Gradient Methods for Semi-Supervised Learning Problems

Optimization For Data Science (2022/2023)

Cerbaro Anna
2087619
anna.cerbaro@studenti.unipd.it
d'Addario Alessia
2086506
alessia.daddario@studenti.unipd.it

Papadopulos Eleni
2079423
eleni.papadopulos@studenti.unipd.it
Ronzoni Alice
2076675
alice.ronzoni@studenti.unipd.it

Contents

1 Introduction

In this homework, we will study a case of semi-supervised learning for binary classification.
The problem is defined as follows: we have u unlabeled points x_j, $j = 1, 2, \dots, u$ and l labeled points $(\bar{x}_i,\ \bar{y}_i)$, $i = 1, 2, \dots, l$ that represent a small percentage of the overall number of examples. Our goal is to predict the label of the unlabeled points by solving an unconstrained minimization problem.
Starting from this setting, we will implement some algorithms to solve our task that will be tested both on a synthetic dataset and on real datasets.

2 Problem Formalization

2.1 Dataset

To solve this problem, we will use a synthetic dataset made of 6000 points. Since we are dealing with a semi-supervised learning problem, only 1% of data will have a label identified by 1 or -1, corresponding respectively to the first and second class.

2.1.1 Data Creation

We have chosen to generate two spirals that intentionally slightly overlap each other, so that some points are closer to the other class and the classification problem is more meaningful and realistic.
Each spiral is made of 3000 points, only 30 of which are labeled.
In order to create the spirals, we simulated polar coordinates

$$x(t) = x_0 + r(t)\cos(t)$$
$$y(t) = y_0 + r(t)\sin(t)$$

using vectors of numbers drawn from the uniform distribution in $[0, 1]$ of according length. In this way, it is possible to show how different spatial configurations can affect the problem.
In our code we chose to use a seed (161) for reproducibility purpose.
For each cluster of points, we independently generated the labeled and unlabeled points, making sure that their centers and radii coincide.

- Class 1 (label 1): circle centered in $(5, 8.6)$ with maximum radius of length 1.2
- Class 2 (label 0): circle centered in $(4.5, 7.4)$ with maximum radius of length 0.7

In addition, we introduced some noise by adding some random vectors of numbers drawn from a normal distribution with mean 0 and variance $1/30$.

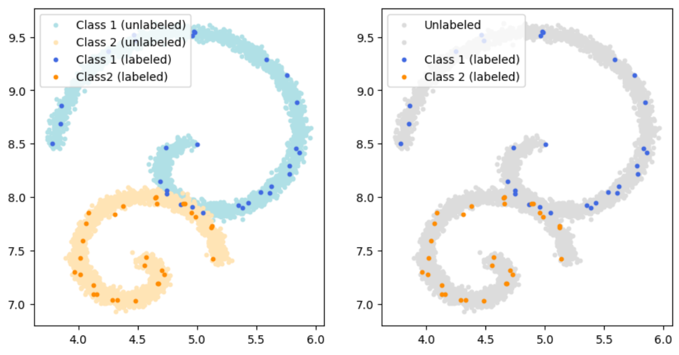


Figure 1: Synthetic Dataset

2.1.2 Similarity Function

The cardinal assumption we make is that examples that have similar features must have similar labels so we had define a similarity function that could encode this property: for this type of problem, we chose to use the inverse of the euclidean distance, to which, in order to avoid the problem of having an indeterminate form, we have added a small epsilon value in the denominator. In this way, points that are closer (more similar) have a higher value in the weights matrix and on the other hand, points that are more distant (less similar) have a lower value in the weights matrix. As further manipulation, we scaled the outputs of the function making them fall in the unitary interval.

$$dist(x_1, x_2) = \frac{1}{(\varepsilon + norm(x_1 - x_2))}$$

We stored in the matrix w_{lu} the similarity between labeled and unlabeled examples and in the matrix $\bar{w}_{uu}$ the similarity between unlabeled examples.

2.1.3 Loss Function

The Loss function for this classification problem is given and it's equal to:

$$f(y) = \sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}(y^j - \bar{y}^i)^2 + \frac{1}{2}\sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(y^i - y^j)^2$$

where w_{ij} is the matrix of the weights given by the similarity between labeled and unlabeled points and $\bar{w}_{ij}$ is the matrix of the weights given by the similarity between unlabeled points.
$\bar{y}^i$ indicates the label of the i-th labeled element and y^i is the label assigned to the i-th unlabeled element.
The convexity of our Loss function is due to the convexity of its components: since it is the summation of quadratic functions, it is convex.

2.1.4 Gradient of the Loss Function

Starting from the loss function, we calculated the gradient w.r.t y^j:

$$\nabla_{y^j} f(y) = 2\sum_{i=1}^{l} w_{ij}(y^j - \bar{y}^i) + 2\sum_{i=1}^{u} \bar{w}_{ij}(y^j - y^i)$$

In our implementation we decided to use a different formulation to make the computations smoother:

$$\underbrace{\underbrace{2\Big(\sum_{i=1}^{l} w_{ij} + \sum_{i=1}^{u} \bar{w}_{ij}\Big)}_{P0} y^j}_{P1} - 2\underbrace{\sum_{i=1}^{l} w_{ij}\bar{y}^i}_{P2} - 2\underbrace{\sum_{i=1}^{u} \bar{w}_{ij} y^i}_{P3}$$

Eventually, we opted for a faster formulation in which we compute each addend in vectorial form:

$$2\sum_{i=1}^{l} w_{ij}y^j - 2\sum_{i=1}^{l} w_{ij}\bar{y}^i + 2\sum_{i=1}^{u} \bar{w}_{ij}y^j - 2\sum_{i=1}^{u} \bar{w}_{ij}y^i$$

2.1.5 Hessian

The Hessian **H** of our function is defined as follows:

$$\nabla_{y^j y^i} f(y^j) = \begin{cases} -2\bar{w}_{ij} & i \neq j \\ 2\Big(\sum_{i=1}^{l} w_{ij} + \sum_{i=1}^{u} \bar{w}_{ij} - \bar{w}_{jj}\Big) & i = j \end{cases}$$

Its computation is necessary for the definition of the Lipschitz constants L and L_i, as we are assuming that the loss function has Lipschitz continuous gradient, even coordinate-wise.
This means that

$$||\nabla f(x) - \nabla f(y)|| \leq L||x - y|| \quad \forall x, y \in R^n$$

and this is valid also for the univariate functions we obtain by fixing one variable at the time.

2.1.6 Model Evaluation

As a way of evaluating the performance of our models, we have compared the value of the loss function against the needed number of iterations and CPU time.
Although we are aware that in real life circumstances it is not possible to compute the accuracy as we don't have labels to compare the results to, we chose to count anyway the number of labels correctly classified (after being rounded to 1 if $y^j \geq 0$ and -1 otherwise).

2.2 Algorithms

To reach our goal, we had to minimize the loss function and to do so we needed some suitable procedures. In particular, we chose three different algorithms, each one with different implementations:

1. Gradient Descent
   - with fixed learning rate ($1/L$)
   - with Armijo Rule
   - with exact search line
2. Block Coordinate Gradient Descent with Randomized Rule (BCGD)
   - with learning rate equal to $1/L_i$ (uniform sampling)
   - with learning rate equal to $1/L_i$ (non-uniform sampling)
3. Block Coordinate Gradient Descent with Gauss-Southwell rule (BCGD GS)
   - with learning rate equal to $1/L$
   - with learning rate equal to $1/L_i$

2.3 Gradient Descent

The aim of gradient descent is to predict the correct label of the unlabeled points. The basic idea of this algorithm is that we compute the descent direction in the current point, we take a step in that direction and then we update the vector of points according to the following formula:

$$y_k = y_{k-1} - \alpha \nabla f(y_{k-1})$$

where $\alpha > 0$ is the given step size.
We repeat these three steps until the following stopping conditions are satisfied:

1. maximum number of iterations, set to 25000;

2. $|f(y_{t+1}) - f(y_t)| < tol$, where tol is a value of tolerance set at 1e-3.

2.3.1 Fixed Learning Rate

The function *gradient_fixed* calculates, at each iteration, the gradient using the previous vector of labels and then it updates the labels using the learning rate equal to $\frac{1}{L}$, where L is the Lipschitz constant that, in this case, consists of the largest eigenvalue of the Hessian matrix relative to our loss function.
In fact, f has Lipschitz continuous gradient if and only if $||\nabla^2 f(y)|| \leq L$: since the loss function we aim to minimize is convex and twice differentiable, its Hessian matrix is positive semidefinite and therefore its norm is equal to its largest eigenvalue.
The following rule keeps running until one of the stop conditions is satisfied:

$$y_k = y_{k-1} - \frac{1}{L} \nabla f(y_{k-1})$$

During the first iteration, the vector of labels corresponds to a random generated vector.

