Exploring Gradient Descent and BCGD Methods for Data Labelling in a Semi-Supervised Learning Context

Riccardo Carangelo

University of Padua, Department of Mathematics, M.Sc.in Data Science, mail: riccardo.carangelo@studenti.unipd.it, ID: 2057432

1 Introduction

In the field of machine learning, an important challenge is the problem of data labelling. In general, it is nowadays easier to find large amounts of data than it is to label them, making this absence of label a significant problem for any given supervised learning task. A potential solution for this problem can be provided by Semi-Supervised Learning (SSL) models, which can handle both labelled and unlabelled data.

In this report, the application of Gradient Descent (GD) and Block Coordinate Gradient Descent (BCGD) algorithms is used and investigated in a semi-supervised classification learning context. The aim is to evaluate the accuracy of the proposed method for a label assignment task, starting from a small portion of label data.

2 Methods

2.1 Problem building

The first step is to consider a dataset with n examples and m features. It is possible to build from such a set a data matrix $\boldsymbol{X} \in M_{n\times(m-1)}(\mathbb{R})$ (from now on vectors and matrices are indicated in bold for distinguish them from scalars), a matrix containing all the datapoints (without considering the target vector) and the vector $\boldsymbol{y} \in \mathbb{R}^n$, containing the target variable with all the labels. A i-th point of $\mathcal{X}$ can be written as $x^{(ij)} = \left(x^{i1}, \dots, x^{im}\right)$, given $i \in \{1, \dots, n\}$.

The main method used in this work consists in hiding a portion of the labels for a specific percentage of examples in order to build a semi-supervised problem. In this way, it is possible to get the following objects, which are a partition of $\mathcal{X}$ and $\boldsymbol{y}$ (Figure 2):

- $\bar{\boldsymbol{X}} \in M_{l\times(m-1)}$, containing the labelled examples.
- $\bar{\boldsymbol{y}} \in \mathbb{R}^l$, containing the labels for the labelled examples (to be used as a target for the learning task).
- $\boldsymbol{X} \in M_{u\times(m-1)}$, containing the unlabelled examples.
- $\boldsymbol{y_t} \in \mathbb{R}^u$, containing the labels for the unlabelled examples (this vector has been hidden and it has been just used for calculating accuracy).

With l referring to labelled data and u referring to unlabelled data, such that $l + u = n$.

Before starting, the data has been normalized, using a standard scaler, which uses the following normalization:

$$\bar{z}^{(ij)} = \frac{\bar{x}^{(ij)} - \mu_X}{\sigma_X} \forall i \in \{1, \dots, l\} \forall j \in \{1, \dots, m-1\}$$

Given an element x_{ij} of the matrix $\bar{\boldsymbol{X}}$ and the mean μ_X and standard deviation σ_X of

$\bar{\boldsymbol{X}}$. In this way the normalized matrix $\bar{\boldsymbol{Z}} \in M_{u\times(m-1)}$ has been made.

Normalized data are then used to extract two Euclidean distance matrices:

$$d_{i_1,i_2} = \sqrt{\sum_{j=1}^{m}(z^{(i_1 j)} - z^{(i_2 j)})^2}\ \forall i_1 \in \{1,\dots,l\}\forall i_2 \in \{1,\dots,u\}$$

$$\bar{d}_{i_1,i_2} = \sqrt{\sum_{j=1}^{m}(z^{(i_1 j)} - z^{(i_2 j)})^2}\ \forall i_1, i_2 \in \{1,\dots,u\}$$

Where i_1 and i_2 are indices for selecting couple of normalized points in $\bar{\boldsymbol{Z}}$.

In this way it is possible to get the matrix $\boldsymbol{D}$, calculated measuring the Euclidean distances between all the possible couples of labelled and unlabelled points, and the matrix $\bar{\boldsymbol{D}}$, calculated measuring the Euclidean distance between all the possible couples of unlabelled and unlabelled points.

