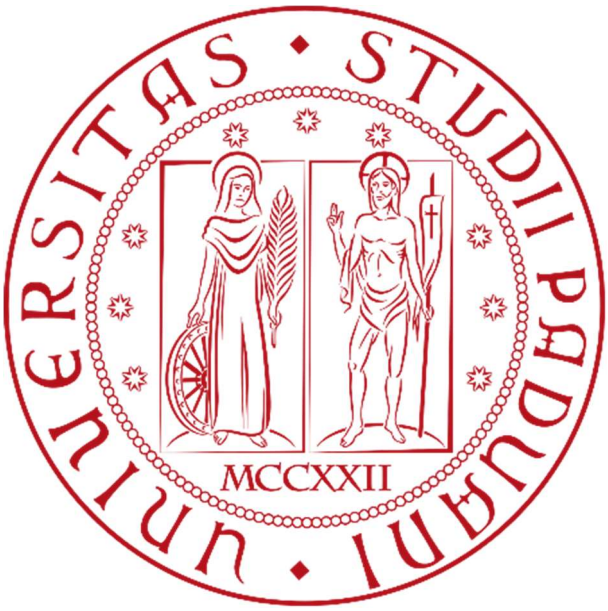
# Optimization for Data Science

Homework 1

Comparison Between GD/BCGD Algorithms in Semi-Supervised Learning Problems




Francesco Vo, 2079413

Marco Uderzo, 2096998

Claudio Palmeri, 2062671

# 1. Introduction

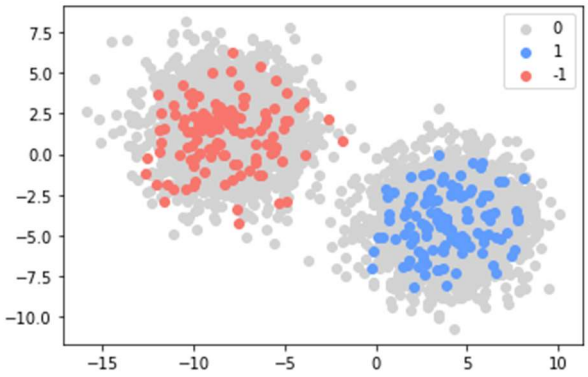
This is a supervised learning problem and thus we are given a dataset where only a small fraction of the data has been labelled. Our goal is to find a way to label the unlabelled data points. The algorithms that we are going to use are:

- Gradient Descent
- Block Gradient Descent with Gauss-Southwell Rule
- Block Gradient Descent with Randomized Rule

# 2. Dataset

## 2.1 Point generation

For this task we generated 2 random clusters of equal size. For each cluster we chose randomly selected 0.25% of the points and we assigned them as the known target values.




## 2.2 Loss function

The loss function was already given, and it defined as:

$$L(y) = \sum_{i=0}^{l}\sum_{j=0}^{u} w_{ij}\left(y^j - \bar{y}^i\right)^2 + \frac{1}{2}\sum_{i=0}^{u}\sum_{j=0}^{u} \bar{w}_{ij}\left(y^i - y^j\right)^2$$

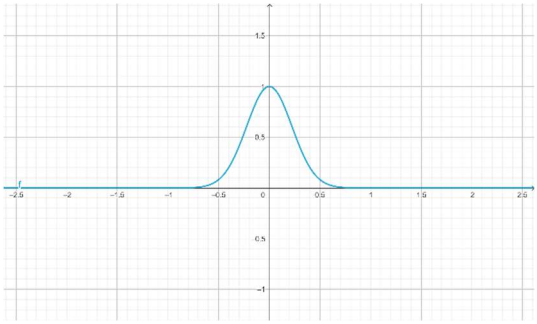
The loss function is divided in 2 terms. In the first term we calculate the loss given by a labelled point and an unlabeled one. The loss is given by the squared difference between the target values, multiplied by the relative similarity measure for each pair.
The second term is similar to the first one, but it is calculated between pairs of unlabeled points and also the total loss is divided by 2 because we it to have a lesser effect on the loss function.

### 2.3 Similarity function

We want the similarity function to be great if the points are near each other. On the contrary, if two points are distant, we want it to be lower.
The principle should be this: the weight added to the loss function should be higher if we label differently two points that are near and lower if the points labelled differently are distant from each other.

We choose the Gaussian norm and applied it to the difference of our points.

$$W(\vec{a},\vec{b}) = e^{-C\left\|\vec{a}-\vec{b}\right\|_2^2}$$



Gaussian weights for C = 10

C is equal to $\frac{1}{2*Var(Dataset)}$, as a way to normalize the weights.

## 3. Algorithms

### 3.1 Gradient

The gradient of the loss function with respect to a generic $y^j$ is equal to:

$$\nabla_{y^j} f(y) = 2\sum_{i=0}^{l} w_{ij}(y^j - \overline{y^i}) + \sum_{i=1}^{u} \overline{w}_{ij}(y^j - y^i)$$

### 3.2 Step-size

We used a fixed step-size for the calculations. To calculate it we calculated the Lipschitz constant L, which is lower bounded by the highest eigenvalue of the weight matrix.
Since the Hessian matrix is constant and positive definite, the value L is independent from the data and therefore constant. Thus we can calculate it only once (otherwise it would be very costly computationally).

### 3.3 Relative Loss

We estimated the relative loss by calculating the relative error between the loss and the loss at the previous step.

$$relative\_loss = \frac{(L^{k+1} - L^k)}{L^k}$$

# 4. Gradient Descent

## 4.1 Update rule

The first algorithm we implemented is the basic version of gradient descent, with the update rule equal to:

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

In this case $\alpha = \frac{1}{L}$ because we used a fixed step-size.

