University of Pisa

Department of Computer Science

Computational Mathematics
Wildcard Project nr. 5 with Machine Learning
Group 35

Support Vector Machines

Author:

Donato Meoli
d.meoli@studenti.unipi.it

May, 2021

Contents

List of Figures

List of Tables

List of Algorithms

List of Theorems

1 Track

`(M1.1)` is a *Support Vector Classifier (SVC)* with the *hinge* loss.

- `(A1.1.1)` is a *momentum descent* approach [1, 2, 3], an *accelerated gradient* method for solving the SVC in its *primal* formulation.
- `(A1.1.2)` is the *Sequential Minimal Optimization (SMO)* algorithm [4, 5], an ad hoc *active set* method for training a SVC in its *Wolfe dual* formulation with *linear*, *polynomial* and *gaussian* kernels.
- `(A1.1.3)` is the *AdaGrad* algorithm [6], a *deflected subgradient* method for solving the SVC in its *Lagrangian dual* formulation with *linear*, *polynomial* and *gaussian* kernels.

`(M1.2)` is a *Support Vector Classifier (SVC)* with the *squared hinge* loss.

- `(A1.2.1)` is a *momentum descent* approach [1, 2, 3], an *accelerated gradient* method for solving the SVC in its *primal* formulation.

`(M2.1)` is a *Support Vector Regression (SVR)* with the *epsilon-insensitive* loss.

- `(A2.1.1)` is a *momentum descent* approach [1, 2, 3], an *accelerated gradient* method for solving the SVR in its *primal* formulation.
- `(A2.1.2)` is the *Sequential Minimal Optimization (SMO)* algorithm [7, 8], an ad hoc *active set* method for training a SVR in its *Wolfe dual* formulation with *linear*, *polynomial* and *gaussian* kernels.
- `(A2.1.3)` is the *AdaGrad* algorithm [6], a *deflected subgradient* method for solving the SVR in its *Lagrangian dual* formulation with *linear*, *polynomial* and *gaussian* kernels.

`(M2.2)` is a *Support Vector Regression (SVR)* with the *squared epsilon-insensitive* loss.

- `(A2.2.1)` is a *momentum descent* approach [1, 2, 3], an *accelerated gradient* method for solving the SVR in its *primal* formulation.

2 Abstract

A *Support Vector Machine* is a learning model used both for *classification* and *regression* tasks whose goal is to construct a *maximum margin separator*, i.e., a decision boundary with the largest distance from the nearest training data points.

The aim of this report is to compare the *primal*, the *Wolfe dual* [9] and the *Lagrangian dual* formulations of this model in terms of *numerical precision*, *accuracy* and *complexity*.

Firstly, I will provide a detailed mathematical derivation of the model for all these formulations, then I will propose two algorithms to solve the optimization problem in case of *constrained* or *unconstrained* formulation of the problem, explaining their theoretical properties, i.e., *convergence* and *complexity*.

Finally, I will show some experiments for *linearly* and *nonlinearly* separable generated datasets to compare the performance of different *kernels*, also by comparing the *custom* results with *sklearn* SVM implementations, i.e., *liblinear* [10] and *libsvm* [11] implementations, and *cvxopt* [12] QP solver.

3 Linear Support Vector Classifier

Given n training points, where each input x_i has m attributes, i.e., is of dimensionality m, and is in one of two classes $y_i = \pm 1$, i.e., our training data is of the form:

$$\{(x_i, y_i), x_i \in \Re^m, y_i = \pm 1, i = 1, \dots, n\} \tag{1}$$

For simplicity we first assume that data are (not fully) linearly separable in the input space x, meaning that we can draw a line separating the two classes when $m = 2$, a plane for $m = 3$ and, more in general, a hyperplane for an arbitrary m.

Support vectors are the examples closest to the separating hyperplane and the aim of support vector machines is to orientate this hyperplane in such a way as to be as far as possible from the closest members of both classes, i.e., we need to maximize this margin.

This hyperplane is represented by the equation $w^T x + b = 0$. So, we need to find w and b so that our training data can be described by:

$$\begin{aligned} & w^T x_i + b \geq +1 - \xi_i, \forall y_i = +1 \\ & w^T x_i + b \leq -1 + \xi_i, \forall y_i = -1 \\ & \xi_i \geq 0 \ \forall_i \end{aligned} \tag{2}$$

where the positive slack variables ξ_i are introduced to allow misclassified points. In this way data points on the incorrect side of the margin boundary will have a penalty that increases with the distance from it.

These two equations can be combined into:

$$\begin{aligned} & y_i(w^T x_i + b) \geq 1 - \xi_i \ \forall_i \\ & \xi_i \geq 0 \ \forall_i \end{aligned} \tag{3}$$

The margin is equal to $\dfrac{1}{\|w\|}$ and maximizing it subject to the constraint in (3) while as we are trying to reduce the number of misclassifications is equivalent to finding:

$$\begin{aligned} \min_{w,b,\xi} \quad & \|w\| + C \sum_{i=1}^{n} \xi_i \\ \text{subject to} \quad & y_i(w^T x_i + b) \geq 1 - \xi_i \ \forall_i \\ & \xi_i \geq 0 \ \forall_i \end{aligned} \tag{4}$$

Minimizing $\|w\|$ is equivalent to minimizing $\dfrac{1}{2}\|w\|^2$, but in this form we will deal with a 1-strongly convex regularization term that has more desirable convergence properties. So we need to find:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \xi_i \\ \text{subject to} \quad & y_i(w^T x_i + b) \geq 1 - \xi_i \ \forall_i \\ & \xi_i \geq 0 \ \forall_i \end{aligned} \tag{5}$$

where the parameter C controls the trade-off between the slack variable penalty and the size of the margin.

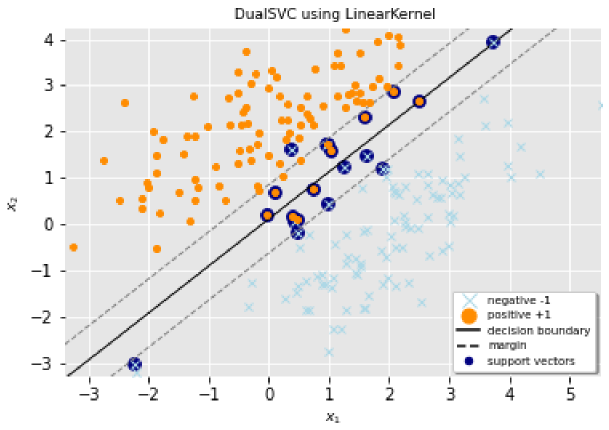


Figure 1: Linear SVC hyperplane

3.1 Hinge loss

The *hinge* loss is defined as:

$$\mathcal{L}_1 = \begin{cases} 0 & \text{if } y(w^T x + b) \geq 1 \\ 1 - y(w^T x + b) & \text{otherwise} \end{cases} \tag{6}$$

or, equivalently:

$$\mathcal{L}_1 = \max(0, 1 - y(w^T x + b)) \tag{7}$$

and it is a nondifferentiable convex function due to its nonsmoothness in 1, but has a subgradient wrt w that is given by:

$$\frac{\partial \mathcal{L}_1}{\partial w} = \begin{cases} -yx & \text{if } y(w^T x + b) < 1 \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

3.1.1 Primal formulation

The general primal unconstrained formulation takes the form:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \mathcal{L}(w, b; x_i, y_i) \tag{9}$$

where $\frac{1}{2}\|w\|^2$ is the *regularization term* and $\mathcal{L}(w, b; x_i, y_i)$ is the *loss function* associated with the observation (x_i, y_i) [13].

The quadratic optimization problem (5) can be equivalently formulated as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b)) \tag{10}$$

where we make use of the *hinge* loss (6) or (7).

The above formulation penalizes slacks ξ linearly and is called $\mathcal{L}_1$-SVC.

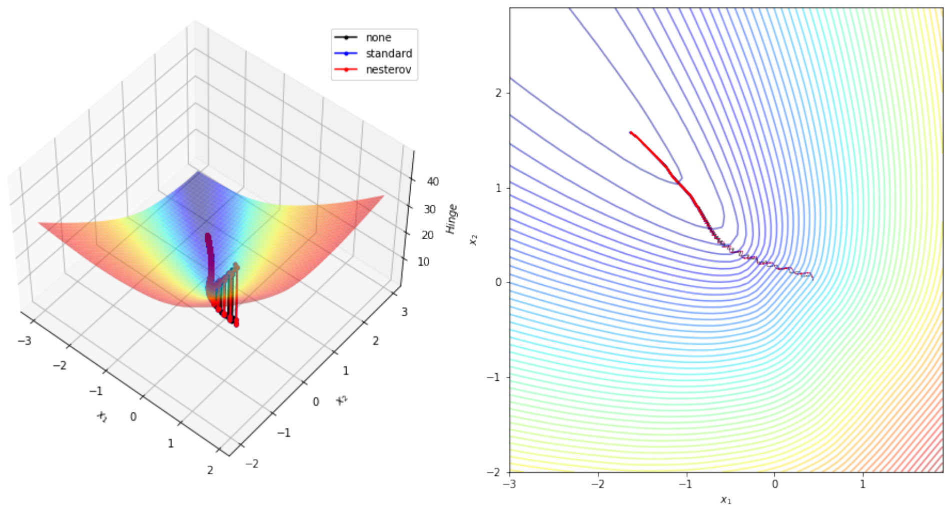


Figure 2: SVC Hinge loss with different optimization steps

To simplify the notation and so also the design of the algorithms, the simplest approach to learn the bias term b is that of including that into the *regularization term*; so we can rewrite (9) as follows:

$$\min_{w,b} \frac{1}{2}(\|w\|^2 + b^2) + C \sum_{i=1}^{n} \mathcal{L}(w, b; x_i, y_i) \tag{11}$$

or, equivalently, by augmenting the weight vector w with the bias term b and each instance x_i with an additional dimension, i.e., with constant value equal to 1:

$$\begin{aligned} \min_{w} \quad & \frac{1}{2}\|\bar{w}\|^2 + C \sum_{i=1}^{n} \mathcal{L}(\bar{w}; \bar{x}_i, y_i) \\ \text{where} \quad & \bar{w}^T = [w^T, b] \\ & \bar{x}_i^T = [x_i^T, 1] \end{aligned} \tag{12}$$

with the advantages of having convex properties of the objective function useful for convergence analysis and the possibility to directly apply algorithms designed for models without the bias term.

In the specific case of the $\mathcal{L}_1$-SVC the objective (10) become:

$$\min_{w,b} \frac{1}{2}(\|w\|^2 + b^2) + C \sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b)) \tag{13}$$

Notice that in terms of numerical optimization the formulation (10) is not equivalent to (13) since in the first one the bias term b does not contribute to the *regularization term*, so the SVM formulation is based on an unregularized bias term b, as highlighted by the *statistical learning theory*. But, in machine learning sense, numerical experiments in [14] show that the accuracy does not vary much when the bias term b is embedded into the weight vector w.

3.1.2 Wolfe Dual formulation

To reformulate the (5) as a *Wolfe dual*, we need to allocate the Lagrange multipliers $\alpha_i \geq 0, \mu_i \geq 0 \ \forall_i$:

$$\max_{\alpha,\mu} \min_{w,b,\xi} \mathcal{W}(w,b,\xi,\alpha,\mu) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^{n}\mu_i\xi_i \tag{14}$$

We wish to find the w, b and ξ_i which minimizes, and the α and μ which maximizes $\mathcal{W}$, provided $\alpha_i \geq 0, \mu_i \geq 0 \ \forall_i$. We can do this by differentiating $\mathcal{W}$ wrt w and b and setting the derivatives to 0:

$$\frac{\partial \mathcal{W}}{\partial w} = w - \sum_{i=1}^{n}\alpha_i y_i x_i \Rightarrow w = \sum_{i=1}^{n}\alpha_i y_i x_i \tag{15}$$

$$\frac{\partial \mathcal{W}}{\partial b} = -\sum_{i=1}^{n}\alpha_i y_i \Rightarrow \sum_{i=1}^{n}\alpha_i y_i = 0 \tag{16}$$

$$\frac{\partial \mathcal{W}}{\partial \xi_i} = 0 \Rightarrow C = \alpha_i + \mu_i \tag{17}$$

Substituting (15) and (16) into (14) together with $\mu_i \geq 0 \ \forall_i$, which implies that $\alpha \leq C$, gives a new formulation being dependent on α. We therefore need to find:

$$\begin{aligned}\max_{\alpha} \mathcal{W}(\alpha) &= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \langle x_i, x_j\rangle \\ &= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i Q_{ij}\alpha_j \text{ where } Q_{ij} = y_i y_j \langle x_i, x_j\rangle \\ &= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\alpha^T Q\alpha \text{ subject to } 0 \leq \alpha_i \leq C \ \forall_i, \sum_{i=1}^{n}\alpha_i y_i = 0\end{aligned} \tag{18}$$

or, equivalently:

$$\begin{aligned}\min_{\alpha} \quad & \frac{1}{2}\alpha^T Q\alpha + q^T\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \\ & y^T\alpha = 0\end{aligned} \tag{19}$$

where $q^T = [1, \dots, 1]$.
By solving (19) we will know α and, from (15), we will get w, so we need to calculate b.
We know that any data point satisfying (16) which is a support vector x_s will have the form:

$$y_s(w^T x_s + b) = 1 \tag{20}$$

and, by substituting in (15), we get:

$$y_s\big(\sum_{m\in S}\alpha_m y_m\langle x_m, x_s\rangle + b\big) = 1 \tag{21}$$

where s denotes the set of indices of the support vectors and is determined by finding the indices i where $\alpha_i > 0$, i.e., nonzero Lagrange multipliers.
Multiplying through by y_s and then using $y_s^2 = 1$ from (2):

$$y_s^2\big(\sum_{m\in S}\alpha_m y_m\langle x_m, x_s\rangle + b\big) = y_s \tag{22}$$

$$b = y_s - \sum_{m \in S} \alpha_m y_m \langle x_m, x_s \rangle \tag{23}$$

Instead of using an arbitrary support vector x_s, it is better to take an average over all of the support vectors in S:

$$b = \frac{1}{N_s} \sum_{s \in S} y_s - \sum_{m \in S} \alpha_m y_m \langle x_m, x_s \rangle \tag{24}$$

We now have the variables w and b that define our separating hyperplane's optimal orientation and hence our support vector machine. Each new point x' is classified by evaluating:

$$y' = \operatorname{sgn} \big(\sum_{i=1}^{n} \alpha_i y_i \langle x_i, x' \rangle + b \big) \tag{25}$$

From (19) we can notice that the equality constraint $y^T \alpha = 0$ arises form the stationarity condition $\partial_b \mathcal{W} = 0$. So, again, for simplicity, we can again consider the bias term b embedded into the weight vector. We report below the box-constrained dual formulation [14] that arises from the primal (11) or (12) where the bias term b is embedded into the weight vector w:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \alpha^T (Q + yy^T) \alpha + q^T \alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \end{aligned} \tag{26}$$

3.1.3 Lagrangian Dual formulation

In order to relax the constraints in the *Wolfe dual* formulation (19) we define the problem as a *Lagrangian dual* relaxation by embedding them into objective function, so we need to allocate the Lagrangian multipliers $\mu \geq 0, \lambda_+ \geq 0, \lambda_- \geq 0$:

$$\begin{aligned} \max_{\mu, \lambda_+, \lambda_-} \min_{\alpha} \mathcal{L}(\alpha, \mu, \lambda_+, \lambda_-) &= \frac{1}{2} \alpha^T Q \alpha + q^T \alpha - \mu^T (y^T \alpha) - \lambda_+^T (u - \alpha) - \lambda_-^T \alpha \\ &= \frac{1}{2} \alpha^T Q \alpha + (q - \mu y + \lambda_+ - \lambda_-)^T \alpha - \lambda_+^T u \end{aligned} \tag{27}$$

where the upper bound $u^T = [C, \dots, C]$.

Taking the derivative of the Lagrangian $\mathcal{L}$ wrt α and settings it to 0 gives:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow Q\alpha + (q - \mu y + \lambda_+ - \lambda_-) = 0 \tag{28}$$

With α optimal solution of the linear system:

$$Q\alpha = -(q - \mu y + \lambda_+ - \lambda_-) \tag{29}$$

the gradient wrt μ, λ_+ and λ_- are:

$$\frac{\partial \mathcal{L}}{\partial \mu} = -y\alpha \tag{30}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_+} = \alpha - u \tag{31}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_-} = -\alpha \tag{32}$$

If the Hessian matrix Q is not positive definite, i.e., the Lagrangian function is not strictly convex since it will be linear along the eigenvectors correspondent to the null eigenvalues and so it will be unbounded below, the Lagrangian dual relaxation will be nondifferentiable, so it will have infinite solutions and for each of them it will have a different subgradient. In order to compute an approximation of the gradient, we will choose α in such a way as the one that minimizes the norm of the residual:

$$\begin{aligned} \min_{\alpha_n \in K_n(Q,b)} \quad & \|Q\alpha_n - b\| \\ \text{where} \quad & b = -(q - \mu y + \lambda_+ - \lambda_-) \end{aligned} \tag{33}$$

Since we are dealing with a symmetric but indefinite linear system we will choose a well-known Krylov method that performs the Lanczos iterate, i.e., symmetric Arnoldi iterate, called *minres*, i.e., symmetric *gmres*, to compute the vector α_n that minimizes the norm of the residual $r_n = Q\alpha_n - b$ among all vectors in $K_n(Q,b) = \mathit{span}(b, Qb, Q^2b, \ldots, Q^{n-1}b)$.

