Università degli Studi di Padova | Dipartimento Matematica

Master's Degree in Data Science

Academic Year 2024/2025

Optimization for Data Science

Homework
Semi-supervised Learning

Ilaria Boschetto	**Federico Clerici**	**Raffaele D'Agostino**	**Gabriele Villa**
ID: 2140725	ID: 2140726	ID: 2159093	ID: 2157873

Contents

1 Introduction

In this homework, we employed various optimization algorithms to perform binary classification using a semi-supervised learning approach. The algorithms we used are **Gradient Descent**, **Block Coordinate Gradient Descent (BCGD)** with **Gauss-Southwell (GS) rule** and **Coordinate Minimization**. We first applied the algorithms to a synthetic dataset, and subsequently evaluated its performance on a real-world dataset.

2 Presentation of the problem

2.1 Loss function

The aim of the problem is to minimize the following loss function:

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

We now describe the meaning of each individual term:

- y: the vector of predicted labels for all u unlabeled data points.
- l: the number of labeled data points;
- u: the number of unlabeled data points;
- w_{ij}: a weight expressing the similarity between a labeled point i and an unlabeled point j;
- $\bar{y}^i$: the known label of the i-th labeled data point;
- $\bar{w}_{ij}$: a weight expressing the similarity between points i and j (both unlabeled);
- $\frac{1}{2}$: a scaling factor to avoid double-counting in the symmetric regularization term;

2.2 Methodology

The workflow we followed consists of the following steps:

1. Compute all necessary quantities, including the **similarity matrix**, **accuracy score**, **stepsize**, **gradient**, and **Hessian**.
2. Generate **artificial data** in 2 dimensions suitable for the semi-supervised learning setting.
3. Apply the optimization algorithms stated before to the synthetic data in order to solve the minimization problem, and analyze their performances in terms of **loss vs. iterations**, **accuracy vs. iterations** and **accuracy vs. CPU time** (effective CPU time measured with `process_time`).
4. Apply the algorithms to a **real-world dataset** and evaluate their performance.

3 Implementation of the problem

3.1 Starting point

We chose to initialize the label vector **randomly**: this choice allows the optimization algorithm to start from an unbiased configuration, avoiding any prior assumptions about the true labels.

3.2 Gradient

The gradient we were given is the following:

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

3.3 Hessian

We calculated the Hessian matrix as follows:

$$2 \cdot \begin{bmatrix} \left(\sum_{i=0}^{l} w_{i1}\right) + \left(\sum_{i=0}^{u} \bar{w}_{i1}\right) - \bar{w}_{11} & -\bar{w}_{12} & \cdots & -\bar{w}_{1n} \\ -\bar{w}_{21} & \left(\sum_{i=0}^{l} w_{i2}\right) + \left(\sum_{i=0}^{u} \bar{w}_{i2}\right) - \bar{w}_{22} & \cdots & -\bar{w}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\bar{w}_{n1} & -\bar{w}_{n2} & \cdots & \left(\sum_{i=0}^{l} w_{in}\right) + \left(\sum_{i=0}^{u} \bar{w}_{in}\right) - \bar{w}_{nn} \end{bmatrix}$$

This matrix was used both in the gradient update step of the block coordinate gradient descent and in the computation of the stepsize.

3.4 Accuracy Metric

For the evaluation of the algorithms, it is necessary to define an appropriate accuracy measure. After running the algorithms and obtaining the predicted label vector, we discretized the continuous output using the **sign function**, which assigns a label of $+1$ if the output is positive and -1 otherwise.

Subsequently, we computed the **accuracy score** using the standard definition:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{Total number of samples}}$$

where TP (True Positives) and TN (True Negatives) represent the number of correctly classified positive and negative examples, respectively.

3.5 Stepsize

The choice of stepsize is crucial for ensuring the convergence of the optimization algorithm. After several empirical trials, we opted to use a fixed stepsize defined as $\alpha = \frac{1}{L}$, where L is the **Lipschitz constant** of the gradient, which in our case corresponds to the **largest eigenvalue of the Hessian matrix**.

This choice of stepsize guarantees convergence for gradient-based methods when the objective function is convex. In our setting, **the objective function is indeed convex**,

as it is the sum of two quadratic functions—both convex by nature— and the sum of convex functions is itself convex.