## 4.2 Implementation

In the notebook we started by assigning labels to unlabeled points randomly and we created an initial vector, our starting point for the analysis.

```
ystart = np.copy(y)

for i in range(n):
    if um[i] == True:
        idx = prng.randint(low=0, high=2)
        ystart[i] = labels[idx]

return ystart
```



For keeping track of the labelled and unlabeled elements we created 2 masking vectors that contain that information.

```
def generate_mask(lidx):
    # lidx: list of index of labelled examples
    umask = np.ones(y.shape, dtype=bool)
    umask[lidx] = False

    lmask = np.zeros(y.shape, dtype=bool)
    lmask[lidx] = True
    return (lmask, umask)
```

Also, we implemented the functions for calculating the loss and the gradient. For the loss function we start with a weight matrix W that memorizes all the weights for each pair of data, and the output y. Instead of doing a double loop like:

```
for i in range(n):
        for j in range(n):
            L[i,j] = W[i,j] * (y[i]-y[j]) ** 2
```

We used a trick to do the calculation faster; we calculate the outer product between y and a vector of ones of the same length of y to get a matrix where one the same row we have we repeat the $i^{th}$ element. With that in mind we calculate the difference between the matrix and the transpose, and we use the result to get the loss.

```
def calculate_loss(W, y, lidx):
    n = W.shape[0]
    loss = 0

    [lm, um] = generate_mask(lidx)

    L = np.zeros((n, n))

    v = np.outer(y, np.ones(n))
    dd = (v - v.T) ** 2

    L = np.multiply(W, dd)

    loss += np.sum(L[:,lm]) + 0.5 * np.sum(L[:,um])
    return loss
```

The gradient function is implemented in a similar manner (see the notebook).

With those functions we implemented the gradient descent:

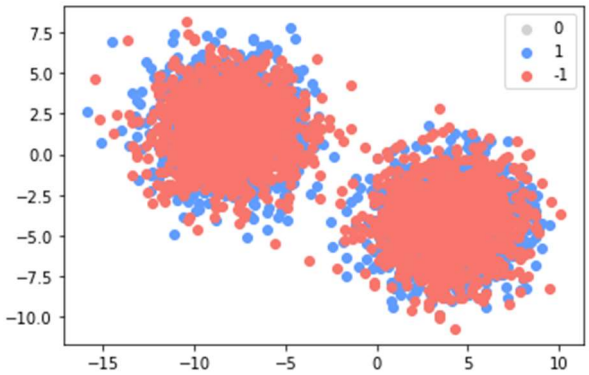
```
[43] # Lastly we calculate the appropriate stepsize:
     alpha = calculate_stepsize(W)
     print(f"alpha = {alpha}\n")
     #We apply here the Gradient Descent algorithm:

     loss_history = np.zeros(MAX_ITERS)
     time = np.zeros(MAX_ITERS)
     s_iter = 0

     pp = np.copy(ystart)
     pp = pp.astype("float64")

     loss = calculate_loss(W, pp, lidx)
     prev_loss = 0

     for iter in range(MAX_ITERS):
         s_iter = iter
         grad = calculate_gradient(W, pp, lidx)
         pp -= alpha*grad

         prev_loss = loss
         loss = calculate_loss(W, pp, lidx)
         loss_history[iter] = loss

         rel_loss = np.abs(loss - prev_loss)/prev_loss
         if rel_loss < EPS:
             break;

         print(f"step\t{iter+1}:\t{loss}")
```

The stopping condition occurs when the relative loss is less than 1e-6.

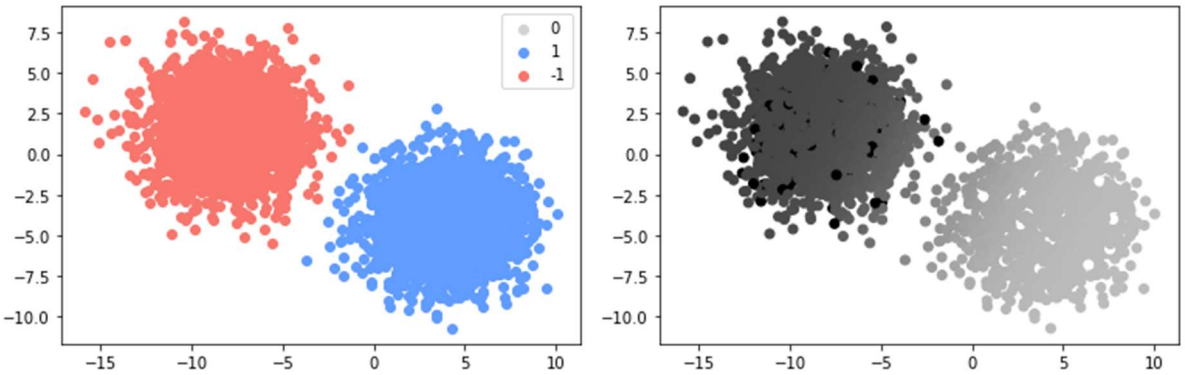
At the end we obtain a probability vector with points in the range of [-1, 1]. If the value is greater of equal 0 we assign it to 1, otherwise it will be labelled -1.

