Optimization for Data Science

Homework

A.y. 2024/25

Elia Carta	2156440
Jianan E	2144167
Gianfranco Mauro	2160477
Davide Volpi	2140728

1 Introduction

Labeling information has become a major challenge in modern real-world scenarios. When Supervised Learning algorithms cannot be used due to the scarcity or cost of labeled data, Semi-Supervised Learning (SSL) techniques offer a valuable alternative. SSL attempts to bridge this gap by exploiting the structure inherent in the large amount of unlabeled data, along with the limited set of labeled data, often achieving better results compared to purely Unsupervised Learning methods.
The core idea of many SSL methods is to generalize the labels from the labeled samples to the unlabeled samples through appropriate smoothing operators, enforcing the assumption that nearby points or points within the same structure (e.g., cluster) are likely to have the same label. The graph Laplacian has proven to be highly effective for this purpose.

The graph Laplacian operator, defined on a graph with n nodes (data points), is given by:

$$L = D - W$$

where $W \in \mathbb{R}^{n \times n}$ is the weighted adjacency matrix (with W_{ij} representing the similarity between point i and point j), and D is the diagonal degree matrix with $D_{ii} = \sum_{j=1}^{n} W_{ij}$. The graph Laplacian is a fundamental smoothing operator for solving various learning tasks on graphs. Key properties include:

- It is symmetric (if W is symmetric).
- It is positive semi-definite, i.e., $y^\top Ly \geq 0$ for all $y \in \mathbb{R}^n$.
- For any vector $y \in \mathbb{R}^n$ representing values (e.g., labels) associated with the nodes:

$$(Ly)_i = \sum_{j=1}^{n} W_{ij}(y_i - y_j) \tag{1}$$

$$y^\top Ly = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} W_{ij}(y_i - y_j)^2 \tag{2}$$

Equation (2) shows that the quadratic form $y^\top Ly$ measures the smoothness of the function y over the graph; it penalizes differences in y values between strongly connected nodes (high W_{ij}) [1].

1.1 Similarity measure

In order to quantify the similarity between data points, which is essential for constructing the weight matrix W, we use a measure based on the distance between points. A common choice is the Squared Euclidean Distance between two points $x_i, x_j \in \mathbb{R}^d$:

$$d(x_i, x_j)^2 = \|x_i - x_j\|_2^2 = \sum_{k=1}^{d} (x_{ik} - x_{jk})^2$$

However, the weights W_{ij} should represent similarity, meaning higher values for closer points. A way to achieve this is by using a Gaussian kernel, also known as the Radial Basis Function (RBF) kernel:

$$W_{ij} = \exp\left(-\frac{\|x_i - x_j\|_2^2}{2\sigma^2}\right)$$

where $\sigma > 0$ is the bandwidth parameter, controlling the rate at which similarity decays with distance. A larger σ means similarities decay more slowly, considering points farther away as still relatively similar. The choice of σ can significantly impact performance. In our implementation, we set $W_{ii} = 0$ for all i.

2 Datasets

2.1 Synthetic dataset

For the synthetic dataset, we generated a set of $n = 1000$ points in $\mathbb{R}^2$, belonging to two distinct classes. We assigned initial labels -1 and 1 to these points. To simulate a semi-supervised scenario, then randomly selected 10% of the points ($\ell = 100$) to keep their labels, removing the labels from the remaining 90% ($u = 900$).
The goal is to infer the labels of the unlabeled points using the known labels and the geometric structure of the entire dataset.

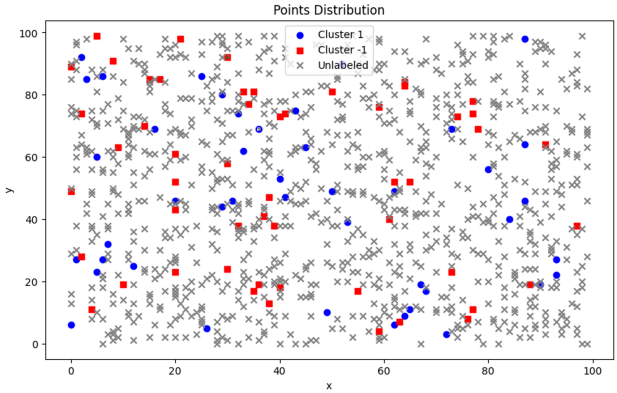

Figure 1: Synthetic dataset plot showing labeled (colored) and unlabeled (grey) points.

2.2 Real dataset

To test our implementations with real-world data, we used a publicly available Kaggle dataset of Uber trips in New York City. We preprocessed it by removing irrelevant attributes, keeping only the latitude and longitude of the pick-up location as features. The "Category" attribute (e.g., 'Business' vs. 'Personal') serves as the class label. We sampled approximately 1,000 data points from the original dataset.
Similar to the synthetic dataset, we designated the 'Business' category as label A (value $+1$) and 'Personal' as label B (value -1). We then applied the same semi-supervised setup, retaining labels for only 10% of the data points and treating the remaining 90% as unlabeled.

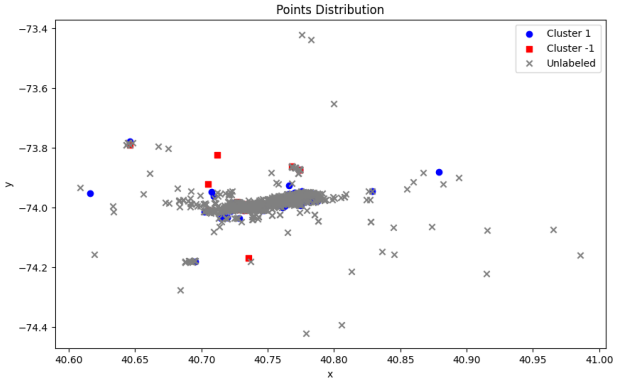

Figure 2: Uber dataset plot showing labeled (colored based on trip category) and unlabeled pickup locations.

3 Problem

We consider the following problem

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

where f is convex and smooth, with

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

where

- y^j is the label assigned to the j-th unlabeled point
- $\bar{y}^i$ is the labeled assigned to the i-th labeled point
- w_{ij} is the weight (similarity) of the i-th labeled point related to the j-th unlabeled point.
- $\bar{w}_{ij}$ is the weight of the i-th unlabeled point related to the j-th unlabeled point.

The left term is minimized when unlabeled points close to labeled ones get similar labels, while the right term is minimized when similar unlabeled data close to other unlabeled data get similar labels.
Since the function is quadratic, we can rewrite it into the form

$$f(y) = \frac{1}{2} y^\top Q y + b^\top y$$

where

- $y \in \mathbb{R}^u$ is the vector of unknown labels of the u unlabeled points.
- $Q \in \mathbb{R}^{u \times u}$ is the symmetric, positive and semi-definite matrix containing the similarity between unlabeled points and their connection to labeled points.
- $b \in \mathbb{R}^u$ is the vector representing the linear influence of the known labels on the unlabeled variables.

To do this, we need to compute these terms:

1. The similarity matrix $W_{\ell u}$, a $\ell \times u$ matrix (where ℓ and u are the number of labeled points and unlabeled points respectively) where each element (i, j) is the squared euclidean distance between the two points normalized with an RBF kernel to map each value to a similarity measure with value $\in [0, 1]$.
2. Construct a diagonal matrix $D_W \in \mathbb{R}^{u \times u}$ where the j-th diagonal element is the sum of similarities between the j-th unlabeled point and all the labeled points.
3. Calculate the b term defined as

$$b = -2 \cdot W_{\ell u}^\top \times \bar{y}$$

4. We can now expand the second term as

$$\sum_i (y^i)^2 \cdot \sum_{j \neq i} \bar{w}_{ij} - \sum_{i, j \neq i} \bar{w}_{ij} \cdot y^i \cdot y^j$$

   which, thanks to the Laplacian operator defined in Section 1, can be rewritten as

$$y^\top \times (D_{smooth} - W_{smooth}) \times y$$

5. The last ingredient we're missing is $\bar{W}_{uu}$ (or W_{smooth}), a $u \times u$ matrix defined almost like $W_{\ell u}$ with the only difference that each computation is made between each unlabeled point.
6. Now that we have every element we need, we can the full Q matrix and the $f(y)$ becomes

$$f(y) = y^\top D_W y + b^\top y + y^\top (D_{smooth} - W_{smooth}) y$$

$$f(y) = y^\top (D_W + D_{smooth} - W_{smooth}) y + b^\top y$$

   Since the initial formula was

$$f(y) = \frac{1}{2} y^\top Q y + b^\top y \quad \rightarrow \quad Q = 2(D_W + D_{smooth} - W_{smooth})$$

7. A small regularization term αI (with $\alpha = 10^{-6}$ in our case) is added to Q for numerical stability, making it strictly positive definite: $Q \leftarrow Q + \alpha I$.

