Optimization for Data Science - report

Ruslan Nuriev

May 19, 2024

1 Introduction

In this homework, I implemented Gradient Descent and Block Coordinate Gradient Descent with Gauss-Southwell methods that were shown in the class by professor Rinaldi. In my sequential code, I first defined some functions that are going to be crucial for our methods to work. This project helped me to dive deeper into the models we saw in the class and to get a better idea of how they work.

2 Analysis of the code

2.1 Loss function

I defined loss function same as the shown in homework guide; I used negative log-likelihood to minimize the loss of our model:

$$\min_{x \in R^{d \times k}} \sum_{i=1}^{m} \left[x_{b_i}^T a_i + log\left(\sum_{c=1}^{k} e^{x_c^T a_i} \right) \right]$$

To avoid for loops in the calculation of the loss function, I tried to adopt a vectorized style. I first defined a variable that is the dot product of our data matrix and weight matrix. Each element of this product will be useful for calculating the $x_{b_i}^T a_i$ part. Then, again, instead of using a for loop for calculating the exponential term in the function, I took the sum of the above mentioned variable and summed them across the rows. Since I had the dot product of $x^T a_i$ previously, I chose the x_{b_i} columns by subsetting the dot product by each row and each b_i. Lastly, the sum of these terms gave me the final loss value.

2.2 Softmax function

Softmax function is used to calculate the likelihood of each training example. The function is given below:

$$Pr[b_i | a_i, X] = \frac{e^{x_{b_i}^T a_i}}{\sum_{c=1}^{k} e^{x_{c_i}^T a_i}}$$

The probabilities that we get from this function are so-called the softmax probabilities of the data. Each row of the dot product of data and weight matrix will sum up to 1. For numerical stability, I extracted the maximum value in each row from all the other values in softmax function definition.

2.3 Gradient function

This is the most important part of our work. The gradient will be used ro define all of our algorithms and it will also be used in our Armijo line search method. The gradient related to each of the weight is defined as:

$$\frac{\partial f(x)}{\partial X_{jc}} = -\sum a_{ij} \left[I(b_i = c) - \frac{e^{x_{c_i}^T a_i}}{\sum_{c'=1}^{k} e^{x_{c_i'}^T a_i}} \right]$$

But, once again, instead of doing explicit for loops I vectorized it. I first computed the softmax probabilities of A dot X, then extracted if from the true labels (indicator variable). Then it's just

a simple dot product between A and these differences. Instead of multiplying by a_{ij}, I took the dot product of A^T with respect to the differences between true labels and softmax probabilities.

2.4 Armijo line search

There are 2 line searches used in the code: fixed step-size and Armijo line search. With fixed step-size, I chose a reasonable learning rate for the weight update. But for the Armijo rule, I implemented the below algorithm:

Algorithm 1 Armijo (backtracking) line search

Require: $m = 0$, $f(x_k)$, $\nabla f(x_k)$, $\gamma \in (0, 1)$, $\delta \in (0, 1/2)$, $\alpha = \delta^m \Delta_k$
  while not $f(x_k + \alpha d_k) \leq f(x_k) + \gamma \alpha \nabla f(x_k)^T d_k$ **do**
    $m = m + 1$
    $\alpha = \delta^m \Delta_k$
  end while

The algorithm decreases the step size at each step if the "while not" is not met. Basically, it suggests that a "good" step length is such that we have "sufficient decrease" in f at the new point.

2.5 Prediction and evaluation functions

They are pretty straightforward functions. Predict function takes the updated weights and calculates the softmax of the data. Then it chooses the maximum value in the row to be the label. Evaluation function takes the predict function and calculates the accuracy of the model by comparing it to the ground-truth labels.

