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

March, 2021

Contents

List of Figures

List of Tables

List of Algorithms

1 Track

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

- `(A1.1.1)` is a *momentum descent* approach [1, 2], an *accelerated gradient* method for solving the SVC in its *primal* formulation.
- `(A1.1.2)` is the *Sequential Minimal Optimization (SMO)* algorithm [3, 4], 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 [5], 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], 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], an *accelerated gradient* method for solving the SVR in its *primal* formulation.
- `(A2.1.2)` is the *Sequential Minimal Optimization (SMO)* algorithm [6, 7], 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 [5], 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], 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 constructs 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* [8] 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 performace of different *kernels*, also by comparing the *custom* results with *sklearn* SVM implementations, i.e., *liblinear* [9] and *libsvm* [10] implementations, and *cvxopt* [11] 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:

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

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 missclassified 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 convex optimization problem 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.

3.1 Hinge loss

The *hinge* loss is defined as:

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

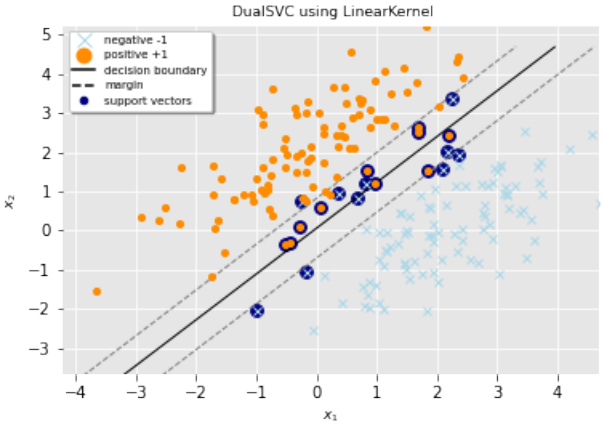


Figure 1: Linear SVC hyperplane

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:

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

where $\mathcal{R}(w, b)$ 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).

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.

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 (10) and (41) as follows:

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

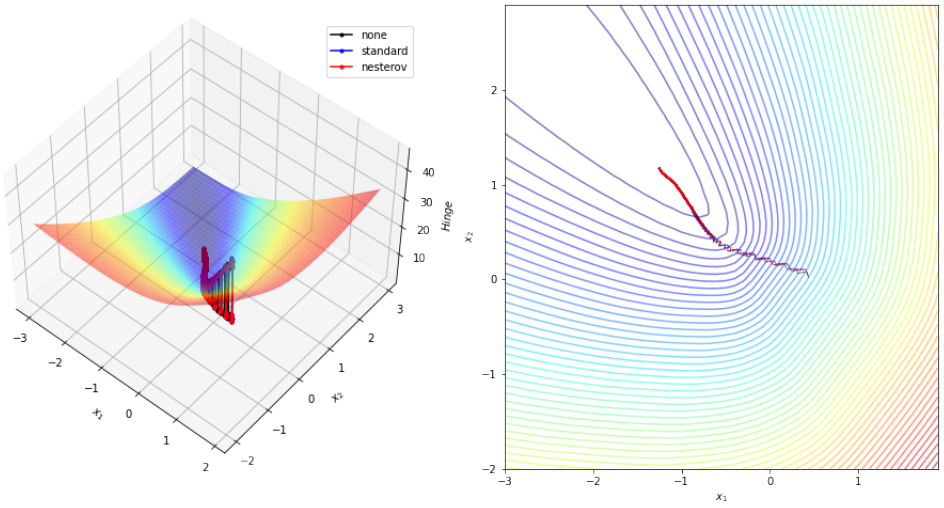


Figure 2: SVC Hinge loss with different optimization steps

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:

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

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.

Notice that in terms of numerical optimization the formulations (10) and (41) are not equivalent to (11) or (12) 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 [12] 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 $\alpha_i \geq 0, \mu_i \geq 0\ \forall_i$:

$$\max_{\alpha,\mu}\min_{w,b,\xi}\mathcal{W}(w,b,\xi,\alpha,\mu) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^{n}\mu_i\xi_i \tag{13}$$

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:

$$\frac{\partial\mathcal{W}}{\partial w} = w - \sum_{i=1}^{n}\alpha_i y_i x_i \Rightarrow w = \sum_{i=1}^{n}\alpha_i y_i x_i \tag{14}$$

$$\frac{\partial \mathcal{W}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{15}$$

$$\frac{\partial \mathcal{W}}{\partial \xi_i} = 0 \Rightarrow C = \alpha_i + \mu_i \tag{16}$$

Substituting (14) and (15) into (13) 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:

$$\begin{aligned} \max_{\alpha} \mathcal{W}(\alpha) &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \\ &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i Q_{ij} \alpha_j \text{ where } Q_{ij} = y_i y_j \langle x_i, x_j \rangle \\ &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \alpha^T Q \alpha \text{ subject to } 0 \leq \alpha_i \leq C \ \forall_i, \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned} \tag{17}$$

or, equivalently:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2} \alpha^T Q \alpha + q^T \alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \\ & y^T \alpha = 0 \end{aligned} \tag{18}$$

where $q^T = [1, \ldots, 1]$.

By solving (18) we will know α and, from (14), we will get w, so we need to calculate b.

We know that any data point satisfying (15) which is a support vector x_s will have the form:

$$y_s(w^T x_s + b) = 1 \tag{19}$$

and, by substituting in (14), we get:

$$y_s\big(\sum_{m \in S} \alpha_m y_m \langle x_m, x_s \rangle + b\big) = 1 \tag{20}$$

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):

$$y_s^2\big(\sum_{m \in S} \alpha_m y_m \langle x_m, x_s \rangle + b\big) = y_s \tag{21}$$

$$b = y_s - \sum_{m \in S} \alpha_m y_m \langle x_m, x_s \rangle \tag{22}$$

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

$$b = \frac{1}{N_s} \sum_{s \in S} y_s - \sum_{m \in S} \alpha_m y_m \langle x_m, x_s \rangle \tag{23}$$

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:

$$y' = \text{sgn}\big(\sum_{i=1}^{n} \alpha_i y_i \langle x_i, x' \rangle + b\big) \tag{24}$$

From (18) 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 [12] that arises from the primal (11) or (12) where the bias term b is embedded into the weight vector w:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T(Q + yy^T)\alpha + q^T\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \end{aligned} \tag{25}$$

3.1.3 Lagrangian Dual formulation

In order to relax the constraints in the *Wolfe dual* formulation (18) we define the problem as a *Lagrangian dual* relaxation by embedding them into objective function, so we need to allocate the Lagrangian multipliers $\mu \geq 0, \lambda_+ \geq 0$, $\lambda_- \geq 0$:

$$\begin{aligned} \max_{\mu,\lambda_+,\lambda_-} \min_{\alpha} \mathcal{L}(\alpha, \mu, \lambda_+, \lambda_-) &= \frac{1}{2}\alpha^T Q\alpha + q^T\alpha - \mu^T(y^T\alpha) - \lambda_+^T(u - \alpha) - \lambda_-^T\alpha \\ &= \frac{1}{2}\alpha^T Q\alpha + (q - \mu y + \lambda_+ - \lambda_-)^T\alpha - \lambda_+^T u \end{aligned} \tag{26}$$

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

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow Q\alpha + (q - \mu y + \lambda_+ - \lambda_-) = 0 \tag{27}$$

With α optimal solution of the linear system:

$$Q\alpha = -(q - \mu y + \lambda_+ - \lambda_-) \tag{28}$$

the gradient wrt μ, λ_+ and λ_- are:

$$\frac{\partial \mathcal{L}}{\partial \mu} = -y\alpha \tag{29}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_+} = \alpha - u \tag{30}$$

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

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:

$$\begin{aligned} \min_{\alpha_n \in K_n(Q,b)} \quad & \|Q\alpha_n - b\| \\ \text{where} \quad & b = -(q - \mu y + \lambda_+ - \lambda_-) \end{aligned} \tag{32}$$

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 $K_n(Q, b) = span(b, Qb, Q^2b, \ldots, Q^{n-1}b)$.

