Support Vector Machines

Michael

School of Mathematics and Statistics, UCD

School of Economics, University of Nottingham

1 History

The original support vector machines (SVM) algorithm was invented by Vladimir N. Vapnik and Alexey Ya. Chervonenkis in 1963. In 1992, Bernhard E. Boser, Isabelle M. Guyon and Vladimir N. Vapnik suggested a way to create nonlinear classifiers by applying the kernel trick to maximum-margin hyperplanes. It takes almost 30 years for people to realize the power of SVM once they twisted the model from linear to nonlinear with the use of kernels. Today SVMs are among the best (and many believes are indeed the best) "off-the-shelf" supervised learning algorithms. According to Professor Patrick Winston, *SMV needs to be in the tool bag of every civilized person.*

This set of notes are based on the lecture delivered by Professor Patrick Winston in MIT. Here is the LINK. I have to say that Professor Patrick Winston is a great teacher as he illustrates this sophisticated model concisely and elegantly.

2 The Support Vector Classifier

Professor Patrick Winston called SVM the 'the widest street' approach. The idea of SVM can be summarised in the following figure.

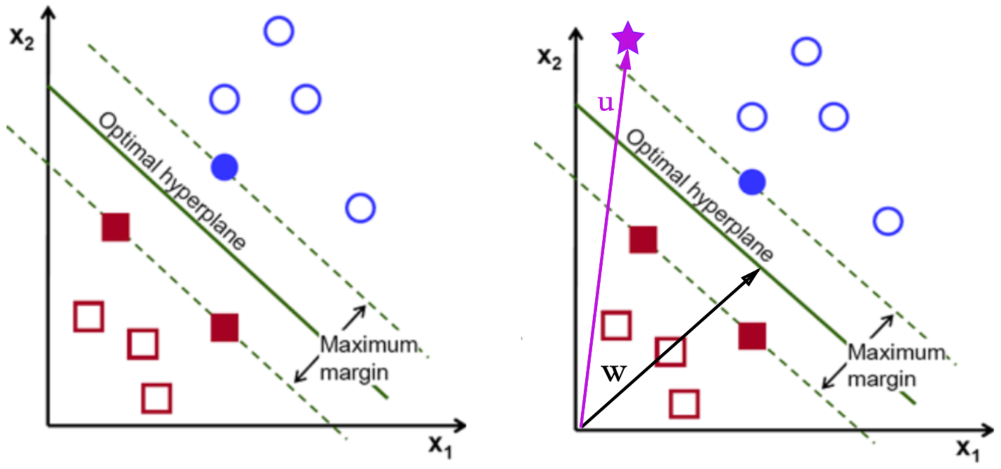

Figure 1: Support vector classifiers

The goal of SVM is to construct an optimal separating hyperplane between two perfectly separated classes. Suppose, our training data consists of N pairs $(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)$, with $x_i \in \mathbb{R}^m$ and $y_i \in \{-1, 1\}$. If y_i is not in the set $\{-1, 1\}$, we can just transfer out dataset. Right now, we focus on the binary classification[1]. As it has shown in Figure 1, we hope to find an optimal hyperplane to separate two classes (square and circle in Figure 1). In Figure 1, we simplify the situation, in which the separation is easy to spot and linear. Later, we will discuss the nonlinear case or the case with the mix of square and circle.

So how do we construct this optimal hyperplane? Like most mathematicians do, we would first assume it exists, and based on its properties to prove its existence. So, let's assume we already have this optimal hyperplane, and we wish to construct a vector w that is orthogonal to this plane. We call this vector w *support vector*[2].

Now, suppose we have a pair (x_u, y_u) represented by a star or the vector u in Figure 1. We don't know whether this star is a square or circle. To classify it, we just project it onto the vector w:

$$Proj_u w = \frac{w \cdot u}{||w||} w$$

Now, we say if $(w \cdot u)\ w \geq c||w||w$, where c is a scalar, then that star should be a circle. So, simplify the math, we have the following decision rule:

Step I

$$\begin{aligned} &w \cdot u \geq c \\ &w \cdot u + b \geq 0 \Leftrightarrow \ y_u = +1 \end{aligned}$$

Now, we further transfer our decision rule to make the model mathematically convenient:

$$\begin{gathered} w \cdot x_+ + b \geq 1 \\ w \cdot x_{-1} + b \leq -1 \end{gathered}$$

Since we have transferred our dataset to have $y_i \in \{-1, 1\}$, then the above decision rule could be further simplified as

$$y_i(w \cdot x_i + b) \geq 1$$

In addition, we add more constraints to our model:

Step II

$$y_i(w \cdot x_i + b) - 1 = 0 \ \ \forall\ x_i \text{ inside the gutter (or maximum margin)}$$

At this stage, it looks like that we got some clues. But we still don't know what kind of w and b we should use to form the decision rule. To find the value of w and b, we first measure the width of 'margin' (or gutter) as it is shown in Figure 2.

