Optimization for data science: HW

Vincenzo Caserta, Sharare Zolghadr, Luca Tusini, Gabriella Giachinta

May 1, 2024

1 Introduction to the mandatory part

The homework is a semi-supervised learning problem in which we have l labeled examples and u unlabeled examples for which we must find the labels. In the first part of the homework we use a randomly generated dataset made of 200 labeled data points and 450 unlabeled data points. To create the dataset we start by generating the labeled data x^i, y^i where $i = 1, .., l$. In the following plot we can see the labeled data:

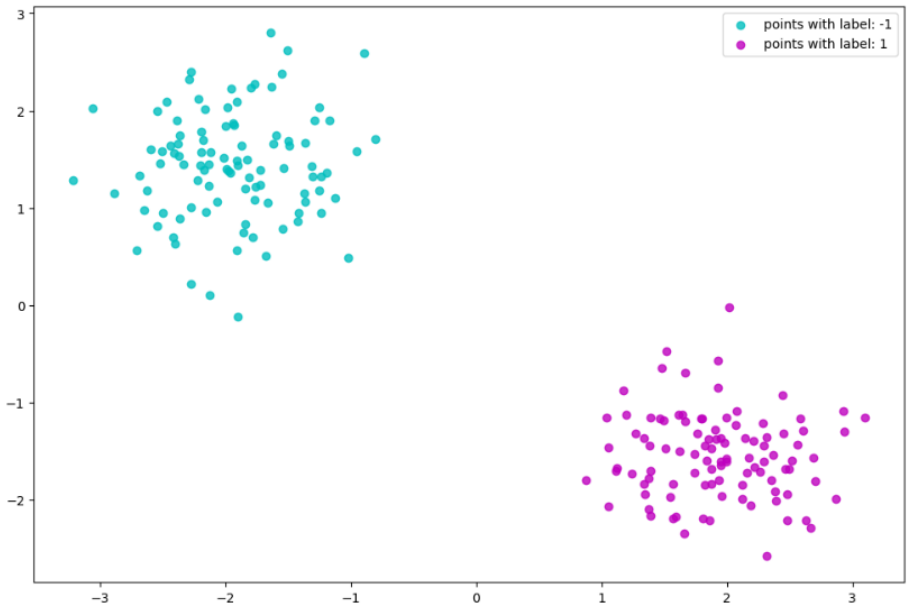

Figure 1: plot of labeled data points

Then we generate new unlabeled data using the uniform distribution, and we plot the unlabeled points together with the labeled ones:

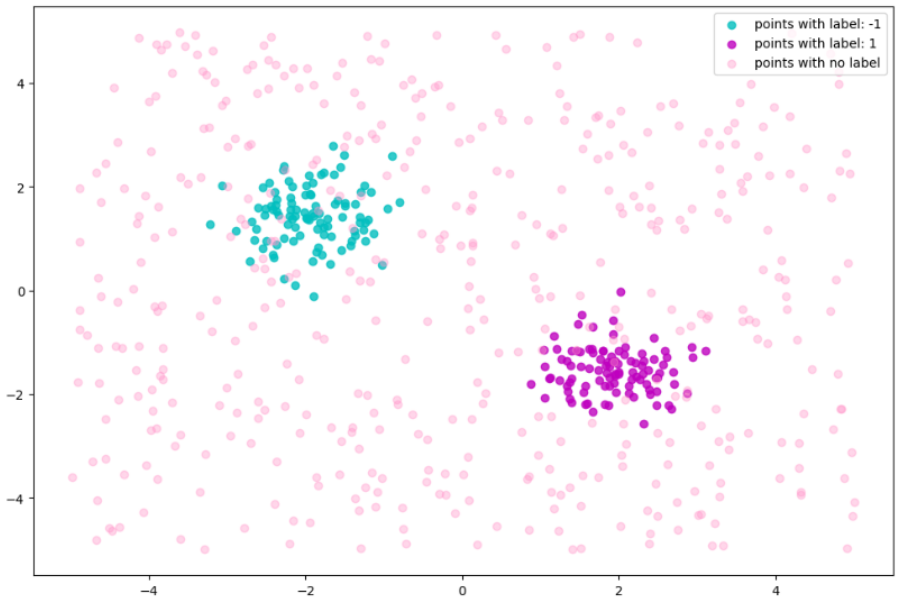

Figure 2: Plot of labeled and unlabeled data points

The goal of the optimization problem is to find the labels -1, 1 for the new unlabeled data points we generated. This is done in a way that maximizes the similarity within the groups and at the same time maximizes the similarity with the data points which already have true labels 1, -1. More precisely, we want to find a labeling that satisfies these two objectives simultaneously. This is done by minimizing the following function:

- **Minimizing the function:**

$$min[y] \sum_{i=1}^{l} \sum_{j=1}^{y} w_{ij}(y^j - yl^i)^2 + \frac{1}{2} \sum_{i=1}^{y} \sum_{j=1}^{y} w_{ij}(y^i - y^j)^2 \qquad (1.1)$$

  Where yl^i are the label of the initial dataset and the y^i are the label that we want find for the new point that we generated in figure 2

- **Assign the right label at new point (pink data points) in figure 2**

We achieve our goals using three different types of Gradient Descent methods:

- **Gradient Descent**
- **BCGD with randomized rule**

- **BCGD with gradient descent rule**

To utilize the gradient descent method, we need to compute the first derivative of the objective function and to compute the Lipschitz constant, we need the second derivative, which is used to calculate the Hessian matrix(we have used Lipschitz constant as our learning rate). However, computing the Hessian matrix can be computationally expensive, especially for large datasets. In our case, since the dataset is not very large, we were able to compute the Hessian matrix, but this may not always be the best option.

$$f'(y) = 2\sum_{i=0}^{l}\sum_{j=0}^{u} w_{ij}(y^j - \bar{y}^i) - \sum_{i=0}^{u}\sum_{j=0}^{u} \bar{w}_{ij}(y^j - y^i) \tag{1.2}$$

How to calculate the stepsize?
In optimization problems, choosing an appropriate step size is a crucial aspect for efficient and effective convergence. There are different methods available for computing the step size, including Armijo rule, line search, and fixed step size. In our implementation, we chose the fixed step size method as our learning rate because it is easy to implement and provided satisfactory results. To set the step size, we assigned the Lipschitz constant to it, which is proved to be a good choice for fixed step size.
The formula for the fixed step size can be written as:
$\alpha = 1/L$,
where L is the Lipschitz constant of the gradient of the objective function. This fixed step size ensures that the algorithm moves toward the minimum point without oscillations or divergence. However, the Lipschitz constant is not always easy to compute and may require additional assumptions about the objective function.

2 Carrying out the task

Once data points with and without labels have been generated, we need a similarity measure. To measure the similarity between the points in Group 1 and the labeled points in Group 2, we used cosine similarity. This was accomplished by creating two matrices: W_unl_unl, which represents the cosine similarity between the unlabeled points, and W_lbl_unl, which represents the cosine similarity between the labeled points and the unlabeled points.
With the code in Figure 3 we calculated matrix similarity between unlabeled points:

```
def Sim_unl_unl(X_unl):
  W_unl_unl = np.zeros((len(X_unl), len(X_unl)))

  for i in range(len(X_unl)):
    for j in range(len(X_unl)):
      W_unl_unl[i][j] = cos_sim(X_unl[i], X_unl[j])
  return W_unl_unl
  #-----------------------------------------------------
  W_unl_unl = Sim_unl_unl(X_unl)
```

Figure 3: calculated matrix similarity between unlabeled points

Then with the code in Figure 4 we calculated the matrix of similarities between labeled and unlabeled points:

```
# Matrix of similarities between labeled and unlabeled points
def Sim_lbl_unl(X_label,X_unl):
  W_lbl_unl = np.zeros((len(X_label), len(X_unl)))

  for i in range(len(X_label)):
    for j in range(len(X_unl)):
      W_lbl_unl[i][j] = cos_sim(X_label[i], X_unl[j])
  return W_lbl_unl
```

Figure 4: Matrix of similarities between labeled and unlabeled points

Before starting the computation, with the function in Figure 5 we assign random labels to the unlabeled points:

```
def generate_Y(X_unl):
# Assigning random labels to unlabeled points
Y = np.zeros(len(X_unl))
for i in range(len(X_unl)):
    Y[i] = random.randint(0, 1)

# Converting labels to [-1,+1]
Y[Y[:]==0] = -1
return Y
```

Figure 5:

And then we plot the data points, as seen in Figure 6.

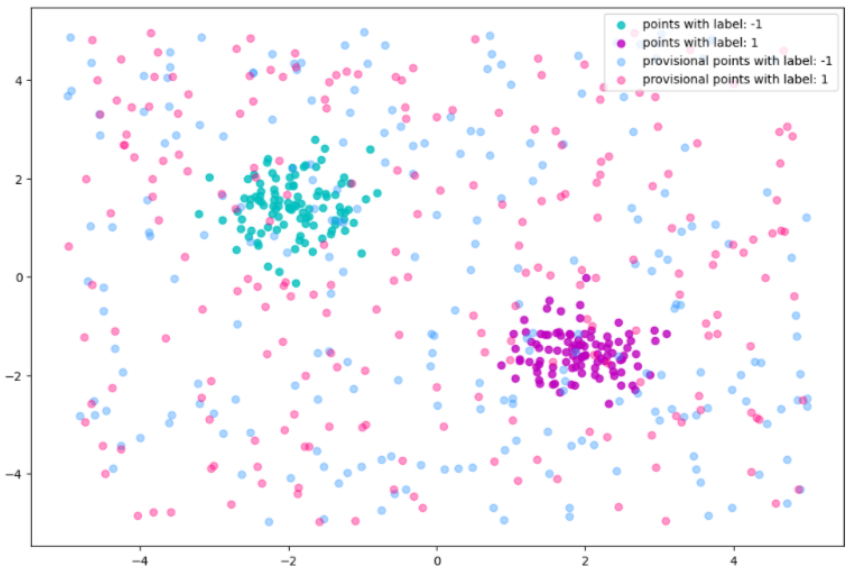

Figure 6: Plot of labeled and unlabeled data points after assigning random labels to unlabeled data

