Support Vector Machines

36-350, Data Mining

19 November 2008

Contents

1 Why We Want to Use Many Nonlinear Features, But Don't Actually Want to Calculate Them

Neural networks work by creating new features in the hidden layer. This, as has been repeatedly emphasized, is one of the keys to successful machine learning: finding the right transformations of the input makes the problem easy in the new features. For example, looking at the data in Figure 1 in Cartesian (rectangular) coordinates, the problem of classifying points as + or − is moderately hard — there is in fact no linear classifier which does better than chance at this. On the other hand, if I have the wit to guess the new features $\rho = \sqrt{x_1^2 + x_2^2}$ and $\theta = \arctan x_2/x_1$, then I get a dead-simple classification problem, where perfect linear separation is easy (Figure 2).

In this example, we kept the number of features the same, but it's often useful to create more new features than we had originally. Figure 3 shows a one-dimensional classification problem which has no linear solution: the class is negative if x is below some threshold *or* above another threshold, but not both.

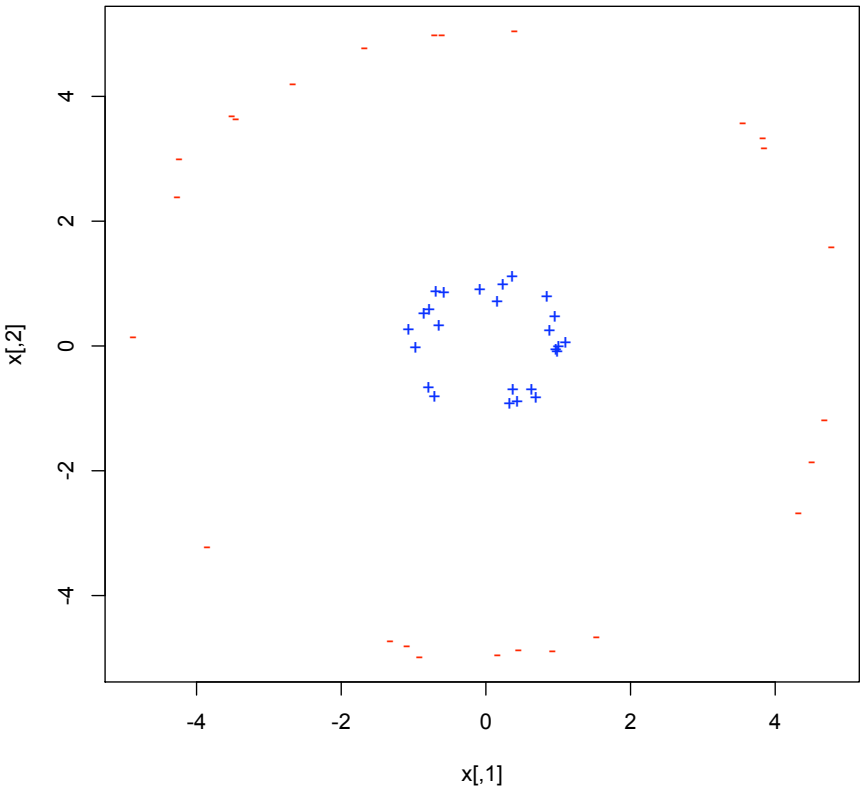

Figure 1: A classification problem with no linear solution. No linear classifier here will do better than chance.

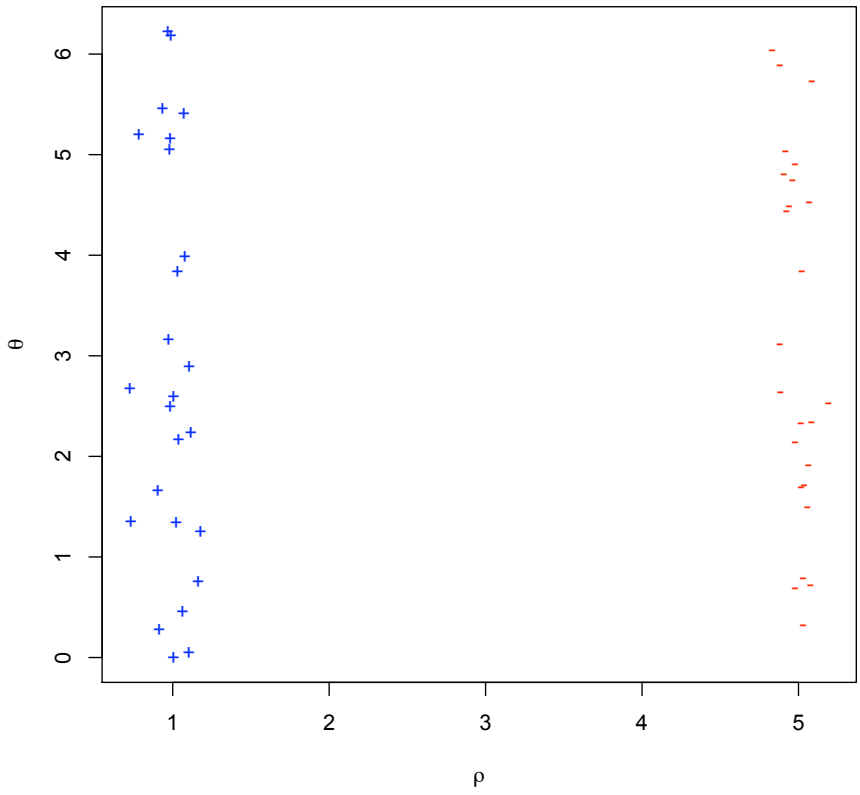

Figure 2: The same data as in Figure 1, but in polar coordinates. Perfect linear classification is easy.

Such **exclusive or** (or **XOR**) problems cannot be solved exactly by any linear method.[1]

The moral we derive from these examples (and from many others like them) is that, in order to predict well, we'd like to make use of lots and lots of nonlinear features. But we would also like to calculate quickly, and to not call the curse of dimensionality down on our heads, and both of these are hard when there are many features.

Support vector machines are ways of getting the advantages of many nonlinear features without the pains. They rest on three ideas: the dual representation of linear classifiers; the kernel trick; and margin bounds on generalization. The dual representation is a way of writing a linear classifier not in terms weights w_j, $j \in 1:p$ over features, but rather in terms of weights α_i, $i \in 1:n$ over training vectors. The kernel trick is a way of implicitly using many, even infinitely many, new, nonlinear features without actually having to calculate them. Finally, margin bounds guarantee that kernel-based classifiers with large margins will continue to classify with low error on new data, and so give as an excuse for optimizing the margin, which is easy.

Notation Input consists of p-dimensional vectors $\vec{X}$. We'll consider just two classes, $Y = +1$ and $Y = -1$. R (for "radius") will be the maximum magnitude of the n training vectors, $R \equiv \max_{1 \leq i \leq n} \|\vec{x}_i\|$.

[1] This fact was discovered by Marvin Minsky and Seymour Papert in the late 1960s; together with the recognition that perceptrons could only learn to do linear classification, it effectively killed off neural networks for several decades. This is an interesting example of people doing good, sound mathematical research, and a field paying attention to the results, with, arguably, highly counter-productive results: it took a surprisingly long time for people to investigate whether multi-layer neural networks could automatically induce useful extra features, and so solve XOR problems.

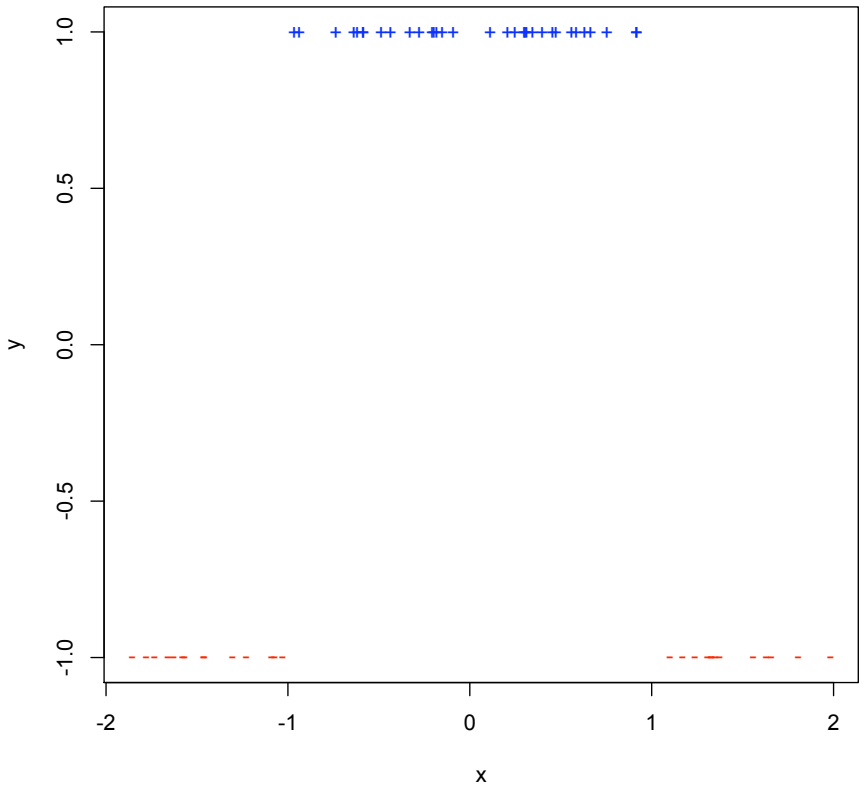

Figure 3: Another classification problem with no linear solution. Here the Y axis just shows the classes (also marked by color and symbol); it is the output to be predicted, rather than an available input feature. Y appears to be negative either if X is too big or it is too small; such **exclusive or** (or **XOR**) problems have no linear solution in the original features.

