**Support Vector Machine Classification**

Let’s consider again that we have some objects,

A screenshot of a computer

Description automatically generated with low confidence

plotted in some feature space. SVC tries to draw a (decision) boundary around the different classes. This is illustrated below, for a 2d case, and 2 classes (unlike table above).

Chart, scatter chart

Description automatically generated Diagram

Description automatically generated

The decision boundary could be linear, as in the example on the left, or nonlinear, as on the right. Linear SVC’s obviously do the (hyper)planar decision boundaries. Nonlinear SVC’s use a *kernel* to create nonlinear decision boundaries.

**Linear Classifier**

Let’s consider the case where a plane can separate the two regions. In that case the data is said to be linearly separable. Say we’re dealing with just two features for simplicity. Then mathematically, we want the formula for a plane f = w1x1 + w2x2 + b, whose value f(x1,x2) > 0 for all (x1,x2) that are classified as v = +1, and whose value f(x1,x2) < 0 for all (x1,x2) that are classified as y = -1. The decision boundary would be where f(x1,x2) = 0. This would be a line in the x1, x2 plane, given by:



So our task would be to determine the general weights and bias, w1, w2, b so that we get the correct plane. Of course in the example to the left, there would be many different decision boundaries, and therefore planes, and therefore values of b, w1,2, that would work. The best one is considered the one which minimizes the loss. The loss, in this case, is called the Hinge Loss.



where fj and yj are the values of f and y at point xj. So if fj < 0 and yj = 1, then the loss for that point will be 1 + |fj|·| yj| > 0. If fj > 0 and yj = -1, then the loss for that point will also be 1 + |fj|·|yj|. So all incorrectly classified points have a loss. But correctly classified points have a loss too. If 1 > fj > 0 and yj = 1 there is loss, and if -1 < fj < 0 and yj = -1 there is loss. This would encourage the algorithm to move the hyperplane boundary far from both the positive and negative class regions, so that by the time f crosses any points in the positive y = 1 region, f is greater than 1, and so that by the time f crosses any points in the negative y = -1 region, f is less than -1. So correctly classified points will have a loss whenever |fjyj| < 1. So only a few points will contribute to the loss: these incorrectly classified points, and the too-weakly-correctly classified points near the boundary. These points are called the **support vectors**. Can see that this loss function encourages, all things being equal, putting the plane as far between the two classes as possible – to not hug any one class too closely.

**Algorithm?**

So generically, we’d plug all our points into f and calculate the total loss as described above. And if it’s not zero, then we’d adjust the parameters w1,2,, b and see if we fare better. Now we want to consider the algorithm for adjusting our weights to get a hyperplane that correctly classifies our data points. Well, they say that it’s,



where,



and α is an arbitrary parameter called the *learning rate*. So basically:

b (new) = b (old) + y(**x**)

**w** (new) = **w** (old) + αy(**x**)**x**

So y(**x**) is the sign value our data points are supposed to have, and this is known. f(**x**) is the value that our plane would predict. If product of two is positive, then the sgn value of our point matches the sgn the hyperplane assigns that point. If negative, then they are mismatched. Can verify that the algorithm will make y(**x**)f(**x**) more positive at least.



x2 is always positive. And y(x)2 = 1. Of course lots of algorithms could make it more positive. But…whatever.

**Exploring the Linear Model and Hyperparameters**

Here's a plot with 500 points and no outliers. Can see the linear SVC gets it pretty much exactly right (we’ll discuss C in a second). Not as exactly right in the three class case. with nearly linear borders. But pretty good.

A diagram of a red and blue diagram

Description automatically generated A diagram of a graph

Description automatically generated

and now we’ll do 10% outliers in the two-class and 15% outliers in the three-class case.

A diagram of a graph

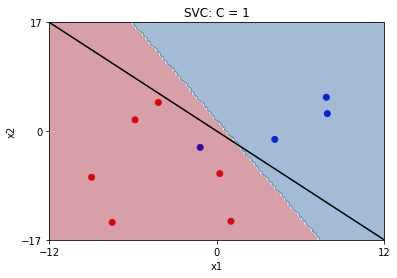
Description automatically generated with medium confidence A diagram of different colored circles

Description automatically generated

Not too bad, but I would’ve expected the blue area to fill out that triangle region a bit more.