From (18) 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 (26) 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 (25).

$$\begin{aligned}\max_{\lambda_+,\lambda_-}\min_{\alpha}\mathcal{L}(\alpha,\lambda_+,\lambda_-) &= \frac{1}{2}\alpha^T(Q+yy^T)\alpha + q^T\alpha - \lambda_+^T(u-\alpha) - \lambda_-^T\alpha \\ &= \frac{1}{2}\alpha^T(Q+yy^T)\alpha + (q+\lambda_+-\lambda_-)^T\alpha - \lambda_+^T u\end{aligned} \tag{33}$$

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:

$$\frac{\partial\mathcal{L}}{\partial\alpha} = 0 \Rightarrow (Q+yy^T)\alpha + (q+\lambda_+-\lambda_-) = 0 \tag{34}$$

With α optimal solution of the linear system:

$$(Q+yy^T)\alpha = -(q+\lambda_+-\lambda_-) \tag{35}$$

the gradient wrt λ_+ and λ_- are:

$$\frac{\partial\mathcal{L}}{\partial\lambda_+} = \alpha - u \tag{36}$$

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

3.2 Squared Hinge loss

The *squared hinge* loss is defined as:

$$\mathcal{L}_2 = \begin{cases}0 & \text{if } y(w^Tx+b)\geq 1\\ (1-y(w^Tx+b))^2 & \text{otherwise}\end{cases} \tag{38}$$

or, equivalently:

$$\mathcal{L}_2 = \max(0, 1-y(w^Tx+b))^2 \tag{39}$$

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

$$\frac{\partial\mathcal{L}_2}{\partial w} = \begin{cases}-2yx & \text{if } y(w^Tx+b)<1\\ 0 & \text{otherwise}\end{cases} \tag{40}$$

3.2.1 Primal formulation

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

$$\min_{w,b}\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\max(0,1-y_i(w^Tx_i+b))^2 \tag{41}$$

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

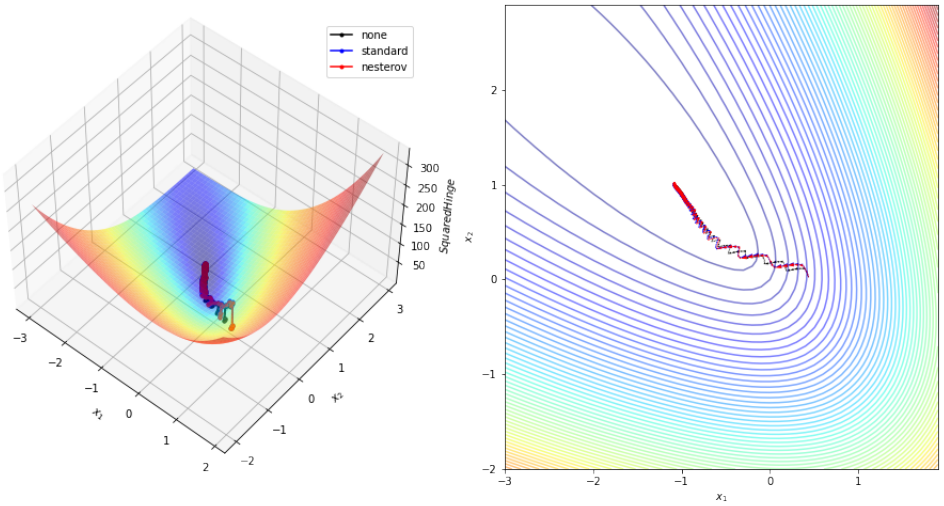


Figure 3: SVC Squared Hinge loss with different optimization steps

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:

$$\{(x_i, y_i), x \in \Re^m, y_i \in \Re, i = 1, \ldots, n\} \tag{42}$$

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 $\xi^- \geq 0 \ \forall_i$:

$$\begin{aligned} y_i &\leq y_i' + \epsilon + \xi^+ \ \forall_i \\ y_i &\geq y_i' - \epsilon - \xi^- \ \forall_i \\ \xi_i^+, \xi_i^- &\geq 0 \ \forall_i \end{aligned} \tag{43}$$

The objective function for SVR can then be written as:

$$\begin{aligned} \min_{w,b,\xi^+,\xi^-} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i^+ + \xi_i^-) \\ \text{subject to} \quad & y_i - w^T x_i - b \leq \epsilon + \xi_i^+ \ \forall_i \\ & w^T x_i + b - y_i \leq \epsilon + \xi_i^- \ \forall_i \\ & \xi_i^+, \xi_i^- \geq 0 \ \forall_i \end{aligned} \tag{44}$$

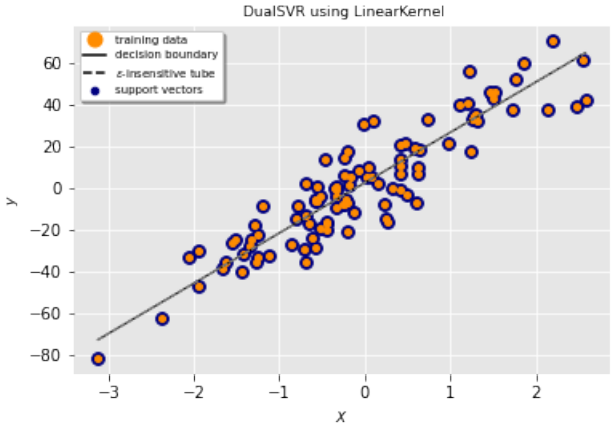


Figure 4: Linear SVR hyperplane

4.1 Epsilon-insensitive loss

The *epsilon-insensitive* loss is defined as:

$$\mathcal{L}_\epsilon = \begin{cases} 0 & \text{if } |y - (w^T x + b)| \leq \epsilon \\ |y - (w^T x + b)| - \epsilon & \text{otherwise} \end{cases} \tag{45}$$

or, equivalently:

$$\mathcal{L}_\epsilon = \max(0, |y - (w^T x + b)| - \epsilon) \tag{46}$$

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:

$$\frac{\partial \mathcal{L}_\epsilon}{\partial w} = \begin{cases} (y - (w^T x + b))x & \text{if } |y - (w^T x + b)| > \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{47}$$

4.1.1 Primal formulation

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

The quadratic optimization problem (44) can be equivalently formulated as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \max(0, |y_i - (w^T x_i + b)| - \epsilon) \tag{48}$$

where we make use of the *epsilon-insensitive* loss (45) or (46).

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

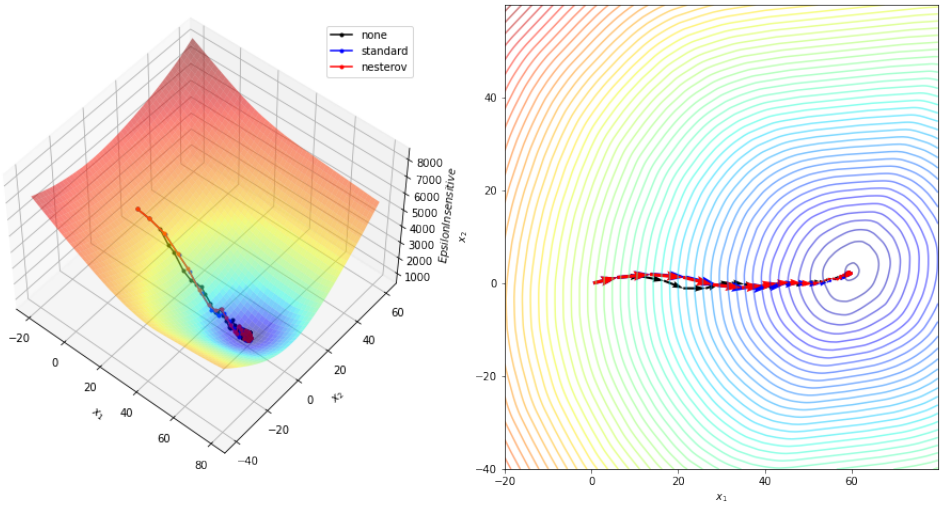


Figure 5: SVR Epsilon-insensitive loss with different optimization steps

4.1.2 Wolfe Dual formulation

To reformulate the (44) as a *Wolfe dual*, we introduce the Lagrange multipliers $\alpha_i^+ \geq 0, \alpha_i^- \geq 0, \mu_i^+ \geq 0, \mu_i^- \geq 0 \; \forall_i$:

$$\begin{aligned}\max_{\alpha^+,\alpha^-,\mu^+,\mu^-} \min_{w,b,\xi^+,\xi^-} \mathcal{W}(w,b,\xi^+,\xi^-,\alpha^+,\alpha^-,\mu^+,\mu^-) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i^+ + \xi_i^-) - \sum_{i=1}^{n}(\mu_i^+\xi_i^+ + \mu_i^-\xi_i^-) \\ - \sum_{i=1}^{n}\alpha_i^+(\epsilon + \xi_i^+ + y_i' - y_i) - \sum_{i=1}^{n}\alpha_i^-(\epsilon + \xi_i^- - y_i' + y_i)\end{aligned} \tag{49}$$

