Machine Learning Algorithms Report

Mahdi Ranjbar Philippe Massouf Maxime Jauroyon
{mahdi.ranjbar, philipe.massouf, maxime.jauroyon}@universite-paris-saclay.fr

March 26, 2023

1 Goal of the lab exercise

Support Vector Machines (SVMs) are machine learning algorithms that work by finding a hyperplane that separates the input data into different classes while maximizing the margin between the two classes. This makes SVMs well suited for binary classification problems.

In this particular study, we will be focusing on training a binary SVM classifier using three different algorithms:

- Sub-gradient descent on the primal problem
- Projected gradient ascent on the dual problem
- Box constrained coordinate ascent on the dual

Each of these algorithms has its own strengths and weaknesses, and we will compare their performance on the given dataset.

The dataset used in this study was generated using scikit-learn, a popular Python library for machine learning. The dataset includes a set of features and corresponding labels, which we will use to train our SVM classifier. For all of the algorithms, denote X our input, Y or Y_gold the diagonal matrix containing the target and y_gold the target in column form.

2 Algorithms explanations

1. Sub-gradient descent on the primal problem Sub-gradient descent is an iterative optimization algorithm used to minimize a convex function. The objective function for the primal formulation of the SVM is:

$$\sum_i \max(0, 1 - X_i a y_i) + \frac{c}{2}\|a\|_2^2 \tag{1}$$

where c is the regularization weight.

Sub-gradient descent works by iteratively updating the weight vector and bias term using the sub-gradient of the objective function at the current point. The sub-gradient is a generalization of the gradient.

We can compute the subgradient by checking the different cases :
If $1 - X_i a y_i \geq 0$:

$$\begin{aligned}\nabla_a(\sum_i (1 - X_i a y_i) + \frac{c}{2}\|a\|_2^2 &= \sum_i (-X_i y_i) + \frac{2ca}{2} \\ &= \sum_i (-X_i y_i) + ca\end{aligned}$$

If $1 - X_i a y_i < 0$:

$$\begin{aligned}\nabla_a(\frac{c}{2}\|a\|_2^2) &= \frac{2ca}{2} \\ &= ca\end{aligned}$$

So :

$$\nabla_a(\sum_i \max(0, 1 - X_i a y_i) + \frac{c}{2}\|a\|_2^2) = \sum_i \mathbb{1}_{1-X_i a y_i \geq 0}(-X_i y_i) + ca$$

Which gives us the following update rule

$$a_{t+1} = a_t - \eta_t * g \tag{2}$$

Where g is the subgradient and η the step size.
Implementation details: We use a mask :

```
mask = mask = (1 - (y_gold * (X @ a))) >= 0
```

The objective function calculated for each epoch becomes:

```
obj = (1 - (y_gold[mask] * (X[mask] @ a))).sum() + (regularization_weight / 2) * (
    a.T @ a)
```

And the subgradient:

```
sub_gradient = (-X[mask].T @ Y_gold[mask]).sum(axis=1) + regularization_weight * a
```

Now we can plot the results of the algorithm on our dataset:

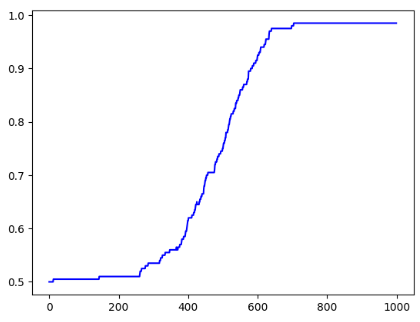
Figure 1: Objective function per epoch

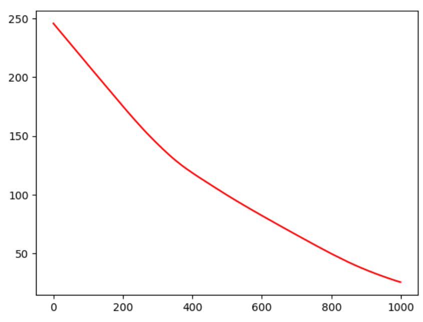
Figure 2: Accuracy per epoch

2. Projected gradient ascent on the dual problem Projected gradient ascent on the dual problem is an iterative optimization algorithm that involves taking small steps in the direction of the gradient of the objective function, subject to the constraint that the new solution remains within the feasible region defined by the constraints.