**Hyperparameter: C**

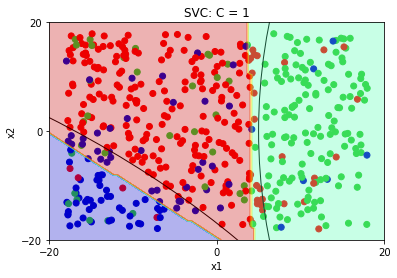
The linear classifier has a regularization parameter, C, which I just think of as ‘curviness’ (more apparent in the nonlinear classifier case). Default value of C = 1. People say C plays a role similar to α apropos Ridge and Lasso regularization, but is inversely related to α, so C ~ 1/α. The higher C is, the more (relatively) penalized are misclassifications, as opposed to correctly-classified-but-still-lossy points (see above). And so the algorithm is more willing to run a plane between the two classes, if it can, even if it’s a tight fit. So higher C might run the risk of overfitting. Here’s three plots for 10 datapoints, with one outlier (blue guy below the black line),

 A diagram of a graph

Description automatically generated with medium confidenceA diagram of a graph

Description automatically generated with medium confidence

So can see that when C = 1, it accepts a misclassification because it can apparently reduce the loss even further by having the two classes relatively far from its border. But when C = 100, misclassifications are heavily penalized, and so it runs a plane just barely in between the classes, even though there will definitely be loss acrued from having correctly classified points so close to the border. Can see its effect more clearly in the 3-class case,

 A diagram of a graph

Description automatically generated with medium confidence A diagram of different colored circles

Description automatically generated

Interestingly, seemingly no matter how high I make C (even up to 100 000), I can’t get the blue region to nudge up closer to the black line.

**Nonlinear Classifier**

So if we have a circular boundary situation, like in the right side picture up top? Then what we do is map the points onto a higher dimension in such a way that they are separable by a plane again. For instance, if we mapped all of our (x1,x2) points into the third dimension (x1, x2, z) via the formula,



you can see that the red dots would lie on a paraboloid, and the stars would form a ring around them at a higher z elevation. So then these points could be separated by a horizontal plane. Here’s a crappy picture,

A diagram of a cone with points and stars

Description automatically generated

I think such a mapping is called the *kernal trick*. I guess I won’t bother trying to figure out how this mapping is determined in general. Should also point out that with kernels, one can have multiple disconnected regions where the class may be, say, +1. I think we are still trying to minimize the Hinge Loss function:



**Exploring the Nonlinear Model and Hyperparameters**

There are a couple ways to create a nonlinear mapping to the higher dimensional space. So that’s what we’ll talk about first.

**Hyperparameter: kernel**

There are a couple possible kernels it can use to do the mapping trick: kernel = rbf, poly, and sigmoid. Here’s a graph of 500 points, a quadratic boundary, and no outliers.

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a diagram of a curve

Description automatically generated with medium confidence

You’d think kernel = poly would do the best with polynomial borders, like the quadratic one I have, but I guess not. And I’ve heard that kernel = rbf general works best for curvy borders.

Here’s pure three class case,

A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of dots

Description automatically generated with medium confidence A diagram of a number of dots

Description automatically generated with medium confidence

Now I’m putting in 10% outliers in the quadratic case,

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated with medium confidence

And 15% outliers in three-class circles case,

A diagram of a number of dots

Description automatically generated with medium confidence A diagram of different colored circles

Description automatically generated A diagram of a graph

Description automatically generated with medium confidence

Cleary rbf is best here.

**Hyperparameter: C**

Just like with the linear kernel, the C parameter is an option here. And it has the same role as before, governing how curvy we can make the surface, how closely it hugs the data points. Here’s an example of 500 points and no outliers. Using the rbf kernel on a quadratic boundary.