Now that we have our Q matrix, we can test which minimization algorithm is the best for our SSL task.

4 Algorithms

In this section, we briefly discuss the algorithms implemented and highlight their performance on the SSL problem. The main idea of the gradient-based methods is using a local approximation of the objective function at the current iteration y_k to compute a search direction d_k and a stepsize α_k. All these algorithms aim at finding the minimum of a smooth and convex function $f(x)$, where $x \in \mathbb{R}^n$. Since the function we are dealing with is strictly convex (due to the positive α term added to our Q matrix) and smooth, the methods converge toward a global minimum.
In our implementation, we deal with:

- Three main algorithms: Gradient Descent (GD), Block Coordinate Gradient Descent with Gauss-Southwell rule (BCGD-GS) and Coordinate Descent (CD).
- 1-dimensional blocks, since the data is made up of 2-dimensional points.
- Three step-sizes.
- Three priority-strategies.

Step-Sizes Strategies

These strategies determine the step-size α_k used in the update rule:

- **Lipschitz constant stepsize (L)**: sets the step-size $\alpha_k = 1/L$, where L is the Lipschitz constant of the gradient $\nabla f(y)$. For our quadratic function $f(y) = \frac{1}{2}y^\top Qy + b^\top y$, the gradient is $\nabla f(y) = Qy + b$, and its Lipschitz constant L is the largest eigenvalue of the Hessian matrix Q, which corresponds to the spectral norm $||Q||_2$.
- **Block Lipschitz constant stepsize (block_L):** use a step-size related to the properties of a specific coordinate. The standard coordinate Lipschitz L_i for coordinate i is $Q[i,i]$. L_i is calculated as the $L2$ norm of the i-th column of Q and the stepsize becomes $\alpha_k = 1/L_i$.
- **Exact Linesearch stepsize (exact):** for a quadratic function, the minimum along a single coordinate direction i can be founded exactly. Minimizing $f(y_k - \alpha e_i)$ with respect to α leads to the optimal step $\alpha^* = \nabla f(y_k)_i / Q[i,i]$. The update rule performs this exact minimization if we set the step-size equal to $1/Q[i,i]$.

Priority Strategies

These strategies are only used within the BCGD-GS algorithm to select which coordinate i to update at each iteration k, based in the GS rule principle of choosing the coordinate that offers the most progress:

- **abs:** selects the coordinate i that maximizes the absolute value of the partial derivative $i = argmax_j |\nabla f(y_k)_j|$. Picks the coordinate along which the function's slope is currently steepest.
- **abs_L:** selects the coordinate i that maximizes $|\nabla f(y_k)_j|/L_j$, where L_j is calculated as the $L2$ norm of the j-th column of Q, so it scales the gradient component by the column norm.
- **max_improvement:** selects the coordinate i that maximizes the predicted decrease in the objective function if the exact stepsize is used for that coordinate. This rule directly targets the coordinate that will return the largest immediate reduction in f(y) in the current iteration, assuming an exact line search along the coordinate axis.

4.1 Gradient Descent

This method uses, at each iteration, the full negative gradient $d_k = -\nabla f(y_k) = -(Qy_k + b)$ as the search direction. It approximates the function decrease using a first-order Taylor expansion. In our implementation, different step-sizes are used, as specified in the section 4.
We pick a random starting point, then at each step we update the values following the rule:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

4.2 BCGD with Gauss-Southwell Updates

The idea behind BCGD methods is to divide the variables into b blocks of dimension $n_i, i = 1, \dots b$ such that $n = n_1 + \cdots + n_b$. At each iteration k, the algorithm select a block i_k according to some specific condition and update only the variables within that block, typically by taking a gradient step computed only with respect to the variables in the selected block.
The specific condition used for selecting the coordinate i_k at iteration k is the Gauss-Southwell (GS) rule. This rule aims to select in a greedy way the coordinate that offers the most progress towards the minimum. It chooses the coordinate i_k that maximizes a priority function applied to the corresponding component:

$$i_k = \arg \max_{j \in 1,\dots,n} P(\nabla_j F(y_k))$$

The search direction is $d_k = -e_{i_k}$, and the update is

$$y_{k+1}[i_k] = y_k[i_k] - \alpha_k \nabla_{i_k} F(y_k)$$

with other coordinates remaining unchanged.
To efficiently implement the GS rule, a max-heap data structure is used to maintain and update the coordinate priorities.

