Extending Linear Methods with Nonlinear Features (Lectures 15 and 16)

Kernel Methods, Support Vector Machines, and Random Features / Random Kitchen Sinks

36-462/662, Spring 2022

15 and 17 March 2022

Contents

1 Notation

$\vec{X}$ will be the p-dimensional vector of features we're using for our predictions.

R (for "radius") will be the maximum magnitude of the n training vectors, $R \equiv \max_{1 \leq i \leq n} \|\vec{x}_i\|$.

Y will be the class we want to predict, usually by not always binary, 0/1.

When we are dealing with binary classes, $Z = 2Y - 1$. That is $Z = +1$ when $Y = 1$, but $Z = -1$ when $Y = 0$. The Z variable is of course redundant when we have Y, but it simplify a lot of the formulas.

$\mathbb{I}\left\{A\right\}$ is 1 when the expression A is true, and 0 otherwise. Similarly, $\operatorname{sgn} a$ is $+1$ when $a > 0$, -1 when $a < 0$, and 0 when $a = 0$.

2 Linear classifiers (recap)

We say that we have a linear classifier, with feature-weights $\vec{w}$ and offset b, when we predict

$$\hat{Y}(\vec{x}_0) = \mathbb{I}\left\{b + \vec{w} \cdot \vec{x}_0 \geq 0\right\}$$

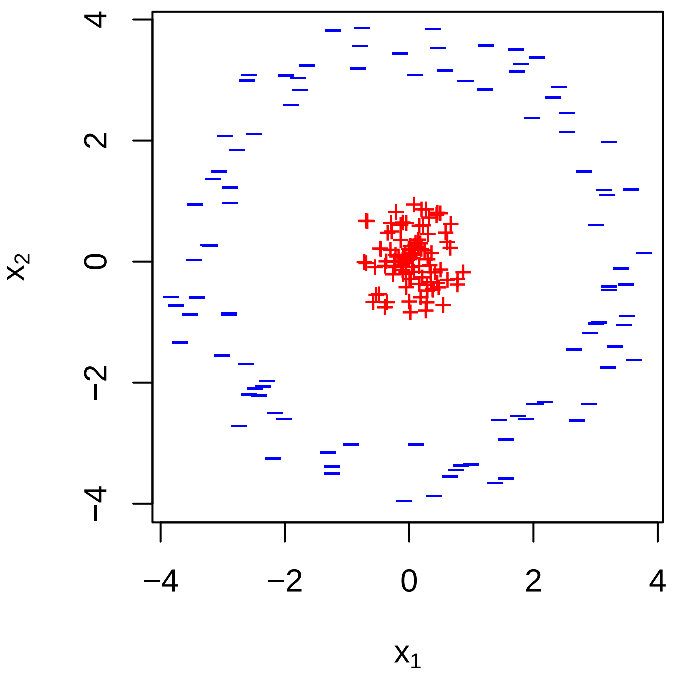
or equivalently

$$\hat{Z}(\vec{x}_0) = \operatorname{sgn}\left(b + \vec{w} \cdot \vec{x}_0\right)$$

As usual, $\vec{x}_0$ is a totally arbitrary point in the feature space. It may or may not have been one of the training points, where we observed the corresponding class.

3 We want to use many nonlinear features, but don't actually want to calculate them all

We've seen some examples of building linear classifiers, and seen that they *can* work well. But there are also situations where they just don't work at all. For example, looking at the data in the next figure[1].



If we use the given features x_1 and x_2, classifying points as $+$ or $-$ is a bit hard. There is no linear classifier here which will work perfectly.[2] The prototype method will fail dismally here, doing no better than guessing at random[3]. You can find a line which does better than guessing at random[4], but not much better.

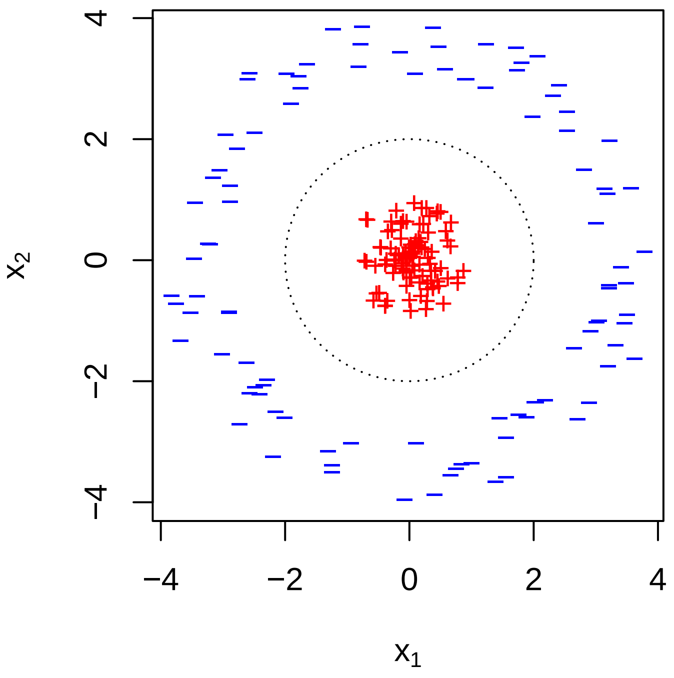
Now of course separating these two classes is trivial if we can use a *nonlinear* boundary, like a circle:

[1]I realize, no one cares about classifying two-dimensional data that looks like a badly drawn bulls-eye. But think of it as a cartoon for a situation like "healthy patients have all their features near some normal values, while sick patients can depart from the normal range in many different directions".

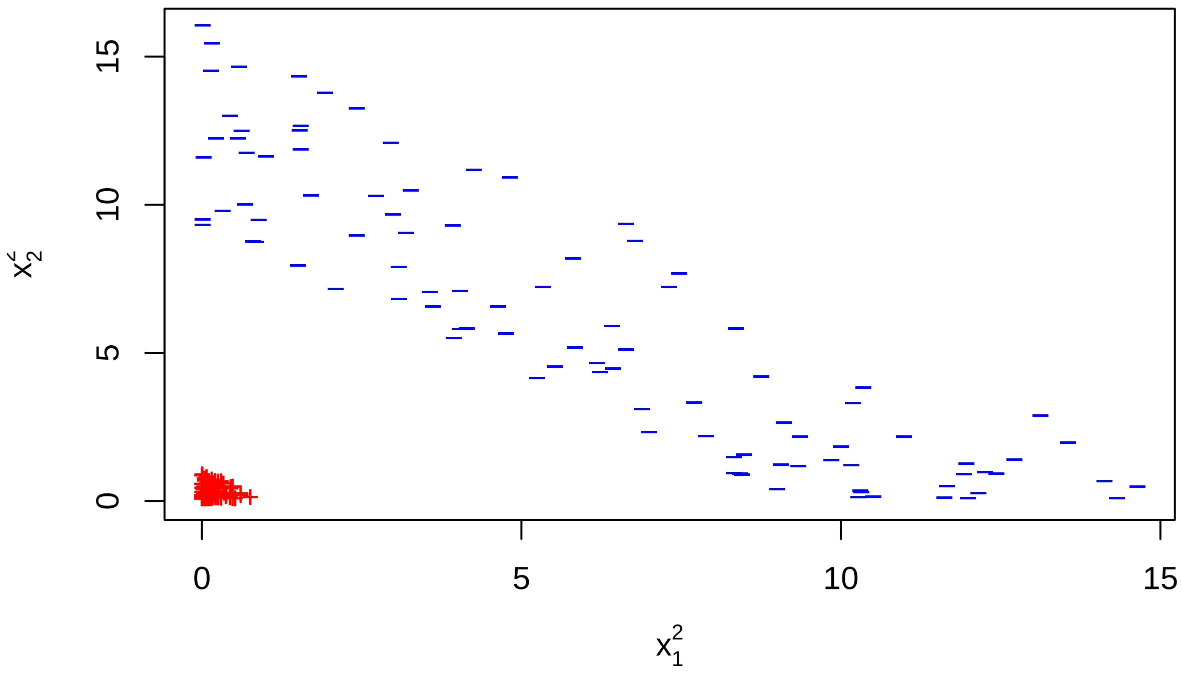
[2]Either all of the $-$ points are on the same side of the line, or there are some $-$ points on both sides of the line. But the $-$ points enclose the $+$ points, so if *all* the $-$ points are on one side of the line, all of the $+$ points are on the *same* side of the line, and the line doesn't separate them. If some $-$ points are on one side of the line and other $-$ points are on the opposite side, then the line cannot perfectly separate the $+$ points from the $-$ points.

[3]Where are the prototypes? Why does their location tell us that the prototype method won't work?

[4]Can you draw one, without doing any calculations? (How do you know it does better than random?)

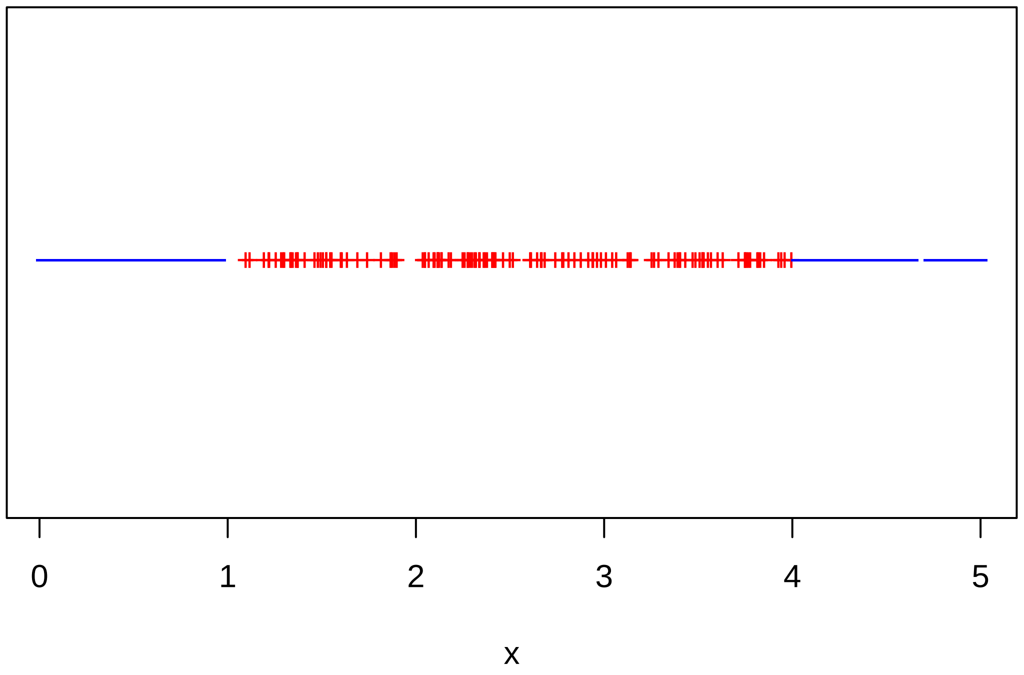


We *could* start thinking about how we would start drawing such boundaries directly, but an alternative strategy has proved more useful. This is to **transform** the original features, nonlinearly, and then do linear classification in the new, transformed or **derived** features. To see how this can work, suppose we did the very simple thing of looking at the *squares* of the original features, x_1^2 and x_2^2.



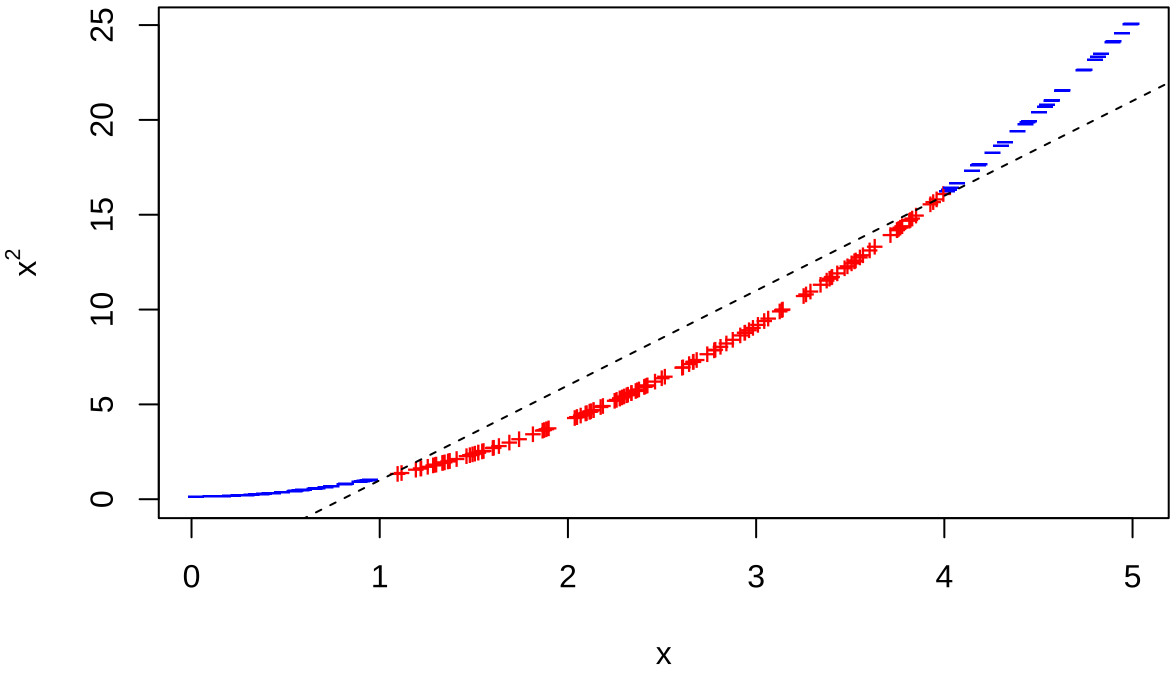
(Of course, the squares of the original features aren't the only derived features we could use. Switching to polar coordinates ($\rho = \sqrt{x_1^2 + x_2^2}$ and $\theta = \arctan x_2/x_1$) would do just as nicely.)

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. This next figure shows a one-dimensional classification problem which also has no linear solution, since the class is negative if x is below one threshold, or above another.



Problems like this, where one of the original features must be either in one range or another ("exclusive-or" or "XOR" problems) cannot be solved exactly by any linear method[5].

Adding a second feature like x^2 makes it easy to linearly separate the classes:



The moral we derive from these examples (and from many others like them) is that, in order to predict well,

[5]This fact was discovered by the pioneering AI researchers Marvin Minsky and Seymour Papert in the 1960s (Minsky and Papert 1969). This effectively killed off the field of neural networks (or, as it was called, "perceptrons") for several decades, because the two-layer networks commonly used then could only learn to do linear classification. This is an interesting example of brilliant scientists doing sound research, and a field paying attention to the results, with, arguably, highly counter-productive consequences: it took a long time for anyone to investigate whether neural networks with more than two layers could solve XOR problems by (implicitly) creating new features. (When that work did happen, in the 1980s, a lot of it was done in the CMU psychology department.)

we'd like to make use of lots and lots of nonlinear features. But we would also like to calculate quickly, and to not overfit all the time, 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 some of the pain 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 of weights over features, w_j, $j \in 1 : p$, but rather in terms of weights over training vectors, α_i, $i \in 1 : n$. 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.

4 Dual Representation and Support Vectors

Recall that a linear classifier predicts $\hat{Y}(\vec{x}) = \mathbb{I}\{b + \vec{x} \cdot \vec{w} \geq 0\}$. That is, it hopes that the data can be separated by the plane with normal (perpendicular) 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, 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*. That is, the classifier predicts

$$\hat{Y}(\vec{x}) = \mathbb{I}\left\{b + \sum_{i=1}^{n} \alpha_i \left(\vec{x}_i \cdot \vec{x}\right) \geq 0\right\}$$

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 prototype method or the perceptron algorithm[6], or if we have more data points than features[7]. But we could also search for those weights directly. We would typically expect α_i to be > 0 if i is in the positive class, and < 0 if i is in the negative class. It's sometimes convenient to incorporate this into the definition, so that we say

$$\hat{Z}(\vec{x}) = \operatorname{sgn}\left(b + \sum_{i=1}^{n} \alpha_i z_i \left(\vec{x}_i \cdot \vec{x}\right)\right)$$

and insist that $\alpha_i > 0$.

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

1. We need to learn the n weights in $\vec{\alpha}$, not the p weights in $\vec{w}$. This can help when $p \geq n$.
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.

[6]To run the perceptron algorithm in the "dual" representation, start with $b = 0$, and $\alpha_i = 0$ for all $i \in 1:n$. Go over the training vectors in order; if data point i is correctly classified, change nothing and go on to the next point; if i is mis-classified, add z_i to α_i, and set $b \leftarrow b + z_i R^2$. If any training point was mis-classified, repeat the loop; exit when there are no mis-classifications.

[7]If $n = p$, then we can use the training vectors as a basis for the vector space, so *any* vector $\vec{v}$ could be written as a weighted sum of the training vectors, $\vec{v} = \sum_{i=1}^{n} c_i \vec{x}_i$. (This presumes that the training vectors "are in general position", i.e., not all in a lower-dimensional linear subspace.) In fact, we did this in class in lecture 16. If $n > p$, we can usually express an arbitrary vector $\vec{v}$ as a weighted sum of the training vectors in multiple different ways. We are about to consider cases where p is very large and even infinite, however, so that won't apply.

5 The Kernel Trick

I'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.
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 over-fit.

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

$$\begin{aligned}
\hat{Z}(\vec{x}_0) &= \mathrm{sgn}\left(b + \sum_{i=1}^{n} \alpha_i z_i \phi(\vec{x}_i) \cdot \phi(\vec{x}_0)\right) && (1)\\
&= \mathrm{sgn}\left(b + \sum_{i=1}^{n} \alpha_i z_i \sum_{j=1}^{q} \phi_j(\vec{x}_i)\phi_j(\vec{x}_0)\right) && (2)\\
&= \mathrm{sgn}\left(b + \sum_{i=1}^{n} \alpha_i z_i K_\phi(\vec{x}_i, \vec{x}_0)\right) && (3)
\end{aligned}$$

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

$$K_\phi(\vec{x}_i, \vec{x}) \equiv \sum_{j=1}^{q} \phi_j(\vec{x}_i)\phi_j(\vec{x}) \quad (4)$$

K_ϕ is the **kernel**[8] corresponding to the features ϕ.

Any classifier of the form

$$\hat{Z}(\vec{x}_0) = \mathrm{sgn}\left(b + \sum_{i=1}^{n} \alpha_i z_i K_\phi(\vec{x}_i, \vec{x}_0)\right)$$

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 features.

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

$$(\vec{x} \cdot \vec{x}' + 1/\sqrt{2})^2 - \frac{1}{2} = \sum_{j=1}^{p}\sum_{k=1}^{p} (x_j x_k)(x'_j x'_k) + \sum_{j=1}^{p} x_j x'_j$$

[8] This sense of the word "kernel" is distinct from the one used in "kernel smoothing" and "kernel density estimation". See the last section.

This says that the left-hand side is the kernel for the ϕ which maps the input features to all quadratic (second-order polynomial) functions of the input features. By taking $(\vec{x} \cdot \vec{x}' + c)^d$, we can evaluate the kernel for polynomials of order d, without having to actually compute all the polynomials. (Changing the constant c changes the weights assigned to higher-order versus lower-order derived features.)

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.

> **Mercer's Theorem** If $K_\phi(\vec{x}, \vec{x}')$ 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 for any set of vectors $\vec{x}_1, \ldots \vec{x}_m$, the matrix formed from $K(\vec{x}_i, \vec{x}_j)$ is symmetric and has non-negative eigenvalues, then there is some feature mapping ϕ for which $K(\vec{x}, \vec{x}') = \phi(\vec{x}) \cdot \phi(\vec{x}')$.

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. (Sometimes the feature space guaranteed by Mercer's theorem is an infinite-dimensional one.) The moral is thus to worry about K, rather than ϕ. Insight into the problem and background knowledge should go into building the kernel. The fundamental job of the kernel is to say how similar two different data points are;

This can be simplified by the fact (which we also will not prove) that sums and products of kernel functions are also kernel functions.

5.0.1 An Example: The Gaussian/Radial Kernel

The Gaussian density function $\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\|\vec{x}-\vec{x}\|^2/2\sigma^2}$ 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 kernel includes polynomials of *all* orders. This means that even though we're just using one kernel function, we're implicitly using infinitely many transformed features!

When working with kernel methods, we typically[9] write $K(\vec{x}, \vec{x}') = \exp -\gamma \|\vec{x} - \vec{x}'\|^2$, so that the normalizing constant of the Gaussian density is absorbed into the dual weights, and $\gamma = 1/2\sigma^2$. This is sometimes also called the **radial** (or **radial basis** or **radial basis function**) kernel. The scale factor γ is a control setting. A typical default is γ is the median of $\|\vec{x}_i - \vec{x}_j\|^2$ over the training data (after first scaling the raw features to unit variance). If you are worried that it matters, you can always tune it by cross-validation.

5.1 Kernelization

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.

Closely examining linear regression 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

[9]Writing the (inverse) scale factor as γ is very common, but it should not be confused with using γ for the margin, which is *also* very common. Unfortunately, the second most common symbol for the inverse scale factor is σ; this σ is almost the *reciprocal* of the usual σ^2 =variance convention of statistics (except for a factor of 2).

addition to kernel classifiers (= kernelized linear classifiers), there is kernelized regression, kernelized principal components, etc.

5.2 Kernelized regression, especially kernel ridge regression

If we want to do linear regression using the features, the prediction is (as I've said *ad nauseam*) given by

$$s(x;\beta) = \sum_{j=1}^{d} \beta_j \phi_j(x)$$

If we want to do ridge regression using the features, we'd pick the coefficients β to solve the penalized least-squares problem

$$\hat{\beta} = \operatorname*{argmin}_{\beta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} (y_i - s(x;\beta))^2 + \beta^T \beta$$

In the dual view, the predictions are

$$s(x;\alpha) = \sum_{i=1}^{n} \alpha_i K(x, x_i)$$

and the corresponding optimization problem is

$$\hat{\alpha} = \operatorname*{argmin}_{\alpha \in \mathbb{R}^n} \frac{1}{n} \sum_{i=1}^{n} (y_i - s(x_i;\alpha))^2 + \lambda \alpha^T \mathbf{K} \alpha$$

(You will show this in HW 8.) That is, we are picking the weights on individual training points so as to balance a small MSE against the weights being too big. (Notice that since $\mathbf{K}$ is positive-definite, the penalty term is ≥ 0.)

The solution, which is called **kernel ridge regression**, is

$$\hat{\alpha} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y}$$

where $\mathbf{y}$ is, in the usual regression notation, the $[n \times 1]$ matrix of y_i values. (Again, you will show this in HW 8.) If you want to see an example of this in action, look at the section on R below.

If you are curious as to how this kind of "kernel regression" compares to what you might have learned as "kernel smoothing" or "kernel regression" in other courses, see the last section of these notes.

6 Margin Bounds

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

$$\hat{Z}(\vec{x}) = \mathrm{sgn}\left(b + \sum_{i=1}^{n} \alpha_i z_i K(\vec{x}_i, \vec{x})\right)$$

and the learning problem is just finding the n dual weights α_i and the off-set b. 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 indicator function means that the error rate depend discontinuously on the α's, 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 classifier, the **margin** of data-point i is its distance to the boundary of the classifier. For a (primal-form) linear classifier $(b, \vec{w})$, this is

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

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. The over-all margin of the classifier is $\gamma = \min_{i \in 1:n} \gamma_i$.

6.1 Generalization error for classifiers with zero in-sample error in terms of margin

Large-margin linear classifiers tend to generalize well to new data (from the same source). The basic reason is that there just aren't many linear surfaces which manage to separate the classes with a large margin. As we demand a higher and higher margin, the range of linear surfaces that can deliver it shrinks. The margin thus effectively controls the capacity for over-fitting: high margin means small capacity, and low risk of over-fitting. I will now quote three specific results for linear classifiers, presented without proof[10].

> **Margin bound for perfect separation** (Cristianini and Shawe-Taylor 2000, Theorem 4.18): Suppose the data come from a distribution where $\|\vec{X}\| \leq R$. Fix any number $\gamma > 0$. If a linear classifier correct classifies all n training examples, with a 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
>
> $$\varepsilon = \frac{2}{n}\left(\frac{64R^2}{\gamma^2} \ln \frac{en\gamma}{8R^2} \ln \frac{32n}{\gamma^2} + \ln \frac{4}{\delta}\right)$$
>
> if $n > \min 64R^2/\gamma^2, 2/\varepsilon$.

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).

[10]The detailed forms of the bounds are complicated, and not worth memorizing, but I give them to convey something of the flavor of results in this area.

6.2 Classifiers with non-zero in-sample error: "soft margin" and "slack"

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

$$h_i(\gamma_0) = \max\{0, \gamma_0 - \gamma_i\}$$

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 $> \gamma_0$. In particular, we can say that a classifier "has margin γ with slacks $h_1, \ldots h_n$" if the minimum margin on the *correctly* classified points is γ, and we use that in to calculate the slacks needed to accommodate the mis-classified points.

> **Soft margin bound/slack bound for imperfect separation** (Cristianini and Shawe-Taylor 2000, Theorem 4.22): Suppose a linear classifier achieves a margin γ on n samples, if allowed slacks $\vec{h}(\gamma) = (h_1(\gamma), h_2(\gamma), \ldots h_n(\gamma))$. Then there's a constant $c > 0$ such that, with probability at least $1 - \delta$, its error rate on new data from the same source is at most
>
> $$\varepsilon = \frac{c}{n}\left(\frac{R^2 + \|\vec{h}(\gamma)\|^2}{\gamma^2}\ln^2 n - \ln\delta\right)$$

(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.)

This suggests that the quantity to optimize is the ratio $\frac{R^2+\|\vec{h}\|^2}{\gamma^2}$. There is a trade-off here: adjusting the boundary to move it away from points we already correctly classify will increase γ, but usually at the cost of increasing the slacks (and so increasing $\|\vec{h}\|^2$). It could even move the boundary enough so that a point we used to get right is now mis-classified, moving one of the slacks from zero to positive.

6.3 Hinge loss

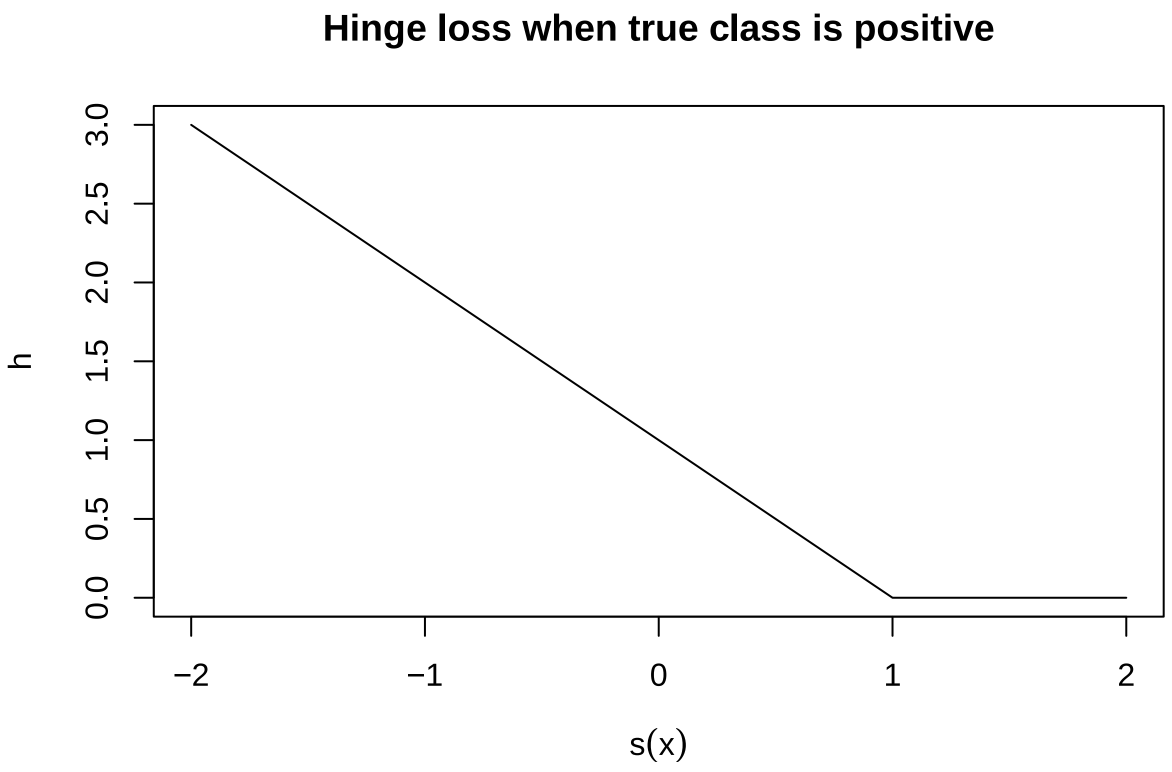
The notion of **hinge loss** is closely related to that of slack. If our classifier is

$$\hat{Z}(\vec{x}) = \operatorname{sgn} s(\vec{x})$$

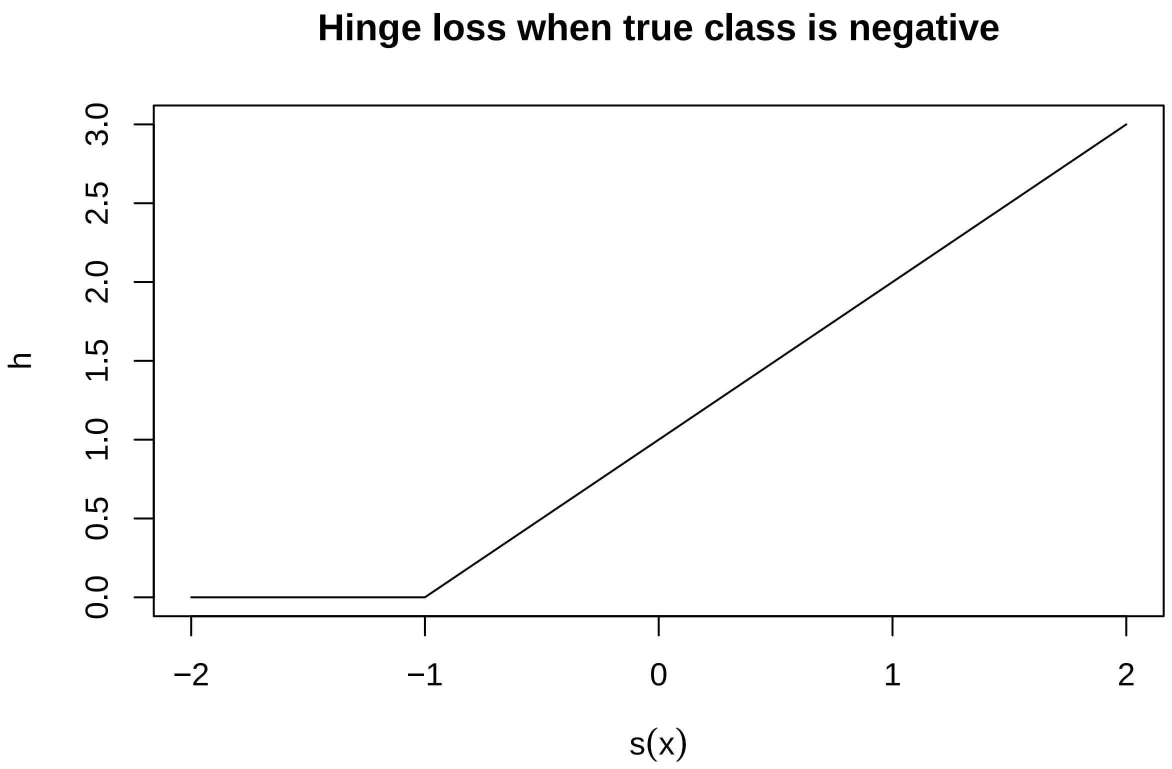
e.g., $s(\vec{x}) = b + \sum_{i=1}^{n} \alpha_i z_i\,(\vec{x}_i \cdot \vec{x})$, then the hinge loss at the point $\vec{x}_i, z_i$ is

$$h_i = \max\{0, 1 - z_i s(\vec{x}_i)\}$$

To see what this does, look at a plot, say for when $Y = Z = 1$.



This says that if our prediction has the right sign and is at least 1, the hinge loss is 0. We get no bonus for being extra sure that a positive point is positive. But we are penalized for *not* being sure that a positive point is positive, and we get increasingly large penalties when we're mis-classifying. Similarly in the other direction, when $Y = 0$, $Z = -1$:



The over-all hinge loss is just the average over the training data:

$$h = \frac{1}{n}\sum_{i=1}^{n} h_i = \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - z_i s(\vec{x}_i)\}$$

Zero hinge loss means that all our training points are correctly classified. Perfect linear classification *almost* implies that the hinge loss is zero[11]. So hinge loss is yet another "proxy" for classification error, which, unlike classification error, changes gradually as we change the parameters of the classifier, whether we think of those as the primal weights on the features, or the dual weights on the training points.

6.4 Being able to ignore some training points while still classifying perfectly

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

> **Compression/sparseness bound** (Cristianini and Shawe-Taylor 2000, Theorem 4.25): Take any classifier-learning algorithm which is trained on a set of n data points. Suppose that the *same* classifier would be returned on a sub-set of the training data with only m data points. If this classifies the n training points perfectly, then with probability at least $1-\delta$, the error rate on new data is at most
> $$\varepsilon = \frac{1}{n-m}\left(m \ln\frac{en}{m} + \ln\frac{n}{\delta}\right)$$

The argument here is simple enough to sketch[12]. 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 $\binom{n}{m}$, 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.

6.5 Error bounds for kernel classifiers

All of this carries over to kernel classifiers, since they are just linear classifiers in a new feature space[13]. We simply have to re-define the margins of the training points in terms of the kernel and the dual representation:

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

[11]If we have perfect classification, but some points have $|s(\vec{x}_i)| < 1$, the hinge loss will be positive. Let $r_i = 1/|s(\vec{x}_i)|$, and $r = \max r_i$. Then multiplying all the weights going in to calculating t by r will ensure that all of the new $|s(\vec{x}_i)| \geq 1$, and, since they all had the right signs before, we'll still have perfect classification. So if we can find a linear classifier which gets all the training data right, we can find another which has got zero hinge loss.

[12]The argument can be extended to handle situations where the training data aren't perfectly classified, but it becomes more complicated. You need to use something like Hoeffding's inequality to control the probability that the actual error rate is much larger than the observed error rate when $n-m$ is large.

[13]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.

7 Support Vector Machines

The three bounds suggest three strategies for learning kernel classifiers:

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

Generally speaking, if we follow strategies (1) or (2), we will get solutions where lots of the $\alpha_i = 0$, which is what (3) would aims for. But directly aiming at (3) moves us back towards hard, discrete, combinatorial optimization problems, rather than easy, continuous, smooth optimization problems.

7.1 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, but see Cristianini and Shawe-Taylor (2000).) In brief, the procedure is to maximize

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} z_i z_j \alpha_i \alpha_j K(\vec{x}_i, \vec{x}_j) \tag{5}$$

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

$$b = -\frac{1}{2}\left(\max_{i:\ y_i=0}\left(\sum_{j=1}^{n} z_j \alpha_j K(\vec{x}_j, \vec{x}_i)\right) + \min_{i:\ y_i=+1}\left(\sum_{j=1}^{n} z_j \alpha_j K(\vec{x}_j, \vec{x}_i)\right)\right) \tag{6}$$

In other words, b 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 the 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 (5). 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 moves us back to difficult discrete optimization problems, rather than easy, smooth continuous optimization.)

