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:

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

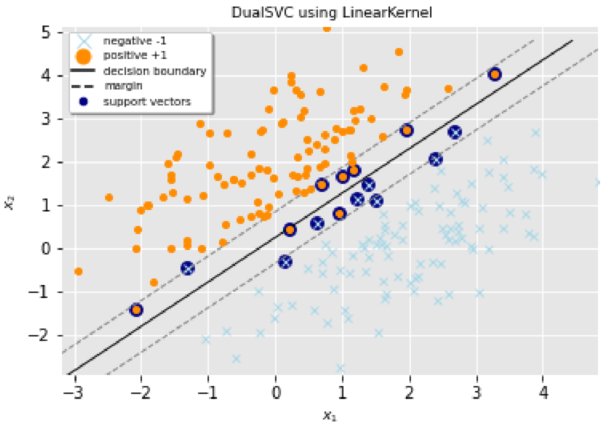


Figure 1: Linear SVC hyperplane

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:

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

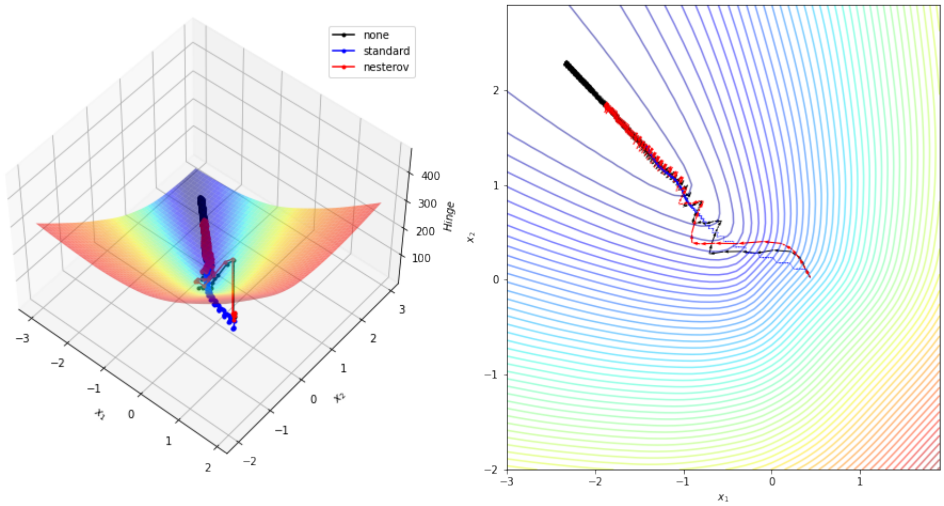


Figure 2: SVC Hinge loss with different optimization steps

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 $\alpha_i \geq 0, \mu_i \geq 0 \ \forall_i$:

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

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:

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

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 $\mu \geq 0, \lambda_+ \geq 0, \lambda_- \geq 0$:

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

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:

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

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

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

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

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.

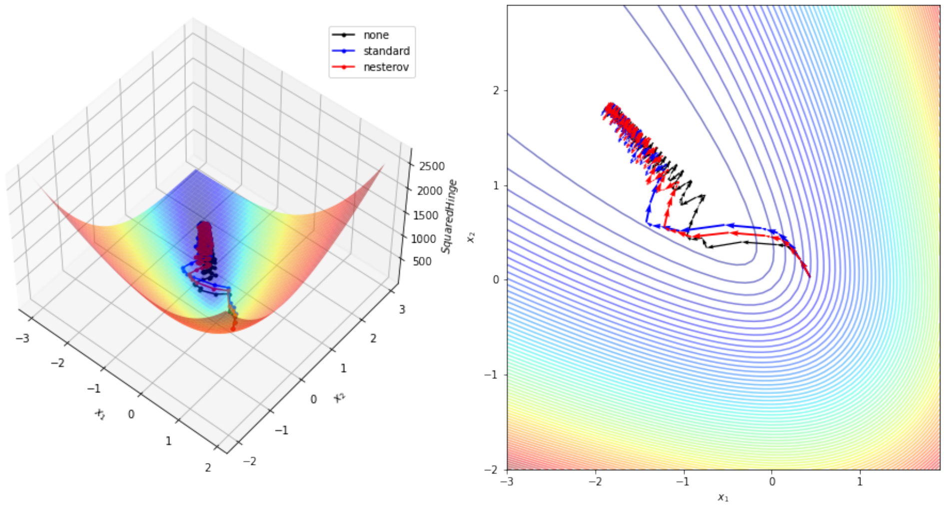


Figure 3: SVC Squared Hinge loss with different optimization steps

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:

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

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:

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

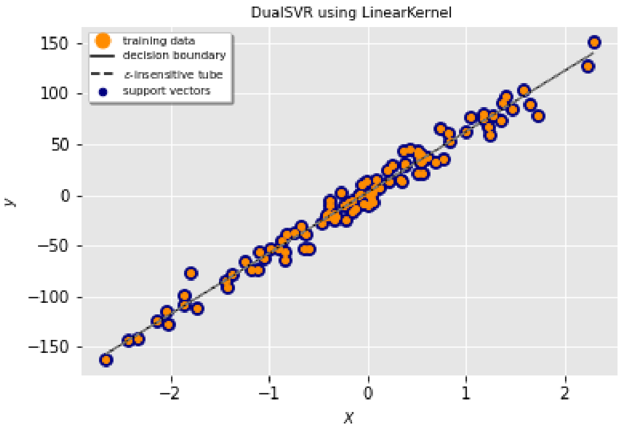


Figure 4: Linear SVR hyperplane

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.

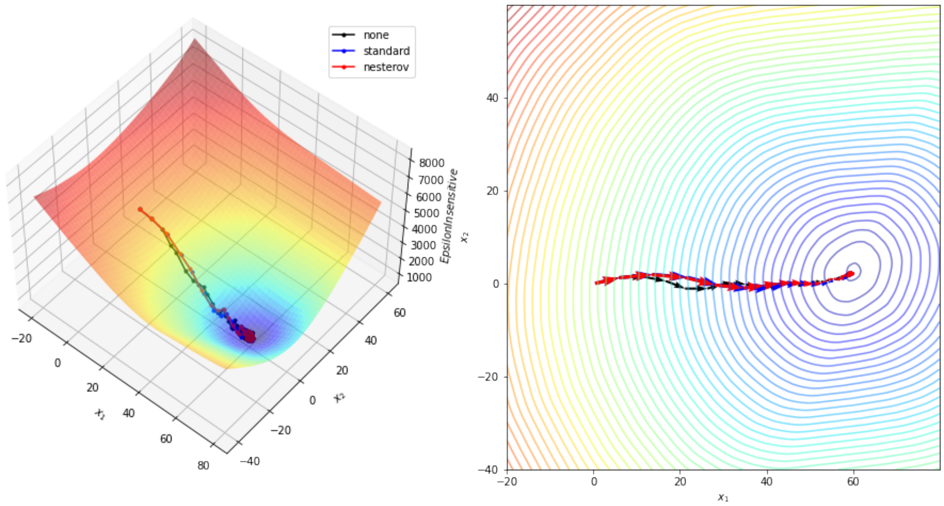


Figure 5: SVR Epsilon-insensitive loss with different optimization steps

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 $q^T = [1, \dots, 1]$.
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 $u^T = [C, \dots, C]$.

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

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

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

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

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

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.

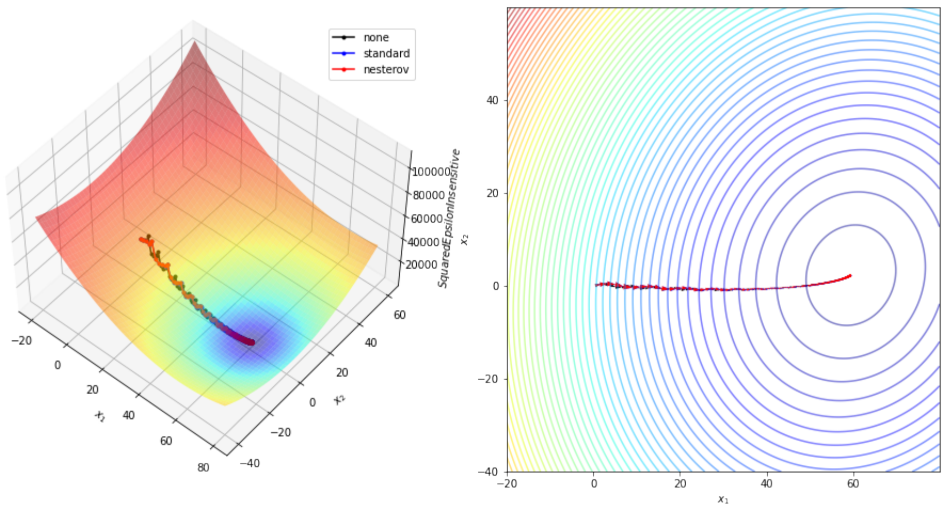


Figure 6: SVC Squared Epsilon-insensitive loss with different optimization steps

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.