From (19) we can notice that the equality constraint $y^T\alpha = 0$ arises form the stationarity condition $\partial_b \mathcal{W} = 0$. So, again, for simplicity, we can again consider the bias term b embedded into the weight vector. In this way the dimensionality of (27) is reduced of 1/3 by removing the multipliers μ which was allocated to control the equality constraint $y^T\alpha = 0$, so we will end up solving exactly the problem (26).

$$\begin{aligned} \max_{\lambda_+, \lambda_-} \min_{\alpha} \mathcal{L}(\alpha, \lambda_+, \lambda_-) &= \frac{1}{2}\alpha^T(Q + yy^T)\alpha + q^T\alpha - \lambda_+^T(u - \alpha) - \lambda_-^T\alpha \\ &= \frac{1}{2}\alpha^T(Q + yy^T)\alpha + (q + \lambda_+ - \lambda_-)^T\alpha - \lambda_+^T u \end{aligned} \tag{34}$$

where, again, the upper bound $u^T = [C, \ldots, C]$.
Now, taking the derivative of the Lagrangian $\mathcal{L}$ wrt α and settings it to 0 gives:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow (Q + yy^T)\alpha + (q + \lambda_+ - \lambda_-) = 0 \tag{35}$$

With α optimal solution of the linear system:

$$(Q + yy^T)\alpha = -(q + \lambda_+ - \lambda_-) \tag{36}$$

the gradient wrt λ_+ and λ_- are:

$$\frac{\partial \mathcal{L}}{\partial \lambda_+} = \alpha - u \tag{37}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_-} = -\alpha \tag{38}$$

3.2 Squared Hinge loss

The *squared hinge* loss is defined as:

$$\mathcal{L}_2 = \begin{cases} 0 & \text{if } y(w^T x + b) \geq 1 \\ (1 - y(w^T x + b))^2 & \text{otherwise} \end{cases} \tag{39}$$

or, equivalently:

$$\mathcal{L}_2 = \max(0, 1 - y(w^T x + b))^2 \tag{40}$$

It is a strictly convex function and its gradient wrt w is given by:

$$\frac{\partial \mathcal{L}_2}{\partial w} = \begin{cases} -2yx & \text{if } y(w^T x + b) < 1 \\ 0 & \text{otherwise} \end{cases} \tag{41}$$

3.2.1 Primal formulation

Since smoothed versions of objective functions may be preferred for optimization, we can reformulate (10) as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b))^2 \tag{42}$$

where we make use of the *squared hinge* loss that quadratically penalized slacks ξ and is called $\mathcal{L}_2$-SVC. The $\mathcal{L}_2$-SVC objective (42) can be rewritten in form (11) or (12) as:

$$\min_{w,b} \frac{1}{2}(\|w\|^2 + b^2) + C\sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b))^2 \tag{43}$$

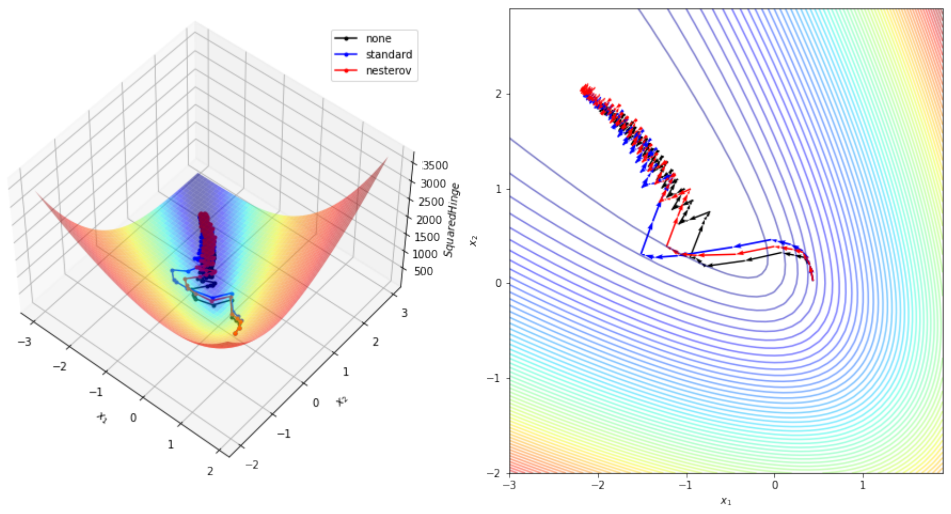


Figure 3: SVC Squared Hinge loss with different optimization steps

3.2.2 Wolfe Dual formulation

In the same way we can derive the dual form of the $\mathcal{L}_2$-SVC by obtaining:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T(Q+D)\alpha + q^T\alpha \\ \text{subject to} \quad & \alpha_i \geq 0 \ \forall_i \\ & y^T\alpha = 0 \end{aligned} \tag{44}$$

or, alternatively, with the regularized bias term by obtaining:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T(Q+yy^T+D)\alpha + q^T\alpha \\ \text{subject to} \quad & \alpha_i \geq 0 \ \forall_i \end{aligned} \tag{45}$$

where the diagonal matrix $D_{ii} = \dfrac{1}{2C} \ \forall_i$.

3.2.3 Lagrangian Dual formulation

In order to relax the constraints in the $\mathcal{L}_2$-SVC *Wolfe dual* formulation (44) we define the problem as a *Lagrangian dual* relaxation by embedding them into objective function, so we need to allocate the Lagrangian multipliers $\mu \geq 0, \lambda \geq 0$:

$$\begin{aligned} \max_{\mu,\lambda} \min_{\alpha} \mathcal{L}(\alpha,\mu,\lambda) &= \frac{1}{2}\alpha^T(Q+D)\alpha + q^T\alpha - \mu^T(y^T\alpha) - \lambda^T\alpha \\ &= \frac{1}{2}\alpha^T(Q+D)\alpha + (q-\mu y-\lambda)^T\alpha \end{aligned} \tag{46}$$

Taking the derivative of the Lagrangian $\mathcal{L}$ wrt α and settings it to 0 gives:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow (Q+D)\alpha + (q-\mu y-\lambda) = 0 \tag{47}$$

With α optimal solution of the linear system:

$$Q\alpha = -(q-\mu y-\lambda) \tag{48}$$

the gradient wrt μ and λ are:

$$\frac{\partial \mathcal{L}}{\partial \mu} = -y\alpha \tag{49}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = -\alpha \tag{50}$$

4 Linear Support Vector Regression

In the case of regression the goal is to predict a real-valued output for y' so that our training data is of the form:

$$\{(x_i, y_i), x \in \Re^m, y_i \in \Re, i = 1, \ldots, n\} \tag{51}$$

The regression SVM use a loss function that not allocating a penalty if the predicted value y'_i is less than a distance ϵ away from the actual value y_i, i.e., if $|y_i - y'_i| \leq \epsilon$, where $y'_i = w^T x_i + b$. The region bound by $y'_i \pm \epsilon \; \forall_i$ is called an ϵ-insensitive tube. The output variables which are outside the tube are given one of two slack variable penalties depending on whether they lie above, ξ^+, or below, ξ^-, the tube, provided $\xi^+ \geq 0$ and $\xi^- \geq 0 \; \forall_i$:

$$\begin{aligned} &y_i \leq y'_i + \epsilon + \xi^+ \; \forall_i \\ &y_i \geq y'_i - \epsilon - \xi^- \; \forall_i \\ &\xi_i^+, \xi_i^- \geq 0 \; \forall_i \end{aligned} \tag{52}$$

The objective function for SVR can then be written as:

$$\begin{aligned} \min_{w,b,\xi^+,\xi^-} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i^+ + \xi_i^-) \\ \text{subject to} \quad & y_i - w^T x_i - b \leq \epsilon + \xi_i^+ \; \forall_i \\ & w^T x_i + b - y_i \leq \epsilon + \xi_i^- \; \forall_i \\ & \xi_i^+, \xi_i^- \geq 0 \; \forall_i \end{aligned} \tag{53}$$

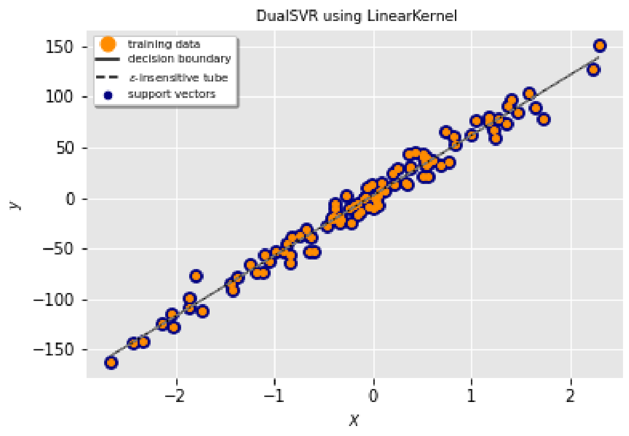


Figure 4: Linear SVR hyperplane

4.1 Epsilon-insensitive loss

The *epsilon-insensitive* loss is defined as:

$$\mathcal{L}_\epsilon^1 = \begin{cases} 0 & \text{if } |y - (w^T x + b)| \leq \epsilon \\ |y - (w^T x + b)| - \epsilon & \text{otherwise} \end{cases} \tag{54}$$

or, equivalently:

$$\mathcal{L}_\epsilon^1 = \max(0, |y - (w^T x + b)| - \epsilon) \tag{55}$$

As the *hinge* loss, also the *epsilon-insensitive* loss is a nondifferentiable convex function due to its nonsmoothness in $\pm\epsilon$, but has a subgradient wrt w that is given by:

$$\frac{\partial \mathcal{L}_\epsilon^1}{\partial w} = \begin{cases} (y - (w^T x + b))x & \text{if } |y - (w^T x + b)| > \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{56}$$

4.1.1 Primal formulation

The general primal unconstrained formulation takes the same form of (9).

The quadratic optimization problem (53) can be equivalently formulated as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^n \max(0, |y_i - (w^T x_i + b)| - \epsilon) \tag{57}$$

where we make use of the *epsilon-insensitive* loss (54) or (55).

The above formulation penalizes slacks ξ linearly and is called $\mathcal{L}_1$-SVR.

The $\mathcal{L}_1$-SVR objective (57) can be rewritten in form (11) or (12) as:

$$\min_{w,b} \frac{1}{2}(\|w\|^2 + b^2) + C \sum_{i=1}^n \max(0, |y_i - (w^T x_i + b)| - \epsilon) \tag{58}$$

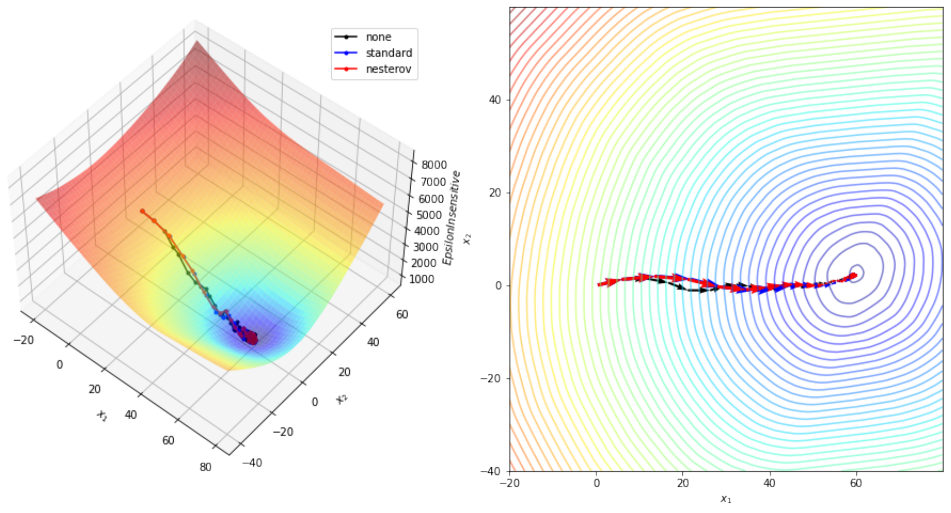


Figure 5: SVR Epsilon-insensitive loss with different optimization steps

4.1.2 Wolfe Dual formulation

To reformulate the (53) as a *Wolfe dual*, we introduce the Lagrange multipliers $\alpha_i^+ \geq 0, \alpha_i^- \geq 0, \mu_i^+ \geq 0, \mu_i^- \geq 0 \ \forall_i$:

$$\begin{aligned} \max_{\alpha^+,\alpha^-,\mu^+,\mu^-} \min_{w,b,\xi^+,\xi^-} \mathcal{W}(w,b,\xi^+,\xi^-,\alpha^+,\alpha^-,\mu^+,\mu^-) &= \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i^+ + \xi_i^-) - \sum_{i=1}^{n}(\mu_i^+\xi_i^+ + \mu_i^-\xi_i^-) \\ &- \sum_{i=1}^{n}\alpha_i^+(\epsilon + \xi_i^+ + y_i' - y_i) - \sum_{i=1}^{n}\alpha_i^-(\epsilon + \xi_i^- - y_i' + y_i) \end{aligned} \tag{59}$$

Substituting for y_i, differentiating wrt w, b, ξ^+, ξ^- and setting the derivatives to 0 gives:

$$\frac{\partial \mathcal{W}}{\partial w} = w - \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-)x_i \Rightarrow w = \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-)x_i \tag{60}$$

$$\frac{\partial \mathcal{W}}{\partial b} = -\sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-) \Rightarrow \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-) = 0 \tag{61}$$

$$\frac{\partial \mathcal{W}}{\partial \xi_i^+} = 0 \Rightarrow C = \alpha_i^+ + \mu_i^+ \tag{62}$$

$$\frac{\partial \mathcal{W}}{\partial \xi_i^-} = 0 \Rightarrow C = \alpha_i^- + \mu_i^- \tag{63}$$

Substituting (60) and (61) in, we now need to maximize $\mathcal{W}$ wrt α_i^+ and α_i^-, where $\alpha_i^+ \geq 0, \ \alpha_i^- \geq 0 \ \forall_i$:

$$\max_{\alpha^+,\alpha^-} \mathcal{W}(\alpha^+,\alpha^-) = \sum_{i=1}^{n} y_i(\alpha_i^+ - \alpha_i^-) - \epsilon\sum_{i=1}^{n}(\alpha_i^+ + \alpha_i^-) - \frac{1}{2}\sum_{i,j}(\alpha_i^+ - \alpha_i^-)\langle x_i, x_j\rangle(\alpha_j^+ - \alpha_j^-) \tag{64}$$

Using $\mu_i^+ \geq 0$ and $\mu_i^- \geq 0$ together with (60) and (61) means that $\alpha_i^+ \leq C$ and $\alpha_i^- \leq C$. We therefore need to find:

$$\begin{aligned} \min_{\alpha^+,\alpha^-} \quad & \frac{1}{2}(\alpha^+ - \alpha^-)^T K(\alpha^+ - \alpha^-) + \epsilon q^T(\alpha^+ + \alpha^-) - y^T(\alpha^+ - \alpha^-) \\ \text{subject to} \quad & 0 \leq \alpha_i^+, \alpha_i^- \leq C \ \forall_i \\ & q^T(\alpha^+ - \alpha^-) = 0 \end{aligned} \tag{65}$$

where $q^T = [1, \dots, 1]$.
We can write the (65) in a standard quadratic form as:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T Q\alpha - q^T\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \\ & e^T\alpha = 0 \end{aligned} \tag{66}$$

where the Hessian matrix Q is $\begin{bmatrix} K & -K \\ -K & K \end{bmatrix}$, q is $\begin{bmatrix} -y \\ y \end{bmatrix} + \epsilon$, and e is $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$.
Each new predictions y' can be found using:

$$y' = \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-)\langle x_i, x'\rangle + b \tag{67}$$

A set S of support vectors x_s can be created by finding the indices i where $0 \leq \alpha \leq C$ and $\xi_i^+ = 0$ or $\xi_i^- = 0$.

This gives us:

$$b = y_s - \epsilon - \sum_{m \in S} (\alpha_m^+ - \alpha_m^-)\langle x_m, x_s \rangle \tag{68}$$

As before it is better to average over all the indices i in S:

$$b = \frac{1}{N_s} \sum_{s \in S} y_s - \epsilon - \sum_{m \in S} (\alpha_m^+ - \alpha_m^-)\langle x_m, x_s \rangle \tag{69}$$

From (66) we can notice that the equality constraint $e^T\alpha = 0$ arises form the stationarity condition $\partial_b \mathcal{W} = 0$. So, again, for simplicity, we can again consider the bias term b embedded into the weight vector. We report below the box-constrained dual formulation [14] that arises from the primal (11) or (12) where the bias term b is embedded into the weight vector w:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T(Q + ee^T)\alpha + q^T\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \end{aligned} \tag{70}$$

4.1.3 Lagrangian Dual formulation

In order to relax the constraints in the *Wolfe dual* formulation (65) we define the problem as a *Lagrangian dual* relaxation by embedding them into objective function, so we need to allocate the Lagrangian multipliers $\mu \geq 0, \lambda_+ \geq 0, \lambda_- \geq 0$:

$$\begin{aligned} \max_{\mu,\lambda_+,\lambda_-} \min_{\alpha} \mathcal{L}(\alpha, \mu, \lambda_+, \lambda_-) &= \frac{1}{2}\alpha^T Q\alpha + q^T\alpha - \mu^T(e^T\alpha) - \lambda_+^T(u - \alpha) - \lambda_-^T\alpha \\ &= \frac{1}{2}\alpha^T Q\alpha + (q - \mu e + \lambda_+ - \lambda_-)^T\alpha - \lambda_+^T u \end{aligned} \tag{71}$$

where the upper bound $u^T = [C, \dots, C]$.

Taking the derivative of the Lagrangian $\mathcal{L}$ wrt α and settings it to 0 gives:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow Q\alpha + (q - \mu e + \lambda_+ - \lambda_-) = 0 \tag{72}$$

With α optimal solution of the linear system:

$$Q\alpha = -(q - \mu e + \lambda_+ - \lambda_-) \tag{73}$$

the gradient wrt μ, λ_+ and λ_- are:

$$\frac{\partial \mathcal{L}}{\partial \mu} = -e\alpha \tag{74}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_+} = \alpha - u \tag{75}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_-} = -\alpha \tag{76}$$

If the Hessian matrix Q is not positive definite, i.e., the Lagrangian function is not strictly convex since it will be linear along the eigenvectors correspondent to the null eigenvalues and so it will be unbounded below, the Lagrangian dual relaxation will be nondifferentiable, so it will have infinite solutions and for each of them it will have a different subgradient. In order to compute an approximation of the gradient, we will choose α in such a way as the one that minimizes the norm of the residual:

$$\begin{aligned} \min_{\alpha_n \in K_n(Q,b)} \quad & \|Q\alpha_n - b\| \\ \text{where} \quad & b = -(q - \mu e + \lambda_+ - \lambda_-) \end{aligned} \tag{77}$$

Since we are dealing with a symmetric but indefinite linear system we will choose a well-known Krylov method that performs the Lanczos iterate, i.e., symmetric Arnoldi iterate, called *minres*, i.e., symmetric *gmres*, to compute the vector α_n that minimizes the norm of the residual $r_n = Q\alpha_n - b$ among all vectors in $K_n(Q, b) = span(b, Qb, Q^2b, \dots, Q^{n-1}b)$.