3.6 Stopping Criterion

We adopted a stopping criterion based on the relative change of the objective function. Specifically, the algorithm stops when:

$$\frac{\left|f\left(y^{(k+1)}\right) - f\left(y^{(k)}\right)\right|}{\left|f\left(y^{(k)}\right)\right|} < \varepsilon \tag{1}$$

This ensures that the iterations stop once the objective function value stabilizes within a predefined tolerance. We applied the same stopping tolerance to both GD algorithm and the Coordinate Minimization methods. We set a different tolerance for the BCGD method to better capture the more gradual convergence behavior inherent to coordinate-wise updates.

3.7 Similarity matrices

We chose to construct the similarity matrices W_{ij} and $\bar{W}_{ij}$ using the **Gaussian kernel**, defined as follows:

$$w_{ij} = e^{\left(-k\|x_i - x_j\|^2\right)}$$

where x_i and x_j are feature vectors of data points i and j, and $k > 0$ is a scaling parameter that controls the width of the kernel.

The Gaussian kernel offers several advantages:

- **Robustness to High Dimensions**: while the *curse of dimensionality* can affect many distance-based methods, the Gaussian kernel can mitigate this issue by effectively reducing the influence of distant or irrelevant features.
- **Flexibility**: the parameter k allows tuning of the sensitivity to distances, enabling better control over the neighborhood scale.

We selected the value of the parameter k by analyzing the **histogram of pairwise distances** between data points. The goal was to choose a value that does not compress all distances into a narrow range, nor make them nearly uniform across all pairs. Ideally, k should be chosen so that the similarity measure w_{ij} has a high magnitude when two points are close, and significantly lower when they are far apart.

4 Algorithms

4.1 Gradient Descent

Gradient Descent updates the parameters at each iteration according to the rule

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

where $\alpha > 0$ is the stepsize and $\nabla f(x_k)$ is the gradient of the objective function at iteration k. The algorithm moves in the direction of the negative gradient, which corresponds to the steepest descent direction.

4.2 Block Coordinate Gradient Descent with Gauss-Southwell rule

Block coordinate gradient descent is an optimization algorithm that updates subsets (blocks) of variables at each iteration, rather than the entire variable vector. The **Gauss-Southwell** rule is a greedy selection strategy that, at every step, chooses the block corresponding to the largest component (in absolute value) of the gradient. As required by the assignment, we selected **blocks of size 1** for this algorithm. In this case, the gradient was updated using a first-order Taylor approximation, where H denotes the **Hessian** matrix.

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

This formula, used for **fast gradient updates**, is significantly more efficient because it avoids recomputing the full gradient at each iteration. Instead, it updates the gradient at iteration $k+1$ incrementally, based on the gradient at iteration k, by modifying only the component associated with the selected coordinate $i(k)$.

To implement BCGD with the Gauss–Southwell rule, we designed a custom **heap-based class** to efficiently select the desired coordinate at each iteration. This class maintains a **max-priority queue**, where the priority of each coordinate is updated at every iteration based on the current value of the gradient. In this way, the algorithm always selects the coordinate with the highest potential for reducing the objective function.

4.3 Coordinate Minimization Method

Coordinate Minimization works by fixing all variables except one and minimizing the objective function with respect to that single coordinate.

$$y^j = \frac{\sum_{i=1}^{\ell} w_{ij} \bar{y}^i + \sum_{\substack{i=1 \\ i \neq j}}^{u} \bar{w}_{ij} y^i}{\sum_{i=1}^{\ell} w_{ij} + \sum_{\substack{i=1 \\ i \neq j}}^{u} \bar{w}_{ij}}$$

We implemented this method following the **Jacobi scheme**, which allows us to update each coordinate independently.

5 Semisupervised learning on a fictional dataset

5.1 Artificial Dataset

To verify the correct functioning of our algorithms, we initially generated a synthetic dataset (Figure 1) consisting of **1,000 points** in $\mathbb{R}^2$ distributed across **two distinct clusters**.

Out of these 1,000 points, we retained the labels for 100 of them, which we considered as the labeled subset. The remaining 900 points were treated as unlabeled by discarding their labels during the training phase. However, we retained access to the original labels of these points in order to compute the final accuracy and evaluate the performance of the algorithms.

