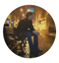
Yet more simple SMO algorithm

Research · October 2020

DOI: 10.13140/RG.2.2.36670.10562

Vitalii Bohdanovych Tymchyshyn
National Academy of Sciences of Ukraine



Andrii Khlevniuk

Yet more simple SMO algorithm

V. B. Tymchyshyn[*,1] and A. V. Khlevniuk

[1] Bogolyubov Institute for Theoretical Physics, National Academy of Sciences, Metrolohichna St. 14-b, Kyiv 03680, Ukraine

October 3, 2020

Basically, training of SVM reduces to solving the dual problem

$$\text{maximize: } \mathscr{L}^* = \sum_i \lambda_i - \frac{1}{2}\sum_{i,j} \lambda_i \lambda_j y_i y_j K(\boldsymbol{x}_i; \boldsymbol{x}_j), \tag{1a}$$

$$\text{with constraints: } 0 \le \lambda_i \le C, \tag{1b}$$

$$\sum_i \lambda_i y_i = 0, \tag{1c}$$

where x_i are datapoints and y_i are their respective classes (± 1), λ_i are dual variables and K is the kernel function (positive and symmetric). Classification is than performed as

$$\text{class} = \operatorname{sign}(f(\boldsymbol{x})), \quad f(\boldsymbol{x}) = \sum_i \lambda_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}) + b, \tag{2}$$

where f is the decision function and b— bias term. The latter cannot be calculated from $\mathscr{L}^*$ minimization thus is obtained, for example, as a mean error that appears when we classify support vectors [1]

$$b = \mathbb{E}_k \left[y_k - \sum_i \lambda_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}_k) \middle| \lambda_k > 0 \right]. \tag{3}$$

A highly efficient method to perform such minimization is the SMO-algorithm [2,3]. In essence, SMO is a modification of coordinate descent— we freeze all λ-s except two, λ_I and λ_J, then perform minimization with respect to these two λ-s only. The latter can be done with certain formulas derived analytically and then a new iteration starts. Additionally, original SMO-algorithm contains many complicated heuristics to choose λ_I and λ_J, as well as convergence control by checking KKT conditions.

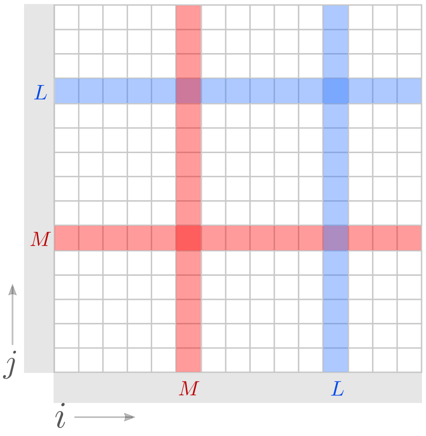
The problem is, if we want to present SMO-implementation to students as an in-class problem, we need to simplify it a decent amount. Discarding heuristics as in [4] makes SMO much simpler but still complicated. Even if we perform fixed number of iterations instead of KKT checking, it is still hardly implementable under 30 minutes. Moreover, full derivation of formulas [5] is painful.

We suggest that the root of complexity in [2–5] is the representation $\lambda_J(\lambda_I) = k\lambda_I + c$ (parameters k and c are such that constraints (1b) and (1c) hold), while this is hardly the best representation of the straight line and makes formulas unwieldy. As an alternative we propose to use vector representation of the line. Extensive use of linear algebra operations from `numpy` makes code even simpler.

To get the vector form we are interested in, let's do the following. Define a matrix

$$\boldsymbol{K} = \begin{pmatrix} y_1 y_1 K(\boldsymbol{x}_1; \boldsymbol{x}_1) & y_1 y_2 K(\boldsymbol{x}_1; \boldsymbol{x}_2) & \dots & y_1 y_N K(\boldsymbol{x}_1; \boldsymbol{x}_N) \\ y_2 y_1 K(\boldsymbol{x}_2; \boldsymbol{x}_1) & y_2 y_2 K(\boldsymbol{x}_2; \boldsymbol{x}_2) & \dots & y_2 y_N K(\boldsymbol{x}_2; \boldsymbol{x}_N) \\ \dots & \dots & \dots & \dots \\ y_N y_1 K(\boldsymbol{x}_N; \boldsymbol{x}_1) & y_N y_2 K(\boldsymbol{x}_N; \boldsymbol{x}_2) & \dots & y_N y_N K(\boldsymbol{x}_N; \boldsymbol{x}_N) \end{pmatrix}. \tag{4}$$

$\boldsymbol{K}$ is symmetric due to kernel function K being symmetric.