Then we compute the Lipschitz constant in the following way: we first define the Hessian matrix, then we can estimate the Lipschitz constant knowing that the constant should be equal to the maximum eigenvalue of the Hessian matrix, as we can see in Figure 7.

```python
# Computing the hessian matrix
def hessian_matrix(W_unl_unl, W_lbl_unl):
    mat = np.copy(-W_unl_unl)
    for i in range(mat.shape[0]):
      mat[i][i] = np.sum(W_lbl_unl[:,i]) + np.sum(W_unl_unl[:,i]) - W_unl_unl[i][i]
    return mat * 2

# Computing the max eigvalue
def estimate_lipschitz_constant(hessian):
    return scipy.linalg.eigh(hessian, subset_by_index=(len(hessian)-1, len(hessian)-1))[0][0]

# Computing the min eigvalue
def estimate_degree_strongly_convex(hessian):
    return scipy.linalg.eigh(hessian, subset_by_index=(0,0))[0][0]

#-----------------------------------------------------------------------
print("Calculating the Hessian matrix")
hessian = hessian_matrix(W_unl_unl, W_lbl_unl)
print("\nCalculating sigma (strongly convex)")
sigma = estimate_degree_strongly_convex(hessian)
strongly_convex = sigma > 0
print(f"Sigma: {sigma}, {'' if strongly_convex else 'not'} strongly convex")
print("\nEstimating Lipschitz constant for the whole function")
L = estimate_lipschitz_constant(hessian)
print(f"Lipschitz constant: {L}")
print("\nEstimating Lipschitz constant for each single variable")
Li = np.array([norm(hessian[i][i]) for i in range(len(hessian))], dtype='float64')
```

Figure 7:

Before we start to implement the three methods to minimize the function, we also compute the accuracy and the Loss function, by using the code shown in Figure 8:

```
# Defining function to compute accuracy
def loss_fun(W_lbl_unl,W_unl_unl,y_label,y_unl):

  # Creating a MinMaxScaler object
  y = np.copy(y_unl).reshape(-1,1)
  y_lbl = np.copy(y_label).reshape(-1,1)
  scaler = MinMaxScaler()

  # Fitting the scaler to your data
  scaler.fit(y)

  # Transforming your data using the scaler
  y_scaled = scaler.fit_transform(y)

  # Broadcasting arr1 and arr2 to shape (100, 200)
  y_scaled  = y_scaled.reshape(-1,1)

  first_part =np.power(y_scaled - y_lbl.T, 2) * W_lbl_unl.T
  second_part = np.power(y_scaled- y_scaled.T, 2) * W_unl_unl.T
  return (np.sum(first_part) + np.sum(second_part)/2)

def Accuracy(y,label):
  num = abs(label+y)/2
  return np.sum(num)/len(label)*100
```

Figure 8:

3 Gradient descent method

For the gradient descent method, we follow this scheme:

- **minimize a differentiable function f on R^n .**
- **We use a linear (first order) approximation of $f(x_k + d)$ to calculate the search direction at each iteration.**
- **In practice, we approximate $f(x_k + d)$ with the function $_k(d)$ defined as follows:** $n_k(d) := f(x_K) + f(x_k)^T d$ **.**
- **Then choose d_k as the direction such that:** $min(_k(d)), ||d|| = 1$ Equivalent to $min((x_k)^T d)$, $||d|| = 1$.
- **Using the Cauchy-Schwarz inequality, we prove that the optimal direction is** $d_k^* = \frac{(x_k)}{||f(x_k)||}$
- **The classic gradient method calculate each iterate as follows** $x_{k+1} = x_k - \alpha^* \frac{f(x_k)}{||f(x_k)||}$
- **By suitably redefining the stepsize** $\alpha_k = \frac{\alpha^*}{||f(x_k)||}$**, we then have** $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$

Algorithm 1 Gradient method

- **1) Choose a point** $x_1 \in R^n$
- **2) For** $k = 1, ...$
- **3) If** x_k **satisfies some specific condition, then STOP**
- **4)Set** $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$ **, with** $\alpha_k > 0$ **a stepsize**
- **5) End for**

Note that at each iteration we compute the gradient(the gradient is the sum between the variable $part1$ and the variable $part2$) and then we compute the Loss function and the accuracy according to the given formula that tells us about the efficiency of the algorithm. After that, we convert the labels to $-1, 1$ and we plot the gradient behavior.

In Figure 9 we report the gradient behavior for every iteration of the algorithm and how the algorithm works.

We can see in Figure 10 that the classification is correct: the points are placed in the space in such a way that we can define an hyperplane that divides them in two regions.

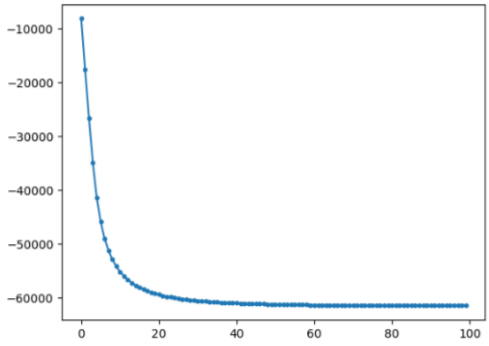

Figure 9: Gradient behavior for every iteration in simple gradient descent method

