Semi-supervised learning with gradient-based methods

Roberto Russo roberto.russo.4@studenti.unipd.it

June 13, 2022

1 Introduction

The aim of this work is to compare Gradient Descent (GD), Randomized BCGD (RBCGD) and Cyclic BCGD (CBCGD) for solving a problem of semi-supervised learning. To do so, these 3 algorithms were tested on an artificial dataset composed of 10, 000 randomly generated points with two classes, in a *2D* space. After comparison on an artificial toy dataset, the models were applied to the real data. As results we will see how the cyclic BCGD method outperform all the others.

2 Artificial Dataset

The artificial dataset is composed of two clusters with a standard deviation equal to 1.5 the center of the two clusters is such that those clusters intersect each other. 5% have labels while the other part is assigned the value 0.5.

We then distinguish the following groups of data:

- l labeled examples of the format

$$(\overline{x}_{1i}, \overline{x}_{2i}; \overline{y}_i) \text{ for every } i = 1, \dots, l;$$

- U unlabeled data of the format

$$(x_1 j, x_2 j) \text{ for every } j = 1, \cdots, U.$$

3 The goal

The goal is to infer the category to which the unlabeled samples belong using the labeled samples. In the

Figure 1: Distribution of data. The two clusters represented

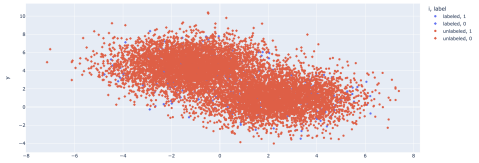

Figure 2: Distribution of data. The blue point represent the labeled data while the red point are without label

semi-supervised learning framework this problem can be formalized in the following way:

$$min_{y \in \mathbb{R}^U} \sum_{i=1}^{l} \sum_{j=1}^{U} w_{ij}(y_j - \overline{y}_i)^2 + \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{U} \overline{w}_{ij}(y_i - y_j)^2$$

where w_{ij} is the weight associated to the points y_j and $\overline{y}_i$, while $\overline{w}_{ij}$ is the weight associated to the points y_j and y_i. The weight measures the similarity between the points and is computed in this way:

$$w_{ij} = \frac{(x_{1i} - \overline{x}_{1j})^2 + (x_{2i} - \overline{x}_{2j})^2}{\sum_{j=1}^{U} \sum_{i=1}^{L} (x_{1i} - \overline{x}_{1j})^2 + (x_{2i} - \overline{x}_{2j})^2}$$

$$\overline{w}_{ij} = \frac{(x_{1i} - x_{1i})^2 + (x_{2i} - x_{2j})^2}{\sum_{j=1}^{U} \sum_{i=1}^{L} (x_{1i} - x_{1j})^2 + (x_{2i} - x_{2j})^2}$$

The search for optimal assignments is performed in the space R^U, where U is the cardinality of the set of unlabeled data and L is the cardinality of the labeled data.

4 Algorithms

Each optimization algorithm is based on gradient and follows this updating rule:

$$x_{k+1} = x_k + \alpha_k d_k,$$

where α_k is the stepsize and d_k it's the descent direction. For every choice regarding the search direction d_k, that in gradient methods is the anti-gradient w.r.t x. For this problem, the gradient for each component i at iteration k is defined as follow:

$$\nabla_i f(y_{ik}) = 2\sum_{j=1}^{L} w_{ij}(y_i - \overline{y_j}) + \sum_{j=1}^{\cup \overline{w}_{ij}(y_i - y_j)}$$

4.1 Gradient Descent

Gradient Descent takes the stepsize α_k fixed and the descent direction as follows:

$$d_k = -\nabla f(x_k), \tag{1}$$

where $f(\cdot)$ is the objective function.

4.2 BCGD methods

BCGD methods differ from simple GD only when x is not a scalar, because the variables of interest are not updated altogether, but they are divided into blocks and each block is updated separately according to its specific rule. Those methods are particularly powerful when the objective function has some specific structure that can be exploited to speed up the computation of the gradient, for example, sparse optimization problems. The variables are partitioned in b, each of them of dimension n_i such that:

$$n = \sum_{i=1}^{b} n_i.$$

b matrices $B_i \in \mathbb{R}^{n \times n_i}$ are defined such that each block $x^(i)$ will be obtained like in the following:

$$x^{(i)} = B_i^T x,$$

the descent direction for each block is then obtained as follows:

$$d_k i = -B_i^T \nabla f(x).$$

For this work $n_i = 1 \; \forall \; i = 1, \ldots, n$ where $n = U$ given in input. Hence, each block U_i will be a vector corresponding to updating one variable at the time.

The several BCGD methods implemented differs on the strategy of selecting the blocks at each iteration:

- BCGD with *cyclic strategy*: for every iteration, one coordinate at a time is updated sequentially;
- BCGD with *randomized strategy*: for every iteratiob, a single coordinate is updated and its order it's chosen uniformly random.