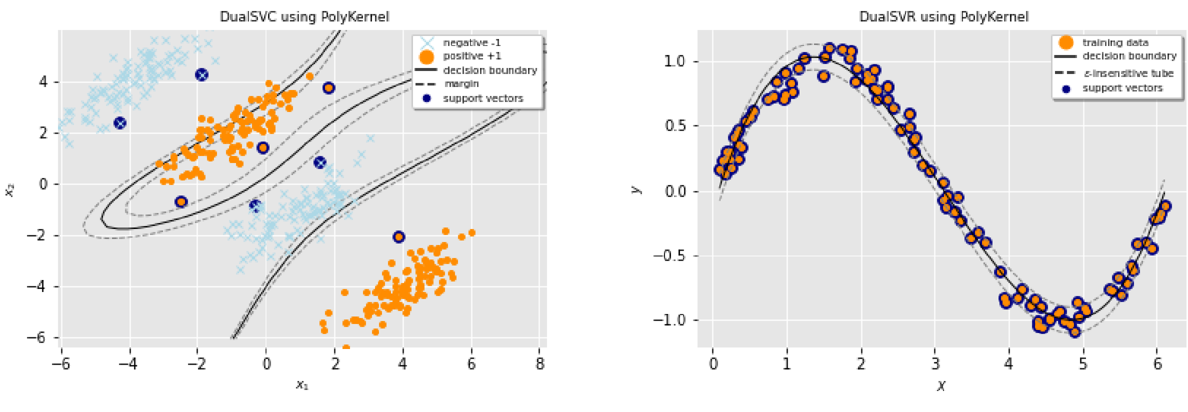
The reason that this *kernel trick* is useful is that there are many classification/regression problems that are nonlinearly separable/regressable in the *input space*, which might be in a higher dimensionality *feature space* given a suitable mapping $x \to \phi(x)$.

5.1 Polynomial kernel

The *polynomial* kernel is defined as:

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

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

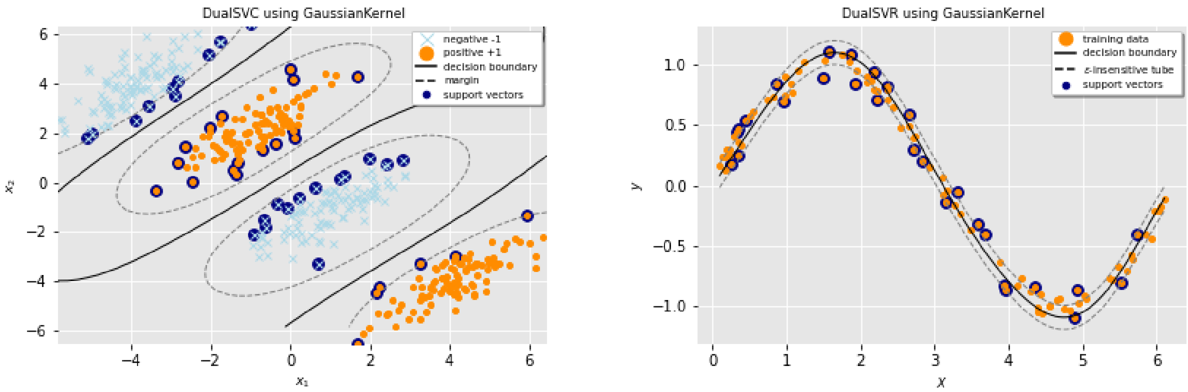
The *gaussian* kernel is defined as:

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

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 $\mathcal{O}\Big(\frac{1}{\sqrt{t}}\Big)$.

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.

