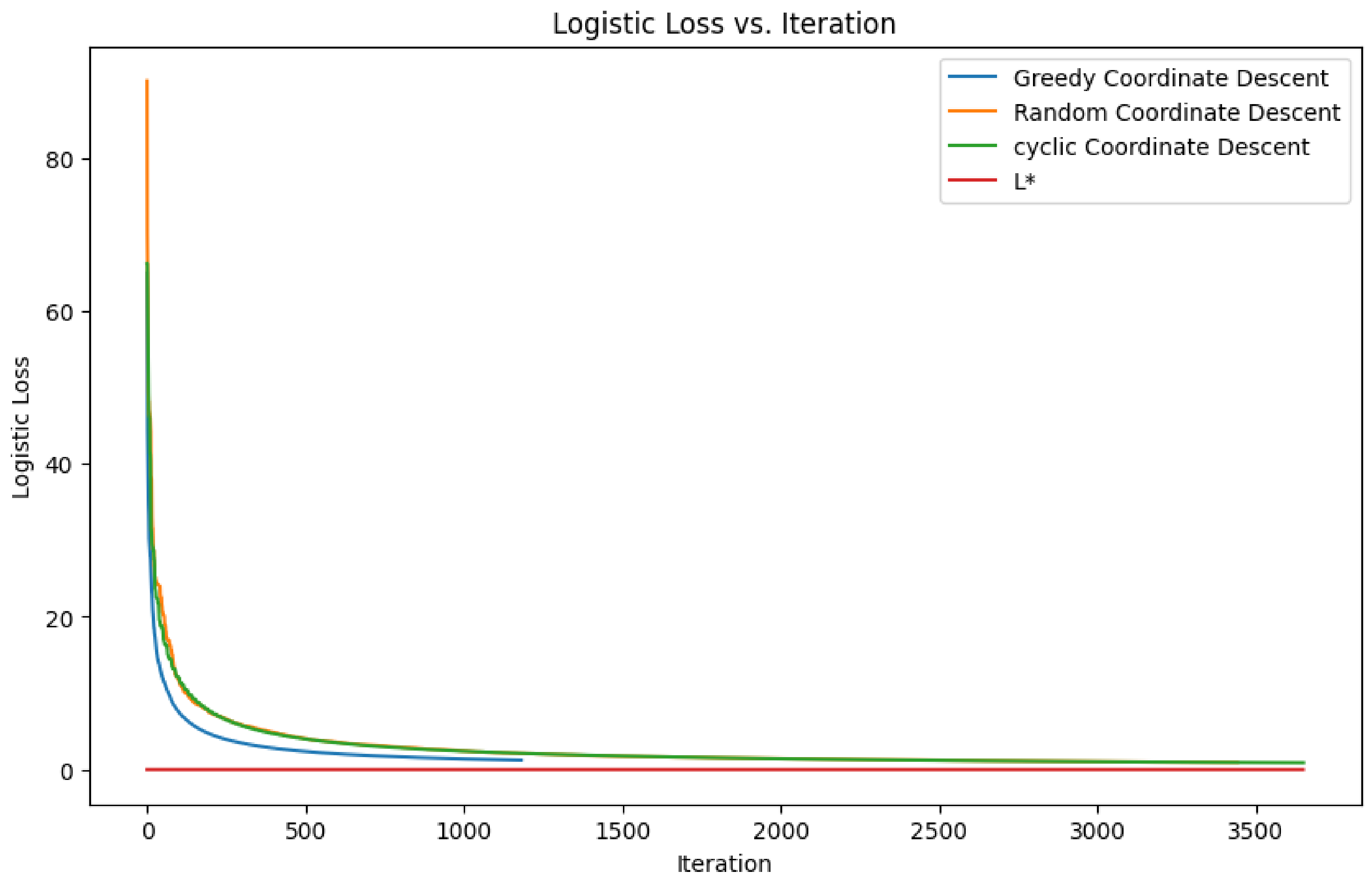
In [13]:

```
w_cyclic1, losses_cyclic1, iter_cyclic1 = cyclic_coordinate_descent_auto_stop(X_scaled, y,
0.01)
print("Number of iterations taken:", iter_cyclic1)
print("Final loss for BCD", compute_loss(X_scaled, y, w_cyclic1))
```

```
Number of iterations taken: 3649
Final loss for BCD 0.8993013792720428
```

In [14]:

```
plt.figure(figsize=(10, 6))
plt.plot(losses_greedy1, label='Greedy Coordinate Descent')
plt.plot(losses_random1, label='Random Coordinate Descent')
plt.plot(losses_cyclic1, label='cyclic Coordinate Descent')
plt.plot([L_star]*iter_cyclic1, label='L*')
plt.xlabel('Iteration')
plt.ylabel('Logistic Loss')
plt.title('Logistic Loss vs. Iteration')
plt.legend()
plt.show()
```



In [15]:

```
def measure_average_runtime(method, *args, runs=5):
    total_time = 0
    total_loss = 0
    for _ in range(runs):
        start_time = time.time()
        w, l, it = method(*args)
        total_time += time.time() - start_time
        total_loss += compute_loss(X_scaled, y, w)
    return total_time / runs, total_loss / runs

# Measure average runtimes for each method
average_runtime_rcd, l_rcd = measure_average_runtime(random_coordinate_descent_auto_stop, X_
scaled, y, 0.01)
average_runtime_gcd, l_gcd = measure_average_runtime(greedy_coordinate_descent_auto_stop, X_
scaled, y, 0.01)
average_runtime_ccd, l_ccd = measure_average_runtime(cyclic_coordinate_descent_auto_stop, X_
scaled, y, 0.01)

print("Average RCD run time:", average_runtime_rcd, "Secs,", "Average loss:", l_rcd)
print("Average GCD run time:", average_runtime_gcd, "Secs,", "Average loss:", l_gcd)
print("Average CCD run time:", average_runtime_ccd, "Secs,", "Average loss:", l_ccd)
```

```
Average RCD run time: 0.20438556671142577 Secs, Average loss: 0.9225740320270553
Average GCD run time: 0.06264090538024902 Secs, Average loss: 1.2321161148832704
Average CCD run time: 0.13578972816467286 Secs, Average loss: 0.8993013792720428
```

In [16]:

```
# Measure average runtimes for each method
average_runtime_rcd, l_rcd = measure_average_runtime(random_coordinate_descent, X_scaled, y,
0.01)
average_runtime_gcd, l_gcd = measure_average_runtime(greedy_coordinate_descent, X_scaled, y,
0.01)
average_runtime_ccd, l_ccd = measure_average_runtime(cyclic_coordinate_descent, X_scaled, y,
0.01)

print("Average RCD run time:", average_runtime_rcd, "Secs,", "Average loss:", l_rcd)
print("Average GCD run time:", average_runtime_gcd, "Secs,", "Average loss:", l_gcd)
print("Average CCD run time:", average_runtime_ccd, "Secs,", "Average loss:", l_ccd)
```

```
Average RCD run time: 1.04323148727417 Secs, Average loss: 0.4009665176823816
Average GCD run time: 1.4621417045593261 Secs, Average loss: 0.2111636319553393
Average CCD run time: 0.8478741645812988 Secs, Average loss: 0.3970475342467698
```

In [16]: