CSE 251A Project 2 : Coordinate Descent

Doheon Kim
dok015@ucsd.edu

Abstract

In this paper we aim to solve unconstrained optimization problem $\min L(w)$ through Coordinate Descent where w $\in R^d$. There are various ways to solve this optimization problem by doing Gradient descent, Stochastic Gradient descent, and we aim to use coordinate descent to achieve similar milestone and plot loss vs iteration of our Coordinate Descent method as well as picking random point in coordinate descent to compare between each descent method.

1 High Level Description and Implementation

1.1 Logistic Regression

Logistic Regression is a optimization method that uses logistic(sigmoid) function to classify binary problems. It aims to find the weights of the target variable at each iteration and this ideally minimizes the loss function, or the objective function.
The process is quite simple, and begins with

1. initializing the weight to zero.
2. Use Gradient Descent to find the steepest gradient that would decrease the loss and update the weight accordingly.
3. Depending on the learning rate, iterations, and bias term, loss converges to a certain value and stops.

This is the figure I got for Logistic Regression Plot after trying the sklearn logistic Regression with 400 iterations of the data.

1.2 Random Coordinate Descent

1. Initializes weight to zero vectors
2. At each iteration of the step, choose a random coordinate out of all the dimensions.
3. Calculate the partial derivative function of the loss function with respect to the coordinate that is chosen.

$$h_\theta(x) = g(\theta^T x)$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Figure 1: Sigmoid Function Visualization

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

Figure 2: Sigmoid Function Code

Cost function in logistic regression is:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log(h_\theta(x^i)) + (1-y^i) log(1-h_\theta(x^i))]$$

Vectorized implementation:

$h = g(X\theta)$

$J(\theta) = \frac{1}{m}(-y^T log(h) - (1-y)^T log(1-h))$

The gradient of the cost is a vector of the same length as θ where j^{th} element (for $j = 0, 1, \ldots, n$) is defined as follows:

$$\nabla J(\theta) = \frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^i) - y^i) \cdot x_j^i)$$

Vectorized: $\nabla J(\theta) = \frac{1}{m} \cdot X^T \cdot (g(X\theta) - y)$

Figure 3: Loss Function Visualization

```
def losses_acc(X,y,weight,b):
  m = y.shape[0]
  h = sigmoid(np.dot(X,weight.T)+b)
  loss = 0
  for i in range(len(y)):
    if y[i] ==0:
      loss -= np.log(1-h[i])
```

Figure 4: Loss code

```
def predict(X,weights):
  predicts = sigmoid(X.dot(weights.T))
  return [1 if p > 0.5 else 0 for p in predicts]
```

Figure 5: Predict function

```
for i in range(400):
  logistic_model = linear_model.LogisticRegression(C=1e10, max_iter=1000)
  logistic_model.fit(X, y)
```

Figure 6: LogisticRegression using sklearn

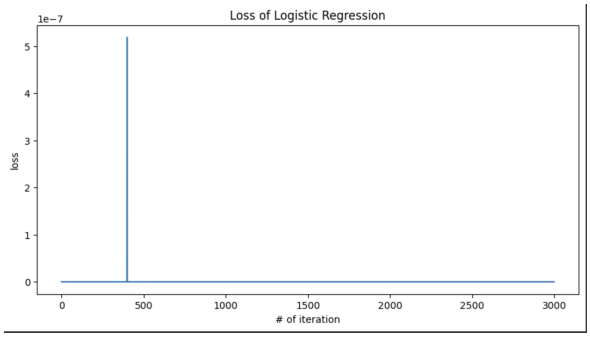

Figure 7: LogisticRegression Plot Loss v Iteration

4. Update the coordinate and the weight each time the partial derivative reduces which minimizes close to zero.
5. Repeat above steps (2),(3),and (4) until it converges.

After writing some code and updating the step sizes accordingly, we could find out that the loss v iteration graph for Random Coordinate Descent looks like this. For Random Coordinate Method,

Algorithm Random Coordinate Descent Method
Input: $x_0 \in R^n$ //starting point
Output: x

set x := x_0

for k := 1, ... **do**
choose coordinate $i \in \{1, 2, \ldots, n\}$, uniformly at random
update $x^{(i)} = x^{(i)} - \frac{1}{L_i} \nabla_i f(x)$
end for

Figure 8: Random Coordinate Descent Pseudo-code

```
def random_coordinate_descent(X, y, weights, bias, learning_rate, max_iter):
    m, n = X.shape

    loss_history = []

    for iteration in range(max_iter):
        # Randomly select a coordinate to update
        j = np.random.randint(0, n)

        h = sigmoid(np.dot(X, weights) + bias)


        # Compute the gradient for weight j
        gradient_w = np.dot((h - y), X[:, j]) / m
        weights[j] = weights[j] - learning_rate * gradient_w


        loss = compute_loss(y, h)
        loss_history.append(loss)

    return weights, bias,loss_history

# Initialize weights and bias
weights = np.random.randn(n)
bias = np.random.randn()
```

Figure 9: Random Coordinate Descent code

coordinate to update is chosen at random from all the coordinates in 13 dimensions. It is selected at random probabilities. The w_i is then chosen by getting the partial derivative and putting in the coordinates so that loss function minimizes. With the update on step size and learning rate of

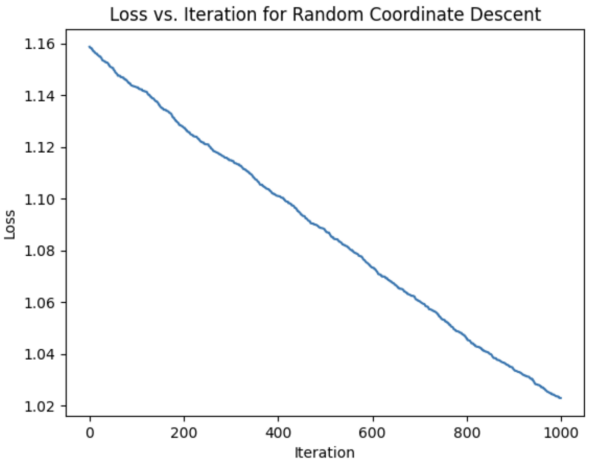

Figure 10: Random Coordinate Descent loss graph

hyperparameters, weight updated.

2 Greedy Coordinate Descent

Greedy Coordinate Descent is a method where we choose w_i so that the loss function is greatly minimized, and to always update the partial derivative with the steepest value. We want to choose best of both algorithm, Greedy and Random so that we can reduce the loss without having to do more iterations. Therefore, one of the paper we found was Efficient Greedy Coordinate Descent by Variable Partitioning.

Algorithm 2 Hybrid coordinate descent

Input: $\mathbf{x}^{(0)}, \mathcal{B} = \{B_i\}_{i=1}^k$.
for $t = 0, 1, 2, \ldots$ **do**
$I = \emptyset$
for $j = 1, 2, \ldots, k$ **do**
[**Random rule**] uniform randomly choose a $i_j \in B_j$
and let $I = I \cup \{i_j\}$
end for
[**Greedy rule**] $i \in \arg\max_{j \in I} |\nabla_j f(\mathbf{x}^{(t)})|$
$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{1}{L_i} \nabla f_i(\mathbf{x}^{(t)}) \mathbf{e}_i$
end for

Figure 11: Hybrid Coordinate Descent pseudo code

We first need to randomly choose a coordinate and then use the greedy rule, such that gradient is the steepest improvement in the loss, not considering the step sizes over iterations.

For greedy and hybrid method, we want to choose the highest gradient coordinate when finding the partial derivate respect to the loss function. we also use the random coordinate method to find

```
hybrid_coordinate_descent(f, grad_f, L, x0, B, max_iter=100):

x = np.array(x0)
for t in range(max_iter):
    I = set()
    # Random
    for B_j in B:
        i_j = np.random.choice(B_j)
        I.add(i_j)

    # Greedy
    gradients = {i: grad_f(x)[i] for i in I}
    i_max = max(gradients, key=lambda i: abs(gradients[i]))
```

Figure 12: Hybrid Coordinate Descent pseudo code

```
def compute_loss(y, h):
    return -np.mean(y * np.log(h) + (1 - y) * np.log(1 - h))
```

Figure 13: Hybrid Coordinate Descent pseudo code

the most viable coordinate to choose so that more coordinates could be explored.

Updating of w_i would involve finding the path that would be able to minimize the loss function the greatest, but also implementing random method where calculating the loss function with respect to the coordinate that is selected.

3 Convergence

We would know from our pseudo code and the code that we implemented for the graph that the greedy method, or the the random coordinate descent method would both perform quite well into choosing the coordinate next to minimize the loss function by selecting the one with the steepest gradient. Each iteration, the loss will be minimized, almost reduced to zero at the end. For the method to converge faster, the loss function needs to be more smooth and convex, meaning differentiable and have Hessian functions, also have local minimum as global minimum at each stages. The inital choice of point could also play huge role in the convergence.

4 Critical Evaluation and Conclusion

We have used the UCI Wine Dataset to explore the Coordinate descent method, and we have 71 and 59 points in the first two targets to understand the optimization problem that have been proposed to solve. We tested the methods on Logistic Regression problem, introduced all the pseudocode, the method established, discussing the papers that we have been able to find in. We use sklearn's library in doing logisticregression and get two different graphs for random and hybrid coordinate descents. We merge these two graphs together, and found out that the Hybrid Coordinate descent, the one with both greedy selection of the coordinates using steepest gradient and randomly choosing the points works out the best, by reducing the iteration of the descent greatly while the loss is significantly reduced. We could have also tried various other methods to minimize the losses such as stochastic gradient descent, or gradient descent that was not explicitly described here. However, our Hybrid descent method from the Variable Partitoning paper, seemed to be a promising method.

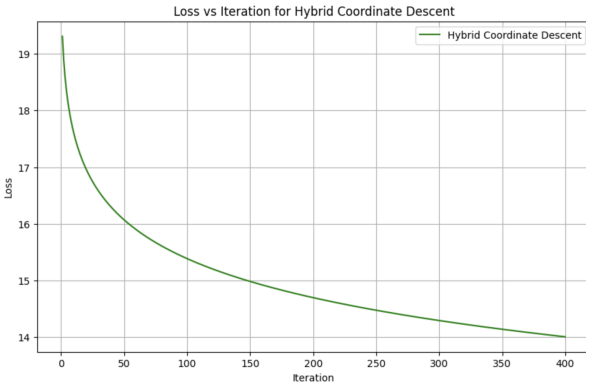

Figure 14: Hybrid Coordinate Descent Graph

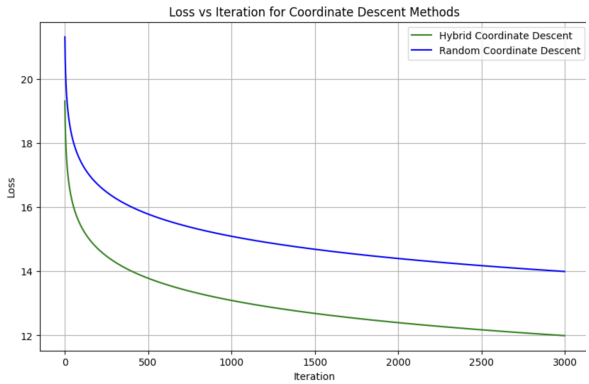

Figure 15: HCD vs RCD graph

5 Citations

[1] Coordinate Descent Algorithms, Stephen J.Wright

[2] Efficient Greedy Coordinate Descent via Variable Partitioning, Huang Fang, Guanhua Fang, Tan Yu, Ping Li

[3] Wikipedia.org/wiki/Random coordinate descent

[4] https://www.cs.ubc.ca/ schmidtm/Courses/5XX-S22/S3.pdf

[5] https://arxiv.org/pdf/1502.04759.pdf