Greedy Coordinate Descent

Yousef Hawas
yhawas@ucsd.edu

Abstract

This report contains a description of a greedy algorithm used for coordinate descent in the context of logistic regression on a wine dataset. The algorithm is compared to a baseline coordinate descent algorithm where the coordinates are picked at random.

1 Algorithm Description

The proposed coordinate descent algorithm must tackle two different problems. The first problem to solve is how to initialize our model weights. The second is how to pick which weights coordinate to perform an update on.

1.1 Weights Initialization

For weights initialization, the algorithm initializes all the weights to 0. The intuition behind this is extracted when observing the sigmoid function which is at the core of how logistic regression works. When observing the sigmoid function, it appears that there is a certain range where the function has active gradients. This region is when the values are close to 0. As the values grow larger, the gradient vanishes or tends to zero. Therefore, when initializing the weights to 0 we are guaranteeing that as the algorithm commences there will be gradients with which to perform the update.

1.2 Picking the Coordinate

After initialization, our coordinate descent algorithm commences. In each iteration of the algorithm, a decision has to be made on which weight coordinate to update. The proposed algorithm is greedy in nature, it aims to take the largest possible step towards the loss function's minimum. This is done by first computing the partial derivative of the loss function with respect to each coordinate. Once we have our gradients, we observe them and take note of the coordinate at which we have the largest gradient in magnitude. That coordinate will be the one at which the update will be performed.

2 Convergence

In order for the proposed coordinate descent algorithm to eventually terminate, there have to be certain conditions that are met. First of all, our method exclusively uses first derivatives to perform the update, so the loss function must be differentiable everywhere. Furthermore, the loss function must be convex as if it is not, our method would get stuck in local minima and fail to reach optimal loss. Finally, the learning rate has to be set carefully so as to prevent the loss from continually overshooting the minimum. If all these conditions are satisfied, the algorithm should successfully reach the optimal loss.

3 Experiment and Results

3.1 Data and Preprocessing

In order to evaluate the performance of the proposed algorithm, we used a wine dataset, utilizing 2 of its 3 classes in order to make the task a binary classification. The labels are then set to -1 and 1 in order to utilize a simplified form of the log loss and it's derivative. Finally, we preprocess our features by utilizing sklearn's StandardScaler transformer, bounding the feature values between [-1,1].

3.2 Loss Function and Gradient

For our loss function, we use a logarithmic loss that is defined as follows:

$$L(w) = \sum_{i=1}^{N} \ln\left(1 + e^{-y_i(w.x_i)}\right) \tag{1}$$

As for the gradients that are used to perform the update, they are derived as follows:

$$\frac{\partial L}{\partial w_k} = \frac{-y.x_{ik}}{1 + e^{y_i(w.x_i)}} \tag{2}$$

After computing the gradient for a certain weight, we apply the gradient using the following update rule where η is our learning rate.

$$w_{k,t+1} = w_{k,t} - \eta \frac{\partial L}{\partial w_{k,t}} \tag{3}$$

3.3 Comparison of Methods

At the beginning of the experiment, sklearn's LogisticRegression module is used and fit to the data in order to obtain our L* or optimal loss. The loss of the obtained model is **0.0002793**. Then we move on to coordinate descent, applying both our algorithm and random coordinate descent. Our end goal was to compare the greedy algorithm to that which picks coordinates at random, and showcase that the greedy one would reach convergence faster. The following graph, shows the losses obtained from each algorithm, compared to our optimal loss obtained through sklearn.

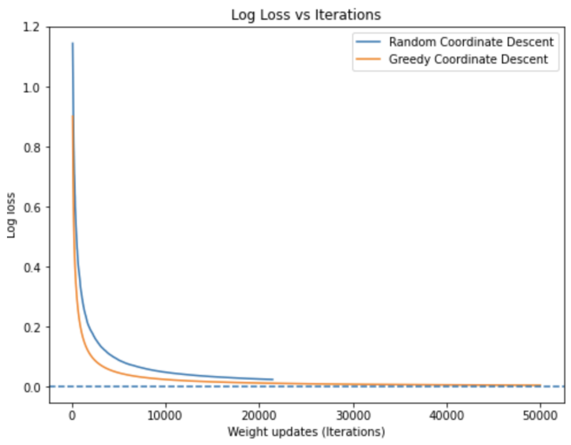


Figure 1: Log loss (y axis) vs iterations (x axis). In orange we have the proposed greedy coordinate descent and in blue we have random coordinate descent. The dashed blue line is the optimal loss L*

As shown in the figure, the random coordinate descent terminates after approximately 20,000 iterations after failing to significantly improve on the loss according to our predefined termination conditions (failing to make significant loss improvement in 10 consecutive updates). On the other hand, the proposed greedy coordinate descent continually improves on the loss at a rate which is faster than that of random coordinate descent.

4 Critical Evaluation

While the proposed algorithm is shown to work well in the context of logistic regression tasks. It can lack flexibility for other tasks or loss functions. As stated in convergence section, the algorithm requires that the loss function be both differentiable over its domain as well as convex in nature. This means that the algorithm would fail to converge with certain non-convex or non-continuous loss functions. In order to combat this rigidity, there are techniques we can incorporate in our method that would make it converge on any type of functions.

In order to make the proposed algorithm converge on loss functions that aren't differentiable on the entire domain, we can utilize sub-gradients. We can use these sub-gradients to perform updates at points where our gradient is not defined.

As for convexity of loss functions, inspiration can be drawn from other state of the art gradient descent algorithms that work well with non-convex functions (Adam, RMSProp etc.). These algorithms make approximations for the second derivative through the concept of "momentum". Applying momentum in our coordinate descent would allow us to escape local minima and avoid getting stuck.

Project 2

February 19, 2022

[214]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import numpy as np
import math
import random
random.seed(0)
np.random.seed(0)
```

[202]:
```
X_train, Y_train = [], []
with open('wine.data') as f:
    for line in f.readlines():
        split_line = line.split(',')
        split_line[-1] = split_line[-1].replace('\n', '')
        label = int(split_line[0])
        feats = [float(x) for x in split_line[1:]]
        if label == 1 or label == 2:
            X_train.append(feats)
            if label == 1:
                label = -1
            else:
                label = 1
            Y_train.append(label)
```

[203]:
```
scaler = StandardScaler()
scaler.fit(X_train)
scaled_X_train = scaler.transform(X_train)
```

[204]:
```
log_reg = LogisticRegression(penalty="none",verbose=0, fit_intercept=False)
```

[205]:
```
log_reg.fit(scaled_X_train, Y_train)
```

[205]:
```
LogisticRegression(fit_intercept=False, penalty='none')
```

[206]:
```
def loss_function(weights, x_tr, y_tr):
    loss = 0
    for i in range(len(x_tr)):
        loss += math.log(1 + math.exp(-y_tr[i] * np.dot(weights, x_tr[i])))
    return loss
```

[207]:
```
coefs = log_reg.coef_
l_star = loss_function(coefs[0], scaled_X_train, Y_train)
print(l_star)
```

```
0.0002793446179302977
```

[117]:
```
def calculate_gradient(coord, weights, x_tr, y_tr):
    grad = 0
    for i in range(len(x_tr)):
        grad += -(y_tr[i] * x_tr[i][coord]) / (1 + math.exp(y_tr[i]*np.
 ↪dot(weights, x_tr[i])))
    return grad
```

[237]:
```
def random_coord_descent(x_tr, y_tr, lr =1e-1):
    losses = []
    updates = []
    prev_loss = 1e8
    weights = [0] * len(x_tr[0])
    i=1
    cnt = 0
    indices = range(13)
    while True:
        coord_to_update = random.sample(indices,1)[0]
        grad = calculate_gradient(coord_to_update, weights, x_tr, y_tr)
        weights[coord_to_update] = weights[coord_to_update] - lr*grad
        loss = loss_function(weights, x_tr, y_tr)
        losses.append(loss)
        updates.append(i)
        if prev_loss - loss > 1e-7:
            cnt=0
        else:
            cnt+=1
        if i==50000:
            break
        if cnt==10:
            break
        prev_loss = loss
        i+=1


    return weights, losses, updates
```

[238]:
```
random_coord_weights, losses, updates = random_coord_descent(scaled_X_train,␣
 ↪Y_train)
```

[240]:
```
print(losses[-1])
```