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

and the three-class circular boundary,

A diagram of a red circle with blue circles

Description automatically generated with medium confidence A diagram of a red circle with blue circles

Description automatically generated with medium confidenceA diagram of a number of dots

Description automatically generated with medium confidence

C = 1 does better in the pure case, than the higher C’s, counterintuitively. And here it is with 10% outliers. Looks pretty much the same,

A diagram of a graph

Description automatically generatedA diagram of a graph

Description automatically generatedA diagram of a graph

Description automatically generated

and 15% outliers in the circle three-class case,

A diagram of a number of dots

Description automatically generated A diagram of different colored circles

Description automatically generated A diagram of a number of dots

Description automatically generated with medium confidence

Here C = 1 didn’t even pick up on the green class.

**Hyperparameter: γ**

Besides C, we now have a gamma parameter as well. I don’t really know what it does, but it seems to function kind of like K in the KNN algorithm, but γ ~ 1/K. The higher γ is, the more likely the algorithm will be to form an additional region around clusters of individual points. Here’s a couple gammas for a pure sample of 500 points, using the rbf kernel.

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generatedA diagram of red and blue dots

Description automatically generated

and the three-class circle case,

A diagram of a number of dots

Description automatically generated A diagram of red and blue circles

Description automatically generated A diagram of a number of dots

Description automatically generated with medium confidence

And here it is with 10% outliers (binary case),

A diagram of a graph

Description automatically generated A diagram of red and blue dots

Description automatically generated A diagram of red and blue dots

Description automatically generated

and 15% outliers (trinary case)

A diagram of a number of dots

Description automatically generated A diagram of a number of dots

Description automatically generated

A diagram of a number of dots

Description automatically generated A diagram of a number of dots

Description automatically generated

So none of the usual γ values looks good for the trinary case. But if we set γ = 10, then we do get a decent result, as evinced by the last graph. So we can see that higher γ does make SVC algorithm more apt to form new groups.

**Comparison to other Models**

We were complaining that the KNN approach was to responsive to outliers (I suppose if k is low anyway) and made kind of funky decision boudaries. SVC attempts to redress this issue by focusing on kind of regressing the decision boundary(s). This makes it less prone to outliers compared to KNN.

If the distinction between the classes gets really muddied, i.e., if the classes have significant overlap, then it might be better to do logistic regression/classification? Logistic regression is better at handling outliers, I read. And it does seem designed to take them into account, as flogistic(x) never actually equals 0 or 1, just approaches those values.

**Appendix: Some Plane old Math**

Just want to consider some details of hyperplanes. First, recall that for any function f:



so grad f points in the direction of most rapidly increasing f. So since,



Then in 2D,



This is perpendicular to the plane boundary and in the direction of increasing f. And |**w**| is the magnitude of that rate. Generalization to more than 2 dimensions is straightforward. Interested in the distance of closest approach between the line and the origin. Is that b? Not quite. I’ll find minimum distance via Lagrange multipliers.



Minimizing w/r to x1, and using 2λ as our Lagrange multiplier,



Then minimizing w/r to x2,



Then plugging these equations into the constraint, we can solve for λ,



and then plugging them into d2, we can get the distance,



and so,



So we can write our plane equation as:



d is signed distance of closest approach of the f = 0 line to the origin. d is positive if we go from the decision boundary f = 0 towards the negative region to get to the origin. And d is negative if we go from the decision boundary f = 0 towards the positive region to get to the origin. d = **x** gives equation for the f = 0 line. **w** gives the magnitude and direction of rate of fastest increase of f. So there are three independent variables, as should be, in our present case. Seems like |**w**| would be an extraneous variable though, for our present purposes? Well no actually. Larger |**w**| could minimize the Hinge loss because |f| > 1 is necessary for loss to be zero.

What is the distance between a point and a line? I’ll use Lagrange Multipliers again. So distance between a point (x1p, x2p) and a line is:



Minimizing w/r to x1, and using 2λ as our Lagrange multiplier,



Then minimizing w/r to x2,



Then plugging these equations into the constraint, we can solve for λ,



and then plugging them into d2, we can get the distance,



and so the (unsigned) distance is:



This reduces to our previous formula when x1p = x2p = 0. So that’s good. Can easily generalize this to more than 2D:



So our algorithm would find this distance for each of the points,



and then minimize this Dp w/r to **b** and **w**. Well maybe it’d minimize Dp2 = Σdp2, rather. And note that we’d really only have one relevant d.o.f., which is d itself.