We first need to compute the objective for the dual problem. Let's rewrite the primal problem:

$$\min_a \sum_i \max\left(0, 1 - X_i a y_i\right) + \frac{c}{2}\|a\|_2^2 = \min_a \sum_i l(X_i a y_i) + c \times h(a)$$

with $l(t_i) = max(0, 1 - t_i)$ and $h(a) = \frac{1}{2}\|a\|_2^2$

$$= \min_a f(XYa) + H(a)$$

with $f(XYa) = \sum_i l(X_i a y_i)$ and $H(a) = c \times h(a)$

we use $U = YX$ and $t = a$

$$= \min_a f(Ut) + H(t)$$

and with Fenchel dual problem theorem we have:

$$= \max_\lambda -f^*(\lambda) - H^*\left(-U^\top \lambda\right)$$

- Let's start with H^*

$$\begin{aligned} H^*(t) &= \sup_{j \in \operatorname{dom} h} \langle v, t\rangle - H(v) = \sup \langle v, t\rangle - ch(v) \\ &= c \sup_{u \in \operatorname{dom} h} \frac{1}{c}\langle v, t\rangle - h(u) \\ &= c \sup_{u \in \operatorname{dom} h} \langle u, \frac{t}{c}\rangle - h(u) \\ &= ch^*\left(\frac{t}{c}\right) \end{aligned}$$

So now we need to compute $h*$.

$$\begin{aligned} h^*(t) &= \sup_{u \in \operatorname{dom} h} \langle u, t\rangle - h(u) \\ &= \sup_{u \in \operatorname{dom} h} \langle u, t\rangle - \frac{1}{2}|u|^2 \end{aligned}$$

Let $\hat{u}$ be a maximizer, then:

$$\begin{aligned} \frac{\partial}{\partial \hat{u}}\left(\langle \hat{u}, t\rangle - \frac{1}{2}|\hat{u}|_2^2\right) &= 0 \\ h'(\hat{u}) = \hat{u} &= t \end{aligned}$$

So:

$$\begin{aligned} h^*(t) &= \langle t, t\rangle - h(t) \\ &= \|t\|_2^2 - \frac{1}{2}\|t\|_2^2 = \frac{1}{2}\|t\|_2^2 = h(t) \end{aligned}$$

We are now able to compute $ch^*(\frac{t}{c})$

$$\begin{aligned} ch^*(\frac{t}{c}) &= ch(\frac{t}{c}) = c\frac{1}{2}\|\frac{t}{c}\|_2^2 \\ &= \frac{c}{2}\langle\frac{t}{c}, \frac{t}{c}\rangle = \frac{c}{2c^2}\langle t, t\rangle \\ &= \frac{1}{2c}\langle t, t\rangle = \frac{1}{2c}\|t\|^2 \end{aligned}$$

So:

$$\begin{aligned} H^*(-U^T\lambda) &= \frac{1}{2c}\| - U^T\lambda\|_2^2 = \frac{1}{2c}\lambda^T U U^T \lambda \\ &= \frac{1}{2c}\lambda^T Y X X^T Y \lambda \end{aligned}$$

Let's compute f^* now:

$$f^*(t) = \sum_i l^*(t_i)$$

And for $l(w) = max(0, 1 - w)$ then $l^*(t) = t + f_{[-1,0]}(t)$, so:

$$\begin{aligned} f^*(t) &= \sum_i t_i \\ S.T. - 1 &\leq t_i \leq 0, \forall 1 \leq i \leq n \end{aligned}$$

We can now compute the new objective:

$$\begin{aligned} &\max_\lambda - f^*(\lambda) - H^*(-U^T\lambda) \\ &= \max_\lambda - \sum_i \lambda_i - \frac{1}{2c}\lambda^T Y X X^T Y \\ &S.T. - 1 \leq \lambda_i \leq 0, \qquad \forall 1 \leq i \leq n \end{aligned}$$

We will need to recover the primal variables from dual variables. From KKT's stationarity conditions, for optimal primal and dual variables we have:

$$\begin{aligned}\nabla_a L(a,t,\lambda) &= 0\\ \nabla_a\Big(\sum_i max(0,1-t_i)+\frac{c}{2}\|a\|_2^2+\lambda^T(YXa-t)\Big) &= 0\\ ca+(\lambda^T YX)^T &= 0\end{aligned}$$

So:

$$a = \frac{-X^T Y\lambda}{c}$$

We can now compute the gradient:

$$\begin{aligned}\nabla_\lambda\Big(-\sum_i \lambda_i - \frac{1}{2c}\lambda^T YXX^T Y\lambda\Big) &= -1-\frac{1}{2c}2(YXX^TY)\lambda\\ &= -1-\frac{1}{c}(YXX^TY)\lambda\end{aligned}$$