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

$$\frac{\partial \mathcal{W}}{\partial w} = w - \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-)x_i \Rightarrow w = \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-)x_i \tag{50}$$

$$\frac{\partial \mathcal{W}}{\partial b} = -\sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-) \Rightarrow \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-) = 0 \tag{51}$$

$$\frac{\partial \mathcal{W}}{\partial \xi_i^+} = 0 \Rightarrow C = \alpha_i^+ + \mu_i^+ \tag{52}$$

$$\frac{\partial \mathcal{W}}{\partial \xi_i^-} = 0 \Rightarrow C = \alpha_i^- + \mu_i^- \tag{53}$$

Substituting (50) and (51) in, we now need to maximize $\mathcal{W}$ wrt α_i^+ and α_i^-, where $\alpha_i^+ \geq 0,\ \alpha_i^- \geq 0 \; \forall_i$:

$$\max_{\alpha^+,\alpha^-} \mathcal{W}(\alpha^+,\alpha^-) = \sum_{i=1}^{n} y_i(\alpha_i^+ - \alpha_i^-) - \epsilon\sum_{i=1}^{n}(\alpha_i^+ + \alpha_i^-) - \frac{1}{2}\sum_{i,j}(\alpha_i^+ - \alpha_i^-)\langle x_i, x_j\rangle(\alpha_j^+ - \alpha_j^-) \tag{54}$$

Using $\mu_i^+ \geq 0$ and $\mu_i^- \geq 0$ together with (50) and (51) means that $\alpha_i^+ \leq C$ and $\alpha_i^- \leq C$. We therefore need to find:

$$\begin{aligned} \min_{\alpha^+,\alpha^-} \quad & \frac{1}{2}(\alpha^+ - \alpha^-)^T K(\alpha^+ - \alpha^-) + \epsilon q^T(\alpha^+ + \alpha^-) - y^T(\alpha^+ - \alpha^-) \\ \text{subject to} \quad & 0 \leq \alpha_i^+, \alpha_i^- \leq C \ \forall_i \\ & q^T(\alpha^+ - \alpha^-) = 0 \end{aligned} \tag{55}$$

where $q^T = [1, \ldots, 1]$.
We can write the (55) in a standard quadratic form as:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T Q\alpha - q^T\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \\ & e^T\alpha = 0 \end{aligned} \tag{56}$$

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:

$$y' = \sum_{i=1}^{n}(\alpha_i^+ - \alpha_i^-)\langle x_i, x' \rangle + b \tag{57}$$

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:

$$b = y_s - \epsilon - \sum_{m \in S}(\alpha_m^+ - \alpha_m^-)\langle x_m, x_s \rangle \tag{58}$$

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

$$b = \frac{1}{N_s}\sum_{s \in S} y_s - \epsilon - \sum_{m \in S}(\alpha_m^+ - \alpha_m^-)\langle x_m, x_s \rangle \tag{59}$$

From (56) 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 [12] that arises from the primal (11) or (12) where the bias term b is embedded into the weight vector w:

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T(Q + ee^T)\alpha + q^T\alpha \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C \ \forall_i \end{aligned} \tag{60}$$

4.1.3 Lagrangian Dual formulation

In order to relax the constraints in the *Wolfe dual* formulation (55) we define the problem as a *Lagrangian dual* relaxation by embedding them into objective function, so we need to allocate the Lagrangian multipliers $\mu \geq 0, \lambda_+ \geq 0, \lambda_- \geq 0$:

$$\begin{aligned} \max_{\mu,\lambda_+,\lambda_-} \min_{\alpha} \mathcal{L}(\alpha, \mu, \lambda_+, \lambda_-) &= \frac{1}{2}\alpha^T Q\alpha + q^T\alpha - \mu^T(e^T\alpha) - \lambda_+^T(u - \alpha) - \lambda_-^T\alpha \\ &= \frac{1}{2}\alpha^T Q\alpha + (q - \mu e + \lambda_+ - \lambda_-)^T\alpha - \lambda_+^T u \end{aligned} \tag{61}$$

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

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

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow Q\alpha + (q - \mu e + \lambda_+ - \lambda_-) = 0 \tag{62}$$

With α optimal solution of the linear system:

$$Q\alpha = -(q - \mu e + \lambda_+ - \lambda_-) \tag{63}$$

the gradient wrt μ, λ_+ and λ_- are:

$$\frac{\partial \mathcal{L}}{\partial \mu} = -e\alpha \tag{64}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_+} = \alpha - u \tag{65}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_-} = -\alpha \tag{66}$$

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:

$$\begin{aligned} \min_{\alpha_n \in K_n(Q,b)} \quad & \|Q\alpha_n - b\| \\ \text{where} \quad & b = -(q - \mu e + \lambda_+ - \lambda_-) \end{aligned} \tag{67}$$

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 $K_n(Q, b) = span(b, Qb, Q^2b, \ldots, Q^{n-1}b)$.

From (56) 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 (61) 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 (60).

$$\begin{aligned} \max_{\lambda_+,\lambda_-} \min_{\alpha} \mathcal{L}(\alpha, \lambda_+, \lambda_-) &= \frac{1}{2}\alpha^T(Q + ee^T)\alpha + q^T\alpha - \lambda_+^T(u - \alpha) - \lambda_-^T\alpha \\ &= \frac{1}{2}\alpha^T(Q + ee^T)\alpha + (q + \lambda_+ - \lambda_-)^T\alpha - \lambda_+^T u \end{aligned} \tag{68}$$

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:

$$\frac{\partial \mathcal{L}}{\partial \alpha} = 0 \Rightarrow (Q + ee^T)\alpha + (q + \lambda_+ - \lambda_-) = 0 \tag{69}$$

With α optimal solution of the linear system:

$$(Q + ee^T)\alpha = -(q + \lambda_+ - \lambda_-) \tag{70}$$

the gradient wrt λ_+ and λ_- are:

$$\frac{\partial \mathcal{L}}{\partial \lambda_+} = \alpha - u \tag{71}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_-} = -\alpha \tag{72}$$

4.2 Squared Epsilon-insensitive loss

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

$$\mathcal{L}_\epsilon^2 = \begin{cases} 0 & \text{if } |y - (w^T x + b)| \leq \epsilon \\ (|y - (w^T x + b)| - \epsilon)^2 & \text{otherwise} \end{cases} \tag{73}$$

or, equivalently:

$$\mathcal{L}_\epsilon^2 = \max(0, |y - (w^T x + b)| - \epsilon)^2 \tag{74}$$

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:

$$\frac{\partial \mathcal{L}_\epsilon^2}{\partial w} = \begin{cases} 2((y - (w^T x + b))x) & \text{if } |y - (w^T x + b)| > \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{75}$$

4.2.1 Primal formulation

To provide a continuously differentiable function the optimization problem (48) can be formulated as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \max(0, |y_i - (w^T x_i + b)| - \epsilon)^2 \tag{76}$$

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

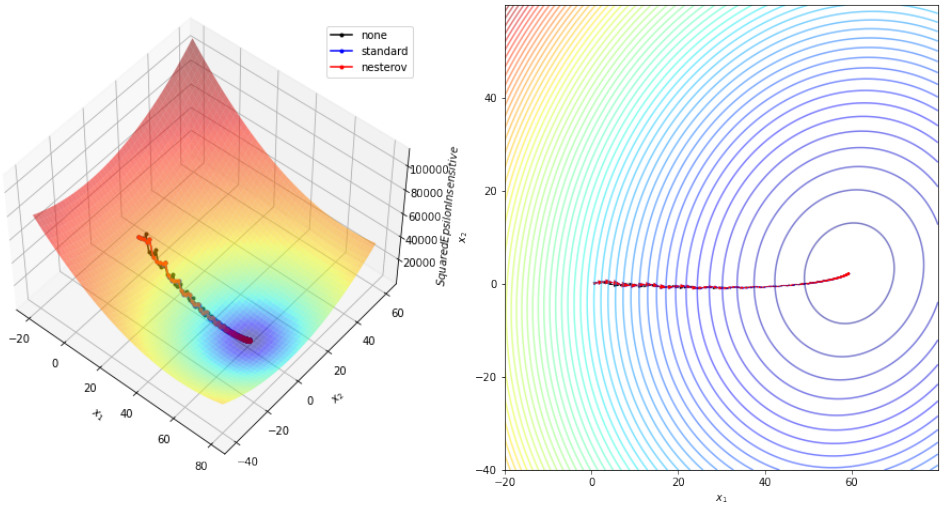


Figure 6: SVC Squared Epsilon-insensitive loss with different optimization steps

5 Nonlinear Support Vector Machines

When applying our SVC to *linearly separable* data in (17), we have started by creating a matrix Q from the dot product of our input variables:

$$Q_{ij} = y_i y_j k(x_i, x_j) \tag{77}$$

or, a matrix K from the dot product of our input variables in the SVR case (55):

$$K_{ij} = k(x_i, x_j) \tag{78}$$

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

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle = \phi(x_i)^T \phi(x_j) \tag{79}$$