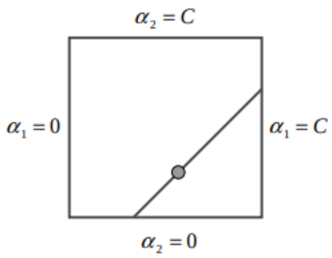


Figure 10: SMO for two Lagrange multipliers

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 α ∈ ℜ^n, α ← 0
    Initialize the empty set I0 ← {i : 0 < α_i < C}
    Initialize the set I1 ← {i : y_i = +1, α_i = 0} to contain all the indices of the training examples of class +1
    Initialize the empty set I2 ← {i : y_i = −1, α_i = C}
    Initialize the empty set I3 ← {i : y_i = +1, α_i = C}
    Initialize the set I4 ← {i : y_i = −1, α_i = 0} to contain all the indices of the training examples of class −1
    Initialize b_up ← −1
    Initialize b_low ← +1
    Initialize the error cache vector errors ∈ ℜ^n, errors ← 0
    while num_changed > 0 or examine_all = True do
        num_changed ← 0
        examine_all ← True
        if examine_all = True then
            for i ← 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_i 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: $i2$-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 (83) or (84)
    if $L = H$ **then**
      return False
    end if
    Compute η by (85) ▷ 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 α_2^{new} by (86)
      Compute $\alpha_2^{new,clipped}$ by (87)
    end if
    if changes in $\alpha_2^{new,clipped}$ are larger than some eps **then**
      Compute α_1^{new} by (88)
      Update $\alpha_2^{new,clipped}$ and α_1^{new}
      for i in $I0$ **do**
        Update $errors_i$ using new Lagrange multipliers
      end for
      Update α 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

6.3.2 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, ε, tol)
    Initialize the Lagrange multipliers vector α⁺ ∈ ℜⁿ, α⁺ ← 0
    Initialize the Lagrange multipliers vector α⁻ ∈ ℜⁿ, α⁻ ← 0
    Initialize the empty set I0 ← {i : 0 < α_i⁺, α_i⁻ < C}
    Initialize the set I1 ← {i : α_i⁺ = 0, α_i⁻ = 0} to contain all the indices of the training examples
    Initialize the empty set I2 ← {i : α_i⁺ = 0, α_i⁻ = C}
    Initialize the empty set I3 ← {i : α_i⁺ = C, α_i⁻ = 0}
    Initialize i_up ← 0                ▷ or any other target index i_up from the training examples
    Initialize i_low ← 0               ▷ or any other target index i_low from the training examples
    Initialize b_up ← y_{i_up} + ε
    Initialize b_low ← y_{i_low} − ε
    Initialize the error cache vector errors ∈ ℜⁿ, errors ← 0
    while num_changed > 0 or examine_all = True do
        num_changed ← 0
        examine_all ← True
        if examine_all = True then
            for i ← 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_i⁺ and α_i⁻ 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 (108)
    return α⁺, α⁻, 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
  Compute L and H using (93), (96), (99) or (102)
  $finished = False$
  while not $finished$ **do** ▷ this loop is passed at most three times
    if $L < H$ **then**
      Compute η by (85) ▷ 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 (94), (100) or (97), (103) respectively
        Compute $\alpha_2^{+,new,clipped}$ or $\alpha_2^{-,new,clipped}$ by (87)
      end if
      Compute $\alpha_1^{+,new}$ or $\alpha_1^{-,new}$ using (95), (98) or (101), (104) 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 α^+ and α^- 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

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.

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	0.751530	2883	0.982494	0.979949	37	18
		standard	0.553684	1921	0.984981	0.984999	33	17
		nesterov	0.481777	1921	0.984981	0.984999	33	17
	10	none	0.845532	3382	0.984981	0.984999	10	6
		standard	0.637896	2624	0.987487	0.989974	10	6
		nesterov	0.528413	2071	0.984981	0.984999	10	6
	100	none	0.442558	1700	0.989994	0.989974	9	5
		standard	0.319567	1126	0.989994	0.989974	7	4
		nesterov	0.140702	405	0.989994	0.989974	8	5
liblinear	1	-	0.001454	428	0.989994	0.989974	11	6
	10	-	0.001551	744	0.987487	0.984999	5	4
	100	-	0.001547	1000	0.985000	0.989974	7	2

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	0.076588	61	0.985000	0.989974	12	12
	10	0.076181	73	0.985000	0.979949	7	7
	100	0.373685	1196	0.985000	0.989974	6	6
libsvm	1	0.003042	99	0.987487	0.989974	12	12
	10	0.002565	90	0.987487	0.984999	7	7
	100	0.003477	2647	0.985000	0.984999	6	6
cvxopt	1	0.020731	10	0.985000	0.989974	12	12
	10	0.016735	10	0.985000	0.989974	7	7
	100	0.029245	10	0.985000	0.989974	8	8