[1]You might have learned that one can treat all classification as binary case with 'oneVersusall' approach.

[2]The points that help determine the hyperplane are called "support vectors".

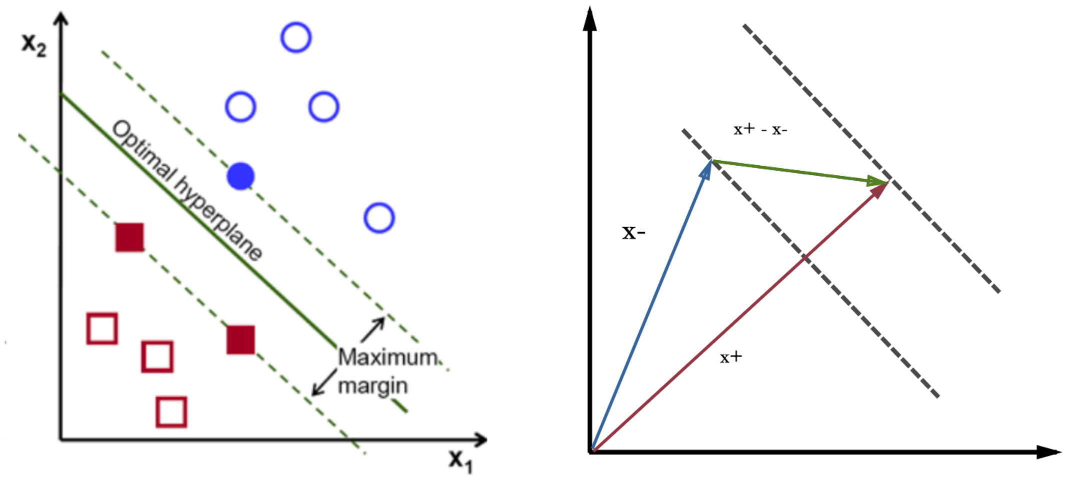

Figure 2: Measure the width: projection again

Once we have $x_+ - x_-$, then we do the projection again to measure the width as we know w is a normal vector. Then, the width of maximum margin should be

$$\text{width} = (x_+ - x_-) \cdot \frac{w}{||w||}$$

From Step II, we know when we have pari (x_+, y_+), we can get:

$$\begin{aligned} 1(w \cdot x_+ + b) - 1 = 0 &\quad \Rightarrow \quad x_+ \cdot w = 1 - b \\ -1(w \cdot x_- + b) - 1 = 0 &\quad \Rightarrow \quad -w \cdot x_- = 1 + b \end{aligned}$$

This gives us

$$\text{width} = \frac{2}{||w||}$$

With the above equation of width, our goal would be to maximum the width as wider the margin, easier for us to separate the different classes. To maximize the above width is equivalent to minimizing the following one

$$\frac{1}{2}||w||^2$$

Now, it's the time to set the Lagrange:

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum \alpha_i[y_i(w \cdot x_i + b) - 1] \tag{2.1}$$

Do the partial differentiation, we can obtain

$$\frac{\partial L}{\partial w} = 0 \quad \Rightarrow \quad w = \sum \alpha_i y_i x_i \tag{2.2}$$

$$\frac{\partial L}{\partial b} = 0 \quad \Rightarrow \quad \sum \alpha_i y_i = 0 \tag{2.3}$$

Now, we substitute Equation (2.2) and (2.3) into (2.1), and get the Lagrangian dual objective function:

$$L(w,b,\alpha) = \frac{1}{2}\left(\sum \alpha_i y_i x_i\right)\left(\sum \alpha_j y_j x_j\right) - \left(\sum \alpha_i y_i x_i\right)\cdot\left(\sum \alpha_j y_j x_j\right) - \sum \alpha_i y_i b + \sum \alpha_i$$

$$L(\alpha) = \sum \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j \tag{2.4}$$

