2nd Homework Assignment
Project on Support Vector Machines

Vasileios Papageorgiou

June 22, 2024

1 Introduction

The purpose of this project is to implement John Platt's Sequential Minimal Optimization Algorithm (SMO) to train a Support Vector Machine (SVM) for binary classification. Support Vector Machines are supervised learning models used for classification and regression. Their basic principle involves finding the hyperplane that best separates data points of different classes by maximizing the margin between them. This is achieved by solving an optimization problem to identify the support vectors, which are the data points closest to the hyperplane.

The fundamental principle of SMO is that it tries to solve the dual rather than the primal optimization problem. In this project, we will implement a basic version of the SMO algorithm to train an SVM binary classifier using the methodology from Platt's original paper [1]. We will explain each step of the process, providing the analytical background along with code snippets.

2 Problem Formulation

Given a dataset with m training examples and n features, where the i-th example is represented as $(x^{(i)}, y^{(i)})$ with $y^{(i)} \in \{-1, +1\}$ as the label, we aim to address the linear separation problem, potentially involving outliers. The primal problem involves finding a vector $\mathbf{w} = (w_1, w_2, \dots, w_n)$ and a scalar b under the following constraints:

$$\begin{aligned} \min \quad & \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m} s_i \\ \text{s.t.} \quad & y^{(i)}(\mathbf{w}^T x^{(i)} + b) \geq 1 - s_i, \quad i = 1, \dots, m \\ & s_i \geq 0, \quad i = 1, \dots, m \end{aligned}$$

Here, C is the regularization parameter that penalizes the outliers. We allow training samples to have a margin less than 1 and we pay the cost of Cs_i. The parameter controls the trade-off between maximizing the margin and minimizing the margin violations, balancing a wider margin with a smaller number of margin failures.

Using Lagrange duality we can derive the dual problem, corresponding to the primal one, which is formulated as follows:

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \\ \text{s.t.} \quad & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \\ & 0 \leq \alpha_i \leq C, \quad i = 1, \dots, m \end{aligned}$$

We note that the slack variables s_i do not appear in the dual formulation. The optimization problem is now converted into a dual quadratic problem, where the objective function is solely dependent on a set of Lagrange multipliers α_i, under the box constraint $0 \leq \alpha_i \leq C$ and one linear equality constraint $\sum_{i=1}^{m} \alpha_i y^{(i)} = 0$.

The Karush-Kuhn-Tucker (KKT) conditions for the dual problem are for all i:

$$\alpha_i = 0 \iff y_i u_i \geq 1,$$

$$0 < \alpha_i < C \iff y_i u_i = 1,$$

$$\alpha_i = C \iff y_i u_i \leq 1.$$

where u_i is the output of the SVM for the i-th training example.
The KKT conditions reveal crucial insights for SVM training: Support vectors are points where $0 < \alpha_i < C$, situated on the margin's edge. These vectors play a critical role in defining the decision boundary. When $\alpha_i = 0$, the point lies outside the margin and has no impact on the prediction, serving as non-support. Points with $\alpha_i = C$ are inside the margin and may or may not be classified correctly, influencing the SVM's decision boundary.
We anticipate that only a few points will be support vectors (non-bound points), significantly reducing the amount of calculations required.
There is a one-to-one relationship between each Lagrange multiplier α_i and each training example. Once the Lagrange multipliers are determined, the normal vector w can be derived from them:

$$\mathbf{w} = \sum_{i=1}^{N} y_i \alpha_i \mathbf{x}_i$$

.

3 SMO Algorithm

The Sequential Minimal Optimization (SMO) algorithm tackles the SVM training process by focusing on solving the smallest possible optimization problems step by step. Because of the linear constraint, in each step we must always select a pair of Lagrangian multipliers α_i and jointly optimize them. What makes SMO efficient is its ability to solve these pairs analytically, avoiding complex numerical optimizations. The algorithm is built on two main parts: a method for solving these pairs analytically and a heuristic rule for deciding which pairs to optimize next.
In our implementation we have followed the exact steps from Platt's paper. We choose to follow an object oriented approach, creating a class named SVM_classifier that includes the methods *take_step*, or The pair optimization step, *examine_example*, that uses heuristics to identify an eligible pair, and *fit* method that corresponds to the main training loop. The theoretical background and the implementation of those methods will be explained in detail below.