7.2 Soft Margin Maximization

We fix a positive constant λ and maximize

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} z_i z_j \alpha_i \alpha_j \left(K(\vec{x}_i, \vec{x}_j) + \lambda \delta_{ij}\right) \tag{7}$$

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

$$z_i b + z_i \sum_{j=1}^{n} z_j \alpha_j K(\vec{x}_j, \vec{x}_i) = 1 - \lambda \alpha_i \tag{8}$$

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

$$b = \frac{1 - \lambda\alpha_i}{z_i} - \sum_{j=1}^{n} z_j \alpha_j K(\vec{x}_j, \vec{x}_i) \tag{9}$$

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

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.

7.3 Hinge loss minimization

Finally, it's worth mentioning that one can also *numerically* tackle minimizing the hinge loss. Typically, what one does is minimize

$$\frac{1}{n}\sum_{i=1}^{n} h_i = \frac{1}{n}\sum_{i=1}^{n} \max\{0, 1 - z_i s(\vec{x}_i)\} + \lambda\|\vec{w}\|$$

8 R

There are several packages which can implement SVMs. The most friendly may be `kernlab`, which also provides functions for lots of different kernelized methods, and for direct calculation and manipulation of kernel matrices. The main function for fitting SVMs in this package is `ksvm`. It takes a formula argument, like `lm()` or `tree()`. If the response variable is a factor, it does classification, though you can change that; by default, it uses a Gaussian / radial basis function kernel, and guesses the appropriate scale factor based on the data, though again you can change that. (The package even includes a kernel for strings.) Here's how it would work for the running-data example with the rings, which is stored in a data frame called `df`:

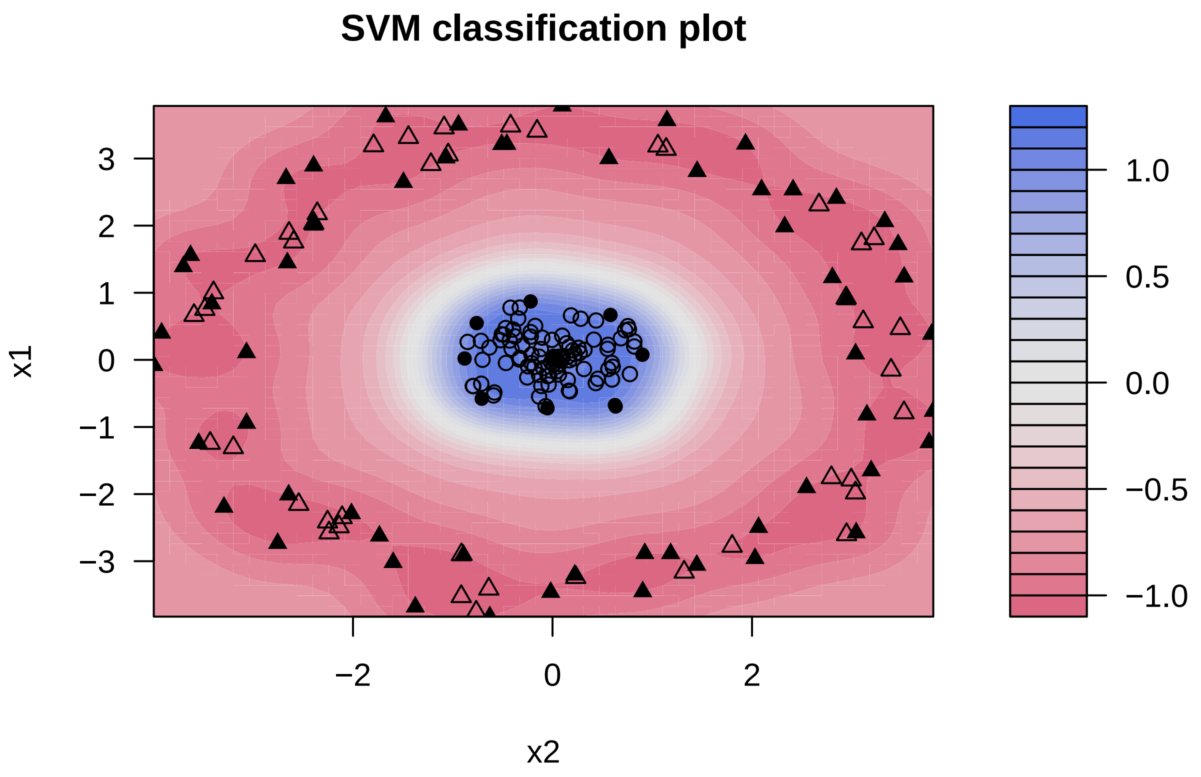
```
library(kernlab)
(df.svm <- ksvm(y ~ x1+x2, data=df))
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  6.94181159580486
##
## Number of Support Vectors : 68
##
## Objective Function Value : -7.739
## Training error : 0
```

The output tells us that this used a Gaussian kernel for classification, what the scale factor it picked, the error rate on the training set, how many training points got non-zero weight, etc. There is also a plotting function, which works OK when there are two "raw" features[14]:

```
plot(df.svm, data=df)
```

[14]If there are more than two features, you need to say which ones you want to vary; see `help(plot.ksvm)`.



Including the `data` argument plots that data set, with the two classes distinguished by their plotting symbols, which are filled in for the support vectors (while non-support vectors get hollow symbols). The colors indicate which class we'd predict at each point, and with what margin. Notice from the plot that one class is coded $+1$ and the other is coded -1, i.e., `ksvm` is computing what I've called Z rather than Y. The white band where the prediction is 0 is thus the decision boundary.

8.1 Other SVM packages in R

The oddly-named `e1071` library also has an `svm()` function, which works very much like the `ksvm()` function in `kernlab`. (Both of them rely on a C++ library called `libsvm`.) Some people find the documentation for `e1071::svm` more user-friendly. The `svmpath` library will actually fit SVMs over a whole range of λ values simultaneously (hence the name); but you have to pick a particular λ for prediction. (It also doesn't take a formula argument or allow you to do regression.)

8.2 A kernel ridge regression example in R

Here is an example of working through kernel ridge regression, using the now-familiar COMPAS data. It is a *little* sketchy to use a regression method to do classification, but we can hope to interpret the prediction as an estimated probability of recidivism (why?).

Unfortunately, `kernlab` does not give a nicely pre-packaged function for kernel ridge regression. Fortunately, there are a number of other add-on packages which rectify that, like `CVST`. Unfortunately, those add-on packages themselves have complicated syntax. Fortunately, making everything work with `kernlab` is both character-building and instructive about how the kernel matrix &c. actually work.

In this example, I'll use the Gaussian kernel, a.k.a. the radial basis function kernel. This requires[15] quantitative input variables, so I'll just use age and number of priors; we know from previous work with this data that

[15]While there *are* kernel functions for qualitative variables, and sums and products of kernels are themselves kernels, so it's perfectly possible to combine qualitative and quantitative regressors in a kernel ridge regression, I'm trying to keep this simple.

those are more important than the qualitative variables anyway.

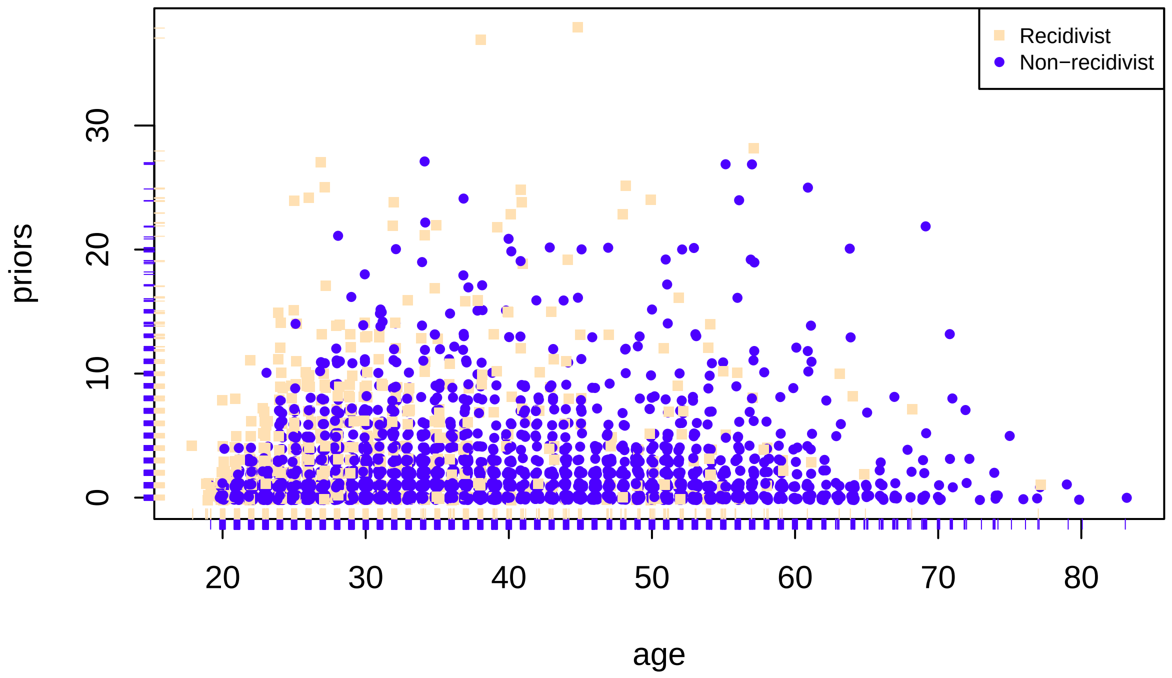


Figure 1: Priors vs. age for the training set, color-coded for recidivism. Points are "jittered" so that multiple individuals with equal features aren't completely on top of each other. The "rugs" along the axes give a sense of the marginal distributions of the two attributes for the two classes.

The first step is to build the kernel matrix for the training set, **K**:

```
# Build the kernel matrix for the training set
  # kernlab parameterizes the Gaussian/RBF kernel by what, in ordinary
  # Gaussian terms, would be 1/2Var, and calls that sigma (!)
    # This is what I wrote as \gamma in the text above
    # see help(rbfdot), and try to think kindly of our non-statistician
    # colleagues
k.train <- kernelMatrix(rbfdot(sigma=1),
                        as.matrix(training.set[,c("age", "priors_count")]))
```

The second step is to get the weights on the training points. This needs a value of λ, and I'll pick one arbitrarily to get started. Once we have that, we've automatically got fitted values on the training points.

```
# Set an initial value of lambda for exploratory purposes
lambda.initial <- 0.1
# Find the weights on the training points
  # solve(a) returns the inverse of the matrix a
  # solve(a,b) returns a^{-1} b where b is a (conforming) vector
compas.krr.wts <- solve(k.train + lambda.initial*diag(nrow(training.set)),
                          matrix(training.set$two_year_recid, ncol=1))
# With weights, we can get fitted values
compas.fits <- k.train %*% compas.krr.wts
```

To look at values on the testing set, we need to compute the output of the kernel machine for each new data point. That is, for each point z_j in the testing set, we want $\sum_{i=1}^{n} \alpha_i K(z_j, x_i)$.

```
# Find the matrix K(z_j, x_i) of every point z_j in the testing set with every
# point x_i in the training set
```

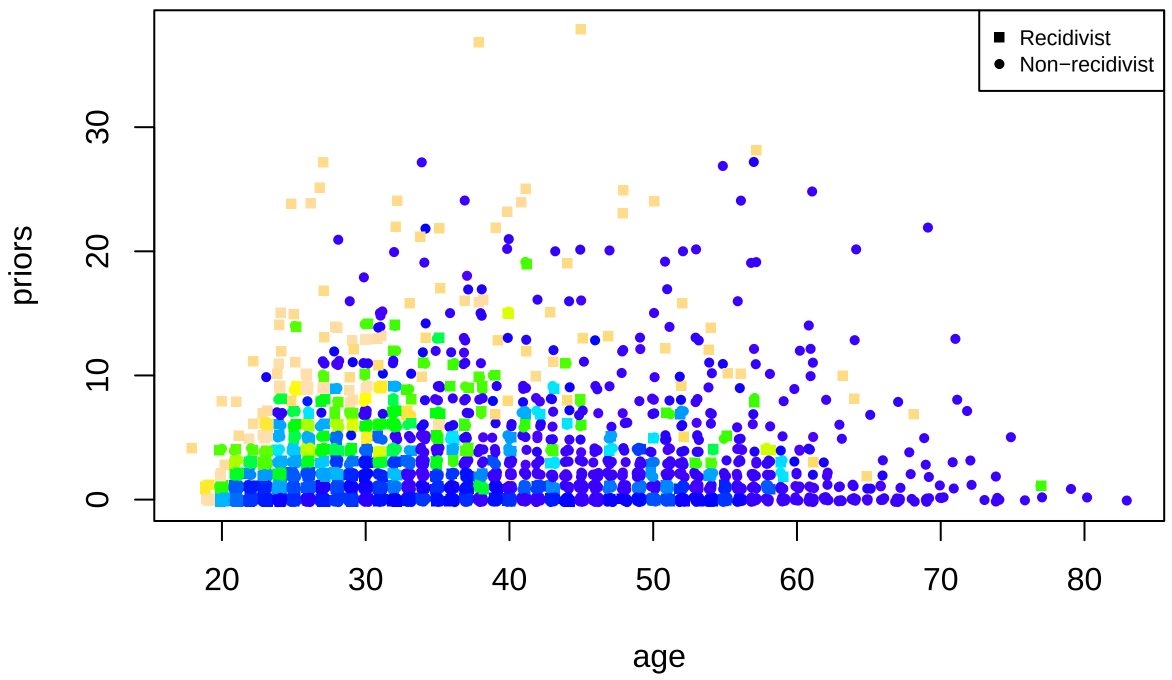


Figure 2: Fits on the training set, with color indicating fitted value (darker being lower predictions of recidivism, lighter higher predictions), and shape indicating actual outcome

```
  # the kernelMatrix() function has odd ideas about which argument should go
  # first here...
K.xz <- kernelMatrix(rbfdot(sigma=1),
                     y=as.matrix(training.set[,c("age", "priors_count")]),
                     x=as.matrix(testing.set[,c("age", "priors_count")]))
# Find the prediction of the kernel ridge regression by matrix multiplication
oos.preds <- K.xz %*% compas.krr.wts
```

At this point, we can begin exploring the impact of changing λ and/or the bandwidth of the kernel (our γ, `kernlab`'s σ), ideally through cross-validation within the training set (so that we're not over-optimistic when we come to the testing set).

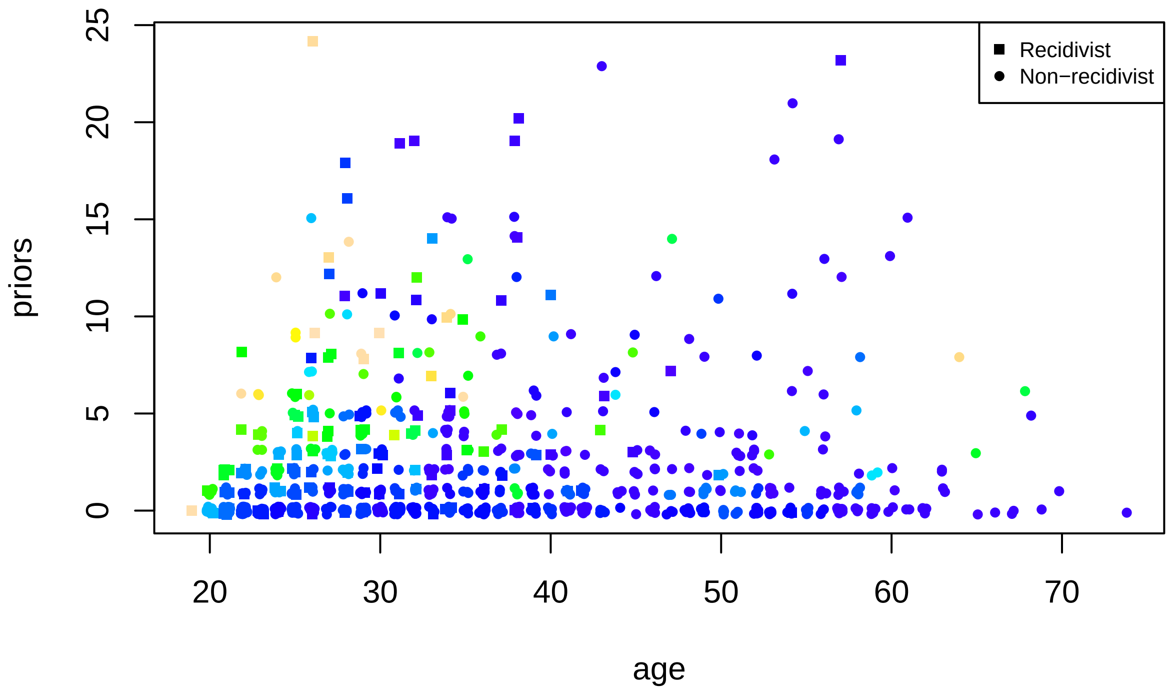


Figure 3: As in the previous plot, but looking at the testing set.

9 Random Features

Before closing, it's worth mentioning a surprisingly straightforward technique which can work *extremely* well. This is to "random features" or "random kitchen sinks" technique, which in essence goes as follows:

- Define a big family or library of functions of your base features. (Usually these will be nonlinear functions; I'll give a concrete example below.) Each function will be described by one or more parameters ω, so the function would be $\phi(\vec{x}; \omega)$.
- Draw $\omega_1, \omega_2, \ldots \omega_q$ from some fixed distribution ρ, without looking at your data.
- Declare your new features to be $\phi(\vec{x}; \omega_1), \ldots \phi(\vec{x}; \omega_q)$.
- run a linear model (classifier, regression, etc.) using these features.