From (66) we can notice that the equality constraint $e^T\alpha = 0$ arises form the stationarity condition $\partial_b \mathcal{W} = 0$. So, again, for simplicity, we can again consider the bias term b embedded into the weight vector. In this way the dimensionality of (71) is reduced of 1/3 by removing the multipliers μ which was allocated to control the equality constraint $e^T\alpha = 0$, so we will end up solving exactly the problem (70).

$$\begin{aligned}\max_{\lambda_+,\lambda_-} \min_{\alpha} \mathcal{L}(\alpha, \lambda_+, \lambda_-) &= \frac{1}{2}\alpha^T(Q + ee^T)\alpha + q^T\alpha - \lambda_+^T(u - \alpha) - \lambda_-^T\alpha \\ &= \frac{1}{2}\alpha^T(Q + ee^T)\alpha + (q + \lambda_+ - \lambda_-)^T\alpha - \lambda_+^T u\end{aligned} \tag{78}$$

where, again, the upper bound $u^T = [C, \dots, C]$.
Now, taking the derivative of the Lagrangian $\mathcal{L}$ wrt α and settings it to 0 gives:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow (Q + ee^T)\alpha + (q + \lambda_+ - \lambda_-) = 0 \tag{79}$$

With α optimal solution of the linear system:

$$(Q + ee^T)\alpha = -(q + \lambda_+ - \lambda_-) \tag{80}$$

the gradient wrt λ_+ and λ_- are:

$$\frac{\partial \mathcal{L}}{\partial \lambda_+} = \alpha - u \tag{81}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_-} = -\alpha \tag{82}$$

4.2 Squared Epsilon-insensitive loss

The *squared epsilon-insensitive* loss is defined as:

$$\mathcal{L}_\epsilon^2 = \begin{cases} 0 & \text{if } |y - (w^T x + b)| \leq \epsilon \\ (|y - (w^T x + b)| - \epsilon)^2 & \text{otherwise} \end{cases} \tag{83}$$

or, equivalently:

$$\mathcal{L}_\epsilon^2 = \max(0, |y - (w^T x + b)| - \epsilon)^2 \tag{84}$$

As the *squared hinge* loss, also the *squared epsilon-insensitive* loss is a strictly convex function and it has a gradient wrt w that is given by:

$$\frac{\partial \mathcal{L}_\epsilon^2}{\partial w} = \begin{cases} 2((y - (w^T x + b))x) & \text{if } |y - (w^T x + b)| > \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{85}$$

4.2.1 Primal formulation

To provide a continuously differentiable function the optimization problem (57) can be formulated as:

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \max(0, |y_i - (w^T x_i + b)| - \epsilon)^2 \tag{86}$$

where we make use of the *squared epsilon-insensitive* loss that quadratically penalized slacks ξ and is called $\mathcal{L}_2$-SVR.

The $\mathcal{L}_2$-SVR objective (86) can be rewritten in form (11) or (12) as:

$$\min_{w,b} \frac{1}{2} (\|w\|^2 + b^2) + C \sum_{i=1}^n \max(0, |y_i - (w^T x_i + b)| - \epsilon)^2 \tag{87}$$

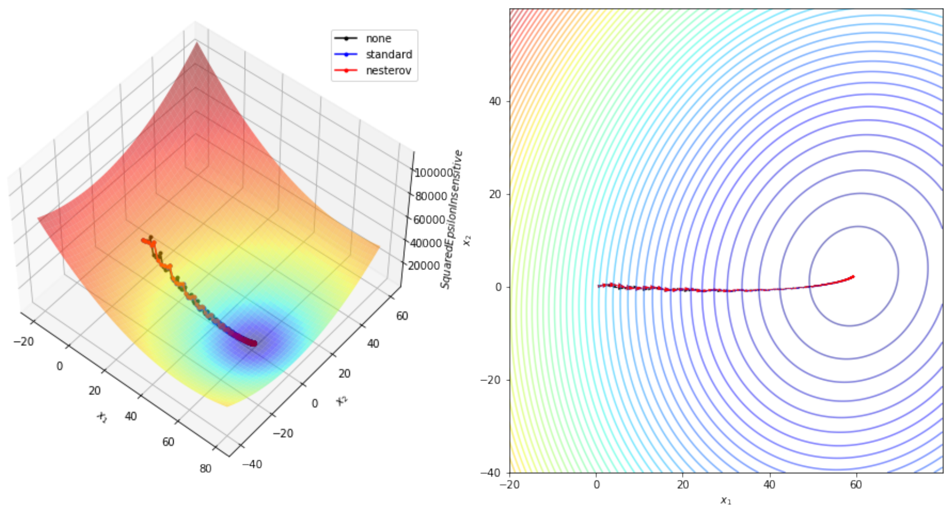


Figure 6: SVC Squared Epsilon-insensitive loss with different optimization steps

4.2.2 Wolfe Dual formulation

In the same way we can derive the dual form of the $\mathcal{L}_2$-SVR by obtaining:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T(Q+D)\alpha + q^T\alpha \\ \text{subject to} \quad & \alpha_i \geq 0 \ \forall_i \\ & e^T\alpha = 0 \end{aligned} \tag{88}$$

or, alternatively, with the regularized bias term by obtaining:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T(Q+ee^T+D)\alpha + q^T\alpha \\ \text{subject to} \quad & \alpha_i \geq 0 \ \forall_i \end{aligned} \tag{89}$$

where the diagonal matrix $D_{ii} = \dfrac{1}{2C} \ \forall_i$.

4.2.3 Lagrangian Dual formulation

In order to relax the constraints in the $\mathcal{L}_2$-SVR *Wolfe dual* formulation (88) we define the problem as a *Lagrangian dual* relaxation by embedding them into objective function, so we need to allocate the Lagrangian multipliers $\mu \geq 0, \lambda \geq 0$:

$$\begin{aligned} \max_{\mu,\lambda} \min_{\alpha} \mathcal{L}(\alpha,\mu,\lambda) &= \frac{1}{2}\alpha^T(Q+D)\alpha + q^T\alpha - \mu^T(e^T\alpha) - \lambda^T\alpha \\ &= \frac{1}{2}\alpha^T(Q+D)\alpha + (q-\mu e-\lambda)^T\alpha \end{aligned} \tag{90}$$

Taking the derivative of the Lagrangian $\mathcal{L}$ wrt α and settings it to 0 gives:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow (Q+D)\alpha + (q-\mu e-\lambda) = 0 \tag{91}$$

With α optimal solution of the linear system:

$$Q\alpha = -(q-\mu e-\lambda) \tag{92}$$

the gradient wrt μ and λ are:

$$\frac{\partial \mathcal{L}}{\partial \mu} = -e\alpha \tag{93}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = -\alpha \tag{94}$$

5 Nonlinear Support Vector Machines

When applying our SVC to *linearly separable* data in (18), we have started by creating a matrix Q from the dot product of our input variables:

$$Q_{ij} = y_i y_j k(x_i, x_j) \tag{95}$$

or, a matrix K from the dot product of our input variables in the SVR case (65):

$$K_{ij} = k(x_i, x_j) \tag{96}$$

where $k(x_i, x_j)$ is an example of a family of functions called *kernel functions* and:

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle = \phi(x_i)^T \phi(x_j) \tag{97}$$

where $\phi(.)$ is the identity function, is known as *linear* kernel.

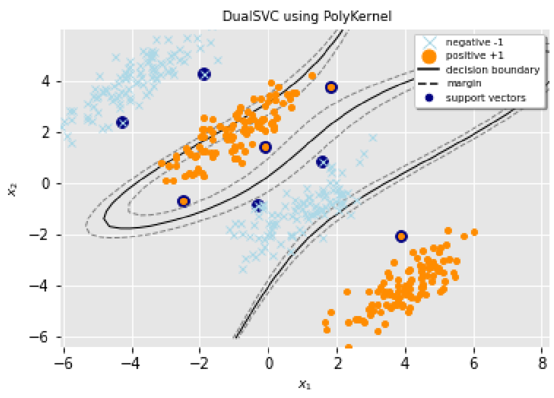
The reason that this *kernel trick* is useful is that there are many classification/regression problems that are nonlinearly separable/regressable in the *input space*, which might be in a higher dimensionality *feature space* given a suitable mapping $x \to \phi(x)$.

5.1 Polynomial kernel

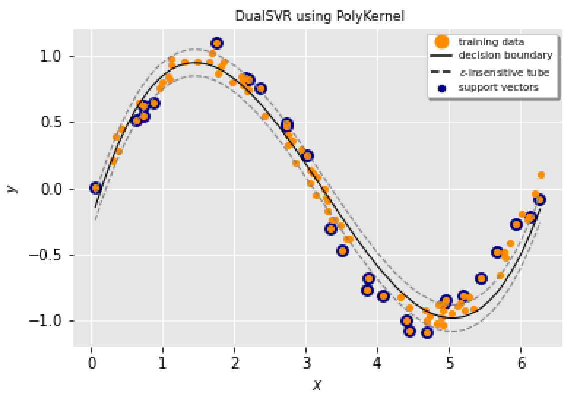
The *polynomial* kernel is defined as:

$$k(x_i, x_j) = (\gamma \langle x_i, x_j \rangle + r)^d \tag{98}$$

where γ define how far the influence of a single training example reaches (low values meaning 'far' and high values meaning 'close').



(a) Polynomial SVC hyperplane



(b) Polynomial SVR hyperplane

Figure 7: Polynomial SVM hyperplanes

5.2 Gaussian RBF kernel

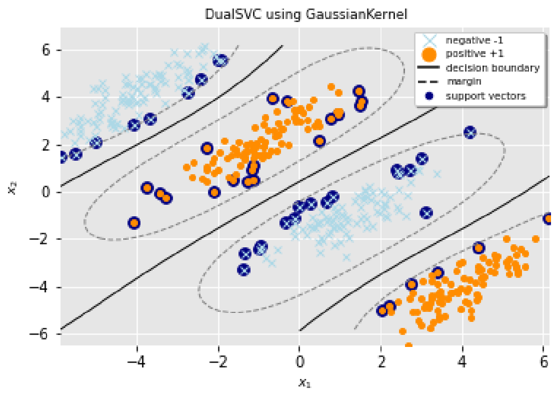
The *gaussian* kernel is defined as:

$$k(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2}) \tag{99}$$

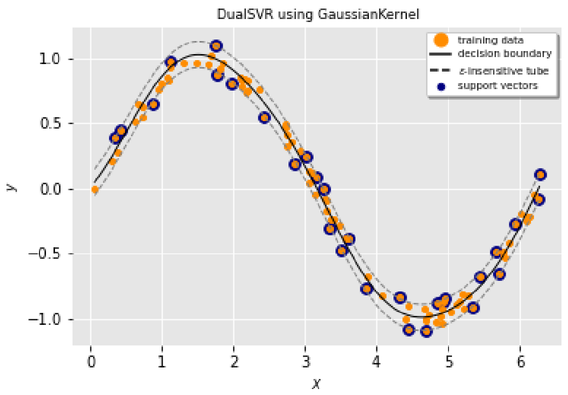
or, equivalently:

$$k(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2) \tag{100}$$

where $\gamma = \dfrac{1}{2\sigma^2}$ define how far the influence of a single training example reaches (low values meaning 'far' and high values meaning 'close').



(a) Gaussian SVC hyperplane



(b) Gaussian SVR hyperplane

Figure 8: Gaussian SVM hyperplanes

6 Optimization Methods

In order to explain the *convergence* and *efficiency* properties of the following optimization methods, we need to introduce some preliminary definitions about *convexity* and the *L-Lipschitz continuity* of a function [15].

Definition 1 (Convexity)**.**

(i) We say that a function $f : \Re^m \to \Re$ is convex if:

$$(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \ \forall \ x, y \in \Re^m, \lambda \in [0, 1]$$

(ii) We say that a differentiable function $f : \Re^m \to \Re$, i.e., $f \in C^1$, is convex if:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle \ \forall \ x, y \in \Re^m$$

(iii) We say that a twice differentiable function $f : \Re^m \to \Re$, i.e., $f \in C^2$ and the Hessian matrix is symmetric, is convex iff:

$$\nabla^2 f(x) \succeq 0 \ \forall \ x \in \Re^m$$

i.e., the Hessian matix is *positive semidefinite.*

Definition 2 (Strict Convexity)**.**

(i) We say that a function $f : \Re^m \to \Re$ is strictly convex if:

$$(\lambda x + (1-\lambda)y) < \lambda f(x) + (1-\lambda)f(y) \ \forall \ x, y \in \Re^m, x \neq y, \lambda \in (0, 1)$$

(ii) We say that a differentiable function $f : \Re^m \to \Re$, i.e., $f \in C^1$, is strictly convex if:

$$f(y) > f(x) + \langle \nabla f(x), y - x \rangle \ \forall \ x, y \in \Re^m, x \neq y$$

(iii) We say that a twice differentiable function $f : \Re^m \to \Re$, i.e., $f \in C^2$ and the Hessian matrix is symmetric, is strictly convex iff:

$$\nabla^2 f(x) \succ 0 \ \forall \ x \in \Re^m$$

i.e., the Hessian matix is *positive definite.*

Definition 3 (Strong Convexity)**.** We say that a function $f : \Re^m \to \Re$ is μ-strongly convex if the function:

$$g(x) = f(x) - \frac{\mu}{2}\|x\|^2$$

is convex for any $\mu > 0$. If f is differentiable, i.e., $f \in C^1$, this is also equivalent to:

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2}\|y - x\|^2 \ \forall \ x, y \in \Re^m$$

and, if f is a twice differentiable function, i.e., $f \in C^2$ and the Hessian matrix is symmetric, then f is μ-strongly convex iff:

$$\nabla^2 g(x) \succ 0 \ \forall \ x \in \Re^m$$

i.e., the Hessian matix is *positive definite*, which is:

$$\nabla^2 f(x) \succeq \mu I \ \forall \ x \in \Re^m$$

Definition 4 (L-Lipschitz continuity)**.** We say that a function $f : \Re^m \to \Re$ is L-smooth, i.e., L-Lipschitz continuous, if:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \ \forall \ x, y \in \Re^m$$

then:

$$f(y) \leq f(x) + \langle \nabla f(x), y - x \rangle + \frac{L}{2}\|y - x\|^2 \ \forall \ x, y \in \Re^m$$

and, if f is a μ-strongly convex function, we give the following Hessian bounds:

$$0 \prec \mu I \preceq \nabla^2 f(x) \preceq LI \ \forall \ x \in \Re^m$$

We say that a function $f : \Re^m \to \Re$ is locally L-smooth, i.e., locally L-Lipschitz continuous, if for every x in $\Re^m$ there exists a neighborhood U of x such that f restricted to U is L-Lipschitz continuous. Every convex function is locally L-Lipschitz continuous.

Definition 5 (Subgradient)**.** Given a function $f : \Re^m \to \Re$ and $x \in \Re^m$, we define a subgradient $g \in \Re^m$ at x to be any point satisfying:

$$f(y) \geq f(x) + \langle g, y - x \rangle \ \forall \ y \in \Re^m$$

Subgradients always exist for convex function.

6.1 Gradient Descent for Primal formulations

The Gradient Descent algorithm is the simplest *first-order optimization* method that exploits the orthogonality of the gradient wrt the level sets to take a descent direction. In particular, it performs the following iterations:

Algorithm 1 Gradient Descent

```
Require: Function $f$ to minimize
Require: Learning rate or step size $\alpha > 0$
  function GRADIENTDESCENT($f, \alpha$)
      Initialize weight vector $x_0$
      $t = 0$
      while not_convergence do
          $x_{t+1} = x_t - \alpha \partial f(x_t)$          ▷ if $f$ is differentiable then $\partial f(x_t) = \nabla f(x_t)$
          $t = t + 1$
      end while
      return $x_t$
  end function
```

Gradient Descent is based on full gradients, since at each iteration we compute the average gradient on the whole dataset:

$$\partial f(x) = \frac{1}{n} \sum_{i=1}^{n} \partial f_i(x)$$

The downside is that every step is very computationally expensive, $\mathcal{O}(nm)$ per iteration, where n is the number of samples in our dataset and m is the number of dimensions.

Since *Gradient Descent* becomes impractical when dealing with large datasets we introduce a stochastic version, called *Stochastic Gradient Descent*, which does not use the whole set of examples to compute the gradient at every step. By doing so, we can reduce computation all the way down to $\mathcal{O}(m)$ per iteration.

Algorithm 2 Stochastic Gradient Descent

```
Require: Function $f$ to minimize
Require: Learning rate or step size $\alpha > 0$
Require: Batch size $k$
  function STOCHASTICGRADIENTDESCENT($f, \alpha, k$)
      Initialize weight vector $x_0$
      $t \leftarrow 0$
      while not_convergence do
          Sample $(i_1, \dots, i_k) \sim \mathcal{U}^k(1, \dots, n)$
          $x_{t+1} \leftarrow x_t - \alpha \frac{1}{k} \sum_{j=1}^{k} \partial f_{i_j}(x_t)$          ▷ if $f$ is differentiable then $\partial f_{i_j}(x_t) = \nabla f_{i_j}(x_t)$
          $t \leftarrow t + 1$
      end while
      return $x_t$
  end function
```

Note that in expectation, we converge like GD, since $\mathbb{E}_{i \sim \mathcal{U}(1,\dots,n)}[\partial f_i(x_t)] = \partial f(x_t)$, therefore, the expected iterate of SGD converges to the optimum.

Now, consider the SGD algorithm introduced previously but where each iteration is projected into the ball $\mathcal{B}(0, R)$ with radius $R > 0$ fixed. So, the following lower bounds on convergence rates are given.