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

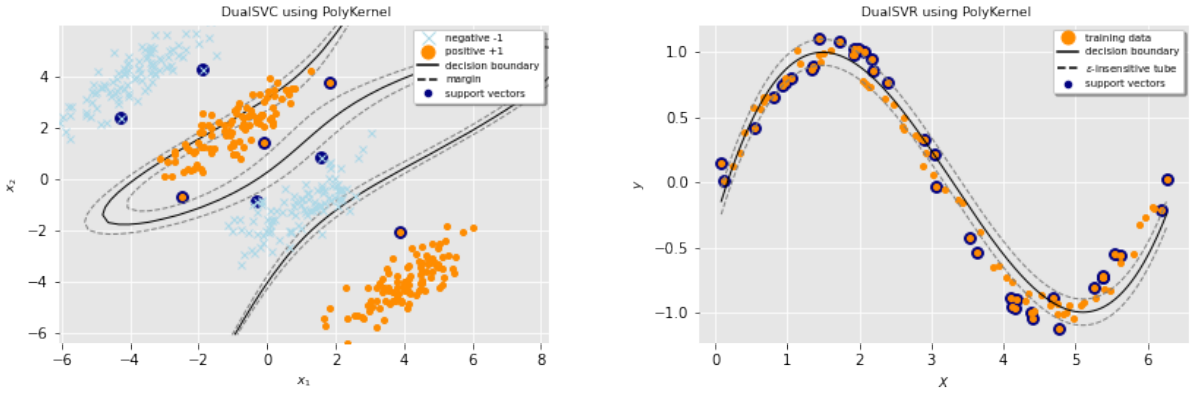
The reason that this *kernel trick* is useful is that there are many classification/regression problems that are nonlinearly separable/regressable in the *input space*, which might be in a higher dimensionality *feature space* given a suitable mapping $x \rightarrow \phi(x)$.

5.1 Polynomial kernel

The *polynomial* kernel is defined as:

$$k(x_i, x_j) = (\gamma \langle x_i, x_j \rangle + r)^d \tag{80}$$

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

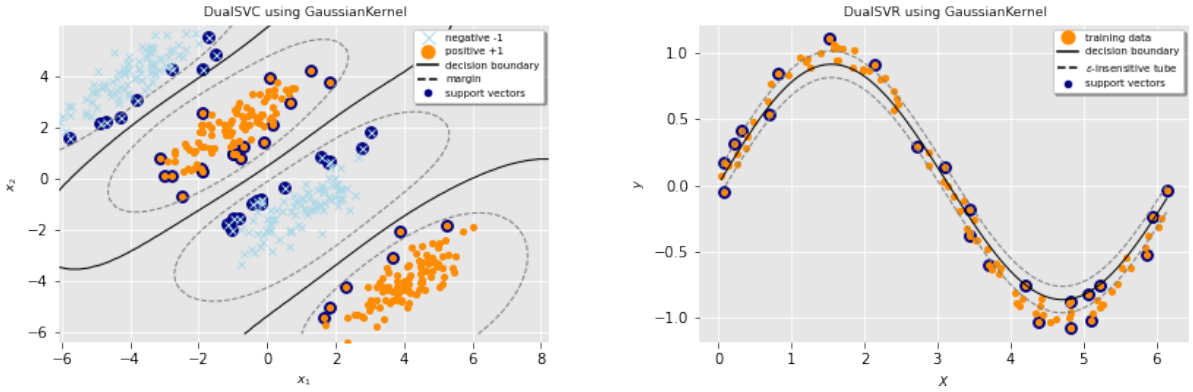
The *gaussian* kernel is defined as:

$$k(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2}) \tag{81}$$

or, equivalently:

$$k(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2) \tag{82}$$

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-smoothness* of a function [13].

First of all, we give three different but equivalent definitions of convexity in terms of the function itself, the Jacobian and the Hessian.

Definition 1 (Convexity)**.** We say that a function $f : \Re^m \to \Re$ is convex if:

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

Definition 2 (Convexity - Jacobian)**.** We say that a differentiable function $f : \Re^m \to \Re$ is convex iff:

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

Definition 3 (Convexity - Hessian)**.** We say that a twice differentiable function, i.e., the Hessian matrix is *symmetric*, $f : \Re^m \to \Re$ is convex iff:

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

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

The definitions of *strong convexity* and *L-smoothness* below will be useful.

Definition 4 (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 some $\mu > 0$. The latter, in terms of the Jacobian, is equivalent to:

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

and, in terms of the Hessian, is equivalent to:

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

which is:

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

Definition 5 (L-smoothness)**.** We say that a function $f : \Re^m \to \Re$ is L-smooth, i.e., L-Lipschitz continuous, if it is differentiable and if:

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

6.1 Gradient Descent

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, α)
    Initialize weight vector x_0
    $t = 0$
    while *not_convergence* **do**
      $x_{t+1} = x_t - \alpha \nabla f(x_t)$
      $t = t + 1$
    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:

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

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, instead of $\mathcal{O}(nm)$.

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, α, k)
    Initialize weight vector x_0
    $t \leftarrow 0$
    while *not_convergence* **do**
      Sample $(i_1, \ldots, i_k) \sim \mathcal{U}^k(1, \ldots, n)$
      $x_{t+1} \leftarrow x_t - \alpha \frac{1}{k} \sum_{j=1}^{k} \nabla f_{i_j}(x_t)$
      $t \leftarrow t + 1$
    end while
    return x_t
  end function

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

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

6.1.1 Momentum

To mitigate the pathological zig-zagging of the SGD method we introduce two acellerated methods [1] and [2] 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 Accelerated Gradient Descent or or 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, α, β)
    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

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 Accelerated Gradient Descent

Require: Function f to minimize
Require: Learning rate $\alpha > 0$
Require: Momentum $\beta \in [0, 1)$
  function NesterovAcceleratedGradientDescent(f, α, β)
    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.

Nesterov momentum brings the rate of convergence from $\mathcal{O}\Big(\frac{1}{t}\Big)$ to $\mathcal{O}\Big(\frac{1}{t^2}\Big)$ and in the case of smooth and strongly convex functions gives the acceleration that we had with Polyak's momentum for quadratic functions. This is great, because we get the guarantee for a more general class of functions.

We can write the iteration complexity of these methods, i.e., the smallest t such that we're within ϵ, for a L-smooth and μ-strongly convex function as $\mathcal{O}\Big(\kappa \log \frac{1}{\epsilon}\Big)$ for the standard GD method, $\mathcal{O}\Big(\sqrt{\kappa} \log \frac{1}{\epsilon}\Big)$ for the Polyak's method and, finally, $\mathcal{O}\Big(\frac{1}{\sqrt{\epsilon}}\Big)$ for the NAG method to get ϵ-close to global optimum and where κ, i.e., the *conditioning number*, is defined as $\kappa = L/\mu$ and where L and μ are also equal to the smallest and the largest eigenvalues λ_{min} and λ_{max} respectively.

6.2 AdaGrad

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* [5] 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 5 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, α, ϵ)
    Initialize weight vector x_0 and the squared accumulated gradients vector $s_t \leftarrow 0$
    $t = 1$
    while *not_convergence* **do**
      $g_t \leftarrow \nabla f(x_t)$
      $s_t \leftarrow s_{t-1} + g_t^2$
      $x_{t+1} \leftarrow x_t - \alpha P^{-1} g_t = x_t - \dfrac{\alpha}{\sqrt{s_t + \epsilon}} \odot g_t$ where $P \leftarrow diag(s_t + \epsilon)^{1/2}$
      $t \leftarrow t + 1$
    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-smooth convex functions is $\mathcal{O}\Big(\dfrac{1}{\sqrt{t}}\Big)$.

6.3 Sequential Minimal Optimization

The *Sequential Minimal Optimization (SMO)* [3] 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.

6.3.1 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.