(a) SMO implementation Details

(a).1 Initial setup

Firstly, we initialize the classifier class setting the default parameters, required by the algorithm.

```
class SVM_classifier:
    def __init__(self, X, y, kernel:str =’linear’,
                 C:float =1, epsilon:float =1e-8,
                 tol:float = 0.001, max_iter:int= 500):
        self.X = X
        self.y = y
        self.kernel = kernel
        self.kernel_func = self.select_kernel(self.kernel)
        self.C = C
        self.epsilon = epsilon # error margin
        self.tol = tol # tolerance for KKT
        self.max_iter = max_iter
        self.m, self.n = np.shape(self.X)
        self.alphas = np.zeros(self.m)
        self.Error_cache = np.zeros(self.m)

        self.w = np.zeros(self.n)
        self.b = 0
```

X and y denote the training samples and the corresponding labels, and C is the regularization parameter. We also set a small error variable ϵ, as well as a tolerance within the KKT conditions should be satisfied. In the 'alphas' array we store the Lagrange multipliers. Here are initialized with zeros. The output of the training process will be the vector w and the threshold b.
A kernel function that measures the similarity or distance between the input and the training vectors is also used. The default is *linear*, which suffices for this project, but we could use also others. Apart from the linear we have also implement an RBF Gaussian kernel:

```
def linear_kernel(self, x1: np.ndarray, x2: np.ndarray) -> np.ndarray:
        return x1 @ x2.T

def rbf_kernel(self, x1, x2):

    gamma = 1
    if np.ndim(x1) == 1: x1 = x1[np.newaxis, :]
    if np.ndim(x2) == 1: x2 = x2[np.newaxis, :]

    dist_squared = np.linalg.norm(x1[:, :, np.newaxis] - \
                   x2.T[np.newaxis, :, :], axis=1) ** 2
    dist_squared = np.squeeze(dist_squared)

    return np.exp(-gamma * dist_squared)
```

(a).2 *take_step* method

The method takes as input two points i_1, i_2 and takes an optimization step. If that is successful the corresponding parameter arrays are updated else nothing happens and we try another pair.
In order to solve for the two Lagrange multipliers, SMO first computes the constraints on these multipliers and then solves for the constrained minimum. The inequality constraints cause the Lagrange multipliers to lie within a $[0, C]$ box, while the linear equality constraint causes the Lagrange multipliers to lie on a diagonal line.
The latter constraint explains why two is the minimum number of Lagrange multipliers that can be optimized together; if SMO optimized only one multiplier, it could not fulfill the linear equality constraint at every step.
The algorithm first computes the second Lagrange multiplier α_2 and computes the ends of the diagonal line segment in terms of it. The ends of the diagonal line segment can be expressed quite simply, as shown below:

$$L = \begin{cases} \max(0, \alpha_2 - \alpha_1), & \text{if } y_1 \neq y_2 \\ \max(0, \alpha_2 + \alpha_1 - C), & \text{if } y_1 = y_2 \end{cases} \qquad H = \begin{cases} \min(C, C + \alpha_2 - \alpha_1), & \text{if } y_1 \neq y_2 \\ \min(C, \alpha_2 + \alpha_1), & \text{if } y_1 = y_2 \end{cases}$$

Those serve as bounds for α_2. Within the *take_step* method this is implemented as follows:

```
if y1!=y2:
    L = max(0,a2-a1)
    H = min(self.C,self.C+a2-a1)
else:
    L = max(0,a2+a1-self.C)
    H = min(self.C,a2+a1)

if L==H:
    return 0
```

If both ends are the same, we exit the method and continue with a different pair.
The second derivative of the objective function along the diagonal line is given by:

$$\eta = K(x_1, x_1) + K(x_2, x_2) - 2K(x_1, x_2).$$

Under normal circumstances, where the objective function is **positive definite**, there exists a minimum along the direction of the linear equality constraint, and $\eta > 0$. In such cases, SMO computes the minimum along this constraint direction as:

$$\alpha_{2,\text{new}} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta},$$

where $E_i = u_i - y_i$ represents the error on the i-th training example and we save it in an *Error_cache* array.
The constrained minimum is then determined by clipping the unconstrained minimum to the boundaries of the line segment:

$$\alpha_{2,\text{new,clipped}} = \begin{cases} H, & \text{if } \alpha_{2,\text{new}} \geq H \\ \alpha_{2,\text{new}}, & \text{if } L < \alpha_{2,\text{new}} < H \\ L, & \text{if } \alpha_{2,\text{new}} \leq L. \end{cases}$$

The corresponding code snippet is as follows:

```
k11 = self.kernel_func(x1,x1)
k22 = self.kernel_func(x2,x2)
k12 = self.kernel_func(x1,x2)

# Compute the second derivative along diagonal
eta = k11 + k22 - 2.0*k12

if eta > 0:

    a2_new = a2 + y2*(E1-E2)/eta
    if a2_new>=H:
        a2_new = H
    if a2_new<=L:
        a2_new = L
```