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	0.006492	1	0.985019	0.989974	131	131
	10	0.006105	1	0.985019	0.989974	131	131
	100	0.005686	1	0.985019	0.989974	131	131
bcqp	1	0.006973	1	0.985019	0.989974	130	130
	10	0.006043	1	0.985019	0.989974	130	130
	100	0.005027	1	0.985019	0.989974	130	130

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	0.356696	88	0.851206	0.678412	28	28
		10	0.270690	105	0.897426	0.660682	9	9
		100	0.255688	198	0.916209	0.727921	8	8
	rbf	1	0.245166	49	1.000000	1.000000	41	41
		10	0.204894	52	1.000000	1.000000	13	13
		100	0.188844	64	1.000000	1.000000	11	11
libsvm	poly	1	0.005154	160	1.000000	0.992481	28	28
		10	0.004598	436	1.000000	0.987469	10	10
		100	0.006573	149	1.000000	0.987469	8	8
	rbf	1	0.004991	115	1.000000	1.000000	41	41
		10	0.002472	164	1.000000	1.000000	13	13
		100	0.003032	268	1.000000	1.000000	12	12
cvxopt	poly	1	0.105617	10	0.851206	0.678412	28	28
		10	0.095708	10	0.911206	0.688026	10	10
		100	0.072921	10	0.916209	0.722889	9	9
	rbf	1	0.093452	10	1.000000	1.000000	43	43
		10	0.073576	10	1.000000	1.000000	14	14
		100	0.069153	10	1.000000	1.000000	13	13

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	1.315738	255	0.733773	0.513784	184	184
		10	1.178677	255	0.733773	0.513784	184	184
		100	1.203893	255	0.733773	0.513784	184	184
	rbf	1	1.241991	229	0.822618	0.727734	186	186
		10	1.696262	376	0.751255	0.503741	153	153
		100	1.312007	305	0.833652	0.586447	178	178
bcqp	poly	1	1.480641	337	0.783729	0.538810	199	199
		10	1.266989	337	0.783729	0.538810	199	199
		100	1.023042	337	0.783729	0.538810	199	199
	rbf	1	0.020001	1	0.998747	0.982568	240	240
		10	0.021672	1	0.998747	0.982568	240	240
		100	0.023396	1	0.998747	0.982568	240	240

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.

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	0.322997	1317	0.982494	0.979949	37	19
		standard	0.219974	869	0.984981	0.979949	34	17
		nesterov	0.222835	869	0.982494	0.979949	34	17
	10	none	0.213107	491	0.984981	0.980024	16	9
		standard	0.105956	309	0.984981	0.980024	15	8
		nesterov	0.101582	311	0.984981	0.980024	15	9
	100	none	0.077881	192	0.989994	0.989974	10	5
		standard	0.076620	199	0.987487	0.989974	8	5
		nesterov	0.076583	169	0.989994	0.989974	9	5
liblinear	1	-	0.001921	366	0.985000	0.989974	18	10
	10	-	0.001965	1000	0.985000	0.989974	14	8
	100	-	0.001718	1000	0.985000	0.980024	13	7

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.

