Project 10 - SVR with Level Bundle Method

Authors:

- Benedetti Gabriele
- Federico Bonaccorsi

Contents

Project 10 - Description

(M) is a SVR-type approach of your choice (in particular, with one or more kernels of your choice).

(**A1**) is an algorithm of the class of level bundle methods, applied to either the primal or the dual formulation of the SVR.

(**A2**) is a general-purpose solver applied to an appropriate formulation of the problem.

Use of an off-the-shelf solver for the Master Problem of the bundle method is allowed.

Introduction

SVR and Level Bundle Method

This project focuses on the development and implementation of an SVR (Support Vector for Regression) capable of learning from a dataset in the form of "feature x target," where "target" must be a vector of dimensions `n x 1`, in accordance with the definition of SVR. In addition to the basic implementation, a significant part of the SVR will leverage the Level Bundle Method for optimizing the dual function.

For the Master Problem of the **SVR with LBM**, the MATLAB function `quadprog` was used. This function is primarily designed for solving quadratic objective functions with linear terms.

Instead, for the **general purpose SVR** the `fmincon` function was used since this function can solve non-linear problems, making it ideal for our use case.

Support Vector for Regression (SVR)

Support Vector Regression (SVR) aims to find a function $f(X)$ that approximates the training data while minimizing a given loss function. The primal optimization problem is formulated as follows:

$$\begin{aligned} \min_{w,b,\xi,t^*} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) \\ \text{s.t.:} \quad & \\ & y_i - \langle w, x_i \rangle - b \leq \varepsilon + \xi_i, \\ & \langle w, x_i \rangle + b - y_i \leq \varepsilon + \xi_i^*, \\ & \xi_i, \xi_i^* \geq 0 \end{aligned}$$

where:

- w and b define the regression hyperplane
- ξ_i, ξ_i^* are slack variables that account for deviations beyond the margin ε
- C is a regularization parameter

By applying Lagrange multipliers and transforming the problem into its dual formulation, we obtain:

$$
\begin{aligned}
\max_{\alpha} \quad & \sum_{i=1}^{n} y_i \alpha_i - \varepsilon \sum_{i=1}^{n} |\alpha_i| - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j K(x_i, x_j) \\
\text{subject to:} \quad & \\
& \sum_{i=1}^{n} \alpha_i = 0, \\
& -C \leq \alpha_i \leq C \quad \forall i = 1, \ldots, n.
\end{aligned}
$$

where $K(x_i, x_j)$ is a kernel function that allows the method to handle non-linear relationships by implicitly mapping the input data to a higher-dimensional space.

Once the dual problem is solved, the **support vectors** are identified as the data points corresponding to nonzero Lagrange multipliers. Specifically, the support vectors are found by selecting the indices (i, j) that satisfy the condition:

$$\{(i, j) \mid |\alpha_i| > \tau\}$$

where τ is a small positive threshold to account for numerical precision. These support vectors are the most influential data points in defining the regression function, as they determine the final predictive model.

Overview of the Level Bundle Method

The Level Bundle Method (LBM) is an optimization approach that refines solutions iteratively by leveraging cutting-plane techniques and a level constraint. It is particularly useful in non-differentiable optimization problems, such as those encountered in support vector regression.

Reformulating the Problem for `quadprog` in MATLAB

Since the SVR function defined before is **non-differentiable**, we apply the **Level Bundle Method** to approximate it iteratively. Especially we find a new solution by solving the LBM objective function defined as follow:

$$\alpha_{k+1} = \arg\min_{\alpha} \left\{ \frac{1}{2} \|\alpha - \hat{\alpha}_k\|^2 \,\middle|\, \hat{f}_k(\alpha) \leq f_k^{\text{level}}, \ \alpha \in X \right\}$$

where:

- α represents the vector of dual variables in the **Support Vector Regression (SVR)** problem.

- $\hat{\alpha}_k$ is the best solution found so far at iteration k.
- f_k^{level} is the current level used to restrict the search space within an acceptable region.
- X is the set of original constraints of the SVR dual problem.
- $\hat{f}_k$ is defined as follow:

$$\hat{f}_k(x) := \max_{j \in \mathcal{B}_k} \{f(x_j) + \langle \xi_j, x - x_j \rangle\}$$

Where:

$$f(x) = \frac{1}{2} x^\top K x + \varepsilon \sum_{i=1}^{n} |x_i| - y^\top x$$

$$\xi = Kx + \epsilon \cdot \text{sign}(x) - y$$

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0, \\ -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0. \end{cases}$$

However, from MATLAB documentaion, the `quadprog` general function is designed to solve quadratic optimization problems in the following standard form:

$$\begin{aligned} \min_x \quad & \frac{1}{2} x^\top H x + f^\top x \\ \text{s.t.} \quad & Ax \leq b, \\ & A_{\text{eq}} x = b_{\text{eq}}, \\ & lb \leq x \leq ub. \end{aligned}$$

Therefore, it is necessary to reformulate the **Level Bundle Method** problem so that it is compatible with the form required by `quadprog`.

Quadratic Norm Expansion

The objective function of the Level Bundle Method can be rewritten explicitly as follows:

$$\frac{1}{2}\|\alpha - \hat{\alpha}_k\|^2 = \frac{1}{2}\left(\alpha^\top \alpha - 2\hat{\alpha}_k^\top \alpha + \hat{\alpha}_k^\top \hat{\alpha}_k\right) = \frac{1}{2}\alpha^\top \alpha - \hat{\alpha}_k^\top \alpha + \frac{1}{2}\hat{\alpha}_k^\top \hat{\alpha}_k.$$

Since the constant term $\frac{1}{2}\hat{\alpha}_k^\top \hat{\alpha}_k$ does not affect the minimization, we can omit it. The function to minimize then becomes:

$$\frac{1}{2}\alpha^\top \alpha - \hat{\alpha}_k^\top \alpha.$$

Reformulation into `quadprog` Format

By comparing this expression with the standard objective function solved by `quadprog`:

$$\frac{1}{2}x^\top Hx + f^\top x,$$

we obtain the following parameters:

- $H = I$ (identity matrix, since the quadratic term is $\frac{1}{2}x^\top x$).
- $f = -\hat{x}_k$ (since the linear term is $-\hat{\alpha}_k^\top \alpha$).

Substituting these values, we obtain:

$$\frac{1}{2}x^\top Hx + f^\top x = \frac{1}{2}x^\top x - \hat{x}_k^\top x.$$

That is exactly the function that we want to minimize.

Constraints

The Level Bundle Method algorithm is subject to the following constraints:

$$\begin{aligned} \text{s.t. } :& \\ & \hat{f}_k(\alpha) \leq f_k^{\text{level}} \\ & \sum_{i=1}^{n} \alpha_i = 0 \\ & -C \leq \alpha_i \leq C \quad \forall i = 1, \ldots, n \end{aligned}$$

Thus, they must be rewritten into a form solvable by `quadprog`, just as we did for the objective function.

Bundle Cuts Constraint

Given the following constraint:

$$\hat{f}_k(\alpha) \leq f_k^{\text{level}}$$

We can implement it through the cutting plane approach, where the function is approximated via a collection of tangent hyperplanes. These hyperplanes are defined by the subgradients of the function and translate into linear constraints of the form:

$$f(\alpha_k) + \langle \xi_k, \alpha - \alpha_k \rangle \leq f^{\text{level}}$$

Where ξ_j are the subgradients computed at those points and $f(\hat{\alpha_k})$ is the real objective value evaluated.

This formulation must be rewritten in the form $Ax \leq b$. Rewriting the constraint:

$$f(\alpha_k) + \langle \xi_k, \alpha - \alpha_k \rangle \leq f^{\text{level}} \iff \langle \xi_k, \alpha \rangle - f^{\text{level}} \leq \langle \xi_j, \alpha_k \rangle - f(\alpha_k)$$

With:

$$A = \begin{bmatrix} \xi_k^\top & -1 \end{bmatrix}, \quad b = \langle \xi_k, \hat{\alpha_k} \rangle - f(\hat{\alpha_k})$$

Note that the current level f_k^{level} is updated at each iteration using the following formula:

$$f_{\text{level}} = \theta f + (1 - \theta) f_{\text{best}}$$

Equality Constraint

$$\sum_{i=1}^{n} \alpha_i = 0.$$

This can be defined directly as:

$$A_{\text{eq}} = \begin{bmatrix} \mathbf{1}^\top \end{bmatrix}, \quad b_{\text{eq}} = 0.$$

Bound Constraints

$$-C \leq \alpha_i \leq C,$$

These can be defined as:

$$lb = [-C, \, -\infty],$$
$$ub = [C, \, f_{\text{level}}].$$

General Notes and Considerations

As previously described, this project aims to implement an SVR that leverages the Level Bundle Method for optimization. However, the goal is not to achieve a fully optimized SVR in terms of hyperparameters or generalization performance on the dataset cause this would require additional tuning techniques such as grid search or k-fold cross-validation, which are beyond the scope of this project.

Nevertheless, we will present the error achieved using the Mean Squared Error (MSE) metric, along with the selected hyperparameters. Additionally, we will compare our implementation against SVR general solver (A1), which will be used as an oracle.

Conclusion

We have demonstrated how the objective function of the **Level Bundle Method** can be rewritten in the standard form required by `quadprog`. In the following sections, we will present the implementation of the algorithm following the steps outlined so far.

Implementation

Dataset

Initially, we used simple functions such as sine, exponential, and step functions with added noise. These synthetic datasets allowed us to test and verify the correctness of our SVR implementation in terms of predictions and performance. Subsequently, we moved our evaluation to larger and more complex real-world datasets for thoroughly assess the true performance and robustness of our SVR model under challenging conditions.

SVR Dual Function implementation

Firstly, we define the SVR dual function, which computes both the function value `f` and the corresponding subgradient `g`:

```
function svr_dual_functon(alpha_hat, bundle, f_level, C):
Input:
    x           % input vector
    y           % target vector
    epsilon     % scalar value
    K           % kernel matrix

Output:
    f           % objective function value
    g           % gradient

f = 0.5 * TRANSPOSE(x) * (K * x) + epsilon * SUM(ABS(x)) - TRANSPOSE(y) * x
g = K * x + epsilon * SIGN(x) - y
```

Before training, the features are normalized.

(A2) SVR general-purpose solver

For the implementation of a generic SVR solver, we formulated the dual problem using MATLAB's lambda functions. Unlike quadprog, fmincon does not require explicit Hessian matrix definitions—a key advantage that significantly improves scalability for large-scale problems

```
svr_dual =  lambda(x) { svr_dual_function(x, K, Y, epsilon) }

# Starting point
alpha0 = ZEROS(n, 1)

# Inequality constraints (none in this case)
A = EMPTY_MATRIX
b = 0

# Equality constraint: SUM(alpha) = 0
Aeq = ONES(1, n)
beq = 0

# Bounds: -C <= alpha <= C
lb = -C * ONES(n, 1)
ub =  C * ONES(n, 1)

alpha = QP_SOLVE(svr_dual, alpha0, A, b, Aeq, beq, lb, ub)
```

Once the execution is completed, we extract the **support vectors** from the solution.

```
% Identify indices of support vectors
% Select indices where the absolute value of alpha is greater than a threshold (tol)
sv_indices = indices where |alpha[i]| > tol

% Compute the bias term for prediction
bias = mean( Y[i] - sum over j of K[i][j] * alpha[j] ), for all i in sv_indices
```

Performance Evaluation

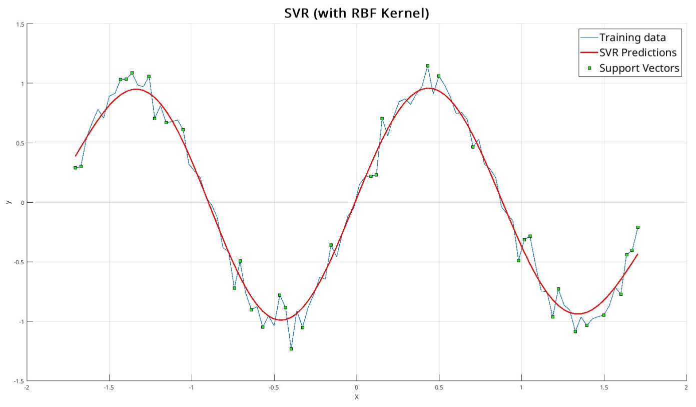
We tested our SVR model on both simple synthetic datasets (e.g., a noisy sine function) and large-scale real-world datasets. The synthetic data allowed us to verify the implementation's correctness, while the real data assessed its practical performance. The results demonstrate that while the solver generally works correctly, its speed significantly decreases when applied to large data volumes.

The following sections will showcase results on benchmark functions (such as noisy sine waves), which demonstrate the SVR implementation's correctness, while the high-dimensional real-world datasets will be addressed in the final chapter.

Synthetic data

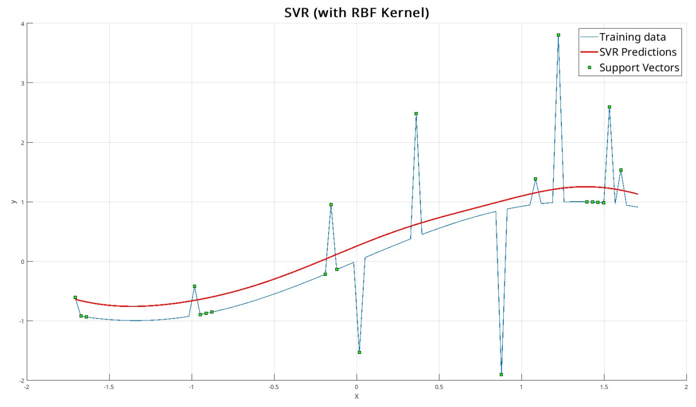
- **Sine function**

$$Y = \sin(3X) + 0.1 \cdot \mathcal{N}(0,1).$$



- **Outliers test**

$$Y = \sin(X) + \underbrace{2 \cdot \mathcal{N}(0,1)}_{\text{10 random points}}.$$



(A1) Level Bundle Method Implementation

Previously, we mathematically formulated the objective function and constraints of the **Level Bundle Method (LBM)** in a format compatible with `quadprog`. We now translate these formulations into code, ensuring a direct correspondence between the mathematical expressions and their implementation.

Hessian Matrix and Linear Coefficients

The objective function of the optimization problem is defined as:

$$H = I, \quad f = -\hat{x}_k.$$

In pseudo-code, this is implemented as:

```
H = BLOCK_DIAG(IDENTITY(n), 0) % the extra 0 is beacuse we have f_level
f = CONCAT(-alpha_hat, 0)
```

Inequality Constraints

The linear constraints are represented in matrix form as:

$$A = \begin{bmatrix} \xi_k^\top & -1 \end{bmatrix}, \quad b = \langle \xi_k, \hat{\alpha_k} \rangle - f(\hat{\alpha_k})$$

```
A = CONCAT_COLUMNS(TRANSPOSE(bundle.g), -ONES(m, 1))
b = SUM_ROWS(bundle.g * bundle.alpha) - bundle.f
```

Equality Constraints

The equality constraints are given by:

$$A_{\text{eq}} = \begin{bmatrix} \mathbf{1}^\top \end{bmatrix}, \quad b_{\text{eq}} = 0.$$

```
Aeq = CONCAT(ONES(1, n), 0)
beq = 0
```

Variable Bounds

The variables are subject to the following bounds:

$$lb = [-C,\ -\infty],$$
$$ub = [C,\ f_{\text{level}}].$$

```
lb = CONCAT(-C * ONES(n, 1), -INF)
ub = CONCAT( C * ONES(n, 1),  f_level)
```

Solving the Optimization Problem

Finally, we use `quadprog` to find the optimal solution. So the ending function is:

```
function mp_solve(alpha_hat, bundle, f_level, C):
    Input:
        alpha_hat   % current estimate of the solution (vector)
        bundle      % set of past points, function values, and gradients
        f_level     % current objective level (level bundle threshold)
        C           % box constraint (‖alpha‖ <= C)

    Output:
        alpha_opt   % vector of alpha dual variables

    n = LENGTH(alpha_hat)
    m = LENGTH(bundle.f)

    H = BLOCK_DIAG(IDENTITY(n), 0)
    f = CONCAT(-alpha_hat, 0)

    A = CONCAT_COLUMNS(TRANSPOSE(bundle.g), -ONES(m, 1))
    b = SUM_ROWS(bundle.g * bundle.alpha) - bundle.f

    Aeq = CONCAT(ONES(1, n), 0)
    beq = 0

    lb = CONCAT(-C * ONES(n, 1), -INF)
    ub = CONCAT( C * ONES(n, 1),  f_level)

    sol = SOLVE_QP(H, f, A, b, Aeq, beq, lb, ub)

    if sol
        alpha_opt = sol[1:n]
    else
        WARN("Best solution not found, keeping previous...")
        alpha_opt = alpha_hat

    return alpha_opt
```

LBM Algorithm: Pseudo-code Implementation

```
Input:
    K           % kernel matrix
    alpha       % current solution (dual vector)
    C           % upper bound on dual variables
    theta       % level parameter
    tol         % convergence tolerance
    max_iter    % maximum number of iterations

% Initialization
f, g = svr_dual_function(alpha)
f_best = f

INIT_BUNDLE(bundle, alpha, f, g)

for iter = 1 to max_iter:
    % Compute the acceptance level
    level = theta * f + (1 - theta) * f_best

    % Solve the master problem to get the new point
    alpha_new = mp_solve(alpha, bundle, level, C)

    % Evaluate the function and subgradient at alpha_new
    f_new, g_new = svr_dual_function(alpha_new, K, y, epsilon)

    % Check if the new point is acceptable
    if f_new < f_best:
        f_best = f_new

    if f_new <= level:
        alpha = alpha_new
        f = f_new

    % Update bundle
    ADD_TO_BUNDLE(bundle, alpha_new, f_new, g_new)

    % Check for convergence
    if ||alpha_new - alpha|| < tol:
        break

return alpha  % Return the optimal solution
```

Performance Evaluation

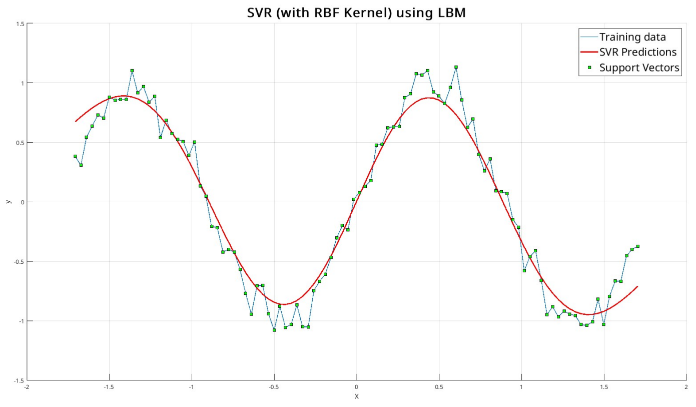
SVR Training with LBM on Synthetic Data

To evaluate the performance of our SVR (Support Vector Regression) model implemented with the Level Bundle Method (LBM), we conducted preliminary testing on predefined synthetic datasets. The results demonstrate that the model retains the generalization capability characteristic of classical SVR while maintaining comparable computational efficiency in terms of execution time.

Below, we present the predictive performance on the sine function as a representative example of the model's generalization capabilities. The remaining functions (omitted for brevity) demonstrate behaviors fully aligned with the standard SVR implementation.

- **Sine function**

$$Y = \sin(3X) + 0.1 \cdot \mathcal{N}(0, 1).$$



Training SVR with LBM on the Abalone Dataset

After validating the SVR's correctness, we evaluated its performance on complex real-world datasets like Abalone. This dataset contains 4177 samples with 8 features each, making it a perfect candidate as it introduces challenges such as:

- A high number of constraints
- Increased memory consumption
- Prolonged training time due to repeated calls to the solver

High Number of Constraints The main issue encountered in the implementation of the **Level bundle method** is the uncontrolled growth of the bundle size at each iteration.

Every cycle of the algorithm adds new constraints to the system, leading to an **exponential increase in the number of conditions** to be handled. This progressive expansion of the bundle has two critical effects:

1. **Memory overload**: The bundle data structure, especially with high-dimensional datasets, consumes resources in a drastically non-linear manner, making allocation unsustainable for large data volumes.
2. **Performance degradation of quadprog**: MATLAB's quadratic programming solver becomes progressively slower—sometimes even leading to total stalls—due to the need to process constraint matrices with thousands of rows.

To address this issue, we introduced the **bundle truncation** technique: once a certain threshold is exceeded, the oldest constraints are removed from the bundle, leaving only the most recent k constraints. This significantly improved both memory efficiency and computational performance.

```
if LENGTH(bundle.alpha) > max_constraints
    REMOVE_FIRST_CONSTRAINT(bundle.alpha)
    REMOVE_FIRST_CONSTRAINT(bundle.f)
    REMOVE_FIRST_CONSTRAINT(bundle.g)
end
```

Memory Consumption Despite the introduction of bundle truncation, memory consumption remained significantly high. This was mainly due to the allocation of large matrices. For instance, given that the dataset contains 4177 features, the matrix H passed to `quadprog` has a size of 4178×4178, which is excessive for an identity matrix.

To mitigate this issue, we decided to switch to **sparse matrices**, which drastically reduced memory usage, as only nonzero elements are stored. Additionally, this optimization ensured that the `quadprog` solver could terminate within a reasonable time frame. So the new `mp_solve` function can be written as:

```
H = SPARSE(BLOCK_DIAG(IDENTITY(n), 0))
f = SPARSE(CONCAT(-alpha_hat, 0))

A = SPARSE(CONCAT_COLUMNS(TRANSPOSE(bundle.g), -ONES(m, 1)))
b = SPARSE(SUM_ROWS(bundle.g * bundle.alpha) - bundle.f)

Aeq = SPARSE(CONCAT(ONES(1, n), 0))
```

```
beq = 0

lb = SPARSE(CONCAT(-C * ONES(n, 1), -INF))
ub = SPARSE(CONCAT( C * ONES(n, 1),  f_level))
```

Performance Optimization: Convergency At this stage, our SVR implementation using the Level Bundle Method (LBM) successfully completes in reasonable time and achieves an **MSE** of **4.3729**, which is very close to the Oracle's performance, thanks to the optimizations implemented thus far. However, the convergence remains relatively slow, requiring more than **200 iterations**, unlike the Oracle which approaches the minimum after approximately **40 iterations**.

This phenomenon is primarily due to the step sizes the algorithm takes: while the Oracle maintains an average step size of approximately 98, our algorithm averages only 1e-1. This limitation is largely attributable to the **matrix H** and the **linear term** that inherently penalize larger steps, thereby slowing convergence.

To address this issue, we introduced a new hyperparameter called "*scale_factor*" that modifies our objective function:

$$\frac{1}{2} \cdot \text{scale\_factor} \cdot \alpha^T \alpha - \text{scale\_factor} \cdot \alpha_k^T \alpha$$

This scaling essentially **reduces the quadratic penalty** while proportionally adjusting the linear term, effectively increasing the **trust region** and consequently allowing larger steps. This modification enabled us to increase average step sizes from *1e-1* to approximately *87*, resulting in an improved **MSE** of **4.2569**, which is comparable to our **Oracle's performance (4.2191).**

Nevertheless, despite the scaling factor, the algorithm still tends to decelerate significantly as it approaches the minimum value. In fact, to achieve that MSE value, we needed to further increase the number of iterations (approximately **120-150**). To overcome this limitation, we introduced **momentum** with **learning rate** applied to the alpha variables:

$$v_{\text{k+1}} = \beta v_k - \eta \cdot \xi_k \alpha_{\text{k+1}} = \alpha_k + v_{\text{k+1}}$$

Where:

- β is the momentum factor
- v_k is the velocity
- η is the learning rate
- ξ is the sungradient

In pseudo-code:

```
velocity = momentum * velocity - learning_rate * g;
alpha = alpha + velocity;
```

```
H = SPARSE(BLOCK_DIAG(IDENTITY(n) * scale_factor, 0))
f = SPARSE(CONCAT(-alpha_hat * scale_factor, 0))
```

This implementation ensures that the method initially takes **large steps** and gradually slows down as it approaches the optimum, while still maintaining adequate step sizes. Unfortunately, this introduced two new hyperparameters (learning rate and momentum) which, if not appropriately selected, can cause the objective function to **diverge**.

Choosing the Solver: Is Quadprog Really the Best Option? The current implementation relies on **quadprog**, with a training time of approximately **50 seconds**, but alternative solvers could significantly improve time efficiency.

Further gains can be achieved by adopting **high-performance solvers** optimized for large-scale problems, as such alternatives leverage best algorithms and parallelization. This could reduce training times by **an order of magnitude** without compromising accuracy.

Conclusions: Achieved Results

In this chapter, we present the results obtained on various datasets, comparing the SVR with Level Bundle Method (LBM) against a classical SVR implementation, referred to as the *Oracle*. The hyperparameters used in each experiment are reported, and efforts were made to keep them as similar as possible to ensure a fair comparison.

As stated in the introductory chapter, the goal of this project is **not** to achieve the best possible MSE through extensive hyperparameter tuning. Instead, the objective is to demonstrate that the Level Bundle Method can deliver **comparable—if not superior—performance** to that of a classical SVR.

Introduction

The dataset used are: Abalone, Airfoil, Red Whine and Friedman function

The parameters for the Level Bundle Method used are:

- **tol**

  Tolerance for stopping criterion.

- **theta**

  Controls the trade-off between cutting plane approximation and descent direction.

- **lr** *(learning rate)*

  Step size for the gradient-based update.

- **momentum**

  Momentum term to accelerate convergence and avoid local minima.

- **scale_factor**

  Scaling coefficient applied to linear and quadratic terms.

- **max_constraints**

  Maximum number of cutting planes (constraints) maintained in the bundle.

Common Parameters

This parameters are used **on all dataset** (except where specified) by both SVR (Oracle and SVR with LBM).

Parameter	Value
Kernel	RBF(sigma=0.5)
C	1
Epsilon	0.05

Abalone

Oracle

Iterations	MSE	Time (s)
60	4.2186	292.2166
90	4.2186	796.9473

SVR with Level Bundle Method (LBM)

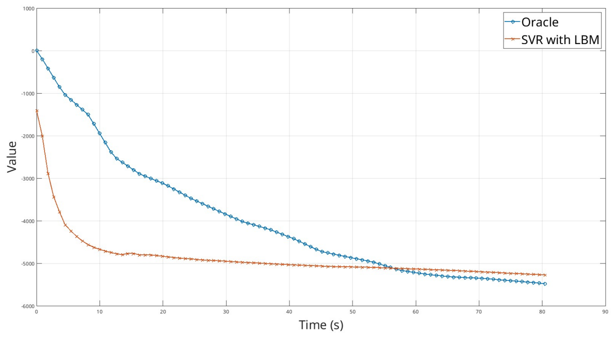
- **tol**: `1e-2`
- **theta**: `0.5`
- **lr** (learning rate): `1e-07`
- **momentum**: `0.3`
- **scale_factor**: `1e-05`
- **max_constraints**: `60`

Iterations	MSE	Time (s)
60	4.2819	28.2933
90	4.1963	88.0550

As observed, the *Oracle* achieves excellent results in just a few iterations, which suggests that `fmincon` is a highly effective solver for this specific type of problem. We also experimented with other alternatives, such as `quadprog`, but the resulting Hessian matrix proved to be unmanageable for large-scale problems like this one, making its use impractical.

An interesting observation is that once the minimum is reached, `fmincon` tends to plateau without further improvements. However, the major drawback is the training time: the average time between iterations is around **6 seconds**, resulting in very high total runtimes.

Our SVR implementation with LBM, on the other hand, performs slightly worse with the same number of iterations, but when increasing the iteration count moderately, it achieves an **even lower MSE** than the Oracle.



This graph shows the behavior of the function at the same time t, using the best parameters for both SVRs. Since the times for the 2 SVRs are markedly different, the values have been interpolated to the values at the common time, so it is easier to understand what happens at time t.

One particularly interesting aspect, evident from this time-based plot, is that up to around **60 seconds**, our LBM-SVR approaches the function minimum much faster, while the Oracle lags significantly behind. Notably, there exists a time `t` at which both models reach **similar performance levels**.

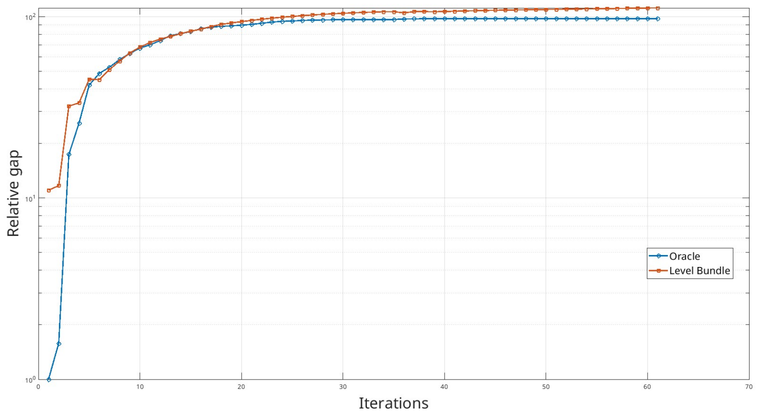


Figure 1: Relative gap between the Oracle and LBM function evaluations

Another key observation comes from the **relative gap** plot: despite the differences in runtime, the learning behavior is remarkably comparable. This indicates that the introduction of regularization had a **stabilizing effect** on the optimization dynamics.

Airfoil

Parameter	Value
Kernel	RBF(sigma=0.6)

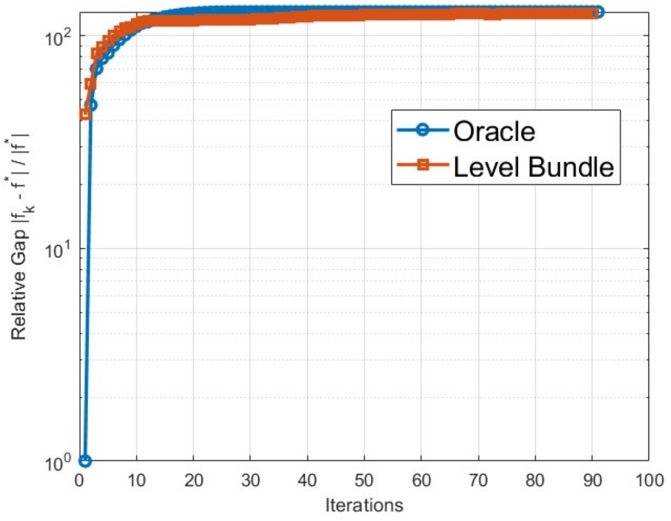
Oracle

Iterations	MSE	Time (s)
60	10.7398	54.05
90	10.7397	94.32

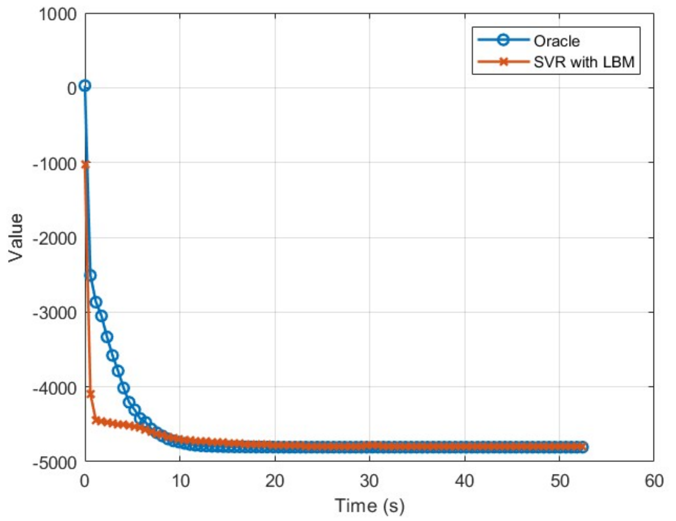
SVR with Level Bundle Method (LBM)

- **tol**: `1e-2`
- **theta**: ‘0.57
- **lr** (learning rate): `1e-07`
- **momentum**: ‘0.34
- **scale_factor**: `4.6416e-05`
- **max_constraints**: `60`

Iterations	MSE	Time (s)
60	10.7976	19.13
90	10.7354	59.04



Unlike the Abalone dataset results, here the two SVR models show a highly synchronized learning behavior, supported by the minimal discrepancy in their MSE values



The time-dependent behavior is noteworthy: while the function evaluation shows a substantial initial discrepancy, the difference almost completely disappears after 10 seconds, resulting in virtually indistinguishable outputs.

Red Whine

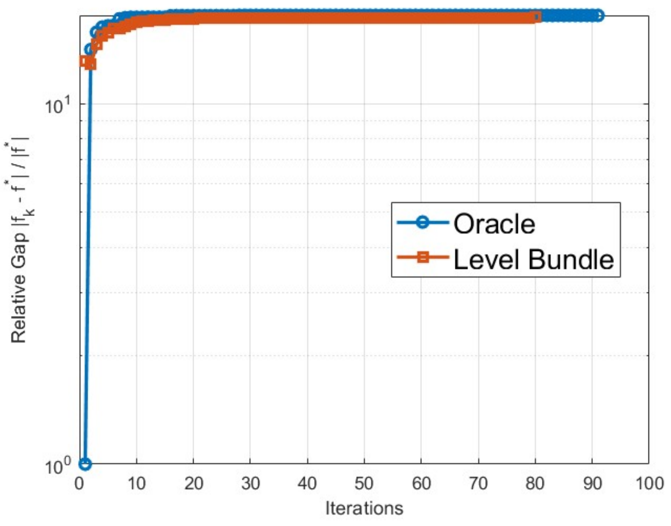
Oracle

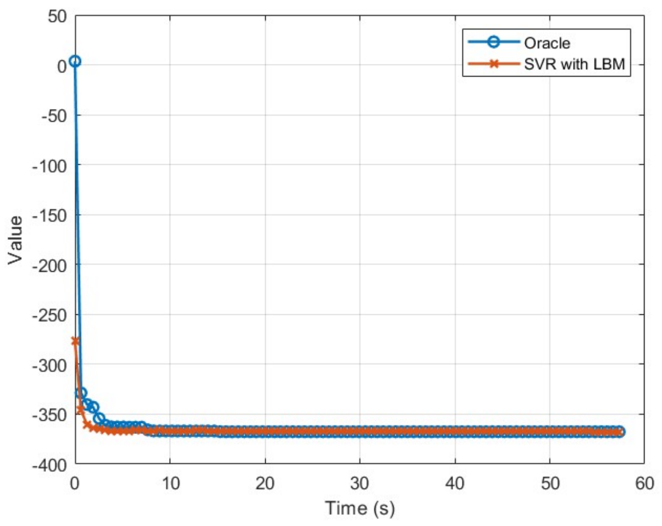
Iterations	MSE	Time (s)
60	0.055518	33.87
90	0.055519	59.08

SVR with Level Bundle Method (LBM)

- **tol**: `1e-2`
- **theta**: `0.4878`
- **lr** (learning rate): `4.1169e-06`
- **momentum**: `0.654`
- **scale_factor**: `3.7483e-05`
- **max_constraints**: `60`

Iterations	MSE	Time (s)
60	0.057641	26.29
90	0.055734	51.52





As with the Red Wine dataset, here too we observe no significant differences between the two SVR models: both exhibit excellent and comparable convergence, evidenced by the small MSE gap and similar computational time.

Summary

Dataset	Metodo	Iterazioni	MSE	Tempo (s)
Abalone	Oracle	60	4.2186	292.2166
Abalone	LBM (SVR)	60	4.2819	28.2933
Abalone	Oracle	90	4.2186	796.9473
Abalone	LBM (SVR)	90	4.1963	88.0550
Airfoil	Oracle	60	10.7398	54.05
Airfoil	LBM (SVR)	60	10.7976	19.13
Airfoil	Oracle	90	10.7397	94.32
Airfoil	LBM (SVR)	90	10.7354	59.04
Red Whine	Oracle	60	0.055518	33.87
Red Whine	LBM (SVR)	60	0.057641	26.29
Red Whine	Oracle	90	0.055519	59.08
Red Whine	LBM (SVR)	90	0.055734	51.52