### 4.3 Results

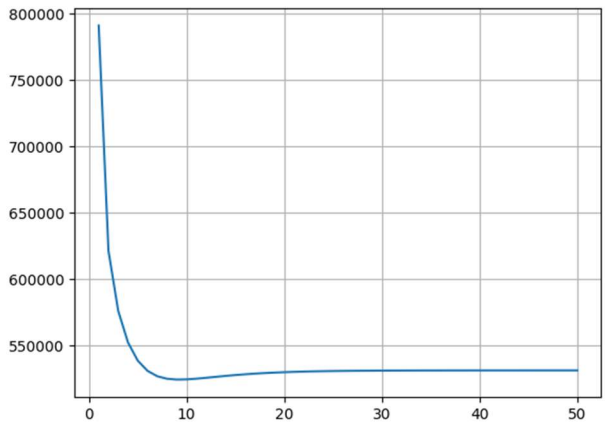
We start with a dataset with 5000 data points and 5% of labelled examples:




After the 50$^{th}$ iterations we get:




Let's plot the loss:



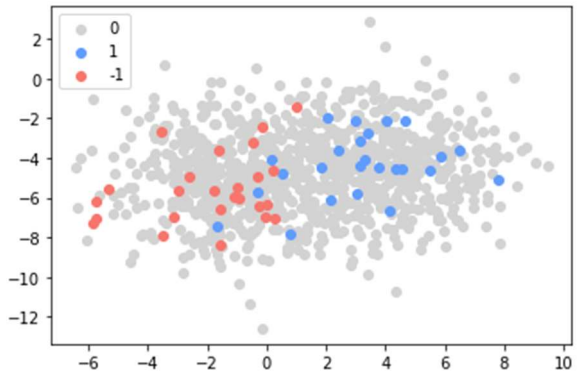
### 4.4 Notes

1. The algorithm is slow. (53 seconds)

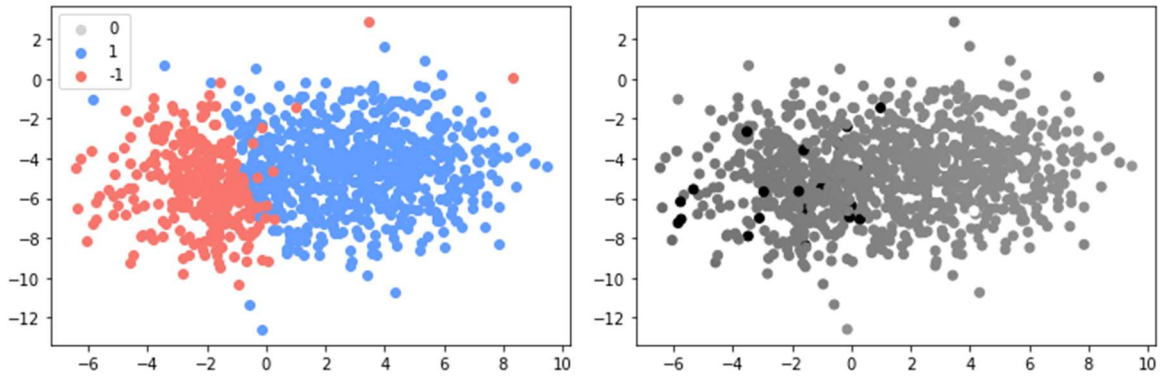
2. If you have a dataset where you have a very small number of unlabeled data the algorithm converges to a point where all the unlabeled points are set to 0, this happens because if you have a pair with each example labelled with 0 the loss relative to that pair will also be also 0; the negative effect on the loss of the labelled examples won't be great enough to get the desired outcome.

3. The algorithms works well on datasets that are divided in clusters and well separated from each other. On the contrary if you have a dataset where you don't have a clear separation the result will be similar to point 2) where all the unlabeled examples tend to 0. This problem can be solved by thresholding the result, with good results.

We look at a dataset of 1000 points:




Here we have the result:



We see that in the grey-map that the values tend towards 0 (they have the same color) but if we apply a threshold the result can be good but not perfect as we see in the red-blue map there are outliers that are very distant from their centers.

# 5. Gauss-Southwell Block Coordinate Gradient Descent

## 5.1 Update rule

The second algorithm we implemented is the Block Coordinate Gradient Descent with Gauss-Southwell rule.

Formally, we choose the block $i_k$ such that $i_k = Argmax \left|\left|\nabla_j f(x_k)\right|\right|$, with j∈{1,...,b} . This means we choose the block based on the maximum absolute value of the gradient at each iteration. Note that, as required by the specifications, the block dimension is equal to 1.

The update rule is $x_{k+1} = x_k - \alpha U_{i_k} \nabla_{i_k} f(x_k)$ , with $\alpha$ being set to 1/L.

In the code, $U_{i_k}$ is not implemented, as it is only necessary when dealing with the theory of the algorithm itself. Indeed, in the code it is sufficient to access the correct block.

## 5.2 Implementation

Before talking about the algorithm implementation, we need to introduce two functions.

```
def update_gradient(grad, x, W, lidx, idx, x_old):
    n = W.shape[0]
    (lm, _) = generate_mask(lidx)

    xi = x[idx]
    dd = xi * np.ones(n) - x
    L = np.multiply(dd, W[idx, :])
    L[lm] *= 2

    grad[idx] = np.sum(L)

    # update all the other gradients
    dn = x - xi * np.ones(n)
    do = x - x_old * np.ones(n)
    Lt = np.multiply(dn, W[:,idx]) - np.multiply(do, W[:,idx])
    Lt[lm] = 0
    Lt[idx] = 0
    grad += Lt
    return grad
```

Instead of recalculating the gradient from the beginning, it is more efficient to use the aforementioned function, that only updates the gradient with respect to the coordinate we are considering.
The update rule chosen subtracts the single loss with respect to the older value and adds the single loss with respect to the new value. The values of the gradient for all other values have to be updated as well, as x[idx] changes.

```
def single_loss(W, x, lidx, idx):
    n = W.shape[0]
    (lm, um) = generate_mask(lidx)

    xi = x[idx]
    dd = (xi * np.ones(n) - x) ** 2
    L1 = np.multiply(dd, W[idx, :])
    L2 = np.multiply(dd, W[:, idx])

    single_loss = np.sum(L1[lm]) + 0.5 * np.sum(L1[um])
    single_loss += 0.5 * np.sum(L2)
    return single_loss
```

This function is responsible for computing the loss for a single coordinate of x.