If we solve the model, we can have the vector w orthogonal to the *support vectors*,

$$\hat{w} = \sum_{i=1}^{N} \hat{\alpha}_i y_i x_i$$

In equation (2.4), the maximization depends on the dot product of feature vectors. Since $x_i^T x_j$ is linear, then the decision rule is also linear. To do nonlinear separation, we need construct the *kernel.*

Kernel

$$K(x_i, x_j') = < h(x_i), h(x_j') >$$

where $h(x_i)$ are the transformations based on the input features. Three population choices for K in the SVM literature are

- dth-Degree polynomial: $K(x,x') = (1+ < x, x' >)^d$,
- Radial basis: $K(x,x') = \exp(-\gamma||x - x'||^2)$,
- Neural network: $K(x,x') = \tanh(\kappa_1 < x, x' > + \kappa_2)$.

3 Slack Variable

In the above section, we assumed that the hyperplane is fixed. However, this is not realistic as we always have noise in our datasets, which means the boundaries between different classes are not clear. To solve this problem, we introduce the 'slack' variable ξ (or margin variation). Then, the optimization problem becomes

$$\min_{w\in\mathbb{R}^m, \xi_i\in\mathbb{R}^+} \frac{1}{2}||w||^2 + C\sum_{i}^{N} \xi_i \tag{3.1}$$

subject to

$$y_i(w^T x_i + b) \geq 1 - \xi_i \quad \text{for } i = 1, \cdots N$$

- Every constraint can be satisfied if ξ_i is sufficiently large
- C is a regularization parameter:
  - small C allows constraints to be easily ignored $\rightarrow$ large margin
  - large C makes constraints hard to ignore $\rightarrow$ narrow margin

  - $C = \infty$ enforces all constraints: hard margin

- This is still a quadratic optimization problem and there is a unique minimum. Note, there is only one parameter, C.

4 Dual SVM Derivation

In equation (2.4), we have the primal Lagrange Multiplier:

$$\min_{w,b} \max_{\alpha \geq 0} \frac{1}{2}||w||^2 - \sum_i^N \alpha_i [y_i(w \cdot x_i + b) - 1]$$

According to the minimax theory, then we can swap min and max get the dual form:

$$\max_{\alpha \geq 0} \min_{w,b} \frac{1}{2}||w||^2 - \sum_i^N \alpha_i [y_i(w \cdot x_i + b) - 1]$$

In Equation (2.4), we have the simplified form after substituting w and b into the primal form. We just need solve the following maximization problem:

$$\max_{\alpha \geq 0, \sum_i \alpha_i y_i = 0} L(\alpha) = \sum \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j x_i^T x_j$$

5 Implementation of SVM in Python

First, we notice that the solution function $f(x)$ can be written as

$$\begin{aligned} f(x) &= h(x)^T w + b \\ &= \sum_{i=1}^N \alpha_i y_i < h(x), h(x_i) > +b \end{aligned}$$

This gives the formulation for calculating the bias b.

When it comes to calculating the Lagrange multipliers, we need employ the optimisation package called `CVXOPT`. This package has a function called solvers:

```
cvxopt.solvers.qp(P, q, G, h, A, b]
```

where the corresponding matrices should be:

$$\begin{aligned} \min \ & \frac{1}{2} x^T P x + q^t x \\ s.t. \ & Gx \leq h \\ & Ax = b \end{aligned}$$

In the dual problem, we have

$$\max_{\alpha \geq 0, \sum_i \alpha_i y_i = 0} L(\alpha) = \sum \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j x_i^T x_j$$

Now, let H be the matrix such that $H = y_i y_j x_i^T x_j$, then the optimization becomes:

$$\begin{aligned} &\max_{\alpha} \sum_{i}^{N} \alpha_i - \frac{1}{2}\alpha^T H \alpha \\ &s.t.\ \alpha_i \geq 0 \\ &\qquad \sum_{i}^{N} \alpha_i y_i = 0 \end{aligned}$$

We convert the sums into vector form and multiply both the objective and the constraint by -1 which turns this into a minimization problem and reserves the inequality:

$$\begin{aligned} &\min_{\alpha} \frac{1}{2}\alpha^T H \alpha - 1^T \alpha \\ &s.t.\ -\alpha_i \leq 0 \\ &s.t.\ y^T \alpha = 0 \end{aligned}$$

Therefore, we can have:

- $P := H$ a matrix of size $N \times N$
- $q := -1$ a vector of size $N \times 1$
- $G := -diag[1]$ a diagonal matrix of -1s of size $N \times N$
- $h := 0$ a vector of zero of $N \times 1$
- $A := y$ the label vector of size $N \times 1$
- $b := 0$ a scalar

Here is the steps we will take to estimate our SVM model:

- Step I : calculate the matrix $H = \sum_i^N \sum_j^N y_i y_j x_i^T x_j$
- Step II : calculate Lagrange multipliers
- Step III: calculate the bias $b = f(x) - \sum_i^N \alpha_i y_i < h(x), h(x_i) >$ (you need calculate for each entry of dataset and take the average)
- Step IV: fit the model based on the decision rule $f(x) = \sum_i^N \alpha_i y_i < h(x), h(x_i) > +b$ and sign$[f(x)]$.
- Step V: visualise the boundary

In next page, you can find the code implemented in Python.

```python
# @ Michael
# Reference: Nagi El Hachem
class SVM:
    """
    Attributes:
        weights : w
        bias : b (in the formula b = y - estimatiton)
        alphas: lagrange multipliers
        C : slack penalty
        gram_matrix : gram matrix,
                    contains the computed inner product (xi.T).xj * yi *yj

    """

    def __init__(self):
        self.kernel = linear_kernel
        self.kernel_args = []
        self.colors = ['red', 'blue', 'green', 'purple', 'orange', 'grey']
        self.cmaps = ['Reds', 'Blues', 'Greens', 'Purples', 'Oranges', '
                                          Grey']

    def set_kernel(self, kernel, *args):
        self.kernel = kernel
        self.kernel_args = args

    def __compute_gram_matrix(self):
        """computes the gram coefficients for each pair
        where gram_coef = yi * yj * (xi.T). xj
        """

        gram_matrix = np.zeros([self.labels.shape[0], self.labels.shape[0]]
                                              )
        size = self.labels.shape[0]
        for i in range(size):
            for j in range(size):
                gram_matrix[i, j] = (self.labels[i] * self.labels[j]
                                     * self.kernel(self.data[:, i],
                                                   self.data[:, j],
                                                   *self.kernel_args))
        return gram_matrix

    def __compute_lagrange_multipliers(self):
        # set up solver inputs
        data_size = self.labels.shape[0]
        P = matrix(self.gram_matrix, tc='d')   # d means floats
        q = matrix(np.full(self.labels.shape, -1, dtype=float), tc='d')
        G = matrix(-np.identity(data_size), tc='d') if self.C is None \
            else matrix(np.concatenate((-np.identity(data_size),
                                        np.identity(data_size))), tc='d')
        b = matrix(np.zeros(1), tc='d')
        A = matrix(self.labels, tc='d').T
        h = matrix(np.zeros(data_size), tc='d') if self.C is None \
            else matrix(np.concatenate((np.zeros(data_size), self.C
                                        * np.ones(data_size))), tc='d')
        solvers.options['show_progress'] = self.show_progress
        solution = solvers.qp(P, q, G, h, A, b)['x']  # get coptimal values
        return np.asarray(solution).reshape((data_size, ))  # make it array
```

```
def compute_bias(self, EPSILON=1e-2):
    self.lagrange_multipliers[self.lagrange_multipliers < EPSILON] = 0
    self.support_vectors_idx = np.where(self.lagrange_multipliers > 0)[
                                                 0]
    if self.support_vectors_idx.shape[0] == 0:
        return 0
    bias = 0
    for i in self.support_vectors_idx:
        kernels = np.array([self.kernel(self.data[:, j],
                                    self.data[:, i],
                                    *self.kernel_args)
                            for j in range(self.data.shape[1])])
        bias += self.labels[i] - np.sum(self.lagrange_multipliers
                                    * self.labels * kernels)
    return bias / self.support_vectors_idx.shape[0]

def __train(self, data, labels):
    """ find the separator coordiantes
    """

    self.data, self.labels = data, labels
    self.gram_matrix = self.__compute_gram_matrix()
    self.lagrange_multipliers = self.__compute_lagrange_multipliers()
    self.bias = self.compute_bias()

def train(self, data, labels, C=None, show_progress=False):
    self.C = C
    self.show_progress = show_progress
    classes = np.unique(labels)
    Y = np.empty(shape=labels.shape)
    N = classes.shape[0]
    self.multi_lagrange_multipliers = [None] * N
    self.multi_bias = [None] * N
    self.multi_labels = [None] * N
    self.labels_all = labels
    for i, c in enumerate(classes):
        # if we have more than 2 classes
        Y[:] = -1
        idx = np.where(labels == c)[0]
        Y[idx] = 1
        self.__train(data, Y)
        temp_short_line = svm.lagrange_multipliers.copy()
        self.multi_lagrange_multipliers[i] = temp_short_line
        self.multi_labels[i] = Y.copy()
        self.multi_bias[i] = self.bias

# now we are ready for classifying
def decision(self, X):
    kernels = np.array([self.kernel(self.data[:, i], X, *self.
                                             kernel_args)
                        for i in range(self.data.shape[1])])
    desc = np.sum(self.lagrange_multipliers*self.labels*kernels)+self.
                                             bias
    return desc

def process(self, data):
    """
```

```
        returns a list of labels of data
        """
        labels = self.decision(data)
        labels[labels <= 0] = -1
        labels[labels > 0] = 1
        return labels

    def print_2Ddecision(self, nb_samples=50, print_sv=True, print_non_sv=
                                                              True,
                         levels=[0., 1.], color='seismic'):
        eps = 0.2
        xmin, ymin = np.min(self.data, axis=1) - eps
        xmax, ymax = np.max(self.data, axis=1) + eps

        # generate nb_samples floats between ximin, xmax
        x = np.linspace(xmin, xmax, nb_samples)
        y = np.linspace(ymin, ymax, nb_samples)
        x, y = np.meshgrid(x, y)  # generates a grid with x, y values

        num_categories = len(self.multi_lagrange_multipliers)
        c_z = np.empty(shape=(num_categories, ) + x.shape)
        alpha = 1

        for c in range(num_categories):
            svm.lagrange_multipliers = self.multi_lagrange_multipliers[c]
            svm.labels = self.multi_labels[c]
            svm.bias = self.multi_bias[c]

            for i in range(x.shape[0]):
                for j in range(x.shape[1]):
                    c_z[c, i, j] = svm.decision(np.array((x[i, j], y[i, j])
                                                       ))

            if num_categories != 2:
                plt.pcolor(x, y, c_z[c, :, :],
                           cmap=self.cmaps[c],
                           vmin=-1, vmax=1, alpha=alpha)
            elif c == 0:
                plt.pcolor(x, y, c_z[c, :, :], cmap=color,
                           vmin=-1, vmax=1)

            if 0. in levels:
                plt.contour(x, y, c_z[c, :, :], levels=[0.], colors='white'
                                                                      ,
                            alpha=0.5, linewidths=3)
            if 1. in levels:
                plt.contour(x, y, c_z[c, :, :], levels=[1.],
                            colors=self.colors[c], linestyles='dashed',
                            linewidths=1.5)
            if -1. in levels:
                plt.contour(x, y, c_z[c, :, :], levels=[-1.],
                            colors=self.colors[c],
                            linestyles='dashed', linewidths=0.5)

            alpha /= 2

        plt.axis([x.min(), x.max(), y.min(), y.max()])
        self.__plot_data(print_sv, print_non_sv)
    def __plot_data(self, print_sv, print_non_sv):
        classes = np.unique(self.labels_all)
        for i, c in enumerate(classes):
            idx = None
            # filter support vector
            lagrange_multipliers = self.multi_lagrange_multipliers[i]
            if print_sv and not print_non_sv:
                idx = np.where(lagrange_multipliers != 0)[0]
            elif not print_sv and print_non_sv:
                idx = np.where(lagrange_multipliers == 0)[0]
            elif not print_sv and not print_non_sv:
                idx = [-1]
            # get class data
            tmp = np.where(self.labels_all == c)[0]
            idx = tmp if idx is None else np.intersect1d(idx, tmp)
            x_sub = self.data[:, idx]
            # scatter filtered data
            plt.scatter(x_sub[0, :], x_sub[1, :], c=self.colors[i])
```

Reference

Lecture notes by *David Sontag*, Link(click)
Lecture by *Patrick Winston*, Link(click)
Online blog by *Xavier Bourret Sicotte*, Link(click)