Theorem 6 (Stochastic Gradient Descent convergence for convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous convex function and assume that exists $b > 0$ satisfying:*

$$\|f_i(x)\| \leq b \ \forall \ x \in \mathcal{B}(0, R)$$

Besides, assume that all minima of f belong to $\mathcal{B}(0, R)$. Then the Stochastic Gradient Descent with step size $\alpha = \dfrac{2R}{b\sqrt{k}}$ satisfies:

$$\mathbb{E}\left[f\left(\frac{1}{k}\sum_{t=1}^{k} x_t\right)\right] - f(x^*) \leq \frac{3Rb}{\sqrt{k}}$$

Theorem 7 (Stochastic Gradient Descent convergence for strongly convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous, μ-strongly convex function and assume that exists $b > 0$ satisfying:*

$$\|f_i(x)\| \leq b \ \forall \ x \in \mathcal{B}(0, R)$$

Besides, assume that all minima of f belong to $\mathcal{B}(0, R)$. Then the Stochastic Gradient Descent with step size $\alpha = \dfrac{2}{\mu(k+1)}$ satisfies:

$$\mathbb{E}\left[f\left(\frac{2}{k(k+1)}\sum_{t=1}^{k} t x_{t-1}\right)\right] - f(x^*) \leq \frac{2b^2}{\mu(k+1)}$$

SGD's *convergence rate* for L-Lipschitz continuous convex functions is $\mathcal{O}\left(\dfrac{1}{\sqrt{t}}\right)$ and $\mathcal{O}\left(\dfrac{1}{t}\right)$ for L-Lipschitz continuous and strongly convex functions. More iterations are needed to reach the same accuracy as GD, but the iterations are far cheaper.

6.1.1 Nonsmooth

First, consider a nonsmooth, i.e., nondifferentiable, convex function. So, the following lower bounds on convergence rates are given.

Theorem 8 (Subgradient Descent convergence for convex functions with Polyak's stepsize)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous convex function. Then the Subgradient Descent with Polyak's step size $\alpha_t = \dfrac{f(x_t) - f(x^*)}{\|g_t\|^2}$ satisfies:*

$$f(x_t) - f(x^*) \leq \frac{L\|x_0 - x^*\|^2}{\sqrt{t+1}}$$

Unfortunately, Polyak's stepsize rule requires knowledge of $f(x^*)$, which is often unknown a priori, so we might often need simpler rule for setting stepsizes.

Theorem 9 (Subgradient Descent convergence for convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous convex function. Then the Subgradient Descent with step size $\alpha_t = \dfrac{1}{\sqrt{t}}$ satisfies:*

$$f(x_t) - f(x^*) \leq \frac{\|x_0 - x^*\|^2 + L^2 \log t}{\sqrt{t}}$$

Theorem 10 (Subgradient Descent convergence for strongly convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous and μ-strongly convex function. Then the Subgradient Descent with step size $\alpha_t = \dfrac{2}{\mu(t+1)}$ satisfies:*

$$f(x_t) - f(x^*) \leq \frac{2L^2}{\mu}\frac{1}{t+1}$$

The Subgradient Descent *convergence rate* for L-Lipschitz continuous convex functions is $\mathcal{O}\left(\frac{1}{\sqrt{t}}\right)$ and $\mathcal{O}\left(\frac{1}{t}\right)$ for L-Lipschitz continuous and strongly convex functions. We can also write the *iteration complexity*, i.e., the smallest t such that we're within ϵ-close to global optimum, as $\mathcal{O}\left(\frac{1}{\epsilon^2}\right)$ for L-Lipschitz continuous convex functions and as $\mathcal{O}\left(\frac{1}{\epsilon}\right)$ for L-Lipschitz continuous and strongly convex functions.

Among algorithms that only use subgradient, these *convergence rates* cannot be futher improved.

6.1.2 Smooth

Now, consider a smooth, i.e., differentiable, convex function. So, the following lower bounds on convergence rates are given.

Theorem 11 (Gradient Descent convergence for convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous convex function. Then the Gradient Descent with step size $\alpha \leq 1/L$ satisfies:*

$$f(x_t) - f(x^*) \leq \frac{\|x_0 - x^*\|^2}{2\alpha t}$$

In particular, for $\alpha = 1/L$:

$$f(x_t) - f(x^*) \leq \frac{L\|x_0 - x^*\|^2}{2t}$$

Theorem 12 (Gradient Descent convergence for strongly convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous and μ-strongly convex function. Then the Gradient Descent with step size $\alpha \leq 1/L$ satisfies:*

$$f(x_t) - f(x^*) \leq (1 - \alpha\mu)^t \|x_0 - x^*\|^2$$

In particular, for $\alpha = 1/L$:

$$\begin{aligned} f(x_t) - f(x^*) \leq& \left(1 - \frac{\mu}{L}\right)^t \|x_0 - x^*\|^2 \\ =& \left(1 - \frac{1}{\kappa}\right)^t \|x_0 - x^*\|^2 \end{aligned}$$

where $\kappa = L/\mu$.

Theorem 13 (Gradient Descent convergence for convex quadratic functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous and μ-strongly convex quadratic function. Then the Gradient Descent with step size $\alpha = \frac{2}{L + \mu}$ and momentum $\beta = \frac{\kappa - 1}{\kappa + 1} = 1 - \frac{2}{\kappa + 1}$ satisfies:*

$$\|x_t - x^*\| = \left(\frac{\kappa - 1}{\kappa + 1}\right)^t \|x_0 - x^*\|$$

where $\kappa = L/\mu$.

The Gradient Descent *convergence rate* for L-Lipschitz continuous convex functions is $\mathcal{O}\left(\frac{1}{t}\right)$ and $\mathcal{O}\left(\exp\left(-\frac{t}{\kappa}\right)\right)$ for L-Lipschitz continuous and strongly convex functions. We can also write the *iteration complexity*, i.e., the smallest t such that we're within ϵ-close to global optimum, as $\mathcal{O}\left(\frac{1}{\epsilon}\right)$ for L-Lipschitz continuous convex functions and as $\mathcal{O}\left(\kappa \log \frac{1}{\epsilon}\right)$ for L-Lipschitz continuous and strongly convex functions.

6.1.3 Momentum

To mitigate the pathological zig-zagging by speeding up the *convergence rate* of the SGD method, we introduce two accelerated methods [1] and [2, 3] that exploits information from the history, i.e., past iterates, to add some inertia, i.e., the momentum, to yield smoother trajectory.

In the Polyak's method [1] the velocity vector v_t is calculated by applying the β momentum to the previous v_{t-1} displacement, and subtracting the gradient step to x_t.

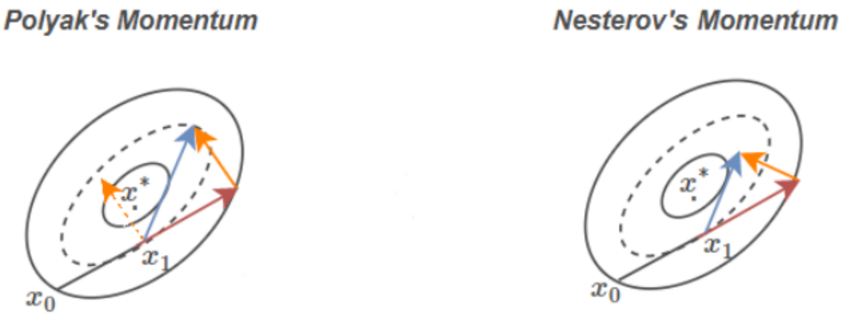


Figure 9: Polyak's and Nesterov's Momentum

Algorithm 3 Polyak's Accelerated Gradient Descent or Polyak Heavy-Ball method

```
Require: Function $f$ to minimize
Require: Learning rate or step size $\alpha > 0$
Require: Momentum $\beta \in [0, 1)$
  function PolyakAcceleratedGradientDescent($f, \alpha, \beta$)
    Initialize weight vector $x_1 \leftarrow x_0$ and velocity vector $v_0 \leftarrow 0$
    $t \leftarrow 1$
    while $not\_convergence$ do
      $v_t = \beta v_{t-1} + \alpha \nabla f(x_t)$
      $x_{t+1} = x_t - v_t$
      $t \leftarrow t + 1$
    end while
    return $x_t$
  end function
```

Theorem 14 (Polyak's Accelerated Gradient Descent convergence for convex quadratic functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous and μ-strongly convex quadratic function. Then the Polyak's Accelerated Gradient Descent with step size $\alpha = \frac{4}{(\sqrt{L}+\sqrt{\mu})^2}$ and momentum $\beta = \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} = 1 - \frac{2}{\sqrt{\kappa}+1}$ satisfies:*

$$\|x_t - x^*\| = \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^t \|x_0 - x^*\|$$

where $\kappa = L/\mu$.

Leveraging the idea of momentum introduced by Polyak, Nesterov introduced a slightly altered update rule that has been shown to converge not only for quadratic functions, but for general convex functions. In the Nesterov's method [2], instead, the velocity vector v_t is calculated by applying the β momentum to the previous v_{t-1} displacement, and subtracting the gradient step to $x_t + \beta v_{t-1}$, which is the point where the momentum term leads from x_t.

Algorithm 4 Nesterov's Accelerated Gradient Descent or Nesterov Heavy-Ball method

```
Require: Function $f$ to minimize
Require: Learning rate $\alpha > 0$
Require: Momentum $\beta \in [0, 1)$
  function NesterovAcceleratedGradientDescent($f, \alpha, \beta$)
      Initialize weight vector $x_1 \leftarrow x_0$ and velocity vector $v_0 \leftarrow 0$
      $t \leftarrow 1$
      while $not\_convergence$ do
          $\hat{x}_t \leftarrow x_t + \beta v_{t-1}$
          $v_t \leftarrow \beta v_{t-1} + \alpha \nabla f(\hat{x}_t)$
          $x_{t+1} \leftarrow x_t - v_t$
          $t \leftarrow t + 1$
      end while
      return $x_t$
  end function
```

Comparing the algorithm 3 with the algorithm 4, we can see that Polyak's method evaluates the gradient before adding momentum, whereas Nesterov's algorithm evaluates it after applying momentum, which intuitively brings us closer to the minimum x^*, as showb in figure 9.

Theorem 15 (Nesterov's Accelerated Gradient Descent convergence for convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous convex function. Then the Nesterov's Accelerated Gradient Descent with step size $\alpha \leq 1/L$ and momentum $\beta_{t+1} = t/(t+3)$ satisfies:*

$$f(x_t) - f(x^*) \leq \frac{2\|x_0 - x^*\|^2}{\alpha(t+1)^2}$$

In particular, for $\alpha = 1/L$:

$$f(x_t) - f(x^*) \leq \frac{2L\|x_0 - x^*\|^2}{(t+1)^2}$$

Theorem 16 (Nesterov's Accelerated Gradient Descent convergence for strongly convex functions)**.** *Let $f : \Re^m \to \Re$ be a L-Lipschitz continuous and μ-strongly convex function. Then the Nesterov's Accelerated Gradient Descent with step size $\alpha \leq 1/L$ and momentum $\beta_t = \dfrac{1 - \sqrt{\mu/L}}{1 + \sqrt{\mu/L}} = \dfrac{1 - 1/\sqrt{\kappa}}{1 + 1/\sqrt{\kappa}}$ satisfies:*

$$\begin{aligned} f(x_t) - f(x^*) \leq & \frac{\|x_0 - x^*\|^2}{\alpha} \left(1 - \sqrt{\frac{\mu}{L}}\right)^t \\ = & \frac{\|x_0 - x^*\|^2}{\alpha} \left(1 - \frac{1}{\sqrt{\kappa}}\right)^t \end{aligned}$$

In particular, for $\alpha = 1/L$:

$$\begin{aligned} f(x_t) - f(x^*) \leq & L\|x_0 - x^*\|^2 \left(1 - \sqrt{\frac{\mu}{L}}\right)^t \\ = & L\|x_0 - x^*\|^2 \left(1 - \frac{1}{\sqrt{\kappa}}\right)^t \end{aligned}$$

where $\kappa = L/\mu$.

Nesterov's momentum brings the *convergence rate* from $\mathcal{O}\left(\frac{1}{t}\right)$ to $\mathcal{O}\left(\frac{1}{t^2}\right)$ and in the case of strongly convex functions gives the acceleration that we had with Polyak's momentum for quadratic functions, i.e., $\mathcal{O}\left(\exp\left(-\frac{t}{\sqrt{\kappa}}\right)\right)$. This is great because we get the guarantee for a more general class of functions but this rate of convergence cannot be further improved only using first-order information. We can also write the *iteration complexity*, i.e., the smallest t such that we're within ϵ-close to global optimum, for a L-Lipschitz continuous and μ-strongly convex function as $\mathcal{O}\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$ for the accelerated methods where κ, i.e., the *conditioning number*, is defined as $\kappa = L/\mu$ and where L and μ are also equal to the largest λ_{max} and the smallest λ_{min} eigenvalues respectively. Finally, NAG's *iteration complexity* for L-Lipschitz continuous convex functions is $\mathcal{O}\left(\frac{1}{\sqrt{\epsilon}}\right)$.

6.2 Sequential Minimal Optimization for Wolfe Dual formulations

The *Sequential Minimal Optimization (SMO)* [4] method is the most popular approach for solving the SVM QP problem without any extra Q matrix storage required by common QP methods. The advantage of SMO lies in the fact that it performs a series of two-point optimizations since we deal with just one equality constraint, so the Lagrange multipliers can be solved analitically.

6.2.1 Classification

At each iteration, SMO chooses two α_i to jointly optimize, let α_1 and α_2, finds the optimal values for these multipliers and update the SVM to reflect these new values. In order to solve for two Lagrange multipliers, SMO first computes the constraints over these and then solves for the constrained minimum. Since there are only two multipliers, the box-constraints cause the Lagrange multipliers to lie within a box, while the linear equality constraint causes the Lagrange multipliers to lie on a diagonal line inside the box. So, the constrained minimum must lie there as shown in 10.

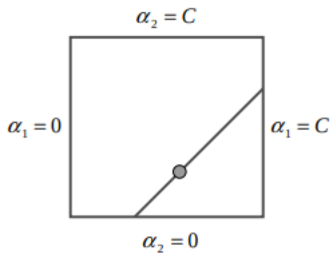


Figure 10: SMO for two Lagrange multipliers

In case of classification the ends of the diagonal line segment, i.e., the lower and upper bounds, can be espressed as follow if the target $y_1 \neq y_2$:

$$\begin{aligned} L &= max(0, \alpha_2 - \alpha_1) \\ H &= min(C, C + \alpha_2 - \alpha_1) \end{aligned} \tag{101}$$

or, alternatively, if the target $y_1 = y_2$:

$$\begin{aligned} L &= max(0, \alpha_2 + \alpha_1 - C) \\ H &= min(C, \alpha_2 + \alpha_1) \end{aligned} \tag{102}$$

The second derivative of the objective quadratic function along the diagonl line can be expressed as:

$$\eta = K(x_1, x_1) + K(x_2, x_2) - 2K(x_1, x_2) \tag{103}$$

that will be grather than zero if the kernel matrix will be positive definite, so there will be a minimum along the linear equality constraints that will be:

$$\alpha_2^{new} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta} \tag{104}$$

where $E_i = y_i - y'_i$ is the error on the i-th training example and y'_i is the output of the SVC for the same.

Then, the box-constrained minimum is found by clipping the unconstrained minimum to the ends of the line segment:

$$\alpha_2^{new,clipped} = \begin{cases} H & \text{if } \alpha_2^{new} \geq H \\ \alpha_2^{new} & \text{if } L < \alpha_2^{new} < H \\ L & \text{if } \alpha_2^{new} \leq L \end{cases} \tag{105}$$

Finally, the value of α_1 is computed from the new clipped α_2 as:

$$\alpha_1^{new} = \alpha_1 + s(\alpha_2 - \alpha_2^{new,clipped}) \tag{106}$$

where $s = y_1 y_2$.

Since the *Karush-Kuhn-Tucker* conditions are necessary and sufficient conditions for optimality of a positive definite QP problem and the KKT conditions for the classification problem (19) are:

$$\begin{aligned} \alpha_i = 0 &\Leftrightarrow y_i y_i' \geq 1 \\ 0 < \alpha_i < C &\Leftrightarrow y_i y_i' = 1 \\ \alpha_i = C &\Leftrightarrow y_i y_i' \leq 1 \end{aligned} \tag{107}$$

the steps described above will be iterate as long as there will be an example that violates them.