5 Implementation

5.1 Common setting

. For each algorithm, the vector of unlabeled samples is initialized setting every value equal to 0.5. For each test, the same dataset is used for all three algorithms and the same samples are been unlabeled. In all cases there are two stopping conditions:

- The maximum number of iteration equal to 100 is reached;
- the norm of the gradient drops below a tolerance threshold set equal to 0.1 for the first experiment and 10 for the second.

The tolerance is chosen after preliminary experiments, since there is a region where the two clusters are overlapped, it is sensible to assume that would be impossible to reach the value of zero for the objective function, and 10 has shown to be a tolerance that allows to reach almost the best performance we could get (using, for example, more labeled data). But for the first experiment, I chose 0.1 to see how much time and how many iterations it gets for each algorithm to reach the

stopping condition, and to observe the behavior of the cost and accuracy. The weights are initialized and stored in two matrices W and $\overline{W}$, respectively for unlabeled-labeled of dimension UxL and unlabeled-unlabeled of dimension UxU, where each element is the similarity measure between two points before the optimization starts. The algorithms are evaluated according to four quantities:

- The value of the objective function at the end of the optimization;
- the accuracy;
- the number of iterations;
- the number of seconds.

The accuracy is computed classically, comparing the y to the vector of the correct label that has been previously stored, but in this case, the elements of y are rounded to the nearest integer (0 or 1).

5.2 Gradient Descent

For GD I choose to use a fixed step size set to 0.1 after preliminary experiments. For sake of CPU times, implemented a cashing strategy that stores the common operand of the gradient and the cost function at each iteration, i.e. $y_i - \overline{y_j}$ and $y_i - y_j$, and for each couple (i, j) of both terms, the differences are computed in parallel and stored in two matrices $\overline{Q_{y\overline{y}}}$ and Q_{yy} as for the weights. Since the cost function can be seen as the sum of all the elements of the element-wise product of $W\overline{Q^2_{y\overline{y}}}$ summed to the sum of all the elements of the element-wise product of $\overline{W}\overline{Q^2_{yy}}$, where the presence of the square means that every element of the matrix has been squared, also the products between the weights and the squared differences have been computed in parallel. The same holds for the gradient, except for the summation of the element of the matrix, that has been made w.r.t the rows, yielding two vectors of dimension U that are summed element-wise to yield the final gradient. The cost and the accuracy are computed and stored at every iteration.

5.3 BCGD methods

For the BCGD methods used here, the implementation is almost the same, so their implementation will be discussed together mentioning the differences. The formula of the gradient can be rewritten in this way:

$$\nabla_i f(y_{ik}) = 2y_i \sum_{j=1}^{L} w_{ij} - 2\sum_{j=1}^{L} w_{ij}\overline{y_j} - y_i \sum_{j=1}^{U} \overline{w}_{ij} - \sum_{j=1}^{U} \overline{w}_{ij} y_j \quad (2)$$

Because y_i remains constant during the summation it can be put outside the summation, what remains inside, in the first and third terms, never changes. Hence, those values are calculated once and then stored, and then used when it is time to update the related variable. Also, the second term never changes, so it is stored and used as described before. Only the last term is computed every time, saving CPU time. For this implementation, I use the exact line search. If I call in order the four terms of the equation of the gradient a, b, c, d, the exact line search is:

$$lr = \frac{y_i}{\nabla_i f(y_{ik})} + \frac{2b + d}{2a + c}\nabla_i f(y_{ik}) \quad (3)$$

That is very efficient because it uses all terms that have already been computed. Since the gradient appears in the denominator, I impose a condition that when the gradient is zero, I do not update the component and remove that component from the list of components to update, for the rest of the optimization. To compute the cost in this case I do not exploit any caching strategy but it is still very efficient as every independent operation is performed in parallel. For the Randomized BCGD, since for every iteration it updates just one variable, to make the algorithm faster and avoid useless computation of the cost and the accuracy, (because they would probably not change in a few iterations), and also to make the number of iteration comparable with the other algorithms, the number of maximum iterations that is fed in input is multiplied by U. The cost and the accuracy are calculated once every U iteration. While for cyclic BCGD the cost and the accuracy are computed at the end of each iteration. For the randomized BCGD, the norm of the gradient is computed only starting from when 85% of the variables have been updated but only when the number of iterations is a multiple of U.

6 Results and discussion

Thus, figures 3 and **??** illustrates the loss decreasing versus iterations. I first run the experiment with one dataset to analyze iteration by iteration the behavior of the algorithms, then I run another experiment on ten different datasets to see if the behavior stays the same, for each dataset the unlabeled samples were the same for each algorithm. We can see the performances of the first experiment 3

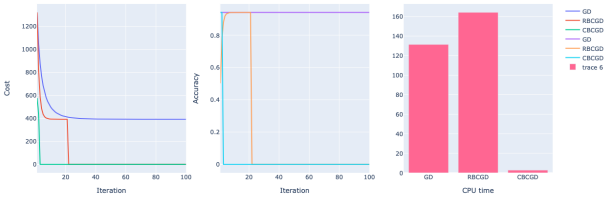

Figure 3: The cost, The accuracy, and the number of seconds of the three algorithms. In the first two plots, when there is the drastic drop, it means that the algorithm has stopped because the gradient dropped below the tolerance

We can observe as all the algorithms converge fast, and how cyclic BCGD outperforms the others. The accuracy from the beginning is really high, because having set every unlabeled sample equal to 0.5 any change in the right direction will make, by rounding to the nearest integer, to get the exact label. This is also why we can afford to use a relatively high tolerance, if we were interested in decreasing the cost, we could set a lower tolerance, but after a certain point, the cost will stop decreasing, never touching zero but staying near to the cost value that we can reach with tolerance equal to 10.

