Computational Mathematics for Learning and Data Analysis

Carmine Caserio, Simone Schirinzi

2018/2019

1 Introduction

Support Vector Machine for Regression[1] (SVR in the following) is a machine learning model for regression problems, while Frank-Wolfe is an iterative first-order optimization algorithm for constrained convex optimization.

In this project we face up to both implementation, in an optimized way, and testing using a small dataset.

2 Formalization

The problem we're facing is the implementation of an SVR that uses the Frank-Wolfe method[2] for the resolution of the optimization problem.

About the function the SVR requires, we start by the Vapnik formulation, that uses the slack variables ξ, that is:

$$\text{minimize } \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i - \xi_i^*) \tag{1}$$

$$\text{subject to } \begin{cases} -b - \langle w, x_i \rangle + y_i \leq \epsilon + \xi_i \\ b + \langle w, x_i \rangle - y_i \leq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{cases}$$

Where the constant C determines the trade-off between the flatness of f and the amount up to which deviations larger than ϵ are tolerated. It is correlated to the ϵ-tube and corresponds to address with the ϵ-intensive loss function.

Since we are supposing to handle with a convex objective function, we proceed by introducing a dual set of variables as in the following:

$$\begin{aligned} L := \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) - \sum_{i=1}^{n}(\eta_i\xi_i + \eta_i^*\xi_i^*) \\ -\sum_{i=1}^{n}\alpha_i(\epsilon + \xi_i - y_i + \langle w, x_i\rangle + b) - \sum_{i=1}^{n}\alpha_i^*(\epsilon + \xi_i^* - y_i + \langle w, x_i\rangle + b) \end{aligned} \tag{2}$$

From the saddle point condition, we have that the partial derivatives over the primal variables w, b, ξ_i, ξ_i^* have to go away, for the optimality condition.

So, we have that:

$$\partial_b L = \textstyle\sum_{i=1}^{n}(\alpha_i^* - \alpha_i) = 0 \qquad \partial_w L = w - \textstyle\sum_{i=1}^{n}(\alpha_i - \alpha_i^*)x_i = 0$$

$$\partial_{\xi_i} L = C - \alpha_i - \eta_i = 0 \qquad \partial_{\xi_i^*} L = C - \alpha_i^* - \eta_i^* = 0$$

By substituting these partial derivatives in (2) we obtain the dual optimization problem:

$$\begin{aligned} \text{maximize } & -\frac{1}{2}\sum_{i,j=1}^{n}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)\langle x_i, x_j\rangle \\ & -\epsilon\sum_{i=1}^{n}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{n}y_i(\alpha_i - \alpha_i^*) \\ \text{subject to } & \sum_{i=1}^{n}(\alpha_i - \alpha_i^*) = 0 \text{ and } 0 \le \alpha_i, \alpha_i^* \le C, \ \forall i \end{aligned} \tag{3}$$

Now we have two problems to face up with:

- Compute the regression function;
- Solve the quadratic optimization.

2.1 Compute Regression Function

From the derivation (3), we can see that the dual variables η_i, η_i^* have been removed. This implies that:

$$\begin{cases} \eta_i = C - \alpha_i \\ \eta_i^* = C - \alpha_i^* \end{cases}$$

This means that:

$$\partial_w L = w - \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)x_i = 0 \Longrightarrow w = \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)x_i$$

and, subsequently,

$$f(x) = \sum_{i=1}^{n} (\alpha_i - \alpha_i^*) \langle x_i, x \rangle + b$$

The notation can be simplified by considering that

$$\beta_i = \alpha_i - \alpha_i^* \quad \text{subject to} \quad -C \leq \beta \leq C$$

then, the objective function becomes:

$$f(x) = \sum_{i=1}^{n} \beta_i \langle x_i, x \rangle + b$$

where, defined E as $\{ \ i \mid \epsilon \ < \ \mid \beta_i \mid \ < \ C - \epsilon \}$

$$b = \begin{cases} {}^{1}/_{|E|} \ \sum_{i \in E} \ y_i - \epsilon \cdot \operatorname{sgn}(\beta_i) - \langle \beta, \ K[i] \rangle & \text{if } |E| > 0 \\ {}^{\max(y)}/_{\min(y)} & \text{otherwise} \end{cases}$$

2.2 Solve quadratic optimization

We have to solve the function that (3) maximizes.
The notation can be simplified by considering that

$$\beta_i = \alpha_i - \alpha_i^*$$

Since we don't know the sign of β, we introduce the variable γ subject to:

$$\gamma \geq \begin{cases} \beta & \text{if } \beta \geq 0 \\ -\beta & \text{if } \beta \leq 0 \end{cases}$$

$$\equiv \gamma \geq \begin{cases} \beta & \text{if } \alpha_i \geq \alpha_i^* \\ -\beta & \text{if } \alpha_i \leq \alpha_i^* \end{cases}$$

Since the Frank-Wolfe method is used to minimize the objective function, while (3) maximizes the function, and supposing we're handling with convex functions, the following equality holds:

$$\text{maximize } f(c_1, ..., c_n) = -\text{minimize } - f(c_1, ..., c_n)$$

We would like to learn a nonlinear function using a regression rule applied to the transformed data points $\varphi(\vec{x}_i)$ using a non linear mapping function.