The Euclidean distance matrices are finally used to extract feature weights $\boldsymbol{W}$ (weights between couple of points from labelled and unlabelled data, respectively) and $\overline{\boldsymbol{W}}$ (weights between couple of points from unlabelled and unlabelled data, respectively), which are fundamental for the definition of a useful loss function. The extraction has been performed by using a Radial Basis Function (RBF) kernel as a similarity measure:

$$w_{i_1,i_2} = e^{-\gamma d_{i_1,i_2}}\forall i_1 \in \{1,\dots,l\}\forall i_2 \in \{1,\dots,u\}$$

$$\bar{w}_{i_1,i_2} = e^{-\gamma \bar{d}_{i_1,i_2}}\forall i_1, i_2 \in \{1,\dots,u\}$$

In which γ is a free hyperparameter that can be tuned for the best results and the best balance for getting a good generalization and avoiding overfitting.

2.2 Loss function and gradient

The Loss function f proposed for solving the labelling problem uses the target vector $\bar{\boldsymbol{y}}$ and the weights calculated in the previous paragraph is the following:

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

Here, $\boldsymbol{y}$ is the vector of the labels that must be found.

Thus, the main problem consists in finding:

$$\min_{\boldsymbol{y}\in\mathbb{R}^u} f(\boldsymbol{y})$$

For this purpose, the following gradient is considered (here shown for the component j component of $\boldsymbol{y}$, given $j \in \{1,\dots,u\}$):

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

2.3 Learning rate

In order to get a good reference for the choice of the proper learning rate to be used in the loss function minimization task, the Hessian matrix has been calculated for extracting the Lipschitz constant using its eigenvalues. The form of the Hessian matrix is the following:

$$\boldsymbol{H_{jk}} = \nabla^2_{y^{(j)}y^{(k)}} f(\boldsymbol{y}) = \begin{cases} 2\left[\left(\sum_{i=1}^{l} w_{ij} + \sum_{i=1}^{u}\bar{w}_{ij}\right) - \bar{w}_{ij}\right] & \text{if } h = k \\ -2\bar{w}_{jk} & \text{if } j \neq k \end{cases}$$

The proof for this form is provided in the *Supplementary Materials* paper (Sect. 1).

For this part, it is important to notice that the numerical method used by NumPy for calculating the Hessian matrix can lead to an approximation and, thus, to an instability of the value of the Lipschitz constant through different calls on the function. For reducing this effect, a Tikhonov Regularization term has been

introduced and also a condition number has been calculated, so as to keep monitored the conditioning status of the matrix. The multiplicative scalar α for the regularization term as been coded as a free hyperparameter, which can be tuned for each problem, following the value of the condition number. In this way it is possible to get the following Tikhonov Regularization for the Hessian matrix:

$$\boldsymbol{H_{jk,reg} = H_{jk} + \alpha I_{\mathbb{R}^{u \times u}}}$$

In this way the value of the Lipschitz constant is more stable through multiple runs of the code. Even for getting the Lipschitz constant, NumPy has been used (for further information, *Supplementary Materials,* Sect. 3)

2.4 Algorithms

The two algorithms tested are a gradient descent (GD) and a block coordinate gradient descent (BCGD).

The gradient descent uses the simplest gradient-based updating rule, taking the gradient previously defined, multiplying it by the learning rate β and using the result for updating $\boldsymbol{y}$ multiple times through a specific number of iterations. At the iteration h the algorithm update $\boldsymbol{y_{h+1}}$ as follows:

$$\boldsymbol{y_{h+1} = y_h - \beta \nabla f(y_h)}$$

The number of iterations is a free hyperparameter which can be differently tuned for different problem types.

The BCGD exploits the same concept but updating blocks of coordinates instead of the entire vector at a time. For this purpose, two different methods have been used for block selection.