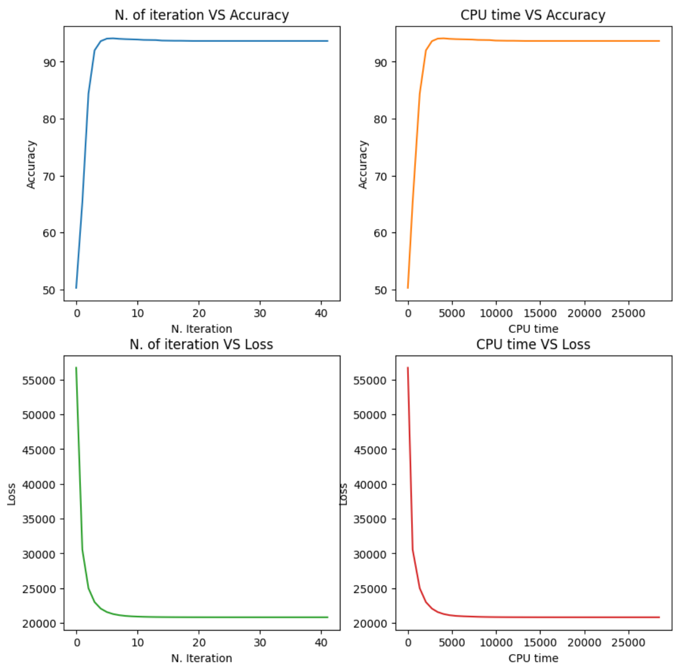


Figure 2: Plots of the performance of GD with fixed learning rate

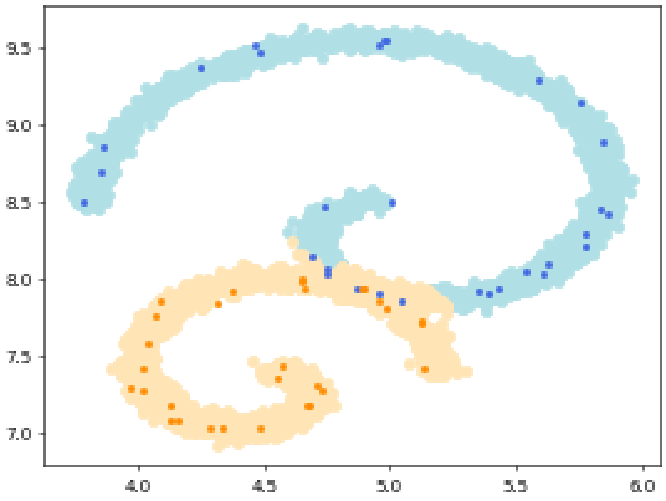


Figure 3: Plot of the classes predicted by GD with fixed learning rate

2.3.2 Armijo Rule

The function *GD_Armijo* calculates the target labels using the Armijo rule. The function *Armijo* calculates the learning rate using the following formula:

$$\alpha = \delta^m \triangle_k \quad for \quad m = 0, 1, \dots,$$

where δ is a constant value chosen between $(0, 1)$ and $\triangle_0$ is the starting step size. In our algorithm these constants have the following values:

$$\delta = 0.5 \quad \gamma = 1e-4$$

The update of the alpha value keeps going on until the stop condition

$$f(x_k + \alpha d_k) \le f(x_k) + \gamma \alpha \nabla f(x_k)^T d_k$$

is satisfied and we can set $\alpha_k = \alpha$.
In the formula d represent the descent direction (the antigradient).

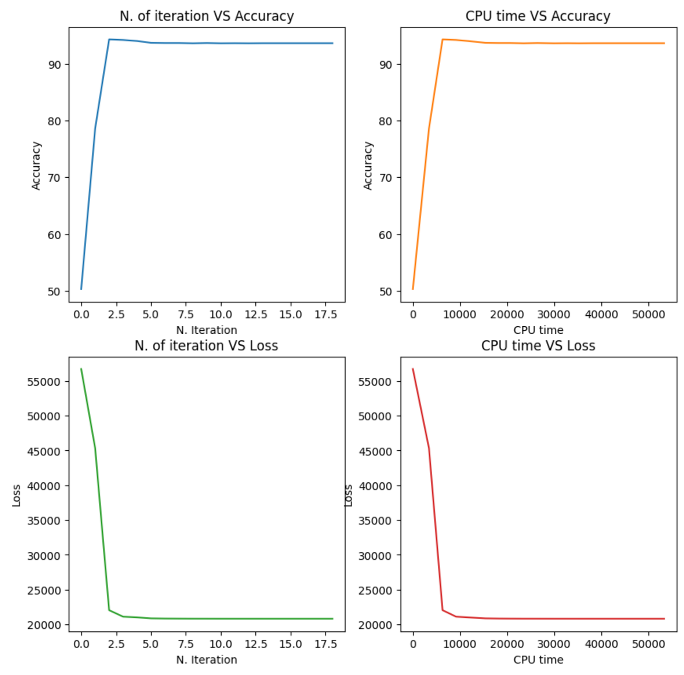


Figure 4: Plots of the performance of GD with Armijo rule

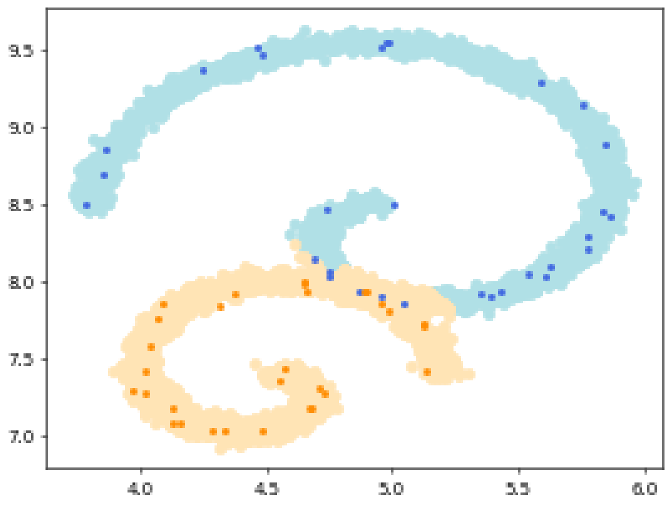


Figure 5: Plot of the classes predicted by GD with Armijo rule

2.3.3 Exact Line Search

The function *gradient_descent_ELS* calculates the unknown labels of the unlabeled points. The function *exact_line_search* calculates the learning rate using the exact line search rule, that consist of computing the derivative of $f(x + \alpha d)$ with respect to α, finding the value of the step size α that minimizes the loss at each iteration as showed in formula (1)

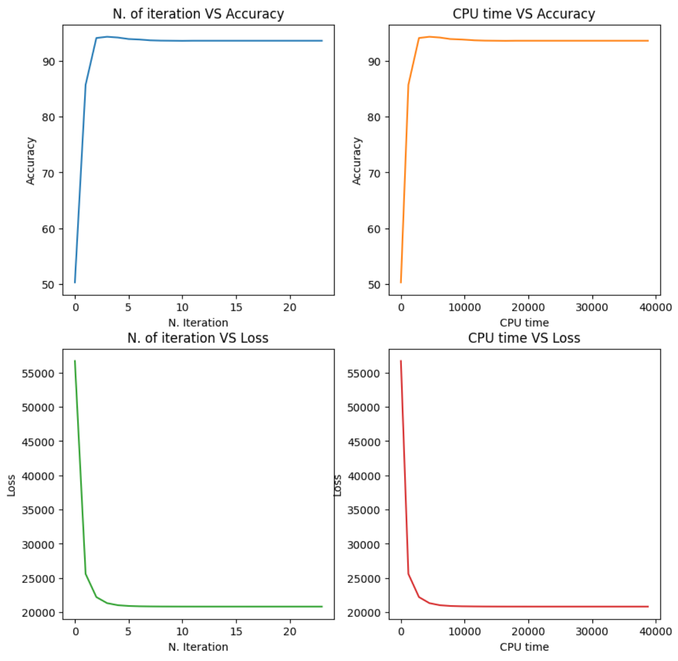


Figure 6: Plots of the performance of GD with exact line search

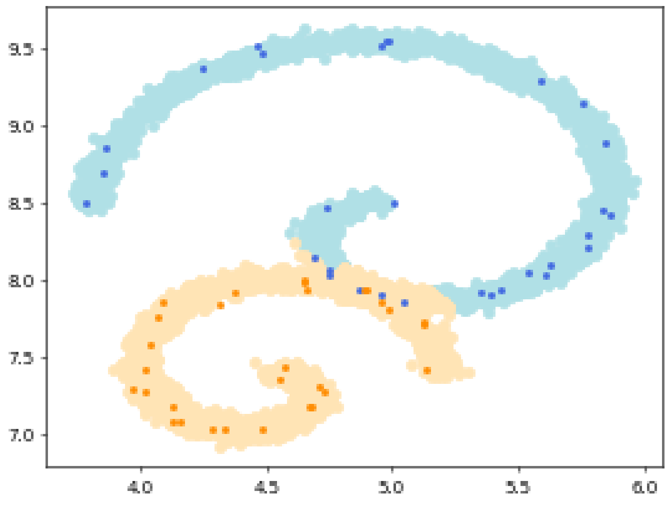


Figure 7: Plot of the classes predicted by GD with exact line search

Starting from

$$f(x+\alpha d) = \sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}(y^j + \alpha d^j - \bar{y}^i)^2 + \frac{1}{2}\sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(y^i + \alpha d^i - y^j - \alpha d^j)^2 \implies \tag{1}$$

$$f'_\alpha(x+\alpha d) = \sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} 2 d^j (y^j + \alpha d^j - \bar{y}^i) + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(d^j - d^i)(y^i + \alpha d^i - y^j - \alpha d^j)$$

$$= 2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} d^j (y^j - \bar{y}^i) + 2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}\alpha (d^j)^2 + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(d^j - d^i)(y^i - y^j) + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(d^j - d^i)^2\alpha$$

Solving $f'_\alpha(x+\alpha d) = 0$ for α we get:

$$\Big(2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}(d^j)^2 + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(d^j - d^i)^2\Big)\alpha = -\Big(2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} d^j (y^j - \bar{y}^i) + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(d^j - d^i)(y^i - y^j)\Big)$$

from which:

$$\alpha = \frac{-\Big(2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} d^j (y^j - \bar{y}^i) + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(d^j - d^i)(y^i - y^j)\Big)}{\Big(2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}(d^j)^2 + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}(d^j - d^i)^2\Big)}$$

2.3.4 Conclusions and Comparison

All the three algorithms achieve an accuracy of 95.65%. As we can see from the Figure 11, gradient descent with fixed learning rate and exact line search are the ones that make the Loss decrease in the fastest way in terms of CPU time. Gradient descent with Armijo rule, on the other hand, exhibits a slower growth in accuracy compared to the other algorithms and takes the longest CPU time to converge, likely due to the computation of the learning rate at each iteration.
It is worth noticing that gradient descent with exact line search calculates the α at each iteration, but it takes a shorter CPU time to converge probably because the function *exact_line_search* is well optimized.
On the other hand, gradient descent with fixed learning rate has the biggest number of iterations to reach convergence but at the same time it takes the smallest CPU time.

In view of what has been said, we consider gradient descent with exact search line to be the algorithm with the best performance for this specific task as it gives the best trade-off between the number of iterations (23) and a small CPU time, despite the additional effort of computing α every time.

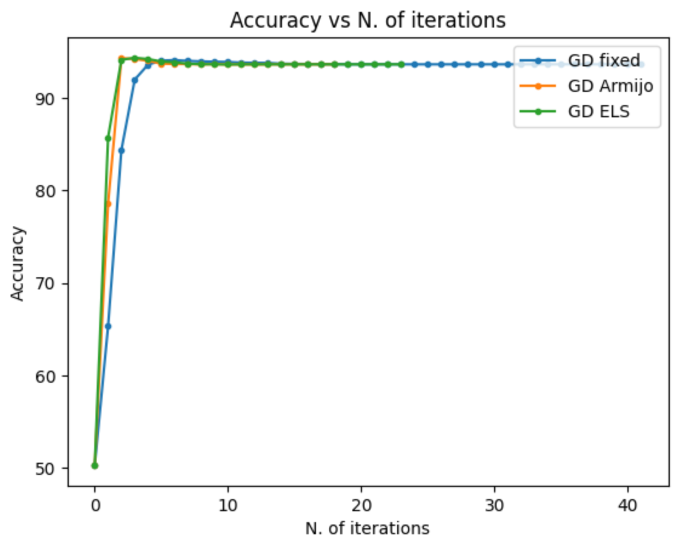


Figure 8: GD: Accuracy VS N. of iterations

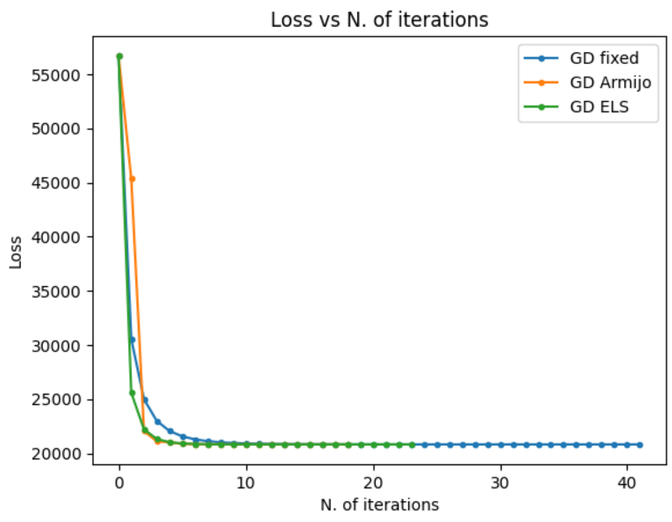


Figure 9: GD: Loss VS N. of Iterations

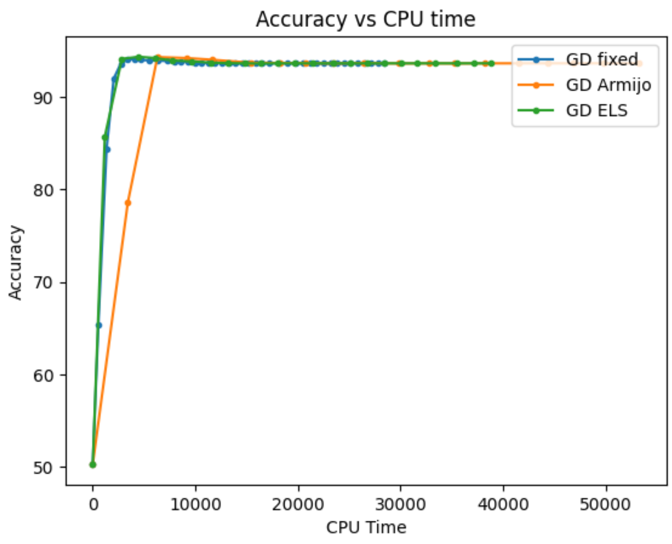


Figure 10: GD: Accuracy VS CPU time

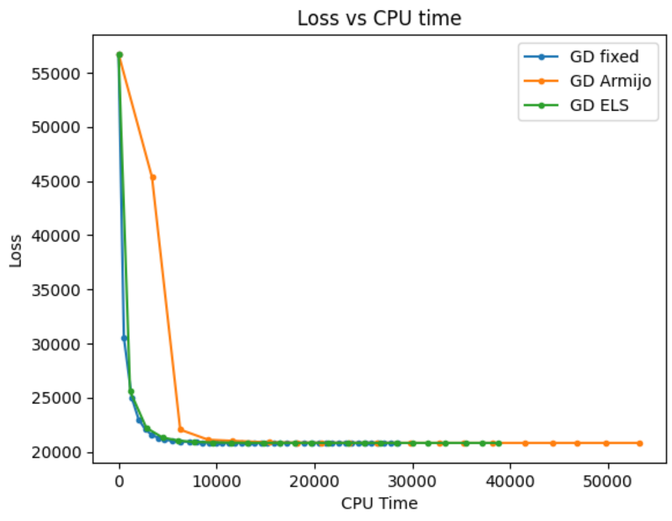


Figure 11: GD: Loss VS CPU time

2.4 Block Coordinate Gradient Descent

Block Coordinate Gradient Descent (BCGD) is a general algorithm designed to solve unconstrained optimization problems. In this case, instead of updating the whole vector, this algorithm updates only a few coordinates based on different rules.
The basic idea behind BCGD is that we divide vector coordinates into b blocks of a given dimension and then perform the gradient descent algorithm only on those blocks.
The update rule of this algorithm is the following:

$$y_k = y_{k-1} - \alpha_k U_{i_k} \nabla_{i_k} f(y_{k-1})$$

where U_{i_k} is the i-th block of the identity matrix at the k-th iteration and α_k is the learning rate. As suggested, we implemented the size of the block as 1, so in this case we have the same number of blocks as the number of unlabeled points.
Other than the standard code, we implemented a second procedure for each method: in fact, due to the amount of blocks we are using, they can be costly algorithms. That's why we did not fully compute the gradient at each iteration from scratch but rather, since only one of the coordinates of the vector is changed, it is possible to update it given the previous gradient and the step size.

Let the gradient be:

$$2\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}\Big) y^j - 2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}\bar{y}^i - 2\sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij} y^i$$

Let k be the drawn index, so that $y_{t+1}^k = y_t^k - \alpha$ with $\alpha_k U_{i_k} \nabla_{i_k} f(y_{t-1}^k)$
If $k = j$:

$$\begin{aligned}
&\nabla f_{t+i}(y^j) = \\
&= 2\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}\Big)(y_t^k - \alpha) - 2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}\bar{y}^i \\
&- 2\Big(\bar{w}_{1j}y_t^1 + \cdots + \bar{w}_{jj}(y_t^j - \alpha) + \cdots + \bar{w}_{uj}y_t^u\Big) \\
&= 2\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}\Big)y_t^k - 2\alpha\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}\Big) - 2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}\bar{y}^i \\
&- 2\Big(\bar{w}_{1j}y_t^1 + \cdots + \bar{w}_{jj}(y_t^j) + \cdots + \bar{w}_{uj}y_t^u\Big) - 2\alpha\bar{w}_{jj} \\
&= \nabla f_t(y^k) - 2\alpha\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}\Big) - 2\alpha\bar{w}_{jj} \\
&= \nabla f_t(y^k) - 2\alpha\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij} - \bar{w}_{jj}\Big)
\end{aligned}$$

if $k \neq j$:

$$\begin{aligned}
&\nabla f_{t+i}(y^j) = \\
&= 2\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}\Big)y_t^k - 2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}\bar{y}^i \\
&- 2\Big(\bar{w}_{1j}y_t^1 + \cdots + \bar{w}_{jj}(y_t^j - \alpha) + \cdots + \bar{w}_{uj}y_t^u\Big) \\
&= 2\Big(\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij} + \sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}\Big)y_t^k - 2\sum_{i=1}^{l}\sum_{j=1}^{u} w_{ij}\bar{y}^i \\
&- 2\Big(\sum_{i=1}^{u}\sum_{j=1}^{u} \bar{w}_{ij}y_t^j\Big) + 2\alpha w_{ij} \\
&= \nabla f_t(y^j) + 2\alpha w_{ij}
\end{aligned}$$

Also, for the same reason mentioned above, it was possible to define the function *element_loss*, that, given an index and the value of the vector in said index, returns the contribution of each single coordinate to the total loss function.

$$\sum_{j=1}^{u} \underbrace{\Big(\sum_{i=1}^{l} w_{ij}(y^j - \bar{y}^i)^2 + \frac{1}{2}\sum_{i=1}^{u} \bar{w}_{ij}(y^j - y^i)^2\Big)}_{element\_loss(y_j, j)}$$

There are different ways to choose the block at each iteration. In this paper, we are going to analyze randomized BCGD and Gauss-Southwell BCGD. Given that in BCGD algorithm with randomized rule, it may happen that it is randomly drawn a number such that $|f(y_{k+1}) - f(y_k)| < tol$, we thought it was best to adopt the following stopping criteria:

1. Maximum number of iterations, set to 20000;
2. $||\nabla f(y_k)|| \leq tol$, where tol equals 1e-3.

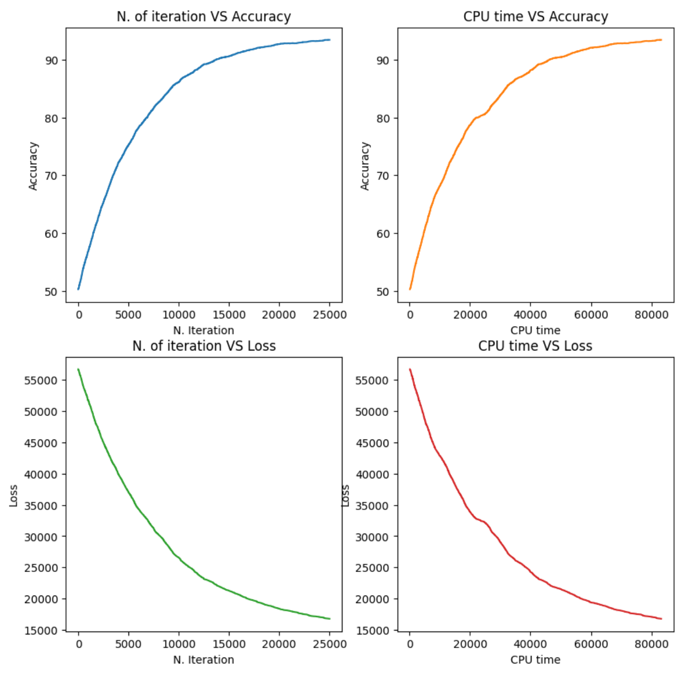
Regarding the choice of the stepsize, we considered a total of two different choices:

1. $\frac{1}{L}$ as in the gradinet descent methods
2. $\frac{1}{L_i}$ where L_i is the Lipschitz constant related to each block.
   So for the aformentioned reasons, L_i is the only eigenvalue of an Hessian matrix (of dimension 1x1) of a univariate function and consequently L_i coincides with the matrix itself. When computing the second derivatives of the loss function with respect to just one variable, it's clear that they correspond to the diagonal of the initial Hessian. Hence

$$L_i = 2\Big(\sum_{i=1}^{l} w_{ij} + \sum_{i=1}^{u} \bar{w}_{ij} - \bar{w}_{ii}\Big)$$

2.4.1 Randomized BCGD

In randomized BCGD the choice of the block b is done randomly.
For this algorithm, we decided to use $\alpha_k = \frac{1}{L_i}$, testing two different versions: in the first one, each block is drawn with uniform probability $P(i_k = i) = \frac{1}{b}$, where b is the number of blocks while in the second algorithm we used Nesterov's version in which each block has probability $P(i_k = i) = \frac{L_i}{\sum L_i}$ to be chosen and in this way blocks with a larger Lipschitz constant are drawn more often.



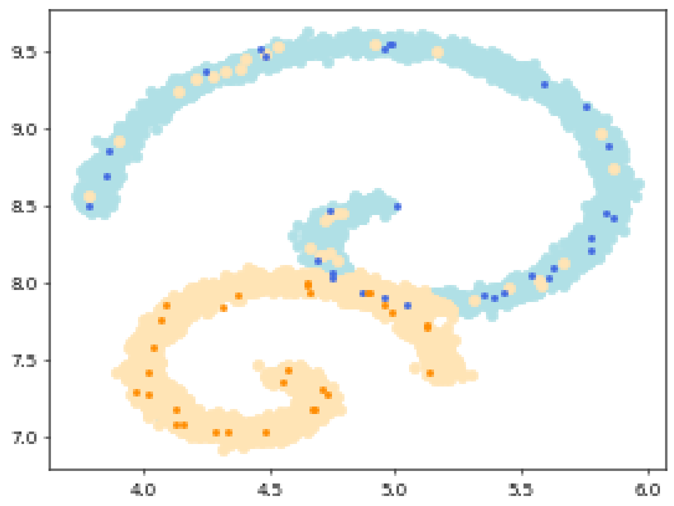
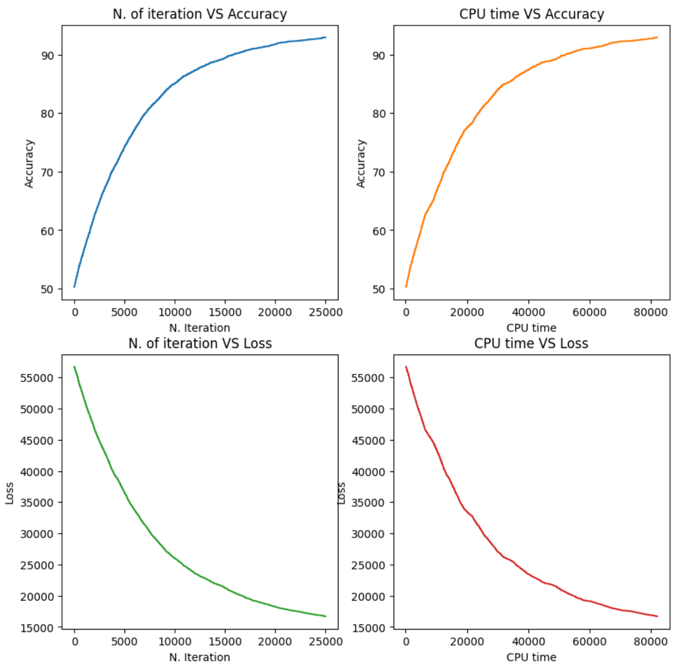


Figure 12: Classes predicted by Random BCGD with Uniform sampling



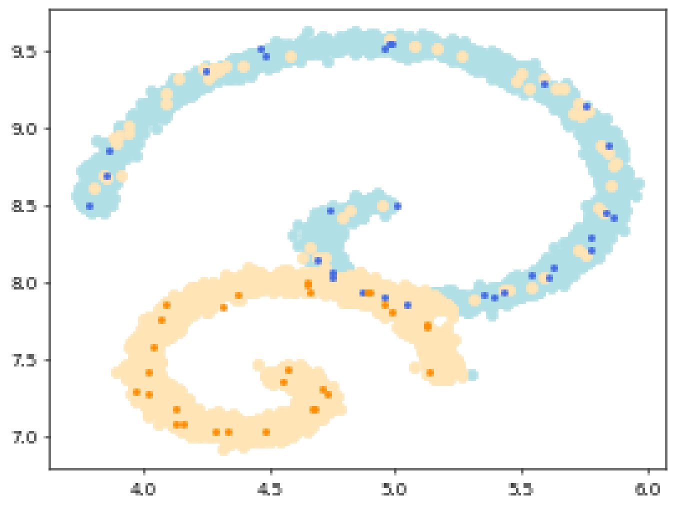


Figure 13: Classes predicted by Random BCGD with non Uniform sampling (Nesterov)

2.4.2 Gauss Southwell BCGD

In Gauss-Southwell BCGD algorithm, we consider the block i that satisfies:

$$i_k = \arg\max_{j \in \{1,\dots,b\}} |\nabla_j f(x_k)|$$

Then, during each iteration, we update only the coordinate that corresponds to the highest absolute value of the gradient. As a learning rate, we decided to use both $\frac{1}{L}$ and $\frac{1}{L_i}$.