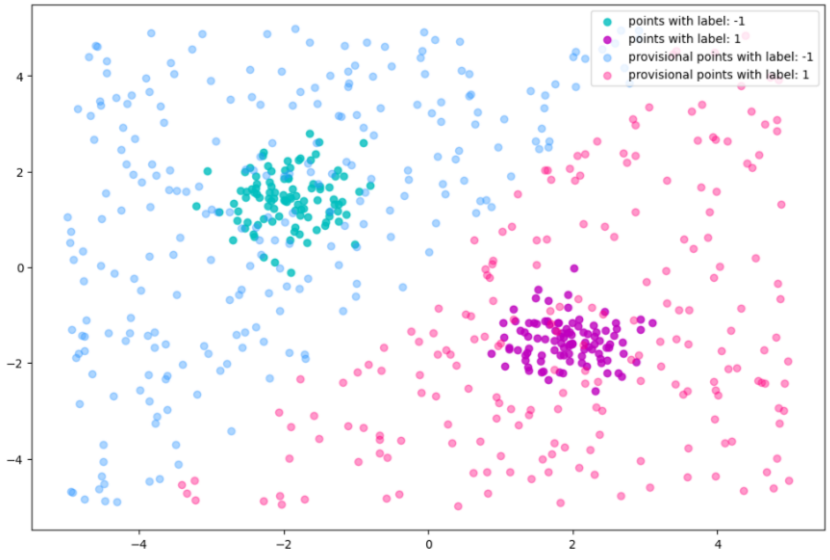

Figure 10: classification made with the learned labels in BCGD algorithm with simple gradient descent method

4 Block Coordinate Gradient Descent with randomized rule

Note that in this algorithm we divided the coordinate into b blocks which is the number of the unlabeled point that we have i.e. the length of the $y_(unl)$ variable. Then we select each block with the probability $P(i_k = i) = \frac{Li}{\sum_{i=1}^{b} Li}$ (non-uniform sampling),and update the corresponding gradient descent and parameters. The approach used here involves utilizing the 'random.choice' command(in python) to generate an array with a length equal to that of the unlabeled data set. This enables us to iterate through a set of blocks in each iteration randomly.

We could consider a uniform distribution to randomly pick the block.(but we have use the improved version) Since i_k is a discrete random variable we have that $P(i_k = i) = \frac{1}{b}$, i = 1, . . . , b. Keep in mind that the variables i_1, . . . , i_k are all independent.

Improving the Rate: There exist different strategies to improve the rate of the randomized BCGD algorithm.

- First idea: use larger stepsizes (replace $_k = \frac{1}{L}$ with $_k = \frac{1}{L_{i_k}}$).
- Second idea: use non-uniform sampling. Nesterov's idea: use $P(i_k = i) = \frac{Li}{\sum_{i=1}^{b} Li}$, i.e. we choose block with larger Lipschitz constant more frequently.
- Stopping condition: if all the components of the gradient array are below the tollerance (1e-10) or if we reached max iteration number (100)
- Step size: $_k = \frac{1}{L_{i_k}}$

Algorithm 2 Randomized BCGD method

1) Choose a point $x_1 R^n$
2) For k = 1, . . . 3) If x_k satisfies some specific condition, then STOP
4) Randomly pick $i_k 1, ..., b$
5) Set $x_{k+1} = x_k -_k U_{i_k i_k} f(x_k)$
6) End for

We implemented the algorithm with the code in Figure 11.
In Figure 12 we report the gradient behavior for every iteration of the algorithm, and in Figure 13 we can see that the classification made with the learned labels is correct.

```python
def BCGD_R(W_lbl_unl,W_unl_unl,X_unl,Y,X_label,y_label,num=100):
  max_iter = num
  tolerance = 1e-10
  step_size = 0.3

  gradient = np.ones(len(X_unl))

  loss_stack_BCGDR = []
  acc_BCGD_R = []
  y_unl = np.copy(Y)
  iter_count = 0

  # The number of blocks is equal to len(y_unl)
  # Randomly select some block i with uniform probability 1/b to update (we only pick one block in this case).


  # Starting the algorithm
  start = timer()

  while (np.any(abs(gradient) > tolerance)) and (iter_count < max_iter):

    # Computing the gradient array
    iter_count +=1
    if iter_count % (10) == 0:
      print(f'--- iteration: {iter_count}')

    indexes =  np.random.choice(X_unl.shape[0],X_unl.shape[0],replace=False,p= np.copy(Li) / np.sum(Li))

    for i in range(X_unl.shape[0]):
      index = indexes[i]

      part1 = np.dot(2* W_lbl_unl[:,index],(y_unl[index]-y_label[:]))
      part1 = part1/norm(2* W_lbl_unl[:,index]* (y_unl[index]-y_label[:]))

      part2 = np.dot(2* W_unl_unl[:,index],(y_unl[index]-y_unl[:]))
      part2 = part2/norm(2* W_unl_unl[:][index]* (y_unl[index]-y_unl[:]))

      gradient[index] = part1 + part2
      y_unl[index] = y_unl[index] - (1/Li[index])*gradient[index]
    current_loss = loss_fun(W_lbl_unl,W_unl_unl,y_label,y_unl)
    loss_stack_BCGDR.append(current_loss)
    acc_BCGD_R.append(1- abs(current_loss))

  end =timer()

  exe_time_BCGD = end - start
  print(f"Time required: {(exe_time_BCGD)}")

  return y_unl,loss_stack_BCGDR,exe_time_BCGD
```

Figure 11: BCGD algorithm with randomized rule

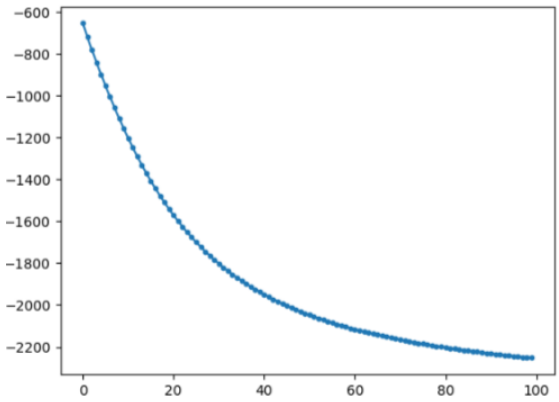

Figure 12: Gradient behavior for every iteration in BCGD algorithm with randomized rule