The first method used is the Gauss-Southwell method, which takes at each iteration the block i from the set $\{1, \dots, b\}$ of all the blocks, such that:

$$i = \underset{j \in \{1,\dots,b\}}{\operatorname{argmax}} \left\| \nabla_j f(\boldsymbol{y_k}) \right\|$$

A problem of this method is the computational expensiveness, since, using a greedy approach, it would be necessary to calculate all the gradient for each iteration, while updating only one component of the vector at a time (the one whose norm maximises the gradient). To easily solve this problem, it is possible to use a dynamic programming approach, keeping track of the previously calculated gradients and modify only the gradient of interest, which is the one used to update the vector in the last iteration. In this way the Gauss-Southwell implementation, even if taking more memory than the greedy approach, showed a quite faster run time.

The second method used is the random permutation method, which takes in order all the blocks after performing a random permutation of them $\{1, \dots, b\}$.

Before starting the gradient computation, the $\boldsymbol{y}$ vector is randomly initialized and then given as input to the gradient descent or to the BCGD. During the iterations, the entries of the $\boldsymbol{y}$ vector should tend to the actual numbers of the class labels. The simplest way to test the accuracy in this case is to round all the entries of the $\boldsymbol{y}$ vector to closest discrete units, since in this case numerical class has been considered as discrete classes with values in $\mathbb{N}$. After the rounding step, a simple entry-by-entry comparison between $\boldsymbol{y}$ and $\boldsymbol{y_t}$(the target hidden vector extracted before) components can give a basic accuracy measure in the interval $[0,1]_{\mathbb{R}}$.

During the execution of the iterations, several informative values are collected:

the simple accuracy described above, the loss value and the time required for calculation. All these values are used for final analyses and visualizations.

2.5 Code and libraries

All the problem has been coded using Python 3.8, both in simple *.py* file format (for coding the libraries) and in *.ipyinb* notebook format (for the actual tests of the model). Several libraries have been used for visualizing and manipulating data, for introducing numerical calculus tool and for importing and managing datasets. Among the external libraries used there are NumPy, Pandas, Scikit-Learn, Matplotlib and Seaborn.

3. Datasets

The main problem has been faced by dividing it into two subsections, a first section for exploring and calibrating the model, using a synthetic dataset and a toy dataset, and a second section in which the tested model is applied to a real dataset.

3.1 Exploratory Analysis and Model Calibration

In this part the methods are tested and analysed using two datasets:

- An artificially generated synthetic dataset.
- A toy dataset ("Penguins") taken from the Seaborn library examples.

The purpose here is to establish an efficient and fast method for the convergence of a given gradient in order to apply this method to a real dataset for the second part of this work.

The first used dataset is a simple bidimensional synthetic dataset which is generated using a specific `make_data` function. This function is able to create blobs, moons and concentric circles (the latter is the set used for this first part) (Figure 1). The advantage of this kind of set is the possibility of choosing several characteristics, such as the shape of the set, the noise and the variance. Here, the target vector is a simple numerical vector with two classes: 0 and 1.

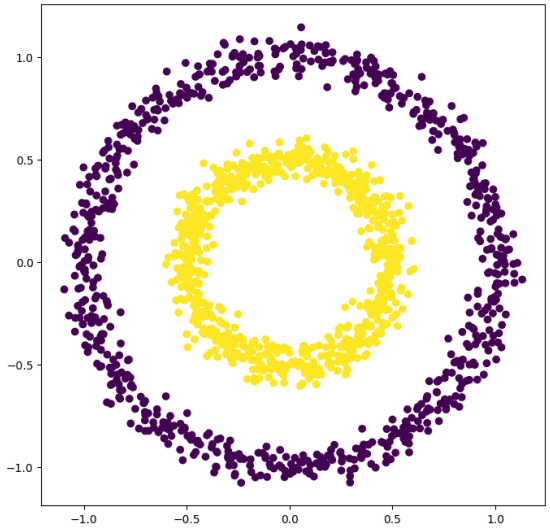

Figure 1 – 2D concentric circles datapoints with two different classes, highlighted in yellow and dark purple colours. This is the first set used to test the model.