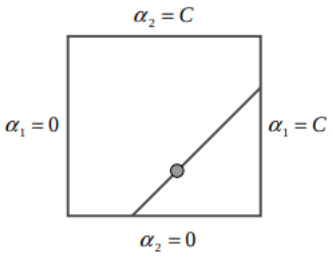


Figure 10: SMO for two Lagrange multipliers

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 $y_1 \neq y_2$:

$$\begin{aligned} L &= max(0, \alpha_2 - \alpha_1) \\ H &= min(C, C + \alpha_2 - \alpha_1) \end{aligned} \tag{83}$$

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

$$\begin{aligned} L &= max(0, \alpha_2 + \alpha_1 - C) \\ H &= min(C, \alpha_2 + \alpha_1) \end{aligned} \tag{84}$$

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

$$\eta = K(x_1, x_1) + K(x_2, x_2) - 2K(x_1, x_2) \tag{85}$$

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:

$$\alpha_2^{new} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta} \tag{86}$$

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:

$$\alpha_2^{new,clipped} = \begin{cases} H & \text{if } \alpha_2^{new} \geq H \\ \alpha_2^{new} & \text{if } L < \alpha_2^{new} < H \\ L & \text{if } \alpha_2^{new} \leq L \end{cases} \tag{87}$$

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

$$\alpha_1^{new} = \alpha_1 + s(\alpha_2 - \alpha_2^{new,clipped}) \tag{88}$$

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 (18) are:

$$\begin{aligned} \alpha_i = 0 &\Leftrightarrow y_i y_i' \geq 1 \\ 0 < \alpha_i < C &\Leftrightarrow y_i y_i' = 1 \\ \alpha_i = C &\Leftrightarrow y_i y_i' \leq 1 \end{aligned} \tag{89}$$

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:

$$b_{up} = E_1 + y_1(\alpha_1^{new} - \alpha_1)K(x_1, x_1) + y_2(\alpha_2^{new,clipped} - \alpha_2)K(x_1, x_2) + b \tag{90}$$

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

$$b_{low} = E_2 + y_1(\alpha_1^{new} - \alpha_1)K(x_1, x_2) + y_2(\alpha_2^{new,clipped} - \alpha_2)K(x_2, x_2) + b \tag{91}$$

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:

$$b = \begin{cases} b_{up} & \text{if } 0 < \alpha_1 < C \\ b_{low} & \text{if } 0 < \alpha_2 < C \\ \dfrac{b_{up} + b_{low}}{2} & \text{otherwise} \end{cases} \tag{92}$$

Algorithm 6 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 training examples
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 empty set $I4 \leftarrow \{i : y_i = -1, \alpha_i = 0\}$
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 ← num_changed + ExamineExample(i)
        end for
    else
        for i in I0 do                    ▷ loop over examples where α are not already at their bounds
            num_changed ← num_changed + ExamineExample(i)
            if b_up > b_low − 2tol then                    ▷ check if optimality on I0 is attained
                num_changed ← 0
                break
            end if
        end for
    end if
    if examine_all = True then
        examine_all ← False
    else if num_changed = 0 then
        examine_all ← True
    end if
  end while
  Compute b by (92)
  return α, b
 end function
Require: Training example i
 function ExamineExample(i)
  TODO
 end function
Require: First Lagrange multiplier i1
Require: Second Lagrange multiplier i2
 function TakeStep(i1, i2)
  if i1 = i2 then
   return False
  end if
  Compute L and H using (83) or (84)
  Compute α_2^new by (86)
  Compute α_2^new,clipped by (87)
  Compute α_1^new by (88)
  if L = H then
   return False
  end if
  Compute η by (85)          ▷ we assume that η > 0, i.e., the kernel matrix K is positive definite
  if changes in α_2^new,clipped and α_1^new are larger than some eps then
   update α_2^new,clipped and α_1^new
  end if
  if changes in α_2^new,clipped and α_1^new are larger than some eps then
   for i in I0 do
    Update errors_i using new Lagrange multipliers
    Update α using the new Lagrange multipliers
    Update I0, I1, I2, I3 and I4
    Update errors_i1 and errors_i2
    for i in I0 ∪ {i1, i2} do
     Compute (i_low, b_low) by b_low = max{errors_i : i ∈ I0 ∪ I3 ∪ I4}
     Compute (i_up, b_up) by b_up = min{errors_i : i ∈ I0 ∪ I1 ∪ I2}
    end for
   end for
```

```
        return True
    else
        return False
    end if
end function
```

6.3.2 Regression

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

$$\begin{aligned} L &= max(0, \gamma - C) \\ H &= min(C, \gamma) \end{aligned} \tag{93}$$

$$\alpha_2^{+,new} = \alpha_2^+ - \frac{E_1 - E_2}{\eta} \tag{94}$$

$$\alpha_1^{+,new} = \alpha_1^+ - (\alpha_2^{+,new,clipped} - \alpha_2^+) \tag{95}$$

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 $E_1 - E_2 > 2\epsilon$)):

$$\begin{aligned} L &= max(0, -\gamma) \\ H &= min(C, -\gamma + C) \end{aligned} \tag{96}$$

$$\alpha_2^{-,new} = \alpha_2^- + \frac{(E_1 - E_2) - 2\epsilon}{\eta} \tag{97}$$

$$\alpha_1^{+,new} = \alpha_1^+ + (\alpha_2^{-,new,clipped} - \alpha_2^-) \tag{98}$$

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 $E_1 - E_2 < -2\epsilon$)):

$$\begin{aligned} L &= max(0, \gamma) \\ H &= min(C, C + \gamma) \end{aligned} \tag{99}$$

$$\alpha_2^{+,new} = \alpha_2^+ - \frac{(E_1 - E_2) + 2\epsilon}{\eta} \tag{100}$$

$$\alpha_1^{-,new} = \alpha_1^- + (\alpha_2^{+,new,clipped} - \alpha_2^+) \tag{101}$$

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 $E_1 - E_2 > 0$)):

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

$$\alpha_2^{-,new} = \alpha_2^- + \frac{E_1 - E_2}{\eta} \tag{103}$$

$$\alpha_1^{-,new} = \alpha_1^- - (\alpha_2^{-,new,clipped} - \alpha_2^-) \tag{104}$$

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 (85) and (87) respectively.

The KKT conditions for the regression problem (55) are:

$$\begin{aligned} \alpha_i^+ - \alpha_i^- = 0 &\Leftrightarrow |y_i - y_i'| < \epsilon \\ -C < \alpha_i^+ - \alpha_i^- < C &\Leftrightarrow |y_i - y_i'| = \epsilon \\ \alpha_i^+ + \alpha_i^- = C &\Leftrightarrow |y_i - y_i'| > \epsilon \end{aligned} \tag{105}$$

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:

$$b_{up} = E_1 + ((\alpha_1^+ - \alpha_1^-) - (\alpha_1^{+,new} - \alpha_1^{-,new}))K(x_1, x_1) + ((\alpha_2^+ - \alpha_2^-) - (\alpha_2^{+,new,clipped} - \alpha_2^{-,new,clipped}))K(x_1, x_2) + b \tag{106}$$

$$b_{low} = E_2 + ((\alpha_1^+ - \alpha_1^-) - (\alpha_1^{+,new} - \alpha_1^{-,new}))K(x_1, x_2) + ((\alpha_2^+ - \alpha_2^-) - (\alpha_2^{+,new,clipped} - \alpha_2^{-,new,clipped}))K(x_2, x_2) + b \tag{107}$$

$$b = \begin{cases} b_{up} & \text{if } 0 < \alpha_1^+, \alpha_1^- < C \\ b_{low} & \text{if } 0 < \alpha_2^+, \alpha_2^- < C \\ \dfrac{b_{up} + b_{low}}{2} & \text{otherwise} \end{cases} \tag{108}$$

The improvements described in [4, 7] 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 7 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 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 α^+ and α^- 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 ← False
    else if num_changed = 0 then
        examine_all ← True
    end if
  end while
  Compute b by (108)
  return α⁺, α⁻, b
end function
Require: Training example i2
function ExamineExample(i2)
  if i2 ∈ I0 then
    E_2 ← errors_{i2}
  else
    E_2 ← y_{i2} − (α⁺ − α⁻) ⊙ K_{i2}
    errors_{i2} ← 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: First Lagrange multiplier i1
Require: Second Lagrange multiplier i2
function TakeStep(i1, i2)
  if i1 = i2 then
    return False
  end if
  Compute η by (85)                ▷ we assume that η > 0, i.e., the kernel matrix K is positive definite
  Compute L and H using (93), (96), (99) or (102)
  if L < H then
    Compute α_2^{+,new} and α_2^{−,new} using (94), (100) or (97), (94) respectively
    Compute α_2^{+,new,clipped} and α_2^{−,new,clipped} by (87)
    Compute α_1^{+,new} and α_1^{−,new} using (95), (98) or (101), (104) respectively
    if changes in α_2^{+,new,clipped}, α_2^{−,new,clipped}, α_1^{+,new} and α_1^{−,new} are larger than some eps then
      update α_2^{+,new,clipped}, α_2^{−,new,clipped}, α_1^{+,new} and α_1^{−,new}
    end if
  end if
  if changes in α_2^{+,new,clipped}, α_2^{−,new,clipped}, α_1^{+,new} and α_1^{−,new} are larger than some eps then
    for i in I0 do
      Update errors_i using new Lagrange multipliers
      Update α⁺ and α⁻ using the new Lagrange multipliers
```

```
            Update I0, I1, I2 and I3
            Update errors_{i1} and errors_{i2}
            for i in I0 ∪ {i1, i2} do
                Compute (i_low, b_low) and (i_up, b_up)
            end for
        end for
        return True
    else
        return False
    end if
end function
```

7 Experiments

The following experiments refer to 3-fold cross-validation over *linearly* and *nonlinearly* separable generated datasets of size 100, so the reported results are to considered as a mean over the 3 folds.

