University of Padova
Department of Mathematics

Semi-Supervised Learning via Gradient Descent, Block Coordinate Gradient Descent & Coordinate Minimization

A Theoretical & Empirical Study of the Implementation of the Solvers

Group Members: Lennart Niels Bredthauer 2143470
Mikhail Isakov 2141897
Max Hans-Jürgen Henry Horn 2143471

Supervisor: Prof. Dr. Francesco Rinaldi

Date: May 2025

Table of Contents

1 Introduction 2

2 Problem Setup 2

3 Dataset & Preprocessing 3

3.1 Synthetic Dataset . . . 3
3.2 Real-World Dataset . . . 3
3.3 Preprocessing . . . 3

4 Graph Construction 3

5 Optimization Formulation 4

6 Optimization Methods 5

6.1 Gradient Descent . . . 5
6.1.1 Method Overview . . . 5
6.1.2 Gradient Derivation . . . 5
6.1.3 Step size Selection . . . 5
6.1.4 Heavy Ball Method . . . 6
6.1.5 Nesterov Accelerated Gradient . . . 6
6.1.6 Convergence Behavior . . . 6
6.2 BCGD with Gauss-Southwell rule . . . 7
6.2.1 Method Overview . . . 7
6.2.2 Step size Selection . . . 7
6.2.3 Convergence Behavior . . . 7
6.3 Coordinate Minimization . . . 7
6.3.1 Method Overview . . . 7
6.3.2 Derivation of the Update Rule . . . 8
6.4 No step size needed . . . 9
6.5 Convergence Behavior . . . 9

7 Experiments & Results 10

7.1 Synthetic Dataset . . . 10
7.2 Real World Data . . . 12
7.2.1 Accuracy over Iterations . . . 12
7.2.2 Accuracy over CPU Time . . . 13
7.2.3 Loss vs. Iterations . . . 13
7.2.4 Loss vs. CPU Time . . . 13
7.2.5 Convergence Behavior . . . 13
7.2.6 Limitations and Value of BCGD . . . 14
7.3 Conclusion . . . 15

1 Introduction

In many real-world applications collecting labeled data is expensive and time consuming. Semi supervised learning (SSL) addresses this problem by exploiting both, the labeled and unlabeled data. In this project, we will focus on the graph-based approach, where each data point is considered a node in a graph and the edges represent the similarity between points, typically measured by the RBF-Kernel. The underlying idea of those methods is, that points which are close by in feature space are likely to share the same label.

In this paper, we will introduce and compare three different methods to solve graph-based semi-supervised learning problems, being: **Gradient Descent**, **Block Coordinate Gradient Descent (BCGD)** using Gauss-Southwell rule and **Coordinate Minimization**. We evaluate these methods on both synthetic and real-world datasets and analyze the performance in terms of classification accuracy and convergence speed.

Gradient Descent methods showed the fastest convergence behavior across all implemented methods for both synthetic and real world datasets.

2 Problem Setup

This section defines the mathematical and conceptual foundations of the project. In SSL problems, we are given a set of points $\{x^i\} \subset \mathbb{R}^d$, where a small subset is labeled. The labeled data is denoted as $\left\{(\bar{x}^i, \bar{y}^i)\right\}_{i=1}^{l}$ and the unlabeled data as $\left\{\bar{x}^j\right\}_{j=1}^{u}$. The goal is to infer the missing labels y^i for the unlabeled points.

The underlying assumption behind graph-based SSL is, that points that are close in the feature space are likely to share the same label. Therefore, the loss function is designed to satisfy the two key conditions:

1. It encourages unlabeled points to take values close to the labels of nearby labeled data.
2. It penalizes dissimilar labels between nearby points.

Hence, we define the loss function as

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

Symbol	**Meaning**
l	Number of labeled data points
u	Number of unlabeled data points
$\bar{y}^i$	True label of point i
y^i , y^j	Label of the i-th and j-th unlabeled point (to be optimized)
w_{ij}	Similarity between labeled point i and unlabeled point j
$\bar{w}_{ij}$	Similarity between two unlabeled points i and j

Table 1: Description of the symbols used in the loss function

The first term encourages consistency between the predicted labels of unlabeled data and the true labels that are nearby, weighted by the similarity matrix w_{ij}. Thus, this will enforce similar pairs to have similar value labels. The second term imposes smoothness constraint across the unlabeled data points. It assumes that the label function should vary smoothly - meaning it has no sudden jumps in predicting similar points. Therefore, it penalizes the difference in label values between similar unlabeled points and helps maintain consistency across the data.

3 Dataset & Preprocessing

3.1 Synthetic Dataset

We first created a two-dimensional synthetic dataset consisting of two clearly separated clusters to evaluate the performance of semi-supervised learning algorithms in a controlled setting. The data was generated as follows:

- Cluster 1 has 500 points sampled from a two-dimensional Gaussian distribution centered at (0,0)
- Cluster 2 also has 500 points sampled from the same Gaussian distribution; however, centered at (5,0)

We then randomly labeled a subset of 20 points in total - 10 from each cluster. The remaining 980 points were treated as unlabeled. The described setup simulates a typical semi-supervised learning problem, due to the fact that only a small subset of the data points is labeled and the remaining points are unlabeled. The low dimensionality and the clear separation of the data of the two clusters make it possible to visualize the label propagation. A scatterplot of labeled and unlabeled data is shown in Section 7.

3.2 Real-World Dataset

For the real-world dataset, we used the digits dataset provided by the sklearn.dataset package. This dataset contains 8x8 gray-scaled handwritten digits from 0 to 9. Each image is flattened into a 64-dimensional vector. Due to our loss function being binary, we converted the multi-class problem into a binary classification task by setting Class 1 as: all images labeled as digit 0 and Class 2 as: all other digits. This results in an imbalanced binary classification task, where positive class samples are rare.

3.3 Preprocessing

To enhance the performance of similarty-based methods on the dataset, we performed the following preprocessing steps:

- Standardization:
  We standardized each feature to have zero mean and unit variance. Doing so, ensures that each feature contributes equally to the Euclidean distance measure used in the RBF-Kernel. Skipping that step would cause features that have a larger numerical scale to dominate the computation.
- Dimensionality Reduction:
  Even though the optimization and similarity computations can be performed in full space, we applied Principal Component Analysis (PCA) to reduce the dimensionality to 10. This helps in speeding up computations, especially for kernel matrix computations and it reduces the noise, by getting rid of low-variance directions.

4 Graph Construction

The weights in Equation (1) capture the similarity between two points i and j. The similarity function should assign high weights to points that are close in the feature space, and low weights to those that are distant. Hence, we make use of the RBF-Kernel, defined as

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2\right) \tag{2}$$

Symbol	**Meaning**
$\mathbf{x}, \mathbf{x}'$	Points in Input space
$\|\mathbf{x} - \mathbf{x}'\|^2$	Euclidean Distance (also called $L2$-Norm) between $\mathbf{x}$ and $\mathbf{x}'$
γ	A positive parameter, which controls the spread of the Kernel

Table 2: Descriptions of symbols used in the RBF Kernel

In our experiment, we set the kernel parameter dynamically as $\gamma = \frac{1}{d}$, where d denotes the number of input features after applying principal component analysis (PCA). This choice works well in practice, especially when features were standardized to have zero mean and unit variance. This balances the trade-off between local and global similarity, since high values of γ would cause similarity to decay too quickly, whereas low values of γ would even make distant points appear similar. Since we used $d = 10$ principal components, this gives $\gamma = 0.1$ in practice. The RBF-Kernel function will always produce outputs in the range $K(\mathbf{x}, \mathbf{x}') \in (0, 1]$, since $-\gamma\|\mathbf{x} - \mathbf{x}'\|^2 \leq 0$, and thus $exp(\cdot) \in (0, 1]$. A value of 1 indicates that the two points are identical, whereas values close to 0 correspond to points far apart in feature space. The RBF-Kernel offers several advantages for graph-based SSL problems. Firstly, it preserves locality, so that nearby points get a high similarity score and distant points a low similarity. Moreover, the Kernel decays smoothly and continuously with distance, leading to better optimization behavior. However, the RBF-Kernel is based on the Euclidean distance, such that feature scaling becomes important. Features with larger numerical ranges would dominate the distance computation. To prevent this from happening, we scale all features to have zero mean and a unit variance. Doing so will ensure that each feature contributes equally to the similarity measure.

5 Optimization Formulation

Based on the loss function previously defined in section 2, we now describe the optimization problem and its properties. The goal of optimization is to minimize the loss $f(y)$ on the vector $y^{(u)} = [y_1^{(u)}, y_2^{(u)}, ..., y_u^{(u)}]$. To do so, we treat the labels as real-valued in the interval $[0, 1]$ during the optimization. Because each label must remain in the interval, the optimization is subject to the box constraint. After optimization, the predicted soft labels $y^j \in [0, 1]$ are converted into binary class predictions using thresholding:

$$\hat{y}^j = \begin{cases} 1 & \text{if } y^j > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

The loss function is composed of two squared-difference terms, both of which are quadratic polynomials of degree 2. When summing these terms up, we obtain a loss function that possesses the global form $f(y) = y^T Ay + b^T y + c$. This quadratic function has a positive, semi-definite Hessian matrix. This is due to all squared terms and non-negative weights. As a result, the function is convex. By the properties of a convex function, we know that a global optimum exists. Given the nature of the problem, we apply first-order optimization methods such as Gradient Descent, BCGD (with Gauss-Southwell rule) and Coordinate Minimization, which are all guaranteed to converge to the global minimum. This guarantee holds under standard conditions, such as that proper step size was being chosen.

6 Optimization Methods

6.1 Gradient Descent

6.1.1 Method Overview

Gradient Descent is an iterative first-order optimization algorithm that updates the solution in the negative direction of the gradient of the objective function. Doing so ensures that at each iteration, we will follow the steepest decent and therefore minimizing the loss function. A key assumption of this algorithm is that the underlying objective function has to be differentiable so that its gradient can be computed. The update rule can mathematically be expressed as

$$y^{(t+1)} = y^{(t)} - \alpha_k \nabla f(y^{(t)}) \tag{3}$$

Symbol	**Meaning**
y^t	The current iterative
y^{t+1}	The updated iterative after one step
α_k	The learning rate (step size)
$\nabla f(y^{(t)})$	Gradient of the loss function evaluated at y^t

Table 3: Gradient Descent Update Rule: Symbol Explanation

6.1.2 Gradient Derivation

Based on the loss function defined in Section 2, we derive the gradient in three steps. First, we compute the partial derivative with respect to an unlabeled variable $y_j^{(u)}$

$$\frac{\partial f(y)}{\partial y_j^{(u)}} = 2\left(\sum_{i=1}^{l} w_{ij}^{lu}(y_j - \bar{y}_i) + \sum_{i=1}^{u} w_{ij}^{uu}(y_j - y_i)\right) \tag{4}$$

Next we stack the partial to form the full gradient vector

$$\nabla f(y^{(u)}) = \left[\frac{\partial f}{\partial y_1^u}, \ldots, \frac{\partial f}{\partial y_u^u}\right]^\top \tag{5}$$

Finally, for computational efficiency and clarity, we rewrite the full gradient in vectorized matrix form. This allows us to express the gradient in terms of matrix-vector products as

$$\nabla f(y) = 2\left((W^{lu}\mathbf{1} + W^{uu}\mathbf{1}) \circ y - \left((W^{lu})^\top \bar{y} + (W^{uu})^\top y\right)\right) \tag{6}$$

This formulation enables fast computation of the gradient when used with the Numpy package of python. This last expression matches our implemented full gradient in our project.

6.1.3 Step size Selection

To ensure that Gradient Descent converges the choice of the step size α_k is important. As shown above, our loss function is quadratic, smooth and convex and therefore grantees a global optimum. In this project we explored two different step sizes being

1. Fixed step size $\alpha_k = \frac{1}{L}$: This is the default choice for functions that are convex and are Lipschitz smooth. L denotes the Lipschitz constant of the gradient (i.e its the largest eigenvalue of the Hessian matrix).

2. Accelerated step size $\alpha_k = \frac{2}{L+\sigma}$: Here we assume that the loss function is not only smooth, but also strongly convex, as given in our case. The parameter σ denotes that the function has some guaranteed curvature at any given point. Here we take the steepness of the function into account (through the parameter L) and how strongly the function curves upwards toward the minimum (through the parameter σ). This step size allows us to take bigger steps without overshooting and therefore converging faster.

6.1.4 Heavy Ball Method

The Heavy Ball method, introduced by Polyak (1964), improves the convergence speed of Gradient Descent by adding a momentum term to the update. This helps reduce oscillations and accelerates movement along consistent directions. The update rule is given by:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) + \beta_k(x_k - x_{k-1}) \tag{7}$$

where α_k is the learning rate and β is the momentum coefficient. The term $\beta_k(x_k - x_{k-1})$ carries information from the previous step to smooth the optimization trajectory. The elements of the equation are described in Table 4.

6.1.5 Nesterov Accelerated Gradient

The Nesterov Accelerated Gradient (NAG) method, proposed by Nesterov (1983), modifies the Heavy Ball idea by applying a look-ahead strategy before computing the gradient. Each iteration consists of two steps:

1. **Extrapolation step:**
$$y_k = x_k + \beta_k(x_k - x_{k-1}) \tag{8}$$

2. **Gradient step:**
$$x_{k+1} = y_k - \alpha_k \nabla f(y_k) \tag{9}$$

Here, the gradient is evaluated at the extrapolated point y_k, allowing faster and more stable convergence. Typically, α_k is chosen as $\alpha_k = 1/L$, where L is the Lipschitz constant of the gradient.

Symbol	**Meaning**
x_k	Current iterate at step k
x_{k-1}	Previous iterate at step $k-1$
y_k	Extrapolated point for gradient evaluation
α_k	Learning rate (step size)
$\nabla f(x_k)$	Gradient of loss function at x_k
$\nabla f(y_k)$	Gradient of loss function at y_k
β_k	Momentum coefficient

Table 4: Heavy Ball & Nesterov Accelerated Gradient: Symbol Explanation

6.1.6 Convergence Behavior

We implemented and compared both step sizes in our experiment. For both step sizes Graident Descent showed stable convergence across all runs. Both step sizes converge rapidly; however, the accelerated step size achieves the optimal performance in fewer iterations and lower wall-clock. A detailed comparison is provided in Section 7.

6.2 BCGD with Gauss-Southwell rule

6.2.1 Method Overview

Block Coordinate Gradient Descent (BCGD) is an iterative optimization algorithm that updates only one coordinate at each step from our solution vector $y^{(u)} = [y_1^{(u)}, y_2^{(u)}, ..., y_u^{(u)}]$, instead of the full gradient (as in Gradient Descent). This reduces the computational cost per iteration significantly, which can have big advantages when dealing with high-dimensional datasets.

We implemented the Gauss-Southwell rule to select the best coordinate to upgrade. We defined the best coordinate to upgrade as the one having the largest component in the gradient. Hence, at each iteration, we we choose the index j as:

$$j = \arg\max_k |\nabla f(y)_k|$$

This rule prioritizes the coordinate where the potential for improvement is the highest. However, this does not always guarantee the most effective update globally, as it focuses on local gradient information without accounting for interactions with other variables.

Once the coordinate j is selected, we update it using:

$$y_j^{(t+1)} = y_j^{(t)} + \Delta_j \quad \text{with} \quad \Delta_j = -\alpha_k \nabla_j f(y)$$

To avoid recomputing the full gradient at each iteration, we incrementally update the gradient as follows:

$$\nabla f \leftarrow \nabla f - \Delta_j W_{:,j}^{uu} \quad \text{and} \quad \nabla_j f \leftarrow \nabla_j f + \Delta_j \left(\sum_{i=1}^{l} w_{ij}^{lu} + \sum_{i=1}^{u} w_{ij}^{uu} \right)$$

This makes the method faster, but it might lose some precision compared to full updates. We reuse most of the existing gradient while correcting the parts that where affected by the update.

6.2.2 Step size Selection

In our implementation of BCGD, we use a fixed step size of $\alpha_k = \frac{1}{L}$, were L denotes the global Lipschitz constant of the gradient. We chose this step size because it works well in theory and gave us stable results in practice. While the coordinate-specific step size $\alpha_{kj} = \frac{1}{L_j}$ can perform better, it also introduces variance in converging behavior - especially in higher dimensional settings. Given our goal of a stable and consistent convergence across the synthetic and real-world dataset, we choose the global step size as mentioned above.

6.2.3 Convergence Behavior

Since our objective function is convex and has a Lipschitz-continuous gradient, the BCGD method with the Gauss-Southwell rule is guaranteed to converge to the global optimum. This algorithm can be more efficient then Gradient Descent, if the computation of the full gradient is very expensive.

6.3 Coordinate Minimization

6.3.1 Method Overview

Coordinate Minimization is a optimization method where we cyclically update one coordinate of the solution vector $y^{(u)} = [y_1^{(u)}, y_2^{(u)}, \ldots, y_u^{(u)}]$ at a time while keeping the others fixed. At each iteration, the algorithm solves for the optimal value for a single coordinate that minimizes the objective function w.r.t that coordinate. The minimizer will be computed in closed form and for our quadratic function, the update of the j-th coordinate is given by:

$$y_j = \frac{\sum_{i=1}^{l} w_{ij}^{lu}\bar{y}_i + \sum_{i=1}^{u} w_{ij}^{uu} y_i - w_{jj}^{uu} y_j}{\sum_{i=1}^{l} w_{ij}^{lu} + \sum_{i=1}^{u} w_{ij}^{uu} - w_{jj}^{uu}}$$

The method sweeps through the coordinates in a cyclic fashion, updating one y_j per iteration.

6.3.2 Derivation of the Update Rule

We begin by isolating the terms of the loss function that depend on a single coordinate y_j. From the loss defined in Equation (1), the relevant part is:

$$f(y_j) = \sum_{i=1}^{l} w_{ij}^{lu}(y_j - \bar{y}_i)^2 + \frac{1}{2}\sum_{i=1}^{u} w_{ij}^{uu}(y_j - y_i)^2 + \frac{1}{2}\sum_{i=1}^{u} w_{ji}^{uu}(y_i - y_j)^2$$

Note that in the second and third sums, y_j appears both as $y_j - y_i$ and $y_i - y_j$. Since $w_{ij}^{uu} = w_{ji}^{uu}$, we can combine these and write:

$$f(y_j) = \sum_{i=1}^{l} w_{ij}^{lu}(y_j - \bar{y}_i)^2 + \sum_{i=1}^{u} w_{ij}^{uu}(y_j - y_i)^2$$

To find the optimal y_j, we take the derivative with respect to y_j and set it to zero:

$$\frac{\partial f}{\partial y_j} = 2\sum_{i=1}^{l} w_{ij}^{lu}(y_j - \bar{y}_i) + 2\sum_{i=1}^{u} w_{ij}^{uu}(y_j - y_i) = 0$$

Solving for y_j, we get:

$$y_j \left(\sum_{i=1}^{l} w_{ij}^{lu} + \sum_{i=1}^{u} w_{ij}^{uu} \right) = \sum_{i=1}^{l} w_{ij}^{lu}\bar{y}_i + \sum_{i=1}^{u} w_{ij}^{uu} y_i$$

Finally, we isolate y_j:

$$y_j = \frac{\sum_{i=1}^{l} w_{ij}^{lu}\bar{y}_i + \sum_{i=1}^{u} w_{ij}^{uu} y_i}{\sum_{i=1}^{l} w_{ij}^{lu} + \sum_{i=1}^{u} w_{ij}^{uu}}$$

In practice, we remove the self-interaction term w_{jj}^{uu} from the numerator and denominator (as the update should not depend on the current value of y_j), leading to the implemented form:

$$y_j = \frac{\sum_{i=1}^{l} w_{ij}^{lu}\bar{y}_i + \sum_{i=1}^{u} w_{ij}^{uu} y_i - w_{jj}^{uu} y_j}{\sum_{i=1}^{l} w_{ij}^{lu} + \sum_{i=1}^{u} w_{ij}^{uu} - w_{jj}^{uu}}$$

6.4 No step size needed

In contrast to Gradient Descent or BCGD, Coordinate Minimization does not require a manually chosen step size. This is due to the update being computed directly minimizing the objective with respect to a single coordinate while keeping all others fixed. This is done by using the closed form.

6.5 Convergence Behavior

Coordinate Minimization is guaranteed to converge to the global optimum under convexity and that $y^j \in [0, 1]$. However, convergence may be slower compared to Gradient Descent in terms of iteration count, since only one coordinate is updated per step. Despite this, the per-iteration cost is low, making it attractive for high-dimensional problems.

7 Experiments & Results

This section presents an empirical comparison of four optimization solvers:

- Gradient Descent ($1/L$)
- Gradient Descent ($\frac{2}{L+\sigma}$)
- Gradient Descent (Heavy Ball, Polyak)
- Gradient Descent (Accelerated Gradient, Nesterov)
- Block Coordinate Gradient Descent (BCGD)
- Coordinate Minimization

We evaluated their performance in four aspects: iteration count, CPU time, accuracy, and loss.

7.1 Synthetic Dataset

A synthetic dataset for semi-supervised learning is generated to benchmark and analyze the test performance of different solvers.

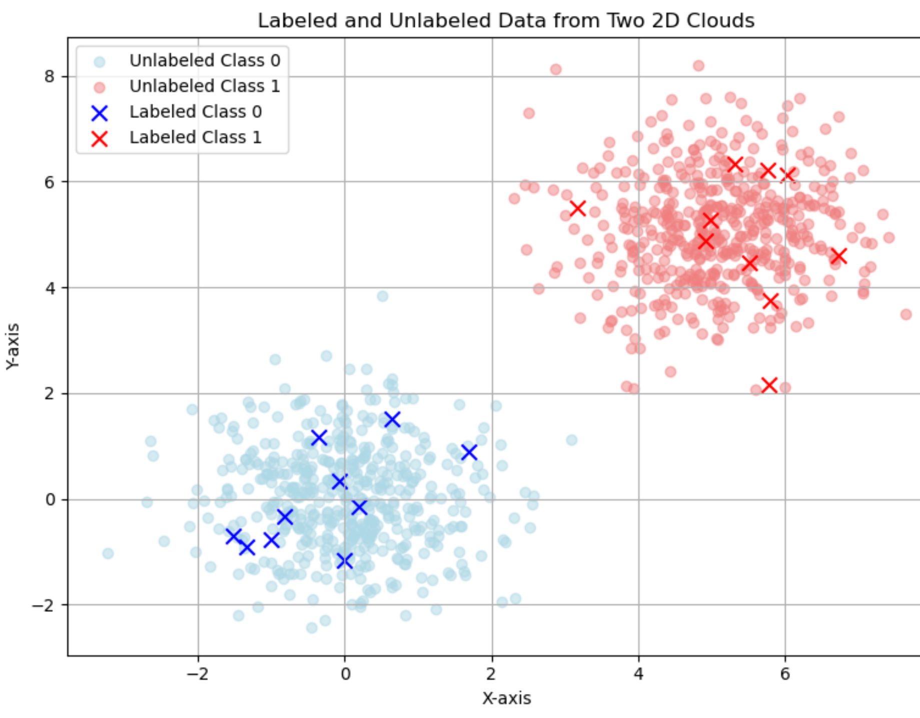

Figure 1: Labeled and Unlabeled Generated Data

With application of the solver models the following result is achieved:

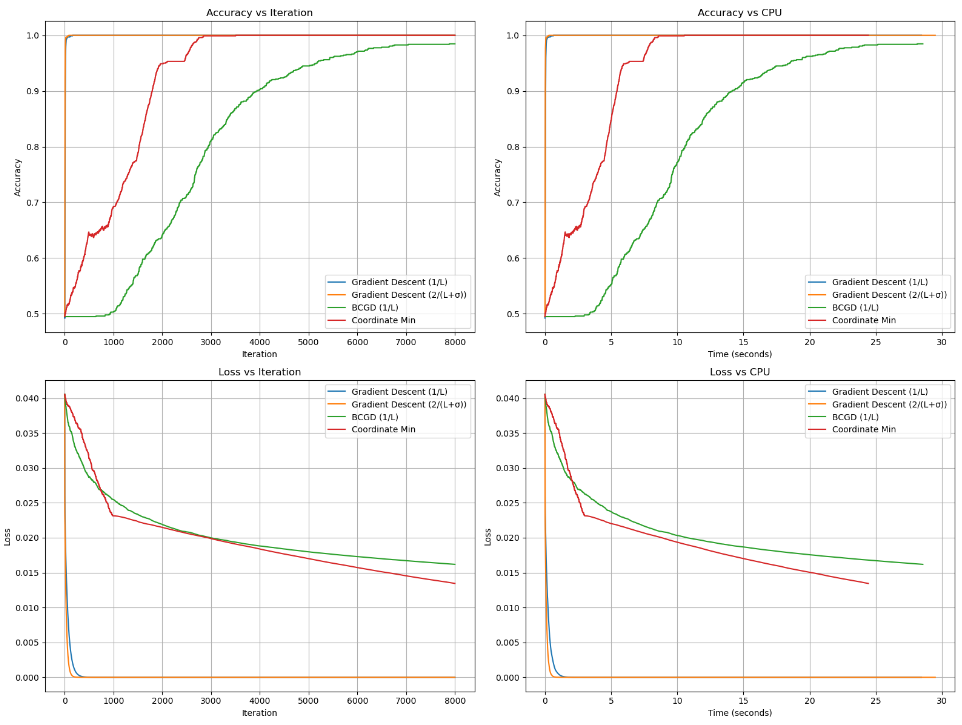

Figure 2: Comparison of the optimization solvers Gradient Descent with step size $\frac{1}{L}$, Gradient Descent with accelerated step size $\frac{2}{L+\sigma}$, BCGD with step size $\frac{1}{L}$, and Coordinate Minimization. The top row shows *Accuracy over Iterations* (left) and *Accuracy over CPU Time* (right). The bottom row displays *Loss over Iterations* (left) and *Loss over CPU Time* (right).

- **Gradient Descent** (both) reach optimal the accuracy immediately (very few iterations needed). In addition, the loss function shows a steep and immediate drop in loss over $Iterations$ and CPU.
- **Coordinate Minimization** requires approximately 3000 iterations to converge to a similar accuracy. However, it is relatively efficient in CPU time with $< 10 seconds$ to reach the peak accuracy. Additionally, the loss is decreased more gradually over the iterations. In the aspect of $CPU - time$ it is catching up better than BCGD, meaning **Coordinate Minimization** is **more efficient per iteration**.
- **BCGD** is the slowest in iterations and CPU times.

In conclusion, all methods eventually converge towards a similar accuracy and final loss value, but the rate of convergence for reaching low loss varies. Gradient Descent is in reaching high accuracy and reducing loss by far the fastest.

7.2 Real World Data

The analysis is based on the data set digits dataset provided by the "sklearn.dataset" module, on which our solvers are trained. The structure of the analysis from subsection 7.1 is preserved.

Figure 3 provides a first impression of the progression of the curves of the different solvers.

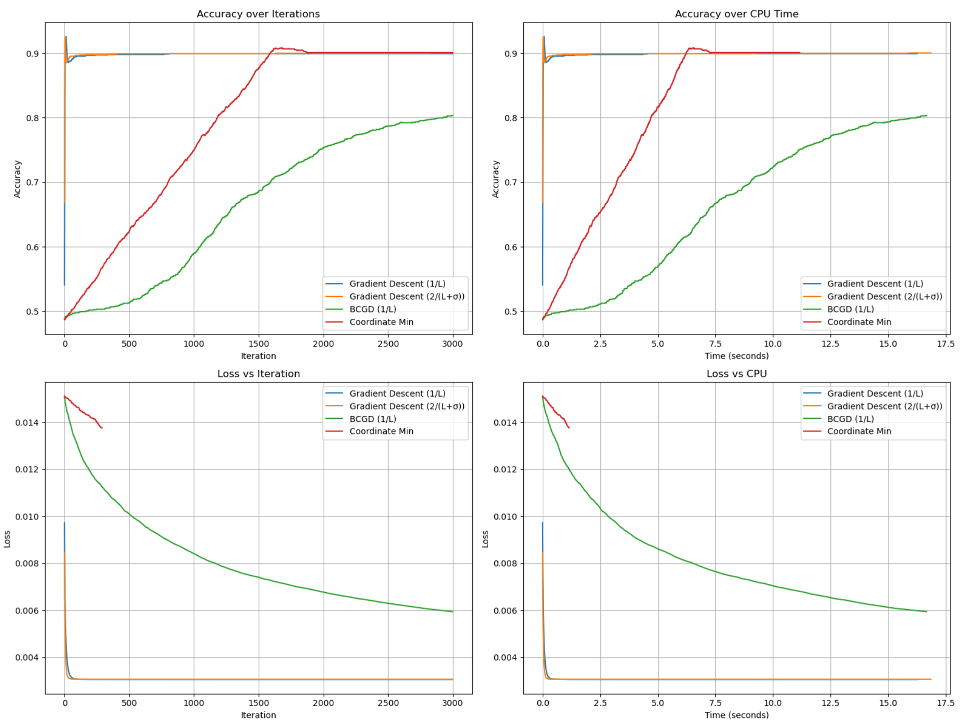

Figure 3: Convergence & Loss Behavior with 3000 Iterations

7.2.1 Accuracy over Iterations

The fastest convergence in terms of iterations is observed with Gradient Descent, both for the step sizes $\frac{1}{L}$ and $\frac{2}{L+\sigma}$, which reach high accuracy almost immediately. Coordinate Minimization follows, showing a steady increase and achieving competitive accuracy within approximately 2000 iterations. In contrast, BCGD shows the slowest convergence.

To determine the best fastest converging method in Gradient Descent the figure 4 is considered, which reflects the approximate converging method to the maximum precision at 50 iterations. In addition, Heavy Ball & Accelerated Gradient Gradient Descent methods are introduced.

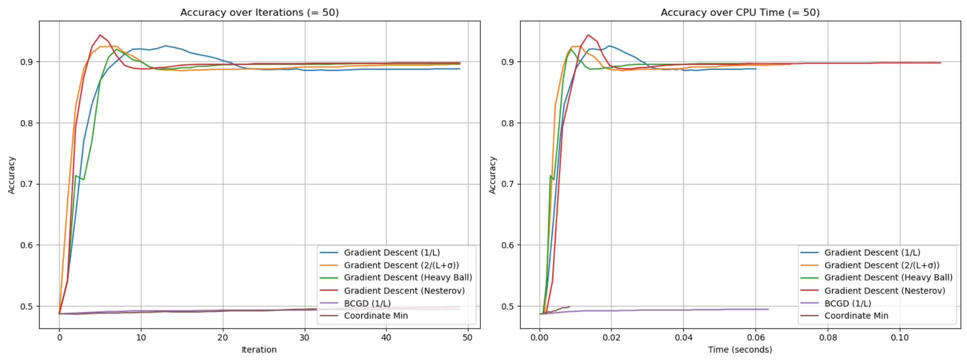

Figure 4: 50 Iteration Zoom for Gradient Descent Methods

This analysis demonstrates that employing Nesterov's Accelerated Gradient is the fastest over *Iterations* to converge to the maximum *Accuracy*, while Gradient Descent Heavy Ball is the fastest over $CPU(time)$. For general information purposes, we retain the old view as in Figure 3.

7.2.2 Accuracy over CPU Time

In terms of computational time:

- **Gradient Descent** and **Coordinate Minimization** reach optimal the accuracy in under 2 seconds.
- **BCGD** requires over 40 seconds to converge to a similar accuracy.

This makes Coordinate Minimization attractive for fast, real-time applications.

7.2.3 Loss vs. Iterations

The Loss plots in the lower part indicate:

- Both **Gradient Descent** and **Coordinate Minimization** exhibit rapid loss reduction. However, **Coordinate Minimization** converges prematurely at around 350 iterations, likely because of the model's failure to predict any positive class labels beyond this stage.
- **BCGD** shows a gradual but consistent decrease in loss over time.

7.2.4 Loss vs. CPU Time

Consistent with earlier observations, Gradient Descent and Coordinate Minimization achieve rapid reduction of losses. BCGD continues to improve steadily but remains comparatively slow.

7.2.5 Convergence Behavior

- **Gradient Descent** (both variants): Have an excellent convergence, but require full gradient evaluations, which may be computationally expensive in large-scale settings.
- **Coordinate Minimization**: Offers fast convergence with low computational cost per iteration, ideal for sparse or structured problems.
- **BCGD**: Slower convergence.

Figure 5 illustrates that all solvers converge to a similar accuracy level, therefore confirming the validity of the models. The Loss functions have not changed significantly.

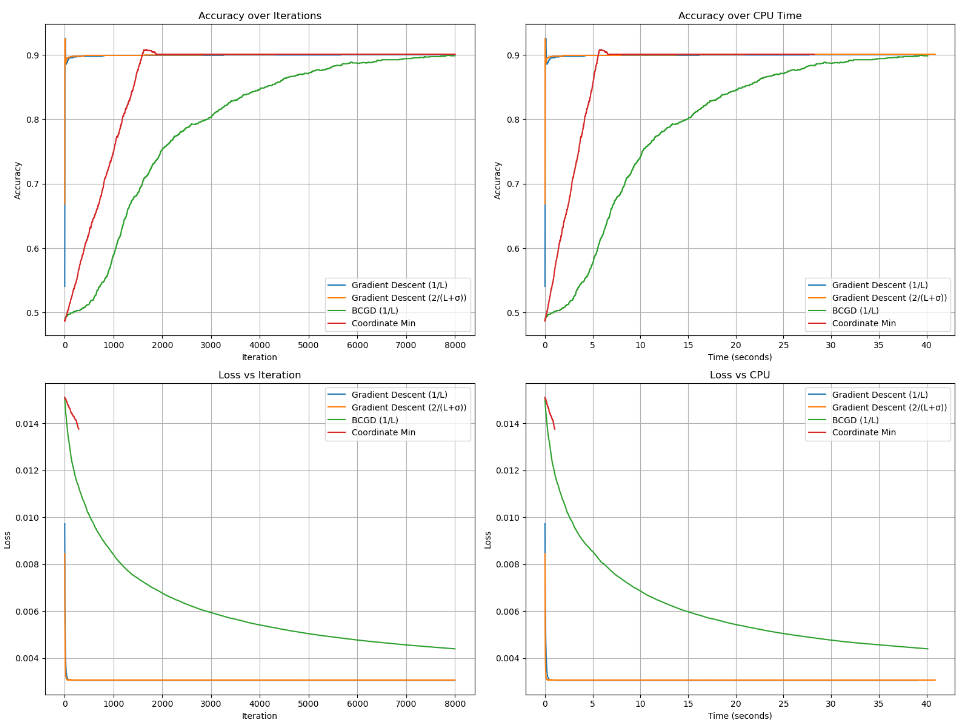

Figure 5: Convergence & Loss Behavior with 8000 Iterations

To check whether the 90% accuracy is consistent and not just due to the number of iterations, we ran an additional analysis. This robustness analysis examines the results across different sizes of training sets. Table 5 introduces a comparison of the solvers' accuracies under varying percentages of labeled training data. Therefore, it is possible to validate the existence of convergence.

Solver Model	**0.01**	**0.03**	**0.05**	**0.08**	**0.10**
Gradient Descent $\left(\frac{1}{L}\right)$	0.89	**0.90**	0.89	0.89	0.90
Gradient Descent $\left(\frac{2}{L+\sigma}\right)$	0.89	**0.90**	0.89	0.89	0.90
Gradient Descent Heavy Ball	0.89	**0.90**	0.90	**0.90**	**0.90**
Gradient Descent Nesterov	**0.90**	**0.90**	0.90	**0.90**	**0.90**
BCGD $\left(\frac{1}{L}\right)$	0.72	0.88	0.86	0.88	0.89
Coordinate Minimization	**0.93**	**0.90**	**0.91**	**0.90**	**0.90**

Table 5: Comparison of the Accuracy Across Different Training Set Sizes

Table 5 shows a similarity in the values the different solvers are converging to. This value is set to approximately 0.9. This matches the final accuracy shown in figure 5. Therefore the convergence to 90% Accuracy is proven. In addition, is noticeable that Nesterov's Gradient Descent shows consistency in accuracy, with the size of the training set changing.

7.2.6 Limitations and Value of BCGD

Despite being the slowest in both iteration and time:

- BCGD is highly scalable and good practice for high-dimensional problems, since one block per step is computed which reduces the computational complexity. The block structure can be adjusted depending on the problem, which makes it easier to work with in different settings.

These features make BCGD a valuable method in practical large-scale optimization, even if it underperforms in small-scale scenarios.

7.3 Conclusion

In our tests, Gradient Descent and Coordinate Minimization both worked efficiently and reached good results quickly. Nonetheless, BCGD remains a relevant and practical choice for problems where computational resources or memory access patterns limit the feasibility of full gradient methods.