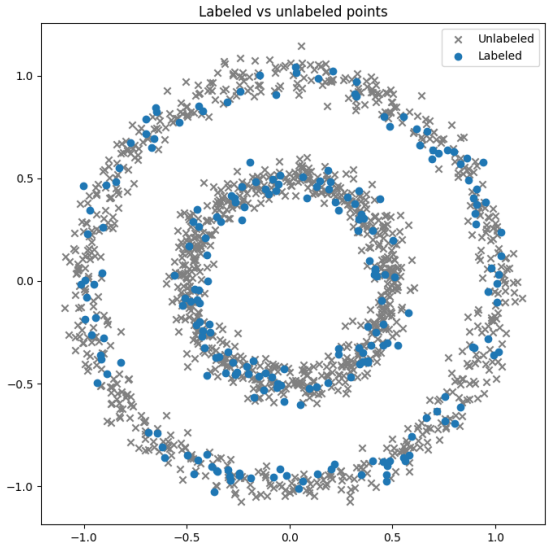

Figura 2 – concentric circles datapoints with highlights for labelled data (randomly selected), showing also the unlabelled data, which constitutes the majority of the dataset.

The other dataset used is the penguin toy dataset from Seaborn example dataset, a

set containing several information about body size, geographical distribution and species of several penguins, with a two classes target variable for predicting sex. For this dataset was necessary some brief EDA and pre-processing, for deleting null values and converting the string categorical classes to discrete numerical classes.

3.2 Real dataset application

The method developed in the previous testing part has been then used for this second part in a real dataset case. The real dataset chosen for this work is the Diabetes prediction dataset (available on Kaggle at https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset?resource=download), a collection of medical and demographic data from patients. The target variable is the diabetes status of each patient, which has two classes: positive or negative.

The dataset has a size of 100000 samples, with 8 features. For the purpose of this work, the full dataset has been imported and then subsampled to include only a random subsample of 1% of the total dataset, in order to show that just a 1000 subsample, apart from obviously providing a faster execution compared to the full sample of 100000 patients, provide in the meantime enough information for reaching nice results (Table 1).

Even in this case, some string format categorical classes has been converted to discrete numerical classes before starting with the gradient's calculation.

To further slim the whole process, to lead to more stable and faster algorithms and to provide higher accuracy results, a brief simple feature selection step has introduced, in order to discard less informative features, using mutual information, a model independent method based on the mutual dependence of 2 variables (*Supplementary Materials*, Sect. 2).

For each of the three datasets the weights have been extracted using the pipeline shown before. Also, the inverse of the Lipschitz constant has been calculated and used as a reference for studying the accuracy variation for several learning rates in the two first test datasets, by taking several values from the neighbourhood of $\frac{1}{L}$. This grid search-like approach driven by $\frac{1}{L}$ has been chosen in order to apply a purely empirical method for analysing the accuracy behaviour in relation to the learning rate variation around $\frac{1}{L}$ (Table 1).

4 Results

Both the synthetic dataset and the Penguins dataset showed that the $\frac{1}{L}$ is a good choice when performing the gradient descent and the BCGD (Figure 3 and Figure 4), even if the $\frac{1}{L}$ value tends to slightly underestimate the best learning rate in the case of BCGD (all the plots that clearly shows this phenomenon can be seen in *Supplementary Materials* Sect. 5). This empirical evidence suggest that a block-based method could be not fully compatible with the Hessian method performed before, which instead seems to perfectly fit the gradient descent context.

Basing on these ideas, the naïve strategy for the part regarding the real dataset analysis was to use the value of $\frac{1}{L}$ as a learning rate value for the gradient descent, while greater values in the

interval $\left[\frac{10}{L}, \frac{15}{L}\right]_{\mathbb{R}}$ was tested for the BCGD (*Supplementary Materials*, Sect. 4).