To get the update rule, we have to add the projection operator because of the constraint on λ, so we'll clip the result between -1 and 0 :

$$\lambda^{t+1} = \text{Clip}_{[-1,0]}(\lambda^t + \eta * (-1 - \frac{1}{c}(YXX^TY)\lambda)$$

Implementation details: We make a function to get the dual variables from the primal variables :

```python
def dual_vars_to_primal_vars(dual_vars, X, Y_gold, regularization_weight):
    a = -1/regularization_weight*X.T @ Y_gold @ dual_vars
    return a
```

The objective function is:

```python
obj = -dual_vars.sum(axis=0)-1/(2*regularization_weight)*( dual_vars.T @ Y_gold @
    X @ X.T @ Y_gold @ dual_vars)
```

And the gradient :

```python
grad = -1- (1/regularization_weight)*(Y_gold.T @ X @ X.T @ Y_gold @dual_vars)
```

And the projection :

```python
dual_vars = np.clip(dual_vars + step_size * grad, -1, 0)
```

We get the following results for the second algorithm :

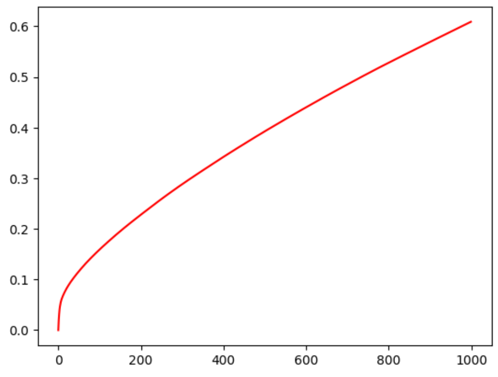
Figure 3: Objective function per epoch

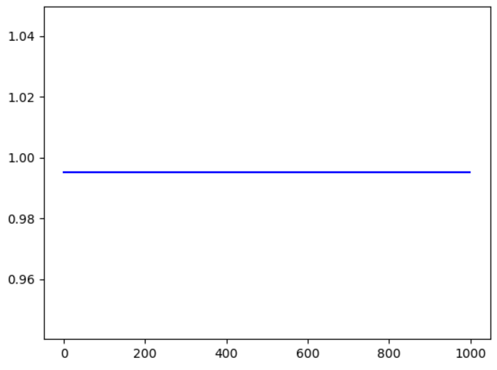
Figure 4: Accuracy plot per epoch

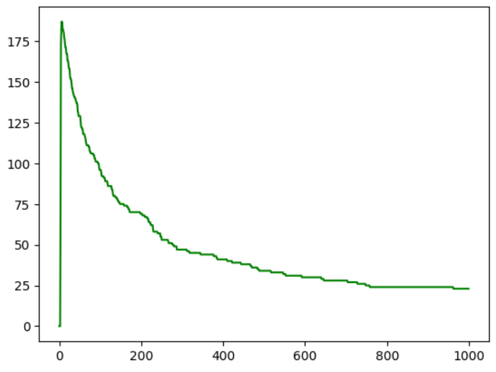
Figure 5: Support vectors per epoch

3. Box constrained coordinate ascent on the dual Box constrained coordinate ascent on the dual is a simple and computationally efficient optimization algorithm used to solve the dual form of the support vector machine (SVM) problem. It updates the solution iteratively by optimizing the objective function with respect to one variable at a time, while keeping the other variables fixed, subject to the constraints imposed by the feasible region.

It's the same objective as previously:

$$\max_{\lambda} - \sum_i \lambda_i - \frac{1}{2c}\lambda^\top Y X X^\top Y \lambda$$
$$S.T \quad -1 \leq \lambda_i \leq 0 \quad \forall i, n \geq i \geq 1$$

Here we don't compute the gradient but the partial derivative for a coordinate K of λ, to achieve it we rewrite the objective for

$$f(\lambda) = b^\top \lambda + \frac{1}{2c}\lambda^\top Q \lambda$$

$$\begin{aligned}
&\text{with } b = \begin{pmatrix} -1 \\ . \\ . \\ . \\ -1 \end{pmatrix}, Q = (-YXX^\top Y) \\
&= \sum_i b_i \lambda_i + \frac{1}{2c} \sum_i \lambda_i \sum_j \lambda_j Q_{j,i} \\
&= \sum_i b_i \lambda_i + \frac{1}{2c} \sum_i \sum_j \lambda_i \lambda_j Q_{j,i} \\
&= \sum_i b_i \lambda_i + \frac{1}{2c} \sum_{i \neq j} \lambda_i \lambda_j Q_{j,i} + \frac{1}{2c} \sum_i \lambda_i^2 Q_{i,i}
\end{aligned}$$

the partial derivative is:

$$\begin{aligned}
\frac{\partial}{\partial \lambda_k} f(\lambda) &= b_k + \frac{1}{2c} \sum_{i \neq k} 2\left(\lambda_i Q_{k,i}\right) + \lambda_k Q_{k,k} \\
&= b_k + \frac{1}{c} \sum_{i \neq k} \lambda_i Q_{k,i} + \lambda_k Q_{k,k}
\end{aligned}$$