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	0.122940	238	0.410998	0.402111	66	33
			0.2	0.110922	238	0.410998	0.402111	66	33
			0.3	0.112015	238	0.410998	0.402111	66	33
		standard	0.1	0.082897	152	0.427455	0.418695	66	33
			0.2	0.064338	152	0.427455	0.418695	66	33
			0.3	0.068753	153	0.430658	0.421952	66	33
		nesterov	0.1	0.066637	152	0.427207	0.418447	66	33
			0.2	0.072682	152	0.427207	0.418447	66	33
			0.3	0.069598	153	0.430409	0.421703	66	33
	10	none	0.1	0.119095	278	0.975885	0.971366	66	33
			0.2	0.112762	276	0.975752	0.971251	66	33
			0.3	0.106941	274	0.975685	0.971108	65	32
		standard	0.1	0.068403	174	0.976441	0.971938	66	33
			0.2	0.062866	174	0.976441	0.971939	66	32
			0.3	0.067308	173	0.976363	0.971751	65	32
		nesterov	0.1	0.075617	174	0.976374	0.971878	66	33
			0.2	0.069544	174	0.976374	0.971879	66	32
			0.3	0.073690	174	0.976347	0.971756	65	32
	100	none	0.1	0.042772	113	0.977986	0.973377	65	33
			0.2	0.038640	114	0.977986	0.973380	65	32
			0.3	0.037846	118	0.977986	0.973380	64	31
		standard	0.1	0.027462	72	0.977997	0.973440	66	33
			0.2	0.027894	72	0.977997	0.973440	65	32
			0.3	0.026422	73	0.977997	0.973441	64	31
		nesterov	0.1	0.033902	78	0.977995	0.973448	66	33
			0.2	0.028685	69	0.977994	0.973450	65	32
			0.3	0.027387	77	0.977995	0.973450	64	31
liblinear	1	-	0.1	0.000778	11	0.918768	0.916773	66	33
			0.2	0.000682	10	0.918763	0.916602	65	32
			0.3	0.000695	13	0.919296	0.917061	65	32
	10	-	0.1	0.000881	162	0.977849	0.972087	65	33
			0.2	0.001189	178	0.977852	0.972041	65	33
			0.3	0.000820	113	0.977871	0.972151	64	33
	100	-	0.1	0.001235	638	0.977725	0.974270	65	33
			0.2	0.001333	686	0.977664	0.974138	66	33
			0.3	0.001179	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	1	0.1	0.017338	15	0.917773	0.914442	66	66
		0.2	0.022597	13	0.918341	0.915019	66	66
		0.3	0.039421	60	0.918942	0.915576	66	66
	10	0.1	0.053832	56	0.977920	0.972445	66	66
		0.2	0.139534	219	0.977926	0.972457	65	65
		0.3	0.053870	38	0.977953	0.972544	65	65
	100	0.1	0.524643	1508	0.977788	0.974139	66	66
		0.2	0.266404	394	0.977742	0.974022	66	66
		0.3	0.383196	900	0.977737	0.973939	66	66
libsvm	1	0.1	0.003532	63	0.917627	0.915448	66	66
		0.2	0.003926	102	0.918194	0.915985	66	66
		0.3	0.002088	54	0.918786	0.916554	66	66
	10	0.1	0.001896	282	0.977852	0.972051	66	66
		0.2	0.001990	193	0.977851	0.972025	65	65
		0.3	0.001814	593	0.977870	0.972135	65	65
	100	0.1	0.003821	2621	0.977723	0.974270	66	66
		0.2	0.003551	2709	0.977673	0.974122	66	66
		0.3	0.003090	4141	0.977655	0.974045	66	66
cvxopt	1	0.1	0.017656	9	0.917772	0.914479	67	67
		0.2	0.017439	9	0.918341	0.915058	67	67
		0.3	0.013357	10	0.918942	0.915614	66	66
	10	0.1	0.024514	9	0.977920	0.972466	67	67
		0.2	0.018264	9	0.977926	0.972474	67	67
		0.3	0.025372	10	0.977954	0.972562	66	66
	100	0.1	0.020310	9	0.977788	0.974150	67	67
		0.2	0.024708	9	0.977742	0.974033	67	67
		0.3	0.021550	9	0.977737	0.973956	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	0.711464	653	0.876534	0.870926	67	67
		0.2	0.820021	653	0.876534	0.870927	67	67
		0.3	0.648664	653	0.876534	0.870927	67	67
	10	0.1	0.510787	519	0.731825	0.722021	67	67
		0.2	0.558384	524	0.731825	0.722021	67	67
		0.3	0.519468	530	0.731825	0.722020	67	67
	100	0.1	0.631134	519	0.731825	0.722021	67	67
		0.2	0.520121	524	0.731825	0.722021	67	67
		0.3	0.539273	530	0.731825	0.722020	67	67
bcqp	1	0.1	0.630620	522	0.731073	0.721200	67	67
		0.2	0.639167	524	0.731073	0.721199	67	67
		0.3	0.630822	526	0.731073	0.721199	67	67
	10	0.1	0.610205	539	0.733638	0.723925	67	67
		0.2	0.657547	541	0.733638	0.723924	67	67
		0.3	0.637749	543	0.733638	0.723924	67	67
	100	0.1	0.635853	539	0.733638	0.723925	67	67
		0.2	0.527298	541	0.733638	0.723924	67	67
		0.3	0.494042	543	0.733638	0.723924	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

