Coordinate Descent for Logistic Regression: Integrating A* Search and Weighted Selection

Jintong Luo
jil386@ucsd.edu

Abstract

Optimizing logistic regression via coordinate descent is enhanced by integrating A* search with a weighted selection mechanism. This paper introduces methods that strategically select coordinates using A* search heuristics and gradient magnitude weighting. Our empirical studies demonstrate these methods' superior performance over traditional random coordinate selection, providing substantial efficiency gains in logistic regression optimization tasks.

1 Description

Our coordinate descent methods address two fundamental problems in the optimization of logistic regression: which coordinate to update (problem a) and how to determine the new value for the coordinate (problem b).

For problem (a), we utilize an A* search-based heuristic to transform the coordinate selection process into a graph-path-finding problem, allowing for a more informed selection strategy. Concurrently, the WeightedLargestCoordinateDescent method selects coordinates based on the magnitude of their gradients, introducing a preference for coordinates that are likely to yield a more significant decrease in the loss function. The CyclicCoordinateDescent method loops across all coordinates and update the weight for each predictor cyclically.

To address problem (b), a gradient descent step is applied to the selected predictor.

This necessitates the cost function L(·) to be differentiable, as the update relies on the computation of gradients. Our approach employs the logistic regression function, which is inherently differentiable, ensuring the applicability of gradient-based updates. Through extensive experiments, we establish that these methods outperform the RandomCoordinateDescent method.

2 Convergence

2.1 General Coordinate Descent

Convergence in General Coordinate Descent is determined by the change in loss with respect to the tolerance level set for the updates on the weights of predictors. We specify a maximum number of iterations (*max_iter=20000*) and a tolerance (*tol=1e-9*).

The algorithm is considered to have converged for a specific coordinate if the absolute change in loss is less than the tolerance. If all coordinates meet this criterion continuously, we declare overall convergence. However, if any coordinate's loss change exceeds the tolerance, the algorithm must re-evaluate the convergence state for all coordinates. This process continues iteratively until the maximum number of iterations is reached or convergence is achieved.

2.2 AStar Coordinate Descent

AStar Coordinate Descent necessitates an additional convergence condition due to the A* algorithm's distinct operation of not continuously updating a single model. Besides the General Coordinate Descent convergence criterion, we consider AStarCoordinateDescent to have converged if the loss is below the tolerance for an sequence of iterations (specifically, ten times the number of coordinates). This condition ensures that convergence is not prematurely declared and that the solution is, with high probability, close to optimal.

3 Experimental result

In our experimental evaluation, we rigorously tested various coordinate descent methods to assess their convergence speed, stability, and final loss. The methods included:

- RandomCoordinateDescent: Selects a random coordinate and performs a gradient descent

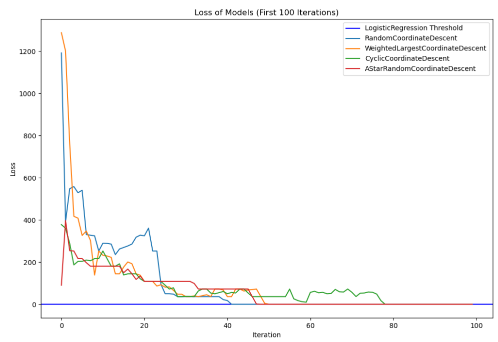

Figure 1: Loss vs Iteration

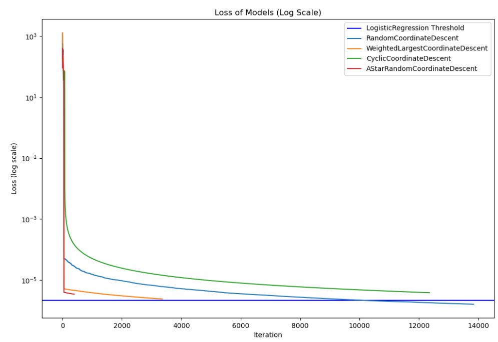

Figure 2: Log Loss vs Iteration

update.

- WeightedLargestCoordinateDescent: Utilizes the absolute gradient values of all coordinates to form a probability distribution, from which the next coordinate is probabilistically selected for updating.

- CyclicCoordinateDescent: Sequentially processes each coordinate in a fixed order, applying gradient descent updates.

- AStarCoordinateDescent: Employs a heuristic function $h(x) = \text{loss} \times (1 + \text{cur\_iter}/\text{max\_iter})$ to estimate the priority of models. At each iteration, it forms three updated models by selecting coordinates based on WeightedLargestCoordinateDescent and iterates the A* algorithm.

All methods were standardized on the following hyperparameters: *learning_rate=20*, *max_iter=20000*, *tolerance=1e-9*, and *lr_schedule* set to constant. AStar Coordinate Descent, capable of more stable and robust updates, was allocated a higher *learning_rate* to facilitate its heuristic-driven exploration.

The experimental setup was carefully designed to ensure comparability across the different methods. We recorded the iteration count at convergence, the stability of convergence across iterations, and the final loss value attained by each method. These results were then plotted to visualize the convergence behavior over the iterations. Result shows in Table1, Figure 1 & 2.

4 Critical evaluation

4.1 Analysis

Our analysis is based on the interpretation of the table, loss graphs and log loss graphs. These graphs reveal insights into convergence speed, final loss results, and the stability of different coordinate descent methods over a range of iterations.

- Convergence Speed (Iterations)

  The convergence speed is a key performance indicator in optimization algorithms. Based on the data, we observe that AStarCoordinateDescent exhibits the fastest convergence, followed by WeightedLargestCoordinateDescent, CyclicCoordinateDescent, and RandomCoordinateDescent. This hierarchy suggests that updating based on the magnitude of the gradient facilitates convergence, as larger gradients are indicative of the steepest descent direction. The application of A* search within AStarCoordinateDescent enhances the stability and speed of convergence by utilizing heuristic guidance for coordinate selection.

- Stability

  Stability during convergence is crucial for the robustness of an optimization method. The loss graph shows that AStarCoordinateDescent maintains a more stable descent compared to the other methods, with minimal oscillation in the loss curve. As the number of iterations increases, the loss curve of AStarCoordinateDescent smoothens out, indicating a stable approach towards the optimal loss.

- Final Loss

Method	Loss	Iteration
LogisticRegression	2.1681613e-06	-
RandomCoordinateDescent	1.6135818e-06	13843
WeightedLargestCoordinateDescent	2.3908566e-06	3362
CyclicCoordinateDescent	3.8498294e-06	12354
AStarCoordinateDescent	3.3911857e-06	392

Table 1: Performance comparison of optimization methods.

All methods ultimately converge to a final loss comparable to that of the standard LogisticRegression loss, which corroborates the appropriateness of our hyperparameter settings. The convergence of all methods to a similar final loss value demonstrates that the optimizations are accurate and that the different methods are correctly implemented and configured.

4.2 Further scope

While our study focuses on the optimization of logistic regression using coordinate descent methods, there is potential for further research that integrates pre-processing techniques such as Principal Component Analysis (PCA). PCA has the capacity to transform the predictors into a set of orthogonal coordinates, which simplifies the optimization landscape by ensuring that updates to one coordinate do not affect others. This could potentially lead to improvements in convergence speed and stability due to the more precise direction of regression each coordinate update would provide.

This approach was not adopted in our current study due to the significant computational overhead of PCA and our focus on direct logistic regression optimization rather than reconstruction of the data set. Investigating the balance between PCA's computational demand and optimization efficiency remains an area for subsequent studies.

Appendix: Code

```
import json

import numpy as np
from sklearn.linear_model import
    LogisticRegression as
    SklearnLogisticRegression
from sklearn.datasets import load_wine
from sklearn.model_selection import
    train_test_split
from sklearn.preprocessing import
    StandardScaler
from sklearn.metrics import log_loss
from sklearn.metrics import
    classification_report
from queue import PriorityQueue
import itertools
import matplotlib.pyplot as plt


class BaseRegression:
    def __init__(self,
        learning_rate=0.01,
        max_iter=20000, tol=1e-6):
        self.initial_lr = learning_rate
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.tol = tol
        self.weights = None

    def predict(self, X):
        pass

    def predict_proba(self, X):
        pass

    def get_loss(self, X, y):
        # probabilities =
            self.predict_proba(X)
        # loss = np.sum(np.log(1 +
            np.exp(-y * np.dot(X,
            self.weights))))
        probabilities =
            self.predict_proba(X)
        # loss =
            np.sum(-y*np.log(probabilities)-(1-y)*np.log
        return log_loss(y, probabilities,
            normalize=False)
        # return loss

    def test(self, X_test, y_test):
        # Predict using the trained model
        predictions = self.predict(X_test)

        # Generate classification report
        report =
            classification_report(y_test,
            predictions)
        print("\n--------------------\n\nClassification
            Report for:",
            self.__class__.__name__)
        print(report)


class
    LogisticRegression(BaseRegression):
    def fit(self, X, y):
        self.model =
            SklearnLogisticRegression(penalty=None,
            solver='lbfgs',
            max_iter=self.max_iter,
            tol=self.tol)
```

```python
        self.model.fit(X, y)
        self.weights = self.model.coef_[0]

    def predict(self, X):
        return self.model.predict(X)

    def predict_proba(self, X):
        return 1 / (1 + np.exp(-np.dot(X,
            self.weights)))
        # return
            self.model.predict_proba(X)


class CoordinateDescent(BaseRegression):
    def __init__(self, learning_rate=5,
        max_iter=20000, tol=1e-6,
        lr_schedule=('decreasing',
        0.001)):
        super().__init__(learning_rate,
            max_iter, tol)
        self.lr_schedule = lr_schedule
        self.loss_history = []
        self.converge_feature = None

    def init_weight(self, shape):
        self.weights = np.zeros(shape)
        self.converge_feature =
            np.zeros(shape).astype(bool)
        self.loss_history = []

    def update_weight(self, X, y):
        pass

    def update_loss(self, X, y,
        index=None):
        if index is None:
            index =
                list(np.ndindex(self.converge_feature.shape))
        loss = self.get_loss(X, y)
        self.loss_history.append(loss)
        # print("Update index: {},
            gradient: {}".format(j,
            gradient))
        # Check convergence
        if len(self.loss_history) > 1 and
            np.abs(self.loss_history[-1]
            - self.loss_history[-2]) <
            self.tol:
            self.converge_feature[index] =
                True
            if
                np.all(self.converge_feature):
                return True
        else:
            self.converge_feature[:] =
                False
        return False

    def update_learning_rate(self,
        iteration):
        """
        Update the learning rate based on
            the iteration number and the
            chosen schedule.
        """
        if self.lr_schedule[0] ==
            'constant':
            # Keep the learning rate
                constant
            self.learning_rate =
                self.initial_lr
        elif self.lr_schedule[0] ==
            'decreasing':
            # Decrease the learning rate
                over iterations
            self.learning_rate =
                self.initial_lr / (1 +
                self.lr_schedule[1] *
                iteration)

    def fit(self, X, y):
        self.init_weight(X.shape[1])

        for i in range(self.max_iter):
            if self.update_weight(X, y):
                break
            self.update_learning_rate(i)

    def predict_proba(self, X):
        # Logistic function
        prod = -np.dot(X, self.weights)
        prod[prod > 100] = 100
        return 1 / (1 + np.exp(prod))

    def predict(self, X):
        proba = self.predict_proba(X)
        # For binary classification,
            compare the probability of
            the positive class with the
            threshold 0.5
        predictions = (proba >=
            0.5).astype(int)
        return predictions

    def onehot_encode(self, y):
        n_classes = np.unique(y).size
        return np.eye(n_classes)[y]


class
    AllCoordinateDescent(CoordinateDescent):
    def update_weight(self, X, y):
        # Update the j-th weight
        for j in range(X.shape[1]):
            gradient = np.dot(X[:, j],
                self.predict_proba(X) - y)
            # gradient = -np.sum(y * X[:,
                j] * (1 -
                self.predict_proba(X)))
            self.weights[j] -=
                self.learning_rate *
                gradient

        return self.update_loss(X, y)


class
    RandomCoordinateDescent(CoordinateDescent):
    def update_weight(self, X, y):
        # Randomly pick a coordinate
        j = np.random.randint(0,
            X.shape[1])

        # Update the j-th weight
        gradient = np.dot(X[:, j],
            self.predict_proba(X) - y)
        # gradient = -np.sum(y * X[:, j]
            * (1 - self.predict_proba(X)))
```

```python
        self.weights[j] -=
            self.learning_rate * gradient

        return self.update_loss(X, y, j)


class
    LargestCoordinateDescent(CoordinateDescent):
    def update_weight(self, X, y):
        largest_gradient = 0
        index = -1
        for j in range(X.shape[1]):
            gradient = np.dot(X[:, j],
                self.predict_proba(X) - y)
            if gradient > largest_gradient:
                largest_gradient = gradient
                index = j
        # gradient = -np.sum(y * X[:, j]
            * (1 - self.predict_proba(X)))
        self.weights[index] -=
            self.learning_rate *
            largest_gradient

        return self.update_loss(X, y,
            index)


class
    WeightedLargestCoordinateDescent(CoordinateDescent):
    def update_weight(self, X, y):
        gradient = np.zeros(X.shape[1])
        for j in range(X.shape[1]):
            gradient[j] = np.dot(X[:, j],
                self.predict_proba(X) - y)

        # gradient = -np.sum(y * X[:, j]
            * (1 - self.predict_proba(X)))
        prob = np.abs(gradient) /
            np.sum(np.abs(gradient))
        index =
            np.random.choice(len(prob),
            p=prob)
        self.weights[index] -=
            self.learning_rate *
            gradient[index]

        return self.update_loss(X, y,
            index)


class
    CyclicCoordinateDescent(CoordinateDescent):
    def update_weight(self, X, y, index):
        index = index % X.shape[1]
        gradient = np.dot(X[:, index],
            self.predict_proba(X) - y)
        # gradient = -np.sum(y * X[:, j]
            * (1 - self.predict_proba(X)))
        self.weights[index] -=
            self.learning_rate * gradient

        return self.update_loss(X, y,
            index)

    def fit(self, X, y):
        self.init_weight(X.shape[1])

        for i in range(self.max_iter):
            if self.update_weight(X, y, i):
                self.update_learning_rate(i)
                    break

class
    AStarRandomCoordinateDescent(CoordinateDescent):
    def __init__(self, learning_rate=5,
        max_iter=20000, tol=1e-6,
        lr_schedule='decreasing'):
        super().__init__(learning_rate,
            max_iter, tol, lr_schedule)
        self.open_set = PriorityQueue() #
            Nodes (models) to be evaluated
        self.best = None
        self.counter = itertools.count()

    def fit(self, X, y):
        self.a_star_search(X, y)

    def test(self, X_test, y_test):
        if self.best is not None:
            self.best.test(X_test, y_test)
        else:
            print("\n--------------------\n\nClassificati
                Report for:",
                self.__class__.__name__, "
                NOT EXIST!")

    def heuristic(self, loss,
        iterations):
        return loss * (1 + iterations /
            self.max_iter)

    def a_star_search(self, X, y):
        start_node = self.create_node() #
            Create a new instance of
            RandomCoordinateDescent
        start_node.init_weight(X.shape[1])
        # start_node.iterations = 0 #
            Initialize iterations
        # start_node.loss =
            start_node.get_loss(X, y) #
            Initial loss
        start_f_score =
            self.heuristic(start_node.get_loss(X,
            y),
            len(start_node.loss_history))

        self.open_set.put((start_f_score,
            next(self.counter),
            start_node))
        best_loss = 9999
        converge_iters = 0

        for i in range(self.max_iter):
            cur_loss, _, current_node =
                self.open_set.get()
            self.loss_history.append(cur_loss)
            if i % 1000 == 0:
                print("Size:",
                    len(self.open_set.queue),
                    ", Loss:", cur_loss)

            if best_loss > cur_loss:
                best_loss = cur_loss
                self.best = current_node

            if len(self.loss_history) > 1
                and
                np.abs(self.loss_history[-1]
                - self.loss_history[-2]) <
```

```
            self.tol:
           converge_iters += 1
           if converge_iters >= 10 *
               X.shape[1]:
              break

        for j in range(3):
           new_node =
               self.create_node(current_node)
           if
               new_node.update_weight(X,
               y):
              self.best = new_node
              break
           new_node.update_learning_rate(i)

           new_f_score =
               self.heuristic(new_node.get_loss(X,
               y),
               len(new_node.loss_history))
           if not self.open_set.full():
              self.open_set.put((new_f_score,
                  next(self.counter),
                  new_node)) # Push
                  the updated state

     self.weights =
         np.copy(self.best.weights)

  def create_node(self,
      base_model=None):
     # Create a new instance of
         RandomCoordinateDescent with
         the same or given state
     new_instance =
         WeightedLargestCoordinateDescent(
        learning_rate=self.learning_rate,
        max_iter=self.max_iter,
        tol=self.tol,
        lr_schedule=self.lr_schedule)
     if base_model:
        new_instance.weights =
            np.copy(base_model.weights)
        new_instance.loss_history =
            base_model.loss_history.copy()
        new_instance.converge_feature
            =
            np.copy(base_model.converge_feature)
        new_instance.learning_rate =
            base_model.learning_rate
        # new_instance.iterations =
            base_model.iterations
     return new_instance


def
    load_and_preprocess_data(test_size=0.0):
  # Load wine dataset
  wine_data = load_wine()
  X, y = wine_data.data,
      wine_data.target

  # Filter out only the first two
      classes
  filter_mask = y < 2
  X_filtered = X[filter_mask]
  y_filtered = y[filter_mask]

  # Preprocess the data
  scaler = StandardScaler()
```

```
  X_scaled =
      scaler.fit_transform(X_filtered)
  # X_scaled = np.hstack((X_scaled,
      np.ones((X_scaled.shape[0], 1))))

  # Split the data
  if test_size == 0:
     X_train, X_test, y_train, y_test
         = X_scaled, None, y_filtered,
         None
  else:
     X_train, X_test, y_train, y_test
         = train_test_split(X_scaled,
         y_filtered,
         test_size=test_size,
         random_state=42)

  return X_train, X_test, y_train,
      y_test


def get_index(loss_history, thres):
  # Find indices of elements less than
      the threshold
  indices = np.where(loss_history <
      thres)
  first_index = indices[0][0] if
      indices[0].size > 0 else None
  return first_index


def run_all(run_dict, X_train, y_train):
  result_dict = dict()
  for method in run_dict:
     model = run_dict[method]()
     model.fit(X_train, y_train)
     # model.test(X_test, y_test)
     if method == 'LogisticRegression':
        result_dict[method] = {
           "Loss":
               model.get_loss(X_train,
               y_train)
        }
     else:
        result_dict[method] = {
           "Loss":
               model.get_loss(X_train,
               y_train),
           "Loss History":
               model.loss_history,
           "Iteration":
               len(model.loss_history),
           "Thres Index":
               get_index(model.loss_history,
               result_dict['LogisticRegression']['Los
        }
     print("{} done, within {}
         iterations".format(method,
                                         result_dict[me
                                         if
                                             'Iteration
                                             in
                                             result_dic
                                             else
                                             "Unknown")

  dump_dict = dict()
  for method in result_dict:
     dump_dict[method] =
         result_dict[method].copy()
```

```
        if "Loss History" in
            dump_dict[method]:
            dump_dict[method].pop("Loss
                History")

    with open("result.json", "w") as
        outfile:
        # json.dump(dump_dict, outfile)
        print(dump_dict, file=outfile)

    return result_dict


def plot(result_dict):
    plt.figure(figsize=(12, 8))

    for method in result_dict:
        if method == 'LogisticRegression':
            # Plotting a horizontal bar
                for LogisticRegression
            plt.axhline(y=result_dict[method]["Loss"],
                color='blue',
                linestyle='-',
                label=f'{method}
                Threshold')
        else:
            # Plotting loss history for
                other models, limit to
                first 200 iterations
            plt.plot(result_dict[method]["Loss
                History"][:100],
                label=method)
    plt.ylabel('Loss')
    plt.xlabel('Iteration')
    plt.title('Loss of Models (First 100
        Iterations)')
    plt.legend()
    plt.show()

    # Setting a logarithmic scale for
        the Y-axis
    plt.figure(figsize=(12, 8))
    plt.yscale('log')
    for method in result_dict:
        if method == 'LogisticRegression':
            plt.axhline(y=result_dict[method]["Loss"],
                color='blue',
                linestyle='-',
                label=f'{method}
                Threshold')
        else:
            plt.plot(result_dict[method]["Loss
                History"], label=method)

    plt.ylabel('Loss (log scale)')
    plt.xlabel('Iteration')
    plt.title('Loss of Models (Log
        Scale)')
    plt.legend()
    plt.show()


def main():
    # Usage
    np.random.seed(43)
    X_train, X_test, y_train, y_test =
        load_and_preprocess_data()

    run_dict = {
        "LogisticRegression": lambda:
            LogisticRegression(learning_rate=0.01,
            max_iter=20000, tol=1e-6),
        # "AllCoordinateDescent": lambda:
            AllCoordinateDescent(
        # learning_rate=5,
            max_iter=20000,
            tol=1e-6,lr_schedule=('decreasing',0.01)),
        "RandomCoordinateDescent":
            lambda:
            RandomCoordinateDescent(
           learning_rate=20,
               max_iter=20000, tol=1e-9,
               lr_schedule=('constant',
               0.00)),
        #
            "RandomCoordinateDescent_Decreasing":
            lambda:
            RandomCoordinateDescent(
        #    learning_rate=5,
            max_iter=20000, tol=1e-6,
            lr_schedule=('decreasing',
            0.01)),
        "WeightedLargestCoordinateDescent":
            lambda:
            WeightedLargestCoordinateDescent(
           learning_rate=20,
               max_iter=20000, tol=1e-9,
               lr_schedule=('constant',
               0.00)),
        "CyclicCoordinateDescent":
            lambda:
            CyclicCoordinateDescent(
           learning_rate=20,
               max_iter=20000, tol=1e-9,
               lr_schedule=('constant',
               0.00)),
        # "CyclicCoordinateDescent":
            lambda:
            CyclicCoordinateDescent(
        #    learning_rate=5,
            max_iter=20000, tol=1e-6,
            lr_schedule=('decreasing',
            0.01)),
        "AStarCoordinateDescent": lambda:
            AStarRandomCoordinateDescent(
           learning_rate=50,
               max_iter=20000, tol=1e-9,
               lr_schedule=('constant',
               0.00))
    }

    result_dict = run_all(run_dict,
        X_train, y_train)
    plot(result_dict)

    # # Logistic Regression
    # lr =
        LogisticRegression(learning_rate=0.1,
        max_iter=20000, tol=1e-6)
    # lr.fit(X_train, y_train)
    # lr.test(X_train, y_train)
    # # lr.test(X_test, y_test)
    # print("Logistic Regression
        Weights:", lr.weights)
    # print("Logistic Regression Loss:",
        lr.get_loss(X_train, y_train))

    # # Coordinate Descent
```

```
    # rcd =
      RandomCoordinateDescent(learning_rate=20,
      max_iter=20000, tol=1e-9,
    #
      lr_schedule=('constant', 0.00))
    # rcd.fit(X_train, y_train)
    # rcd.test(X_train, y_train)
    # # rcd.test(X_test, y_test)
    # print("RandomCoordinateDescent
      Weights:", rcd.weights)
    # print("RandomCoordinateDescent
      Loss:", rcd.get_loss(X_train,
      y_train))
    # print("Converge after {}
      iterations".format(len(rcd.loss_history)))


if __name__ == "__main__":
    main()
```