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 1 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. *Liblinear* uses the *Coordinate Gradient Descent* algorithm which minimize one coordinate at a time.

Table 1: SVC Primal formulation results with Hinge loss

solver	C	momentum	fit_time	n_iter	train_accuracy	val_accuracy	train_n_sv	val_n_sv
sgd	1	none	0.710145	2743	0.947462	0.939846	40	20
		standard	0.520384	2101	0.947462	0.939846	37	19
		nesterov	0.516617	2099	0.947462	0.939846	37	19
	10	none	0.518066	1770	0.954962	0.944821	19	11
		standard	0.357653	1324	0.954962	0.944821	17	10
		nesterov	0.362264	1256	0.954962	0.944821	17	10
	100	none	0.317060	813	0.954962	0.944821	14	7
		standard	0.184168	528	0.954962	0.944821	13	8
		nesterov	0.144800	449	0.954962	0.944821	14	7
liblinear	1	-	0.001606	386	0.952456	0.939846	17	9
	10	-	0.001603	980	0.962462	0.949872	12	6
	100	-	0.001761	1000	0.964968	0.954998	10	5

Linear Dual formulations *Libsvm* also uses the *SMO* algorithm to solve the dual. The experiments results shown in 3 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.5 for the *AdaGrad* algorithm.

Table 2: Linear SVC Wolfe Dual formulation results with Hinge loss

solver	C	fit_time	n_iter	train_accuracy	val_accuracy	train_n_sv	val_n_sv
smo	1	0.054521	41	0.967493	0.969998	15	15
	10	0.178687	216	0.967493	0.964948	12	12
	100	0.463555	1039	0.967493	0.964948	11	11
libsvm	1	0.001915	58	0.969981	0.964948	15	15
	10	0.003209	728	0.969981	0.964948	12	12
	100	0.004323	4114	0.967493	0.964948	11	11
cvxopt	1	0.032567	10	0.967493	0.969998	15	15
	10	0.020921	10	0.967493	0.969998	14	14
	100	0.043281	10	0.967493	0.964948	24	24

Table 3: Linear SVC Lagrangian Dual formulation results with Hinge loss

dual	C	fit_time	n_iter	train_accuracy	val_accuracy	train_n_sv	val_n_sv
qp	1	0.009340	1	0.977518	0.980024	130	130
	10	0.005536	1	0.977518	0.980024	130	130
	100	0.006986	1	0.977518	0.980024	130	130
bcqp	1	0.006575	1	0.964987	0.980024	129	129
	10	0.005960	1	0.964987	0.980024	129	129
	100	0.004786	1	0.964987	0.980024	129	129

Nonlinear Dual formulations The experiments results shown in 4 and 5 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. *Libsvm* also uses the *SMO* algorithm to solve the dual. The experiments results shown in 5 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.5 for the *AdaGrad* algorithm.

Table 4: Nonlinear SVC Wolfe Dual formulation results with Hinge loss

solver	kernel	C	fit_time	n_iter	train_accuracy	val_accuracy	train_n_sv	val_n_sv
smo	poly	1	0.300656	94	0.816287	0.708488	32	32
		10	0.304991	108	0.926131	0.743650	10	10
		100	0.265495	172	0.957426	0.828302	8	8
	rbf	1	0.253971	50	0.998747	1.000000	43	43
		10	0.275428	93	1.000000	1.000000	14	14
		100	0.220114	81	1.000000	1.000000	11	11
libsvm	poly	1	0.003757	270	0.998747	0.992481	32	32
		10	0.003405	319	1.000000	0.992481	10	10
		100	0.003105	274	1.000000	0.992481	8	8
	rbf	1	0.003656	99	1.000000	1.000000	44	44
		10	0.004566	149	1.000000	1.000000	14	14
		100	0.003053	205	1.000000	1.000000	11	11
cvxopt	poly	1	0.085239	10	0.815039	0.705981	32	32
		10	0.074714	10	0.926131	0.743650	10	10
		100	0.066410	10	0.957426	0.828302	9	9
	rbf	1	0.096586	10	0.998747	1.000000	44	44
		10	0.082683	10	1.000000	1.000000	15	15
		100	0.073029	10	1.000000	1.000000	14	14

Table 5: Nonlinear SVC Lagrangian Dual formulation results with Hinge loss

dual	kernel	C	fit_time	n_iter	train_accuracy	val_accuracy	train_n_sv	val_n_sv
qp	poly	1	0.635557	126	0.774971	0.521266	205	205
		10	0.579004	121	0.782462	0.533797	205	205
		100	0.612130	121	0.782462	0.533797	205	205
	rbf	1	1.728111	269	0.758741	0.501253	136	136
		10	0.553077	82	0.750002	0.501253	109	109
		100	1.160888	359	0.831151	0.578947	149	149
bcqp	poly	1	1.149153	347	0.784959	0.543785	206	206
		10	0.986511	347	0.784959	0.543785	206	206
		100	0.718925	347	0.784959	0.543785	206	206
	rbf	1	0.022847	1	0.998747	0.992519	249	249
		10	0.020183	1	0.998747	0.992519	249	249
		100	0.020089	1	0.998747	0.992519	249	249

7.1.2 Squared Hinge loss

Primal formulation The experiments results shown in 6 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. *Liblinear* uses the *Coordinate Gradient Descent* algorithm which minimize one coordinate at a time.

Table 6: SVC Primal formulation results with Squared Hinge loss

solver	C	momentum	fit_time	n_iter	train_accuracy	val_accuracy	train_n_sv	val_n_sv
sgd	1	none	0.313461	1273	0.944956	0.939846	40	20
		standard	0.253786	930	0.947462	0.939846	37	19
		nesterov	0.245869	934	0.947462	0.939846	37	19
	10	none	0.117412	384	0.952456	0.944821	26	13
		standard	0.078306	239	0.952456	0.944821	25	13
		nesterov	0.075893	244	0.952456	0.944821	25	13
	100	none	0.038147	78	0.952456	0.944821	18	10
		standard	0.051053	83	0.959975	0.944821	15	8
		nesterov	0.045448	78	0.957468	0.944821	16	8
liblinear	1	-	0.001865	600	0.964968	0.954847	29	14
	10	-	0.002564	1000	0.962462	0.954847	28	14
	100	-	0.002419	1000	0.959956	0.934946	27	15

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 7 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. *Liblinear* uses the *Coordinate Gradient Descent* algorithm which minimize one coordinate at a time.