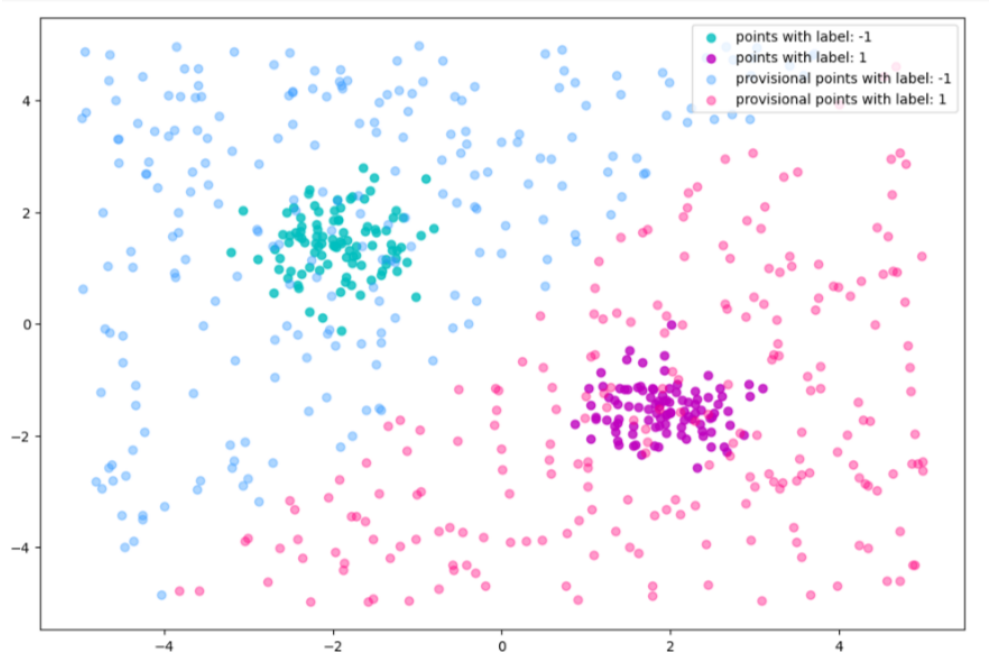

Figure 13: classification made with the learned labels in BCGD algorithm with randomized rule

5 Block Coordinate Gradient Descent with GS rule

Finally we report the third and last algorithm that is the BCGD with GS rule.

Algorithm 1 Gauss-Southwell BCGD method

1) Choose a point $x_1 \bar{R}^n$
2) For k = 1, . . .
3) If x_k satisfies some specific condition, then STOP
4) Pick block i_k such that $i_k = Argmax_{j1,\ldots,b}||_j f(x_k)||$.
5) Set $x_{k+1} = x_k \frac{1}{L} U_{i_k i_k} ||f(x_k)||$
6) End for

- Stopping condition: if all the components of the gradient array are below the tollerance (1e-10) or if we reached max iteration number (100)
- Step size: fixed to $\frac{1}{L}$
- Remark: To run the algorithm, we need to compute the gradient at least once before starting the iterations.

In Figure 14 it is shown the Loss value for every iteration, and we can see that its behavior is as expected. As we can see in Figure 15 the results obtained with the classification can be considered satisfying.