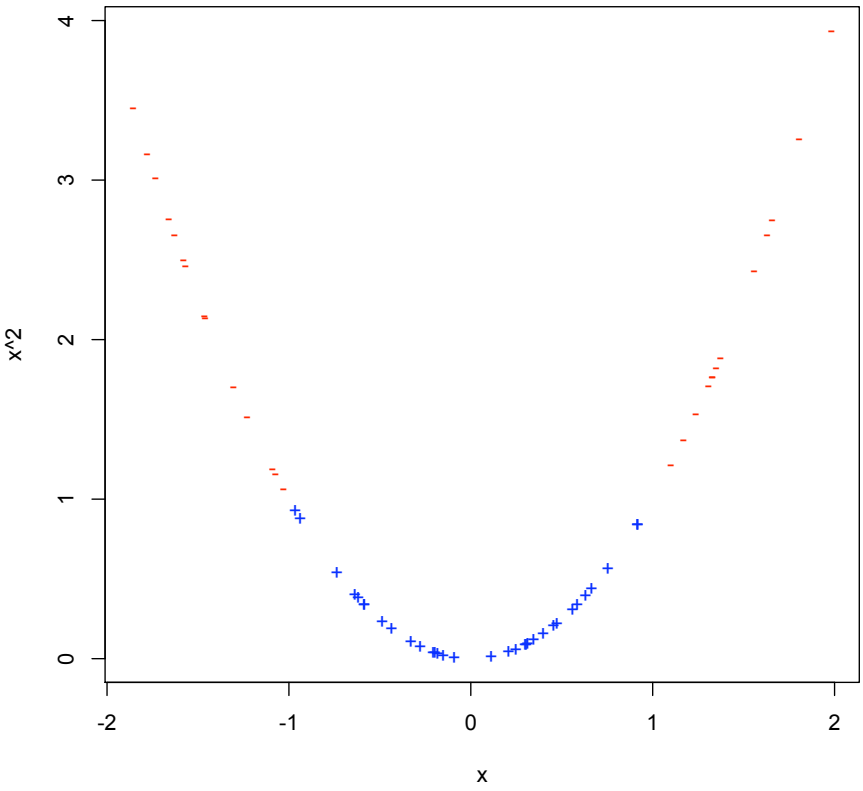

Figure 4: The same data as in Figure 3, but adding a nonlinear (quadratic) feature, namely x^2. The classes are now clearly linearly separable.

2 Dual Representation and Support Vectors

Recall that a linear classifier predicts $\hat{Y}(\vec{x}) = \operatorname{sgn} b + \vec{x} \cdot \vec{w}$. That is, it assumes that the data can be separated by the plane with normal vector $\vec{w}$, offset a distance b from the origin. We have been looking at the problem of learning linear classifiers as the problem of selecting good weights $\vec{w}$ for *input features*. This is called the **primal representation**, and we've seen several ways to do it — the prototype method, the perceptron algorithm, logistic regression, etc.

The weights $\vec{w}$ in the primal representation are weights on the features, and functions of the training vectors $\vec{x}_i$. A **dual representation** gives weights to the *training vectors*, which are (implicitly) functions of the features. That is, the classifier predicts

$$\hat{Y}(\vec{x}) = \operatorname{sgn} \beta + \sum_{i=1}^{n} \alpha_i y_i \left(\vec{x}_i \cdot \vec{x}\right) \tag{1}$$

where α_i are now weights over the training data. We can always find such dual representations when $\vec{w}$ is a linear function of the vectors, as in the perceptron or the prototype method. But we could also use them directly.

(The perceptron algorithm can be run in the dual representation. Start with $\beta = 0$, $\vec{\alpha} = 0$. Go over the training vectors; if $\vec{x}_i$ is mis-classified, increase α_i by 1, and set $\beta \leftarrow \beta + y_i R^2$. If any training vector was mis-classified, repeat the loop; exit when there are no mis-classifications.)

There are a couple of things to notice about dual representations like equation 1.