To get a sense of how this might work, it's worth thinking about what are called "expansions" or "decompositions' or even"analyses" of functions — breaking complicated functions up into combinations of more elementary ones. The classic approach to this is what's called **Fourier decomposition** or **Fourier analysis**. In one dimension, it basically says that any reasonable[16] function $f(\vec{x})$ can be written as a combination of trigonometric functions:

$$f(x) = \int a(\vec{\omega}) \cos(\vec{\omega} \cdot \vec{x}) d\vec{\omega} + \int b(\vec{\omega}) \sin \vec{\omega} \cdot \vec{x}) d\omega = \int_{c(\vec{\omega})} \cos(\vec{\omega} \cdot \vec{x} + \beta(\vec{\omega})) d\vec{\omega}$$

(The last equality implicitly defines c and β, using the trigonometric identity $\cos(a+b) = \cos a \cos b - \sin a \sin b$.) On the other hand, if we pick $\vec{\omega}$ according to some distribution ρ, *in expectation* we'll get

$$\int \rho(\vec{\omega}) \cos(\vec{\omega} \cdot \vec{x}) d\vec{\omega}$$

and the law of large numbers says that for large q,

$$\frac{1}{q} \sum_{k=1}^{q} \cos(\vec{\omega}_k \cdot \vec{x}) \approx \int \rho(\omega) \cos(\vec{\omega} \cdot \vec{x}) d\vec{\omega}$$

So if we want to approximate some function f, we just need to figure out what its **Fourier transform** $c(\vec{\omega})$ is, and use that as the distribution from which we draw a bunch of $\vec{\omega}$, and use those to define our new features. If we don't know exactly what function f we want, we can draw from some pretty broad distribution, and then weight the random functions and use that to define our new (approximate) f. It turns out that in practice, we don't even need q to be that big, often somewhere between 30 and 1000 suffices.

Now, at this point, you might be raising a lot of doubts and objections. Don't we need to draw random phases β as well as frequencies ω? (Yes.) Does this work for approximating kernel $K(\vec{x}, \vec{x}')$, which are functions of *two* vectors? (Yes, take the inner product between the random features.) What if $\int c(\omega) d\omega \neq 1$? (Scale it up or down until it does; we're using a linear method anyway in the end so that's fine.) What if $c(\omega) < 0$ for some ω? (It turns out we don't have to worry about this *for kernel functions*, because of Mercer's theorem.) What if we don't want to use cosines as our basis functions, but something else? (Just about any basis will do, so long as you can figure out a good ρ.) Suffice it to say that there is some non-trivial math in showing that the random features technique works, but it has been done, and it leads to some very impressive, streamlined results.

9.1 In R

I'll illustrate with the `expandFunctions` package, which gives a lot of tools for expanding a data frame or matrix with additional columns (features) calculated from the columns you start with. One of those tools is the "Random Affine Projection Transformation" (RAPT). That is to say, if we start with an $n \times p$ data

[16]There are precise definitions of "reasonable", which you can find in references on Fourier analysis, like the very user-friendly Körner (1988).

matrix **x**, it generates a random $p \times q$ matrix **W**, and a random $1 \times q$ matrix B, and then calculates **xW** and adds **B** to each row of the product. Here the columns of **W** play the role of the vectors $\vec{\omega}$, and the entries in B play the role of the phases $\beta(\omega)$. The function `raptMake()` creates a persistent object containing the **W** matrix and B vector, and the function `rapt()` applies such an object to a matrix,. Here, for example, we'll create random affine transformations mapping two-dimensional data to 30 dimensions:

```
library(expandFunctions)
raptObj <- raptMake(p=2, q=30, WdistOpt=list(sd=1),
                    bDistOpt=list(min=-pi, max=pi))
```

(The default distribution for the entries in **W** is Gaussian; the default distribution for the entries in **B** is uniform. We want the latter to be phases, so I've adjusted the limits to make them sensible angles.) Now we can apply this to our rings data:

```
dim(rapt(as.matrix(df[,c("x1","x2")]), raptObj))
```

```
## [1] 200  30
```

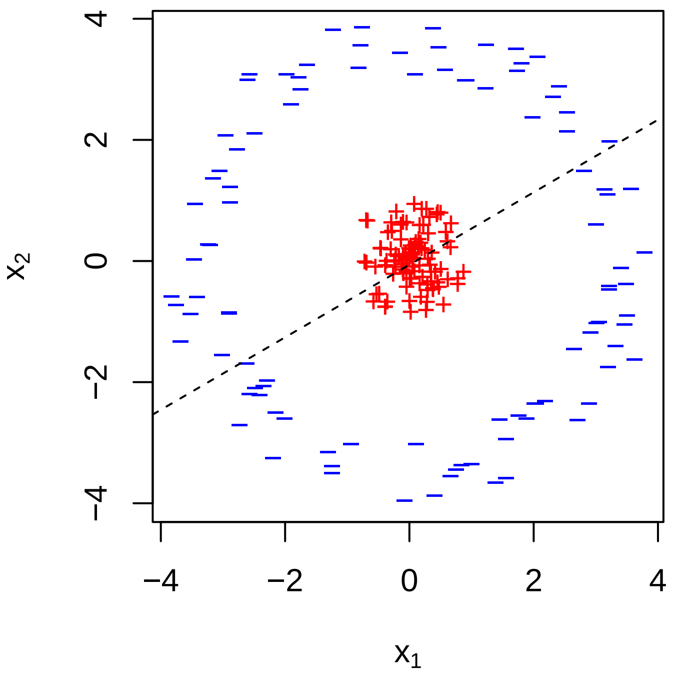
This shows that we get 30-dimensional features out. The linear transformation done by `rapt()` isn't super useful for our purposes here. (We will see uses for random linear transformations later in the course.) What's more helpful is to take the cosine:

```
df.augmented <- data.frame(y=df$y, cos(rapt(as.matrix(df[,c("x1","x2")]), raptObj)))
```

We've now added on the new, nonlinear features.

Remember that before, it was basically impossible to do linear classification with the x_1 and x_2 features. For instance, if we tried logistic regression, it would work badly:

```
rings.logistic <- glm(y ~ x1+x2, data=df, family="binomial")
```



(Most of the + points are below the line and most of the − points are above, but this is still pathetic.) But with the extra features, we can get perfect linear separation of the classes in the expanded feature space:

```
glm.rks <- glm(y~., data=df.augmented, family="binomial")
```

```
## Warning: glm.fit: algorithm did not converge
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
table(predict(glm.rks, type="response") >= 0.5, df.augmented$y)
```

```
##
##           0   1
##   FALSE 100   0
##   TRUE    0 100
```

(Obviously I'm not going to show you the classification boundary in 30-dimensional space.) If we wanted to make predictions on new data, we'd have to apply the same transformation to those new data points, which is why `raptMake()` creates a persistent object. For instance, let's make a grid for plotting over the region where we have data:

```
plotting.grid <- with(df, expand.grid(x1=seq(from=min(x1),
                                             to=max(x1),
                                             length.out=100),
                                      x2=seq(from=min(x2),
                                             to=max(x2),
                                             length.out=100)))
```

And now let's see what predictions we get from the (linear) part of the logistic regression model:

```
featurized.grid <- cos(rapt(as.matrix(plotting.grid), raptObj))
logodds <- coefficients(glm.rks)[1] + featurized.grid %*% coefficients(glm.rks)[-1]
```

```
par(pty="s")
plot(x2 ~ x1, data=df,
     pch=ifelse(y=="1", "+", "-"),
     col=ifelse(y=="1", "red", "blue"),
     xlab=expression(x[1]), ylab=expression(x[2]))
contour(x=seq(from=min(x1), to=max(x1), length.out=100),
        y=seq(from=min(x2), to=max(x2), length.out=100),
        z=matrix(logodds, ncol=100),
        levels=pretty(range(logodds), 10),
        nlevels=10,
        add=TRUE)
```

Of course, if q is large compared to n, we might not get a very stable fit, so we might want to apply some sort of penalization to the size of the coefficients on the new features (ridge regression or lasso, say), but that's a refinement.

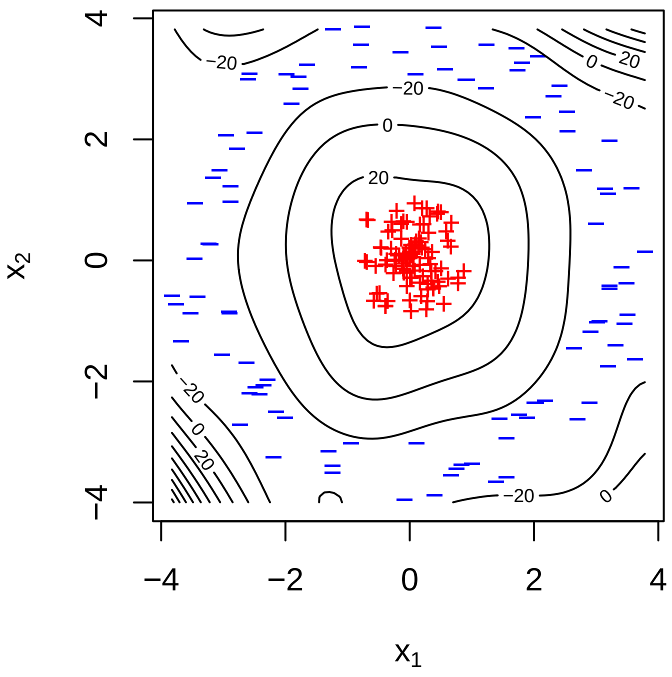


Figure 4: Contour lines of predictions from our random-kitchen-sink logistic regression. The predictions here are on the log-odds scale, so the classification boundary is the 0 contour between the outer ring and the inner blob.