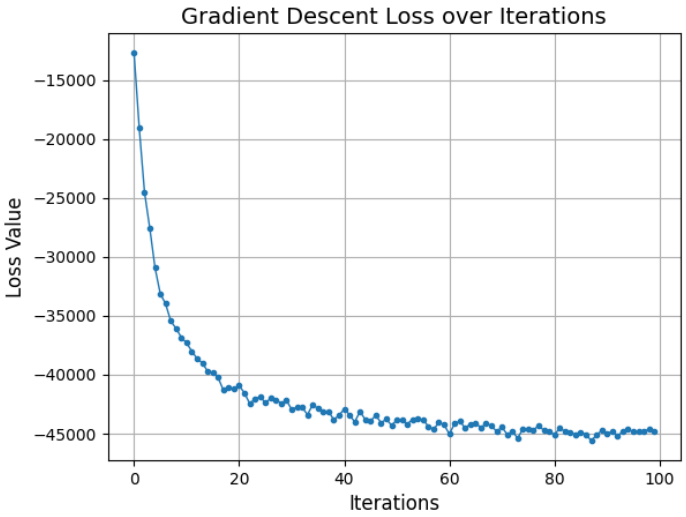

Figure 14: Gradient behavior for every iteration in BCGD algorithm with GS rule

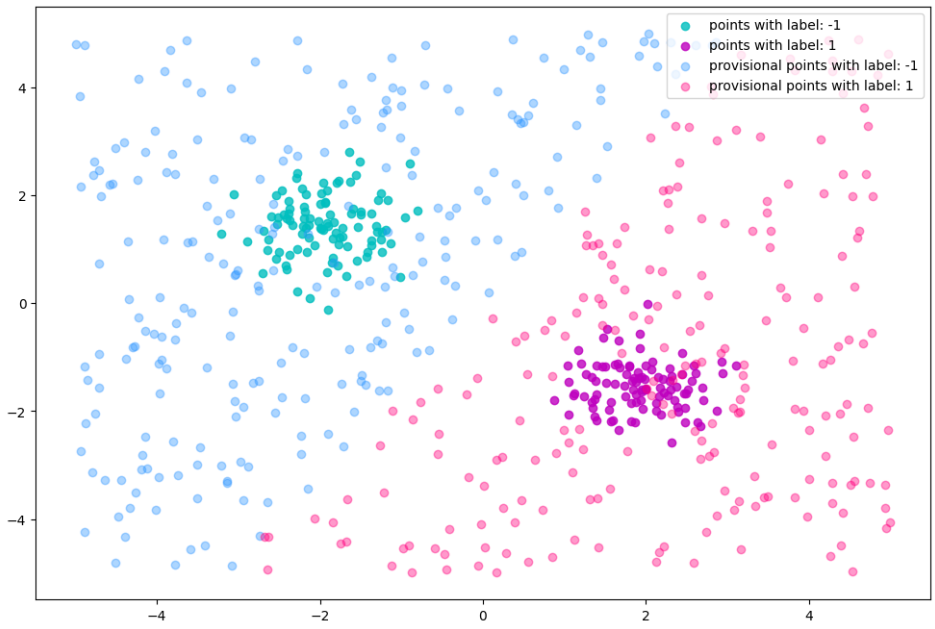

Figure 15: classification made with the learned labels in BCGD algorithm with GS rule

6 Comparison between the three methods

The time complexity, loss value, and convergence rate can vary depending on the problem and implementation details, but here are some general comparisons between randomized block coordinate gradient descent (BCGD), Gauss-Southwell block coordinate gradient descent (GS-BCGD), and standard gradient descent (GD):

Time Complexity: BCGD and GS-BCGD have a lower time complexity per iteration compared to GD. This is because BCGD and GS-BCGD only update a subset of the variables at each iteration, while GD updates all variables. However, BCGD and GS-BCGD require more iterations to converge due to the smaller step size. In general, BCGD and GS-BCGD can be faster than GD for problems with a large number of variables and a sparse structure.

Loss Value: The loss value convergence rate can depend on the problem and implementation details. In general, GS-BCGD can converge faster than BCGD and GD. This is because GS-BCGD uses an approximate Hessian matrix to compute the step size, which can improve convergence. However, GS-BCGD requires more computational resources to compute the Hessian approximation.

Convergence: BCGD and GS-BCGD can converge to a suboptimal solution due to the smaller step size. However, in practice, BCGD and GS-BCGD can converge to a good solution faster than GD due to the ability to exploit the problem's structure. GD can converge to the optimal solution but requires a larger number of iterations and computational resources.

While the theoretical analysis suggests that Block Coordinate Gradient Descent (BCGD) algorithms may have certain advantages over the standard Gradient Descent (GD) algorithm, in practice, our experiments show that GD outperforms BCGD. As evidenced by the plots of execution time and loss function, GD converges faster, executes in less time, and achieves higher accuracy. Therefore, in practice, GD may be a more appropriate choice for optimizing the given objective function.

In summary, the choice between BCGD, GS-BCGD, and GD depends on the problem's structure, the availability of computational resources, and the desired trade-off between time complexity, loss value convergence rate, and convergence quality.

We compared the three methods by plotting the gradient descent loss over iterations. As it's shown in Figure 16 the simple gradient descent method reaches smaller values of loss than the Gauss-Southwell BCGD and the Randomized BCGD.
As we can see in Figure 17 the simple BCGD is also the fastest in terms of execution time.

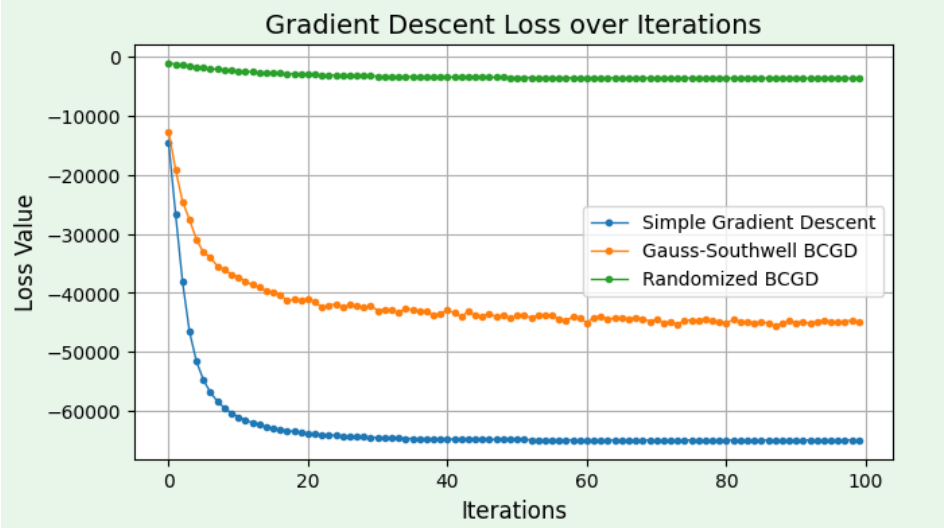

Figure 16:

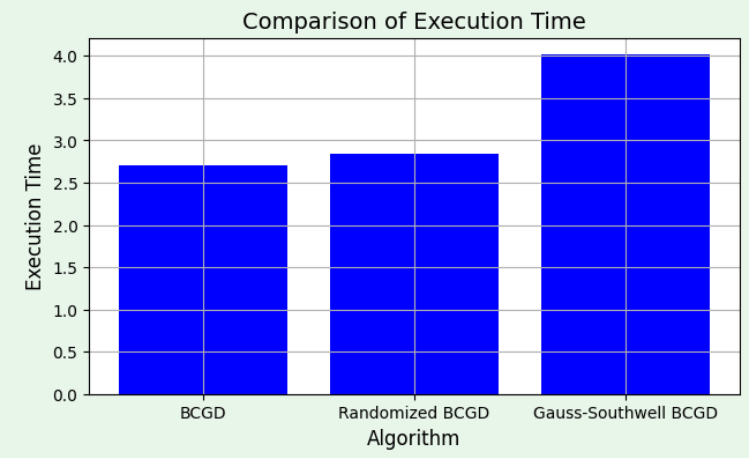

Figure 17:

7 Testing of the algorithms with a real-life Dataset

For the second part of the homework we tested the three previous methods on a real-life dataset that contains data on breast cancer. The variable to predict in this case is the cancer diagnosis, that we represented again with 1, -1.
In order to achieve accurate and reliable results in real datasets, it is important to standardize the data. This ensures that all features have an equal impact on the analysis and prevents the model from being skewed towards a particular feature. Additionally, dimentionality reduction techniques like Principal Component Analysis (PCA) can be applied to reduce the number of features and obtain a lower-dimensional representation of the data. In this case, PCA was used to reduce the data to 2 dimensions for better visualization and analysis.

We divided the dataset in two parts: one that contains the labeled data, with 171 labeled data points, and the other one that contains the unlabeled data, with 398 data points.

We will use this two parts in the same way as before to see how our semi-supervised learning algorithms work, then we will calculate the accuracy by comparing the predicted labels with the true labels.

In Figure 18, Figure 19 and Figure 20 we can see the classification made with the learned labels by using the three methods:

- simple gradient descent (Figure 18)
- BCGD algorithm with randomized rule (Figure 19)
- BCGD algorithm with GS rule (Figure 20)

Figure 21 is a comparison of the Loss function behavior over every iteration for the three methods used.

In Figure 22 we can see the execution time for each method we used, and we notice that the simple gradient descent was the fastest method.

In terms of accuracy, the simple BCGD and the Randomized BCGD reach a higher level than the Gauss-Southwell BCGD, as we can see in Figure 23.