1. We need to learn the n weights in $\vec{\alpha}$, not the p weights in $\vec{w}$. This can help when $p \gg n$.
2. The training vector $\vec{x}_i$ appears in the prediction function only in the form of its inner product with the text vector $\vec{x}$, $\vec{x}_i \cdot \vec{x} = \sum_{j=1}^{p} x_{ij} x_j$.
3. We can have $\alpha_i = 0$ for some i. If $\alpha_i \neq 0$, then $\vec{x}_i$ is a **support vector**. The fewer support vectors there are, the more **sparse** the solution is.

The first two attributes of the dual representation play in to the kernel trick. The third, unsurprisingly, turns up in the support vector machine.

3 The Kernel Trick

We've mentioned several times that linear models can get more power if instead of working directly with the input features $\vec{x}$, one first calculates new, nonlinear features $\phi_1(\vec{x}), \phi_2(\vec{x}), \ldots \phi_q(\vec{x})$ from the input. Together, these features form a vector, $\phi(\vec{x})$. One then uses linear methods on the derived feature-vector $\phi(\vec{x})$. To do polynomial classification, for example, we'd make the functions all the powers and combinations of powers of the input features up to some maximum

order d, which would involve $q = \binom{p+d}{d}$ derived features. Once we have them, though, we can do linear classification in terms of the new features.

There are three difficulties with this approach; the kernel trick solves two of them.

1. We need to construct useful features. Not just anything will do, and most functions are actively bad.
2. The number of features may be very large. (With order-d polynomials, the number of features goes roughly as d^p.) Even just calculating all the new features can take a long time, as can doing anything with them.
3. In the primal representation, each derived feature has a new weight we need to estimate, so we seem doomed to suffer the curse of dimensionality.

The only thing to be done for (1) is to actually study the problem at hand, use what's known about it, and experiment. Items (2) and (3) however have a computational solution.

Remember, in the dual representation, training vectors only appear via their inner products with the test vector. If we are working with the new features, this means that the classifier can be written

$$\hat{Y}(\vec{x}) = \operatorname{sgn}\beta + \sum_{i=1}^{n} \alpha_i y_i \phi(\vec{x}_i) \cdot \phi(\vec{x}) \tag{2}$$

$$= \operatorname{sgn}\beta + \sum_{i=1}^{n} \alpha_i y_i \sum_{j=1}^{q} \phi_j(\vec{x_i})\phi_j(\vec{x}) \tag{3}$$

$$= \operatorname{sgn}\beta + \sum_{i=1}^{n} \alpha_i y_i K_\phi(\vec{x}_i, \vec{x}) \tag{4}$$

where the last line defines K, a (nonlinear) function of $\vec{x}_i$ and $\vec{x}$:

$$K_\phi(\vec{x}_i, \vec{x}) \equiv \sum_{j=1}^{q} \phi_j(\vec{x_i})\phi_j(\vec{x}) \tag{5}$$

K_ϕ is the **kernel**[2] corresponding to the features ϕ. Any classifier of the same form as equation 4 is a **kernel classifier**.

The thing to notice about kernel classifiers is that the actual features matter for the prediction only to the extent that they go into computing the kernel K_ϕ. If we can find a short-cut to get K_ϕ without computing all the features, we don't actually need the latter.

To see that this is possible, consider the expression $(\vec{x} \cdot \vec{z} + 1/\sqrt{2})^2 - 1/2$. A little algebra (EXERCISE: do the algebra!) shows that

$$(\vec{x} \cdot \vec{z} + 1/\sqrt{2})^2 = \sum_{j=1}^{p}\sum_{k=1}^{p}(x_j x_k)(z_j z_k) + \sum_{j=1}^{p} x_j z_j \tag{6}$$

[2]This sense of the word "kernel" is distinct from the one used in kernel smoothing. Both ultimately derive from the idea of a kernel in abstract algebra. While this is confusing, it's nowhere near as bad as the ambiguity of "normal".

which is to say, it's the kernel for the ϕ which takes the input features to all quadratic (second-order polynomial) functions of the input features. By taking $(\vec{x} \cdot \vec{z} + c)^d$, we can evaluate the kernel for polynomials of order d, without having to actually compute all the polynomials.[3]

In fact, we do not even have to define the features explicitly. The kernel is the dot product (a.k.a. inner product) on the derived feature space, which says how similar two feature vectors are. We really only care about similarities, so we can get away with any function K which is a reasonable similarity measure. The following theorem will not be proved here, but justifies just thinking about the kernel, and leaving the features implicit.