Therefore, these two functions speed up the algorithm.

We start by assigning labels to unlabeled points randomly and we create an initial vector, our starting point for the analysis, and we also calculate the gradient and loss.

```
ystart = yinit(y, lidx, 42)
x = np.copy(ystart)
x = x.astype("float64")

grad = calculate_gradient(W, x, lidx)
loss = calculate_loss(W, x, lidx)
loss_history = np.zeros(BCGD_MAX_ITERS)

s_iter = 0
```

The Gauss-Southwell Block Coordinate Gradient Descent algorithm is implemented as follows:

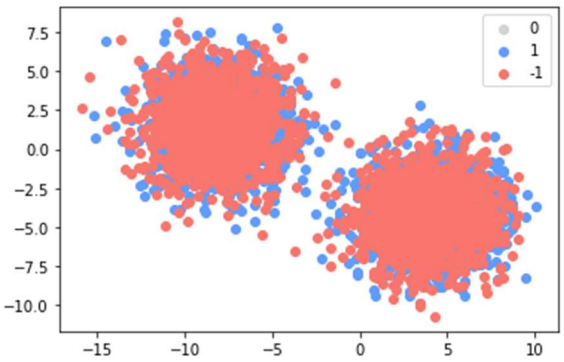
```
for iter in range(BCGD_MAX_ITERS):
    s_iter = iter
    loss_old = 0
    loss_new = 0
    x_old = 0

    idx = np.argmax(np.abs(grad))
    loss_old = single_loss(W, x, lidx, idx)
    x_old = x[idx]

    x[idx] -= alpha * grad[idx]
    loss_new = single_loss(W, x, lidx, idx)

    grad = update_gradient(grad, x, W, lidx, idx, x_old)
    prev_loss = loss
    loss = loss - loss_old + loss_new
    loss_history[iter] = loss

    rel_loss = np.abs(loss - prev_loss)/prev_loss * DATA_DIM

    if rel_loss < EPS:
        break;

    print(f"step\t{iter+1}:\t{loss}")
```

The stopping condition occurs when the relative loss is less than 1e-6, which is a form of early stopping.

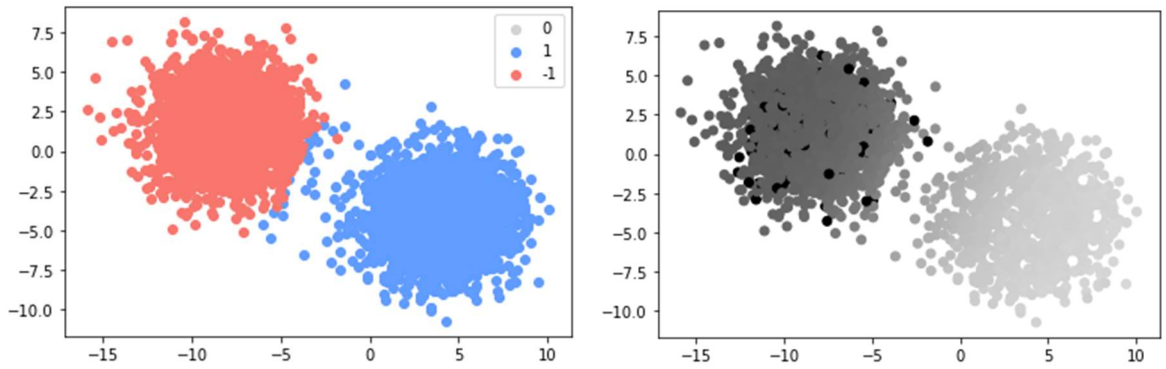
At the end we obtain a probability vector with points in the range of [-1, 1]. If the value is greater of equal 0 we assign it to 1, otherwise it will be labelled -1.

## 5.3 Results

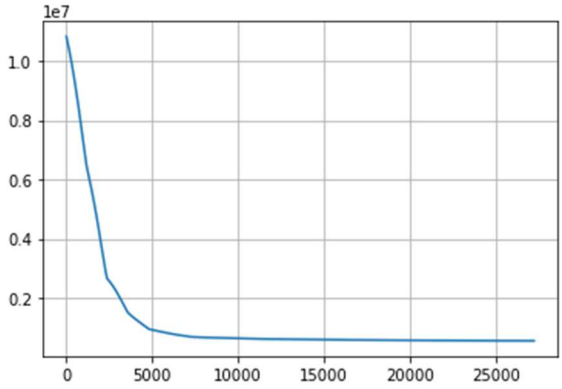
We start with a dataset with 5000 data points and 5% of labelled examples:



Early stopping terminates the algorithm at iteration 27203, resulting in the following scenario.




Let's look at the loss




### 5.4 Notes

1. As expected, Gauss-Southwell BCGD is much faster than Gradient Descent (32 seconds)

## 6. Randomized Block Coordinate Gradient Descent

### 6.1 Update rule

The third algorithm we implemented is the Randomized Block Coordinate Gradient Descent.

We choose the block $i_k$ randomly.

The update rule is $x_{k+1} = x_k - \alpha_k U_{i_k} \nabla_{i_k} f(x_k)$ , with $\alpha$ being set to 1/L.