We can avoid to compute the $\varphi(\vec{x}_i)$, by using the kernel trick, that is by using the function k which satisfies $k(\vec{x}_i, \vec{x}_j) = \varphi(\vec{x}_i) \cdot \varphi(\vec{x}_j)$.

The training data points $\vec{x}_i$, that are the support vectors, lies on a non-linearly separable space, for this reason we use the kernel function which maps the data points to a highly dimensional feature space, where they can be linearly separable.

The kernel function used in the following is the gaussian one, that is:

$$k(\vec{x}_1, \vec{x}_2) = e^{-\frac{1}{2\sigma^2}\|\vec{x}_1 - \vec{x}_2\|}$$

The resolution of the objective function is shown in the following, where we have that the ϵ in the following steps should not be confused with the ϵ used in the Frank-Wolfe algorithm, since in this case it is the ϵ used for the ϵ-*tube*:

$$-\min\left\{\epsilon\sum_{i=1}^{n}(\gamma_i) + \frac{1}{2}\sum_{i=1}^{n}\beta_i\beta_j k(\vec{x}_i, \vec{x}_j) - \sum_{i=1}^{n} y_i\beta_i\right\}$$

$$\equiv -\min\left\{\langle\epsilon, \gamma\rangle + \frac{1}{2}\beta K\beta - \langle y, \beta\rangle\right\} \tag{4}$$

The feasible region of the formulation above is:

$$S = \begin{pmatrix} -C \leq \beta \leq C \\ \sum_{i=1}^{n}\beta_i = 0 \\ \gamma \geq \beta \\ \gamma \geq -\beta \end{pmatrix} \tag{5}$$

Where the gradient can be computed as follows:

$$\nabla f(\beta, \gamma) = \begin{pmatrix} K\beta - y \\ \\ \epsilon \end{pmatrix}$$

In this formulation, we have that K is the kernel matrix, obtained by precomputing the kernel function of the various x_i, x_j among themselves:

$$K = \begin{pmatrix} k(\vec{x}_1, \vec{x}_1) & \cdots & k(\vec{x}_1, \vec{x}_n) \\ \vdots & \ddots & \vdots \\ k(\vec{x}_n, \vec{x}_1) & \cdots & k(\vec{x}_n, \vec{x}_n) \end{pmatrix}$$

3 Proposed solution

We can now expose the Frank-Wolfe algorithm, as follows:

Algorithm 1 Frank-Wolfe method

```
1: procedure X = FWM($f, \beta, \gamma, C, \epsilon_{FW}$)
2:     while ($||\nabla f(\beta, \gamma)|| > \epsilon_{FW}$) do
3:         $\begin{pmatrix}\bar{z}_\beta \\ \bar{z}_\gamma\end{pmatrix} \leftarrow \left(\text{argmin}\left\{\langle\nabla f(\beta,\gamma),(z_\beta,z_\gamma)\rangle : \begin{matrix} -C \leq z_\beta \leq C \wedge \sum_{i=1}^{n} z_{\beta_i} = 0 \\ \forall i \in [1,n] : z_{\gamma_i} \geq z_{\beta_i} \wedge z_{\gamma_i} \geq -z_{\beta_i}\end{matrix}\right\}\right)$;
4:         $\begin{pmatrix}d_\beta \\ d_\gamma\end{pmatrix} \leftarrow \begin{pmatrix}\bar{z}_\beta - \beta \\ \bar{z}_\gamma - \gamma\end{pmatrix}$;
5:         $\alpha \leftarrow$ Line_Search($f, \beta, \gamma, d_\beta, d_\gamma, 1$);
6:         $\begin{pmatrix}\beta \\ \gamma\end{pmatrix} \leftarrow \begin{pmatrix}\beta + \alpha\ d_\beta \\ \gamma + \alpha\ d_\gamma\end{pmatrix}$;
```

3.1 *argmin* proposed solution - step 3

For solving the *argmin* in the step 3 of the Frank-Wolfe algorithm, we could compute the components of the gradient G, found in the following way:

$$\begin{pmatrix} z_\beta \\ z_\gamma \end{pmatrix} \leftarrow \left(\text{argmin} \left\{ \langle \nabla f(\beta,\gamma),(z_\beta,z_\gamma)\rangle : \begin{array}{c} -C \ \le \ z_\beta \ \le \ C \ \wedge \ \sum_{i=1}^n z_{\beta_i} \ = \ 0 \\ \forall i \in [1,n] \ : \ z_{\gamma_i} \ge z_{\beta_i} \ \wedge \ z_{\gamma_i} \ge -z_{\beta_i} \end{array} \right\} \right)$$

These constraints can be solved as a linear programming problem by viewing them as a vector in which the first n entries are the z_β and the second n entries are the z_γ, obtaining the concatenation of the z_β and z_γ, where n is the length of the vectors $z_\beta,\ z_\gamma$.

To avoid abuse of notation we use β and γ instead of z_β and z_γ.

So, the primal problem can be written as:

$$\min \epsilon \textstyle\sum_{i=1}^n \gamma_i + \sum_{i=1}^n g_i \ \beta_i$$

$$\begin{array}{rlll} z^+, z^-) & \gamma_i \ge \beta_i, & \gamma_i \ge -\beta_i, & \textbf{with } \gamma_i \ge 0 \\ w^+, w^-) & C \ge \beta_i, & C \ge -\beta_i & \\ \lambda) & \sum_{i=1}^n \beta_i = 0 & & \end{array} \tag{6}$$

and the dual problem, with complementary slackness, as:

$$\max C \textstyle\sum_{i=1}^n (w_i^+ + w_i^-)$$

$$\begin{array}{rll} a) & \multicolumn{2}{c}{\lambda + w_i^+ + z_i^+ - w_i^- - z_i^- = g_i} \\ b) & \multicolumn{2}{c}{-z_i^+ - z_i^- \le \epsilon} \\ & & \\ c,d) & \lambda \lesseqgtr 0, & z_i^+, z_i^-, w_i^+, w_i^- \le 0 \\ & & \\ e,f) & w_i^+(C-\beta_i) = 0 & w_i^-(C+\beta_i) = 0 \\ g,h) & z_i^+(\gamma_i - \beta_i) = 0 & z_i^-(\gamma_i + \beta_i) = 0 \end{array} \tag{7}$$

3.1.1 Discussion about optimal solution

We assume to have g vector sorted in ascending order.

From (7.e) and (7.f) we know that w_i^+ and w_i^- cannot be both $\neq 0$ and the one different from 0 determines β_i equals C or $-C$.

At the same way, from (7.g) and (7.h) we know that z_i^+ and z_i^- cannot be both $\neq 0$ and the one different from 0 determines $\gamma_i = \beta_i$ or $\gamma_i = -\beta_i$.

From (7.a) we obtain that $w_i^+ + z_i^+ - w_i^- - z_i^- = g_i - \lambda$. Since w_i^+ and w_i^- are both negative and are both in the objective function, to maximize it we need to take them as less as possible, using instead z_i^+ and z_i^- that are sufficient where $|g_i - \lambda| \le \epsilon$.

To solve the dual we can say that the most important step is to find a good value for λ.

Once chosen, it happens that it goes to fall in a point in the (sorted) sequence of g_i and, knowing that $\epsilon \ge 0$, determines four situation:

$$\begin{array}{lcccccc}
(a) & g_i - \lambda < -\epsilon & \Rightarrow & w_i^+ < 0, z_i^+ < 0 & \Rightarrow & \beta_i = C, \gamma_i = C \\
(b) & -\epsilon \le g_i - \lambda \le 0 & \Rightarrow & w_i^+ = 0, w_i^- = 0, z_i^+ < 0 & \Rightarrow & -C \le \beta_i \le C, \gamma_i = \beta_i \\
(c) & 0 \le g_i - \lambda \le \epsilon & \Rightarrow & w_i^+ = 0, w_i^- = 0, z_i^- < 0 & \Rightarrow & -C \le \beta_i \le C, \gamma_i = -\beta_i \\
(d) & g_i - \lambda > \epsilon & \Rightarrow & w_i^- < 0, z_i^- < 0 & \Rightarrow & \beta_i = -C, \gamma_i = C
\end{array} \tag{8}$$

Anyway, we have to look at the primal objective function. Since g is sorted, for each $C \cdot g_i < 0$ and then $-C \cdot g_{n-i} < 0$ taken we need to pay $2 \cdot C \cdot \epsilon$ which can become easily unsustainable.

The optimal solution of this problem is not so easy, thus we leave it as a future work and we use a general tool for linear programming.

3.1.2 General linear programming solution

The argmin problem can be solved using a general linear programming tool. Re-formulating (6), we obtain:

$$\begin{gathered}
\min \epsilon \textstyle\sum_{i=1}^n z_{\gamma_i} + \sum_{i=1}^n g_i \; z_{\beta_i} \\
z_{\beta_i} - z_{\gamma_i} \le 0, \quad -z_{\beta_i} - z_{\gamma_i} \le 0 \\
\textstyle\sum_{i=1}^n z_{\beta_i} = 0 \\
-C \le z_{\beta_i} \le C
\end{gathered} \tag{9}$$

we derive the parameter for `linprog()`:

$$\text{f}: \quad \left[\frac{\partial f(\beta,\gamma)}{\partial \beta}, \; \frac{\partial f(\beta,\gamma)}{\partial \gamma}\right]$$

$$\text{A}_{ub} \le b_{ub}: \quad \left(\begin{array}{ccc|ccc}
1 & & & -1 & & \\
 & \ddots & & & \ddots & \\
 & & 1 & & & -1 \\
\hline
-1 & & & -1 & & \\
 & \ddots & & & \ddots & \\
 & & -1 & & & -1
\end{array}\right) \le \left(\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline 0 \\ \vdots \\ 0 \end{array}\right)$$

$$\text{A}_{eq} = b_{eq}: \quad \left(\begin{array}{ccc|ccc} 1 & \cdots & 1 & 0 & \cdots & 0 \end{array}\right) = \left(0\right)$$

$$\text{bounds}: \quad \left(\begin{array}{ccc|ccc} (-C, C) & \cdots & (-C, C) & (-\infty, +\infty) & \cdots & (-\infty, +\infty) \end{array}\right)$$

3.1.3 Solution and bounding

Given z the result of the problem, we obtain:

- z_β as first half of z;
- z_γ as second half of z.

Then, we stabilize our result adding some box constraints to the master problem, such that $||(z_\beta, z_\gamma) - (\beta, \gamma)||_\infty < t$

The new point is chosen in the box of relative size t around the current point. The component z_i is allowed to change by no more than $\pm t$.
To obtain this new bounded point we need to solve a small different optimization problem, using linear programming anyway.

Re-formulating (9) as follows:

$$\begin{gathered}\min \epsilon \textstyle\sum_{i=1}^{n} z_{\gamma_i} + \sum_{i=1}^{n} g_i \ z_{\beta_i} \\[1ex] z_{\beta_i} - z_{\gamma_i} \le 0, \quad -z_{\beta_i} - z_{\gamma_i} \le 0 \\ \textstyle\sum_{i=1}^{n} z_{\beta_i} = 0 \\ -C \le z_{\beta_i} \le C \\ \beta - t \le z_{\beta_i} \le \beta + t \\ \gamma - t \le z_{\gamma_i} \le \gamma + t\end{gathered} \tag{10}$$

Where the derived parameter for `linprog()` are the same describer before except for:

$$\text{bounds}: \quad \begin{pmatrix} (\max(-C, \beta_1 - t), \min(C, \beta_1 + t)) & \cdots & (\max(-C, \beta_n - t), \min(C, \beta_n + t)) \\ (\gamma_1 - t, \gamma_1 + t) & \cdots & (\gamma_n - t, \gamma_n + t) \end{pmatrix}$$

Since in order to calculate the lower bound of the problem we must have an unrestricted solution, the bounding of the problem with parameter t, in any case requires us to solve two problems. So we have to:

- Solve unbounded problem (9)
- Compute lower bound
- Solve bounded problem (10)
- Compute line search

Anyway, if `t` $= 0$, then we use the non-stabilized version of the algorithm.

3.2 *line search* proposed solution - step 5

To obtain the optimal stepsize we have to solve the following problem

$$\min f(\beta + \alpha d_\beta, \gamma + \alpha d_\gamma)$$

by solving

$$\frac{\partial f}{\partial \alpha} = 0$$

$$\frac{\partial\ {}^1\!/\!_2(\beta + \alpha d_\beta)K(\beta + \alpha d_\beta) - y(\beta + \alpha d_\beta) + \epsilon(\gamma + \alpha d_\gamma)}{\partial \alpha} = 0$$

$$\frac{\partial {}^1\!/\!_2\ \alpha^2(d_\beta K d_\beta) + \alpha(d_\beta K \beta - d_\beta y + \epsilon d_\gamma)[+ \text{ const}]}{\partial \alpha} = 0$$

$$\Longrightarrow$$

$$\alpha = \frac{d_\beta(y - K\beta) - d_\gamma \epsilon}{d_\beta K d_\beta}$$

subject to $0 \leq \alpha \leq 1$.

4 SVR and Frank-Wolfe implementation

The software, written in python, consists of a unique file implementing an SVR using Frank-Wolfe for solving quadratic optimization.

The class support the following parameter:

`kernel` : SVR kernel. It supports **'rbf'**. Where the corresponding kernel function is:

$$\textbf{'rbf'} : \exp^{\texttt{gamma}\cdot|x_1 - x_2|^2}$$

`degree` : parameter for polynomial kernel only

`gamma` : parameter for sigmoid and rbf kernel only

`coef0` : parameter for polynomial and sigmoid kernel only

`C` : SVR data-point tolerance

`epsilon` : SVR epsilon tube

`verbose` : True or False for info printing

`tol` : Frank-Wolfe solution tolerance

`max-iter` : Frank-Wolfe maximum number of iteration

`t` : Frank-Wolfe ball regularization

`kernel-matrix` : precomputed kernel matrix

The class follows the scikit-learn BaseEstimator interface in order to use an automatic Grid Search Cross Validation function for parameter estimation. So it implements:

fit : In order to fit the model it:

- Computes the kernel matrix according to the chosen kernel function;
- Solves the quadratic optimization in order to find the beta multiplier;
- Obtains the bias according to the formula defined in subsection 2.1, from average of support vectors with interpolation error `e`.
  This means that the alphas related to the support vectors with interpolation error `e` range in: $0 < \alpha < C$.

predict : It returns predicted y value from a test set, using the beta and the bias previously found. That is:

$$y = \sum_{i=1}^{TR_{size}} \beta_i K(TR - X_i, X) + b$$

and this is repeated for all test set.

An SVR produces an unique output. To use multiple outputs we need to train multiple SVRs.

score : Giving a test set and a trained model, it predicts the TR and compares to true value using the Mean Euclidean Error function.

$$MEE = \frac{1}{n} \sum_{i=1}^{n} \sqrt{(y_{true_x} - y_{pred_x})^2 + (y_{true_y} - y_{pred_y})^2}$$

4.1 Frank-Wolfe

The algorithm works iteratively, until one of the following happens. It's guaranteed to terminate (since iteration counter is always updated) whether:

$gap < \epsilon_{FW}$: that is, whether the relative error in the solution, obtained as $\frac{f(x^i) - f_*}{|f_x|}$, is less than a threshold. Where f_* is a first order approximation and is updated at each iteration as the value of the lower bound l_k:

$$l_k := \max\left(l_{k-1},\ f(\beta_k) + \nabla f(\beta_k)(z_k - \beta_k)\right)$$

$iteration > max\_iter$: that is, after n iterations.

The pseudo-code related to this discussion is shown below:

```
while True:
    v = 0.5 * beta * K * beta - y * beta + eps * gamma
    g_beta = K * beta - y; g_gamma = eps;

    # Step 3 unbounded
    # Solve first problem to compute lower bound
    c = concatenate((g_beta, g_gamma))
    res = linprog(c, A_ub, b_ub, A_eq, b_eq, bound)
    z_beta = res['x'][0:n]; z_gamma = res['x'][n:2n]

    # Step 4 unbounded
    d_beta = z_beta - beta
    d_gamma = z_gamma - gamma

    # Compute lower bound
    lb = v + g_beta * d_beta + g_gamma * d_gamma
    best_lb = lb if lb > best_lb else best_lb
    gap = (v - best_lb) / max([abs(v), 1])

    if gap <= eps_fw return beta

    # Step 3 bounded
    # Solve second problem to compute stabilized solution
    if t > 0:
        res = linprog(c, A_ub, b_ub, A_eq, b_eq, T_bound)
        z_beta = res['x'][0:n]; z_gamma = res['x'][n:2n]

        # Step 4 bounded
        d_beta = z_beta - beta
        d_gamma = z_gamma - gamma

    # Step 5
    num = -g_beta * d_beta + -g_gamma * d_gamma
    den = d_beta * K * d_beta
    alpha = max(0.0, min(1.0, num / den))

    # Step 6
    beta += alpha * d_beta
    gamma += alpha * d_gamma

    iteration += 1
    if iteration > max_iter return beta
```

In the following it can be seen a simple example.

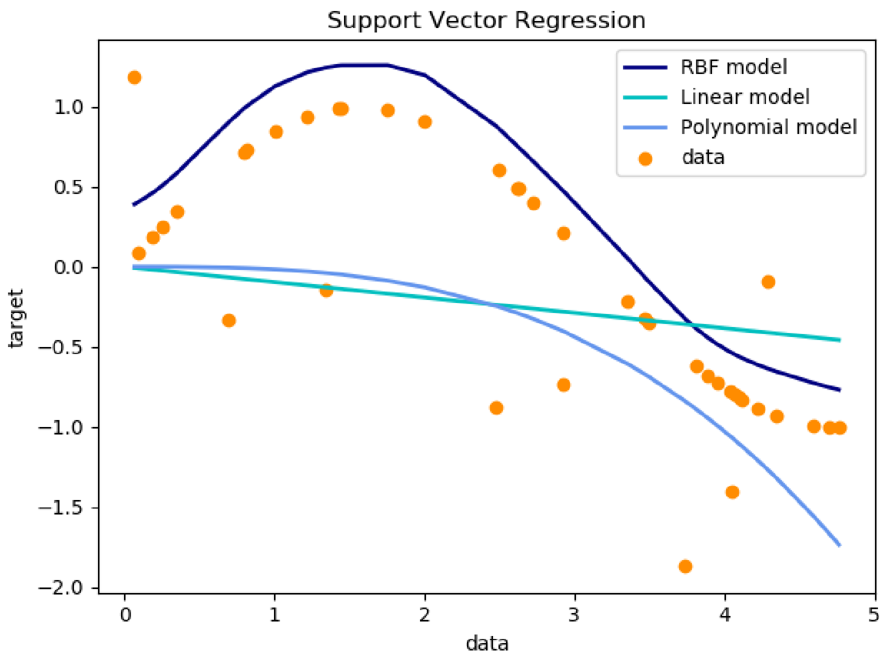


Figure 1: Random sin-like 2D point

4.2 Experiments

For the experiments we decide to keep fixed:

`kernel`='rbf' : Because we don't have previous knowledge on the data and we want to keep this function as general as possible;

`tol`=$1e-6$, `max-iter`=50 : Because we don't want hyperparameter that converge too slowly, so `max-iter` is not so large and `tol` is quite low (does not have to be a real limit).

And we look for:

`gamma` : Parameter for the kernel function;

`C` : SVR data-point tolerance;

`epsilon` : SVR epsilon tube;

`t` : Frank-Wolfe ball regularization.

Where only `t` regards to the Frank-Wolfe algorithm; the other ones are related to the SVR.

4.3 Configuration

All the experiments run on a desktop configuration with:

CPU : i5-3470, 4 core at 3.2Ghz

RAM : 16GB DDR3 1867Mhz

Python : 3.7.2

Windows : 10

4.4 Dataset

We use ML-CUP dataset, made of 1016 record with 10 features per record and 2 expected output per record.

Since SVR expect only 1 output, for each y, from now y_0 and y_1, we train a different SVR.

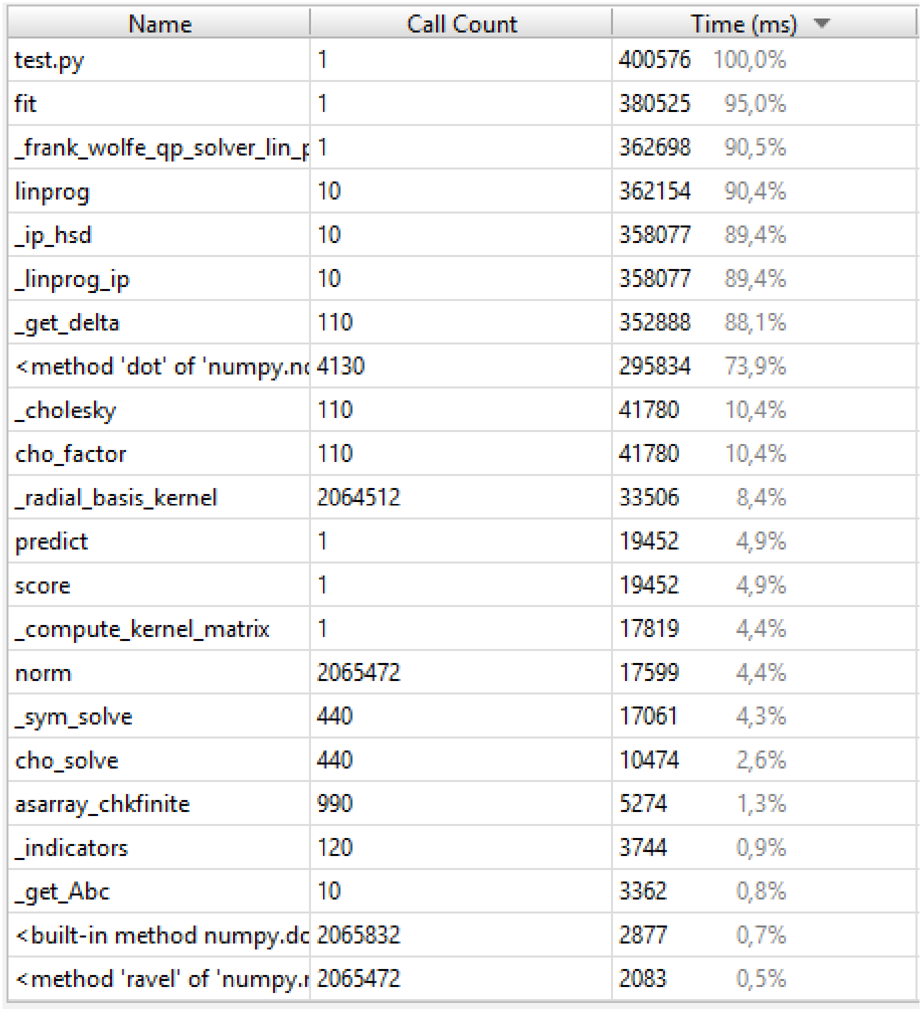
We start from a generic grid search, then we tune it by analyzing first result:

- `C` $\in [10, 1, 0.1]$,
- `epsilon` $\in [1, 0.1, 0.01]$,
- `t` $\in [1, 0.1, 0.01]$,
- `gamma` $\in [1, 0.1, 0.01, 0.001]$.

4.5 Tests and results

First grid search was made using **scikit-learn**, since using `linprog()` in our code is very expensive for an extensive search.

As can be seen from the figure (2), the 90% of the time is taken by `linprog()`, and the 5% (but this time is independent from number of iterations) from computing the kernel matrix, which cannot be easily parallelized because it is made of a very high number of sub-computations of small size.



Name	Call Count	Time (ms)
test.py	1	400576 100,0%
fit	1	380525 95,0%
_frank_wolfe_qp_solver_lin_p	1	362698 90,5%
linprog	10	362154 90,4%
_ip_hsd	10	358077 89,4%
_linprog_ip	10	358077 89,4%
_get_delta	110	352888 88,1%
<method 'dot' of 'numpy.nc	4130	295834 73,9%
_cholesky	110	41780 10,4%
cho_factor	110	41780 10,4%
_radial_basis_kernel	2064512	33506 8,4%
predict	1	19452 4,9%
score	1	19452 4,9%
_compute_kernel_matrix	1	17819 4,4%
norm	2065472	17599 4,4%
_sym_solve	440	17061 4,3%
cho_solve	440	10474 2,6%
asarray_chkfinite	990	5274 1,3%
_indicators	120	3744 0,9%
_get_Abc	10	3362 0,8%
<built-in method numpy.dc	2065832	2877 0,7%
<method 'ravel' of 'numpy.r	2065472	2083 0,5%

Figure 2: Simple test with 1016 record and 10 iterations

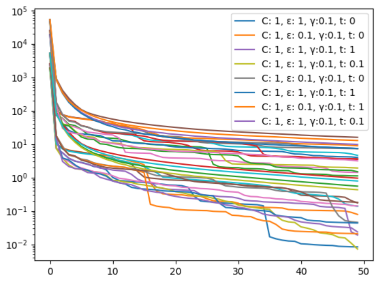
4.5.1 ML Tests

From the generic grid search made using scikit-learn, we select the top 4 parameters and then we train our model on that by varying `t`.
Scikit-learn obtains an `r2` score of 0.9 when with the same value we obtain -3.
In the following, the scores are computed using MEE.

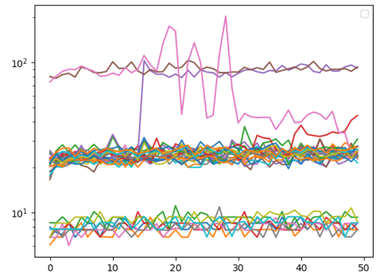
Score	**Gap**	**C**	ϵ	γ	**t**
8.559230987778388	0.008385274777684915	1	1	0.1	0
9.248012862083325	0.07837669998874275	1	0.1	0.1	0
10.109655627940716	1.1171385863891585	10	1	0.1	0
9.968845049768545	3.6241706986010436	10	0.1	0.1	0
8.757533807816706	0.018990123543460604	1	1	0.1	1
9.06977043174573	0.17276801409006773	1	0.1	0.1	1
9.53438333945378	1.4171691654691962	10	1	0.1	1
9.483423663601183	7.259849851482331	10	0.1	0.1	1
8.554145459594745	0.007235326158265439	1	1	0.1	0.1
9.06511124013254	0.13774986804360043	1	0.1	0.1	0.1
8.993612595143828	3.978983009477025	10	1	0.1	0.1
9.203217368862276	8.879937858101105	10	0.1	0.1	0.1
8.33594719332409	0.5490592241403589	1	1	0.1	0.01
8.6471403671181	0.953344402087729	1	0.1	0.1	0.01
8.33594719332409	9.760861973365321	10	1	0.1	0.01
8.6471403671181	15.92993600626834	10	0.1	0.1	0.01
16.659454321703304	0.14077334027119282	1	1	0.1	0
16.775212136084566	0.045577839108495125	1	0.1	0.1	0
16.723400786509846	1.9199612890320519	10	1	0.1	0
16.577004126667195	4.592407685060145	10	0.1	0.1	0
16.652998951957933	0.043473288267837036	1	1	0.1	1
16.77501173127357	0.020827660038806685	1	0.1	0.1	1
16.69611970223087	1.4903412111572354	10	1	0.1	1
16.936439728565716	3.506052486812343	10	0.1	0.1	1
16.47399210012951	0.023499770300537362	1	1	0.1	0.1
16.658112090221366	0.18036871564315982	1	0.1	0.1	0.1
16.53634581897853	3.1388951312551914	10	1	0.1	0.1
16.668891356896438	7.35917170879379	10	0.1	0.1	0.1
16.590670846293552	0.4303325061149694	1	1	0.1	0.01
16.553874422072447	0.7849443215513666	1	0.1	0.1	0.01
16.590670846293552	7.353071056023913	10	1	0.1	0.01
16.553874422072447	12.77397296336014	10	0.1	0.1	0.01

Scikit-learn give us more accuracy and is very fast. Times are computer in the following. We can say that a single train require from 10 to 20 minutes with a small accuracy.

4.5.2 Frank-Wolfe Tests

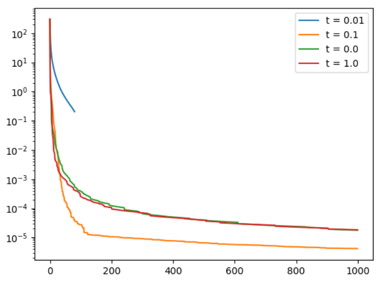


(a) Gap descending with different configuration, showed when gap < 0.1

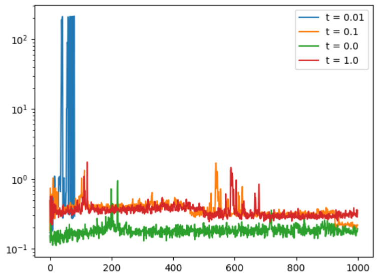


(b) Iteration time

Figure 3: Full ML-CUP dataset experiments



(a) Gap descending with same configuration: C=1, ϵ=0.1, γ=0.1, t=0



(b) Iteration time

Figure 4: First 100 elements ML-CUP dataset experiments

In all figures there are iteration counter on x axis and gap (figures 3a and 4a), or time (figures 3b and 4b), on y axis.

Frank-Wolfe is monotone, as can be seen in figure (3a), since the gap can only decrease. At the beginning it starts with a very high gap since initial point is 0 for all experiments, but in the first iteration it goes down quickly until it stabilizes and decreases more or less slowly, depending on `t` parameter.

In the small experiments, figure (4a), we obtain a good accuracy, reaching a 9.99515e-06 gap after 234 iterations and a final one of 4.19621e-06. Experiments with `t` $= 0.01$ was interrupted early because of the slow gap decrease and because of its very high completion time [figure (4b)].

The completion time of each iteration may vary a lot. In figure (3b) we obtain a very high completion time for the whole computation in two cases, and for a part of it in just one case. Moreover the two big ranges in the figure represent experiments with and without stabilization, because the two `linprog()` in the code double the execution time. We can also say that the completion time can be very large because simplex algorithm is exponential at worst case, as it can be seen with the blue line in figure (4b).

Using the stabilization sometime may be useless, since the final accuracy is not influenced [figure (4a), `t` $= 0$ and `t` $= 1$] but the time doubles [figure (4b)].

y	C	ϵ	γ	Frank-Wolfe		quadprog	
				score (min,max)	time (sec) (min,max)	score	time (sec)
0	1	0.1	0.1	8.64 - 9.24	381 - 1211	9.20	50
0	1	1	0.1	8.33 - 8.75	409 - 1235	8.58	49
0	10	0.1	0.1	8.64 - 9.96	405 - 1244	10.58	54
0	10	1	0.1	8.33 - 10.10	453 - 1298	9.73	52
1	1	0.1	0.1	16.55 - 16.77	489 - 4493	16.75	50
1	1	1	0.1	16.47 - 16.65	394 - 3436	16.58	52
1	10	0.1	0.1	16.55 - 16.93	421 - 1398	17.47	55
1	10	1	0.1	16.59 - 16.72	460 - 3704	16.89	55

Table 1: Comparison between Frank-Wolfe and generic QP solver

As we can see in a final comparison with the `quadprog()` in matlab, we achieve a very similar result but with a very slow time.

5 Conclusions

We implement an SVR model using Frank-Wolfe for the resolution of the optimization problem. The SVR works quite well giving us a good score, but it's very slow because of the linear programming problem that has to be solved at each iteration of Frank-Wolfe which represents a very big cost for the whole computation.

References

[1] https://en.wikipedia.org/wiki/Support-vector_machine#Regression

[2] https://en.wikipedia.org/wiki/Frank-Wolfe_algorithm