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

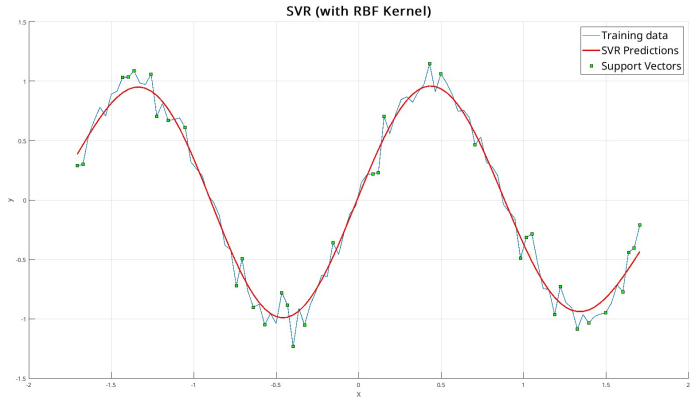
We tested our SVR model on both simple synthetic datasets (e.g., a noisy sine function) and large-scale real-world datasets. The synthetic data allowed us to verify the implementation's correctness, while the real data assessed its practical performance. The results demonstrate that while the solver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

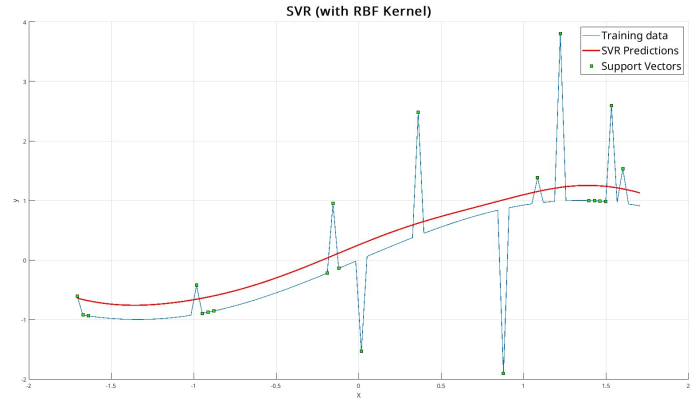
- **Sine function**

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



- **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

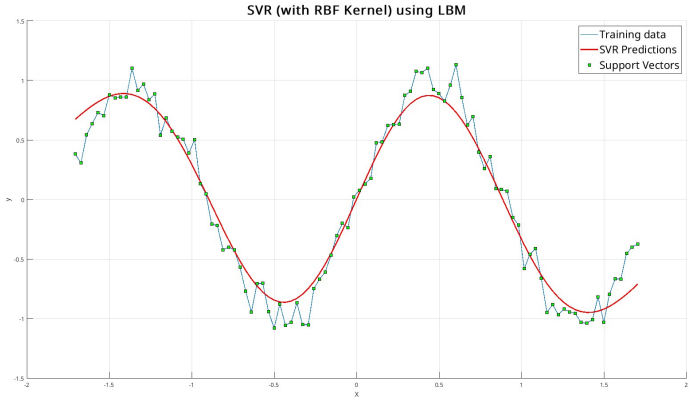
SVR Training with LBM on Synthetic Data

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.