In the code, $U_{i_k}$ is not implemented, as it is only necessary when dealing with the theory of the algorithm itself. Indeed, in the code it is sufficient to access the correct block.

## 6.2 Implementation

The same functions defined before are used.

We start by assigning labels to unlabeled points randomly and we create an initial vector, our starting point for the analysis, and we also calculate the gradient and loss.

```
ystart = yinit(y, lidx, 42)
x = np.copy(ystart)
x = x.astype("float64")

grad = calculate_gradient(W, x, lidx)
loss = calculate_loss(W, x, lidx)
loss_history = np.zeros(BCGD_MAX_ITERS)

s_iter = 0
```

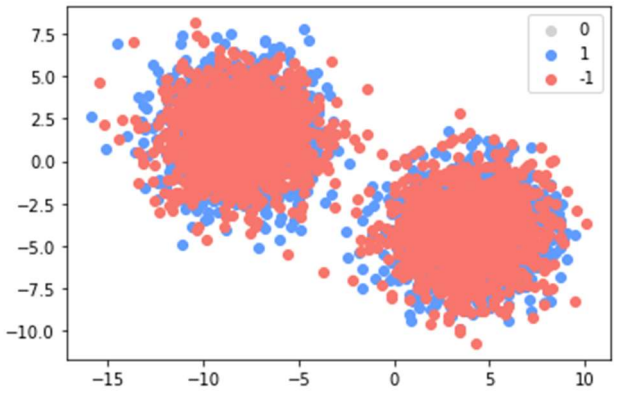
The Randomized Block Coordinate Gradient Descent algorithm is implemented as follows:

```
for iter in range(BCGD_MAX_ITERS):
    s_iter = iter
    loss_old = 0
    loss_new = 0
    x_old = 0

    idx = np.random.choice(uidx[0], 1)
    idx = idx[0]
    loss_old = single_loss(W, x, lidx, idx)
    x_old = x[idx]

    x[idx] -= alpha * grad[idx]
    loss_new = single_loss(W, x, lidx, idx)

    grad = update_gradient(grad, x, W, lidx, idx, x_old)
    prev_loss = loss
    loss = loss - loss_old + loss_new
    loss_history[iter] = loss

    rel_loss = np.abs(loss - prev_loss)/prev_loss * DATA_DIM

    if rel_loss < EPS*0.01:
        break;

    print(f"step\t{iter+1}:\t{loss}")
```

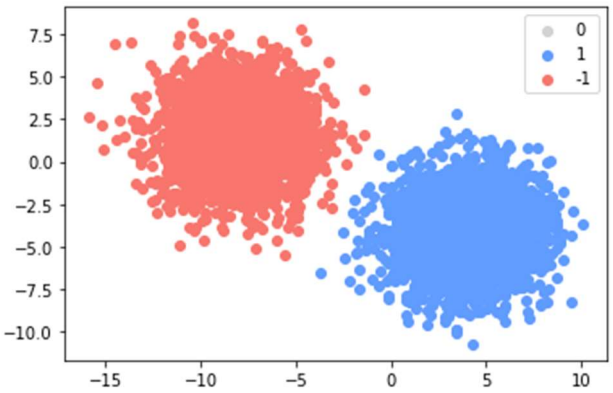
The stopping condition occurs when the relative loss is less than 1e-8, which is a form of early stopping.

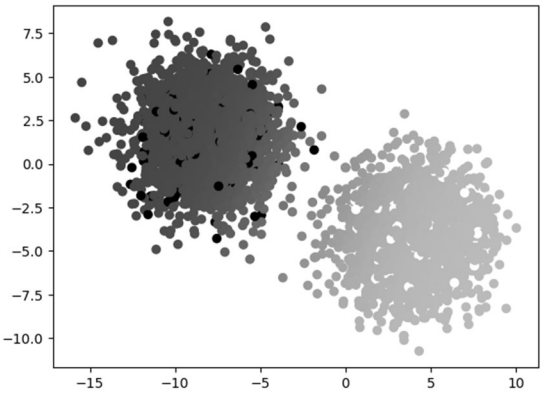
### 6.3 Results

We start with a dataset with 5000 data points and 5% of labelled examples:

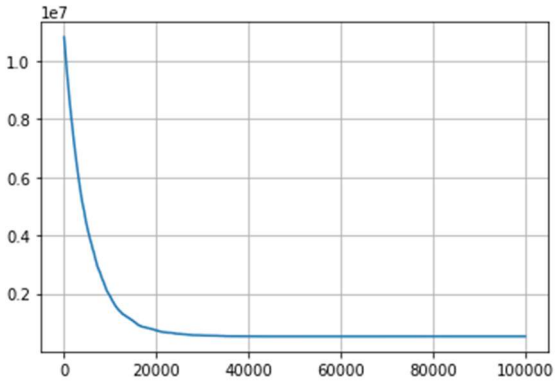



The algorithm runs until maximum iterations are reached.




Let's look at the loss:



### 6.4 Notes

1. Randomized Block Coordinate Gradient Descent is very slow (116s)

## 7. Public available dataset

### 7.1 Dataset description

The dataset consists of medical data from 5 different hospitals. This data informs us of the presence of coronary heart disease.

Our goal is to predict the presence of this condition given the other parameters.

This dataset has a 12 parameters and 918 observations.