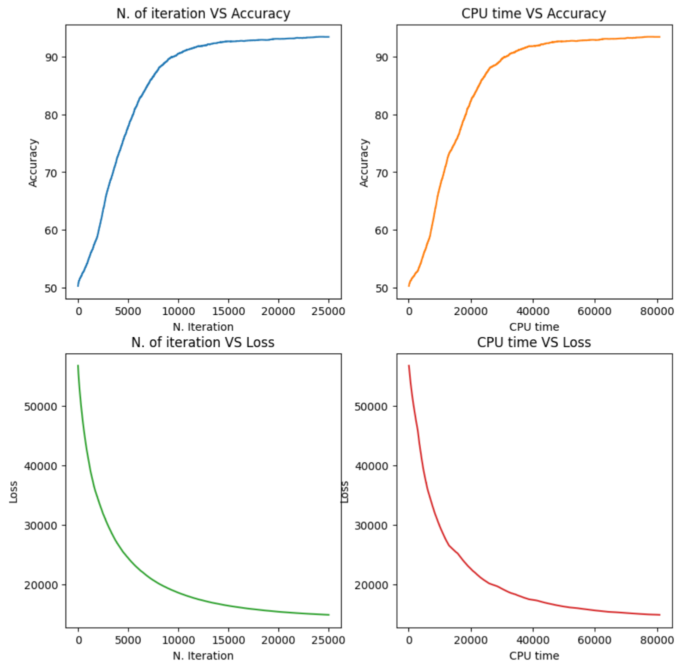


Figure 14: Plots of the performance of BCGD Gauss Southwell with $1/L$

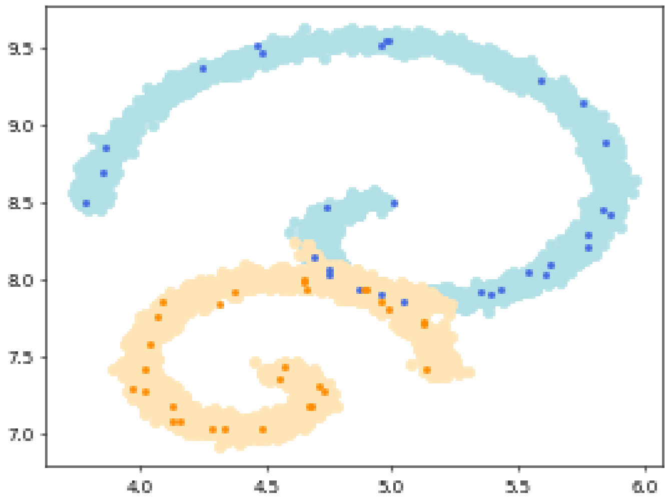


Figure 15: Plot of the classes predicted by BCGD Gauss Southwell with $1/L$

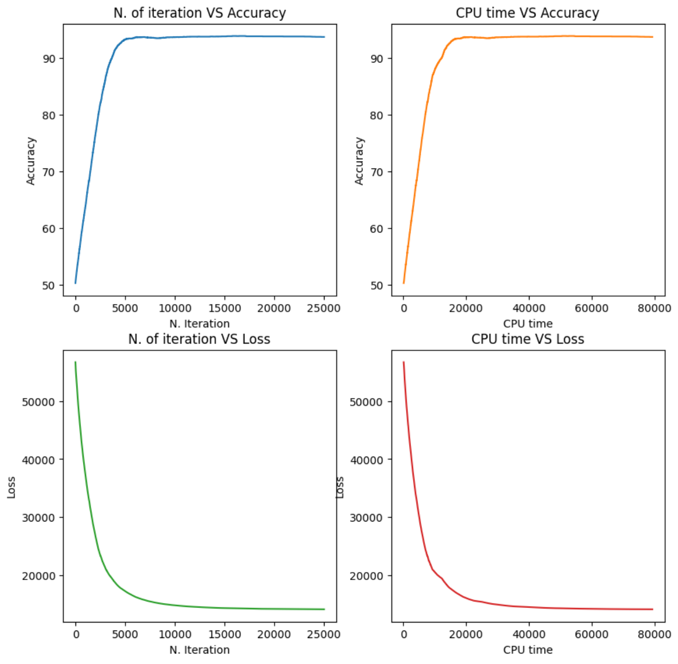


Figure 16: Plots of the performance of BCGD Gauss Southwell with $1/L_i$

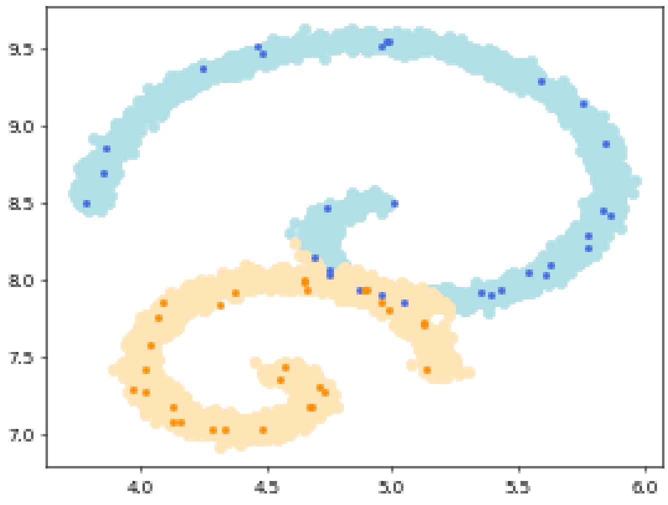


Figure 17: Plot of the classes predicted by Random BCGD Gauss Southwell with $1/L_i$

2.4.3 Conclusions and Comparison

As we can see from the plot 18, the best BCGD algorithm is Gauss-Southwell with the learning rate equal to $1/L_i$.
It reaches an accuracy of 93.68% and from the plot 19 it is clear that it minimizes the Loss in the least amount of iterations reaching an accuracy of 90% before 5000 iterations.
If we consider Gauss-Southwell with learning rate equal to *1/L* it is noticeable that it has a worseele performance with respect to Gauss-Southwell with learning rate equal to $1/L_i$: even though it reaches an accuracy 93.46%, the loss function decreases more slowly. This proves that

calculating the learning rate as $1/L_i$ improves the accuracy compared to $1/L$. Nevertheless, it is still better than Randomized BCGD. In fact, it is clear that dealing with non-deterministic methods can decrease the performances because of the increase of cache missing.
Looking at 20 it is evident that Randomized algorithms behave not that differently from each other but, unexpectedly, Randomized BCGD with uniform sampling performed slightly better (accuracy 93.4%) than non-uniform one (accuracy 92.97%), even if in the other trials that we did, we noticed that it was the other way around as supported by the theory.

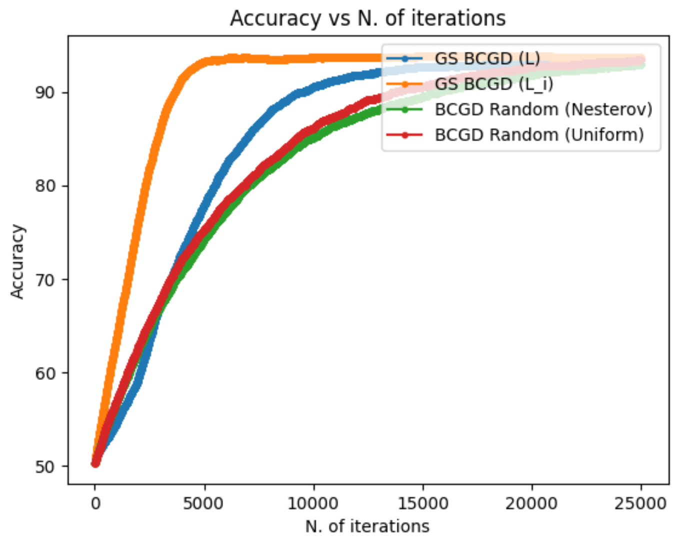


Figure 18: BCGD: Accuracy VS N. of iterations

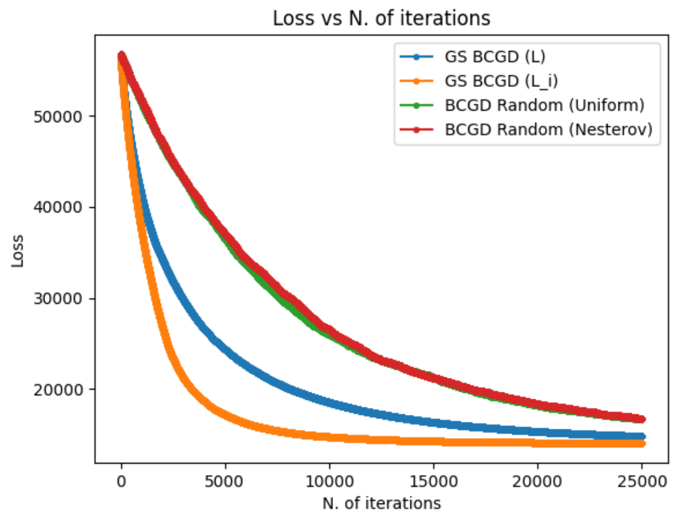


Figure 19: BCGD: Loss VS N. of Iterations

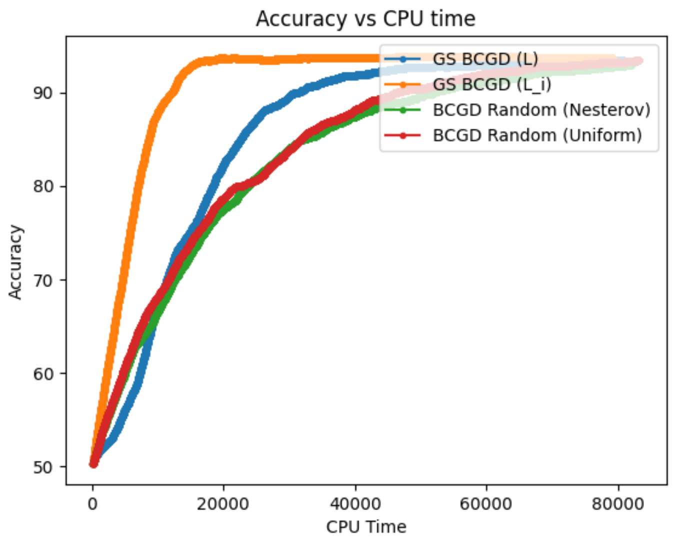


Figure 20: BCGD: Accuracy VS CPU time

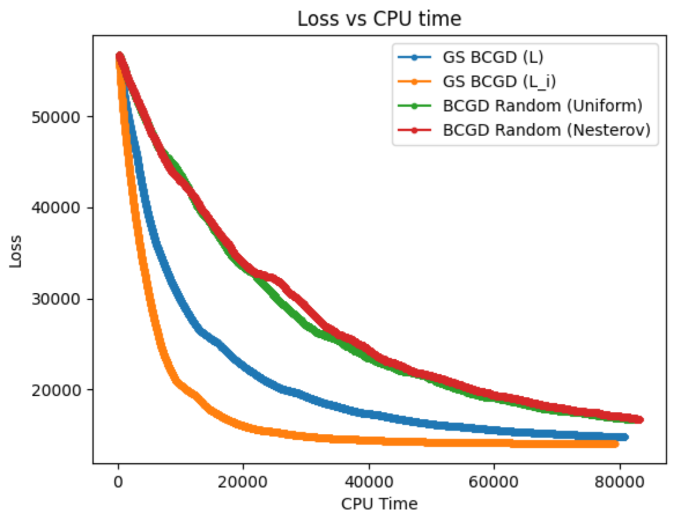


Figure 21: BCGD: Loss VS CPU time

3 Public Dataset

We evaluated our algorithms on a public dataset known as Banknote authentication dataset (`https://archive.ics.uci.edu/ml/datasets/banknote+authentication`), which consists of 1372 samples of images of real and fake banknotes, along with 5 attributes. The target column is called *class* and it classifies real banknotes as 1 and fake ones as -1.
Using the correlation matrix, we selected the two features *'variance of WTI'* and *'skewness of WTI'* that best separate the data points.
Next, we randomly chose 5% of the data points to preserve the original labels, while setting aside the labels for the remaining 95% for the final accuracy calculation.
The dataset do not contain any NaN values.
Additionally, since the weights are based on the

distance between points, the data were min-max-scaled.