```
0.024496247174775113
21429
```

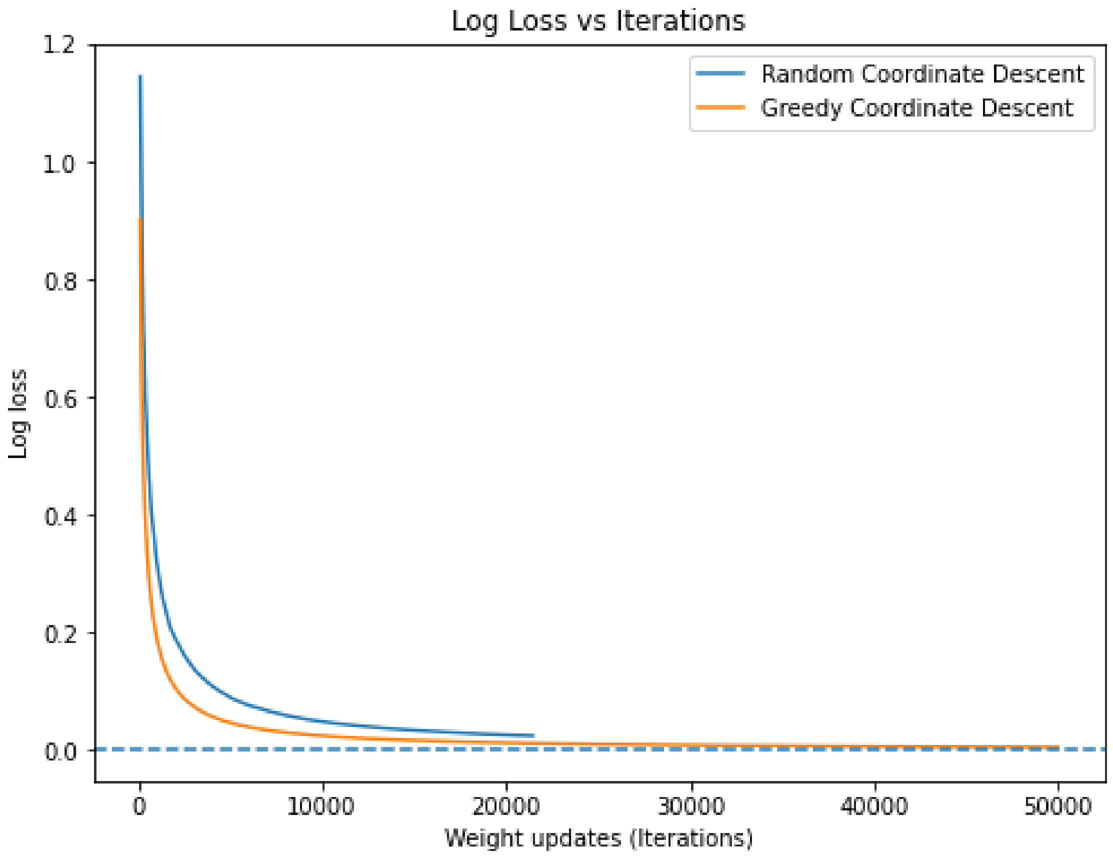
[241]:
```
def greedy_coord_descent(x_tr, y_tr, lr =1e-1):
    losses = []
    updates = []
    prev_loss = 1e8
    weights = [0] * len(x_tr[0])
    i=1
    cnt = 0
    indices = range(13)
    while True:
        grads = [calculate_gradient(i, weights, x_tr, y_tr) for i in␣
 ↪range(len(weights))]
        abs_grads = [abs(grad) for grad in grads]
        coord_to_update = np.argmax(abs_grads)
        weights[coord_to_update] = weights[coord_to_update] -␣
 ↪lr*grads[coord_to_update]
        loss = loss_function(weights, x_tr, y_tr)
        losses.append(loss)
        updates.append(i)
        if prev_loss - loss > 1e-7:
            cnt=0
        else:
            cnt+=1
        if i==50000:
            break
        if cnt==10:
            break
        prev_loss = loss
        i+=1


    return weights, losses, updates
```

[242]:
```
greedy_coord_weights, greedy_losses, greedy_updates =␣
 ↪greedy_coord_descent(scaled_X_train, Y_train)
print(greedy_losses[-1])
```

```
0.0052062996061273265
```

[243]:
```
import matplotlib.pyplot as plt
```

[246]:
```
plt.figure(figsize=(8,6))
plt.plot(updates[100:], losses[100:], label= 'Random Coordinate Descent')
plt.plot(greedy_updates[100:], greedy_losses[100:], label = 'Greedy Coordinate␣
 ↪Descent')
plt.axhline(0.00027934, ls='--')
plt.legend()
plt.xlabel('Weight updates (Iterations)')
plt.ylabel('Log loss')
plt.title("Log Loss vs Iterations")
plt.show()
```



```
[ ]:
```