The parameters are:

1. **Age**: age of the patient [years]

2. **Sex**: sex of the patient [M: male, F: female]

3. **ChestPainType**: chest pain type [TA: typical angina, ATA: atypical angina, NAP: Non-anginal pain, ASY: asymptomatic]

4. **RestingBP**: resting blood pressure [mmHg]

5. **Cholesterol**: serum cholesterol [mm/dl]

6. **FastingBS**: blood sugar level after an overnight fast. [1: if FastingBS $>$ 120 mg/dl, 0: otherwise]

7. **RestingECG**: resting electrocardiogram results [Normal: normal, ST: having ST-T wave abnormality, LVH: showing probable or definite left ventricular hypertrophy by Estes' criteria]

8. **MaxHR**: maximum heart rate achieved [numeric value between 60 and 202]

9. **ExerciseAngina**: exercise-induced angina [Y: yes, N: no]

10. **Oldpeak**: depression of the ST slope [numeric value between -2.6 and 6.2]

11. **ST_Slope**: the slope of the peak exercise ST segment [Up: upsloping, Flat: flat, Down: downsloping]

12. **HeartDisease**: output class [1: heart disease, 0: normal]

### 7.2 Data preparation

Firstly we imported our dataset on Python.

The numerical variables contain N/A values that are indicated by the number 0.

#### 7.2.1 Features

After having selected the 11 features we convert them in a numpy 2 dimensional array.

**Numerical features:**

There are 5 numerical features.

Firstly we replaced every N/A values with the median of the respective feature. (We prefer the median compared to the mean since it is less sensible to outliers in the dataset)

Then we normalized each parameter by dividing each value by the maximum value of the respective parameter.

```
median=np.median(data[:,[0,3,4,7,9]],axis=0)
maxx=np.max(data[:,[0,3,4,7,9]],axis=0)
print(median)
for idx,col in enumerate([0,3,4,7,9]):
  for row in range(918):
    if data[row,col]==0:
      data[row,col]=median[idx]
    data[row,col]=data[row,col]/maxx[idx]
```

**Categorical features:**

The loss function provided in the specification doesn't support categorical variables.

Thus we used one hot encoding, creating for a n valued categorical feature n 0-1 vectors, for example for the variable ChestPainType:

```
for i in range(918):
  if data[i,2]=='TA':
    pain[i,0]=1
  elif data[i,2]=='ATA':
    pain[i,1]=1
  elif data[i,2]=='NAP':
    pain[i,2]=1
  elif data[i,2]=='ASY':
    pain[i,3]=1
```

#### 7.2.2 Label

Firstly we selected it from the original dataset and converted it in a numpy array.

Then we randomly selected some of the labels that will be known by the algorithm, while keeping all the other equal to 0.

```
y=np.zeros(len(heart))
for i in range(len(heart)):
  k=np.random.uniform(0,1,1)
  if heart[i]==0:
    heart[i]=-1 #The original dataset has 0 and 1 as labels, we want -1 and 1.
  if k<P_LABLP and heart[i]==1:
    y[i] = 1
  elif k<P_LABLN and heart[i]==-1:
    y[i] = -1
```

### 7.3 Algorithm implementation

Firstly we calculated the weights and the appropriate stepsize.

We also defined a metho capable of calculating various performance metrics such as:

accuracy, precision and recall.

```
def metrics(heart,y,uidx):
  TP=0
  TN=0
  FN=0
  FP=0
  for i in uidx[0]:
    if heart[i]==1 and y[i]>0:
      TP+=1
    elif heart[i]==1 and y[i]<0:
      FN+=1
    elif heart[i]==-1 and y[i]>0:
      FP+=1
    elif heart[i]==-1 and y[i]<0:
      TN+=1

  ACC=(TP+TN)/(TP+TN+FN+FP)
  PRE=(TP)/(TP+FP)
  REC=(TP)/(TP+FN)
  return ACC,PRE,REC
```

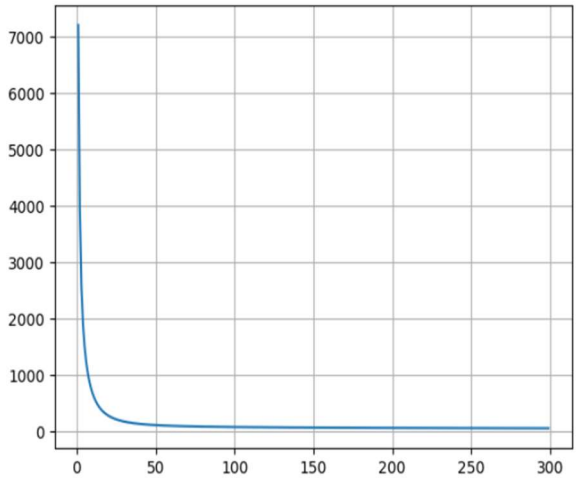
#### 7.3.1 Gradient Descent

We set the number of iteration to 300 and ran the gradient descent algorithm.

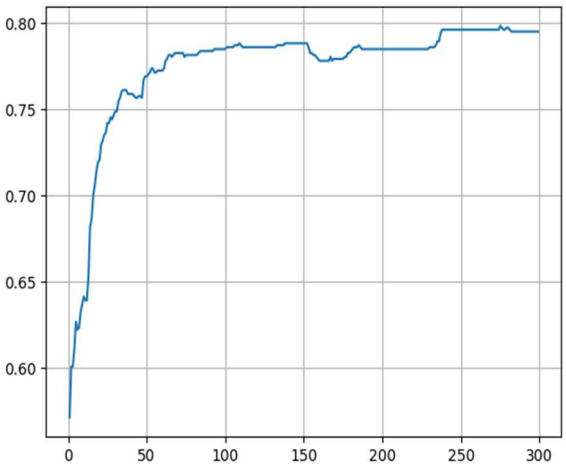
We also tracked the time passed at each iteration with the library time.