Theorem 1 (Mercer's Theorem) *If $K_\phi(\vec{x}, \vec{z})$ is the kernel for a feature mapping ϕ, then for any finite set of vectors $\vec{x}_1, \ldots \vec{x}_m$, the $m \times m$ matrix $K_{ij} = K_\phi(\vec{x}_i, \vec{x}_j)$ is symmetric, and all its eigenvalues are non-negative. Conversely, if $K(\vec{x}, \vec{z})$ has this property, then there is some feature mapping for which K is the kernel.*

So long as a kernel function K behaves like an inner product should, it *is* an inner product, on some feature space, albeit possibly a weird one. (Often the feature space guaranteed by Mercer's theorem is an infinite-dimensional one.) The moral is thus to worry about K, and forget about ϕ. Insight into the problem and background knowledge should go into building the kernel. This can be simplified by the fact (which we also will not prove) that sums and products of kernel functions are also kernel functions.

The advantages of the kernel trick are that (1) we get to implicitly use many nonlinear features of the data, without wasting time having to compute them; and (2) by combining the kernel with the dual representation, we need to learn only n weights, rather than one weight for each new feature. We can even hope that the weights are sparse, so that we really only have to learn a few of them.

Kernelization Closely examining linear models shows that almost everything they do with training vectors involves only inner products, $\vec{x}_i \cdot \vec{x}$ or $\vec{x}_i \cdot \vec{x}_j$. These inner products can be replaced by kernels, $K(\vec{x}_i, \vec{x})$ or $K(\vec{x}_i, \vec{x}_j)$. Making this substitution throughout gives the **kernelized** version of the linear procedure. Thus in addition to kernel classifiers (= kernelized linear classifiers), there is kernelized regression, kernelized principal components, etc. For instance, in kernelized regression, we try to approximate the conditional mean by a function of the form $\sum_{i=1}^{n} \alpha_i K(\vec{x}_i, \vec{x})$, where we pick α_i to minimize the residual sum of squares. In the interest of time I am not going to follow through any of the math.

An Example: The Gaussian/Radial Kernel The Gaussian density function $\frac{1}{\sqrt{2\pi\sigma^2}} \exp -\|\vec{x} - \vec{x}\|^2/2\sigma^2$ is a valid kernel. In fact, from the series expansion $e^u = \sum_{n=0}^{\infty} \frac{u^n}{n!}$, we can see that the implicit feature space of the Gaussian

[3] Changing the constant c changes the weights assigned to higher-order versus lower-order derived features.

kernel includes polynomials of *all* orders, though it gives less and less weight to higher and higher order polynomials. When working with SVMs, the kernel is typically written as $K(\vec{x}, \vec{z}) = \exp -\gamma \|\vec{x} - \vec{z}\|^2$, so that the normalizing constant of the Gaussian density is absorbed into the dual weights, and $\gamma = 1/2\sigma^2$. (Everyone writes the scale factor as γ, but it should not be confused with the margin, which everyone writes as γ.) Basically this just absorbs the normalizing constant of the Gaussian density into the dual weights. This is sometimes also called the **radial** kernel. The weighting factor γ (or, equivalently, σ) is a control setting. A typical default is $\gamma = 1/p$, p being the number of input features. If you are worried that it matters, you can always cross-validate.

4 Margin Bounds

To recap, once we fix a kernel function K, our kernel classifier has the form

$$\hat{Y}(\vec{x}) = \text{sgn} \sum_{i=1}^{n} \alpha_i y_i K(\vec{x}_i, \vec{x}) \tag{7}$$

and the learning problem is just finding the n dual weights α_i. There are several ways we could do this.

1. *Kernelize/steal a linear algorithm* Take any learning procedure for linear classifiers; write it so it only involves inner products; then run it, substituting K for the inner product throughout. This would give us a kernelized perceptron algorithm, for instance.

2. *Direct optimization* Treat the in-sample error rate, or maybe the cross-validated error rate, as an objective function, and try to optimize the α_i by means of some general-purpose optimization method. This is tricky, since the sign function means that the error rate depend discontinuously on $\vec{\alpha}$ in any one sample, and changing predictions on one training point may mess up other points. This sort of discrete, inter-dependent optimization problem is generically *very hard*, and best avoided.

3. *Optimize something else* Since what we really care about is the generalization to new data, find a formula which tells us how well the classifier will generalize, and optimize that. Or, less ambitiously, find a formula which puts an *upper bound* on the generalization error, and make that upper bound as small as possible.

The margin-bounds idea consists of variations on the third approach.

Recall that for a (primal-form) linear classifier $\vec{w}$, the **margin** of a point $\vec{x}_i, y_i$ is

$$\gamma_i = y_i \left(\frac{b}{\|\vec{w}\|} + \vec{x}_i \cdot \frac{\vec{w}}{\|\vec{w}\|} \right) \tag{8}$$

This quantity is positive if the point is correctly classified. It shows the "margin of safety" in the classification, i.e., how far the input vector would have to move

before the predicted classification flipped. (This is the **geometric** margin, as distinct from the **functional** margin, which is the geometric margin multiplied by $\|\vec{w}\|$.) The over-all margin of the classifier is $\gamma = \min_i \gamma_i$.