4.3 Coordinate Minimization

Coordinate Minimization, also called Coordinate Descent (CD) is a method for solving optimization problems by repeating steps.
Instead of calculating the full gradient $\nabla f(x)$, which can be slow when n is large, CD updates one random coordinate (variable) i at each iteration. The search direction d_k is $-e_i$, where e_i is the i-th standard basis vector. In our implementation, different step-sizes are used, as specified in the section 4. In each step, it picks a coordinate $i \in \{1, ..., n\}$ and minimizes $f(x^{(i)}, x^{-i})$, where x^{-i} means all other coordinates are kept the same. The new value

$$s_k^{(i)} = \arg \min_{x^{(i)} \in \mathbb{R}} f(x^{(i)}, x^{-i}),$$

we obtain is then used to update the i-th coordinate.
The process continues by randomly sampling points from the dataset without replacement. Once all points have been sampled, the process restarts with a shuffled dataset and repeats until the stop condition is met or the maximum number of iterations is reached. At the best solution x^*, the partial derivative with respect to each coordinate should be zero, meaning

$$\nabla_i f(x^*) = 0 \quad \text{for all} \quad i.$$

5 Results

We evaluate the computational performance of the implemented algorithms on both the synthetic and real datasets. Since we are dealing with a strictly convex function, the unique optimal solution y^* can be found by solving $Qy^* = -b$, and the optimal value is $f(y^*) = \frac{1}{2}(y^*)^\top Qy^* + b^\top y^*$. Performance is measured by convergence speed, visualized by plotting the distance to this optimal value, $|f(y_k) - f(y^*)|$, against both the number of iterations k and the computation time. The stopping criterion for the algorithms was either reaching a maximum of $N = 50,000$ iterations or reaching the tolerance $\epsilon = 10^{-6}$ on the relevant measure (gradient norm $\|\nabla f(y_k)\|_2$ for GD/CD, or the maximum priority value $\max_j \text{priority}(j)$ for BCGD-GS). All algorithms were run starting from the same random initial point y_0, and we used a fixed random seed for reproducibility.

5.1 Synthetic data

5.1.1 Iterations View

- Gradient Descent shows significantly faster initial convergence and eventually reaches the lowest final distance to the optimal solution (around 10^{-3}, surpassing even the best BCGD-GS variant (`exact + max_improvement`).
- BCGD-GS performs better than Coordinate descent, making the `exact + max_improvement` variant the best among the coordinate methods.
- Coordinate Descent shows the slowest convergence across all algorithms, ending at the highest error value.

On the synthetic data, the results align with typical expectations, since coordinate minimization methods are optimized for sparse datasets.

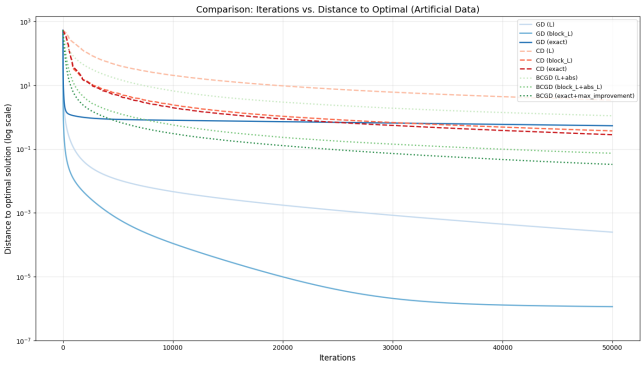

Figure 3: Synthetic Data: Objective function value $f(y_k)$ vs. iteration number for GD, BCGD-GS, and CD.

5.1.2 Time View

- The time plot underscores the speed difference for reaching moderate accuracy. The BCGD-GS methods achieve substantial error reduction almost instantaneously (within 1-2 seconds). CD is slower but still much faster than GD initially.
- GD requires significantly more computation time (around 90 seconds) to reach its final, lowest error value shown in the iteration plot.