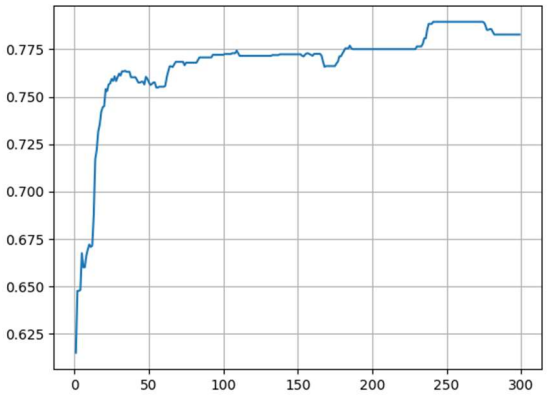
These are the results:

Loss:

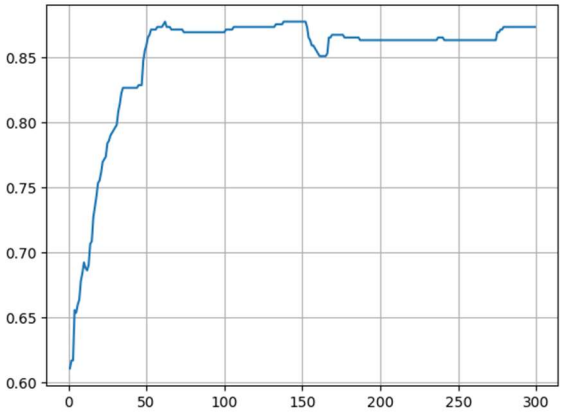



Accuracy:



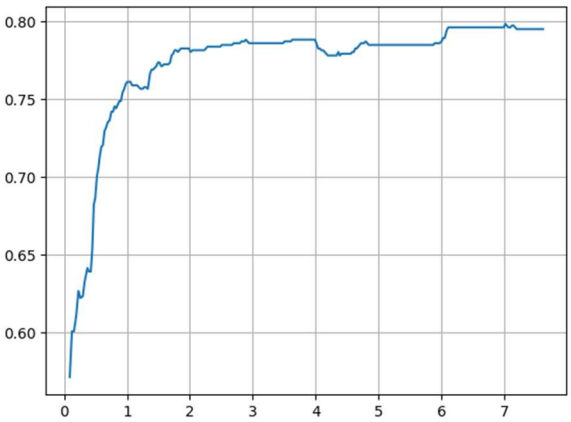




Precision:




Recall:

We can see that the loss always decreased and that all 3 metrics increased and then stabilized around or at its maximum level (slightly lower 80% for the first two, above 85% for the recall)



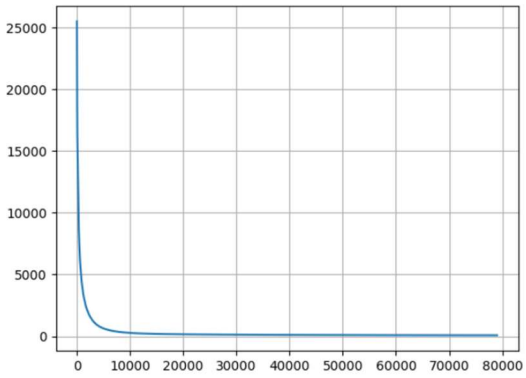
The graph above shows the evolution the accuracy with respect to the time passed. We can see that the accuracy firstly jumped and then stabilized slightly below 80%

### 7.3.2 Gauss-Southwell BCGD

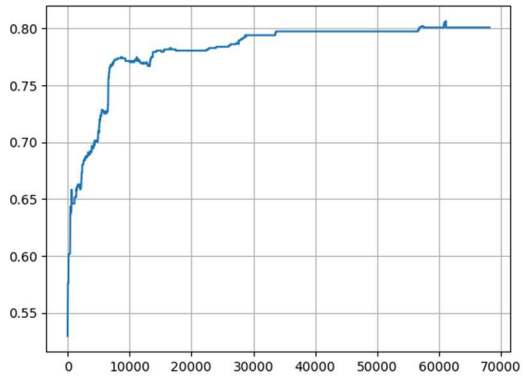
We tracked the time passed at each iteration with the library time.

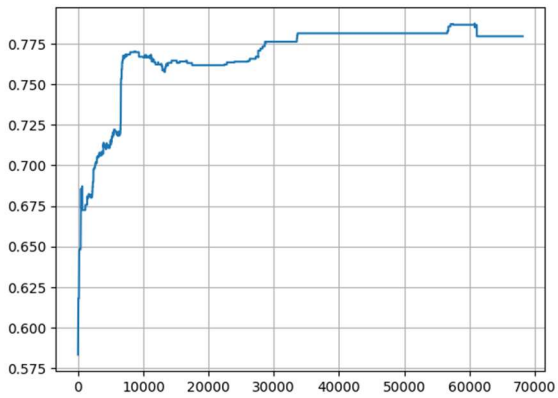
These are the results:

Loss:

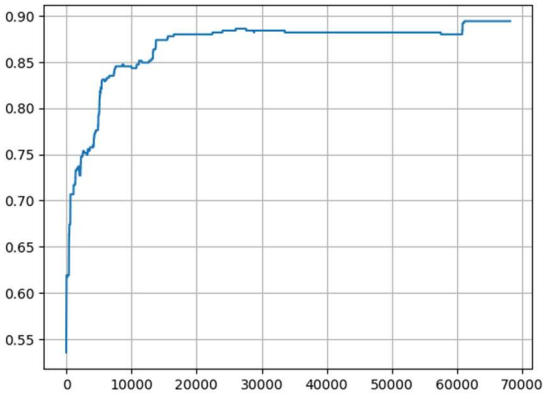



Accuracy:



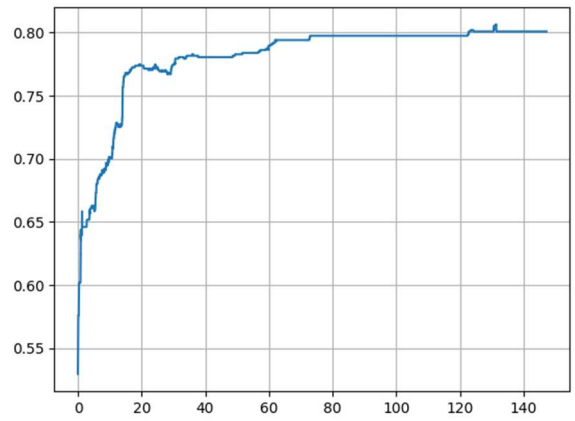




Precision:




Recall:

We can see that the loss always decreased and that all 3 metrics increased and then stabilized around or at its maximum level (slightly lower than 80% for the first 2, almost 90% for the recall)



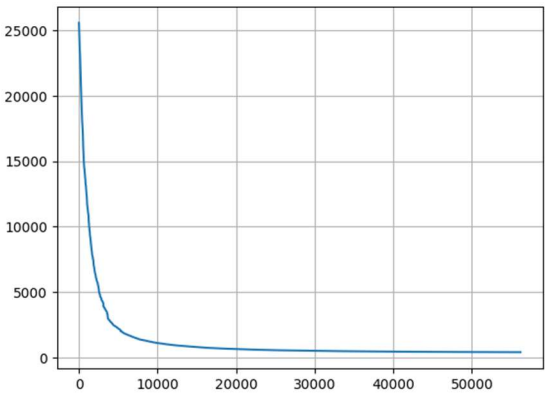

The above graph shows the evolution the accuracy with respect to the time passed. The accuracy rapidly jumped in the beginning and then stabilized.

### 7.3.3 Randomized BCGD

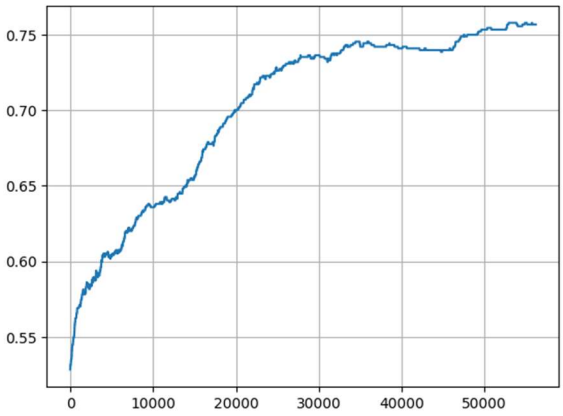
We tracked the time passed at each iteration with the library time.

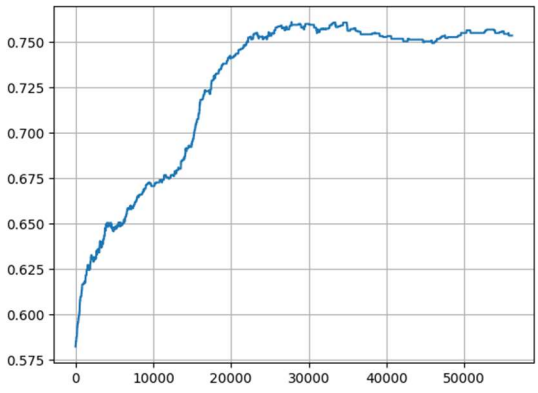
These are the results:

Loss:

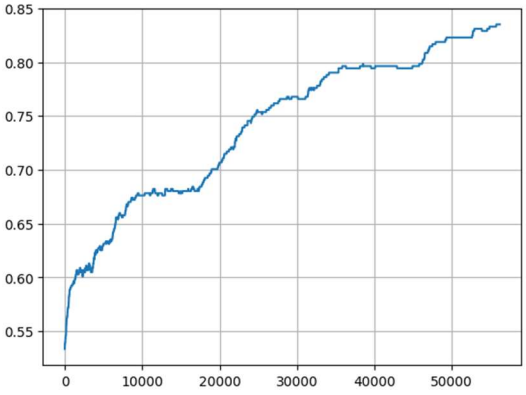



Accuracy:




Precision:




Recall:

We can see that the loss always decreased and that all 3 metrics increased and then stabilized around or at its maximum level (slightly above 75% for the first 2, slightly lower than 85% for the recall)

### 7.4 Conclusion

We obtained the following results as the final value of our metric accuracy,precision and recall:

Gradient descent:

```
(0.7952488687782805, 0.7828467153284672, 0.8737270875763747)
```

Gauss-Southwell BCGD:

```
(0.8009049773755657, 0.7797513321492007, 0.8940936863543788)
```

Randomized BCGD:

```
(0.7567873303167421, 0.7536764705882353, 0.835030549898167)
```

Randomized BCGD has worse performance than the previous two models in all 3 metrics.

Gauss-Southwell has 0.5% higher accuracy,0.3% lower precision, 2.1% higher recall compared to Gradient descent.

In order to decide which model is the best one we need to observe that a FP (someone who was predicted to have an heart failure but didn't) is much more preferable than a FN (someone who was predicted to be fine and then had an heart failure farther from the hospital).

Thus we want to prioritize recall over precision.

We can conclude that:

The best model is the one obtained with Gauss-Southwell BCGD

The second best model is the one obtained with Gradient descent

The worst model is the one obtained with Randomized BCGD