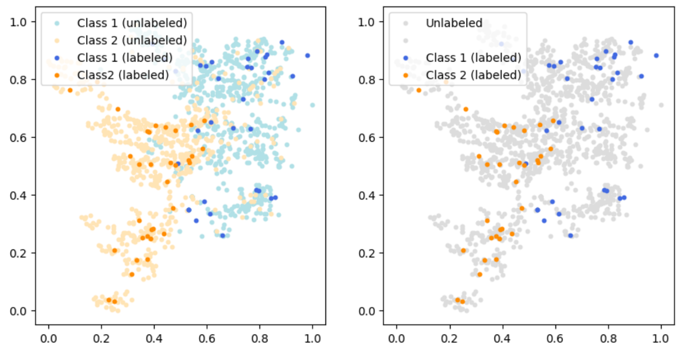


Figure 22: Public Dataset

3.1 Gradient Descent

3.1.1 Fixed Learning Rate

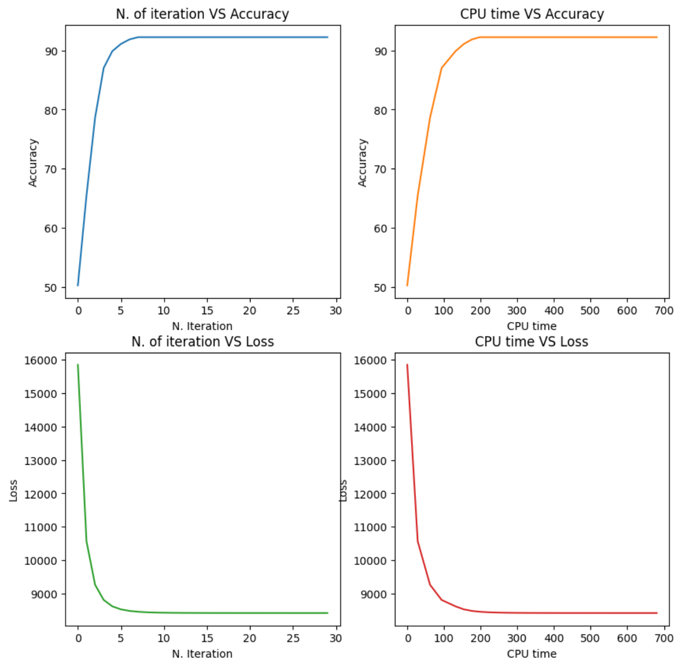


Figure 23: Performance of GD with 1/L

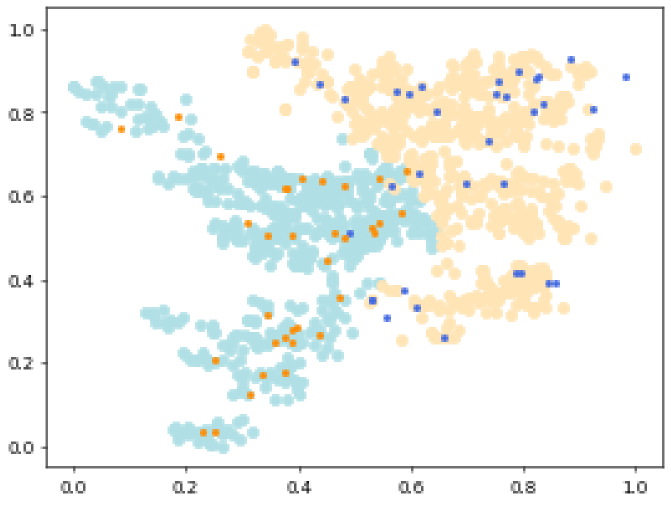


Figure 24: Classes predicted by GD with 1/L

3.1.2 Armijo Rule

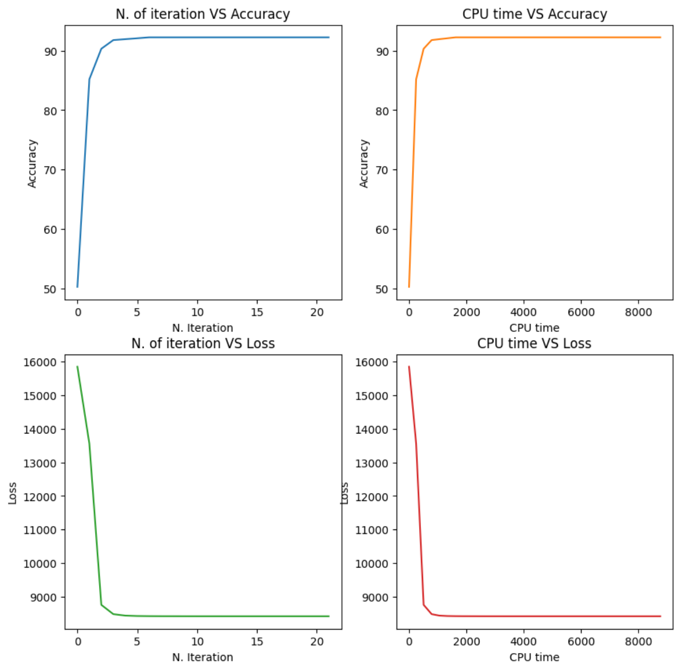


Figure 25: Performance of GD with Armijo rule

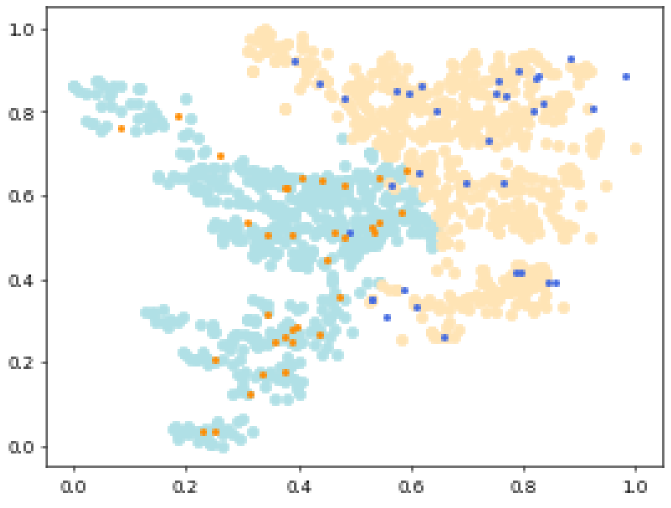


Figure 26: Classes predicted by GD with Armijo rule

3.1.3 Exact Line Search

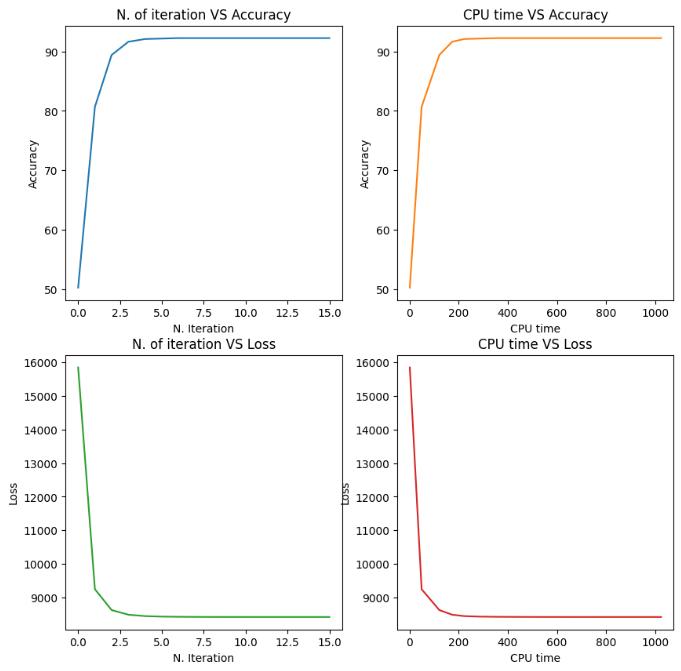


Figure 27: Plots of the performance of GD with exact line search

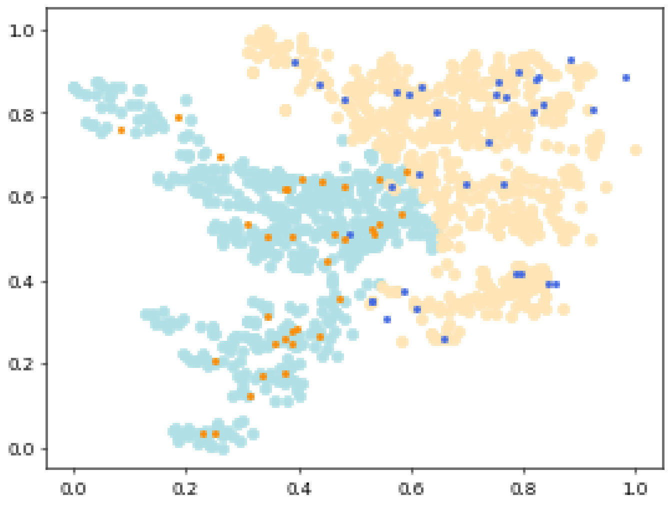


Figure 28: Plot of the classes predicted by GD with exact line search

3.1.4 Conclusions and Comparison

All the tested algorithms on the Banknote dataset achieve an accuracy of approximately 92.2546%. The gradient descent algorithm with fixed step size and exact line search shows similar performance, with a small CPU time (∼90 ms) and 9 iterations. As the figure 31 shows, the Armijo rule method requires slightly more computation time due to the additional computations involved. However, all the algorithms reach their maximum accuracy in less than 12 iterations, with the loss decreasing from around 16,000 to 6,500.

These results are consistent with those obtained on the synthetic dataset.

