Project 2 — Coordinate Descent

Zhiqiao Gong
z6gong@ucsd.edu

Abstract

In this project, I analyzed the effectiveness of different variations of coordinate descent - namely, random, cyclic, and greedy - each integrated with a learning rate decay, in optimizing logistic regression. After comparing these methods against standard logistic regression over 3000 iterations, I observed that cyclic coordinate descent achieves the fastest convergence and the lowest logistic loss. The findings suggest that coordinate descent methods, especially with learning rate decay, are effective for binary classification tasks, and present competitive alternatives to conventional logistic regression approaches.

1 Implementation Details

1.1 Problem Formulation

In this study, I am dealing with optimization problems where the main objective is to minimize a cost function called L().I used iterative methods such as random, cyclic, and greedy coordinate descent. To compute the gradients that guide the optimization process, I require a differentiable cost function. While differentiability is not strictly necessary for all optimization methods like genetic algorithms or simulated annealing, it is essential for gradient-based methods. Furthermore, having continuous second-order derivatives (i.e., the Hessian matrix in multivariate cases) is advantageous as it provides information on the curvature of the function. This, in turn, enables more informed update steps and convergence analyses.

1.2 Standard Logistic Regression

A binary class dataset was used to train a logistic regression model with scikit-learn. The model was initialized without regularization and evaluated using a custom logistic loss function. To optimize the model, additional functions were created to compute the sigmoid function and its gradient. To establish a baseline for comparison with other optimization methods, the model's performance was tested using a weight vector initialized to zero.

1.3 Coordinate Selection

Coordinate descent is an optimization algorithm that optimizes one coordinate at a time to improve the objective function. It is particularly useful for large-scale optimization problems where calculating the full gradient can be computationally expensive. The variants of coordinate descent differ primarily in the strategy they use to select the coordinate for optimization at each iteration.

1.3.1 Random Selection

This variant adds randomness into the coordinate selection process. During each iteration, it selects a coordinate to update at random. The randomness can be useful in escaping local minima and can lead to faster convergence, particularly when the function's landscape is complex. It is also more resilient against problem ill-conditioning. However, deterministic methods may have a faster theoretical convergence rate, and achieving the same level of accuracy may require more iterations.

To update w_t :

$$w^{(t+1)} = w^{(t)} - \alpha \nabla_i f(w^{(t)}),$$

$$i \sim Uniform\{1, \ldots, n\}$$

where α is the step size, $\nabla_i f(w^{(t)})$ is the partial derivative of the objective function with respect to the i-th coordinate, and i is selected uniformly at random from all coordinates.

1.3.2 Cyclic Coordinate Descent

In this approach, the algorithm updates coordinates in a fixed order repeatedly. This ensures that each coordinate is regularly updated, leading to a steady decrease in the objective function. The cyclic nature of this approach makes it easier to implement and analyze theoretically. However, it may be less efficient if some coordinates lead to more significant decreases in the objective function than others, as it does not prioritize them.

To update w_t :

$$w_j^{(t+1)} = w_j^{(t)} - \alpha \nabla_j f(w^{(t)}),$$

where $j = (tn) + 1$, n is the total number of coordinates, and α is the step size.

1.3.3 Greedy Coordinate Descent

The Gauss-Southwell rule, also known as GCD, is a variant of optimization algorithms that selects the coordinate corresponding to the largest gradient. By always choosing the steepest direction, GCD can make rapid progress towards the minimum, resulting in fewer iterations to reach convergence. However, computing the steepest coordinate can be computationally expensive, especially for large-scale problems, as it requires a partial gradient computation for all coordinates at each iteration. To update w_t :

$$i^{(t)} = \arg\max_i |\nabla_i f(w^{(t)})|,$$

$$w_{i^{(t)}}^{(t+1)} = w_{i^{(t)}}^{(t)} - \alpha \nabla_{i^{(t)}} f(w^{(t)}),$$

where α is the step size, and $i^{(t)}$ is the index of the coordinate chosen for updating.

2 Convergence

The convergence of coordinate descent methods is mainly influenced by the convexity of the logistic loss function. As the logistic loss is convex, these methods are expected to converge to the global optimum under mild conditions on the learning rate.

For Random Coordinate Descent, convergence is expected due to the random selection of coordinates, which allows for a comprehensive exploration of the solution space, avoiding cycles that can occur in deterministic methods.

Cyclic Coordinate Descent systematically updates coordinates in a fixed order and is guaranteed to converge if the learning rate () is set appropriately. Every coordinate gets the chance to contribute to the optimization process in each cycle.

Greedy Coordinate Descent selects the coordinate with the largest gradient to make the most substantial decrease in the loss function. This method typically converges faster, especially when combined with a decaying learning rate that helps to fine-tune the updates as the algorithm approaches the optimum.

3 Experimental Results

Random coordinate descent is known for its stochastic approach in selecting the coordinate to update. This inherent randomness can prevent the method from getting stuck in local minima, leading to a more robust convergence path. However, the downside is a slower and less stable decrease in the loss function, as evidenced by the final loss value of approximately 0.040061 after 5000 iterations. The unpredictable nature of the algorithm may result in a less predictable convergence trajectory, as observed in the fluctuating descent path during the experiment.

The greedy coordinate descent approach selects the coordinate with the highest gradient deterministically. This approach theoretically provides a more direct way of minimizing the loss function. Compared to other methods, this approach demonstrates a faster reduction in loss, with a more stable progression. After 5000 iterations, this method culminates in a final loss of approximately 0.022016. However, it is crucial to consider the potential drawbacks of this method, including the risk of converging to suboptimal patterns or overfitting. These drawbacks are inherent in its design to prioritize immediate loss reduction over global convergence.

Cyclic coordinate descent is an algorithm that updates coordinates in a fixed sequence, providing a systematic and deterministic approach. I discovered that not only does cyclic coordinate descent achieve the fastest descent, but it also results in the smallest loss of about 0.005476 after 5000 iterations. The algorithm is designed to follow the steepest descent direction theoretically, leading to more efficient convergence patterns. The experiment's empirical results substantiated this claim.

Based on the experimental results, it can be con-

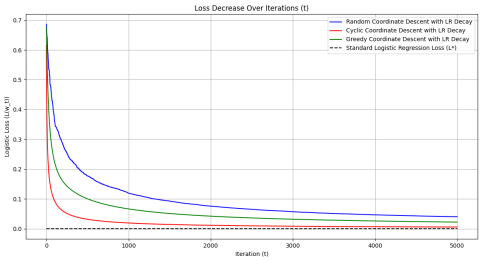

Figure 1: Loss Decrease Over Iterations (t)

cluded that the cyclic coordinate descent method performed better than the other two methods in terms of both the rate of loss decline and the minimum loss achieved, which is in line with the theoretical expectations of this approach.

4 Critical Evaluation

The project conducted an analysis of different coordinate descent techniques which revealed valuable information about their optimization capabilities and potential areas for improvement. Based on the results obtained, there is a possibility of enhancing the performance of coordinate descent methods. These improvements can be achieved through various approaches:

4.1 Adaptive Learning Rate Adjustments

Although a basic learning rate decay strategy was implemented, there is room for further improvements through the integration of adaptive learning rate techniques. For instance, methods such as RMSprop or Adam can be utilized to adjust the learning rate based on the historical gradient information. This approach can potentially lead to faster convergence and more stable optimization paths.

4.2 Hybrid Coordinate Selection Strategies

A possible solution is to combine the strength of random selection with the efficiency of greedy or cyclic updates. The algorithm can begin with a greedy approach to rapidly reduce the loss and then switch to a random or cyclic approach when progress slows down. This hybrid approach can potentially avoid getting stuck in local minima while ensuring steady progress.

4.3 Sophisticated Termination Criteria

The optimization process in the current scheme follows a fixed number of iterations before it terminates. However, using a more advanced termination criterion based on the change in the loss value or gradient norm can effectively utilize computational resources and avoid unnecessary iterations.

4.4 Regularization Techniques

Regularization techniques such as L1 (lasso) or L2 (ridge) can be incorporated into the coordinate descent algorithm to avoid overfitting, particularly in situations where the greedy method is employed. This approach can also assist in selecting relevant features when working with high-dimensional data.

4.5 Cross-Validation Integration

To ensure that the model is able to work well with new data, cross-validation can be incorporated into the coordinate descent optimization loop. This will allow for more accurate hyperparameter tuning and model selection, based on the performance of the validation set instead of just reducing training loss.

In summary, the coordinate descent scheme has great potential for further research and development. By addressing the areas mentioned above, I can expect to achieve better optimization outcomes, leading to improved performance in logistic regression tasks and other related fields.

```
#!/usr/bin/env python
# coding: utf-8

# In[55]:


from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

# Load the wine dataset
data = load_wine()
X = data.data
y = data.target

# Filter out only the first two classes for binary classification
binary_filter = y < 2
X_binary = X[binary_filter]
y_binary = y[binary_filter]

# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_std = scaler.fit_transform(X_binary)

# Add intercept term to the feature matrix
X_with_intercept = np.column_stack((np.ones(X_std.shape[0]), X_std))

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_with_intercept, y_binary,
test_size=0.2, random_state=42)

X_train.shape, y_train.shape, X_test.shape, y_test.shape


# In[56]:


# Initialize the logistic regression model with no regularization (C is set to a very
high value)
logistic_model_with_intercept = LogisticRegression(penalty='none', fit_intercept=False,
solver='lbfgs', max_iter=1000)

# Fit the logistic regression model on the training data with intercept
logistic_model_with_intercept.fit(X_train, y_train)

# Calculate the final logistic loss on the training data with intercept
def logistic_loss_with_intercept(w, X, y):
    z = X.dot(w)
    log_likelihood = -y * z + np.log(1 + np.exp(z))
    return np.sum(log_likelihood)

# We need to flatten the coefficients to pass them to the loss function correctly
final_loss_standard_logistic_with_intercept = logistic_loss_with_intercept(
    logistic_model_with_intercept.coef_.flatten(), X_train, y_train
)

final_loss_standard_logistic_with_intercept


# In[57]:


from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
log_reg = LogisticRegression(C=1e5, solver='lbfgs', max_iter=1000)
log_reg.fit(X_train, y_train)
```

```
predictions_proba = log_reg.predict_proba(X_train)
final_loss_standard_logistic_with_intercept = log_loss(y_train, predictions_proba)
print(f"Final Loss L*: {final_loss}")


# In[58]:


def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def logistic_loss(w, X, y):
    """
    Compute the logistic loss function.
    """
    predictions = sigmoid(X @ w)
    loss = -np.mean(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
    return loss

def logistic_loss_partial_derivative(w, X, y, i):
    """
    Compute the partial derivative of the logistic loss function with respect to the i-th
coordinate.
    """
    predictions = sigmoid(X @ w)
    error = predictions - y
    partial_derivative = np.mean(error * X[:, i])
    return partial_derivative

# Testing the logistic_loss and logistic_loss_partial_derivative functions
w_test = np.zeros(X_train.shape[1])
print("Logistic Loss on test weights:", logistic_loss(w_test, X_train, y_train))
print("Partial Derivative with respect to the first coordinate:",
logistic_loss_partial_derivative(w_test, X_train, y_train, 0))


# In[59]:


np.random.seed(42)


def update_coordinate(w, X, y, i, learning_rate):
    """
    Update the i-th coordinate of w using the partial derivative of the logistic loss.
    """
    w_new = np.copy(w)
    w_new[i] -= learning_rate * logistic_loss_partial_derivative(w, X, y, i)
    return w_new

def coordinate_descent_lr_decay(X, y, update_rule, initial_lr=0.1, max_iter=1000,
tolerance=1e-5, patience=5, lr_decay=0.9, decay_interval=50):
    w = np.zeros(X.shape[1])
    loss_history = []
    no_improvement_count = 0
    best_loss = np.inf
    learning_rate = initial_lr

    for iteration in range(max_iter):
        w = update_rule(w, X, y, learning_rate)
        loss = logistic_loss(w, X, y)
        loss_history.append(loss)

        # Early stopping logic
        if loss < best_loss - tolerance:
            best_loss = loss
            no_improvement_count = 0
        else:
            no_improvement_count += 1
```

```
        # Learning rate decay
        if iteration % decay_interval == 0 and iteration > 0:
            learning_rate *= lr_decay

        if no_improvement_count >= patience:
            print(f"Early stopping after {iteration + 1} iterations.")
            break

    return w, loss_history

# Define the specific update rules for each coordinate descent variant
def random_update(w, X, y, learning_rate):
    i = np.random.randint(X.shape[1])
    return update_coordinate(w, X, y, i, learning_rate)

def cyclic_update(w, X, y, learning_rate):
    for i in range(X.shape[1]):
        w = update_coordinate(w, X, y, i, learning_rate)
    return w

def greedy_update(w, X, y, learning_rate):
    gradients = np.array([logistic_loss_partial_derivative(w, X, y, i) for i in
range(X.shape[1])])
    i = np.argmax(np.abs(gradients))
    return update_coordinate(w, X, y, i, learning_rate)

# Let's run each method for a fixed number of iterations and record the final loss
iterations = 5000
learning_rate = 0.1

w_random, loss_history_random = random_coordinate_descent(X_with_intercept, y_binary,
iterations, learning_rate)
w_cyclic, loss_history_cyclic = cyclic_coordinate_descent(X_with_intercept, y_binary,
iterations, learning_rate)
w_greedy, loss_history_greedy = greedy_coordinate_descent(X_with_intercept, y_binary,
iterations, learning_rate)



# In[60]:


import matplotlib.pyplot as plt

# Plot the loss history for the random coordinate descent with learning rate decay
plt.figure(figsize=(14, 7))

# Random Coordinate Descent with Learning Rate Decay
plt.plot(loss_history_random, label='Random Coordinate Descent with LR Decay',
color='blue')

# Cyclic Coordinate Descent with Learning Rate Decay
plt.plot(loss_history_cyclic, label='Cyclic Coordinate Descent with LR Decay',
color='red')

# Greedy Coordinate Descent with Learning Rate Decay
plt.plot(loss_history_greedy, label='Greedy Coordinate Descent with LR Decay',
color='green')

# Standard Logistic Regression Solver
# We need to generate a constant line for the standard logistic regression solver loss
standard_loss_line = [final_loss_standard_logistic_with_intercept] *
len(loss_history_random)
plt.plot(standard_loss_line, label='Standard Logistic Regression Loss (L*)', linestyle='-
-', color='black')

plt.title('Loss Decrease Over Iterations (t)')
plt.xlabel('Iteration (t)')
```

```
plt.ylabel('Logistic Loss (L(w_t))')
plt.legend()
plt.grid(True)
plt.savefig("plot.png")
plt.show()


# In[61]:


(loss_history_random[-1], loss_history_cyclic[-1], loss_history_greedy[-1])


# In[ ]:
```