Data Analysis and Machine Learning: Support Vector Machines

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Nov 5, 2018

Support Vector Machines, overarching aims

A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex but small-sized or medium-sized datasets.

The case with two well-separated classes only can be understood in an intuitive way in terms of lines in a two-dimensional space separating the two classes (see figure below).

The basic mathematics behind the SVM is however less familiar to most of us. It relies on the definition of hyperplanes and the definition of a **margin** which separates classes (in case of classification problems) of variables. It is also used for regression problems.

With SVMs we distinguish between hard margin and soft margins. The latter introduces a so-called softening parameter to be discussed below. We distinguish also between linear and non-linear approaches. The latter are the most frequent ones since it is rather unlikely that we can separate classes easily by say straight lines.

Hyperplanes and all that

The theory behind support vector machines (SVM hereafter) is based on the mathematical description of so-called hyperplanes. Let us start with a two-dimensional case. This will also allow us to introduce our first SVM examples. These will be tailored to the case of two specific classes, as displayed in the figure here.

We assume here that our data set can be well separated into two domains, where a straight line does the job in the separating the two classes. Here the two classes are represented by either crosses or circles.

What is a hyperplane?

The aim of the SVM algorithm is to find a hyperplane in an p-dimensional space, where p is the number of features that distinctly classifies the data points.

In a p-dimensional space, a hyperplane is what we call an affine subspace of dimension of $p-1$. As an example, in two dimension, a hyperplane is simply as straight line while in three dimensions it is a two-dimensional subspace, or stated simply, a plane.

In two dimensions, with the variables x_1 and x_2, the hyperplane is defined as

$$b + w_1x_1 + w_2x_2 = 0,$$

where b is the intercept and w_1 and w_2 define the elements of a vector orthogonal to the line $b + w_1x_1 + w_2x_2 = 0$. In two dimensions we define the vectors $\boldsymbol{x} = [x1, x2]$ and $\boldsymbol{w} = [w1, w2]$. We can then rewrite the above equation as

$$\boldsymbol{w}^T\boldsymbol{x} + b = 0.$$

A p-dimensional space of features

We limit ourselves to two classes of outputs y_i and assign these classes the values $y_i = \pm 1$. In a p-dimensional space of say p features we have a hyperplane defines as

$$b + wx_1 + w_2x_2 + \cdots + w_px_p = 0.$$

If we define a matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_p]$ of dimension $n \times p$, where n represents the observations for each feature and each vector x_i is a column vector of the matrix $\boldsymbol{X}$,

$$\boldsymbol{x}_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \cdots \\ \cdots \\ x_{ip} \end{bmatrix}.$$

If the above condition is not met for a given vector $\boldsymbol{x}_i$ we have

$$b + w_1x_{i1} + w_2x_i2 + \cdots + w_px_{ip} > 0,$$

if our output $y_i = 1$. In this case we say that $\boldsymbol{x}_i$ lies on one of the sides of the hyperplane and if

$$b + w_1x_{i1} + w_2x_i2 + \cdots + w_px_{ip} < 0,$$

for the class of observations $y_i = -1$, then $\boldsymbol{x}_i$ lies on the other side.

Equivalently, for the two classes of observations we have

$$y_i \left(b + w_1 x_{i1} + w_2 x_i 2 + \cdots + w_p x_{ip}\right) > 0.$$

When we try to separate hyperplanes, if it exists, we can use it to construct a natural classifier: a test observation is assigned a given class depending on which side of the hyperplane it is located.

The two-dimensional case

Let us try to develop our intuition about SVMs by limiting ourselves to a two-dimensional plane. To separate the two classes of data points, there are many possible lines (hyperplanes if you prefer a more strict naming) that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

What a linear classifier attempts to accomplish is to split the feature space into two half spaces by placing a hyperplane between the data points. This hyperplane will be our decision boundary. All points on one side of the plane will belong to class one and all points on the other side of the plane will belong to the second class two.

Unfortunately there are many ways in which we can place a hyperplane to divide the data. Below is an example of two candidate hyperplanes for our data sample.

Getting into the details

Let us define the function

$$f(x) = \boldsymbol{w}^T \boldsymbol{x} + b = 0,$$

as the function that determines the line L that separates two classes (our two features), see the figure here.

Any point defined by $\boldsymbol{x}_i$ and $\boldsymbol{x}_2$ on the line L will satisfy $\boldsymbol{w}^T(\boldsymbol{x}_1 - \boldsymbol{x}_2) = 0$.

The signed distance δ from any point defined by a vector $\boldsymbol{x}$ and a point $\boldsymbol{x}_0$ on the line L is then

$$\delta = \frac{1}{||\boldsymbol{w}||}(\boldsymbol{w}^T \boldsymbol{x} + b).$$

First attempt at a minimization approach

How do we find the parameter b and the vector $\boldsymbol{w}$? What we could do is to define a cost function which now contains the set of all misclassified points M and attempt to minimize this function

$$C(\boldsymbol{w}, b) = -\sum_{i \in M} y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b).$$

We could now for example define all values $y_i = 1$ as misclassified in case we have $\boldsymbol{w}^T\boldsymbol{x}_i + b < 0$ and the opposite if we have $y_i = -1$. Taking the derivatives gives us

$$\frac{\partial C}{\partial b} = -\sum_{i \in M} y_i,$$

and

$$\frac{\partial C}{\partial \boldsymbol{w}} = -\sum_{i \in M} y_i x_i.$$

Solving the equations

We can now use the Newton-Raphson method or gradient descent to solve the equations

$$b \leftarrow b + \eta \frac{\partial C}{\partial b},$$

and

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \frac{\partial C}{\partial \boldsymbol{w}},$$

where η is our by now well-known learning rate.

There are however problems with this approach, although it looks pretty straightforward to implement. In case we separate our data into two distinct classes, we may up with many possible lines, as indicated in the figure and shown by running the following program. For small gaps between the entries, we may also end up needing many iterations before the solutions converge and if the data cannot be separated properly into two distinct classes, we may not experience a converge at all.

A better approach

A better approach is rather to try to define a large margin between the two classes (if they are well separated from the beginning).

Thus, we wish to find a margin M with $\boldsymbol{w}$ normalized to $||\boldsymbol{w}|| = 1$ subject to the condition

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq M \; \forall i = 1, 2, \ldots, p.$$

All points are thus at a signed distance from the decision boundary defined by the line L. The parameters b and w_1 and w_2 define this line.

We seek thus the largest value M defined by

$$\frac{1}{||\boldsymbol{w}||} y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq M \; \forall i = 1, 2, \ldots, n,$$

or just

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq M||\boldsymbol{w}|| \; \forall i.$$

If we scale the equation so that $||\boldsymbol{w}|| = 1/M$, we have to find the minimum of $\boldsymbol{w}^T\boldsymbol{w} = ||\boldsymbol{w}||$ (the norm) subject to the condition

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) \geq 1 \; \forall i.$$

We have thus defined our margin as the invers of the norm of $\boldsymbol{w}$. We want to minimize the norm in order to have a as large as possible margin M. Before we proceed, we need to remind ourselves about Lagrangian multipliers.

A quick reminder on Lagrangian multipliers

Consider a function of three independent variables $f(x, y, z)$. For the function f to be an extreme we have

$$df = 0.$$

A necessary and sufficient condition is

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z} = 0,$$

due to

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz.$$

In many problems the variables x, y, z are often subject to constraints (such as those above for the margin) so that they are no longer all independent. It is possible at least in principle to use each constraint to eliminate one variable and to proceed with a new and smaller set of independent varables.

The use of so-called Lagrangian multipliers is an alternative technique when the elimination of variables is incovenient or undesirable. Assume that we have an equation of constraint on the variables x, y, z

$$\phi(x, y, z) = 0,$$

resulting in

$$d\phi = \frac{\partial \phi}{\partial x}dx + \frac{\partial \phi}{\partial y}dy + \frac{\partial \phi}{\partial z}dz = 0.$$

Now we cannot set anymore

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z} = 0,$$

if $df = 0$ is wanted because there are now only two independent variables! Assume x and y are the independent variables. Then dz is no longer arbitrary.

Adding the muliplier

However, we can add to

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz,$$

a multiplum of $d\phi$, viz. $\lambda d\phi$, resulting in

$$df + \lambda d\phi = (\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial x})dx + (\frac{\partial f}{\partial y} + \lambda\frac{\partial \phi}{\partial y})dy + (\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial z})dz = 0.$$

Our multiplier is chosen so that

$$\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial z} = 0.$$

We need to remember that we took dx and dy to be arbitrary and thus we must have

$$\frac{\partial f}{\partial x} + \lambda\frac{\partial \phi}{\partial x} = 0,$$

and

$$\frac{\partial f}{\partial y} + \lambda\frac{\partial \phi}{\partial y} = 0.$$

When all these equations are satisfied, $df = 0$. We have four unknowns, x, y, z and λ. Actually we want only x, y, z, λ needs not to be determined, it is therefore often called Lagrange's undetermined multiplier. If we have a set of constraints ϕ_k we have the equations

$$\frac{\partial f}{\partial x_i} + \sum_k \lambda_k \frac{\partial \phi_k}{\partial x_i} = 0.$$

Setting up the problem

In order to solve the above problem, we define the following Lagrangian function to be minimized

$$\mathcal{L}(\lambda, b, \boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_{i=1}^{n} \lambda_i \left[y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) - 1\right],$$

where λ_i is a so-called Lagrange multiplier subject to the condition $\lambda_i \geq 0$.

Taking the derivatives with respect to b and $\boldsymbol{w}$ we obtain

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_i \lambda_i y_i = 0,$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 = \boldsymbol{w} - \sum_i \lambda_i y_i \boldsymbol{x}_i.$$

Inserting these constraints into the equation for $\mathcal{L}$ we obtain

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i\lambda_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j,$$

subject to the constraints $\lambda_i \geq 0$ and $\sum_i \lambda_i y_i = 0$. We must in addition satisfy the Karush-Kuhn-Tucker (KKT) condition

$$\lambda_i \left[y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) - 1\right] \ \forall i.$$

1. If $\lambda_i > 0$, then $y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1$ and we say that x_i is on the boundary.
2. If $y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) > 1$, we say x_i is not on the boundary and we set $\lambda_i = 0$.

When $\lambda_i > 0$, the vectors $\boldsymbol{x}_i$ are called support vectors. They are the vectors closest to the line (or hyperplane) and define the margin M.

The problem to solve

We can rewrite

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i\lambda_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j,$$

and its constraints in terms of a matrix-vector problem where we minimize w.r.t. λ the following problem

$$\frac{1}{2}\boldsymbol{\lambda}^T \begin{bmatrix} y_1y_1\boldsymbol{x}_1^T\boldsymbol{x}_1 & y_1y_2\boldsymbol{x}_1^T\boldsymbol{x}_2 & \dots & \dots & y_1y_n\boldsymbol{x}_1^T\boldsymbol{x}_n \\ y_2y_1\boldsymbol{x}_2^T\boldsymbol{x}_1 & y_2y_2\boldsymbol{x}_2^T\boldsymbol{x}_2 & \dots & \dots & y_1y_n\boldsymbol{x}_2^T\boldsymbol{x}_n \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ y_ny_1\boldsymbol{x}_n^T\boldsymbol{x}_1 & y_ny_2\boldsymbol{x}_n^T\boldsymbol{x}_2 & \dots & \dots & y_ny_n\boldsymbol{x}_n^T\boldsymbol{x}_n \end{bmatrix} \boldsymbol{\lambda} - \mathbb{1}\boldsymbol{\lambda},$$

subject to $\boldsymbol{y}^T\boldsymbol{\lambda} = 0$. Here we defined the vectors $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots, \lambda_n]$ and $\boldsymbol{y} = [y_1, y_2, \dots, y_n]$.

The last steps

Solving the above problem, yields the values of λ_i. To find the coefficients of your hyperplane we need simply to compute

$$\boldsymbol{w} = \sum_i \lambda_i y_i \boldsymbol{x}_i.$$

With our vector $\boldsymbol{w}$ we can in turn find the value of the intercept b (here in two dimensions) via

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1,$$

resulting in

$$b = \frac{1}{y_i} - \boldsymbol{w}^T\boldsymbol{x}_i,$$

or if we write it out in terms of the support vectors only, with N_s being their number, we have

$$b = \frac{1}{N_s}\sum_{j\in N_s}\left(y_j - \sum_{i=1}^{n}\lambda_i y_i \boldsymbol{x}_i^T\boldsymbol{x}_j\right).$$

With our hyperplane coefficients we can use our classifier to assign any observation by simply using

$$y_i = \text{sign}(\boldsymbol{w}^T\boldsymbol{x}_i + b).$$

Below we discuss how to find the optimal values of λ_i. Before we proceed however, we discuss now the so-called soft classifier.

A soft classifier

Till now, the margin is strictly defined by the support vectors. This defines what is called a hard classifier, that is the margins are well defined.

Suppose now that classes overlap in feature space, as shown in the figure here. One way to deal with this problem before we define the so-called **kernel approach**, is to allow a kind of slack in the sense that we allow some points to be on the wrong side of the margin.

We introduce thus the so-called **slack** variables $\boldsymbol{\xi} = [\xi_1, x_2, \ldots, x_n]$ and modify our previous equation

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1,$$

to

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1 - \xi_i,$$

with the requirement $\xi_i \geq 0$. The total violation is now $\sum_i \xi$. The value ξ_i in the constraint the last constraint corresponds to the amount by which the prediction $y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1$ is on the wrong side of its margin. Hence by bounding the sum $\sum_i \xi_i$, we bound the total amount by which predictions fall on the wrong side of their margins.

Misclassifications occur when $\xi_i > 1$. Thus bounding the total sum by some value C bounds in turn the total number of misclassifications.

Soft optmization problem

This has in turn the consequences that we change our optmization problem to finding the minimum of

$$\mathcal{L} = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_{i=1}^{n}\lambda_i\left[y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) - (1-\xi_)\right] + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\gamma_i\xi_i,$$

subject to

$$y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) = 1 - \xi_i\ \forall i,$$

with the requirement $\xi_i \geq 0$.

Taking the derivatives with respect to b and $\boldsymbol{w}$ we obtain

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_i \lambda_i y_i = 0,$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 = \boldsymbol{w} - \sum_i \lambda_i y_i \boldsymbol{x}_i,$$

and

$$\lambda_i = C - \gamma_i \; \forall i.$$

Inserting these constraints into the equation for $\mathcal{L}$ we obtain the same equation as before

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j,$$

but now subject to the constraints $\lambda_i \geq 0$, $\sum_i \lambda_i y_i = 0$ and $0 \leq \lambda_i \leq C$. We must in addition satisfy the Karush-Kuhn-Tucker condition which now reads

$$\lambda_i \left[y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) - (1 - \xi_) \right] = 0 \; \forall i,$$

$$\gamma_i \xi_i = 0,$$

and

$$y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) - (1 - \xi_) \geq 0 \; \forall i.$$

Kernels and non-linearity

The cases we have studied till were all characterized by two classes with a close to linear separability. The classifiers we have described so far find linear boundaries in our input feature space. It is possible to make our procedure more flexible by exploring the feature space using other basis expansions such higher-order polynomials, wavelets, splines etc.

If our feature space is not easy to separate, as shown in the figure here, we can achieve a better separation by introducing more complex basis functions. The ideal would be, as shown in the next figure, to, via a specific transformation to obtain a separation between the classes which is almost linear.

The change of basis, from $x \to z = \phi(x)$ leads to the same type of equations to be solved, except that we need to introduce for example a polynomial transformation to a two-dimensional training set.

The equations

Suppose we define a polynomial transformation of degree two (we continue to live in a plane with x_1 and x_2 as variables)

$$z = \phi(x) = \big(1, x_1, x_2, x_1^2, x_2^2, x_1 x_2\big).$$

With our new basis, the equations we solved earlier are basically the same, that is we have now (without the slack option for simplicity)

$$\mathcal{L} = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i\lambda_j y_i y_j \boldsymbol{z}_i^T \boldsymbol{Z}_j,$$

subject to the constraints $\lambda_i \geq 0$, $\sum_i \lambda_i y_i = 0$, and for the support vectors

$$y_i(\boldsymbol{w}^T\boldsymbol{z}_i + b) = 1 \; \forall i,$$

from which we also find b.

Different kernels

Quadratic coefficient matrix

Mercer's theorem

How do we solve these problems

If we use Python as programming language and wish to venture beyond **scikit-learn**, **tensorflow** and similar software which makes our lives so much easier, we need to dive into the wonderful world of quadratic programming. We can, if we wish, solve the minimization problem using say standard gradient methods or conjugate gradient methods. However, these methods tend to exhibit a rather slow converge. So, welcome to the promised land of quadratic programming.

The functions we need are contained in the quadratic programming package **CVXOPT** and we need to import it

```
import numpy
import cvxopt
```

Let us first set up the standard form the of quadratic programming (QP) equations by defining the problem as

min

subject to Gx u. Note that x itself is not provided to the solver, since it is an internal variable being optimized over. In particular, this means that the solver has no explicit knowledge of x itself; everything is implicity defined by the supplied parameters. It is essential that the same variable order is maintained for the relevant parameters (e.g., qi Non-convexity implies the existence of local optima, making it difficult to find global optima.

collapsed all inequality constraints into a single G matrix of the standard form. Since there are no equality constraints, we do not need to provide the empty A, b. Note that even though y 2 did not appear in the original objective, we had to include it with zero coefficients in P because the solver parameters must be defined using the full set of variables. Even if certain variables only appear in constraints, they will still need to be expressed with zero coefficients in

the objective parameters, and vice versa. Let us first define the above parameters in Python. CVXOPT supplies its own matrix object; all arguments given to its solvers must be in this matrix type. There are two ways to do this. The first is to define the matrix directly with (potentially nested) lists: from cvxopt import matrix

```
P = matrix([[1.0,0.0],[0.0,0.0]])
q = matrix([3.0,4.0])
G = matrix([[-1.0,0.0,-1.0,2.0,3.0],[0.0,-1.0,-3.0,5.0,4.0]])
h = matrix([0.0,0.0,-15.0,100.0,80.0])
```