Under unusual circumstances, η will not be positive. In such cases, the objective function Ψ should be evaluated at each end of the line segment by the following equations:

$$
\begin{aligned}
f_1 &= y_1(E_1 + b) - \alpha_1 K(x_1, x_1) - s\alpha_2 K(x_1, x_2),\\
f_2 &= y_2(E_2 + b) - s\alpha_1 K(x_1, x_2) - \alpha_2 K(x_2, x_2),\\
L_1 &= \alpha_1 + s(\alpha_2 - L),\\
H_1 &= \alpha_1 + s(\alpha_2 - H),\\
\Psi_L &= L1 f_1 + L f_2 + \frac{1}{2} L^2 K(x_1, x_1) + \frac{1}{2} L^2 K(x_2, x_2) + sLL_1 K(x_1, x_2),\\
\Psi_H &= H1 f_1 + H f_2 + \frac{1}{2} H^2 K(x_1, x_1) + \frac{1}{2} H^2 K(x_2, x_2) + sHH_1 K(x_1, x_2).
\end{aligned}
$$

SMO will move the Lagrange multipliers to the endpoint that yields the lowest value of the objective function Ψ. If the objective function is equal at both ends (within a small ϵ for round-off error) the joint minimization cannot make further progress, and the method is exited. Below is the relevant code:

```
f1 = y1*(E1 + self.b) - a1*k11 - s*a2*k12
f2 = y2*(E2 + self.b) - s*a1*k12 - a2*k22
L1 = a1 + s*(a2 - L)
H1 = a1 + s*(a2 - H)
psi_L = L1*f1 + L*f2 + 0.5*L1*L1*k11 + 0.5*L*L*k22 + s*L*L1*k12
psi_H = H1*f1 + H*f2 + 0.5*H1*H1*k11 + 0.5*H*H*k22 + s*H*H1*k12

if psi_L < (psi_H - self.epsilon):
    a2_new = L
elif psi_L > (psi_H + self.epsilon):
    a2_new = H
else:
    a2_new = a2

if np.abs(a2_new - a2) < (self.epsilon * (a2_new + a2 + self.epsilon)):
    return 0
```

Now we can calculate the The value of α_1 based on the new clipped α_2 using the following equation (let $s = y_1 y_2$):

$$\alpha_{1,\text{new}} = \alpha_1 + s(\alpha_2 - \alpha_{2,\text{new,clipped}}).$$

The threshold b is recalculated after each optimization step to satisfy the KKT conditions for both optimized examples. $b1$ ensures the SVM outputs $y1$ when the input is $x1$:

$$b_1 = E_1 + y1(\alpha_{1,\text{new}} - \alpha_1)K(x_1, x_1) + y2(\alpha_{2,\text{new,clipped}} - \alpha_2)K(x_1, x_2) + b.$$

Similarly, b_2 ensures the SVM outputs y_2 when the input is x_2:

$$b_2 = E_2 + y_1(\alpha_{1,\text{new}} - \alpha_1)K(x_1, x_2) + y_2(\alpha_{2,\text{new,clipped}} - \alpha_2)K(x_2, x_2) + b.$$

When both b_1 and b_2 are valid and equal, or when both new Lagrange multipliers are at bounds (and $L \neq H$), any threshold between b_1 and b_2 satisfies the KKT conditions. SMO selects the average between b_1 and b_2.

For linear kernel, we can easily update and store the the weight vector w using the following:

$$w_{\text{new}} = w + y_1(\alpha_{1,\text{new}} - \alpha_1)x_1 + y_2(\alpha_{2,\text{new,clipped}} - \alpha_2)x_2.$$

For all non boundary elements we can also update efficiently the error cache with the help of some algebra and it yields:

$$E_i^{new} = E_i^{old} + y_1(\alpha_1^{new} - \alpha_1^{old})K_{1i} + y_2(\alpha_2^{new} - \alpha_2^{old})K_{2i} + (b_{new} - b)$$