Even though this strategy was relatively simple and based on simplistic intuition, it worked quite well for the real dataset, providing high accuracies and fast calculations even for a 1% subsample of the total set (Table 1).

The loss and accuracy plots in the *Supplementary Materials* paper (Sect. 6) paper show the loss descent and the accuracy trend for the real dataset, while the results accuracy summary is shown in Figure 5. Finally, all the hyperparameters chosen for each dataset can be seen in a unique table in the *Supplementary Materials* paper (Sect. 4).

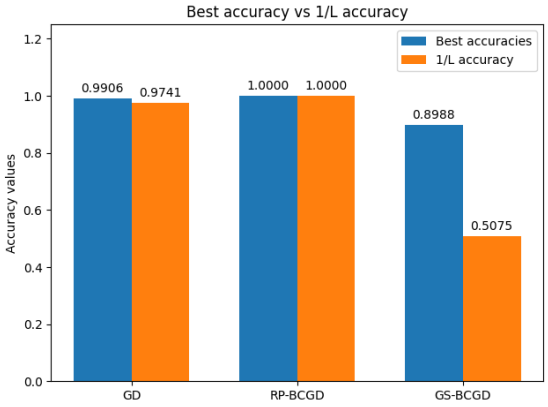

Figure 3 – Side-by-side barplot for comparing the accuracies found using the value of $\frac{1}{L}$ and "best accuracies" found for the synthetic dataset.

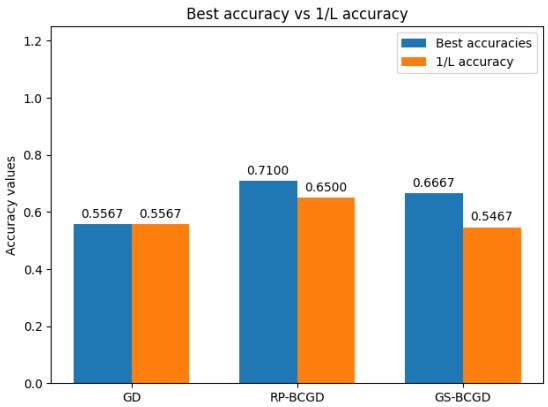

Figure 4 – Side-by-side barplot for comparing the accuracies found using the value of $\frac{1}{L}$ and "best accuracies" found for the Penguins dataset.

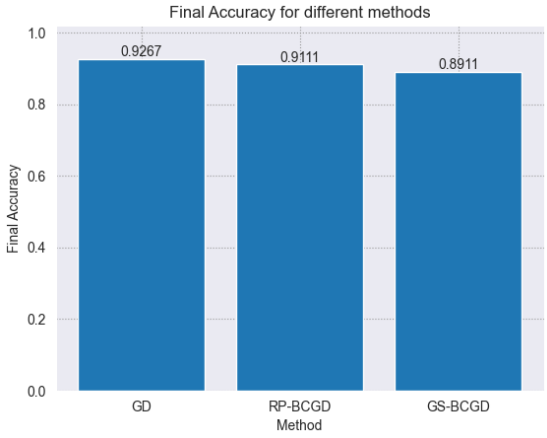

Figure 5 –Barplot for comparing the accuracies for each method applied to the Diabetes dataset subsample.

Concerning the results in Table 1, it must be noticed that, although great care was taken in using fixed seeds for each step involving random choices of values or random generations of data (in order to ensure the maximum consistency and the highest possible level of reproducibility), loss and accuracy results may vary slightly in different runs, due to the numerical estimation methods of the Hessian matrix used by NumPy, which may lead to slightly different values of the Lipschitz constant between two distinct runs and, therefore, to models with slightly different accuracy results.

Dataset	GD	RP-BCGD	GS_BCGD
Synthetic	0.9306	1.000	0.8988
Penguins	0.5567	0.7100	0.6667
Diabete	0.9267	0.9111	0.8911

Table 1 – Final accuracies results for all the datasets, reported in a heatmap format with lighter colours for higher values.