2.6 Gradient Descent function

This is our first algorithm. It implements the gradient descent algorithm with two choices of step size (fixed line search and Armijo rule). It first creates one-hot-encoded version of the labels to be passed on to the gradient function (because we subtract it from the softmax output and softmax output has columns equal to the number of classes). Then it creates a random normal matrix, called W. This will be our weight matrix. Then it sets some empty lists to be filled in later on (lists for accuracies and losses). Then there's an error counter which acts as a monitor. So if the loss doesn't improve over some epochs, it breaks the loop and returns the last value of the weight (alongside loss and accuracy). The main part of the algorithm is like:

Algorithm 2 Gradient Descent

Require: $k = 0$, $\nabla f(W_k)$, W_k, α
  while $k <$ max iter. **do**
    If W satisfies some specific condition, then stop
    $k = k + 1$
    $W_k = W_k + \alpha * \nabla f(W_k)$
  end while

Here α is either fixed size or Armijo function. It is a straightforward algorithm that updates the weight matrix at each iteration. Then it returns the updated weight, loss, and accuracy for the input data.

2.7 Block Coordinate Gradient Descent - Gauss-Southwell rule

The algorithm is a little different from the previous one. Instead of updating the whole weight matrix with the whole gradient, we choose a block of the weight matrix to be updated at each iteration. And this block is the one with the highest gradient norm. Let's define $i_k = \text{argmax}_{i \in B} ||\nabla_j f(x_k)||$ where $B = \{1, 2, ..., k\}$ (k classes). After choosing the row with the highest norm, we only update the relevant row of the weight matrix. It sequentially eliminates the highest norm gradients at each iteration. The algorithm is:

Algorithm 3 Block Coordinate Gradient Descent - Gauss Southwell

Require: $k = 0$, $\nabla f(W_k)$, W_k, α
  while $k <$ max iter. **do**
    If W satisfies some specific condition, then stop
    $k = k + 1$
    Pick the block $i_k = \text{argmax}_{i \in B} ||\nabla_j f(x_k)||$
    $W_{k+1} = W_k + \alpha * U_{i_k} \nabla_{i_k} f(W_k)$
  end while

3 Results

3.1 Iris Dataset

The Iris flower dataset (Fisher, 1936) is one of the most popular datasets in the machine learning sphere. It is one of the most used datasets for softmax regression. It is a 150×4 dataset with 3 different classes in the target variable. The columns of the data contain different measurements of the flower. Here are the results for the 2 algorithms for 2 different line searches.

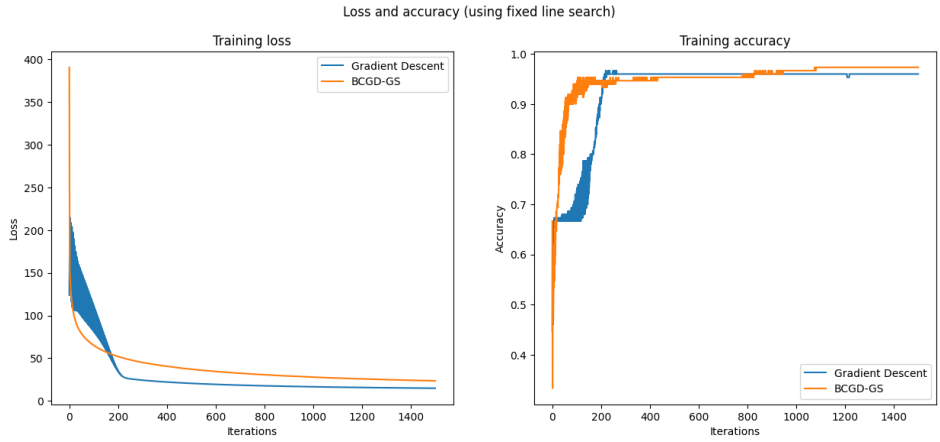

Figure 1: The loss and accuracy plots for fixed line search.

As we can see from the results, both of the algorithms have low loss and high accuracies (loss close to zero and accuracy around 97%). The fixed line search was chosen to be 0.001 in this example, but other values were tried as well. This rate hits the sweet-spot between being too big and too small. And the maximum allowed iteration was 1500, in which case it seems the models have run to the allowed maximum point. Let's now take a look at the CPU times of the algorithms on the next page.