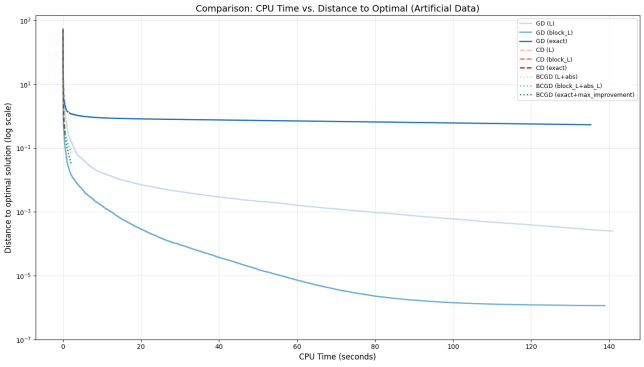

Figure 4: Synthetic Data: Objective function value $f(y_k)$ vs. CPU time (seconds) for GD, BCGD-GS, and CD.

5.2 Real data

On the real (Uber) dataset, the results for GD might appear surprisingly different from the synthetic case.

5.2.1 Iterations View

- GD converges extremely rapidly, reaching a distance below 10^{-10} within the first few iterations.
- CD and all BCGD-GS variants converge much more slowly per iteration than GD in this specific experiment, taking tens of thousands of iterations to reach worse distances within the 50000 iteration limit.

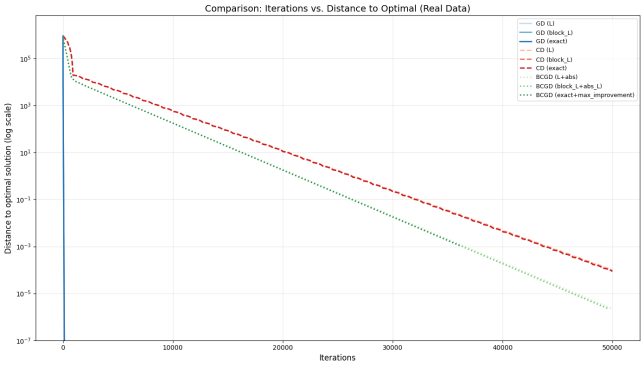

Figure 5: Uber Data: Objective function value $f(y_k)$ vs. iteration number for GD, BCGD-GS, and CD.

5.2.2 Time View

- Consistent with the iteration plot, GD converges in negligible CPU time (basically instantaneously on the scale shown).
- CD takes roughly 1-2 seconds to reach its final state, while BCGD-GS methods take significantly longer, up to around 35 seconds, to reach their final accuracy levels shown.

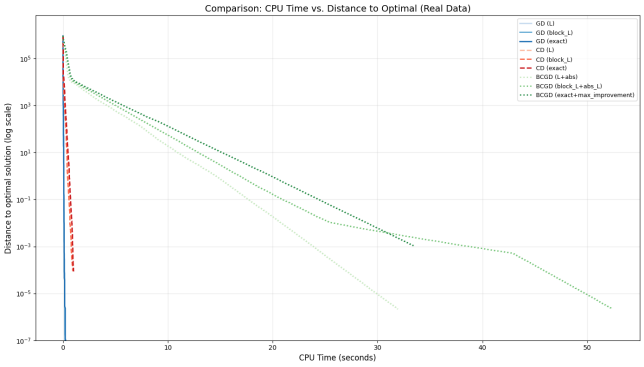

Figure 6: Uber Data: Objective function value $f(y_k)$ vs. CPU time (seconds) for GD, BCGD-GS, and CD.

6 Discussions and Conclusions

Given that we tested our code with different seeds and that each one of them allowed GD to satisfy the tolerance condition after significantly fewer steps than coordinate based methods, we can infer that these methods are indeed better optimized for sparse datasets and initialization does not interfere with our results.
The experiments highlight the importance of testing algorithms on multiple datasets, as performance can be instance-dependent.
On the synthetic dataset, the results showed a trade-off: coordinate methods (especially BCGD-GS) were significantly faster in terms of CPU time to reach moderate accuracy, but standard Gradient Descent eventually achieved a slightly better final accuracy when run for a fixed large number of iterations. This aligns with theoretic results, stating that coordinate methods have cheaper iterations but GD might make more globally informed progress per step, although slowly.
On the other hand, the results on the real dataset demonstrated a seemingly atypical outcome where GD converged extremely quickly and significantly outperformed all coordinate methods in both iterations and time for the specific random seed used. This is attributed to the distribution of points in the dataset, which are really close to each other and make GD the better option in this specific case. Further investigation, possibly with different problem scaling, would be beneficial.