After optimizing α_1 and α_2, we select the threshold b such that the KKT conditions are satisfied for x_1 and x_2. If, after optimization, α_1 is not at the bounds, i.e., $0 < \alpha_1 < C$, then the following threshold b_{up} is valid, since it forces the SVC to output y_1 when the input is x_1:

$$b_{up} = E_1 + y_1(\alpha_1^{new} - \alpha_1)K(x_1, x_1) + y_2(\alpha_2^{new,clipped} - \alpha_2)K(x_1, x_2) + b \tag{108}$$

similarly, the following threshold b_{low} is valid if $0 < \alpha_2 < C$:

$$b_{low} = E_2 + y_1(\alpha_1^{new} - \alpha_1)K(x_1, x_2) + y_2(\alpha_2^{new,clipped} - \alpha_2)K(x_2, x_2) + b \tag{109}$$

If, after optimization, both $0 < \alpha_1 < C$ and $0 < \alpha_2 < C$ then both these thresholds are valid, and they will be equal; else, if both α_1 and α_2 are at the bounds, i.e., $\alpha_1 = 0$ or $\alpha_1 = C$ and $\alpha_2 = 0$ or $\alpha_2 = C$, then all the thresholds between b_{up} and b_{low} satisfy the KKT conditions, so we choose the threshold to be halfway in between b_{up} and b_{low}. This gives the complete equation for b:

$$b = \begin{cases} b_{up} & \text{if } 0 < \alpha_1 < C \\ b_{low} & \text{if } 0 < \alpha_2 < C \\ \dfrac{b_{up} + b_{low}}{2} & \text{otherwise} \end{cases} \tag{110}$$

Algorithm 5 Sequential Minimal Optimization for Classification

```
Require: Training examples matrix $X \in \Re^{n \times m}$
Require: Training target vector $y \in \pm 1^n$
Require: Kernel matrix $K \in \Re^{n \times n}$
Require: Regularization parameter $C > 0$
Require: Tolerance value $tol$ for stopping criterion
  function SMOClassifier($X, y, K, C, tol$)
    Initialize the Lagrange multipliers vector $\alpha \in \Re^n, \alpha \leftarrow 0$
    Initialize the empty set $I0 \leftarrow \{i : 0 < \alpha_i < C\}$
    Initialize the set $I1 \leftarrow \{i : y_i = +1, \alpha_i = 0\}$ to contain all the indices of the training examples of class $+1$
    Initialize the empty set $I2 \leftarrow \{i : y_i = -1, \alpha_i = C\}$
    Initialize the empty set $I3 \leftarrow \{i : y_i = +1, \alpha_i = C\}$
    Initialize the set $I4 \leftarrow \{i : y_i = -1, \alpha_i = 0\}$ to contain all the indices of the training examples of class $-1$
    Initialize $b_{up} \leftarrow -1$
    Initialize $b_{low} \leftarrow +1$
    Initialize the error cache vector $errors \in \Re^n, errors \leftarrow 0$
    while $num\_changed > 0$ or $examine\_all = True$ do
      $num\_changed \leftarrow 0$
      $examine\_all \leftarrow True$
      if $examine\_all = True$ then
        for $i \leftarrow 0$ to $n$ do                    ▷ loop over all training examples
          $num\_changed \leftarrow num\_changed +$ ExamineExample($i$)
        end for
      else
        for $i$ in $I0$ do        ▷ loop over examples where $\alpha_i$ are not already at their bounds
          $num\_changed \leftarrow num\_changed +$ ExamineExample($i$)
          if $b_{up} > b_{low} - 2tol$ then              ▷ check if optimality on $I0$ is attained
            $num\_changed \leftarrow 0$
            break
          end if
        end for
      end if
      if $examine\_all = True$ then
        $examine\_all \leftarrow False$
      else if $num\_changed = 0$ then
        $examine\_all \leftarrow True$
      end if
    end while
    Compute $b$ by (110)
    return $\alpha, b$
  end function
```

```
Require: $i2$-th Lagrange multiplier
  function ExamineExample($i2$)
      if $i2$ in $I0$ then
          $E_2 \leftarrow errors_{i2}$
      else
          Compute $E_2$
          $errors_{i2} \leftarrow E_2$
          Update $(b_{low}, i_{low})$ or $(b_{up}, i_{up})$ using $(E_2, i2)$
      end if
      if optimality is attained using current $b_{low}$ and $b_{up}$ then
          return 0
      else
          Find an index $i1$ to do joint optimization with $i2$
          if TakeStep($i1, i2$) = True then
              return 1
          else
              return 0
          end if
      end if
  end function
```

```
Require: $i1$-th Lagrange multiplier
Require: $i2$-th Lagrange multiplier
  function TakeStep($i1, i2$)
      if $i1 = i2$ then
          return False
      end if
      Compute $L$ and $H$ using (101) or (102)
      if $L = H$ then
          return False
      end if
      Compute $\eta$ by (103)        ▷ we assume that $\eta > 0$, i.e., the kernel matrix $K$ is positive definite
      if $\eta < 0$ then
          Choose $\alpha_2^{new,clipped}$ between $L$ and $H$ according to the largest value of the objective function at these
  points
      else
          Compute $\alpha_2^{new}$ by (104)
          Compute $\alpha_2^{new,clipped}$ by (105)
      end if
      if changes in $\alpha_2^{new,clipped}$ are larger than some eps then
          Compute $\alpha_1^{new}$ by (106)
          Update $\alpha_2^{new,clipped}$ and $\alpha_1^{new}$
          for $i$ in $I0$ do
              Update $errors_i$ using new Lagrange multipliers
          end for
          Update $\alpha$ using new Lagrange multipliers
          Update $I0, I1, I2, I3$ and $I4$
          Update $errors_{i1}$ and $errors_{i2}$
          for $i$ in $I0 \cup \{i1, i2\}$ do
              Compute $(i_{low}, b_{low})$ by $b_{low} = \max\{errors_i : i \in I0 \cup I3 \cup I4\}$
              Compute $(i_{up}, b_{up})$ by $b_{up} = \min\{errors_i : i \in I0 \cup I1 \cup I2\}$
          end for
          return True
      else
          return False
      end if
  end function
```

6.2.2 Regression

In case of regression the bounds and the new multipliers $\alpha_1^{+,new}$ and $\alpha_2^{+,new}$ can be expressed as follows if $(\alpha_1^+ > 0$ or $(\alpha_1^- = 0$ and $E_1 - E_2 > 0))$ and $(\alpha_2^+ > 0$ or $(\alpha_2^- = 0$ and $E_1 - E_2 < 0))$:

$$\begin{aligned} L &= max(0, \gamma - C) \\ H &= min(C, \gamma) \end{aligned} \tag{111}$$

$$\alpha_2^{+,new} = \alpha_2^+ - \frac{E_1 - E_2}{\eta} \tag{112}$$

$$\alpha_1^{+,new} = \alpha_1^+ - (\alpha_2^{+,new,clipped} - \alpha_2^+) \tag{113}$$

or, if $(\alpha_1^+ > 0$ or $(\alpha_1^- = 0$ and $E_1 - E_2 > 2\epsilon))$ and $(\alpha_2^- > 0$ or $(\alpha_2^+ = 0$ and $E_1 - E_2 > 2\epsilon))$:

$$\begin{aligned} L &= max(0, -\gamma) \\ H &= min(C, -\gamma + C) \end{aligned} \tag{114}$$

$$\alpha_2^{-,new} = \alpha_2^- + \frac{(E_1 - E_2) - 2\epsilon}{\eta} \tag{115}$$

$$\alpha_1^{+,new} = \alpha_1^+ + (\alpha_2^{-,new,clipped} - \alpha_2^-) \tag{116}$$

or, if $(\alpha_1^- > 0$ or $(\alpha_1^+ = 0$ and $E_1 - E_2 < -2\epsilon))$ and $(\alpha_2^+ > 0$ or $(\alpha_2^- = 0$ and $E_1 - E_2 < -2\epsilon))$:

$$\begin{aligned} L &= max(0, \gamma) \\ H &= min(C, C + \gamma) \end{aligned} \tag{117}$$

$$\alpha_2^{+,new} = \alpha_2^+ - \frac{(E_1 - E_2) + 2\epsilon}{\eta} \tag{118}$$

$$\alpha_1^{-,new} = \alpha_1^- + (\alpha_2^{+,new,clipped} - \alpha_2^+) \tag{119}$$

or, finally, if $(\alpha_1^- > 0$ or $(\alpha_1^+ = 0$ and $E_1 - E_2 < 0))$ and $(\alpha_2^- > 0$ or $(\alpha_2^+ = 0$ and $E_1 - E_2 > 0))$:

$$\begin{aligned} L &= max(0, -\gamma - C) \\ H &= min(C, -\gamma) \end{aligned} \tag{120}$$

$$\alpha_2^{-,new} = \alpha_2^- + \frac{E_1 - E_2}{\eta} \tag{121}$$

$$\alpha_1^{-,new} = \alpha_1^- - (\alpha_2^{-,new,clipped} - \alpha_2^-) \tag{122}$$

where $\gamma = \alpha_1^+ - \alpha_1^- + \alpha_2^+ - \alpha_2^-$. Notice that η and $\alpha_2^{+,new,clipped}$ or $\alpha_2^{-,new,clipped}$ are identical to (103) and (105) respectively.

The KKT conditions for the regression problem (65) are:

$$\begin{aligned} \alpha_i^+ - \alpha_i^- = 0 &\Leftrightarrow |y_i - y_i'| < \epsilon \\ -C < \alpha_i^+ - \alpha_i^- < C &\Leftrightarrow |y_i - y_i'| = \epsilon \\ \alpha_i^+ + \alpha_i^- = C &\Leftrightarrow |y_i - y_i'| > \epsilon \end{aligned} \tag{123}$$

so, the steps described above will be iterate as long as there will be an example that violates them.

In case of regression we select the threshold b as follows:

$$b_{up} = E_1 + ((\alpha_1^+ - \alpha_1^-) - (\alpha_1^{+,new} - \alpha_1^{-,new}))K(x_1, x_1) + ((\alpha_2^+ - \alpha_2^-) - (\alpha_2^{+,new,clipped} - \alpha_2^{-,new,clipped}))K(x_1, x_2) + b \tag{124}$$

$$b_{low} = E_2+((\alpha_1^+-\alpha_1^-)-(\alpha_1^{+,new}-\alpha_1^{-,new}))K(x_1,x_2)+((\alpha_2^+-\alpha_2^-)-(\alpha_2^{+,new,clipped}-\alpha_2^{-,new,clipped}))K(x_2,x_2)+b \tag{125}$$

$$b = \begin{cases} b_{up} & \text{if } 0 < \alpha_1^+, \alpha_1^- < C \\ b_{low} & \text{if } 0 < \alpha_2^+, \alpha_2^- < C \\ \dfrac{b_{up}+b_{low}}{2} & \text{otherwise} \end{cases} \tag{126}$$

The improvements described in [5, 8] for classification and regression respectively are about the definition of subsets of multipliers to efficiently update them at each iteration by separating the multipliers at the bounds from those who can be further minimized.

Algorithm 6 Sequential Minimal Optimization for Regression

```
Require: Training examples matrix $X \in \Re^{n \times m}$
Require: Training target vector $y \in \Re^{n}$
Require: Kernel matrix $K \in \Re^{n \times n}$
Require: Regularization parameter $C > 0$
Require: Epsilon-tube value $\epsilon \geq 0$ within which no penalty is associated in the epsilon-insensitive loss function
Require: Tolerance value $tol$ for stopping criterion
  function SMORegression($X, y, K, C, \epsilon, tol$)
      Initialize the Lagrange multipliers vector $\alpha^{+} \in \Re^{n}, \alpha^{+} \leftarrow 0$
      Initialize the Lagrange multipliers vector $\alpha^{-} \in \Re^{n}, \alpha^{-} \leftarrow 0$
      Initialize the empty set $I0 \leftarrow \{i : 0 < \alpha_i^{+}, \alpha_i^{-} < C\}$
      Initialize the set $I1 \leftarrow \{i : \alpha_i^{+} = 0, \alpha_i^{-} = 0\}$ to contain all the indices of the training examples
      Initialize the empty set $I2 \leftarrow \{i : \alpha_i^{+} = 0, \alpha_i^{-} = C\}$
      Initialize the empty set $I3 \leftarrow \{i : \alpha_i^{+} = C, \alpha_i^{-} = 0\}$
      Initialize $i_{up} \leftarrow 0$                ▷ or any other target index $i_{up}$ from the training examples
      Initialize $i_{low} \leftarrow 0$               ▷ or any other target index $i_{low}$ from the training examples
      Initialize $b_{up} \leftarrow y_{i_{up}} + \epsilon$
      Initialize $b_{low} \leftarrow y_{i_{low}} - \epsilon$
      Initialize the error cache vector $errors \in \Re^{n}, errors \leftarrow 0$
      while $num\_changed > 0$ or $examine\_all = True$ do
          $num\_changed \leftarrow 0$
          $examine\_all \leftarrow True$
          if $examine\_all = True$ then
              for $i \leftarrow 0$ to $n$ do                          ▷ loop over all training examples
                  $num\_changed \leftarrow num\_changed +$ ExamineExample($i$)
              end for
          else
              for $i$ in $I0$ do          ▷ loop over examples where $\alpha_i^{+}$ and $\alpha_i^{-}$ are not already at their bounds
                  $num\_changed \leftarrow num\_changed +$ ExamineExample($i$)
                  if $b_{up} > b_{low} - 2tol$ then                 ▷ check if optimality on $I0$ is attained
                      $num\_changed \leftarrow 0$
                      break
                  end if
              end for
          end if
          if $examine\_all = True$ then
              $examine\_all \leftarrow False$
          else if $num\_changed = 0$ then
              $examine\_all \leftarrow True$
          end if
      end while
      Compute $b$ by (126)
      return $\alpha^{+}, \alpha^{-}, b$
  end function
```

```
Require: $i1$-th Lagrange multiplier
Require: $i2$-th Lagrange multiplier
  function TakeStep($i1, i2$)
      if $i1 = i2$ then
          return False
      end if
      $finished = False$
      while not $finished$ do
          Compute $L$ and $H$ using (111), (114), (117) or (120)
          if $L < H$ then
              Compute $\eta$ by (103)        ▷ we assume that $\eta > 0$, i.e., the kernel matrix $K$ is positive definite
              if $\eta < 0$ then
                  Choose $\alpha_2^{+,new,clipped}$ or $\alpha_2^{-,new,clipped}$ between $L$ and $H$ according to the largest value of the objective function at these points
              else
                  Compute $\alpha_2^{+,new}$ or $\alpha_2^{-,new}$ using (112), (118) or (115), (121) respectively
                  Compute $\alpha_2^{+,new,clipped}$ or $\alpha_2^{-,new,clipped}$ by (105)
              end if
              Compute $\alpha_1^{+,new}$ or $\alpha_1^{-,new}$ using (113), (116) or (119), (122) respectively
              if changes in $\alpha_2^{+,new,clipped}, \alpha_2^{-,new,clipped}, \alpha_1^{+,new}$ or $\alpha_1^{-,new}$ are larger than some eps then
                  Update $\alpha_2^{+,new,clipped}, \alpha_2^{-,new,clipped}, \alpha_1^{+,new}$ or $\alpha_1^{-,new}$
              end if
          else
              $finished = True$
          end if
      end while
      if changes in $\alpha_2^{+,new,clipped}, \alpha_2^{-,new,clipped}, \alpha_1^{+,new}$ or $\alpha_1^{-,new}$ are larger than some eps then
          for $i$ in $I0$ do
              Update $errors_i$ using new Lagrange multipliers
          end for
          Update $\alpha^+$ and $\alpha^-$ using new Lagrange multipliers
          Update $I0, I1, I2$ and $I3$
          Update $errors_{i1}$ and $errors_{i2}$
          for $i$ in $I0 \cup \{i1, i2\}$ do
              Compute $(i_{low}, b_{low})$ by $b_{low} = \max\{errors_i : i \in I0 \cup I1 \cup I2\}$
              Compute and $(i_{up}, b_{up})$ by $b_{up} = \min\{errors_i : i \in I0 \cup I1 \cup I3\}$
          end for
          return True
      else
          return False
      end if
  end function
```

6.3 AdaGrad for Lagrangian Dual formulations

Due to the sparsity of the weight vector of the *Lagrangian dual*, i.e., the Lagrange multipliers, we might end up in a situation where some components of the gradient are very small and others large. This, in terms of *conditioning number*, i.e., $\kappa = L/\mu \gg 1$, means that the level sets of f are ellipsoid, i.e., we are dealing with an ill-conditioned problem. So, given a learning rate, a standard gradient descent approach might end up in a situation where it decreases too quickly the small weights or too slowly the large ones.

Another method, that is usually deprecated in ML applications due to its increased computational complexity, is Newton's method. Newton's method favors a much faster *convergence rate*, i.e., number of iterations, at the cost of being more expensive per iteration. For convex problems, the recursion is similar to the gradient descent algorithm:

$$x_{t+1} = x_t - \alpha H^{-1} \nabla f(x_t)$$

where α is often close to one (damped-Newton) or one, and H^{-1} denotes the Hessian of f at the current point, i.e., $\nabla^2 f(x_t)$.

The above suggest a general rule in optimization: find any preconditioner, in convex optimization it has to be positive semidefinite, that improves the performance of gradient descent in terms of iterations, but without wasting too much time to compute that precoditioner. The above result into:

$$x_{t+1} = x_t - \alpha P^{-1} \nabla f(x_t)$$

where P is the preconditioner. This idea is the basis of the BFGS quasi-Newton method.

The *AdaGrad* [6] algorithm is just a variant of preconditioned gradient descent, where P is selected to be a diagonal preconditioner matrix and is updated using the gradient information, in particular it is the diagonal approximation of the inverse of the square roots of gradient outer products, until the k-th iteration. The above lead to the algorithm:

Algorithm 7 AdaGrad

```
Require: Function $f$ to minimize
Require: Learning rate or step size $\alpha > 0$
Require: Offset $\epsilon > 0$ to ensures not divide by 0
  function AdaGrad($f, \alpha, \epsilon$)
    Initialize weight vector $x_0$ and the squared accumulated gradients vector $s_t \leftarrow 0$
    $t = 1$
    while $not\_convergence$ do
      $g_t \leftarrow \partial f(x_t)$                    ▷ if $f$ is differentiable then $\partial f(x_t) = \nabla f(x_t)$
      $s_t \leftarrow s_{t-1} + g_t^2$
      $x_{t+1} \leftarrow x_t - \alpha P_t^{-1} g_t = x_t - \frac{\alpha}{\sqrt{s_t + \epsilon}} \odot g_t$ where $P_t \leftarrow diag(s_t + \epsilon)^{1/2}$
      $t \leftarrow t + 1$
    end while
    return $x_t$
  end function
```

In practical terms, *AdaGrad* addresses the problem of the sparse optimal by adaptively scaling the learning rate for each dimension with the magnitude of the gradients. Coordinates that routinely correspond to large gradients are scaled down significantly, whereas others with small gradients receive a much more gentle treatment. *AdaGrad*'s *convergence rate* for L-Lipschitz continuous convex functions is the same of the SGD method described above.

6.4 Losses properties

Several losses and objectives have been presented in section 3 and 4. In our experiments, we will consider four different convex loss functions, two for the *classification* and two for the *regression* tasks. In particular, for what about the *margin-based* losses, i.e., the *classification* losses, both the *hinge* and the *squared hinge* losses are Lipschitz continuous and convex; meanwhile, for twhat about the *distance-based* losses, i.e., the *regression* losses, the *epsilon-insensitive* loss is Lipschitz continuous and convex but the *squared epsilon-insensitive* is convex and for this reason is locally Lipschitz continuous.

In general, if the objective function of a quadratic programming problem is strictly convex, i.e., the associated Hessian matrix is positive definite, the solution is unique. And if the objective function is convex, there may be cases where the solution is nonunique.

Assume that the hard margin SVM has a solution, i.e., the given problem is separable in the feature space. Then, since the objective function of the primal problem is $\frac{1}{2}\|w\|^2$, which is 1-strongly convex, the primal problem has a unique solution for w and b.

Since the $\mathcal{L}_1$-SVM linearly penalizes the misclassified points, the primal objective function is convex. Likewise, the Hessian matrix of the dual objective function is positive semidefinite. Thus the primal and dual solutions may be nonunique. Meanwhile, the objective function of the primal problem for the $\mathcal{L}_2$-SVM is strictly convex, due to the quadratic penalization of the misclassified points. Therefore, w and b are uniquely determined if we solve the primal or dual problem. In summary, the following properties for the SVM's objectives are given:

Table 1: SVM's objectives properties for primal formulations

objective	smooth	Lipschitz continuous	convexity
$\mathcal{L}_1$-SVC (13)	no	globally	convex
$\mathcal{L}_2$-SVC (43)	yes	globally	1-strongly convex
$\mathcal{L}_1$-SVR (58)	no	globally	convex
$\mathcal{L}_2$-SVR (87)	yes	locally	1-strongly convex

And, according to the theoretical analysis, the following *convergence rates* are given for the primal and Lagrangian dual formulations respectively:

Table 2: SVM's objectives convergence rates for primal formulations

objective	SGD convergence rate	Polyak SGD convergence rate	Nesterov SGD convergence rate
$\mathcal{L}_1$-SVM (13, 58)	$\mathcal{O}\left(\frac{1}{\sqrt{t}}\right)$	$\mathcal{O}\left(\frac{1}{\sqrt{t}}\right)$	$\mathcal{O}\left(\frac{1}{\sqrt{t}}\right)$
$\mathcal{L}_2$-SVM (43, 87)	$\mathcal{O}\left(\frac{1}{t}\right)$	$\mathcal{O}\left(\frac{1}{t}\right)$	$\mathcal{O}\left(\exp\left(-\frac{t}{\sqrt{\kappa}}\right)\right)$

Table 3: SVM's objectives convergence rate for Lagrangian dual formulations

objective	AdaGrad convergence rate
$\mathcal{L}_1$-SVM (27, 71) or (34, 78)	$\mathcal{O}\left(\frac{1}{\sqrt{t}}\right)$

7 Experiments

The following experiments refer to *linearly* and *nonlinearly* separable generated datasets of size 100.

7.1 Support Vector Classifier

Below experiments are about the SVC for which I tested different values for the regularization hyperparameter C, i.e., from *soft* to *hard margin*, and in case of nonlinearly separable data also different *kernel functions* mentioned above.

7.1.1 Hinge loss

Primal formulation The experiments results shown in 4 referred to *Stochastic Gradient Descent* algorithm are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 and β, i.e., the *momentum*, equal to 0.4. The batch size is setted to 20. Training is stopped if after 5 iterations the training loss is not lower than the best found so far.

Table 4: SVC Primal formulation results with Hinge loss

			fit_time	accuracy	n_iter	n_sv
solver	momentum	C				
sgd	none	1	0.399575	0.970	314	53
		10	0.480145	0.985	384	19
		100	0.258918	0.980	207	10
	standard	1	0.305286	0.970	229	48
		10	0.350044	0.985	296	16
		100	0.204298	0.980	124	11
	nesterov	1	0.301714	0.970	229	48
		10	0.345403	0.985	288	16
		100	0.159904	0.985	130	11
liblinear	-	1	0.001039	0.985	332	15
		10	0.001164	0.985	554	5
		100	0.001599	0.985	1000	7

The results provided from the *custom* implementation, i.e., the SGD with different momentum settings, are strongly similar to those of *sklearn* implementation, i.e., *liblinear* [10] implementation, in terms of *accuracy* score. More training data points are selected as *support vectors* from the SGD solver but it always requires lower iterations, i.e., epochs, to achieve the same *numerical precision. Standard* or *Polyak* and *Nesterov* momentums always perform lower iterations as expected from the theoretical analysis of the convergence rate.

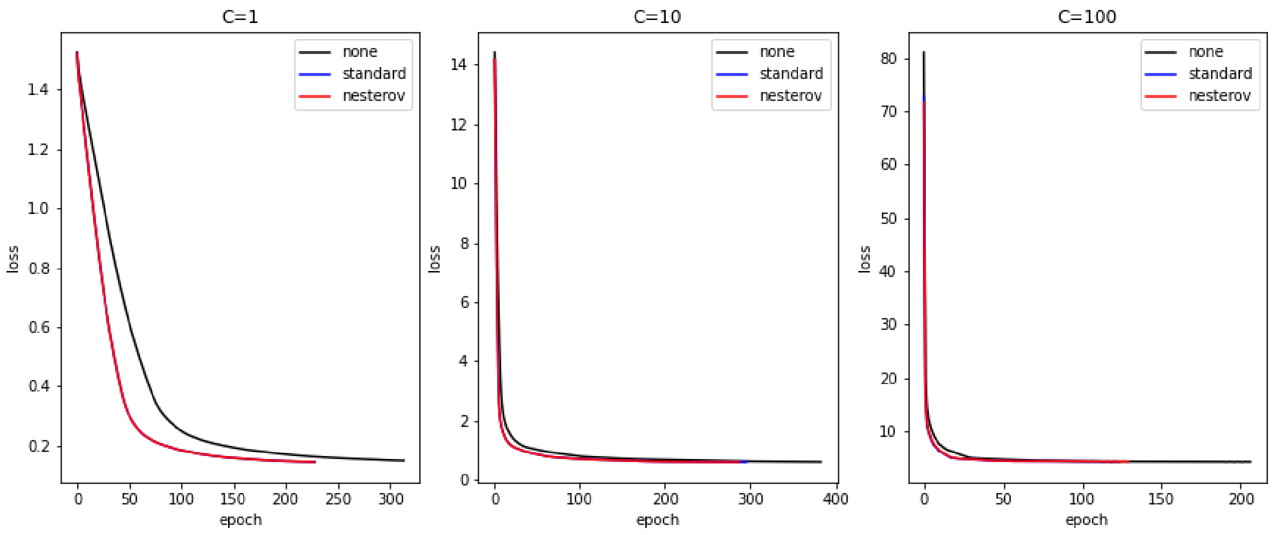


Figure 11: Loss convergence for the Primal formulation of the $\mathcal{L}_1$-SVC

Linear Dual formulations The experiments results shown in 6 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 for the *AdaGrad* algorithm. Notice that the *qp* dual refers to the formulation (27), while the *bcqp* dual refers to the formulation (34).

Table 5: Linear SVC Wolfe Dual formulation results with Hinge loss

		fit_time	accuracy	n_iter	n_sv
solver	C				
smo	1	0.052323	0.980	62	17
	10	0.091658	0.980	295	10
	100	0.137948	0.985	399	8
libsvm	1	0.002719	0.985	243	17
	10	0.002529	0.985	194	10
	100	0.002797	0.985	1602	8
cvxopt	1	0.020693	0.980	10	17
	10	0.023231	0.980	10	11
	100	0.064602	0.985	10	8

For what about the linear *Wolfe dual* formulation we can immediately notice as higher *regularization hyperparameter C* makes the model harder, so the *custom* implementation of the SMO algorithm and also the *sklearn* implementation, i.e., *libsvm* [11] implementation, needs to perform more iterations to achieve the same *numerical precision*; meanwhile the *cvxopt* [12] seems to be insensitive to the increasing complexity of the model. The results in terms of *accuracy* and number of *support vectors* are strongly similar to each others.

Table 6: Linear SVC Lagrangian Dual formulation results with Hinge loss

dual	C	fit_time	accuracy	n_iter	n_sv
qp	1	0.040198	0.985	1	195
	10	0.028154	0.985	1	195
	100	0.024502	0.985	1	195
bcqp	1	0.030617	0.985	1	194
	10	0.029585	0.985	1	194
	100	0.024226	0.985	1	194

For what about the linear *Lagrangian dual* formulation we can see as it seems to be insensitive to the increasing complexity of the model in terms of number of *iterations* but it tends to select many training data points as *support vectors*.

Nonlinear Dual formulations The experiments results shown in 7 and 8 are obtained with d and r hyperparameters equal to 3 and 1 respectively for the *polynomial* kernel; *gamma* is setted to *'scale'* for both *polynomial* and *gaussian RBF* kernels. The experiments results shown in 8 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 for the *AdaGrad* algorithm.

Table 7: Nonlinear SVC Wolfe Dual formulation results with Hinge loss

solver	kernel	C	fit_time	accuracy	n_iter	n_sv
smo	poly	1	0.262513	0.6825	143	30
		10	0.197470	0.9475	65	10
		100	0.140720	0.9775	38	6
	rbf	1	0.289481	1.0000	66	51
		10	0.143812	1.0000	38	13
		100	0.196509	1.0000	56	12
libsvm	poly	1	0.005542	1.0000	233	30
		10	0.004144	1.0000	118	10
		100	0.004042	1.0000	88	6
	rbf	1	0.005162	1.0000	252	50
		10	0.003994	1.0000	134	13
		100	0.005558	1.0000	182	12
cvxopt	poly	1	0.208703	0.6775	10	31
		10	0.194188	0.9475	10	10
		100	0.228154	0.9775	10	6
	rbf	1	0.177948	1.0000	10	50
		10	0.239361	1.0000	10	19
		100	0.206783	1.0000	10	17

Table 8: Nonlinear SVC Lagrangian Dual formulation results with Hinge loss

			fit_time	accuracy	n_iter	n_sv
dual	kernel	C				
qp	poly	1	0.160826	0.640	3	316
		10	0.034587	0.640	3	316
		100	0.053057	0.640	3	316
	rbf	1	0.112672	0.860	9	307
		10	0.144244	0.860	9	307
		100	0.148839	0.860	9	307
bcqp	poly	1	1.139823	0.635	222	317
		10	1.106476	0.635	222	317
		100	1.317503	0.635	222	317
	rbf	1	0.115288	1.000	1	399
		10	0.057738	1.000	1	399
		100	0.092722	1.000	1	399

The same considerations made for the previous linear *Wolfe dual* and *Lagrangian dual* formulations are confirmed also in the nonlinearly separable case. In this setting the complexity of the model coming with higher C regularization values seems to be not paying a tradeoff in terms of the number of *iterations* of the algorithm and, moreover, the *bcqp Lagrangian dual* formulation seems to perform better wrt the *qp* formulation, both tends to select even more training data points as *support vectors*.

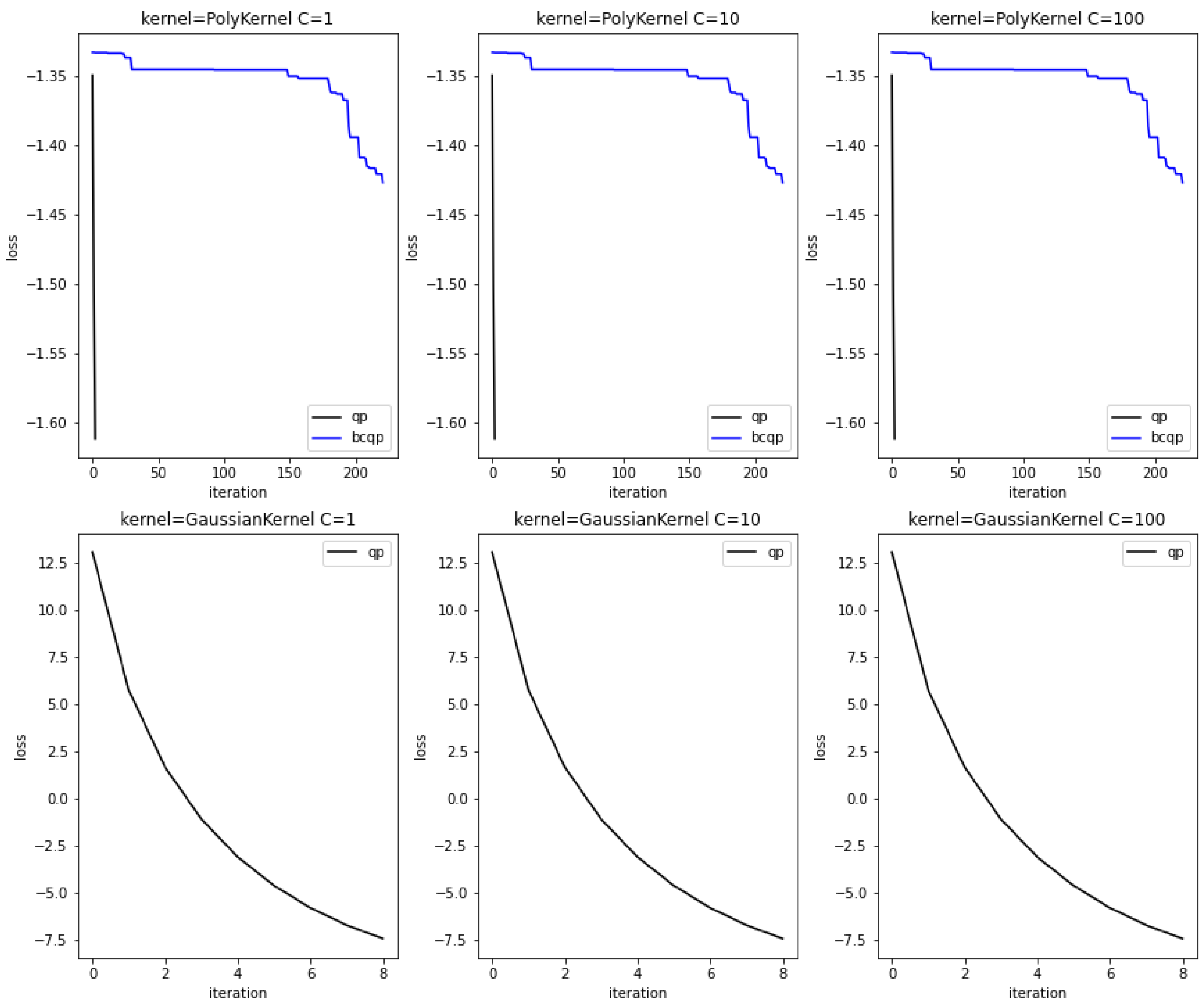


Figure 12: Loss convergence for the Lagrangian Dual formulation of the Nonlinear $\mathcal{L}_1$-SVC

7.1.2 Squared Hinge loss

Primal formulation The experiments results shown in 9 referred to *Stochastic Gradient Descent* algorithm are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 and β, i.e., the *momentum*, equal to 0.4. The batch size is setted to 20. Training is stopped if after 5 iterations the training loss is not lower than the best found so far.

Table 9: SVC Primal formulation results with Squared Hinge loss

			fit_time	accuracy	n_iter	n_sv
solver	momentum	C				
sgd	none	1	0.266864	0.970	118	49
		10	0.057214	0.975	38	31
		100	0.020525	0.975	11	18
	standard	1	0.218742	0.970	86	44
		10	0.042822	0.975	24	29
		100	0.023315	0.975	12	14
	nesterov	1	0.223442	0.970	80	46
		10	0.048704	0.975	25	29
		100	0.018014	0.975	10	15
liblinear	-	1	0.006113	0.980	500	42
		10	0.010983	0.975	1000	38
		100	0.010572	0.975	1000	36

Again, the results provided from the *custom* implementation, i.e., the SGD with different momentum settings, are strongly similar to those of *sklearn* implementation, i.e., *liblinear* [10] implementation, in terms of *accuracy* score. More training data points are selected as *support vectors* from the SGD solver but it always requires even lower iterations, i.e., epochs, to achieve the same *numerical precision*. *Standard* or *Polyak* and *Nesterov* momentums always perform lower iterations as expected from the theoretical analysis of the convergence rate.

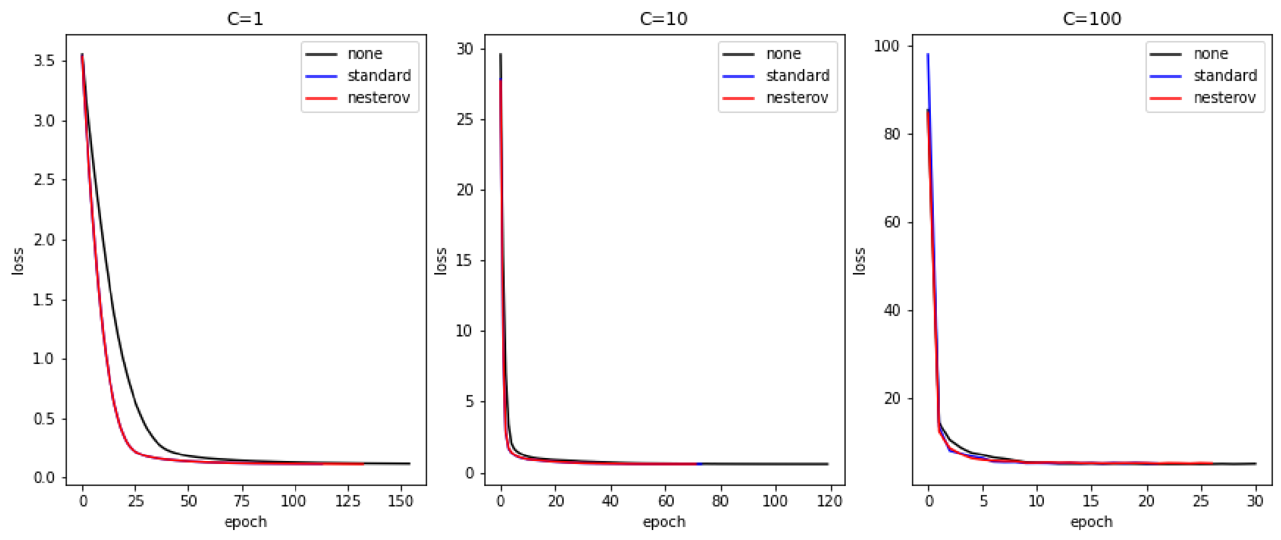


Figure 13: Loss convergence for the Primal formulation of the $\mathcal{L}_2$-SVC

7.2 Support Vector Regression

Below experiments are about the SVR for which I tested different values for regularization hyperparameter C, i.e., from *soft* to *hard margin*, the ϵ penalty value and in case of nonlinearly separable data also different *kernel functions* mentioned above.

7.2.1 Epsilon-insensitive loss

Primal formulation The experiments results shown in 10 referred to *Stochastic Gradient Descent* algorithm are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 and β, i.e., the *momentum*, equal to 0.4. The batch size is setted to 20. Training is stopped if after 5 iterations the training loss is not lower than the best found so far.

Table 10: SVR Primal formulation results with Epsilon-insensitive loss

				fit_time	r2	n_iter	n_sv
solver	momentum	C	epsilon				
sgd	none	1	0.1	0.072204	0.476051	61	100
			0.2	0.053251	0.476051	61	100
			0.3	0.054258	0.476051	61	100
		10	0.1	0.058096	0.976336	65	99
			0.2	0.066921	0.976335	65	97
			0.3	0.059388	0.976335	65	96
		100	0.1	0.026680	0.977543	29	99
			0.2	0.028083	0.977543	28	98
			0.3	0.032191	0.977543	32	96
	standard	1	0.1	0.049673	0.496182	39	100
			0.2	0.040283	0.496182	39	100
			0.3	0.036260	0.496182	39	100
		10	0.1	0.037528	0.976716	41	100
			0.2	0.037285	0.976717	41	100
			0.3	0.043476	0.976717	41	98
		100	0.1	0.010734	0.977554	11	98
			0.2	0.010965	0.977554	11	97
			0.3	0.011849	0.977554	11	97
	nesterov	1	0.1	0.049683	0.495933	39	100
			0.2	0.050985	0.495933	39	100
			0.3	0.035211	0.495933	39	100
		10	0.1	0.037942	0.976679	41	100
			0.2	0.038659	0.976678	41	100
			0.3	0.040040	0.976679	41	98
		100	0.1	0.011368	0.977550	12	99
			0.2	0.011522	0.977550	12	97
			0.3	0.010772	0.977550	11	97
liblinear	-	1	0.1	0.001035	0.964163	12	97
			0.2	0.001166	0.963786	12	97
			0.3	0.000866	0.963443	18	95
		10	0.1	0.000785	0.977559	111	100
			0.2	0.001241	0.977552	126	99
			0.3	0.001293	0.977564	253	98
		100	0.1	0.001776	0.977481	672	99
			0.2	0.001200	0.977442	881	99
			0.3	0.002320	0.977458	1000	98

The results provided from the *custom* implementation, i.e., the SGD with different momentum settings, are strongly similar to those of *sklearn* implementation, i.e., *liblinear* [10] implementation, in terms of *r2* score, except in case of C regularization hyperparameter equals to 1 for which those of SGD are lower. Moreover, the SGD solver always requires lower iterations, i.e., epochs, for higher C regularization values, i.e., for C equals to 10 or 100, to achieve the same *numerical precision.* Again, *Standard* or *Polyak* and *Nesterov* momentums always perform lower iterations as expected from the theoretical analysis of the convergence rate. The results in terms of *support vectors* are strongly similar to each others.

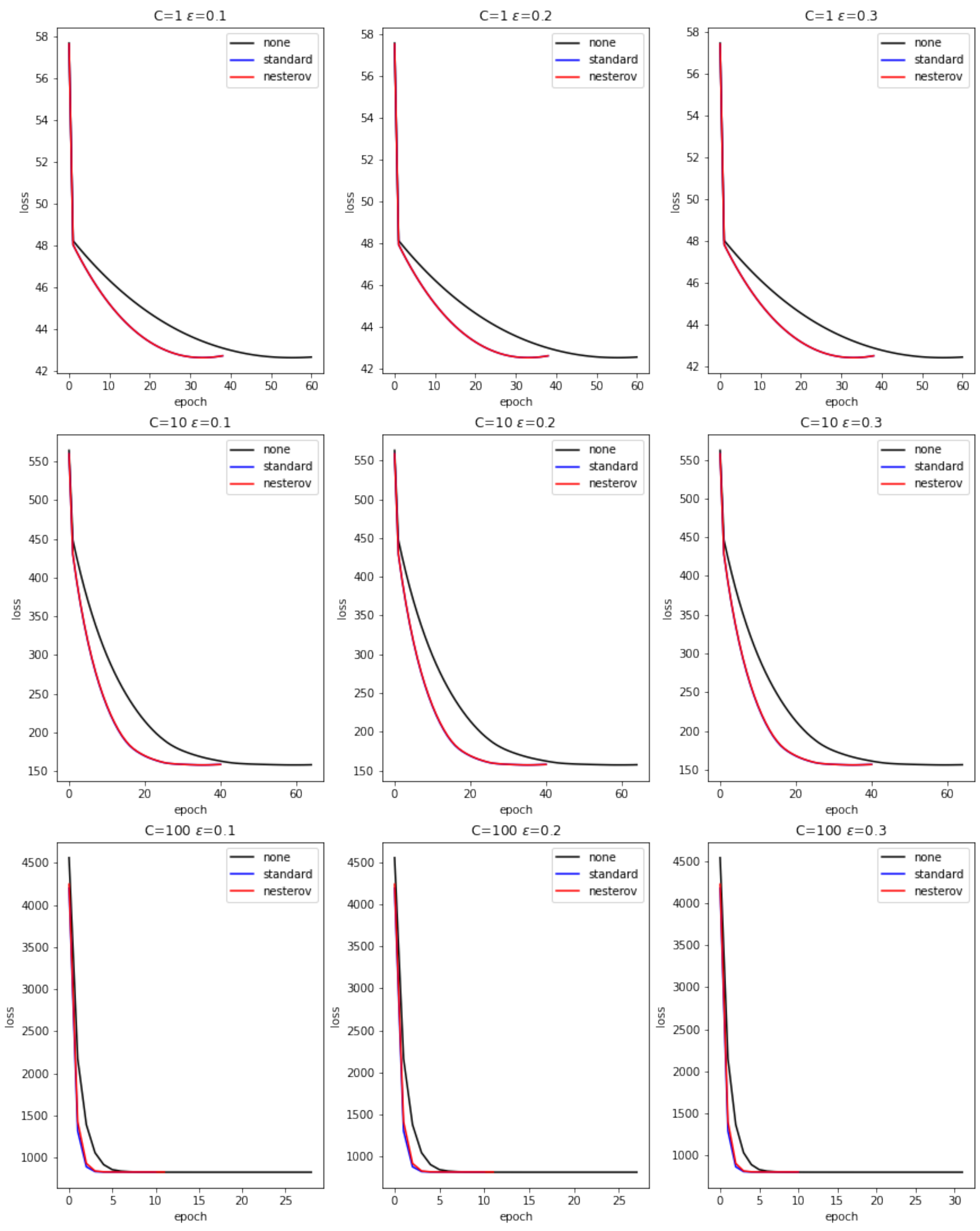


Figure 14: Loss convergence for the Primal formulation of the $\mathcal{L}_1$-SVR

Linear Dual formulations The experiments results shown in 12 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 for the *AdaGrad* algorithm. Notice that the *qp* dual refers to the formulation (71), while the *bcqp* dual refers to the formulation (78).

Table 11: Linear SVR Wolfe Dual formulation results with Epsilon-insensitive loss

			fit_time	r2	n_iter	n_sv
solver	C	epsilon				
smo	1	0.1	0.023381	0.964127	17	98
		0.2	0.044661	0.963707	18	96
		0.3	0.035539	0.963707	14	96
	10	0.1	0.137995	0.977576	69	100
		0.2	0.500972	0.977573	749	100
		0.3	0.129166	0.977573	78	99
	100	0.1	0.544417	0.977515	549	100
		0.2	0.471177	0.977496	723	100
		0.3	0.512489	0.977493	926	99
libsvm	1	0.1	0.003990	0.964103	81	98
		0.2	0.005355	0.963680	81	97
		0.3	0.007024	0.963684	78	96
	10	0.1	0.009187	0.977559	226	100
		0.2	0.005927	0.977554	706	100
		0.3	0.011225	0.977564	181	99
	100	0.1	0.008480	0.977481	1224	100
		0.2	0.017160	0.977450	2126	100
		0.3	0.008723	0.977463	2680	99
cvxopt	1	0.1	0.044434	0.964127	10	100
		0.2	0.040610	0.963709	9	100
		0.3	0.050188	0.963706	9	100
	10	0.1	0.043882	0.977576	8	100
		0.2	0.059017	0.977573	9	100
		0.3	0.037176	0.977573	9	99
	100	0.1	0.057136	0.977515	8	100
		0.2	0.045696	0.977496	9	100
		0.3	0.033306	0.977493	9	100

For what about the linear *Wolfe dual* formulation we can immediately notice as higher *regularization hyper-parameter C* and lower ϵ values makes the model harder, so the *custom* implementation of the SMO algorithm and also the *sklearn* implementation, i.e., *libsvm* [11] implementation, needs to perform more iterations to achieve the same *numerical precision*; meanwhile, again, the *cvxopt* [12] seems to be insensitive to the increasing complexity of the model. The results in terms of *r2* and number of *support vectors* are strongly similar to each others.

Table 12: Linear SVR Lagrangian Dual formulation results with Epsilon-insensitive loss

			fit_time	r2	n_iter	n_sv
dual	C	epsilon				
qp	1	0.1	1.138390	0.731400	1000	100
		0.2	1.259378	0.731400	1000	100
		0.3	1.145181	0.731400	1000	100
	10	0.1	0.953196	0.731400	1000	100
		0.2	1.116884	0.731400	1000	100
		0.3	1.346315	0.731400	1000	100
	100	0.1	1.268649	0.731400	1000	100
		0.2	1.016300	0.731400	1000	100
		0.3	1.278405	0.731400	1000	100
bcqp	1	0.1	1.681605	0.733183	1000	100
		0.2	1.076056	0.733183	1000	100
		0.3	1.286736	0.733183	1000	100
	10	0.1	1.470822	0.733183	1000	100
		0.2	1.387297	0.733183	1000	100
		0.3	1.697497	0.733183	1000	100
	100	0.1	1.848502	0.733183	1000	100
		0.2	1.179248	0.733183	1000	100
		0.3	1.196528	0.733183	1000	100

For what about the linear *Lagrangian dual* formulation we can see as it seems to be insensitive to the increasing complexity of the model in terms of number of *iterations* and require many *iterations* wrt the *Wolfe dual* formulation.

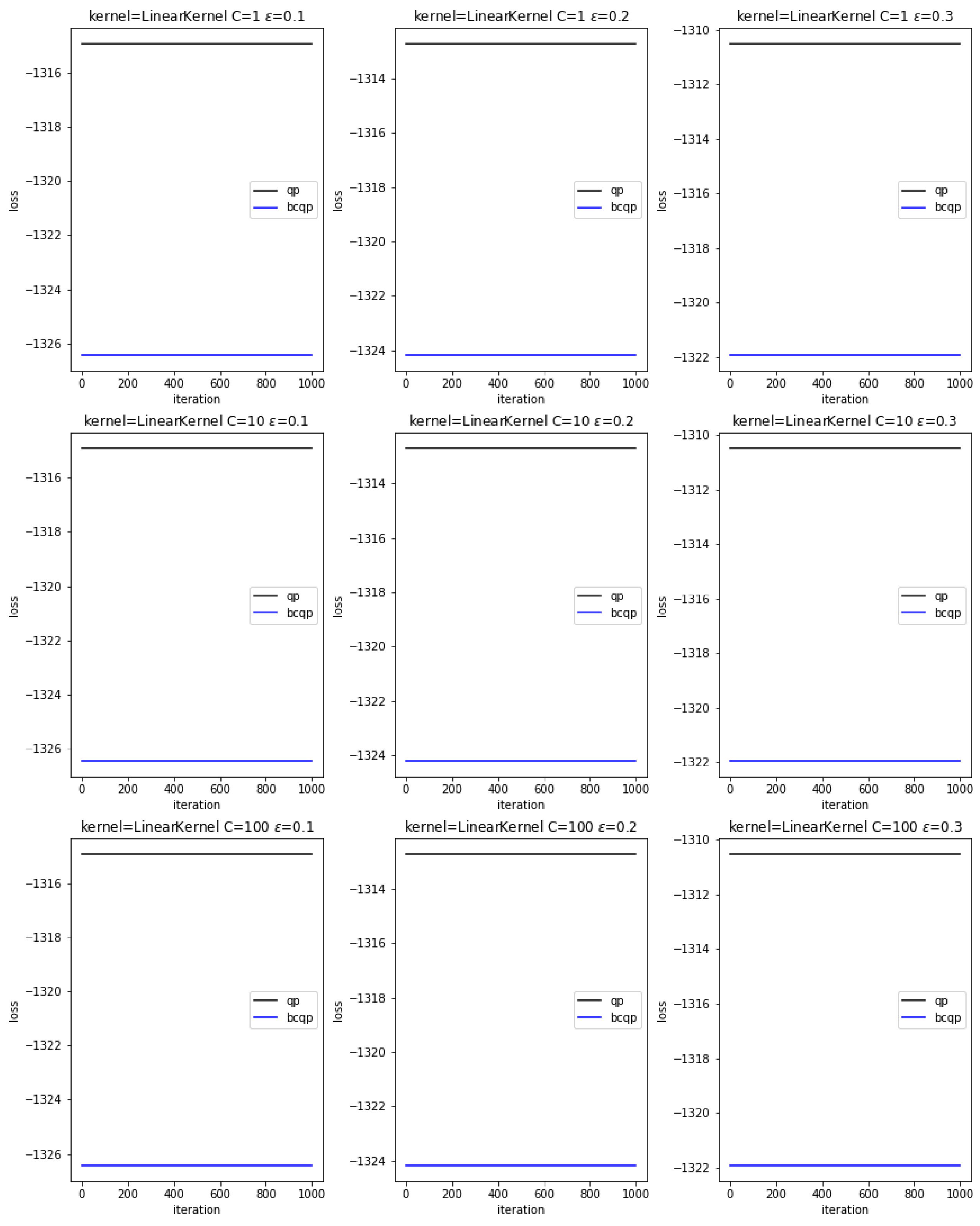


Figure 15: Loss convergence for the Lagrangian Dual formulation of the Linear $\mathcal{L}_1$-SVR

Nonlinear Dual formulations The experiments results shown in 13 and 14 are obtained with d and r hyperparameters both equal to 3 for the *polynomial* kernel; *gamma* is setted to *'scale'* for both *polynomial* and *gaussian RBF* kernels. The experiments results shown in 8 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 for the *AdaGrad* algorithm.

Table 13: Nonlinear SVR Wolfe Dual formulation results with Epsilon-insensitive loss

				fit_time	r2	n_iter	n_sv
solver	kernel	C	epsilon				
smo	poly	1	0.1	16.621604	0.965958	22565	28
			0.2	10.662649	0.915386	18370	7
			0.3	2.206419	-0.019348	2577	4
		10	0.1	162.003244	0.873457	269916	29
			0.2	5.475889	0.773653	7385	4
			0.3	2.235514	-0.019348	2577	4
		100	0.1	1570.878271	0.873592	2837623	29
			0.2	5.905676	0.773653	7385	4
			0.3	1.954492	-0.019348	2577	4
	rbf	1	0.1	0.162744	0.989251	58	20
			0.2	0.042457	0.916460	30	6
			0.3	0.017082	0.863690	7	5
		10	0.1	0.962114	0.988808	855	22
			0.2	0.023530	0.916260	26	6
			0.3	0.010618	0.863690	7	5
		100	0.1	4.986097	0.986641	4714	19
			0.2	0.019692	0.916260	26	6
			0.3	0.017319	0.863690	7	5
libsvm	poly	1	0.1	0.033172	0.982113	58829	28
			0.2	0.010934	0.974153	15817	7
			0.3	0.009176	0.946442	2697	4
		10	0.1	0.166147	0.982366	477441	29
			0.2	0.008310	0.979179	5598	4
			0.3	0.005829	0.946442	2697	4
		100	0.1	2.230161	0.982119	10669851	29
			0.2	0.020086	0.979179	5598	4
			0.3	0.004748	0.946442	2697	4
	rbf	1	0.1	0.009822	0.989178	80	20
			0.2	0.013293	0.982040	37	6
			0.3	0.001457	0.951730	18	5
		10	0.1	0.003906	0.990056	833	20
			0.2	0.007880	0.982014	38	6
			0.3	0.002544	0.951730	18	5
		100	0.1	0.011858	0.990542	12361	19
			0.2	0.005540	0.982014	38	6
			0.3	0.001965	0.951730	18	5
cvxopt	poly	1	0.1	0.087879	0.966650	8	28
			0.2	0.162963	0.389408	10	6
			0.3	0.103764	0.070818	10	4
		10	0.1	0.133711	0.824379	9	30
			0.2	0.139223	0.777645	10	4
			0.3	0.069593	0.070820	10	4
		100	0.1	0.044351	0.952412	9	72
			0.2	0.033852	0.777643	10	4
			0.3	0.036072	0.070818	10	4
	rbf	1	0.1	0.042391	0.989236	10	20
			0.2	0.033710	0.915828	9	6
			0.3	0.050085	0.858355	9	6
		10	0.1	0.043324	0.987421	10	21
			0.2	0.032647	0.915828	10	6
			0.3	0.044080	0.861873	10	5
		100	0.1	0.043129	0.990163	10	29
			0.2	0.025641	0.915828	10	6
			0.3	0.046823	0.861873	10	5

Table 14: Nonlinear SVR Lagrangian Dual formulation results with Epsilon-insensitive loss

				fit_time	r2	n_iter	n_sv
dual	kernel	C	epsilon				
qp	poly	1	0.1	1.426305	0.536892	1000	100
			0.2	1.947305	0.536886	1000	100
			0.3	1.597898	0.529220	1000	100
		10	0.1	1.569864	0.536892	1000	100
			0.2	1.659373	0.536886	1000	100
			0.3	2.155713	0.529220	1000	100
		100	0.1	2.127201	0.536892	1000	100
			0.2	2.003470	0.536886	1000	100
			0.3	1.385318	0.529220	1000	100
	rbf	1	0.1	0.333465	0.733767	128	100
			0.2	1.326574	0.718224	640	100
			0.3	2.035441	0.580564	1000	100
		10	0.1	0.369488	0.733767	128	100
			0.2	1.663891	0.718224	640	100
			0.3	1.971318	0.580564	1000	100
		100	0.1	0.366429	0.733767	128	100
			0.2	1.615145	0.718224	640	100
			0.3	1.913055	0.580564	1000	100
bcqp	poly	1	0.1	1.363023	0.536344	1000	100
			0.2	1.468118	0.536338	1000	100
			0.3	1.414094	0.528759	1000	100
		10	0.1	1.700776	0.536344	1000	100
			0.2	1.394834	0.536338	1000	100
			0.3	1.842973	0.528759	1000	100
		100	0.1	2.344345	0.536344	1000	100
			0.2	1.843466	0.536338	1000	100
			0.3	1.377408	0.528759	1000	100
	rbf	1	0.1	3.017241	0.739809	1000	100
			0.2	0.320919	0.717846	165	100
			0.3	0.398222	0.632389	185	100
		10	0.1	2.460307	0.739809	1000	100
			0.2	0.334468	0.717846	165	100
			0.3	0.389869	0.632389	185	100
		100	0.1	2.811035	0.739809	1000	100
			0.2	0.336349	0.717846	165	100
			0.3	0.400294	0.632389	185	100

The same considerations made for the previous linear *Wolfe dual* and *Lagrangian dual* formulations are confirmed also in the nonlinearly separable case. In this setting, the complexity of the model coming with higher C regularization hyperparameters and lower ϵ values pays a larger tradeoff in terms of the number of *iterations* of the algorithm.

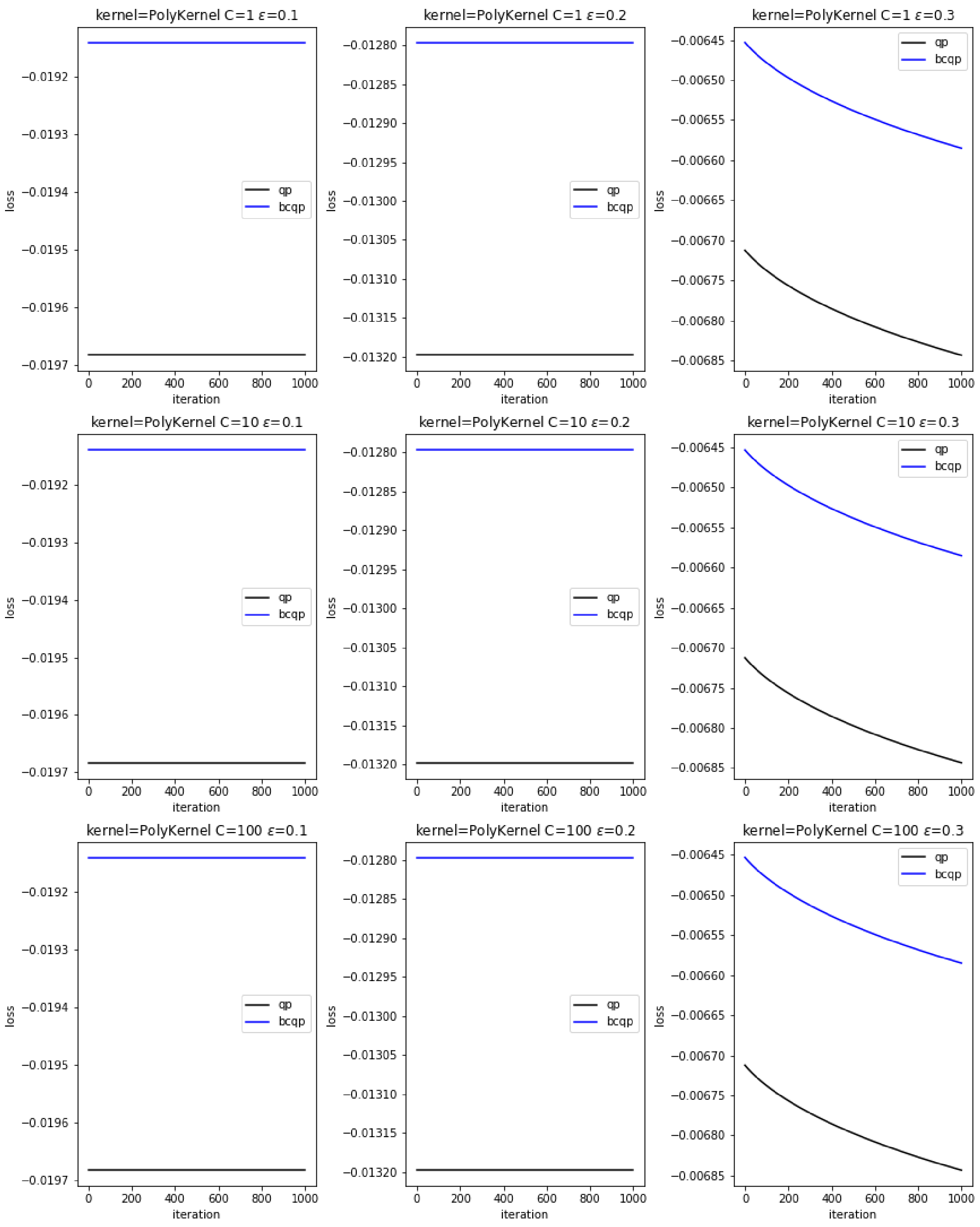


Figure 16: Loss convergence for the Lagrangian Dual formulation of the Polynomial $\mathcal{L}_1$-SVR

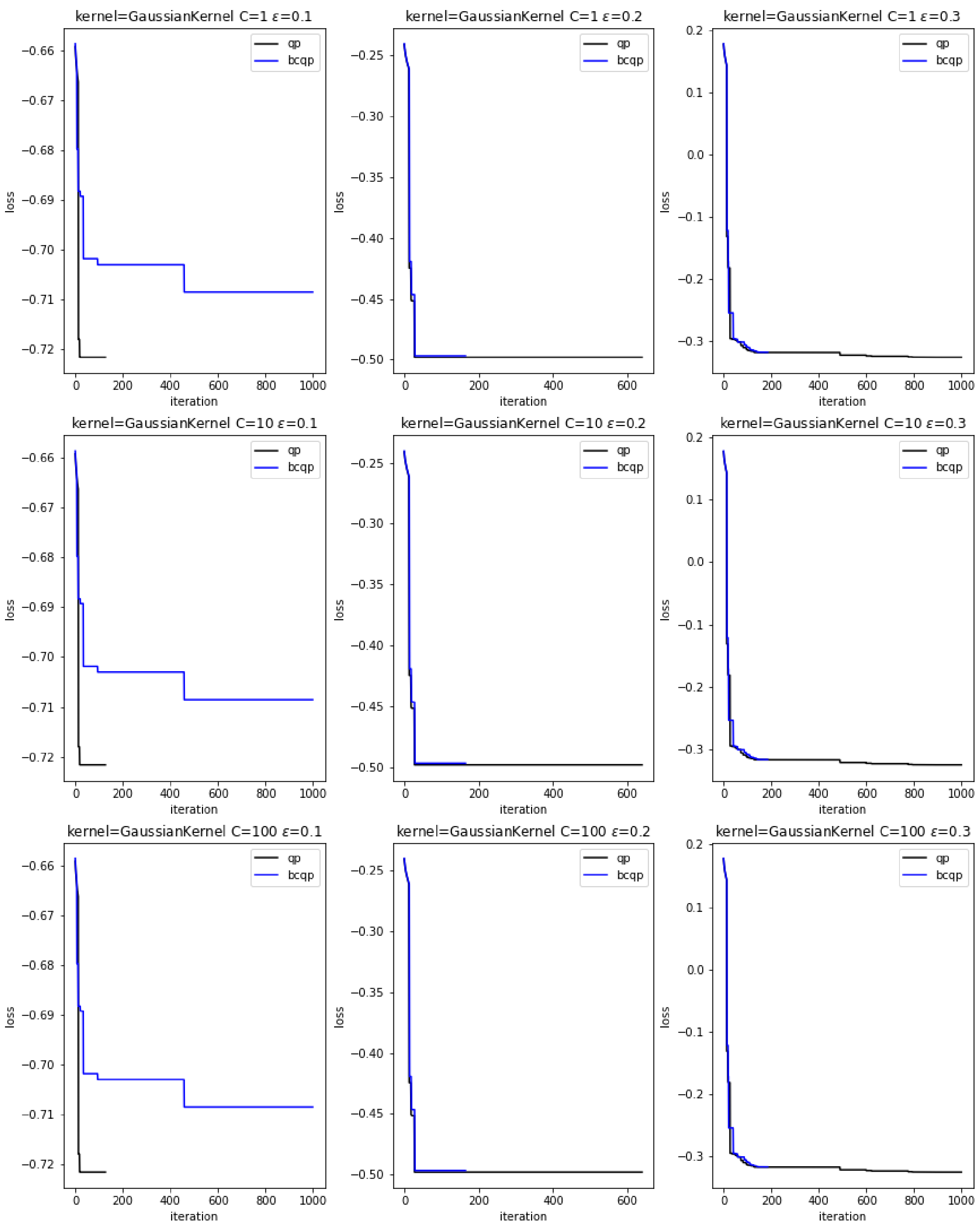


Figure 17: Loss convergence for the Lagrangian Dual formulation of the Gaussian $\mathcal{L}_1$-SVR

7.2.2 Squared Epsilon-insensitive loss

Primal formulation The experiments results shown in 15 referred to *Stochastic Gradient Descent* algorithm are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.001 and β, i.e., the *momentum*, equal to 0.4. The batch size is setted to 20. Training is stopped if after 5 iterations the training loss is not lower than the best found so far.

Table 15: SVR Primal formulation results with Squared Epsilon-insensitive loss

				fit_time	r2	n_iter	n_sv
solver	momentum	C	epsilon				
sgd	none	1	0.1	0.746595	0.977025	641	100
			0.2	0.801106	0.977021	633	99
			0.3	0.864816	0.977016	625	99
		10	0.1	0.111896	0.977573	74	100
			0.2	0.111459	0.977572	70	99
			0.3	0.086778	0.977572	72	99
		100	0.1	0.010679	0.977409	9	100
			0.2	0.011634	0.977408	9	99
			0.3	0.010993	0.977407	9	98
	standard	1	0.1	0.570478	0.977035	398	100
			0.2	0.427103	0.977031	392	99
			0.3	0.502552	0.977025	385	99
		10	0.1	0.046568	0.977572	42	99
			0.2	0.044361	0.977571	40	99
			0.3	0.047390	0.977572	42	99
		100	0.1	0.008482	0.977439	7	99
			0.2	0.008512	0.977438	7	99
			0.3	0.008432	0.977441	7	98
	nesterov	1	0.1	0.537362	0.977035	399	100
			0.2	0.544074	0.977030	392	99
			0.3	0.441734	0.977026	386	99
		10	0.1	0.052316	0.977572	43	99
			0.2	0.081535	0.977572	42	99
			0.3	0.079432	0.977570	40	99
		100	0.1	0.014036	0.977411	7	100
			0.2	0.013736	0.977411	7	99
			0.3	0.013420	0.977413	7	98
liblinear	-	1	0.1	0.004967	0.977554	96	100
			0.2	0.002236	0.977553	96	100
			0.3	0.001312	0.977551	96	100
		10	0.1	0.005617	0.977577	826	100
			0.2	0.007077	0.977576	826	99
			0.3	0.004803	0.977576	839	99
		100	0.1	0.006345	0.977538	1000	100
			0.2	0.004969	0.977540	1000	99
			0.3	0.008882	0.977541	1000	98

Again, the results provided from the *custom* implementation, i.e., the SGD with different momentum settings, are strongly similar to those of *sklearn* implementation, i.e., *liblinear* [10] implementation, in terms of *r2* score. SGD solver always requires even lower iterations, i.e., epochs, for higher C regularization values, i.e., for C equals to 10 or 100, to achieve the same *numerical precision*. *Standard* or *Polyak* and *Nesterov* momentums always perform lower iterations as expected from the theoretical analysis of the convergence rate.

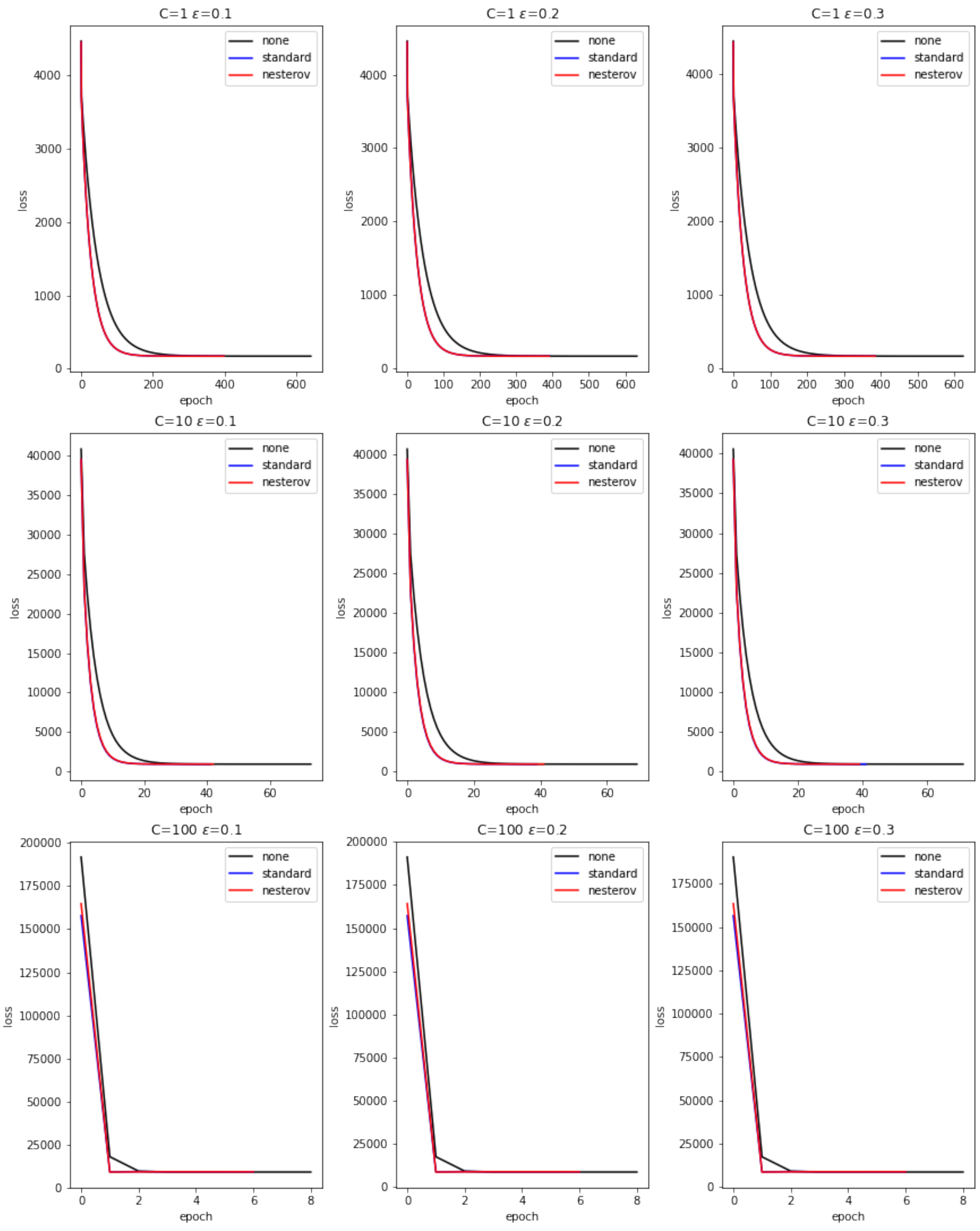


Figure 18: Loss convergence for the Primal formulation of the $\mathcal{L}_2$-SVR

8 Conclusions

For what about the SVM formulations, it is known, in general, that the *primal formulation*, is suitable for large linear training since the complexity of the model grows with the number of features or, more in general, when the number of examples n is much larger than the number of features m, i.e., $n \gg m$; meanwhile the *dual formulation*, is more suitable in case the number of examples n is less than the number of features m, i.e., $n < m$, since the complexity of the model is dominated by the number of examples, or more in general when the training data are not linearly separable in the input space.

From all these experiments we can see as all the *custom* implementations underperforms all the others, i.e., both *cvxopt* [12] and *sklearn* implementations, i.e., *liblinear* [10] and *libsvm* [11] implementations, in terms of *time* obviously due to the different core implementation languages, i.e., Python and C respectively.

In the *primal* formulations the *liblinear* [10] implementation uses an optimization method called *Coordinate Gradient Descent* which minimizes one coordinate at a time.

Meanwhile, for what about the *Wolfe dual* formulations we can notice as *cvxopt* [12] underperforms the *sklearn* implementation, i.e., *libsvm* [11] implementation, in terms of *time* since it is a general-purpose QP solver and it does not exploit the structure of the problem, as SMO does. An interesting consideration can be made about the number of *iterations* of *custom* SMO implementation wrt that in *libsvm* which seems to be always lower thanks to the improvements described in [5, 8] for classification and regression respectively.

Finally, in the *Lagrangian dual* formulations the goodness of the solution in terms of *accuracy* or *r2* values depends on the residue in the solution of the *Lagrangian dual* at each step provided by *minres* algorithm. Moreover, we can see as fitting the intercept in an explicit way, i.e., by adding Lagrange multipliers to control the equality constraint always get lower scores wrt the *Lagrangian dual* of the same problem with the bias term embedded into the weight matrix.

References

[1] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[2] Yurii Nesterov. Introductory lectures on convex programming volume i: Basic course. *Lecture notes*, 3(4):5, 1998.

[3] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[4] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[5] S. Sathiya Keerthi, Shirish Krishnaj Shevade, Chiranjib Bhattacharyya, and Karuturi Radha Krishna Murthy. Improvements to Platt's SMO algorithm for SVM classifier design. *Neural computation*, 13(3):637–649, 2001.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[7] Gary William Flake and Steve Lawrence. Efficient SVM regression training with SMO. *Machine Learning*, 46(1):271–290, 2002.

[8] SK Shevade, SS Keerthi, C Bhattacharyya, and KRK Murthy. Improvements to SMO algorithm for SVM regression (Tech. Rep. No. CD-99-16). *Singapore: Control Division Department of Mechanical and Production Engineering*, 1999.

[9] Tristan Fletcher. Support vector machines explained. *Tutorial paper., Mar*, page 28, 2009.

[10] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *the Journal of machine Learning research*, 9:1871–1874, 2008.

[11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[12] Lieven Vandenberghe. The CVXOPT linear and quadratic cone program solvers. *Online: http://cvxopt. org/documentation/coneprog. pdf*, 2010.

[13] Veronica Piccialli and Marco Sciandrone. Nonlinear optimization and support vector machines. *4OR*, 16(2):111–149, 2018.

[14] Chih-Wei Hsu and Chih-Jen Lin. A simple decomposition method for support vector machines. *Machine Learning*, 46(1):291–314, 2002.

[15] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.