Let's get to minimization. First, we remove from $\mathscr{L}^*$ (1) all summands independent on both λ_M and λ_L (indexes of elements removed from $\sum_{i,j} \lambda_i \lambda_j y_i y_j K(\boldsymbol{x}_i; \boldsymbol{x}_j)$ are shown as white squares in the figure). Then we rearrange other terms to get full summation over i and j (see red and blue "stripes" in figure), the latter three terms compensate for double counting (intersections of "stripes" in figure)

$$\begin{aligned} \bar{\mathscr{L}}^* &= \lambda_M + \lambda_L - \sum_j \lambda_M \lambda_j K_{M,j} - \sum_i \lambda_L \lambda_i K_{L,i} + \\ \text{compensate for double-counting} \rightarrow \quad &+ \frac{1}{2}{\lambda_M}^2 K_{M,M} + \lambda_M \lambda_L K_{M,L} + \frac{1}{2}{\lambda_L}^2 K_{L,L} = \\ &= \lambda_M \left(1 - \sum_j \lambda_j K_{M,j}\right) + \lambda_L \left(1 - \sum_i \lambda_i K_{L,i}\right) + \\ &+ \frac{1}{2}\left({\lambda_M}^2 K_{M,M} + 2\lambda_M \lambda_L K_{M,L} + {\lambda_L}^2 K_{L,L}\right) = \\ &= \boldsymbol{k}_0^T \boldsymbol{v}_0 + \frac{1}{2}\boldsymbol{v}_0^T \boldsymbol{Q}\, \boldsymbol{v}_0, \end{aligned}$$

[*] corresponding author, yu.binkukoku@gmail.com

where

$$\boldsymbol{v}_0 = (\lambda_M, \lambda_L)^T, \tag{5a}$$

$$\boldsymbol{k}_0 = \left(1 - \boldsymbol{\lambda}^T \boldsymbol{K}_M, 1 - \boldsymbol{\lambda}^T \boldsymbol{K}_L\right)^T, \tag{5b}$$

$$\boldsymbol{Q} = \begin{pmatrix} K_{M,M} & K_{M,L} \\ K_{L,M} & K_{L,L} \end{pmatrix}. \tag{5c}$$

Note, that $\boldsymbol{k}_0$ still depends on λ_M and λ_L. To localize this dependence we rewrite $\boldsymbol{k}_0$ as

$$\boldsymbol{k}_0 = \begin{pmatrix} 1 - \lambda_M K_{M,M} - \lambda_L K_{M,L} - \sum_{i \neq M,L} \lambda_i K_{M,i} \\ 1 - \lambda_M K_{L,M} - \lambda_L K_{L,L} - \sum_{i \neq M,L} \lambda_i K_{L,i} \end{pmatrix} = \begin{pmatrix} 1 - \sum_{i \neq M,L} \lambda_i K_{M,i} \\ 1 - \sum_{i \neq M,L} \lambda_i K_{L,i} \end{pmatrix} - \boldsymbol{Q}\boldsymbol{v}_0,$$

Now, following the SMO-algorithm idea, we freeze all λ-s except λ_M and λ_L, then minimize $\bar{\mathscr{L}}^*$ with respect to this two lambdas so that constraints (1c) and (1b) are still satisfied. Let $\boldsymbol{v}$ be the following function of scalar variable t

$$\boldsymbol{v}(t) = \boldsymbol{v}_0 + t\boldsymbol{u}, \tag{6a}$$

$$\boldsymbol{u} = (-y_L, y_M)^T. \tag{6b}$$

We change $\boldsymbol{v}_0 \to \boldsymbol{v}(t)$ everywhere in $\bar{\mathscr{L}}^*$ and perform minimization over t. Note that $\boldsymbol{k}$ also depends on t

$$\boldsymbol{k}(t) = \boldsymbol{k}_0 - t\,\boldsymbol{Q}\boldsymbol{u}.$$

If components of $\boldsymbol{v}_0$, namely λ_M and λ_L, satisfied (1c), components of $\boldsymbol{v}(t)$ (i.e. $\lambda_M(t)$ and $\lambda_L(t)$) satisfy (1c) as well