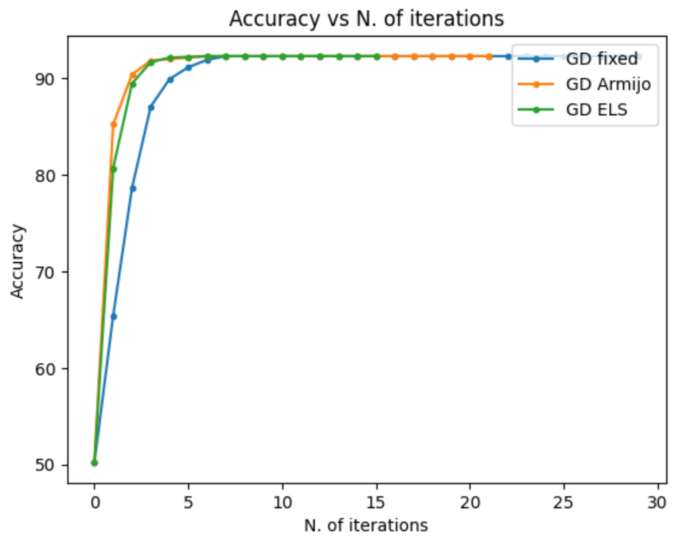


Figure 29: GD: Accuracy VS N. of iterations

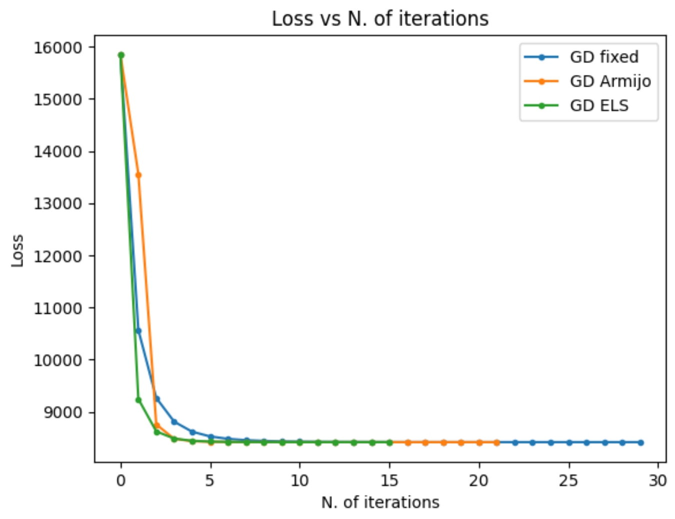


Figure 30: GD: Loss VS N. of Iterations

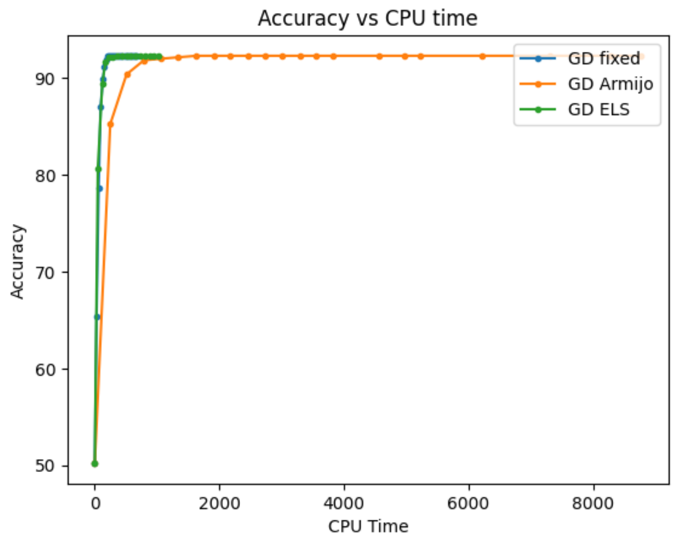


Figure 31: GD: Accuracy VS CPU time

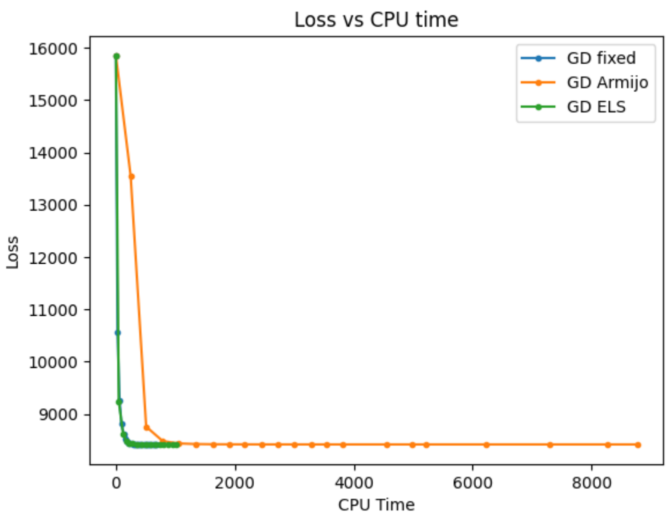


Figure 32: GD: Loss VS CPU time

3.2 Block Coordinate Gradient Descent

3.2.1 Randomized BCGD

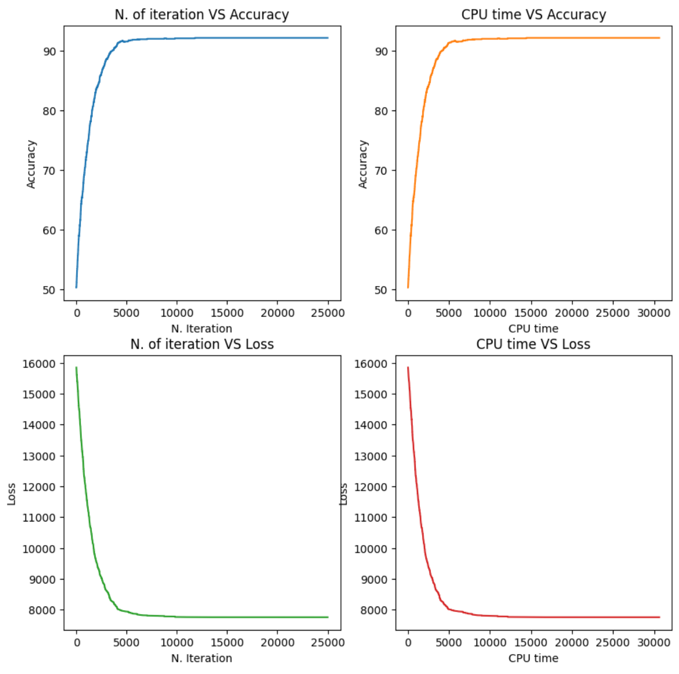


Figure 33: Plots of the performance of Random BCGD with Uniform sampling

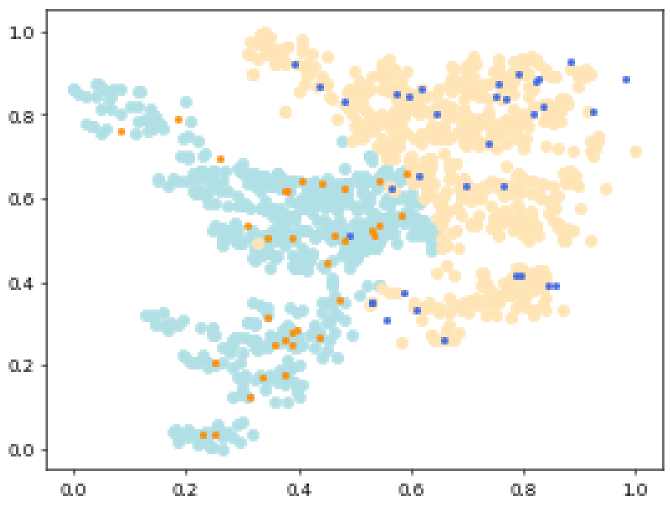


Figure 34: Plot of the classes predicted by Random BCGD with Uniform sampling

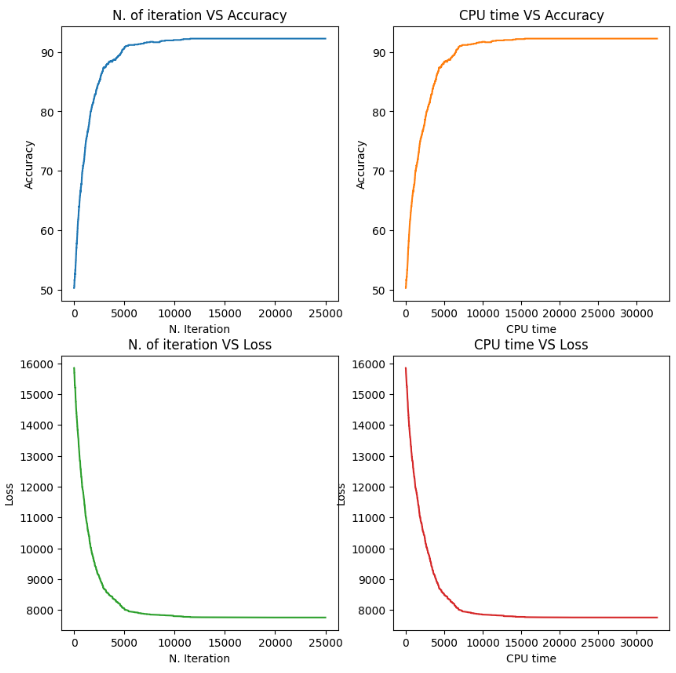


Figure 35: Plots of the performance of Random BCGD with non Uniform sampling (Nesterov)

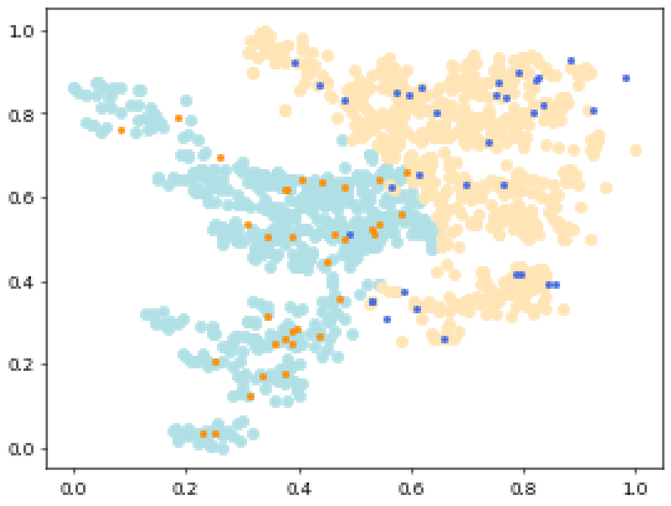


Figure 36: Plot of the classes predicted by Random BCGD with non Uniform sampling (Nesterov)