				fit_time	n_iter	train_r2	val_r2	train_n_sv	val_n_sv
solver	kernel	C	epsilon						
smo	poly	1	0.1	26.889628	38855	0.041472	-20.235896	20	20
			0.2	2.366264	3607	-5.523525	-24.564063	5	5
			0.3	1.063623	1440	0.121370	-48.392546	4	4
		10	0.1	368.357509	939877	0.357291	-11.310659	19	19
			0.2	2.833490	4317	-4.173750	-24.504696	4	4
			0.3	0.861725	1396	0.227988	-48.232256	3	3
		100	0.1	2431.854217	7053217	0.334627	-12.438814	20	20
			0.2	2.970568	4317	-4.173750	-24.504696	4	4
			0.3	0.860159	1396	0.227988	-48.232256	3	3
	rbf	1	0.1	0.070852	53	0.982232	-0.107260	20	20
			0.2	0.015128	14	0.959560	-1.732727	6	6
			0.3	0.011491	12	0.935447	-2.495198	5	5
		10	0.1	0.527717	398	0.981400	0.739074	18	18
			0.2	0.015757	16	0.951327	-1.750890	5	5
			0.3	0.010530	12	0.929203	-2.514675	4	4
		100	0.1	1.742027	2574	0.979866	0.087491	17	17
			0.2	0.013848	16	0.951327	-1.750890	5	5
			0.3	0.008658	12	0.929203	-2.514675	4	4
libsvm	poly	1	0.1	0.019525	70279	0.978147	-12.483928	20	20
			0.2	0.006174	3221	0.967412	-40.598436	5	5
			0.3	0.011255	1293	0.917912	-70.616835	4	4
		10	0.1	0.797140	3301383	0.978557	-11.345334	19	19
			0.2	0.010065	4829	0.969666	-40.590296	4	4
			0.3	0.007448	1322	0.919258	-70.610551	3	3
		100	0.1	2.676930	16883444	0.978709	-10.897898	23	23
			0.2	0.012942	4829	0.969666	-40.590296	4	4
			0.3	0.001841	1322	0.919258	-70.610551	3	3
	rbf	1	0.1	0.008771	88	0.982555	-0.048065	20	20
			0.2	0.001233	21	0.961248	-0.784327	6	6
			0.3	0.007089	24	0.911568	-1.128042	5	5
		10	0.1	0.016236	564	0.984309	0.794712	17	17
			0.2	0.014555	26	0.962020	-0.784287	5	5
			0.3	0.008202	18	0.911949	-1.126343	5	5
		100	0.1	0.008616	3195	0.984358	0.339781	17	17
			0.2	0.002037	26	0.962020	-0.784287	5	5
			0.3	0.004992	18	0.911949	-1.126343	5	5
cvxopt	poly	1	0.1	0.017180	10	0.116876	-16.797868	20	20
			0.2	0.017883	10	-4.948070	-13.077320	5	5
			0.3	0.012003	10	0.681960	-44.133994	4	4
		10	0.1	0.011285	10	0.953326	-10.961264	22	22
			0.2	0.012770	10	-3.642113	-12.971183	4	4
			0.3	0.012743	10	0.783199	-43.962270	3	3
		100	0.1	0.011621	10	0.951861	-9.443374	44	44
			0.2	0.013602	10	-3.642117	-12.971070	4	4
			0.3	0.013014	10	0.783091	-43.966539	3	3
	rbf	1	0.1	0.016085	10	0.981573	-0.007460	20	20
			0.2	0.019686	9	0.958544	-1.311340	7	7
			0.3	0.015890	10	0.932149	-2.681785	5	5
		10	0.1	0.015052	10	0.981853	0.676330	20	20
			0.2	0.022038	10	0.950457	-1.330465	6	6
			0.3	0.016398	10	0.925512	-2.695479	5	5
		100	0.1	0.021050	10	0.979321	0.229817	20	20
			0.2	0.016424	10	0.950458	-1.330472	6	6
			0.3	0.015113	10	0.924244	-2.228071	5	5