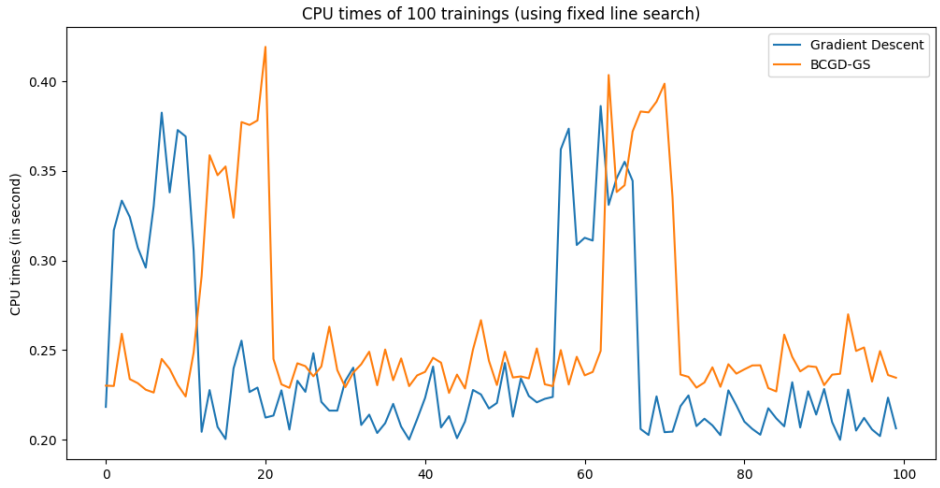

Figure 2: The CPU time plots for fixed line search.

As we can see from the figure, there is too much of deviation, but the average CPU time for both of the models are around 0.25 with BCGD-GS being a little high. I ran the models for 100 times to get a reasonable average. Using Armijo line search instead of fixed, makes the CPU time much slower as we can see from the plot, but the accuracies and losses are almost the same.

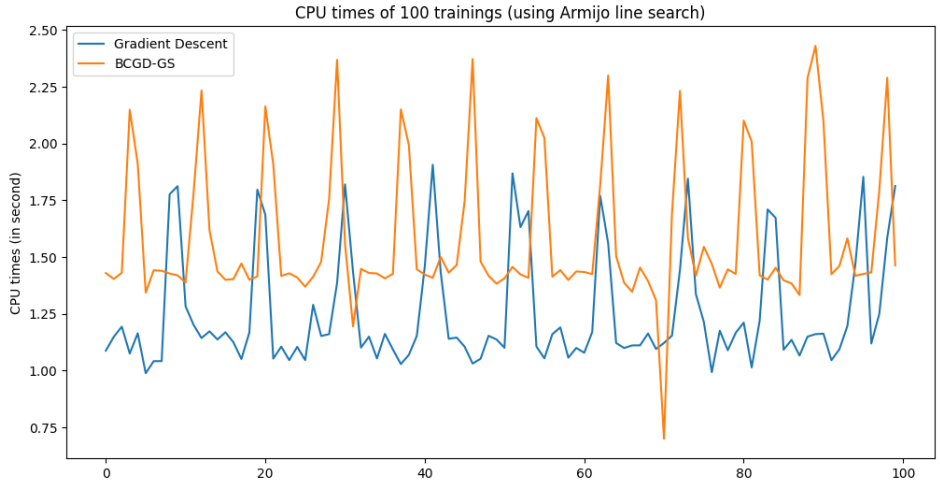

Figure 3: The CPU time plots for Armijo line search.

3.2 Dry Beans Dataset

This dataset consists of 16 features (13,611 samples) and a target variable with 7 classes. The columns measure different aspects of the dry beans, such as form, shape, type and structure. Since it is a huge dataset (I used 20% of the data) compared to the previous one, I iterated for 15 times to measure the CPU time. As seen from the plot, the accuracies are almost the same for 2 methods, being around 91%. We can see that gradient descent has stopped training after 180th iteration which means the loss didn't improve at least for 10 epochs.

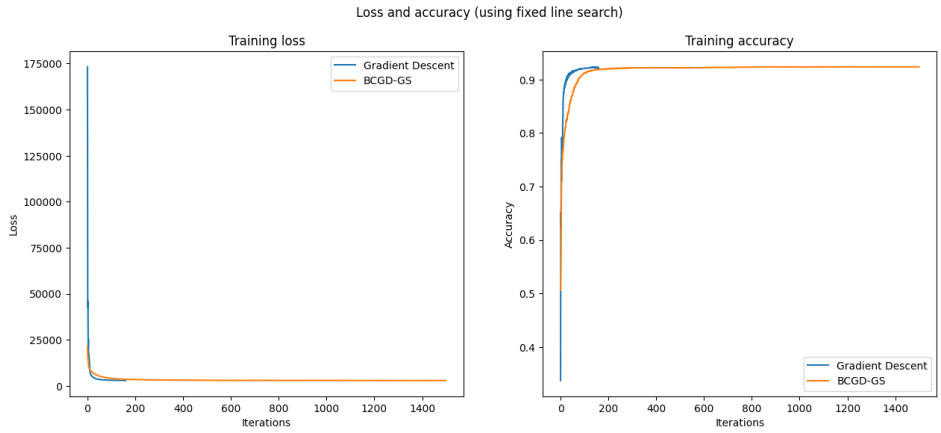

Figure 4: The loss and accuracy plots for fixed line search.

The CPU time is higher for this dataset, as expected. Average time required for gradient descent is 5.7 seconds, whereas for BCGD-GS method it's equal to 25.05 seconds.

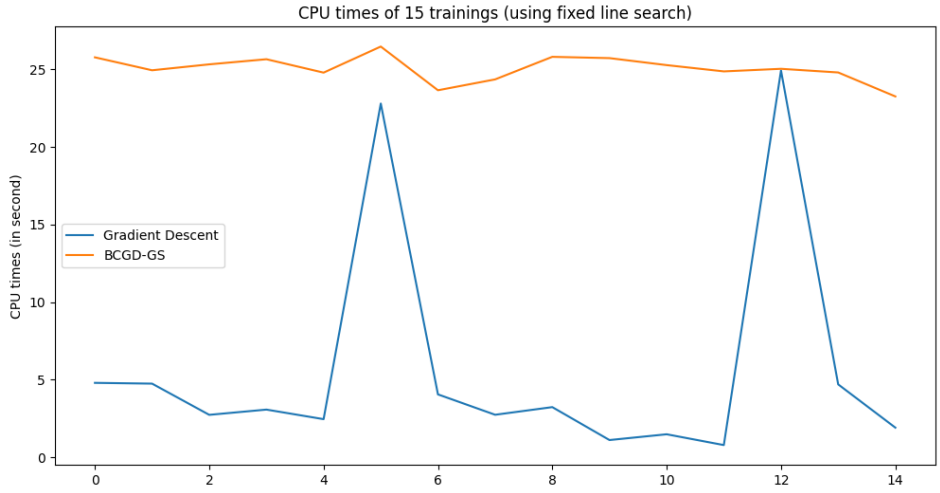

Figure 5: The CPU time plots for fixed line search.