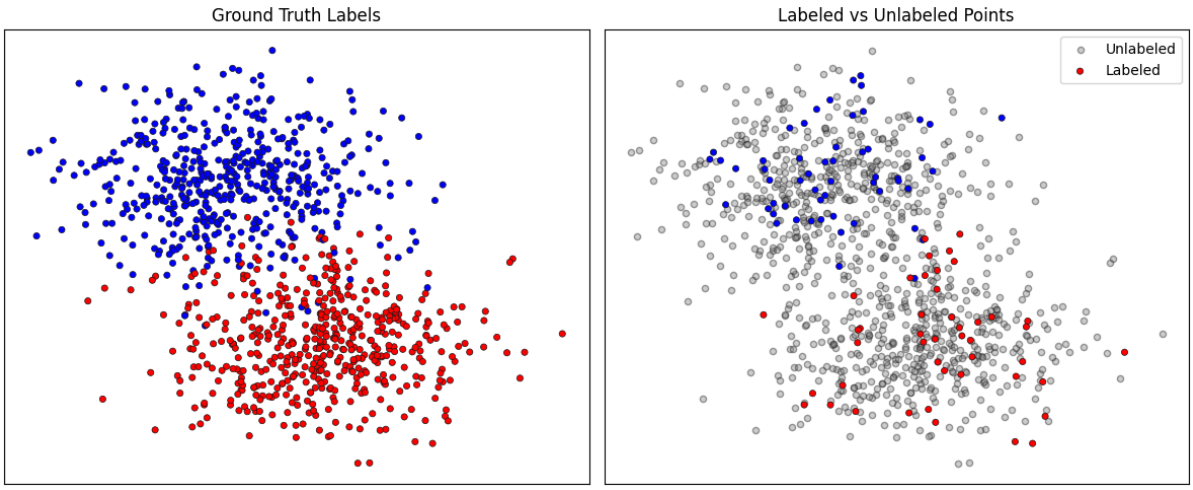

Figure 1: Visualization of the points used to test semisupervised learning algorithms on a fictional dataset. On the left, the plot of the points with their ground truth labels. On the right, the initialization points used to test our algorithms.

5.2 Performance on the synthetic dataset

The artificial dataset used to test the performance of our three semi-supervised learning algorithms is described in the section above (Section 5.1).

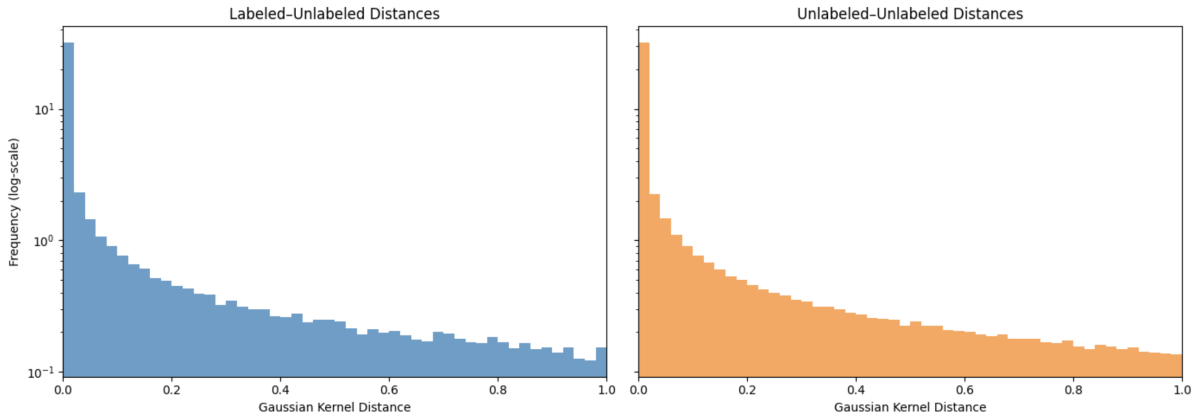

Figure 2: Histogram of the distribution of the distances between points, respectively for labeled-unlabeled points and unlabeled-unlabeled points. The y axis is in logarithmic scale, and both histograms are normalized to unit area.

We analyzed many distribution plots of the distance measures between points, as in Figure 2, and decided to set the Gaussian Kernel parameter used to compute the weight matrices $k = 1$. Since the two classes in our synthetic dataset are balanced, we chose accuracy as a performance metric. We used the activation function $sign(x)$ to determine the label ± 1 after running the algorithms.

For this dataset, we set the stopping tolerance $\epsilon = 10^{-6}$ for both the Gradient Descent and Coordinate Minimization methods, while a more stringent value of $\epsilon = 10^{-12}$ was chosen for the BCGD algorithm to accommodate its slower convergence behavior. We can see in Figure 4 that the convergence of the BCGD method was significantly slower in terms of iterations. In fact, BCGD required 78754 iterations to converge, compared to only 109

for Gradient Descent and 52 for Coordinate Minimization. This slower convergence is due to the fact that BCGD updates only one coordinate of the gradient at each iteration. We extracted the *accuracy* of our algorithms as the value computed after the last iteration, giving as a result:

Accuracy (GD):	95.22%
Accuracy (CM):	95.22%
Accuracy (BCGD):	95.22%

Lastly, we analyzed the CPU time of the three algorithms.