Table 7: SVR Primal formulation results with Epsilon-insensitive loss

solver	C	momentum	epsilon	fit_time	n_iter	train_r2	val_r2	train_n_sv	val_n_sv
sgd	1	none	0.1	0.105133	238	0.410998	0.402111	66	33
			0.2	0.151977	238	0.410998	0.402111	66	33
			0.3	0.121096	238	0.410998	0.402111	66	33
		standard	0.1	0.081215	152	0.427455	0.418695	66	33
			0.2	0.093309	152	0.427455	0.418695	66	33
			0.3	0.079694	153	0.430658	0.421952	66	33
		nesterov	0.1	0.074780	152	0.427207	0.418447	66	33
			0.2	0.061801	152	0.427207	0.418447	66	33
			0.3	0.084097	153	0.430409	0.421703	66	33
	10	none	0.1	0.118272	278	0.975885	0.971366	66	33
			0.2	0.112319	276	0.975752	0.971251	66	33
			0.3	0.110455	274	0.975685	0.971108	65	32
		standard	0.1	0.102871	174	0.976441	0.971938	66	33
			0.2	0.092689	174	0.976441	0.971939	66	32
			0.3	0.091080	173	0.976363	0.971751	65	32
		nesterov	0.1	0.073764	174	0.976374	0.971878	66	33
			0.2	0.071638	174	0.976374	0.971879	66	32
			0.3	0.067307	174	0.976347	0.971756	65	32
	100	none	0.1	0.058506	113	0.977986	0.973377	65	33
			0.2	0.043937	114	0.977986	0.973380	65	32
			0.3	0.039389	118	0.977986	0.973380	64	31
		standard	0.1	0.032581	72	0.977997	0.973440	66	33
			0.2	0.035756	72	0.977997	0.973440	65	32
			0.3	0.029877	73	0.977997	0.973441	64	31
		nesterov	0.1	0.030335	78	0.977995	0.973448	66	33
			0.2	0.028698	69	0.977994	0.973450	65	32
			0.3	0.024535	77	0.977995	0.973450	64	31
liblinear	1	-	0.1	0.001066	11	0.918768	0.916773	66	33
			0.2	0.001025	10	0.918763	0.916602	65	32
			0.3	0.001072	13	0.919296	0.917061	65	32
	10	-	0.1	0.001038	162	0.977849	0.972087	65	33
			0.2	0.001572	178	0.977852	0.972041	65	33
			0.3	0.001319	113	0.977871	0.972151	64	33
	100	-	0.1	0.001538	638	0.977725	0.974270	65	33
			0.2	0.001481	686	0.977664	0.974138	66	33
			0.3	0.001406	891	0.977653	0.974016	65	33

Linear Dual formulations *Libsvm* also uses the *SMO* algorithm to solve the dual. The experiments results shown in 9 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.5 for the *AdaGrad* algorithm.

Table 8: Linear SVR Wolfe Dual formulation results with Epsilon-insensitive loss

solver	C	epsilon	fit_time	n_iter	train_r2	val_r2	train_n_sv	val_n_sv
smo	10	0.2	0.041816	30	0.839199	0.824149	67	67
	100	0.3	0.231624	136	0.838683	0.821843	66	66
		0.1	0.161679	128	0.838229	0.820990	66	66
	10	0.3	0.037710	35	0.839234	0.824092	67	67
		0.1	0.039658	30	0.839160	0.824200	67	67
	1	0.3	0.015873	12	0.816829	0.806018	66	66
		0.2	0.016650	13	0.816266	0.805431	66	66
		0.1	0.018286	13	0.815865	0.804998	66	66
	100	0.2	0.206949	185	0.838469	0.821431	66	66
libsvm	1	0.3	0.003237	57	0.816092	0.803425	66	66
	10	0.1	0.003931	111	0.837101	0.825608	67	67
		0.2	0.001921	114	0.837315	0.825348	67	67
		0.3	0.001678	163	0.837508	0.825064	67	67
	1	0.1	0.003243	52	0.815113	0.803346	66	66
	100	0.1	0.002190	1081	0.836676	0.824330	66	66
		0.2	0.001872	762	0.836958	0.824744	66	66
	1	0.2	0.005966	52	0.815554	0.803517	66	66
	100	0.3	0.001949	1433	0.837205	0.825121	66	66
cvxopt	1	0.2	0.018971	10	0.816274	0.805607	66	66
	100	0.3	0.010127	8	0.838686	0.822088	67	67
		0.2	0.014582	8	0.838473	0.821675	67	67
		0.1	0.012519	8	0.838233	0.821232	67	67
	10	0.3	0.013504	8	0.839237	0.824234	67	67
		0.2	0.013448	8	0.839203	0.824285	67	67
		0.1	0.016795	8	0.839164	0.824329	67	67
	1	0.3	0.012157	9	0.816837	0.806148	67	67
		0.1	0.021911	9	0.815872	0.805162	67	67

Table 9: Linear SVR Lagrangian Dual formulation results with Epsilon-insensitive loss

dual	C	epsilon	fit_time	n_iter	train_r2	val_r2	train_n_sv	val_n_sv
qp	1	0.1	0.082322	44	0.684957	0.676764	67	67
		0.2	0.087632	44	0.684957	0.676764	67	67
		0.3	0.083518	44	0.684957	0.676763	67	67
	10	0.1	0.823205	728	0.736303	0.730094	67	67
		0.2	0.840446	735	0.736303	0.730093	67	67
		0.3	0.794968	741	0.736303	0.730092	67	67
	100	0.1	0.870177	728	0.736303	0.730094	67	67
		0.2	0.520331	735	0.736303	0.730093	67	67
		0.3	0.570781	741	0.736303	0.730092	67	67
bcqp	1	0.1	0.050349	32	0.683134	0.674912	67	67
		0.2	0.051552	32	0.683134	0.674912	67	67
		0.3	0.053362	32	0.683134	0.674911	67	67
	10	0.1	0.242091	207	0.738281	0.732098	67	67
		0.2	0.229870	209	0.738281	0.732097	67	67
		0.3	0.245216	210	0.738281	0.732096	67	67
	100	0.1	0.243215	207	0.738281	0.732098	67	67
		0.2	0.113374	209	0.738281	0.732097	67	67
		0.3	0.186780	210	0.738281	0.732096	67	67

Nonlinear Dual formulations The experiments results shown in 10 and 11 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. *Libsvm* also uses the *SMO* algorithm to solve the dual. The experiments results shown in 5 are obtained with α, i.e., the *learning rate* or *step size*, setted to 0.5 for the *AdaGrad* algorithm.

Table 10: Nonlinear SVR Wolfe Dual formulation results with Epsilon-insensitive loss

solver	kernel	C	epsilon	fit_time	n_iter	train_r2	val_r2	train_n_sv	val_n_sv
cvxopt	poly	1	0.1	0.015057	10	0.848547	-5.090164	23	23
			0.2	0.017265	10	-3.609617	-8.632746	6	6
			0.3	0.011037	10	-0.786399	-8.517434	4	4
		10	0.1	0.010415	10	0.968145	-5.331473	25	25
			0.2	0.022692	10	-3.609013	-8.638973	4	4
			0.3	0.011880	10	-0.780783	-8.483884	4	4
		100	0.1	0.012144	10	0.945501	-5.188429	27	27
			0.2	0.014013	10	-3.608986	-8.638995	4	4
			0.3	0.012482	10	-0.780887	-8.484199	4	4
	rbf	1	0.1	0.014662	10	0.979933	0.327020	14	14
			0.2	0.021198	10	0.961792	-0.943080	6	6
			0.3	0.017357	9	0.890164	-1.687411	5	5
		10	0.1	0.014854	10	0.977704	0.640986	14	14
			0.2	0.016555	10	0.952051	-0.963789	6	6
			0.3	0.014812	10	0.882145	-1.696148	4	4
		100	0.1	0.015500	10	0.974118	0.715636	15	15
			0.2	0.018510	10	0.965122	-0.935738	7	7
			0.3	0.018584	10	0.882145	-1.696145	4	4
smo	poly	1	0.1	80.259683	175472	0.850381	-6.479953	23	23
			0.2	4.845339	6682	-5.669765	-15.026022	6	6
			0.3	0.531903	909	-2.663418	-16.100682	4	4
		10	0.1	660.696393	1352666	0.958635	-6.309580	23	23
			0.2	3.283288	5413	-5.659396	-15.048805	4	4
			0.3	4.703530	7008	-2.652290	-16.086707	4	4
		100	0.1	3838.989156	9325434	0.956258	-6.351469	23	23
			0.2	3.581371	5413	-5.659396	-15.048805	4	4
			0.3	2.518382	7008	-2.652290	-16.086707	4	4
	rbf	1	0.1	0.035963	30	0.979269	0.337883	14	14
			0.2	0.014002	14	0.953442	-0.706423	6	6
			0.3	0.009471	9	0.880480	-1.956576	5	5
		10	0.1	0.294246	198	0.979879	0.614804	14	14
			0.2	0.018293	20	0.943679	-0.729439	5	5
			0.3	0.010107	11	0.872418	-1.965274	4	4
		100	0.1	0.869753	1199	0.976323	0.735233	14	14
			0.2	0.019184	20	0.943679	-0.729439	5	5
			0.3	0.008680	11	0.872418	-1.965274	4	4
libsvm	poly	1	0.1	0.065331	202636	0.981648	-29.427063	20	20
			0.2	0.008808	5634	0.971457	-44.854253	5	5
			0.3	0.012103	1133	0.921669	-67.995416	4	4
		10	0.1	0.577278	2329808	0.981723	-28.867737	18	18
			0.2	0.012278	4967	0.972091	-44.851795	4	4
			0.3	0.001801	925	0.922233	-67.994190	3	3
		100	0.1	1.439916	6416597	0.980670	-14.594558	24	24
			0.2	0.009960	4967	0.972091	-44.851795	4	4
			0.3	0.021700	925	0.922233	-67.994190	3	3
	rbf	1	0.1	0.015557	71	0.986549	-2.969459	16	16
			0.2	0.018801	33	0.964555	-4.149278	5	5
			0.3	0.002971	8	0.912691	-4.815179	4	4
		10	0.1	0.004562	474	0.987401	-2.477526	15	15
			0.2	0.011069	34	0.964563	-4.149231	5	5
			0.3	0.001300	8	0.913367	-4.813685	4	4
		100	0.1	0.006711	2712	0.987693	-1.428809	13	13
			0.2	0.002964	34	0.964563	-4.149231	5	5
			0.3	0.002766	8	0.913367	-4.813685	4	4