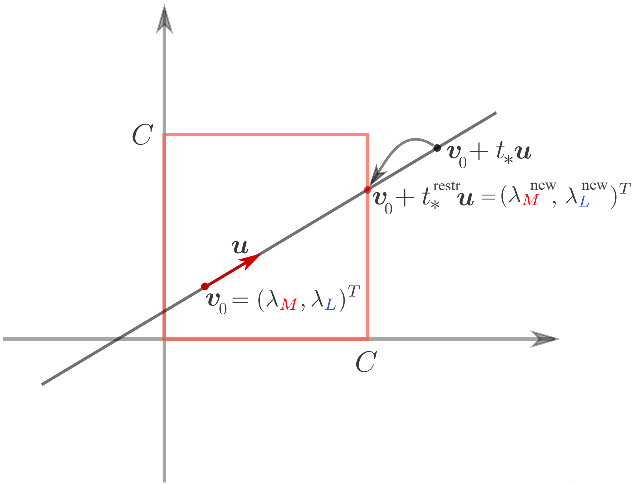
$$y_M \lambda_M(t) + y_L \lambda_L(t) + \sum_{i \neq M,L} y_i \lambda_i = y_M(\lambda_M - \cancel{t y_L}) + y_L(\lambda_L + \cancel{t y_M}) + \sum_{i \neq M,L} y_i \lambda_i = \sum_i y_i \lambda_i = 0.$$

Derivative of the Lagrangian with respect to t reads (use eq.81 from [6] and $Q^T = Q$ property)

$$\frac{d\bar{\mathscr{L}}^*(t)}{dt} = \frac{d\boldsymbol{k}^T}{dt}\boldsymbol{v} + \boldsymbol{k}^T \frac{d\boldsymbol{v}}{dt} + \frac{1}{2}\left(\frac{d(\boldsymbol{v}^T \boldsymbol{Q}\, \boldsymbol{v})}{d\boldsymbol{v}}\right)^T \frac{d\boldsymbol{v}}{dt} = \cancel{-\boldsymbol{u}^T \boldsymbol{Q}\boldsymbol{v}} + \boldsymbol{k}^T \boldsymbol{u} + \cancel{\boldsymbol{v}^T \boldsymbol{Q}\boldsymbol{u}} = \boldsymbol{k}^T \boldsymbol{u}.$$

The latter cancelation uses that $\boldsymbol{u}^T\boldsymbol{Q}\boldsymbol{v}$ and $\boldsymbol{v}^T\boldsymbol{Q}\boldsymbol{u}$ are scalars. Extremum condition leads to

$$\frac{d\bar{\mathscr{L}}^*(t)}{dt} = \boldsymbol{k}^T \boldsymbol{u} = (\boldsymbol{k}_0 - \boldsymbol{Q}\boldsymbol{u})^T \boldsymbol{u} = \boldsymbol{k}_0^T \boldsymbol{u} - t\boldsymbol{u}^T \boldsymbol{Q}\boldsymbol{u} = 0.$$


C
$\boldsymbol{v}_0 + t_* \boldsymbol{u}$
$\boldsymbol{v}_0 + t_*^{\text{restr}} \boldsymbol{u} = (\lambda_M^{\text{new}}, \lambda_L^{\text{new}})^T$
$\boldsymbol{u}$
$\boldsymbol{v}_0 = (\lambda_M, \lambda_L)^T$
C

Since Q is positive semidefinite[1] $\boldsymbol{u}^T\boldsymbol{Q}\boldsymbol{u} \geq 0$ and $\text{argmax}_t\, \bar{\mathscr{L}}^*(t)$ can be written as

$$t_* = \frac{\boldsymbol{k}_0^T \boldsymbol{u}}{\boldsymbol{u}^T \boldsymbol{Q}\boldsymbol{u}}. \tag{7}$$

If we calculate ${\lambda_M}^{\text{new}}$ or ${\lambda_L}^{\text{new}}$ simply using t_*, they may violate (1b). We should restrict lambdas to square $[0, C] \times [0, C]$ so that they remain on the line $\boldsymbol{v}+t\boldsymbol{u}$. This is a simple geometric problem as shown in the figure and algorithmically you can solve it in many ways. Anyway, after restriction we obtain new lambdas

$$({\lambda_M}^{\text{new}}, {\lambda_L}^{\text{new}})^T = \boldsymbol{v}_0 + t_*^{\text{restr}} \boldsymbol{u}. \tag{8}$$

Algorithm implementation as described above with tests can be found on GitHub [7], check out The Algorithm with References to Equations for vanilla SMO with references to formulas of this section or take a look at Algorithm + Visual Comparison to sklearn.svm for the full code. Also check out Appendix: Visual Comparison to sklearn.svm for visuals.

References

[1] `https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2004-11.pdf`

[2] Platt, John. *Fast Training of Support Vector Machines using Sequential Minimal Optimization*, in Advances in Kernel Methods Support Vector Learning, B. Scholkopf, C. Burges, A. Smola, eds., MIT Press (1998).

[3] `http://web.cs.iastate.edu/~honavar/smo-svm.pdf`

[4] `http://cs229.stanford.edu/materials/smo.pdf`

[5] `http://fourier.eng.hmc.edu/e176/lectures/ch9/node9.html`

[6] `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`

[7] `https://github.com/fbeilstein/simplest_smo_ever`

[1] To prove use that kernel function K is symmetric positive-semidefinite and classes y_i are ± 1.

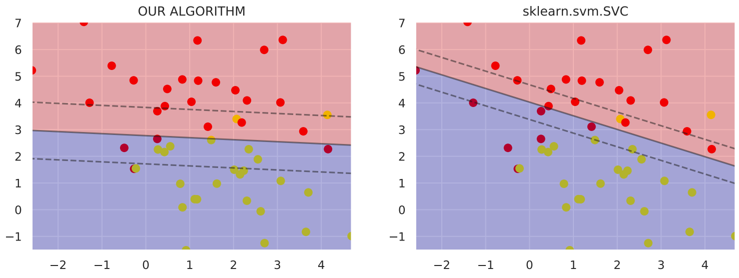
Appendix: The Algorithm with References to Equations

```python
class SVM:
  def __init__(self, kernel='linear', C=10000.0, max_iter=100000, degree=3, gamma=1):
    self.kernel = {'poly'  : lambda x,y: np.dot(x, y.T)**degree,
                   'rbf'   : lambda x,y: np.exp(-gamma*np.sum((y - x[:,np.newaxis])**2, axis=-1)),
                   'linear': lambda x,y: np.dot(x, y.T)}[kernel]
    self.C = C
    self.max_iter = max_iter

  def restrict_to_square(self, t, v0, u):
    t = (np.clip(v0 + t*u, 0, self.C) - v0)[1]/u[1]
    return (np.clip(v0 + t*u, 0, self.C) - v0)[0]/u[0]

  def fit(self, X, y):
    self.X = X.copy()   # store for decision function
    self.y = y * 2 - 1  # convert classes 0 and 1 to -1 and +1, store
    self.lambdas = np.zeros_like(self.y, dtype=float) # dual variables, all zeros satisfy eq.(1b)
    self.K = self.kernel(self.X, self.X) * self.y[:,np.newaxis] * self.y # eq.(4)

    for _ in range(self.max_iter):
      for idxM in range(len(self.lambdas)):            # iterate all lambda_M
        idxL = np.random.randint(0, len(self.lambdas)) # choose randomly lambda_L
        Q = self.K[[[idxM, idxM], [idxL, idxL]], [[idxM, idxL], [idxM, idxL]]]           # eq.(5c)
        v0 = self.lambdas[[idxM, idxL]]                                                  # eq.(5a)
        k0 = 1 - np.sum(self.lambdas * self.K[[idxM, idxL]], axis=1)                     # eq.(5b)
        u = np.array([-self.y[idxL], self.y[idxM]])                                      # eq.(6b)
        t_max = np.dot(k0, u) / (np.dot(np.dot(Q, u), u) + 1E-15) # eq.(7), +1E-15 if idxM == idxL
        self.lambdas[[idxM, idxL]] = v0 + u * self.restrict_to_square(t_max, v0, u) # eq.(8)

    idx, = np.nonzero(self.lambdas > 1E-15) # select indexes of support vectors
    self.b = np.sum((1.0 - np.sum(self.K[idx]*self.lambdas, axis=1))*self.y[idx])/len(idx) # eq.(3)

  def decision_function(self, X):
    return np.sum(self.kernel(X, self.X) * self.y * self.lambdas, axis=1) + self.b # f from eq.(2)
```

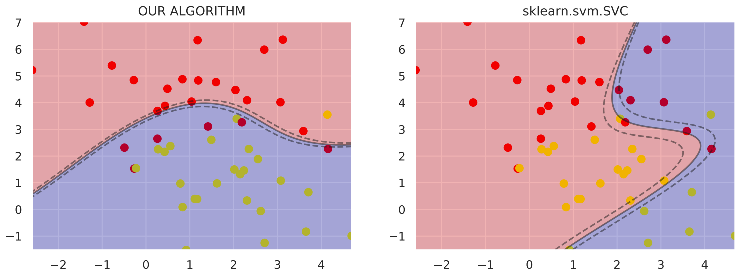
Appendix: Visual Comparison to sklearn.svm

Note that both algorithms are stochastic, thus sometimes the algorithm above performs better than sklearn.svm and sometimes worse. For parameters see next section with full listing or our GitHub [7].

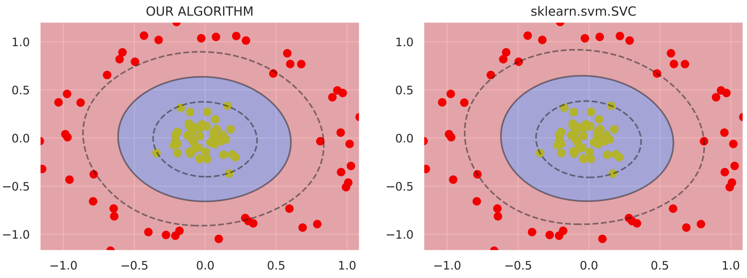
linear kernel



poly kernel



rbf kernel



Appendix: Full Listing

```python
import numpy as np

class SVM:
  def __init__(self, kernel='linear', C=10000.0, max_iter=100000, degree=3, gamma=1):
    self.kernel = {'poly'  : lambda x,y: np.dot(x, y.T)**degree,
                   'rbf'   : lambda x,y: np.exp(-gamma*np.sum((y - x[:,np.newaxis])**2, axis=-1)),
                   'linear': lambda x,y: np.dot(x, y.T)}[kernel]
    self.C = C
    self.max_iter = max_iter

  def restrict_to_square(self, t, v0, u):
    t = (np.clip(v0 + t*u, 0, self.C) - v0)[1]/u[1]
    return (np.clip(v0 + t*u, 0, self.C) - v0)[0]/u[0]

  def fit(self, X, y):
    self.X = X.copy()
    self.y = y * 2 - 1
    self.lambdas = np.zeros_like(self.y, dtype=float)
    self.K = self.kernel(self.X, self.X) * self.y[:,np.newaxis] * self.y

    for _ in range(self.max_iter):
      for idxM in range(len(self.lambdas)):
        idxL = np.random.randint(0, len(self.lambdas))
        Q = self.K[[[idxM, idxM], [idxL, idxL]], [[idxM, idxL], [idxM, idxL]]]
        v0 = self.lambdas[[idxM, idxL]]
        k0 = 1 - np.sum(self.lambdas * self.K[[idxM, idxL]], axis=1)
        u = np.array([-self.y[idxL], self.y[idxM]])
        t_max = np.dot(k0, u) / (np.dot(np.dot(Q, u), u) + 1E-15)
        self.lambdas[[idxM, idxL]] = v0 + u * self.restrict_to_square(t_max, v0, u)

    idx, = np.nonzero(self.lambdas > 1E-15)
    self.b = np.sum((1.0 - np.sum(self.K[idx] * self.lambdas, axis=1)) * self.y[idx]) / len(idx)

  def decision_function(self, X):
    return np.sum(self.kernel(X, self.X) * self.y * self.lambdas, axis=1) + self.b

######### TESTS ############
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.datasets import make_blobs, make_circles
from matplotlib.colors import ListedColormap

X, y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=1.4)
X, y = make_circles(100, factor=.1, noise=.1)

def test_plot(X, y, svm_model, axes, title):
  plt.axes(axes)
  xlim = [np.min(X[:, 0]), np.max(X[:, 0])]
  ylim = [np.min(X[:, 1]), np.max(X[:, 1])]
  xx, yy = np.meshgrid(np.linspace(*xlim, num=700), np.linspace(*ylim, num=700))
  rgb=np.array([[210, 0, 0], [0, 0, 150]])/255.0

  svm_model.fit(X, y)
  z_model = svm_model.decision_function(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

  plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
  plt.contour(xx, yy, z_model, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
  plt.contourf(xx, yy, np.sign(z_model.reshape(xx.shape)), alpha=0.3, levels=2, cmap=ListedColormap(
    rgb), zorder=1)
  plt.title(title)

fig, axs = plt.subplots(nrows=1,ncols=2,figsize=(12,4))
test_plot(X, y, SVM(kernel='rbf', C=10, max_iter=60, degree=3, gamma=1), axs[0], 'OUR ALGORITHM')
test_plot(X, y, SVC(kernel='rbf', C=10, max_iter=60, degree=3, gamma=1), axs[1], 'sklearn.svm.SVC')
```