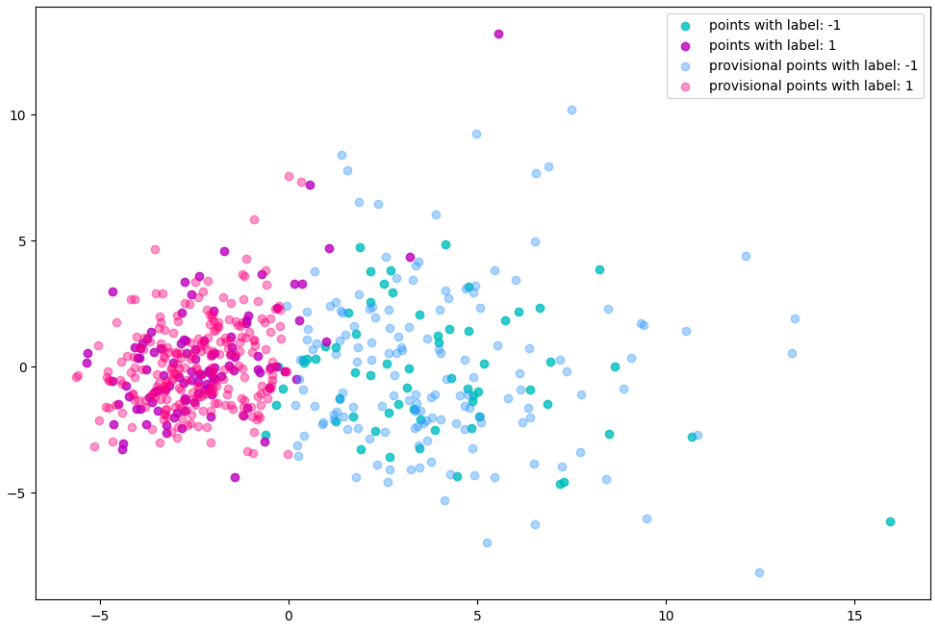

Figure 18: Dataset after applying classic gradient descent

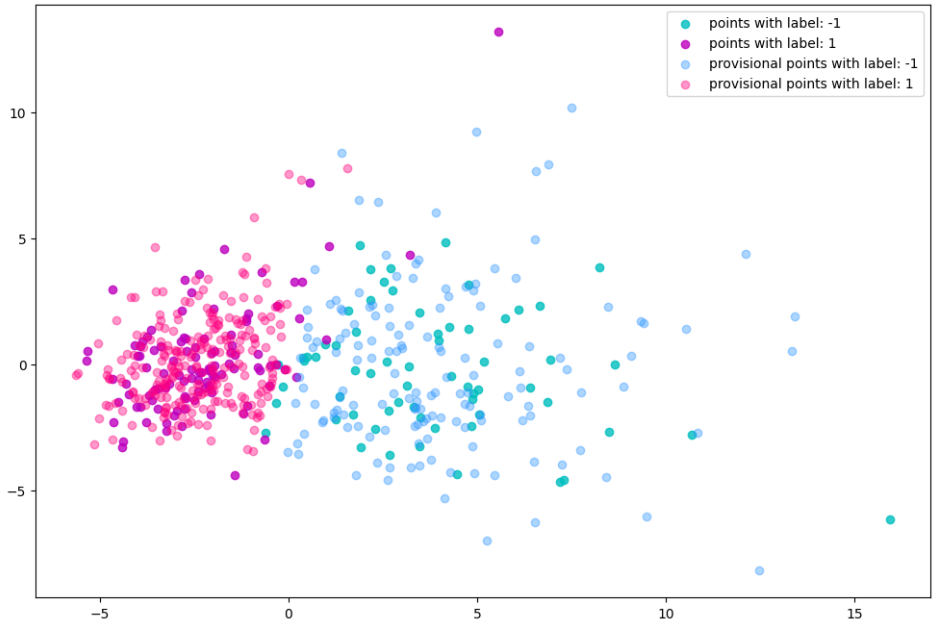

Figure 19: Dataset after applying randomized BCGD

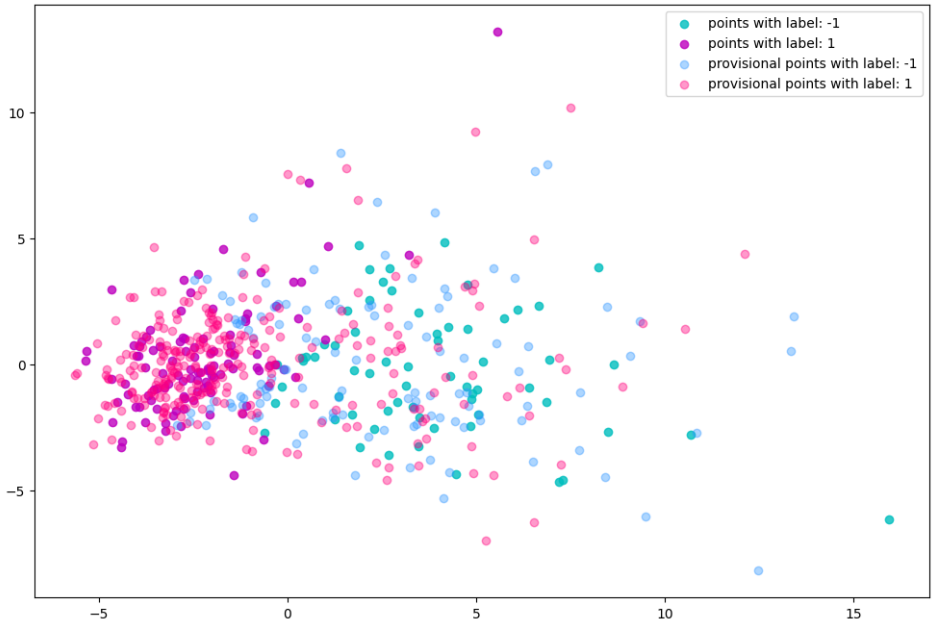

Figure 20: Dataset after applying GS BCGD

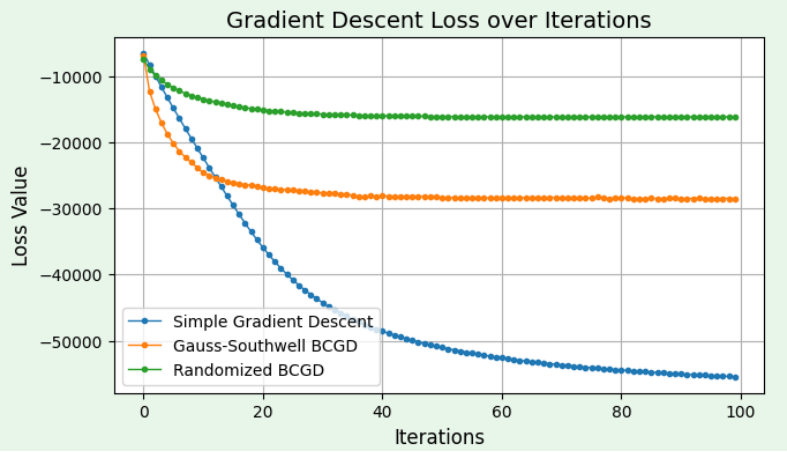

Figure 21:

Figure 22:

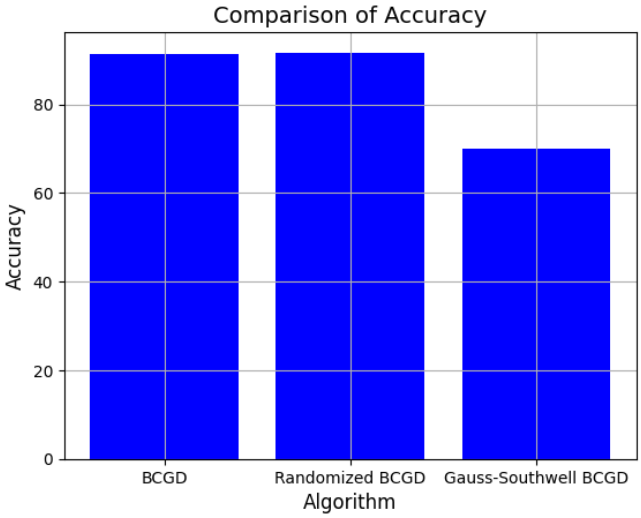

Figure 23: