Support Vector Machines

Sylesh Suresh

October 2018

1 Introduction

Support Vector Machines (SVMs) are one of the most popular supervised learning models today, able to perform both linear and nonlinear classification.

2 Linear Classification

The idea behind SVMs is to create a linear decision boundary called a hyperplane (which is a line when the feature space has two dimensions, a plane when the feature space has three dimensions, and so on). This hyperplane is drawn in a way that maximizes the distance between itself and the samples closest to it in the feature space.

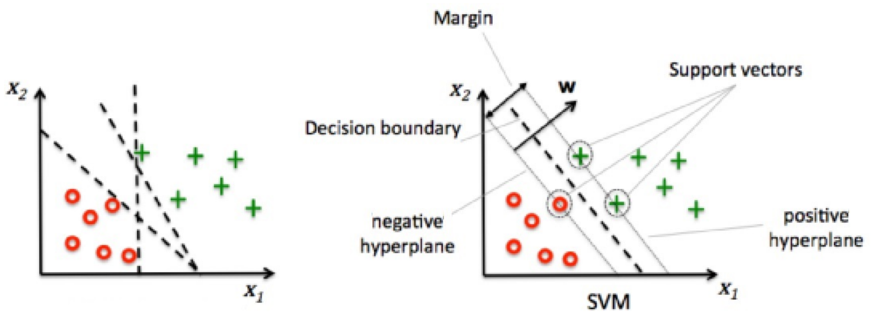

Figure 1: Support Vector Machine

The decision boundaries to the left separate the training data correctly but would not generalize well to unseen data, being too close to the training samples (i.e. having a small margin). On the other hand, the decision boundary to the right marked by the dashed line separates the training data and generalizes well to unseen data, having a large margin. Maximization of the margin allows for the least generalization error.

Support vectors are the data points (which can be represented as vectors) that are closest to our potential decision boundary. The negative support vectors and positive support vectors are on opposite sides of the decision boundary. The negative hyperplane is the hyperplane closest to the negative support vectors that is also parallel to the decision boundary. The positive hyperplane is the hyperplane closest to the positive support vectors that is also parallel to the decision boundary. Our decision boundary will be exactly in the middle of these two hyperplanes.

$\boldsymbol{w}$ is defined as a vector normal to the decision boundary. The positive hyperplane is defined as

$$\boldsymbol{w} \cdot \boldsymbol{x_{pos}} + w_0 = 1$$

Here x_{pos} represents the positive support vectors. while the negative hyperplane is:

$$\boldsymbol{w} \cdot \boldsymbol{x_{neg}} + w_0 = -1$$

x_{neg} represents the negative support vectors. w_0 is a constant that allows us to set the equations equal to 1 or -1 respectively. The idea is that the dot product with the positive support vectors will be greater than the dot product with the negative support vectors.

These equations represent hyperplanes (which are simply lines in our 2-dimensional case) because all the vectors that can be plugged in in the place of x_{pos} and x_{neg} taken together construct a line. That is, if you colored in the tip of every vector whose tail began in the origin and satisfied the above equations, you would see two lines.

The endgame here is to choose the vector $\boldsymbol{w}$ such that the distance between the positive and negative hyerplane (otherwise known as the margin) is maximized. To do this, we play around with the equations until we get some specific value that we can maximize/minimize that will in turn maximize the margin.

We can combine these equations by subtracting the second equation from the first:

$$\boldsymbol{w}(x_{pos} - x_{neg}) = 2 \tag{1}$$

To calculate the margin, first, let us take the difference between a positive support vector and a negative support vector.

$$x_{pos} - x_{neg}$$

This will get us a vector whose magnitude represents the distance between some line segment whose endpoints lie on each of the two hyperplanes. However, this distance is not the perpendicular distance, which is what we are looking for.

So, we need to take the dot product of this with a unit vector perpendicular to the hyperplanes. We earlier defined $\boldsymbol{w}$ to be normal to the hyperplanes, so $\frac{\boldsymbol{w}}{||\boldsymbol{w}||}$ serves this purpose:

$$\frac{\boldsymbol{w}(x_{pos} - x_{neg})}{||\boldsymbol{w}||}$$

Using 1, we arrive at:

$$\frac{\boldsymbol{w}(x_{pos} - x_{neg})}{||\boldsymbol{w}||} = \frac{2}{||\boldsymbol{w}||}$$

We must maximize $\frac{2}{||\boldsymbol{w}||}$ to maximize the margin. For mathematical convenience, we can minimize

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

to achieve the same effect. Of course, we're not going to just blindly maximize the margin - we need an actual classifier. Thus, the constraint for this optimization problem is that the samples are actually classified correctly:

$$\boldsymbol{w} \cdot \boldsymbol{x_i} + w_0 = 1 \text{ if } y_i = 1$$

$$\boldsymbol{w} \cdot \boldsymbol{x_i} + w_0 = -1 \text{ if } y_i = -1$$

where x_i is a particular sample and y_i is the class of the sample. More compactly:

$$y_i(w_0 + \boldsymbol{w} \cdot \boldsymbol{x_i}) - 1 = 0 \tag{3}$$

We now have a classic optimization problem that can be solved through Lagrangian multipliers. The takeaway here is that support vector machines draw some linear decision boundary and maximize that boundary's distance to the nearest data samples so that the boundary generalizes well to new data. We can do this by setting up some equations, doing some math magic, and arriving at a constrained optimization problem which we can solve with multivariable calculus.

3 Soft-Margin Classification

Most of the time, our data will not exactly linearly separable. The standard method is to allow the SVM to misclassify some data points, and pay a cost for each misclassified point. We can accomplish this by adding a slack variable ξ. Our optimization problem becomes minimizing $\frac{1}{2}||\boldsymbol{w}||^2 + C\sum_i \xi_i$ with the constraint $y_i(w_0 + \boldsymbol{w} \cdot \boldsymbol{x_i}) \geq 1 - \xi_i$. C is a regularization hyperparameter; if C is set to be small, more misclassifications are allowed, but if C is set to be large, less misclassifications are allowed. In the end, we are still drawing a linear decision boundary even though the data is itself not exactly linearly separable. We're assuming that a linear decision boundary is a good approximation.

4 Non-binary Classification

SVM's are inherently useful for binary classification. However, they can also be used for data with more than two classes.

4.1 Pairwise Classification

If we train an SVM for every possible combination of pairs of classes, we will train $\frac{n(n+1)}{2}$ SVMs for a dataset with n classes. Classify a point by running it through all the SVMs and adding up the number of times the point is classified into each class. The class with the most number is considered the label.

5 Nonlinear Classification using Kernels

In the real world, data is usually not linearly separable, meaning that the support vector machine as cannot accurately separate the data, even if we try to use a soft-margin (that is, the data is really, genuinely nonlinearly separable). However, we can project the data onto a higher dimensional space where the data is linearly separable using a mapping function $\phi(\cdot)$ For example:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

Using this mapping function allows us to separate the two classes below (indicated by red and blue) with a linear hyperplane. We can then project this back into two-dimensional space where the decision boundary becomes nonlinear.

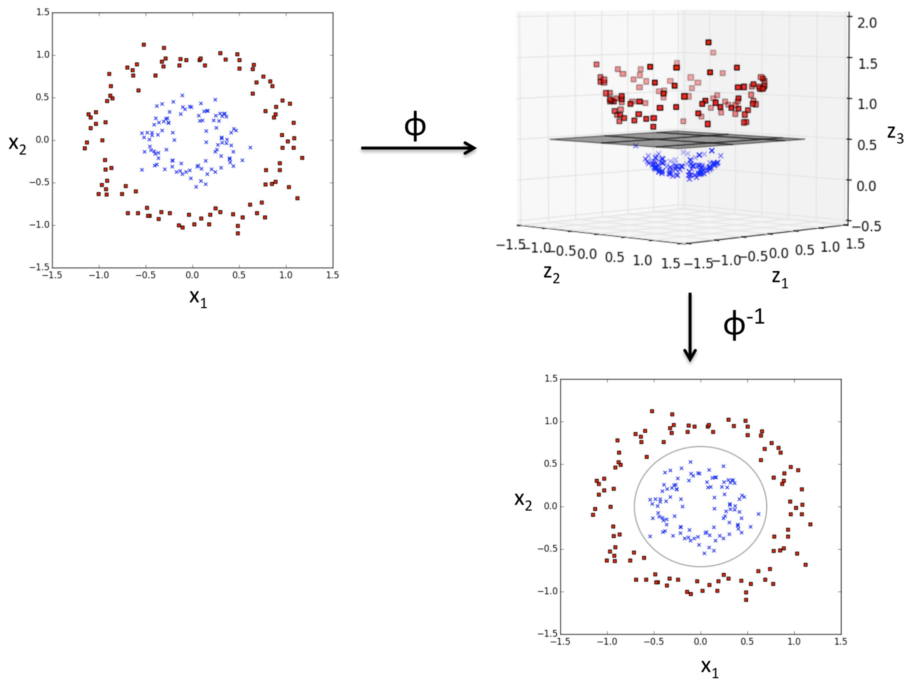

Figure 2: Projecting to higher space

The problem, however, with this approach is its efficiency. When solving the optimization problem of maximizing the margin, the pair-wise dot products of different training samples $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$ must be calculated, a very computationally expensive process in high-dimensional space. To solve this, we can use the kernel trick.

So, the problem is dot products are expensive. $\phi(x_i) \cdot \phi(x_j)$ is hard to compute since there can potentially be many, many dimensions. Dot products are nothing but a similarity measure between two vectors. There are similarity functions called kernels which calculate the similarity between two vectors. These kernels don't map anything onto a higher-dimensional space, they simply calculate the similarity between two vectors. It turns out, though, that we can construct kernel functions so that they implicitly calculate the dot product in higher dimensional space without actually mapping the vectors on to that space. That is, there are functions $K(a, b)$ such that $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$. Again, $K(a, b)$ doesn't actually map anything on to a higher dimensional space. The kernel function works completely within the original feature space and calculates similarity somehow within that feature space and it just so happens that $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$. This is pretty cool; all we care about is the dot products between vectors, so kernel functions are the perfect solution to our problem. The mathematical details of how exactly we create kernel functions so that they happen to implicitly calculate a higher-dimensional dot product in a lower dimensional space is a bit tricky and thus out of the scope of this club.

One of the most popular kernel functions is the Radial Basis Function kernel (RBF kernel) or Gaussian kernel:

$$k(\boldsymbol{x_i}, \boldsymbol{x_j}) = \exp\left(-\gamma ||\boldsymbol{x_i} - \boldsymbol{x_j}||^2\right)$$

γ is a free parameter that can be optimized.

6 Library

To start off the year easy, we will be using Scikit-Learn, which has multi-class support built-in.

So, in practice, what do you need to know about support vector machines to make them classify data well, and when should you use SVMs? If you have a lot of features in your data, SVMs tend to be more computationally efficient than other algorithms such as deep neural networks. Moreover, there are few hyperparameters to be chosen (basically, there is the regularization parameter C and the type of kernel function to use). This means that it is fairly easy to squeeze out the classification accuracy out of SVMs. In the future, you will learn about neural networks and their variants, which are generally the most popular type of classification algorithm for most problems. However, they have many hyperparameters that can be optimized, and in comparison, SVMs are much easier to get the maximimum classification accuracy out of in a short amount of time.

If you choose to use SVMs, the two things you need to toy around with are the regularization parameter C and the type of kernel function. You can choose whichever kernel function gives you the best training accuracy, and you can choose the value of C based on whether you are overfitting or underfitting. Increase C if you are underfitting, and decrease C if you are overfitting.

```
from sklearn import svm

# X = data inputs, Y = Label
X = [[0], [1], [2], [3]]
Y = [0, 1, 2, 3]
clf = svm.SVC(decision_function_shape='ovo')
clf.fit(X, Y)    #train

dec = clf.decision_function([[1]])
dec.shape[1] # 4 classes: 4*3/2 = 6

clf.predict([[2.5]]) #test
```