3.3 Wine Quality Dataset

Wine quality dataset is another popular dataset for multi-class classification. It consists of 11 features and 6497 samples. The target variable has 7 different qualities. I splitted the data into train and test sets, with train set getting 67% of the data. The accuracies of the models were low compared to the previous datasets, but it is because the complexities cannot be captured by our models. I tried the LogisticRegression class from scikit-learn on this dataset, and the accuracy came out similar to mine. The training set and test set accuracies were similar to each other, so there wasn't any overfitting (around 0.45 for training, 0.43 for test). Here's the plot for the loss-accuracy for wine quality dataset:

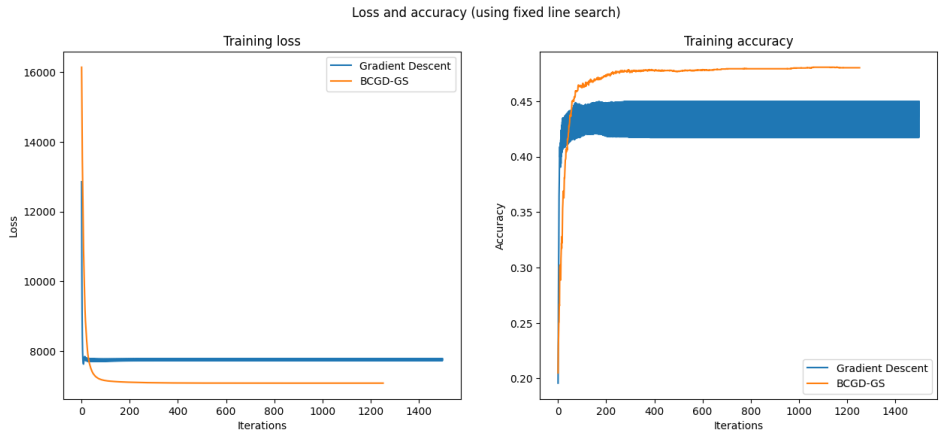

Figure 6: The loss and accuracy time plots for fixed line search.

CPU times for BCGD-GS were substantially lower for the wine dataset compared to the previous one, but the times for gradient descent are similar with 3 seconds of difference:

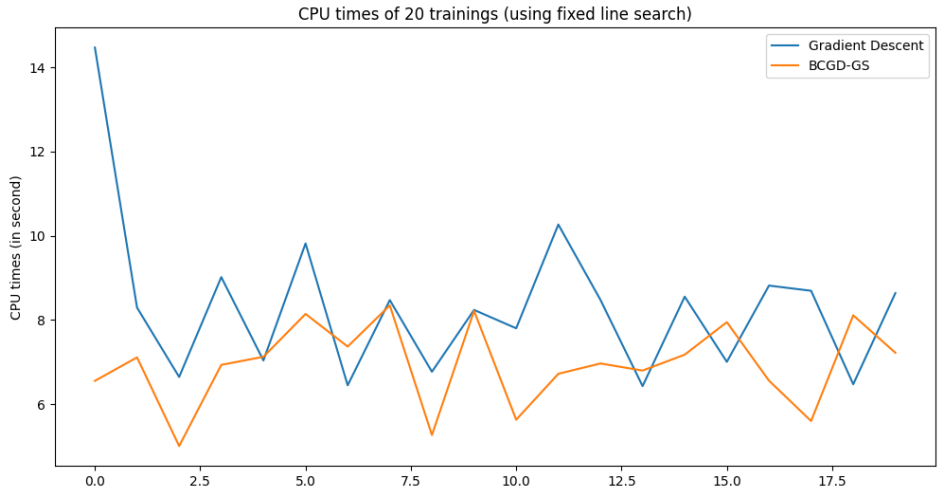

Figure 7: The CPU time plots for fixed line search.

The average time for gradient descent was 8.32 and for BCGD-GS it was 6.94. It is surprising that the CPU time for BCGD-GS was almost always lower than the gradient descent whereas in the other datasets it was the contrary.

4 Conclusion

In this homework, I tried to implement the Gradient Descent and Block Coordinate Gradient Descent (Gauss-Southwell) methods. The models were pretty good, they performed well on most datasets. The results were almost identical with the LogisticRegression class from scikit-learn. The code can further be optimized and be turned to a class. But I think it's better to leave it as a sequential, functional code in order to better understand what each part of the code does.