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	0.013639	9	0.636530	-16.889746	67	67
			0.2	0.032891	13	0.615075	-12.838670	67	67
			0.3	0.026618	20	0.641814	-12.639351	67	67
		10	0.1	0.014575	9	0.636530	-16.889746	67	67
			0.2	0.018876	13	0.615075	-12.838670	67	67
			0.3	0.025337	20	0.641814	-12.639351	67	67
		100	0.1	0.012551	9	0.636530	-16.889746	67	67
			0.2	0.020681	13	0.615075	-12.838670	67	67
			0.3	0.026794	20	0.641814	-12.639351	67	67
	rbf	1	0.1	0.326021	165	0.689146	-4.135603	67	67
			0.2	0.362601	168	0.668805	-4.085457	67	67
			0.3	0.502042	270	0.621731	-4.407841	67	67
		10	0.1	0.079275	40	0.712605	-3.657736	67	67
			0.2	0.151516	84	0.704434	-3.441384	67	67
			0.3	0.185606	113	0.614242	-4.592174	67	67
		100	0.1	0.074331	40	0.712605	-3.657736	67	67
			0.2	0.164555	84	0.704434	-3.441384	67	67
			0.3	0.190906	113	0.614242	-4.592174	67	67
bcqp	poly	1	0.1	0.030819	11	0.636315	-16.926529	67	67
			0.2	0.033619	15	0.612743	-12.193272	67	67
			0.3	0.039668	30	0.636335	-12.001298	67	67
		10	0.1	0.014905	11	0.636315	-16.926529	67	67
			0.2	0.021995	15	0.612743	-12.193272	67	67
			0.3	0.048245	30	0.636335	-12.001298	67	67
		100	0.1	0.014796	11	0.636315	-16.926529	67	67
			0.2	0.022666	15	0.612743	-12.193272	67	67
			0.3	0.042290	30	0.636335	-12.001298	67	67
	rbf	1	0.1	0.081437	47	0.736623	-3.125274	67	67
			0.2	0.214357	125	0.678979	-3.888105	67	67
			0.3	0.491642	327	0.582883	-5.520437	67	67
		10	0.1	0.079863	47	0.736623	-3.125274	67	67
			0.2	0.204089	125	0.678979	-3.888105	67	67
			0.3	0.355141	327	0.582883	-5.520437	67	67
		100	0.1	0.111757	47	0.736623	-3.125274	67	67
			0.2	0.190357	125	0.678979	-3.888105	67	67
			0.3	0.291476	327	0.582883	-5.520437	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.

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	1.358664	3298	0.977343	0.972962	66	33
			0.2	1.384782	3256	0.977337	0.972946	65	33
			0.3	1.264827	3212	0.977329	0.972927	65	33
		standard	0.1	0.884823	2137	0.977359	0.972998	66	33
			0.2	0.958515	2101	0.977354	0.972985	65	33
			0.3	0.890440	2064	0.977349	0.972969	65	33
		nesterov	0.1	1.016360	2137	0.977358	0.972997	66	33
			0.2	1.023388	2104	0.977354	0.972985	65	33
			0.3	0.838863	2062	0.977348	0.972967	65	33
	10	none	0.1	0.153874	397	0.978098	0.973423	66	33
			0.2	0.158299	400	0.978098	0.973424	65	32
			0.3	0.147298	400	0.978097	0.973420	64	32
		standard	0.1	0.121387	245	0.978099	0.973502	66	33
			0.2	0.131064	248	0.978099	0.973503	65	32
			0.3	0.101980	249	0.978099	0.973505	65	32
		nesterov	0.1	0.097967	249	0.978100	0.973491	66	33
			0.2	0.095231	250	0.978100	0.973493	65	32
			0.3	0.100151	252	0.978100	0.973495	65	32
	100	none	0.1	0.024647	62	0.977779	0.973078	65	33
			0.2	0.024243	62	0.977779	0.973078	65	32
			0.3	0.020621	61	0.977778	0.973084	64	32
		standard	0.1	0.015615	34	0.977853	0.973014	66	32
			0.2	0.015170	34	0.977853	0.973017	64	32
			0.3	0.017275	40	0.977853	0.973014	64	31
		nesterov	0.1	0.024583	41	0.977838	0.973043	66	32
			0.2	0.021284	41	0.977838	0.973042	64	32
			0.3	0.020673	41	0.977838	0.973045	64	31
liblinear	1	-	0.1	0.001153	84	0.978134	0.973997	67	32
			0.2	0.001002	84	0.978132	0.974006	66	32
			0.3	0.001159	83	0.978130	0.974011	66	32
	10	-	0.1	0.003323	768	0.978183	0.973959	66	33
			0.2	0.003069	765	0.978183	0.973965	66	33
			0.3	0.002587	765	0.978183	0.973970	66	32
	100	-	0.1	0.003690	1000	0.978025	0.973097	66	33
			0.2	0.004136	1000	0.978029	0.973107	66	33
			0.3	0.003729	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, i.e., Python and C respectively, and due to the *Coordinate Gradient Descent* algorithm used by *liblinear* which minimizes one coordinate at a time.

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 approximations 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 worst 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 approximation 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.