10 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). Shawe-Taylor and Cristianini (2004) explores a broader range of kernel-based methods, and has an extensive treatment of kernels for different data types (for instance, kernels for measuring similarity between strings of text), and theory for combining kernels (for instance, if some features are categorical but others are numerical).

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, even mind-expanding, but remember that you're reading the pronouncements of an opinionated and irascible genius.

The random kitchen sink technique (along with the name) was introduced by Rahimi and Recht (2008), Rahimi and Recht (2009). I have specifically sketched how to use random features to approximate (Gaussian) kernels; there are lots of other possibilities.

11 Kernel machines vs. Nadaraya-Watson smoothing

What we've just introduced as kernel machines are functions of the form

$$s(x) = \sum_{i=1}^{n} \alpha_i K(x, x_i)$$

where the kernel function K has to meet the requirements of symmetry, the Cauchy-Schwarz inequality and Mercer's inequality; this makes sure that it's a (weighted) inner product in some feature space. When we do (unregularized) kernel regression in this sense, $\hat{\alpha} = \mathbf{K}^{-1}\mathbf{y}$, as you'll prove in HW 8; more usefully, if we do kernel ridge regression, $\hat{\alpha} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{y}$.

From other statistics classes, you may be familiar with a technique variously called "kernel regression", "kernel smoothing", "Nadaraya-Watson regression" or "Nadaraya-Watson smoothing" (after Nadaraya (1964) and Watson (1964)), which is

$$\hat{mu x}) = \sum_{i=1}^{n} y_i \frac{G(x, x_i)}{\sum_{j=1}^{n} G(x, x_i)}$$

(It'd be more usual to write K instead of G, but I'm trying to clear up confusion, not amplify it.) This kernel G is usually of the form $G(u, v) = G(u - v)$, and is a probability density function with mean 0 and finite variance.

In a sane universe, we might hope that these two sets of models with the same name would turn out to actually *be* different ways of writing the same thing, perhaps after some math just hard enough that we can feel pleased with ourselves for grasping it. We do not live in such a universe. There are at least two distinct reasons why these are actually two different sets of models:

1. Not every kernel in the first sense, what I wrote K, is a valid choice of a kernel in the second sense, what I wrote as G. The *Gaussian* is, we can use $K(x, x') = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\|x-x'\|/2\sigma^2}$ as both a (weighted) inner product on the space of infinite-order polynomials *and* as a pdf. But not ever valid K is a valid G, or vice versa. (There are valid K kernels which are sometimes negative, which doesn't make sense for G kernels.)
2. In kernel machines, the weights α_i need to be independent of x, and $K(x, x_i)$ needs to be a function of x and x_i alone. If we try to set

$$\sum_{i=1}^{n} \alpha_i K(x, x_i) = \sum_{i=1}^{n} y_i \frac{G(x, x_i)}{\sum_{j=1}^{n} G(x, x_j)}$$

we can see we're in trouble. If we say that $\alpha_i = y_i$, we'd need a K function that changes depending on all the *other* training points, not just x_j. If on the other hand we set $K(x, x_i) = G(x, x_i)$, we've got a weight that isn't constant but changes with x. There is (in general) no way to finesse this to make these two expressions equal to each other.

So: kernel machines, even for regression, and Nadaraya-Watson smoothing, are just not the same. Why on Earth, then are they both called "kernel regression" or "kernel methods"?

To answer this, we need to go back behind modern statistics to pure math. In functional analysis, we often deal with "operators" or "transforms" on entire functions. In particular, a linear operator or a linear transform is something we *do* to one function, say f, to produce another function, say Tf, which obeys linearity: if f and g are two functions, and a and b are two scalars,

$$T(af + bg) = aTf + bTg$$

meaning, more particularly, that at all points x in the domain of f and g,

$$T(af + bg)(x) = a(Tf)(x) + b(Tg)(x)$$

where I'm using parentheses to be fussy/explicit about precedence of operations. Taking derivatives, for instance, is a linear operation on functions in this sense.

Now, among linear operators, an especially important place is held by the "linear integral operators", or "linear integral transformations', which are the ones that can be written as (unsurprisingly) integrals: T is a linear integral operator when

$$(Tf)(x) = \int f(u)K(x, u)du$$

for some two-argument function K. This K is called the **kernel** of the linear integral operator, since"kernel" literally means "seed", and the metaphor was that K is the seed from which the entire transform grows.

We got the two meanings of "kernel regression" because people trained in this branch of math connected it to statistics in two independent ways.

On the one hand, linear integral operators are linear operators, so we can ask about what corresponds to their eigenvectors. Since they act on functions, those are going to be eigenfunctions, i.e., function ϕ where

$$\int \phi(u)K(x, u)du = \lambda\phi(x)$$

for some eigenvalue λ. Kernels in the inner-product-in-feature-space sense turn out to be ones which lead to especially nice eigenfunctions, because then it turns out that

$$K(x, x') = \sum_{j=1}^{\infty} \phi_j(x)\phi_j(x')\lambda_j$$

So kernel regression, in this first sense, turns out to mean "regression where we use the eigenfunctions of the kernel as the features".

On the other hand, one kind of linear integral transformation we can apply to a function is **convolution**, which is where $K(x, u) = G(x - u)$, so

$$(Tf)(x) = \int f(u)G(x - u)du$$

If, in particular, this $G(x - u)$ is a pdf, then convolution becomes **smoothing**, because $(Tf)(x)$ is a weighted average of values of $f(u)$ for u near x. (Exactly what "near" means depends on how far the tails of $G(x - u)$ extend, and how rapidly they tend to 0.) This is basically what Nadaraya-Watson smoothing is doing; the denominator is just to make sure that we're always getting a properly-normalized weighted average. Behind

Nadarya-Watson smoothing, there was the statistical example of kernel density estimation, (Rosenblatt 1956; Parzen 1962), which is just the convolution of the empirical distribution with the kernel pdf.

So: mathematicians used "kernel" to mean part of an integral operator. Statisticians trained in this sort of math connected it to regression in two distinct ways, one group by using the eigenfunctions of integral operators as features, the other group by looking at integral operators that did smoothing. If one side had called what they were doing "eigenfunction-feature machines", and the other "convolution smoothers", we would have different confusions.

(In addition to this mathematical sense of "kernel" in linear integral operators, it is also used for a rather different concept in abstract algebra; the names appear to have been developed independently [https://mathshistory.st-andrews.ac.uk/Miller/mathword/k/]).

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.

Körner, T. W. 1988. *Fourier Analysis.* Cambridge, England: Cambridge University Press.

Minsky, Marvin, and Seymour Papert. 1969. *Perceptrons: An Introduction to Computational Geometry.* Cambridge, Massachusetts: MIT Press.

Nadaraya, E. A. 1964. "On Estimating Regression." *Theory of Probability and Its Applications* 9:141–42. https://doi.org/10.1137/1109020.

Parzen, Emanuel. 1962. "On Estimation of a Probability Density Function and Mode." *Annals of Mathematical Statistics* 33:1065–76. https://doi.org/10.1214/aoms/1177704472.

Rahimi, Ali, and Benjamin Recht. 2008. "Random Features for Large-Scale Kernel Machines." In *Advances in Neural Information Processing Systems 20 (Nips 2007)*, edited by John C. Platt, Daphne Koller, Yoram Singer, and Samuel T. Roweis, 1177–84. Red Hook, New York: Curran Associates. http://papers.nips.cc/paper/3182-random-features-for-large-scale-kernel-machines.

———. 2009. "Weighted Sums of Random Kitchen Sinks: Replacing Minimization with Randomization in Learning." In *Advances in Neural Information Processing Systems 21 [Nips 2008]*, edited by Daphne Koller, D. Schuurmans, Y. Bengio, and L. Bottou, 1313–20. Red Hook, New York: Curran Associates, Inc. https://papers.nips.cc/paper/2008/hash/0efe32849d230d7f53049ddc4a4b0c60-Abstract.html.

Rosenblatt, Murray. 1956. "Remarks on Some Nonparametric Estimates of a Density Function." *Annals of Mathematical Statistics* 27:832–37. https://doi.org/10.1214/aoms/1177728190.

Shawe-Taylor, John, and Nello Cristianini. 2004. *Kernel Methods for Pattern Analysis.* Cambridge, England: Cambridge University Press. https://doi.org/10.1017/CBO9780511809682.

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

Watson, Geoffrey S. 1964. "Smooth Regression Analysis." *Sanhkya* 26:359–72. http://www.jstor.org/stable/25049340.