The method exits successful and all the relevant arrays are updated with their new values:

```
# Update threshold b
b1 = self.b + E1 + y1*(a1_new - a1)*k11 + y2*(a2_new - a2)*k12
b2 = self.b + E2 + y1*(a1_new - a1)*k12 + y2*(a2_new - a2)*k22

if 0 < a1_new < self.C:
    b_new = b1
elif 0 < a2_new < self.C:
    b_new = b2
else:
    b_new = 0.5*(b1 + b2)

# Update weight’s vector if Linear kernel
if self.kernel == ’linear’:
    self.w = self.w + y1*(a1_new - a1)*x1 + y2*(a2_new - a2)*x2

# Update Error_cache using alphas (see reference)

# if a1 & a2 are not at bounds, the error will be 0
self.Error_cache[i1] = 0
self.Error_cache[i2] = 0

# Update error for non boundary elements
inner_indices = [idx for idx, a in enumerate(self.alphas) if 0 < a < self.C]
for i in inner_indices:
    self.Error_cache[i] += \
    y1*(a1_new - a1)*self.kernel_func(x1, self.X[i,:]) \
    + y2*(a2_new - a2)*self.kernel_func(x2, self.X[i,:]) \
    + (self.b - b_new)

# Update alphas
    self.alphas[i1] = a1_new
    self.alphas[i2] = a2_new

# Update b
    self.b = b_new

return 1
```

(a).3 *examine_example* method

4 Experimental Setup

- Description of the dataset used (gisette dataset).
- Preprocessing steps and train-test split (using 4500-5000 points for training).

5 Results

- Presentation of the results obtained from the basic implementation.
- Evaluation metrics used for assessing the performance.

6 Optimization and Fine-Tuning

- Strategies for optimizing the choice of the constant C.
- Any other optimizations or improvements made to the algorithm.
- Comparative results before and after optimization.

7 Discussion

- Analysis of the results and their implications.
- Challenges faced during implementation and how they were addressed.
- Limitations of the current implementation and potential future work.

8 Conclusion

- Summary of the work done.
- Key findings and their significance.
- Final thoughts and potential directions for future research.

A Appendix

- Additional tables, figures, or code snippets.
- Detailed mathematical derivations if necessary.

Theoretical Background

We have the following non linear program:

$$\min\{F(x) = \frac{c^T x}{d^T x} : Ax = b;\ x \geq 0\} \tag{1}$$

Algorithm 1 Bisection Method for Optimal α

```
Given: interval $[L, U]$ that contains optimal $\alpha$
repeat
    $\alpha := \frac{u+l}{2}$
    Solve the feasibility problem:
      $c^T x \leq \alpha d^T x$
      $d^T x > 0$
      $Ax = b$
    Adjust the bounds
    if feasible then
        $U := \alpha$
    else
        $L := \alpha$
    end if
until $U - L \leq \epsilon$
```

```
# Define parameter s
s = y1 * y2

# Compute L, H via equations (13) and (14) from Platt
if y1 != y2:
L = max(0, a2 - a1)
H = min(self.C, self.C + a2 - a1)
```

```
else:
L = max(0, a2 + a1 - self.C)
H = min(self.C, a2 + a1)

if L == H:
return 0
```

Problem 4

(a) Updating the Error Cache

When a Lagrange multiplier is non-bound after being optimized, its cached error is zero. The stored errors of other non-bound multipliers not involved in joint optimization are updated as follows.

$$E_k^{\text{new}} = E_k^{\text{old}} + u_k^{\text{new}} - u_k^{\text{old}} \tag{3.36}$$

$$E_k^{\text{new}} = E_k^{\text{old}} + u_k^{\text{new}} - u_k^{\text{old}} \tag{3.37}$$

For any k-th example in the training set, the difference between its new SVM output value and its old SVM output value, $u_k^{\text{new}} - u_k^{\text{old}}$, is due to the change in α_1, α_2 and the change in the threshold b.

$$u_k^{\text{new}} - u_k^{\text{old}} = y_1 \alpha_1^{\text{new}} k_{1k} + y_2 \alpha_2^{\text{new}} k_{2k} - b^{\text{new}} - \left(y_1 \alpha_1^{\text{old}} k_{1k} + y_2 \alpha_2^{\text{old}} k_{2k} - b^{\text{old}} \right) \tag{3.38}$$

Substituting equation (3.37) into equation (3.36), we have

$$E_k^{\text{new}} = E_k^{\text{old}} + y_1 \left(\alpha_1^{\text{new}} - \alpha_1^{\text{old}} \right) k_{1k} + y_2 \left(\alpha_2^{\text{new}} - \alpha_2^{\text{old}} \right) k_{2k} - \left(b^{\text{new}} - b^{\text{old}} \right) \tag{3.39}$$

References

[1] John Platt. Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. Technical Report MSR-TR-98-14, Microsoft, April 1998. https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/.

[2] Ginny Mak. The Implementation of Support Vector Machines Using the Sequential Minimal Optimization Algorithm. Master's thesis, McGill University, School of Computer Science, Montreal, Canada, April 2000.