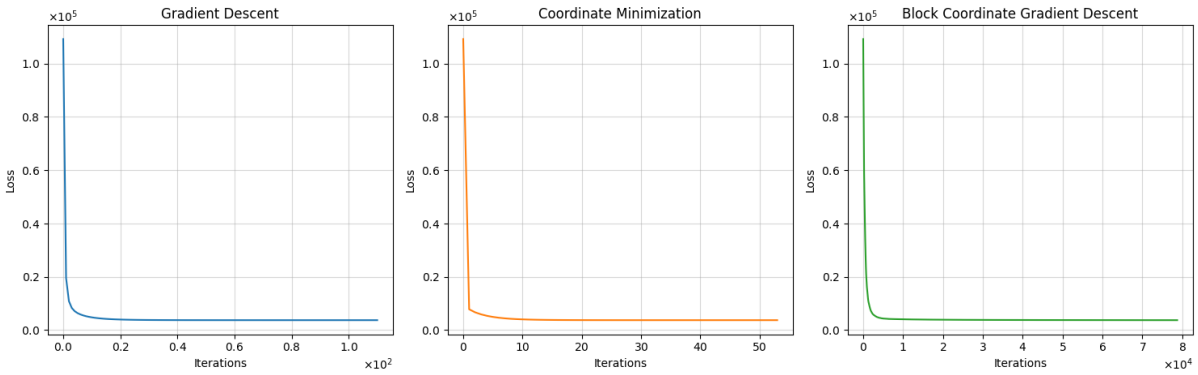

Figure 3: Plot of the loss functions of the three algorithms vs the number of iterations.

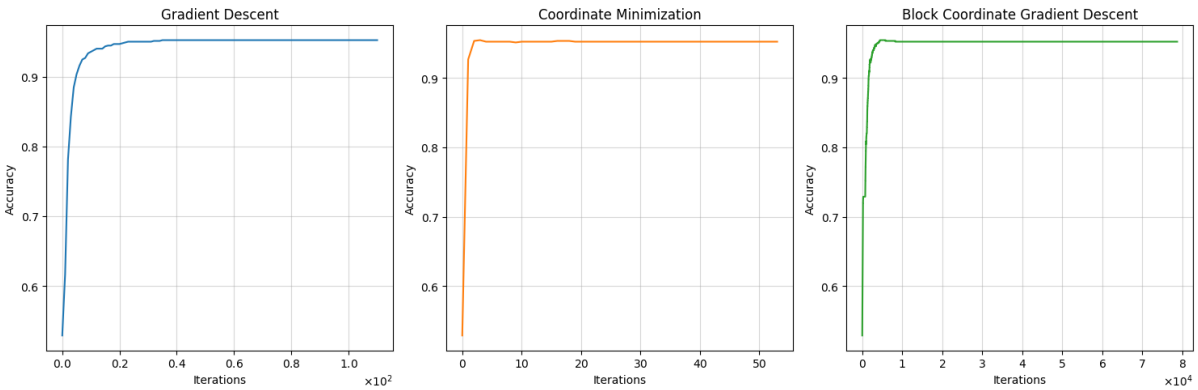

Figure 4: Plots of the accuracy vs iterations for the three algorithms applied to the synthetic dataset.

According to the theory, the Gradient Descent algorithm is expensive in terms of the cost per iteration, resulting in the longest CPU time overall. The Block Coordinate Gradient Descent (BCGD) follows, primarily due to the larger number of iterations required to converge, while still involving gradient computations. Lastly, the Coordinate Minimization method has been shown to be the fastest, as it relies on closed-form updates for each coordinate and does not require explicit gradient computations at runtime.

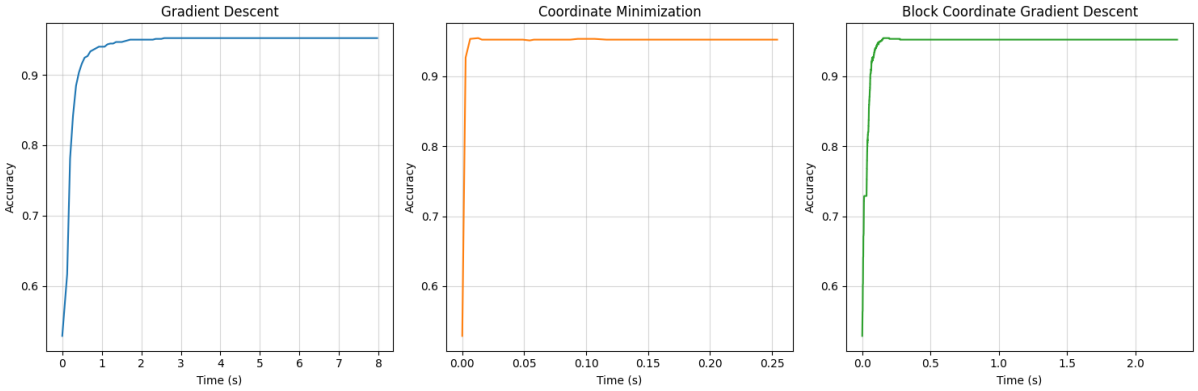

Figure 5: Plot of the accuracy vs CPU time for the three algorithms applied to the synthetic dataset.

In Figure 6, it is possible to visualize the predictions made by our algorithms before applying the activation function $sign(x)$. It is clear that our algorithms cluster the points well, making some mistakes only near the boundary between one class and the other.

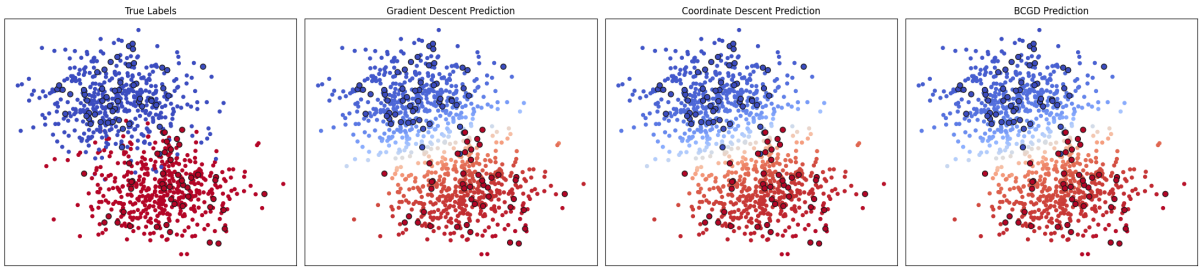

Figure 6: Plot of the algorithms' predictions before applying the activation function $sign(x)$ to assign labels to the data points. The opacity of each point reflects the magnitude of its prediction (ranging from -1 to 1): the more opaque the point, the stronger the prediction and the higher the confidence in the clustering.

6 Semisupervised learning on a real dataset

6.1 Real Dataset

As the final step of our work, we applied the methodologies previously tested on fictional data to a real-world dataset. We selected the **Banknote Authentication**[1] dataset, which contains data extracted from images of both genuine and forged banknote-like specimens. This dataset contains **1,372** instances and **4** continuous features, each representing a specific statistical property of the Wavelet Transformed images of the banknotes: variance, skewness, kurtosis, and entropy.

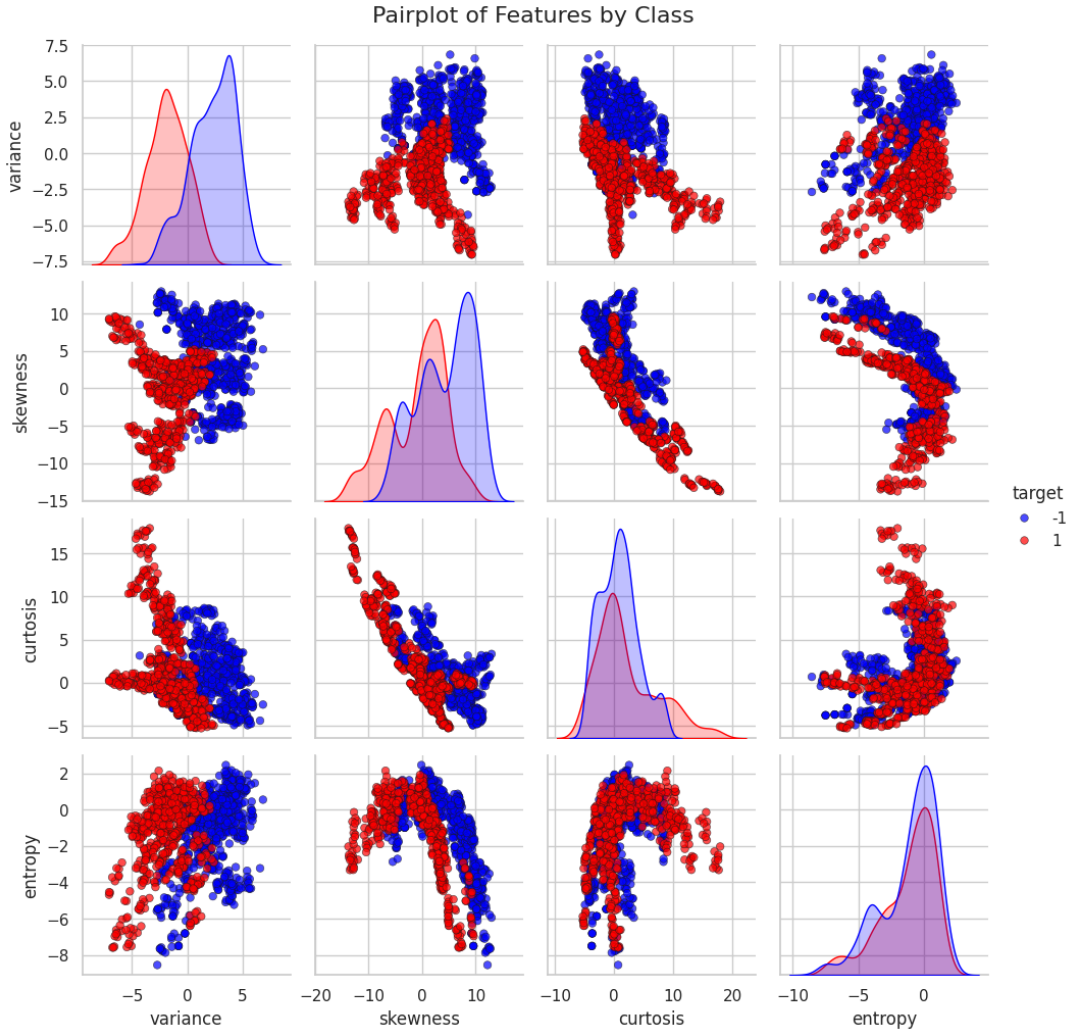

Figure 7: Pairplot visualization of all four features, showing distributions and pairwise relationships.

[1] https://archive.ics.uci.edu/dataset/267/banknote+authentication

We applied Principal Component Analysis (PCA) to identify the features with the greatest discriminative power. Based on the results, all visualizations from this point onward will be shown in the two-dimensional space defined by the most significant features: variance and skewness (Figure 8). However, it is important to note that all algorithms are still executed using the full set of four original features.

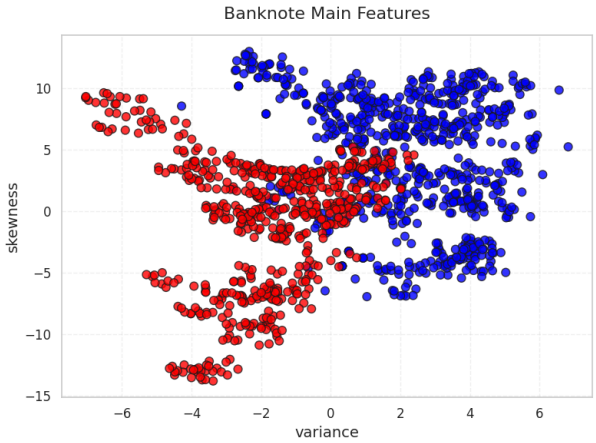

Figure 8: Visualization of the banknote dataset along its two principal features: variance and skewness. Red points are genuine banknotes, blue points are the counterfeit ones.

6.2 Performance on the real dataset

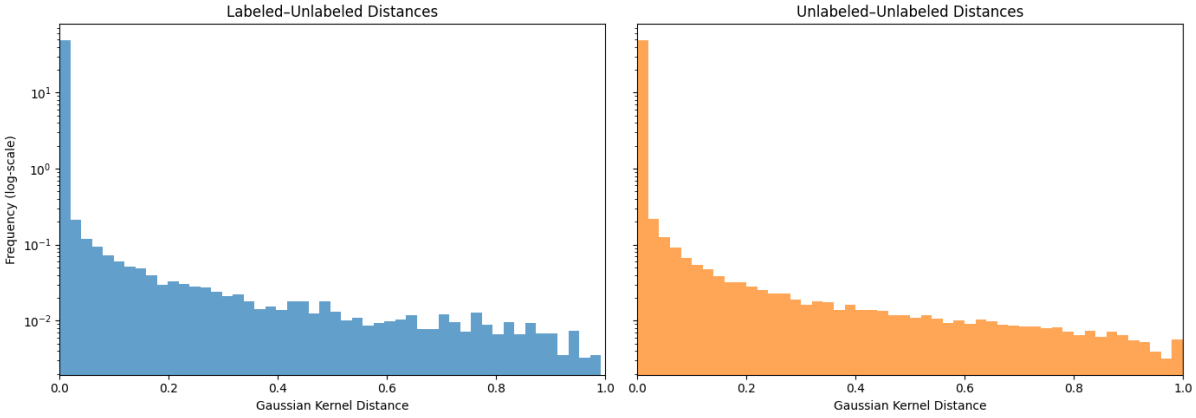

Figure 9: Histogram of the distribution of the distances between points, respectively for labeled-unlabeled points and unlabeled-unlabeled points. The y axis is in logarithmic scale and both histograms are normalized to unit area.

After analyzing the histogram of the distribution of the distances between the points, we chose to set the Gaussian Kernel parameter $k = 1$.

Regarding the stopping criteria, we adopted the same criterion we used on the artificial dataset (Equation 1). In particular, we set the tolerance to 10^{-6} for both Gradient Descent and Coordinate Minimization, whereas for BCGD we used a more stringent threshold of 10^{-14}.

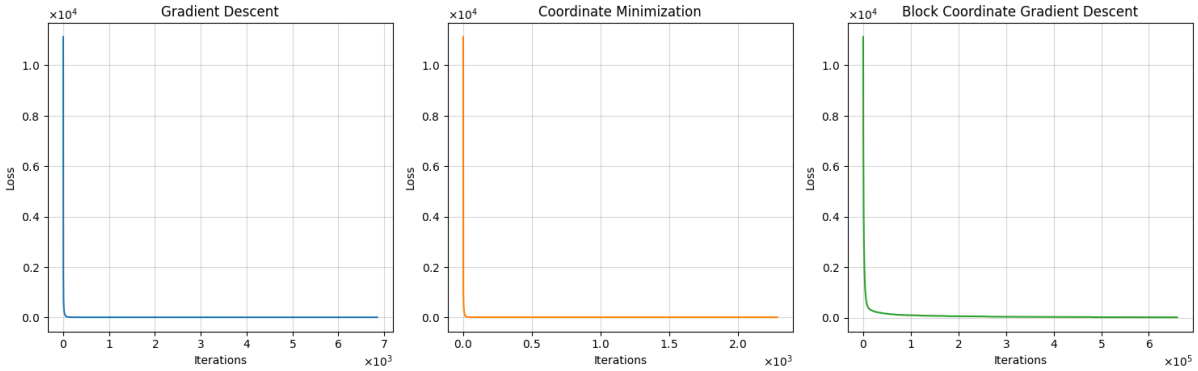

Figure 10: Plot of the loss functions of the three algorithms vs the number of iterations.

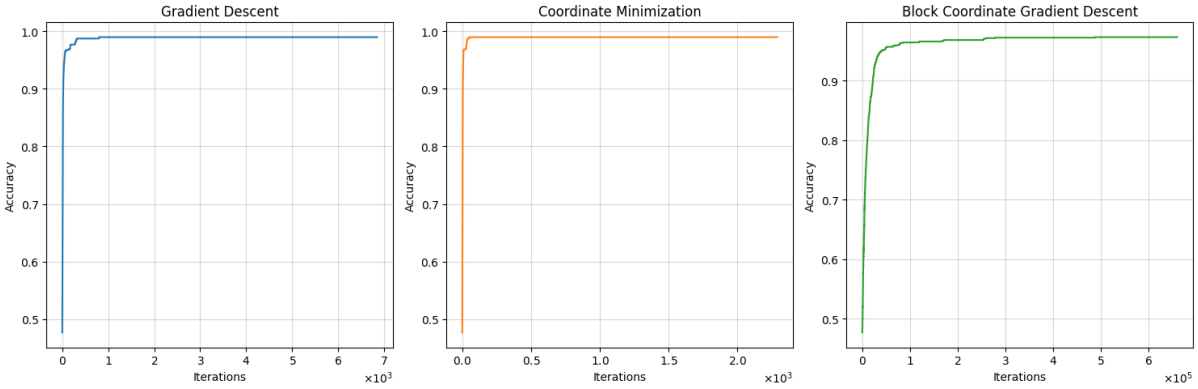

Figure 11: Plots of the accuracy vs iterations for the three algorithms applied to the real dataset.

By examining Figures 10 and 11, we observe that BCGD requires slightly more iterations to converge. This occurs for the same reasons discussed in the case of the synthetic dataset.

Here as well, we computed the accuracies of the three methods.

Accuracy (GD):	98.95%
Accuracy (CM):	98.95%
Accuracy (BCGD):	97.33%

All three algorithms exhibited a very good performance on the real dataset.

Finally, plots showing accuracy as a function of CPU time are provided.

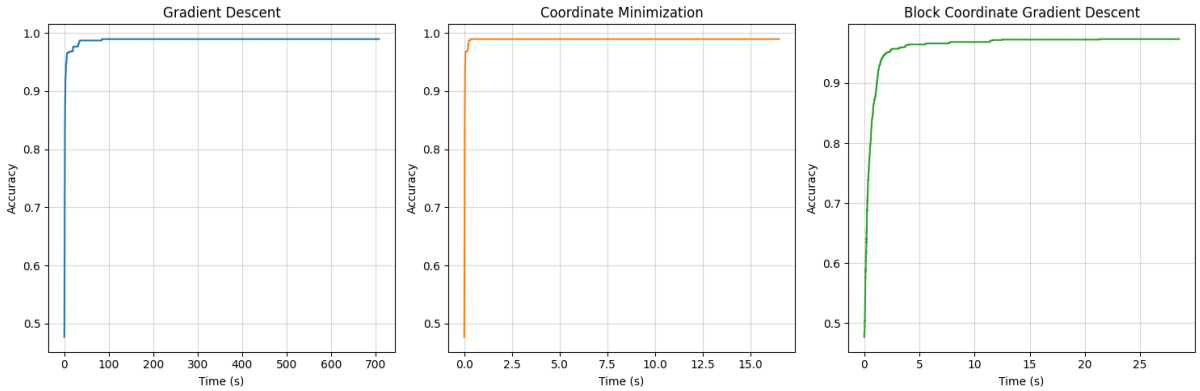

Figure 12: Plot of the accuracy vs CPU time for the three algorithms applied to the real dataset.

As shown in Figure 12, CPU times reflect theoretical assumptions previously recalled, emphasizing the difference between Gradient Descent and BCGD.

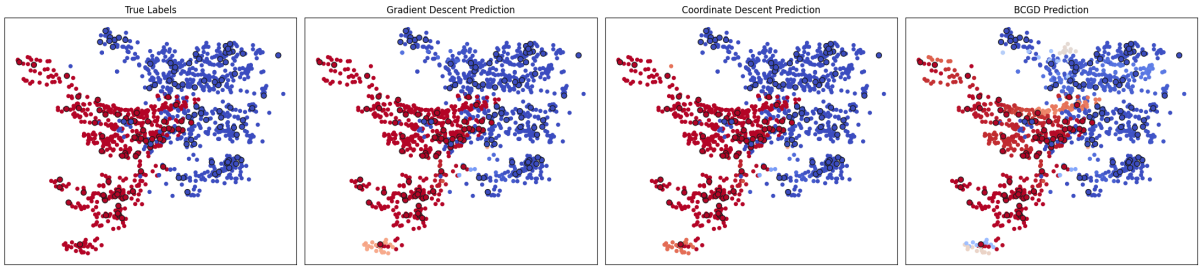

Figure 13: Plot of the algorithms' predictions before applying the activation function $sign(x)$ to assign labels to the data points. The opacity of each point reflects the magnitude of its prediction (ranging from -1 to 1): the more opaque the point, the stronger the prediction and the higher the confidence in the clustering.

7 Conclusion

We obtained excellent performances on both the real and synthetic dataset, as demonstrated by the trends of accuracy and loss over the iterations across all three tested algorithms. The results were better on the real dataset compared to the synthetic one because the real dataset includes additional features that contributed to more effective classification and, consequently, improved overall performance. As expected, it turns out that regular Gradient Descent is not the most efficient choice for large-scale datasets, as computing the full gradient at each iteration can be computationally expensive. However, thanks to the two alternative methods we implemented, we observed that both are significantly less demanding in terms of CPU time, while still achieving the same level of accuracy as standard Gradient Descent.