We saw that the perceptron algorithm converged rapidly, with few training mistakes, when the margin was large (compared to the radius R). Large-margin linear classifiers also have good generalization performance. The basic reason is that the *range* of planes which manage to separate the data with a large margin is much smaller than the range of planes which separate the classes with only a small margin. The margin thus effectively controls the capacity: high margin means small capacity, and small capacity means that the risk of over-fitting is low. I will now quote three specific results for linear classifiers, presented without proof.

Theorem 2 (Margin bound for perfect separation) *Suppose the data come from a distribution where $\|\vec{X}\| \leq R$. Fix any positive γ. If a linear classifier correct classifies all n training examples, with a geometric margin of at least γ, then with probability at least $1 - \delta$, its error rate on new data from the same distribution is at most*

$$\varepsilon = \frac{2}{n}\left(\frac{64R^2}{\gamma^2}\ln\frac{en\gamma}{8R^2}\ln\frac{32n}{\gamma^2} + \ln\frac{4}{\delta}\right) \tag{9}$$

if $n > \min 64R^2/\gamma^2, 2/\varepsilon$.

(Source: (Cristianini and Shawe-Taylor, 2000, Theorem 4.18).) Notice that the promised error rate gets larger and larger as the margin shrinks. This suggests that what we want to do is maximize the margin (since R^2, n, 64, etc., are beyond our control).

The next result applies to imperfect classifiers. Fix a minimum margin γ_0, and define the **slack** ζ_i of each data point as

$$\zeta_i(\gamma_0) = \max 0, \gamma_0 - \gamma_i \tag{10}$$

That is, the slack is the amount by which the margin falls short of γ_0, if it does. If the separation is imperfect, some of the γ_i will be negative, and the slack variables at those points will be $> \gamma_0$.

Theorem 3 (Soft margin bound/slack bound for imperfect separation) *Suppose the data come from a distribution where $\|\vec{X}\| \leq R$. Suppose a linear classifier achieves a margin γ on n samples, with slacks $\vec{\zeta}(\gamma) = (\zeta_1(\gamma), \zeta_2(\gamma), \ldots \zeta_n(\gamma))$. Then with probability at least $1 - \delta$, its error rate on new data is at most*

$$\varepsilon = \frac{c}{n}\left(\frac{R^2 + \|\vec{\zeta}(\gamma)\|^2}{\gamma^2}\ln^2 n - \ln\delta\right) \tag{11}$$

for some positive constant c.

(Source: (Cristianini and Shawe-Taylor, 2000, Theorem 4.22).) This suggests that the quantity to optimize is the ratio $\frac{R^2+\|\vec{\zeta}\|^2}{\gamma^2}$. (Notice that if we can set all the slacks to zero, because there's perfect classification with some positive margin, then we get a bound that looks like, but isn't quite, the perfect-separation bound again.)

A final result does not assume we are using linear classifiers, but rather relies on being able to ignore part of the data.

Theorem 4 (Compression/sparseness bound) *Take any classifier-learning algorithm which is trained on a set of n samples. Suppose that the* same *classifier would be returned on a sub-set of the training data with only m samples. If this classifies the n training points perfectly, then with probability at least $1-\delta$, the error rate on new data is at most*

$$\varepsilon = \frac{1}{n-m}\left(m \ln \frac{en}{m} + \ln \frac{n}{\delta}\right) \tag{12}$$

(Source: (Cristianini and Shawe-Taylor, 2000, Theorem 4.25).) The argument here is simple enough to sketch: the learning algorithm really only uses m data points, but still manages to get the remaining $n-m$ training points right. if the actual error rate is ε (or more), then the probability of doing this would be at most $(1-\varepsilon)^{n-m} \leq e^{-\varepsilon(n-m)}$. However, there is more than one way of picking m points out of a training set of size n, so we need to be sure we didn't just get lucky. The number of subset choices is in fact $\binom{n}{m}$, so the probability that the generalization error rate is ε or more is at most $\binom{n}{m}e^{-\varepsilon(n-m)}$. Set this equal to δ and solve for ε. — This argument can be modified to handle situations where the training data re not perfectly classified, but the result is a lot messier[4].