Table 11: Nonlinear SVR Lagrangian Dual formulation results with Epsilon-insensitive loss

dual	kernel	C	epsilon	fit_time	n_iter	train_r2	val_r2	train_n_sv	val_n_sv
qp	poly	1	0.1	0.013927	6	0.639582	-36.017167	66	66
			0.2	0.453497	345	0.605891	-20.582313	66	66
			0.3	0.437990	352	0.583454	-20.438316	66	66
		10	0.1	0.009494	6	0.639582	-36.017167	66	66
			0.2	0.403863	345	0.605891	-20.582313	66	66
			0.3	0.425059	352	0.583454	-20.438316	66	66
		100	0.1	0.009305	6	0.639582	-36.017167	66	66
			0.2	0.442313	345	0.605891	-20.582313	66	66
			0.3	0.384443	352	0.583454	-20.438316	66	66
	rbf	1	0.1	0.257837	128	0.684133	-4.688319	67	67
			0.2	0.376743	191	0.640105	-5.221665	67	67
			0.3	0.403506	232	0.641703	-5.218774	67	67
		10	0.1	0.115396	67	0.682317	-4.657718	67	67
			0.2	0.126903	76	0.637857	-5.267631	67	67
			0.3	0.179693	108	0.638812	-5.210237	67	67
		100	0.1	0.114128	67	0.682317	-4.657718	67	67
			0.2	0.136213	76	0.637857	-5.267631	67	67
			0.3	0.205061	108	0.638812	-5.210237	67	67
bcqp	poly	1	0.1	0.012580	8	0.640796	-36.909689	67	67
			0.2	0.409737	345	0.599211	-19.429431	66	66
			0.3	0.418357	354	0.581901	-19.286390	66	66
		10	0.1	0.012921	8	0.640796	-36.909689	67	67
			0.2	0.399091	345	0.599211	-19.429431	66	66
			0.3	0.449794	354	0.581901	-19.286390	66	66
		100	0.1	0.011779	8	0.640796	-36.909689	67	67
			0.2	0.385712	345	0.599211	-19.429431	66	66
			0.3	0.274509	354	0.581901	-19.286390	66	66
	rbf	1	0.1	0.245458	134	0.733531	-5.051724	67	67
			0.2	0.373509	244	0.669836	-5.935346	67	67
			0.3	0.467384	300	0.529430	-7.181043	67	67
		10	0.1	0.201872	134	0.733531	-5.051724	67	67
			0.2	0.381918	244	0.669836	-5.935346	67	67
			0.3	0.446960	300	0.529430	-7.181043	67	67
		100	0.1	0.230169	134	0.733531	-5.051724	67	67
			0.2	0.409371	244	0.669836	-5.935346	67	67
			0.3	0.363330	300	0.529430	-7.181043	67	67

7.2.2 Squared Epsilon-insensitive loss

Primal formulation The experiments results shown in 12 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. *Liblinear* uses the *Coordinate Gradient Descent* algorithm which minimize one coordinate at a time.

Table 12: SVR Primal formulation results with Squared Epsilon-insensitive loss

solver	C	momentum	epsilon	fit_time	n_iter	train_r2	val_r2	train_n_sv	val_n_sv
sgd	1	none	0.1	1.505412	3298	0.977343	0.972962	66	33
			0.2	1.594305	3256	0.977337	0.972946	65	33
			0.3	1.286196	3212	0.977329	0.972927	65	33
		standard	0.1	0.823314	2137	0.977359	0.972998	66	33
			0.2	0.889051	2101	0.977354	0.972985	65	33
			0.3	0.776666	2064	0.977349	0.972969	65	33
		nesterov	0.1	0.862395	2137	0.977358	0.972997	66	33
			0.2	0.841281	2104	0.977354	0.972985	65	33
			0.3	0.768695	2062	0.977348	0.972967	65	33
	10	none	0.1	0.156606	397	0.978098	0.973423	66	33
			0.2	0.193395	400	0.978098	0.973424	65	32
			0.3	0.162109	400	0.978097	0.973420	64	32
		standard	0.1	0.094696	245	0.978099	0.973502	66	33
			0.2	0.094505	248	0.978099	0.973503	65	32
			0.3	0.091287	249	0.978099	0.973505	65	32
		nesterov	0.1	0.101287	249	0.978100	0.973491	66	33
			0.2	0.096606	250	0.978100	0.973493	65	32
			0.3	0.088895	252	0.978100	0.973495	65	32
	100	none	0.1	0.026444	62	0.977779	0.973078	65	33
			0.2	0.026268	62	0.977779	0.973078	65	32
			0.3	0.024583	61	0.977778	0.973084	64	32
		standard	0.1	0.014298	34	0.977853	0.973014	66	32
			0.2	0.014198	34	0.977853	0.973017	64	32
			0.3	0.013292	40	0.977853	0.973014	64	31
		nesterov	0.1	0.017095	41	0.977838	0.973043	66	32
			0.2	0.016391	41	0.977838	0.973042	64	32
			0.3	0.016078	41	0.977838	0.973045	64	31
liblinear	1	-	0.1	0.001082	84	0.978134	0.973997	67	32
			0.2	0.001247	84	0.978132	0.974006	66	32
			0.3	0.001004	83	0.978130	0.974011	66	32
	10	-	0.1	0.002923	768	0.978183	0.973959	66	33
			0.2	0.003070	765	0.978183	0.973965	66	33
			0.3	0.003016	765	0.978183	0.973970	66	32
	100	-	0.1	0.003944	1000	0.978025	0.973097	66	33
			0.2	0.004085	1000	0.978029	0.973107	66	33
			0.3	0.003432	1000	0.978033	0.973116	65	32

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.

From all these experiments we can see as, for what about the *primal* formulations, the results provided from the *custom* implementations are strongly similar to those of *sklearn* implementations, i.e., *liblinear* [9] implementations, with a slight exception about the time gap obviously due to the different core implementation languages, Python and C respectively.

Meanwhile, for what about the *dual* formulations we can notice as *cvxopt* [11] underperforms the *sklearn* implementations, i.e., *libsvm* [10] implementations, 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. Despite this, the *custom* implementations does not overperform the *cvxopt* [11] probably due to the gap generated from the different core implementation languages, again Python and C respectively. For these reasons, *sklearn* provides better results in terms of time wrt the other implementations since it is designed to work in a large-scale context and its core is implemented in C. Furthermore, in the SVC example with the polynomial kernel of degree 5, we can see that the time gap is significatively, properly two different orders of magnitude ($\simeq$ 29min vs. $\simeq$ 19ms), and this could not depend just only by the different implementation languages; it's probable that *liblinear* [9] adopts some heuristics, i.e., low rank approssimations of the kernel matrix, to deal with the polynomial kernel in case of high degree.

Important consideration involves the number of support vector machines: the *Lagrangian dual* formulation tends to select all the data points as support vectors, so it makes the model complex and it tends to give low scores wrt the equivalent *Wolfe dual* formulation. In particular, the *Lagrangian relaxation* resulting from the *Wolfe dual* always gives rise to a nonsmooth optimization with an exception for the SVC with a Gaussian kernel where the two formulations solve exactly the same problem. In all the other cases the goodness of the solution depends on the residue in the solution of the *Lagrangian dual* at each step; one of the wrost results certainly concerns the SVC with the polynomial kernel of degree 3, where the residue is in the order of +02/03 and so the approssimation is horrible. Finally, 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] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

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

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