We solve partial derivative equals zero:

$$\begin{aligned}
b_k + \frac{1}{c} \sum_{i \neq k} \lambda_i Q_{k,i} + \lambda_k Q_{k,k} &= 0 \\
\lambda_k &= \frac{-b_k - \frac{1}{c} \sum_{i \neq k} \lambda_i Q_{k,i}}{Q_{k,k}}
\end{aligned}$$

But b_k will always be - 1 we can replace it:

$$\lambda_k = \frac{1 - \frac{1}{c} \sum_{i \neq k} \lambda_i Q_{k,i}}{Q_{k,k}}$$

The update rule is then:

$$\lambda_k = \text{clip}_{[-1,0]} \left(\frac{1 - \frac{1}{c} \sum_{i \neq k} \lambda_i Q_{k,i}}{Q_{k,k}} \right)$$

We clip because we still have the constraint on λ !

For the implementation details:

Since the sum doesn't include the updated dual var coordinate, we need a mask $= \begin{pmatrix} \text{true} \\ . \\ . \\ . \\ \text{false} \\ . \\ . \\ . \\ . \\ \text{true} \end{pmatrix}$

false only at coordinate k.

So now we can compute the update:

```
mask = np.ones(X.shape[0], dtype=bool)
        mask[k] = False
        dual_vars[k] = np.clip((1-(1/regularization_weight)*dual_vars[mask].T @ Q[
    k,mask])/(Q[k,k]), -1, 0)
```

We can plot the graphs showing the objective function, the accuracy and the number of support vectors again :

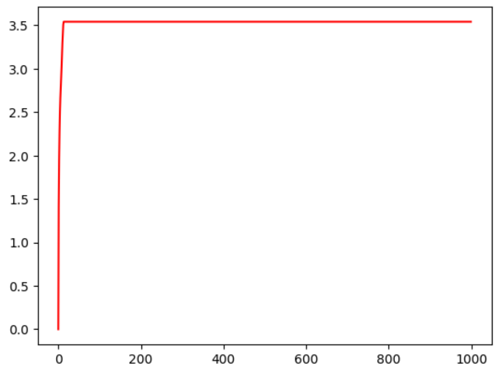

Figure 6: Objective function per epoch

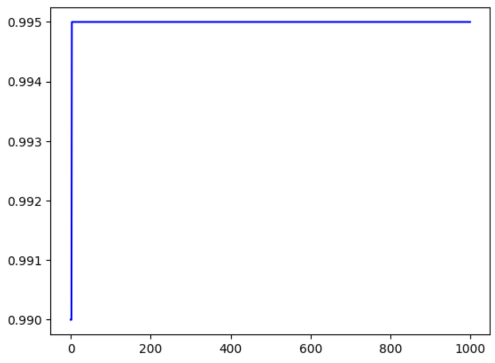

Figure 7: Accuracy plot per epoch

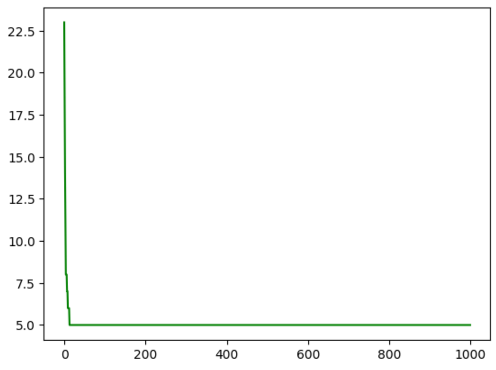

Figure 8: Support vectors per epoch

3 Experiments comparison

We can now compare the results of the different algorithms. We get an accuracy of 0.98 for the sub-gradient descent and 0.995 for the projected gradient ascent and the box constrained coordinate ascent. So we can say that the last two algorithms look more promising. We can also see that the algorithms 2 and 3 converge very fast which is a good point. We reach the optimal accuracy in less than 10 epoch whereas the first algorithm needs 700 epochs for that. Finally, the third algorithm doesn't use a step size which is more practical because that would be another hyperparameter to tune. So overall, the box constrained coordinate ascent seems to be the best algorithm in our case.