All of this carries over to kernel classifiers, since they are just linear classifiers in a new feature space.[5] We simply have to re-define the margins of the training points in terms of the kernel and the dual representation:

$$\gamma_i = y_i \left(\frac{\beta}{\sqrt{\sum_{i=1}^n \alpha_i}} + \frac{1}{\sqrt{\sum_{i=1}^n \alpha_i}} \sum_{j=1}^n \alpha_j K(\vec{x}_j, \vec{x}_i) \right) \tag{13}$$

5 The Support Vector Machine

The three bounds suggest three strategies for learning kernel classifiers:

1. Maximize the margin γ.
2. Minimize the soft margin bound $(R^2 + \|\vec{\zeta}\|^2)/\gamma^2$.
3. Minimize the number of support vectors.

[4] One could, for instance, try to use Hoeffding's inequality to argue that it's really unlikely that the actual error rate is much larger than the observed error rate when $n-m$ is large.

[5] It's important that the kernel K be fixed in advance of looking at the data. If instead it was found by some kind of adaptive search over possible kernels, that would increase the capacity, and we'd need different bounds.

5.1 Maximum Margin SVMs

Maximizing the margin directly turns out to be less than favorable, computationally, than maximizing a related function. (I will not go into the details.) In brief, the procedure is to maximize

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j K(\vec{x}_i, \vec{x}_j) \tag{14}$$

with the constraints that $\alpha_i \geq 0$ and that $\sum_{i=1}^{n} y_i \alpha_i = 0$. (We enforce these constraints through Lagrange multipliers.) Having found the maximizing α_i, the off-set constant β comes from

$$\beta = -\frac{1}{2} \left(\max_{i:\ y_i=-1} \left(\sum_{j=1}^{n} y_j \alpha_j K(\vec{x}_j, \vec{x}_i) \right) + \min_{i:\ y_i=+1} \left(\sum_{j=1}^{n} y_j \alpha_j K(\vec{x}_j, \vec{x}_i) \right) \right) \tag{15}$$

In other words, β is chosen to balance mid-way between the most nearly positive negative points and the most nearly negative positive points, thereby maximizing the margin in the implicit feature space. The geometric margin in feature space is $\gamma = 1/\sqrt{\sum_{i=1}^{n} \alpha_i}$.

If the maximum margin classifier correctly separates the training data, we can apply the first margin bound on the generalization error.

Generally speaking, α_i will be zero for most training points; the ones for which it isn't are (again) the **support vectors**. These turn out to be the only points which matter: notice that if $\alpha_i = 0$, we could remove that data point altogether without affecting the minimum-value solution of (14). This means that we can also apply the sparseness/compression bound on generalization error, with $m =$ the number of support vectors. Because maximum margin solutions are typically quite sparse, it is not common to try to minimize the number of support vectors directly. (Attempting to do so would seem to get us back to difficult discrete optimization problems, rather than easy, smooth continuous optimization.)

5.2 Soft Margin Maximization

We fix a positive constant C and maximize

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \left(K(\vec{x}_i, \vec{x}_j) + \lambda \delta_{ij} \right) \tag{16}$$

with the constraints $\alpha_i \geq 0$, $\sum_{i=1}^{n} y_i \alpha_i = 0$. The off-set constant β has to solve

$$y_i \beta + y_i \sum_{j=1}^{n} y_j \alpha_j K(\vec{x}_j, \vec{x}_i) = 1 - \lambda \alpha_i \tag{17}$$

for each i where $\alpha_i \neq 0$. Pick one of them and solve for β:

$$\beta = \frac{1 - \lambda\alpha_i}{y_i} - \sum_{j=1}^{n} y_j \alpha_j K(\vec{x}_j, \vec{x}_i) \tag{18}$$

The geometric margin is $\gamma = 1/\sqrt{\sum_{i=1} n\alpha_i - \lambda\|\vec{\alpha}\|^2}$, and the slacks are $\zeta_i = \lambda\alpha_i$.

The constant λ here is a tuning parameter, basically controlling the trade-off between wanting a large margin and wanting small slacks. Typically, it would be chosen by cross-validation.

6 R

There are several packages which can implement SVMs. The oddly-named `e1071` library has an `svm` function which will take `lm`-style formulas, and can do either classification or regression with several different kernels. The `svmpath` library will actually fit SVMs over a whole range of λ values simultaneously; but you have to pick a particular λ for prediction. (It also doesn't take a formula argument or allow you to do regression.)

