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:

$$\max_{\alpha} \quad \sum_{i=1}^{n} y_i \alpha_i - \varepsilon \sum_{i=1}^{n} |\alpha_i| - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j K(x_i, x_j)$$

subject to:

$$\sum_{i=1}^{n} \alpha_i = 0,$$
$$-C \leq \alpha_i \leq C \quad \forall i = 1, \ldots, n.$$

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.

Problem Formulation

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:

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

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:

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

Lower Bound Estimation in the Level Bundle Method

In the classical Level Bundle Method, at each iteration a convex overestimator of the dual function is constructed using subgradients. This overestimator can then be used to compute a lower bound on the dual function. Such a lower bound is valuable for guiding the choice of the level parameter, helping the algorithm converge more efficiently toward the optimal solution. Given the constraint function:

$$\hat{f}_k(\alpha) = \max_{i \in B_k} \{ f(z_i) + \langle \xi_i, \alpha - z_i \rangle \}$$

This piecewise-linear and convex function overestimates the true dual objective function, and can be used to compute a global lower bound:

$$f_{\text{lower}} = \min_{\alpha \in X} \hat{f}_k(\alpha)$$

To compute this lower bound in practice, we solve the following convex optimization problem by introducing an auxiliary scalar variable t:

$$\begin{array}{ll} \min_{\alpha,t} & t \\ \text{s.t.} & f(z_i) + \langle \xi_i, \alpha - z_i \rangle \leq t, \quad \forall i \in B_k \\ & \sum_{i=1}^{n} \alpha_i = 0 \\ & -C \leq \alpha_i \leq C, \quad \forall i = 1, \ldots, n \end{array}$$

Then we can evaluate the level by using:

$$f^{\text{level}} = f_{\text{lower}} + \theta * (f_{\text{upper}} - f_{\text{lower}})$$

Where f_{upper} is the current best solution found at iteration k

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

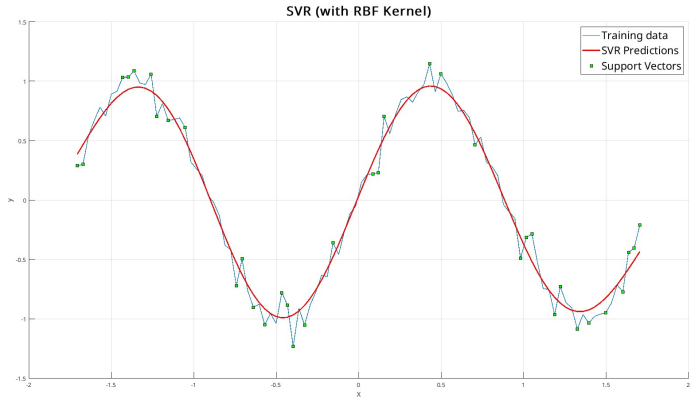
We tested our SVR model on both simple synthetic datasets (e.g., a noisy sine function) and large-scale real-world datasets. The synthetic data allowed us to verify the implementation's correctness, while the real data assessed its practical performance. The results demonstrate that while the solver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

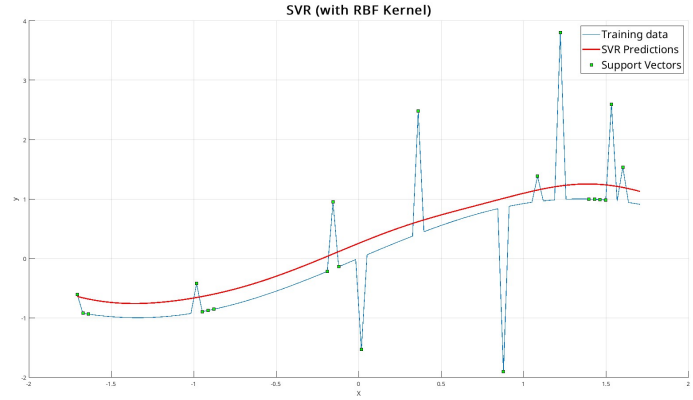
- **Sine function**

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



- **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
```

Lower bound estimation for Level

Given the following optimization problem

$$\begin{array}{ll} \min_{\alpha,t} & t \\ \text{s.t.} & f(z_i) + \langle \xi_i, \alpha - z_i \rangle \leq t, \quad \forall i \in B_k \\ & \sum_{i=1}^{n} \alpha_i = 0 \\ & -C \leq \alpha_i \leq C, \quad \forall i = 1, \ldots, n \end{array}$$

The corrisponding pseudo-code can be:

```
function compute_lower_bound(bundle, C)
    % Input:
    %   bundle: structure containing fields alpha, g, f
    %   C: bounding parameter for variables
    % Output:
    %   lb: computed lower bound

    [n, m] = SIZE(bundle.alpha)  % n = rows, m = columns

    % Build inequality constraints
    A = [TRANSPOSE(bundle.g), -ONES(m, 1)]
    b = SUM(bundle.g .* bundle.z, 1)' - TRANSPOSE(bundle.f)

    % Define linear program
    objective = [ZEROS(n, 1); 1]
    A_eq = [ONES(1, n), 0]
    b_eq = 0
    lower_bounds = [-C * ONES(n, 1); -inf]
    upper_bounds = [ C * ONES(n, 1);  inf]

    % Solve linear program
    [~, lb, status] = SOLVE_LP(objective, A, b, A_eq, b_eq, lower_bounds, upper_bounds)

    return lb
```

Then we can compute the level with:

```
level = lb + theta * (f_best - lb)
```

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 lower bound
    lb = compute_lowerbound(bundle, C)

    % Compute the acceptance level
    level = lb + theta * (f_best - lb)

    % 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

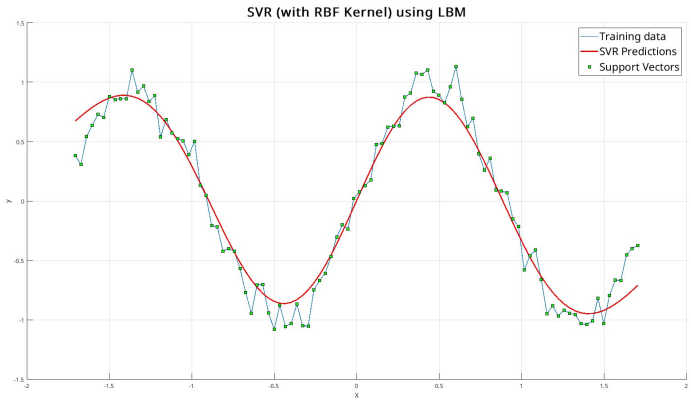
SVR Training with LBM on Synthetic Data

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))
```

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.

For each dataset, two plots will be displayed:

1. **Relative Gap Plot**: This shows the relative gap between the oracle and the SVR with LBM (Level Bundle Method), calculated using the formula:

$$\text{rg} = \frac{|f_k - f^*|}{|f^*|}$$

2. **Function Value Plot**: This displays the function value at the same time t.

Since the two models have different training times, an interpolation between their respective data points was performed using MATLAB's `interp1` function to align their results for comparison.

Introduction

The dataset used are: Abalone, White Whine, Red Whine and Airfoil :

Dataset	Inputs	Features
Abalone	4,177	8
White Wine	4,898	11
Red Wine	1,599	11
Airfoil	1,503	5

The parameters for the Level Bundle Method used are:

- **tol**

  Tolerance for stopping criterion.

- **theta**

  Controls the step balance between the current lower bound and the best-known solution.

- **max_constraints**

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

Abalone

Parameter	Value
Kernel	RBF(sigma=0.4)
C	1
Epsilon	0.1

Oracle

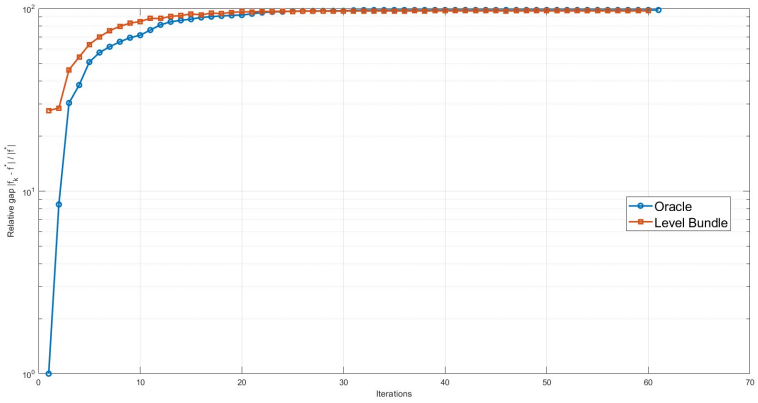
Iterations	MSE	Time (s)
60	4.1976	334.2511

SVR with Level Bundle Method (LBM)

- **tol**: `1e-2`
- **theta**: `0.6`

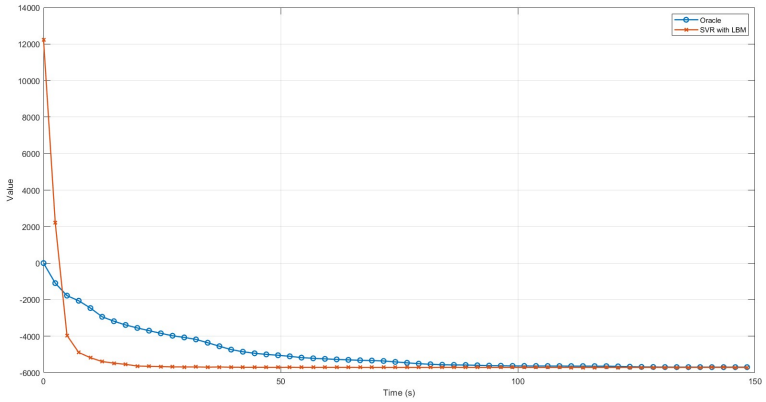
- **max_constraints**: `60`

Iterations	MSE	Time (s)
60	4.1966	148.3282



The plot highlights a clear divergence in the early stages of the iterations. During the first 5–10 iterations, the relative gap for the Oracle exhibits a steep increase indicating a fast convergence to a near-optimal state in a few steps.

In contrast, the Level Bundle Method (LBM) begins with a noticeably higher relative gap and shows a more gradual increase over the initial iterations, characterized by smaller steps. Over the subsequent 15–20 iterations, the performance of both methods starts to align in terms of the magnitude of the relative gap and the step length of its decrease. Eventually, both algorithms reach a plateau where further reductions in the relative gap become minimal, suggesting they have both converged closely to their respective optimal function value, with no further noticeable differences in their convergence behavior.



Observing the temporal performance within the first 50 seconds, the SVR with LBM achieves a considerably faster value reduction compared to the Oracle.

The LBM reaches the value of approximately -5000 in about 10 seconds, whereas the Oracle takes around 50 seconds to reach a similar level. Around the 60-second mark, both methods attain comparable values, after which the LBM continues to exhibit a slight advantage. This faster convergence of the LBM confirms the approximate 50% speedup it provides on this high-dimensional problem.

White Wine

Parameter	Value
Kernel	RBF(sigma=0.5)
C	1
Epsilon	0.01

Oracle

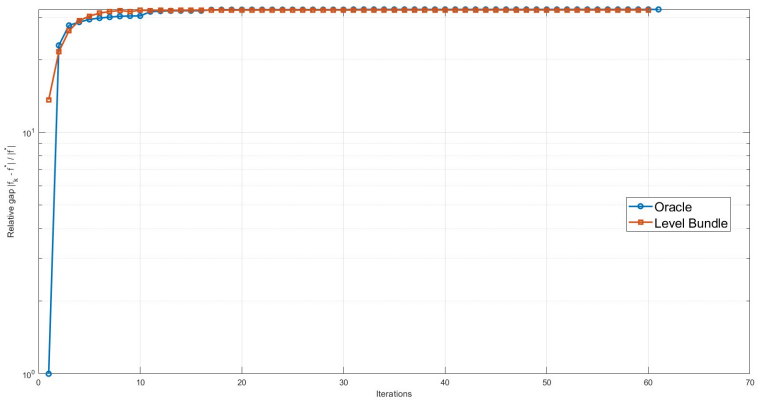
Iterations	MSE	Time (s)
60	0.06494	438.608

SVR with Level Bundle Method (LBM)

- **tol**: `1e-2`
- **theta**: `0.8`

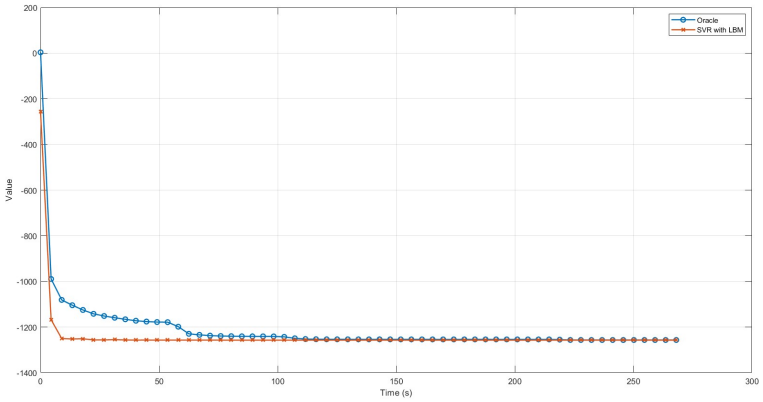
- **max_constraints**: `60`

Iterations	MSE	Time (s)
60	0.064731	268.0662



From the realtive gap plot we observe no significant differences between the two SVR models: after few initial step (where the Oracle has a steeper increase

as noticed also in Abalone results) both exhibit excellent and comparable results, evidenced by the small MSE and identical convergence steps after eleventh iteration.



Consistent with the findings on the Abalone dataset, the Oracle exhibits a significantly slower processing time, particularly noticeable within the first minute where a substantial temporal gap exists.

However, beyond this initial period, thanks to the initially larger Oracle's steps size, the difference diminishes considerably, and the two methods proceed with similar steps within comparable timeframes. Despite this initial and substantial temporal lag, which contributes to an overall runtime for the Oracle exceeding that of the LBM by over 200 seconds, the final convergence in performance remains comparable between the two approaches.

Red Wine

Parameter	Value
Kernel	RBF(sigma=0.5)
C	1
Epsilon	0.01

Oracle

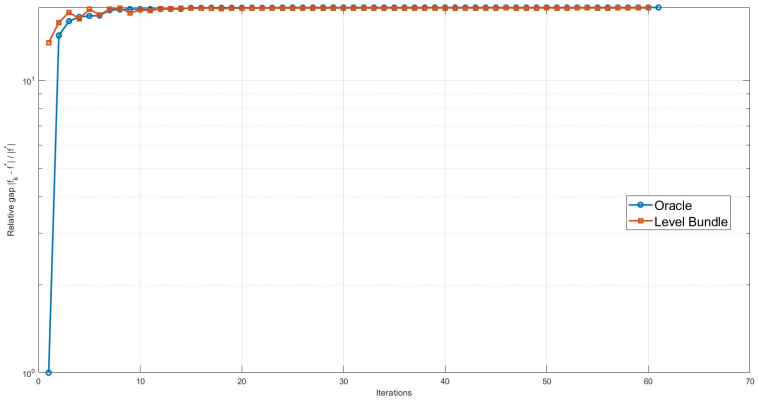
Iterations	MSE	Time (s)
60	0.05532	29.7456

SVR with Level Bundle Method (LBM)

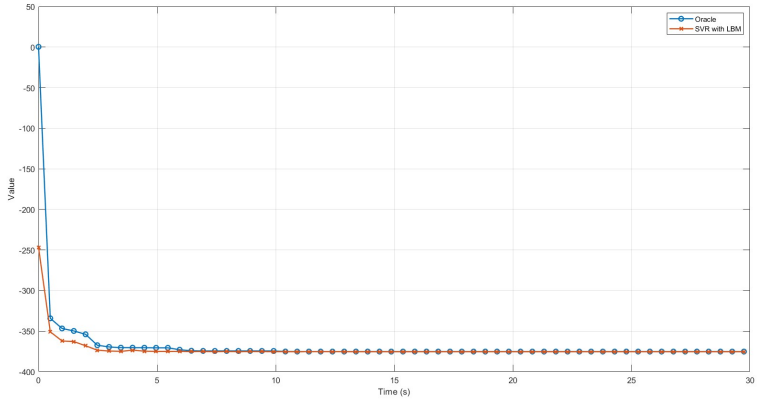
- **tol**: `1e-2`
- **theta**: `0.6`

- **max_constraints**: `60`

Iterations	MSE	Time (s)
60	0.055118	43.705



Similar to the White Wine dataset, both SVR models exhibit comparable and excellent convergence, as evidenced by the minimal relative gap difference observed after the Oracle method's initial iteration.



Analyzing the time-based performance reveals a negligible difference between the two methods. Comparing the plot with the total runtime results, it's noticeable that in this case, the computational simplicity of the Oracle might be favored over the LBM approach, demonstrating an approximate 30% performance advantage for the Oracle in terms of overall computation time.

Airfoil

Parameter	Value
Kernel	RBF(sigma=0.7)
C	1
Epsilon	0.01

Oracle

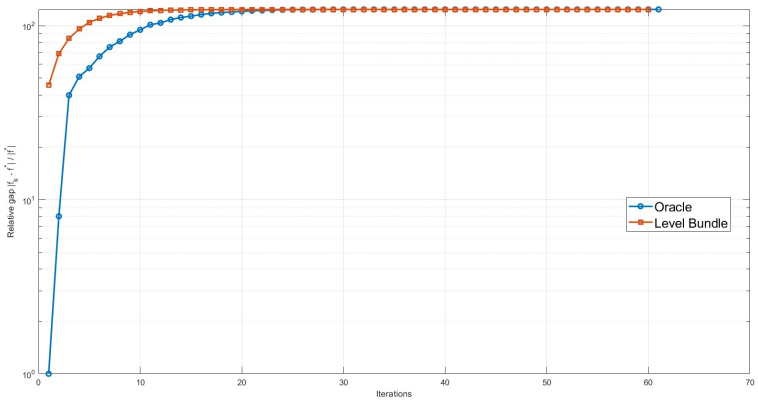
Iterations	MSE	Time (s)
60	10.2377	46.8911

SVR with Level Bundle Method (LBM)

- **tol**: `1e-2`
- **theta**: `0.7`

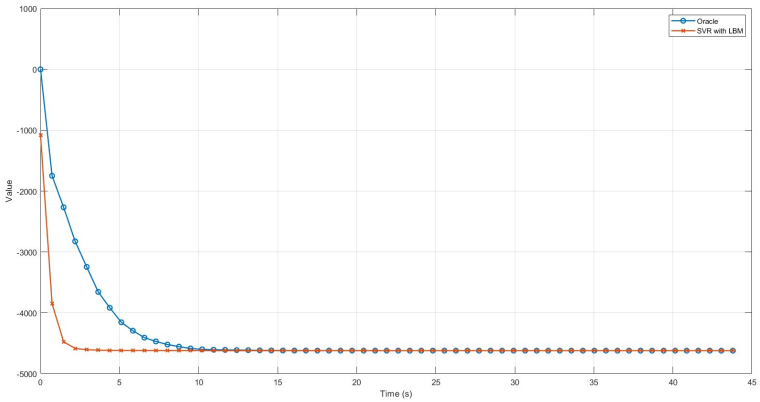
- **max_constraints**: `60`

Iterations	MSE	Time (s)
60	10.2376	43.7915



Similar to previous dataset observations, a noticeable difference in the relative gap between the two methods is apparent in the initial 10 iterations, with their

performance tending to align from approximately 15 to 20 iterations. The Oracle exhibits an initial increase in the relative gap magnitude within the first few steps before rapidly converging towards the LBM's result, eventually reaching a plateau indicative of optimal convergence.



In terms of computational time, the Oracle, thanks to its noticeably larger initial steps in function evaluation, reaches a similar function value as the LBM after roughly 10 seconds, resulting in virtually indistinguishable outputs thereafter. However, this initial temporal lag contributes to an overall performance difference, with the Oracle taking approximately 6% (3 seconds) longer to converge fully compared to the LBM.

Summary

Dataset	Model	Iterations	MSE	Time (s)
Abalone	Oracle	60	4.1976	334.2511
Abalone	LBM (SVR)	60	4.1966	148.3282
White Whine	Oracle	60	0.064935	438.608
White Whine	LBM (SVR)	60	0.064731	268.0662
Red Whine	Oracle	60	0.05532	29.7456
Red Whine	LBM (SVR)	60	0.055118	43.705
Airfoil	Oracle	60	10.2345	46.8911
Airfoil	LBM (SVR)	60	10.2336	43.7915

Across all datasets, the Level Bundle Method (LBM) with Support Vector Regression (SVR) generally achieves a marginally lower MSE compared to the Oracle approach. However, the computational time efficiency of the two methods varies depending on the dataset size and complexity.

For large, high-dimensional datasets such as Abalone and White Wine, the Oracle requires substantially more time to converge, with differences reaching up to 50%, likely due to its more exhaustive evaluation of candidate solutions. In these scenarios, LBM (SVR) significantly reduces the runtime by approximately 45–50% while still delivering a slightly better MSE.

In contrast, for smaller or lower-dimensional datasets like Airfoil and Red Wine, the time difference between the two methods becomes less pronounced. As observed in the Airfoil dataset, the time gap is limited to around 3 seconds (6%) . Interestingly, in the Red Wine dataset (which has a higher number of features but a significantly lower number of input samples compared to Abalone and White Wine), the Oracle is actually faster than LBM (SVR) to reach 60 iterations, despite its MSE remaining marginally higher. This suggests that in cases with limited input data but higher feature dimensionality, the overhead associated with the LBM's bundle maintenance and update processes can outweigh its benefits.

Overall, LBM (SVR) appears to offer the best trade-off for prediction tasks involving a large number of input samples. However, when dealing with datasets characterized by a high number of features but a limited number of input data points, the Oracle tends to outperform LBM in terms of computational time, although its MSE is typically slightly higher.