3.2.2 Gauss Southwell BCGD

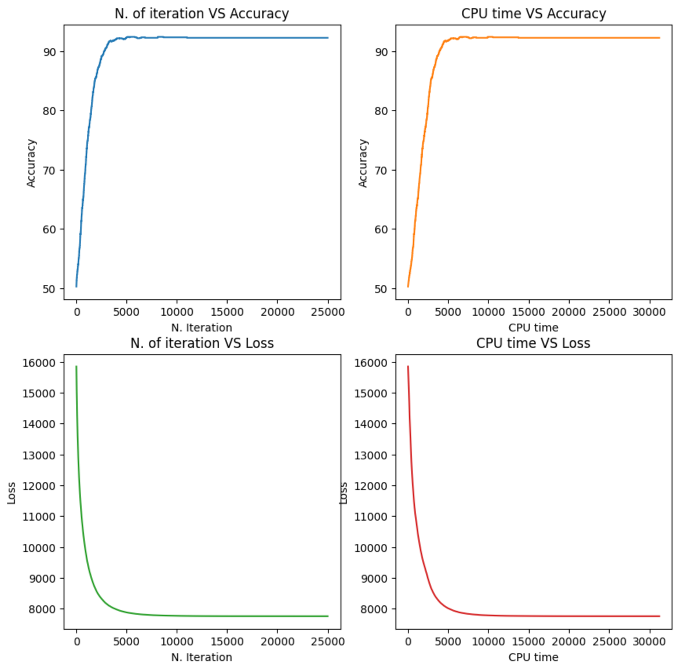


Figure 37: Plots of the performance of BCGD Gauss Southwell with $1/L$

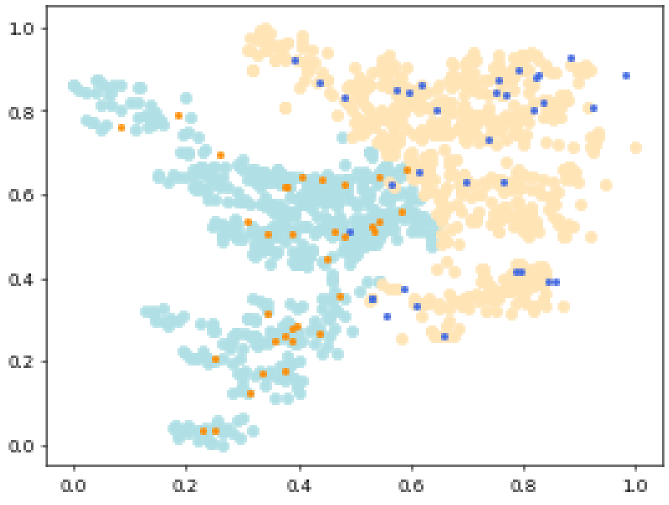


Figure 38: Plot of the classes predicted by BCGD Gauss Southwell with $1/L$

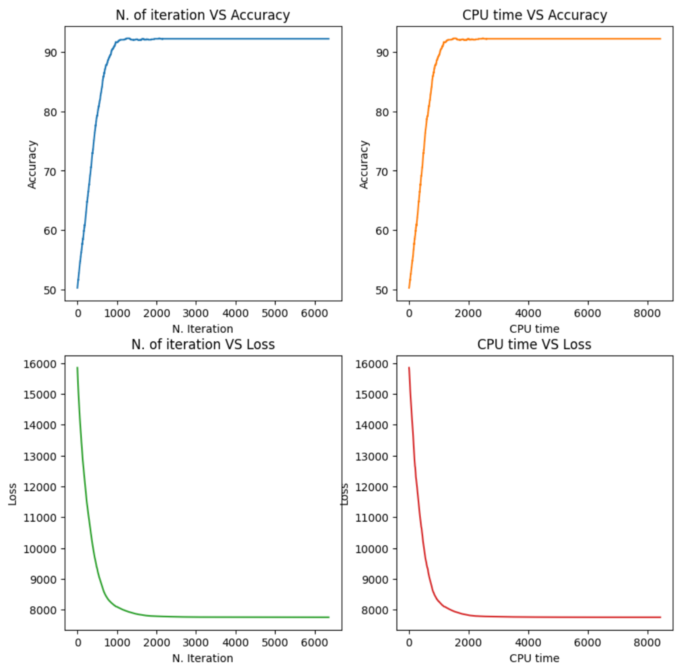


Figure 39: Plots of the performance of BCGD Gauss Southwell with $1/L_i$

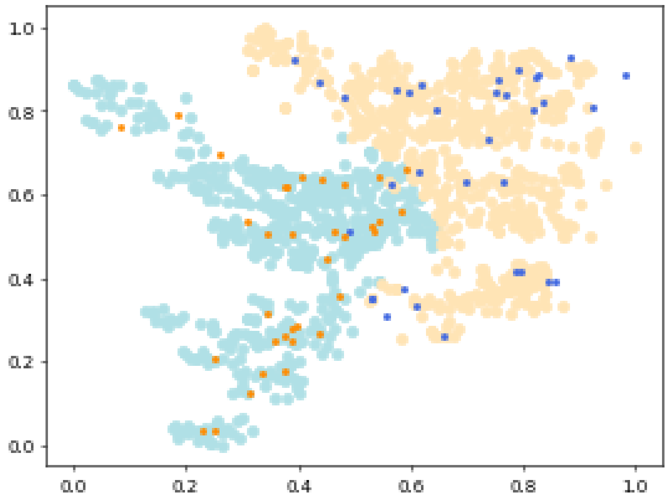


Figure 40: Plot of the classes predicted by Random BCGD Gauss Southwell with $1/L_i$

3.2.3 Conclusions and Comparison

Among the BCGD algorithms, the Gauss-Southwell variant with the learning rate Li performs the best in terms of CPU time and number of iterations.
Analyzing the figure 41, it is evident that it takes less computational time and reaches the maximum accuracy in less than 2,000 iterations. However, the other BCGD algorithms do not exhibit significant differences. They all achieve the minimum loss in approximately 10,000 ms and 10,000 iterations.
It is worth noting that, as suggested by theory, the Randomized BCGD Nesterov variant

achieves slightly higher accuracy than the uniform one but requires more CPU time and iterations.

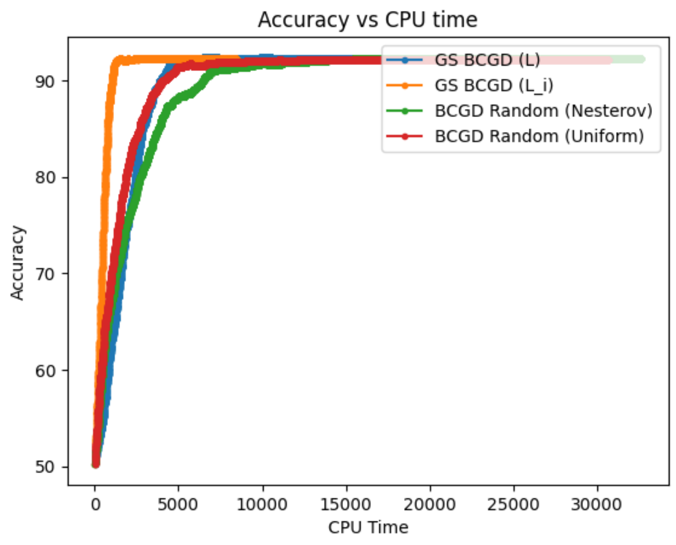


Figure 41: BCGD: Accuracy VS CPU time

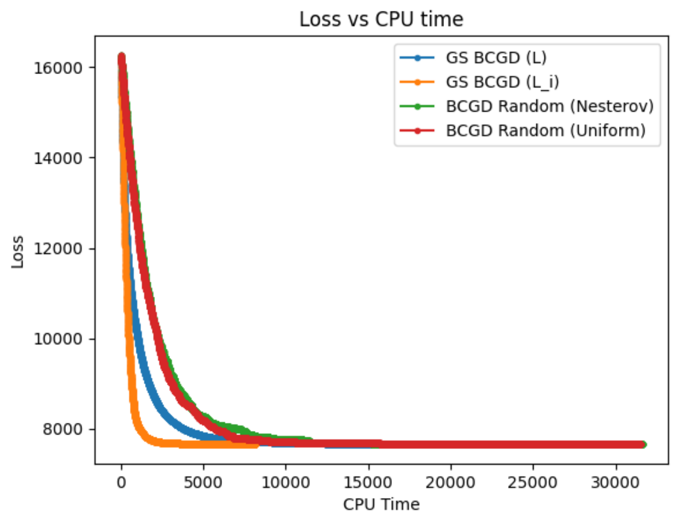


Figure 42: BCGD: Loss VS CPU time

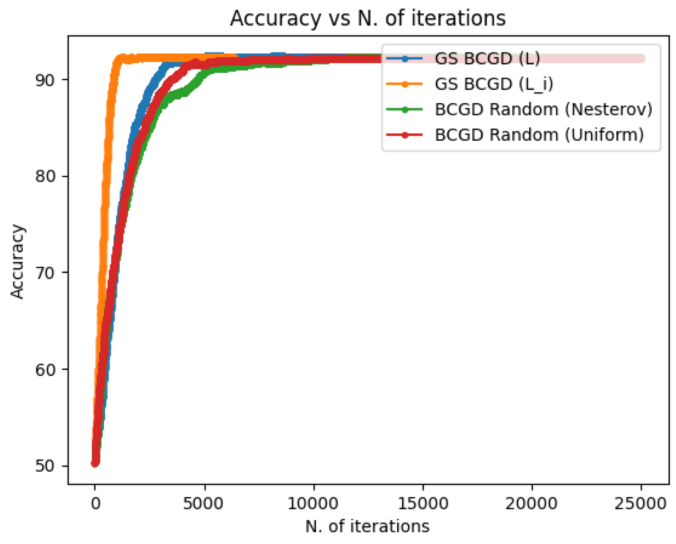


Figure 43: BCGD: Accuracy VS N. of iterations

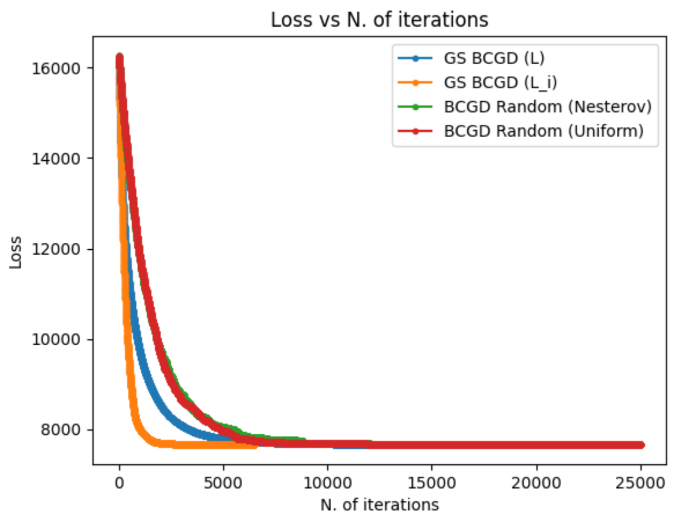


Figure 44: BCGD: Loss VS N. of Iterations

3.3 Results

All the tested algorithms on the Banknote dataset achieve an accuracy of approximately 92.2546%, with the exception of the Randomized BCGD uniform algorithm, which reaches around 92.1779% accuracy.
Considering the small number of data points and the closeness of the clusters, it is reasonable to conclude that all the tested algorithms achieve a good level of accuracy.