For instance, the data for Figure 1 live in a frame called, imaginatively, `rings`, with columns named `x1` (real numbers), `x2` (real numbers) and `y` (categorical values, i.e. "factors" in R). To fit an SVM to this data, use

```
rings.svm = svm(y ~ ., data=rings)
```

The formula `y ~ .` means "include every variable in the frame other than the response in the model".[6] Since the response variable is a factor, the `svm` function defaults to doing classification — if I'd made y have the numerical values $+1$ and -1, `svm` would default to regression, and I'd have to give an extra `type` argument to the function to tell it to classify. (See `help(svm)`.) Other defaults include the kernel (here the Gaussian/radial kernel) and the value of λ (the `svm` function uses a `cost` argument $= 1/\lambda$, defaulting to 1). Because this is such a simple problem, the defaults work perfectly.

```
> sum(predict(rings.svm) != rings$y)
[1] 0
```

We can plot the classifier as well (Figure 5).

```
plot(rings.svm,data=my.frame,color.palette=topo.colors,
    svSymbol="S",dataSymbol="x")
```

The `plot` function for `svm` objects requires a data set, and with more than two input features it would need extra argument saying which variables to plot, and what values to impose on the others. (I don't like the default color or symbol

[6] If you know regular expressions, remember that `.` matches any character. If you don't know regular expressions, don't worry about it.

choices, so I changed them.) In the plot, you'll see an `S` for the data points which are support vectors, and an `x` for the others. You can also see the boundary between the two classes. Note that the support vectors in each class are closer to the boundary than the other vectors in that class, as they should be.

Further Reading

The best starting book on support vector machines, which I've ~~ripped off~~ drawn on heavily is Cristianini and Shawe-Taylor (2000). A more thorough account of SVMs and related methods can be had in Herbrich (2002). SVMs were invented by Vapnik and collaborators, and are, so to speak, the poster-children for the value of statistical learning theory in machine learning and data mining; Vapnik (2000) is *strongly* recommended, but remember that you're reading the pronouncements of an opinionated and irascible genius.

References

Cristianini, Nello and John Shawe-Taylor (2000). *An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods*. Cambridge, England: Cambridge University Press.

Herbrich, Ralf (2002). *Learning Kernel Classifiers: Theory and Algorithms*. Cambridge, Massachusetts: MIT Press.

Vapnik, Vladimir N. (2000). *The Nature of Statistical Learning Theory*. Berlin: Springer-Verlag, 2nd edn.

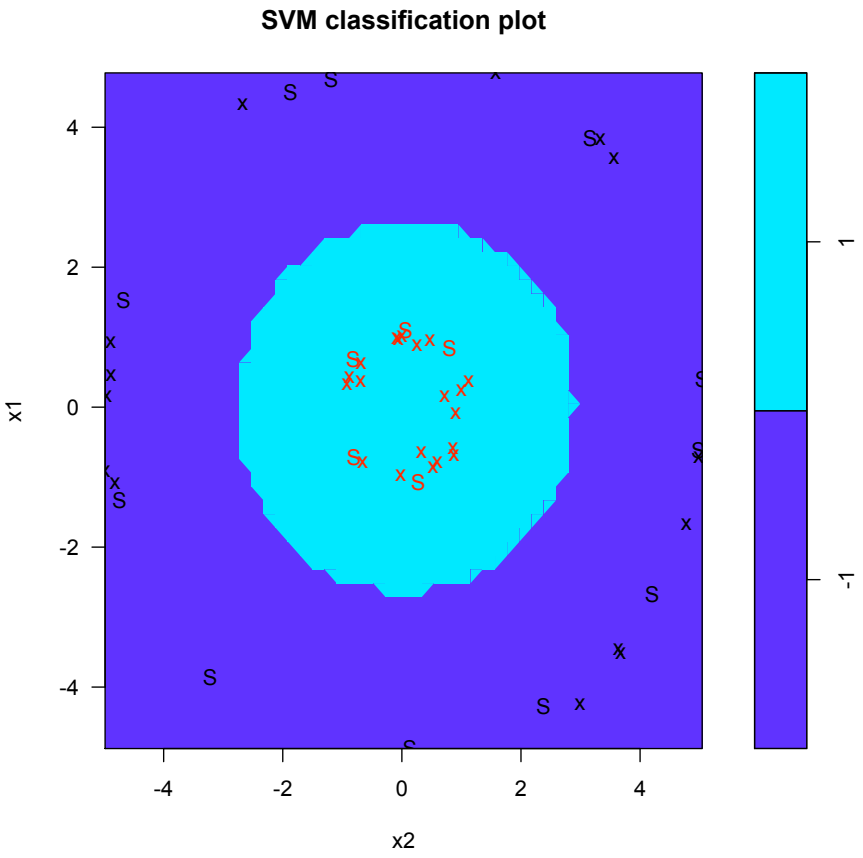

Figure 5: SVM classifier learned from the data in Figure 1 with a Gaussian (radial) kernel. Letters indicate the training data points — support vectors are marked with `S`, others with `x`. Colors shows the classification of training points (all correct), and the inferred boundary between the classes. I was not able to persuade it to plot x_1 on the horizontal axis.