The results of the second part of the experiment are shown in the figures below.

We can see as the performance in terms of cost and accuracy stays similar over the algorithms w.r.t. the same dataset, the cyclic BCGD method still outperform in term of CPU time, and almost always in term of iteration. The fact that the randomized BCGD method has always the same number of iterations is shown in the plot means that

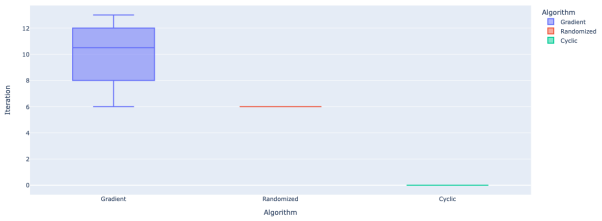

Figure 4: Distribution of the iterations

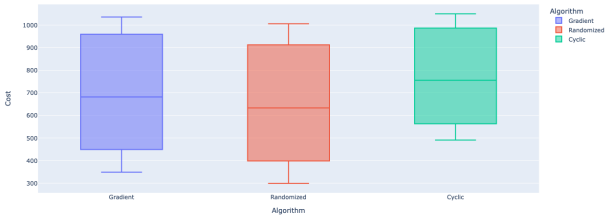

Figure 5: Distribution of the cost

it stops with an iteration number between $6U$ and $7U$. To be noticed also that the cost of cyclic BCGD is slightly higher, due to the fact the tolerance set to 10 makes the algorithm stop at the first iteration.

7 Real dataset

The real dataset that I chose to test the algorithms is a randomly selected subset of 10000 samples taken from the Skin Segmentation Dataset, where every sample is a colored pixel taken from a different image, and the label is 1 if the pixel represents the skin of a human, and 0 otherwise. I downloaded the whole dataset from UCI Machine Learning Repository, then randomly select a subset of it. Each sample has three variables, corresponding to the color channels. Since I wanted to work in the two-dimensional space, I first reduce the dimensionality of the data through PCA, taking the first two components. With this trasformation, most of the information remains preserved, as we can see from the 8. So the final dataset looks like this: The procedure is the same as for the first experiment, the tolerance is set to 1.

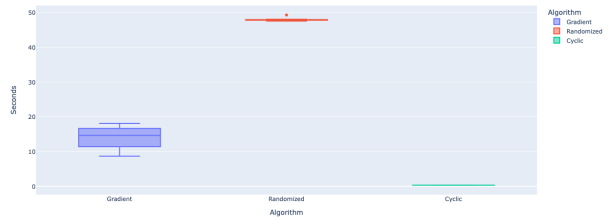

Figure 6: Distribution of the secods

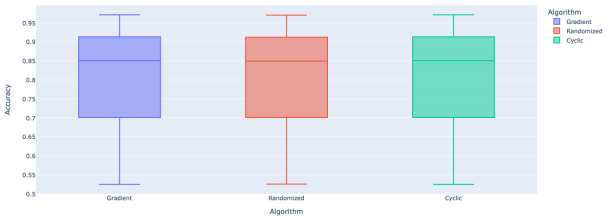

Figure 7: Distribution of the accuracy

The results are shown in the 10.

8 Conclusion

Through this paper I analyzed the performance of three algorithms, first in with artificial examples, and then on a real dataset, showing, in the last case, that even with few labels, it is possible to retrieve much of the information, and with just one pixel we can separate human skin from what is not human skin.

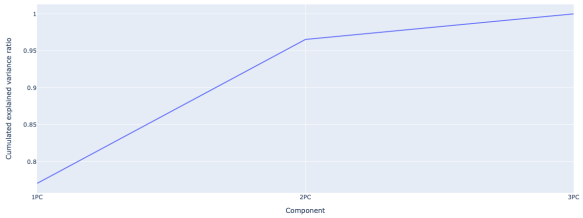

Figure 8: Cumulative explained variance ratio

Figure 9: Real dataset after being trasformed

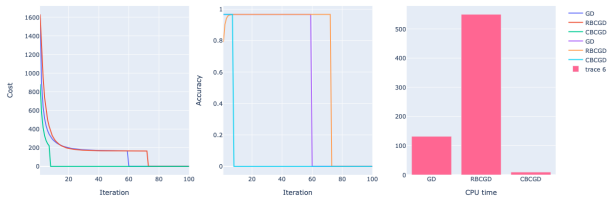

Figure 10: Performance comparison of the algorithms