Statistical Machine Learning Notes 8

Kernel Methods and SVMs

Instructor: Justin Domke

Contents

1 Introduction

Support Vector Machines (SVMs) are a very succesful and popular set of techniques for classification. Historically, SVMs emerged after the neural network boom of the 80s and early 90s. People were surprised to see that SVMs with little to no "tweaking" could compete with neural networks involving a great deal of manual engineering.

It remains true today that SVMs are among the best "off-the-shelf" classification methods. If you want to get good results with a minimum of messing around, SVMs are a very good choice.

Unlike the other classification methods we discuss, it is not convenient to begin with a concise definition of SVMs, or even to say what exactly a "support vector" is. There is a set of ideas that must be understood first. Most of these you have already seen in the notes on linear methods, basis expansions, and template methods. The biggest remaining concept is known as the **kernel trick**. In fact, this idea is so fundamental many people have advocated that SVMs be renamed "Kernel Machines".

It is worth mentioning that the standard presentation of SVMs is based on the concept of "margin". For lack of time, this perspective on SVMs will not be presented here. If you will be working seriously with SVMs, you should familiarize yourself with the margin perspective to enjoy a full understanding.

(**Warning:** These notes are probably the most technically challenging in this course, particularly if you don't have a strong background in linear algebra, Lagrange multipliers, and optimization. Kernel methods simply use more mathematical machinery than most of the other techniques we cover, so you should be prepared to put in some extra effort. Enjoy!)

2 Kernel Ridge Regression

We begin by not talking about SVMs, or even about classification. Instead, we revisit ridge regression, with a slight change of notation. Let the set of inputs be $\{(\mathbf{x}_i, y_i)\}$, where i indexes the samples. The problem is to minimize

$$\sum_i \big(\mathbf{x}_i \cdot \mathbf{w} - y_i\big)^2 + \lambda \mathbf{w} \cdot \mathbf{w}$$

If we take the derivative with respect to $\mathbf{w}$ and set it to zero, we get

$$\begin{aligned} \mathbf{0} &= \sum_i 2\mathbf{x}_i(\mathbf{x}_i^T\mathbf{w} - y_i) + 2\lambda\mathbf{w} \\ \mathbf{w} &= \big(\sum_i \mathbf{x}_i\mathbf{x}_i^T + \lambda I\big)^{-1} \sum_i \mathbf{x}_i y_i. \end{aligned}$$

Now, let's consider a different derivation, making use of some Lagrange duality. If we introduce a new variable z_i, and constrain it to be the difference between $\mathbf{w} \cdot \mathbf{x}_i$ and y_i, we have

$$\begin{aligned} \min_{\mathbf{w},\mathbf{z}} \quad & \frac{1}{2}\mathbf{z} \cdot \mathbf{z} + \frac{1}{2}\lambda\mathbf{w} \cdot \mathbf{w} \\ \text{s.t.} \quad & z_i = \mathbf{x}_i \cdot \mathbf{w} - y_i. \end{aligned} \tag{2.1}$$

Using α_i to denote the Lagrange multipliers, this has the Lagrangian

$$\mathcal{L} = \frac{1}{2}\mathbf{z} \cdot \mathbf{z} + \frac{1}{2}\lambda\mathbf{w} \cdot \mathbf{w} + \sum_i \alpha_i(\mathbf{x}_i \cdot \mathbf{w} - y_i - z_i).$$

Recall our foray into Lagrange duality. We can solve the original problem by doing

$$\max_{\boldsymbol{\alpha}} \min_{\mathbf{w},\mathbf{z}} \mathcal{L}(\mathbf{w}, \mathbf{z}, \boldsymbol{\alpha}).$$

To begin, we attack the inner minimization: For fixed $\boldsymbol{\alpha}$, we would like to solve for the minimizing $\mathbf{w}$ and $\mathbf{z}$. We can do this by setting the derivatives of $\mathcal{L}$ with respect to z_i and $\mathbf{w}$ to be zero. Doing this, we can find[1]

$$z_i^* = \alpha_i, \qquad \mathbf{w}^* = -\frac{1}{\lambda}\sum_i \alpha_i \mathbf{x}_i \tag{2.2}$$

So, we can solve the problem by maximizing the Lagrangian (with respect to $\boldsymbol{\alpha}$), where we substitute the above expressions for z_i and $\mathbf{w}$. Thus, we have an unconstrained maximization.

$$\max_{\boldsymbol{\alpha}} \mathcal{L}(\mathbf{w}^*(\boldsymbol{\alpha}), \mathbf{z}^*(\boldsymbol{\alpha}), \boldsymbol{\alpha})$$

[1]

$$0 = \frac{d\mathcal{L}}{dz_i} = z_i - \alpha_i$$

$$\mathbf{0} = \frac{d\mathcal{L}}{d\mathbf{w}} = \lambda\mathbf{w} + \sum_i \alpha_i \mathbf{x}_i$$

Before diving into the details of that, we can already notice something very interesting happening here: $\mathbf{w}$ is given by a sum of the input vectors $\mathbf{x}_i$, weighted by $-\alpha_i/\lambda$. If we were so inclined, we could avoid explicitly computing $\mathbf{w}$, and predict a new point $\mathbf{x}$ directly from the data as

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{w} = -\frac{1}{\lambda} \sum_i \alpha_i \mathbf{x} \cdot \mathbf{x}_i.$$

Now, let $k(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$ be the "kernel function". For now, just think of this as a change of notation. Using this, we can again write the ridge regression predictions as

$$\boxed{f(\mathbf{x}) = -\frac{1}{\lambda} \sum_i \alpha_i k(\mathbf{x}, \mathbf{x}_i).}$$

Thus, all we really need is the *inner product* of $\mathbf{x}$ with each of the training elements $\mathbf{x}_i$. We will return to why this might be useful later. First, let's return to doing the maximization over the Lagrange multipliers $\boldsymbol{\alpha}$, to see if anything similar happens there. The math below looks really complicated. However, all the we are doing is substituting the expressions for z_i^* and $\mathbf{w}^*$ from Eq. 2.2, then doing a lot of manipulation.

$$\begin{aligned}
\max_{\boldsymbol{\alpha}} \min_{\mathbf{w},\mathbf{z}} \mathcal{L} &= \max_{\boldsymbol{\alpha}} \frac{1}{2} \sum_i \alpha_i^2 + \frac{1}{2}\lambda\big(-\frac{1}{\lambda} \sum_i \alpha_i \mathbf{x}_i\big) \cdot \big(-\frac{1}{\lambda} \sum_j \alpha_j \mathbf{x}_j\big) \\
&\qquad + \sum_i \alpha_i (\mathbf{x}_i \cdot -\frac{1}{\lambda} \sum_j \alpha_j \mathbf{x}_j - y_i - \alpha_i) \\
&= \max_{\boldsymbol{\alpha}} \frac{1}{2} \sum_i \alpha_i^2 + \frac{1}{2\lambda} \sum_i \sum_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j - \frac{1}{\lambda} \sum_i \sum_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_i \alpha_i (y_i + \alpha_i) \\
&= \max_{\boldsymbol{\alpha}} -\frac{1}{2} \sum_i \alpha_i^2 - \frac{1}{2\lambda} \sum_i \sum_j \alpha_i \alpha_j \mathbf{x}_i \cdot \mathbf{x}_j - \sum_i \alpha_i y_i \\
&= \boxed{\max_{\boldsymbol{\alpha}} -\frac{1}{2} \sum_i \alpha_i^2 - \frac{1}{2\lambda} \sum_i \sum_j \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) - \sum_i \alpha_i y_i.}
\end{aligned}$$

Again, we only need inner products. If we define the matrix K by

$$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j),$$

then we can rewrite this in a punchier vector notation as

$$\max_{\boldsymbol{\alpha}} \min_{\mathbf{w},\mathbf{z}} \mathcal{L} = \max_{\boldsymbol{\alpha}} -\frac{1}{2}\boldsymbol{\alpha}\cdot\boldsymbol{\alpha} - \frac{1}{2\lambda}\boldsymbol{\alpha}^T K\boldsymbol{\alpha} - \boldsymbol{\alpha}\cdot\mathbf{y}.$$

Here, we use a capital K to denote the matrix with entries K_{ij} and a lowercase k to denote the kernel function $k(\cdot,\cdot)$. Note that most literature on kernel machines mildly abuses notation by using the capital letter K for both.

The thing on the right is just a quadratic in $\boldsymbol{\alpha}$. As such, we can find the optimum as the solution of a linear system[2]. What is important is the observation that, again, we only need the inner products of the data $k(\mathbf{x}_i,\mathbf{x}_j) = \mathbf{x}_i\cdot\mathbf{x}_j$ to do the optimization over $\boldsymbol{\alpha}$. Then, once we have solved for $\boldsymbol{\alpha}$, we can predict $f(\mathbf{x})$ for new $\mathbf{x}$ again using only inner products. If someone tells us all the inner products, we don't need the original data $\{\mathbf{x}_i\}$ at all!

3 The Kernel Trick

So we can work completely with inner products, rather than the vectors themselves. So what?

One way of looking at things is that we can *implicitly* use basis expansions. If we want to take $\mathbf{x}$, and transform it into some fancy feature space $\boldsymbol{\phi}(\mathbf{x})$, we can replace the kernel function by

$$K_{ij} = k(\mathbf{x}_i,\mathbf{x}_j) = \boldsymbol{\phi}(\mathbf{x}_i)\cdot\boldsymbol{\phi}(\mathbf{x}_j).$$

The point of talking about this is that for certain basis expansions, we can compute k very cheaply without ever explicitly forming $\boldsymbol{\phi}(\mathbf{x}_i)$ or $\boldsymbol{\phi}(\mathbf{x}_j)$. This can mean a huge computational savings. A nice example of this is the kernel function

$$k(\mathbf{x},\mathbf{v}) = (\mathbf{x}\cdot\mathbf{v})^2.$$

We can see that

[2]It is easy to show (by taking the gradient) that the optimum is at

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

$$\begin{aligned} k(\mathbf{x}, \mathbf{v}) &= (\sum_i x_i v_i)^2 \\ &= (\sum_i x_i v_i)(\sum_j x_j v_j) \\ &= \sum_i \sum_j x_i x_j v_i v_j. \end{aligned}$$

It is not hard to see that

$$k(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v}),$$

where $\boldsymbol{\phi}$ is a quadratic basis expansion $\phi_m(\mathbf{x}) = \phi_i(\mathbf{x})\phi_j(\mathbf{x})$. For example, in two dimensions,

$$\begin{aligned} k(\mathbf{x}, \mathbf{v}) &= (x_1 v_1 + x_2 v_2)^2 \\ &= x_1 x_1 v_1 v_1 + x_1 x_2 v_1 v_2 + x_2 x_1 v_2 v_1 + x_2 x_2 v_2 v_2. \end{aligned}$$

while the basis expansions are

$$\boldsymbol{\phi}(\mathbf{x}) = (x_1 x_1, x_1 x_2, x_2 x_1, x_2 x_2),$$

$$\boldsymbol{\phi}(\mathbf{v}) = (v_1 v_1, v_1 v_2, v_2 v_1, v_2 v_2).$$

It is not hard to work out that $k(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v})$.

However, notice that we can compute $k(\mathbf{x}, \mathbf{v})$ in time $O(d)$, rather than the $O(d^2)$ time it would take to explicitly compute $\boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v})$. This is the "kernel trick": getting around the computational expense in computing large basis expansions by directly computing kernel functions. Notice, however, that the kernel trick changes *nothing*, *nada*, *zero* about the statistical issues with huge basis expansions. We get exactly the same predictions as if we computed the basis expansion explicitly, and used traditional linear methods. We just compute the predictions in a different way.

In fact can invent a new kernel function $k(\mathbf{x}, \mathbf{v})$, and, as long as it obeys certain rules, use it in the above algorithm, with out explicitly thinking about basis expansion at all. Some common examples are:

name	$k(\mathbf{x}, \mathbf{v})$
Linear	$\mathbf{x} \cdot \mathbf{v}$
Polynomial	$(r + \mathbf{x} \cdot \mathbf{v})^d$, for some $r \geq 0, d > 0$
Radial Basis Function	$\exp\big(-\gamma \lvert\lvert \mathbf{x} - \mathbf{v} \rvert\rvert^2\big)$, $\gamma > 0$
Gaussian	$\exp\big(-\frac{1}{2\sigma^2} \lvert\lvert \mathbf{x} - \mathbf{v} \rvert\rvert^2\big)$

We will return below to the question of what kernel functions are "legal", meaning there is some feature space $\boldsymbol{\phi}$ such that $k(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v})$.

Now, what exactly was it about ridge regression that let us get away with working entirely with inner products? How much could we change the problem, and preserve this? We really need two things to happen:

1. When we take $d\mathcal{L}/d\mathbf{w} = \mathbf{0}$, we need to be able to solve for $\mathbf{w}$, and the solution needs to be a linear combination of the input vectors $\mathbf{x}_i$.
2. When we substitute this solution back into the Lagrangian, we need to get a solution that simplifies down into inner products only.

Notice that this leaves us a great deal of flexibility. For example, we could replace the least-squares criterion $\mathbf{z} \cdot \mathbf{z}$ with an alternative (convex) measure. We could also change the way in which which measure errors from $z_i = \mathbf{w} \cdot \mathbf{x}_i - y_i$, to something else, (although with some restrictions).

4 Support Vector Machines

Now, we turn to the binary classification problem. Support Vector Machines result from minimizing the hinge loss $(1 - y_i \mathbf{w} \cdot \mathbf{x}_i)_+$ with ridge regularization.

$$\min_{\mathbf{w}} \sum_i (1 - y_i \mathbf{w} \cdot \mathbf{x}_i)_+ + \lambda ||\mathbf{w}||^2.$$

This is equivalent to (for $c = 2/\lambda$)

$$\min_{\mathbf{w}} c \sum_i (1 - y_i \mathbf{w} \cdot \mathbf{x}_i)_+ + \frac{1}{2} ||\mathbf{w}||^2.$$

Because the hinge loss is non-differentiable, we introduce new variables z_i, creating a constrained optimization

$$\begin{aligned} \min_{\mathbf{z},\mathbf{w}} \quad & c \sum_i z_i + \frac{1}{2} ||\mathbf{w}||^2 \\ \text{s.t.} \quad & z_i \geq (1 - y_i \mathbf{w} \cdot \mathbf{x}_i) \\ & z_i \geq 0. \end{aligned} \tag{4.1}$$

Introducing new constraints to simplify an objective like this seems strange at first, but isn't too hard to understand. Notice the constraints are exactly equivalent to forcing that $z_i \geq (1 - y_i \mathbf{w} \cdot \mathbf{x}_i)_+$. But since we are minimizing the sum of all the z_i, the optimization will make it as small as possible, and so z_i will be the hinge loss for example i, no more, no less.

Introducing Lagrange multipliers α_i to enforce that $z_i \geq (1 - y_i \mathbf{w} \cdot \mathbf{x}_i)$ and μ_i to enforce that $z_i \geq 0$, we get the Lagrangian

$$\mathcal{L} = c \sum_i z_i + \frac{1}{2}||\mathbf{w}||^2 + \sum_i \alpha_i (1 - y_i \mathbf{w} \cdot \mathbf{x}_i - z_i) + \sum_i \mu_i (-z_i).$$

A bunch of manipulation changes this to

$$\mathcal{L} = \sum_i z_i \big(c - \mu_i - \alpha_i\big) + \frac{1}{2}||\mathbf{w}||^2 + \sum_i \alpha_i - \mathbf{w} \cdot \sum_i \alpha_i y_i \mathbf{x}_i.$$

As ever, Lagrangian duality states that we can solve our original problem by doing

$$\max_{\boldsymbol{\alpha} \geq \mathbf{0}, \boldsymbol{\mu} \geq \mathbf{0}} \min_{\mathbf{z}, \mathbf{w}} \mathcal{L}.$$

For now, we work on the inner minimization. For a particular $\boldsymbol{\alpha}$ and $\boldsymbol{\mu}$, we want to minimize with respect to $\mathbf{z}$ and $\mathbf{w}$. By setting $d\mathcal{L}/d\mathbf{w} = 0$, we find that

$$\mathbf{w}^* = \sum_i \alpha_i y_i \mathbf{x}_i.$$

Meanwhile, setting $d\mathcal{L}/dz_i$ gives that

$$\alpha_i = c - \mu_i.$$

If we substitute these expressions, we find that μ "disappears". However, notice that since $\mu_i \geq 0$ we must have that $\alpha_i \leq c$.

$$\begin{aligned}
\max_{\boldsymbol{\alpha} \geq \mathbf{0}, \boldsymbol{\mu} \geq \mathbf{0}} \min_{\mathbf{z}, \mathbf{w}} \mathcal{L} &= \max_{0 \leq \boldsymbol{\alpha} \leq C} \sum_i z_i \big(\alpha_i - \alpha_i\big) + \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \\
&\quad + \sum_i \alpha_i - \sum_j \alpha_j y_j \mathbf{x}_j \cdot \sum_i \alpha_i y_i \mathbf{x}_i \\
&= \max_{0 \leq \boldsymbol{\alpha} \leq c} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j
\end{aligned}$$

This is a maximization of a quadratic objective, under linear constraints. That is, this is a quadratic program. Historically, QP solvers were first used to solve SVM problems. However, as these scale poorly to large problems, a huge amount of effort has been devoted to faster solvers. (Often based on coordinate ascent and/or online optimization). This area is still evolving. However, software is widely available now for solvers that are quite fast in practice.

Now, as we saw above that $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$, we can classify new points $\mathbf{x}$ by

$$f(\mathbf{x}) = \sum_i \alpha_i y_i \mathbf{x} \cdot \mathbf{x}_i$$

Clearly, this can be kernelized. If we do so, we can compute the Lagrange multipliers by **the SVM optimization**

$$\boxed{\max_{\mathbf{0} \leq \boldsymbol{\alpha} \leq \mathbf{c}} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j),} \tag{4.2}$$

which is again a quadratic program. We can classify new points by **the SVM classification rule**

$$\boxed{f(\mathbf{x}) = \sum_i \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i).} \tag{4.3}$$

Since we have "kernelized" both the learning optimization, and the classification rule, we are again free to replace k with any of the variety of kernel functions we saw before.

Now, finally, we can define what a "support vector" is. Notice that Eq. 4.2 is the maximization of a quadratic function of $\boldsymbol{\alpha}$, under the "box" constraints that $\mathbf{0} \leq \boldsymbol{\alpha} \leq \mathbf{c}$. It often happens that $\boldsymbol{\alpha}$ "wants" to be negative (in terms ot the quadratic function), but is prevented from this by the constraints. Thus, $\boldsymbol{\alpha}$ is often *sparse*. This has some interesting consequences. First of all, clearly if $\alpha_i = 0$, we don't need to include the corresponding term in Eq. 4.3. This is potentially a big savings. If all α_i are nonzero, then we would need to explicitly compute the kernel function with all inputs, and our time complexity is similar to a nearest-neighbor method. If we only have a few nonzero α_i then we only have to compute a few kernel functions, and our complexity is similar to that of a normal linear method.

Another interesting property of the sparsity of $\boldsymbol{\alpha}$ is that non-support vectors don't affect the solution. Let's see why. What does it mean if $\alpha_j = 0$? Well, recall that the multiplier α_j is enforcing the constraint that

$$z_j \geq 1 - y_j \mathbf{w} \cdot \mathbf{x}_j. \tag{4.4}$$

If $\alpha_j = 0$ at the solution, then this means, informally speaking, that we didn't really need to enforce this constraint at all: If we threw it out of the optimization, it would still "automatically" be obeyed. How could this be? Recall that the original optimization in Eq. 4.1 is trying to minimize all the z_j. There are two things stopping it from flying down to $-\infty$: the constraint in Eq. 4.4 above, and the constraint that $z_j \geq 0$. If the constraint above can be removed with out changing the solution, then it must be that $z_j = 0$. Thus, $\alpha_j = 0$ implies that $1 - y_j \mathbf{w} \cdot \mathbf{x}_j \leq 0$, or, equivalently, that

$$y_j \mathbf{w} \cdot \mathbf{x}_j \geq 1.$$

Thus non-support vectors are points that are *very well* classified, that are comfortably on the right side of the linear boundary.

Now, imagine we take some $\mathbf{x}_j$ with $z_j = 0$, and remove it from the training set. It is pretty easy to see that this is equivalent to taking the optimization

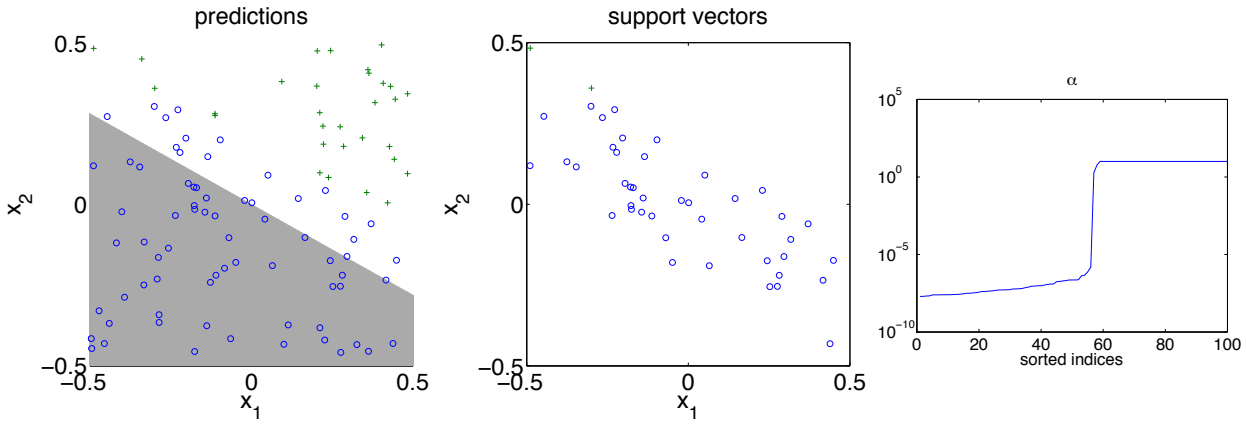
$$\begin{aligned} \min_{\mathbf{z},\mathbf{w}} \quad & c \sum_i z_i + \frac{1}{2}||\mathbf{w}||^2 \\ \text{s.t.} \quad & z_i \geq (1 - y_i \mathbf{w} \cdot \mathbf{x}_i) \\ & z_i \geq 0, \end{aligned}$$

and just dropping the constraint that $z_j \geq (1 - y_j \mathbf{w} \cdot \mathbf{x}_j)$, meaning that z_j "decouples" from the other variables, and the optimization will pick $z_j = 0$. But, as we saw above, this has no effect. Thus, removing a non-support vector from the training set has no impact on the resulting classification rule.

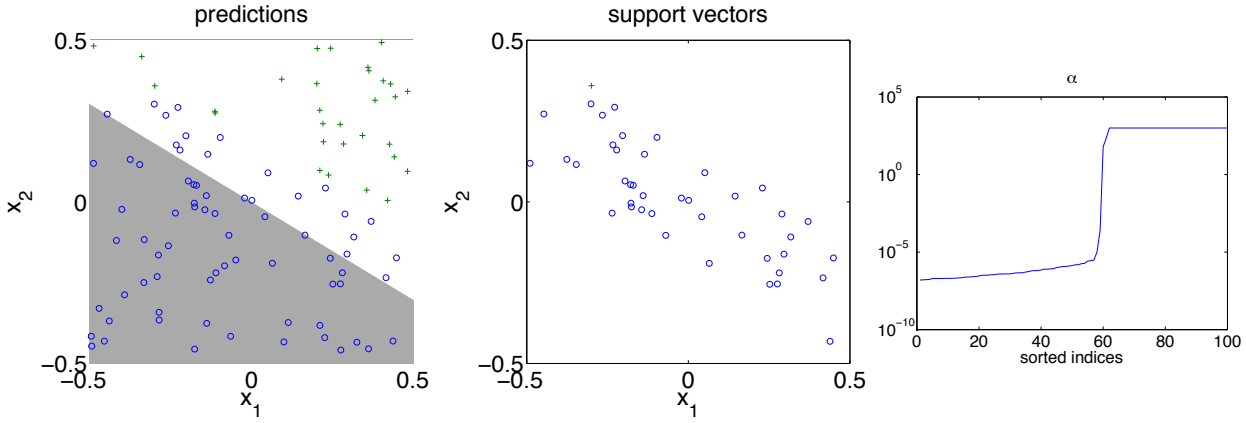
5 Examples

In class, we saw some examples of running SVMs. Here are many more.

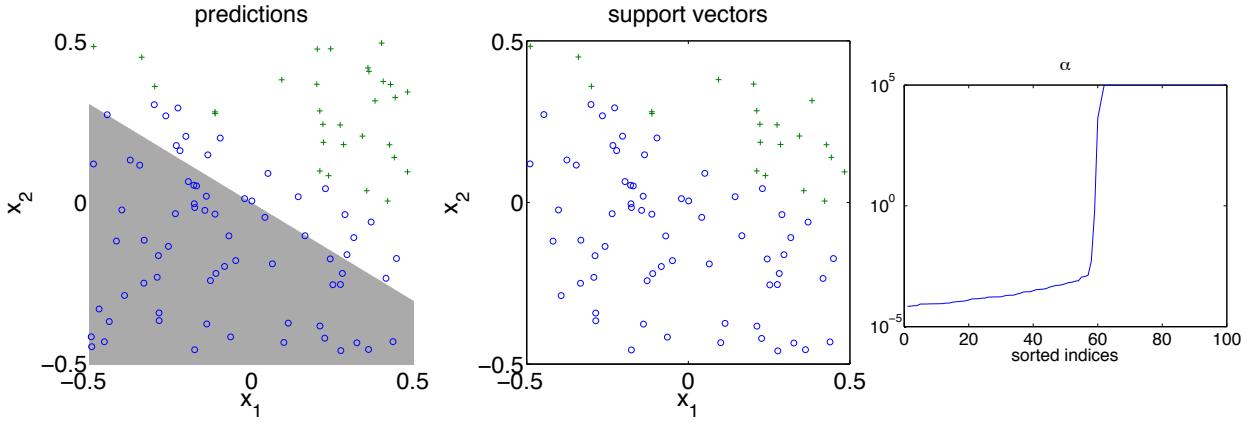
Dataset A, $c = 10$, $k(\mathbf{x}, \mathbf{v}) = \mathbf{x} \cdot \mathbf{v}$.



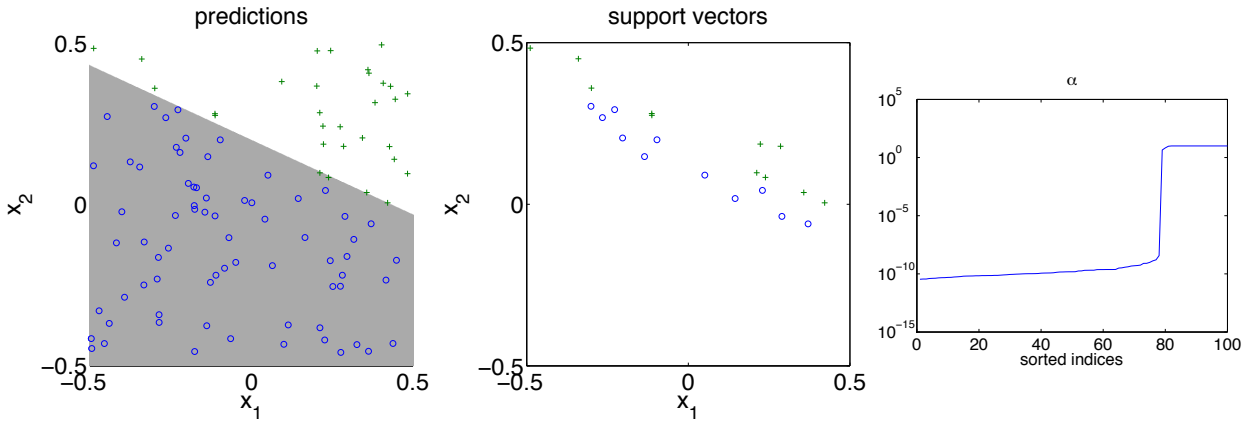
Dataset A, $c = 10^3$, $k(\mathbf{x}, \mathbf{v}) = \mathbf{x} \cdot \mathbf{v}$.



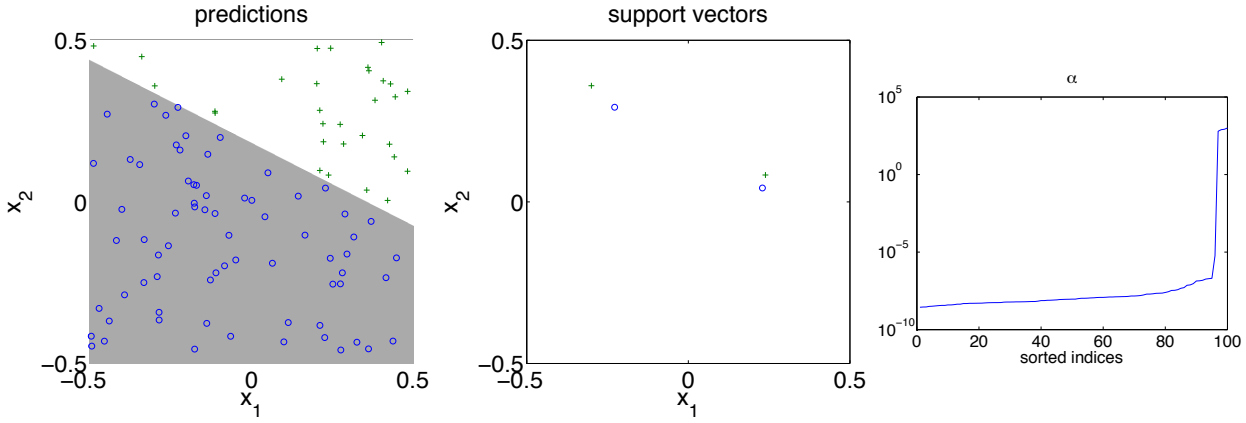
Dataset A, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \mathbf{x} \cdot \mathbf{v}$.



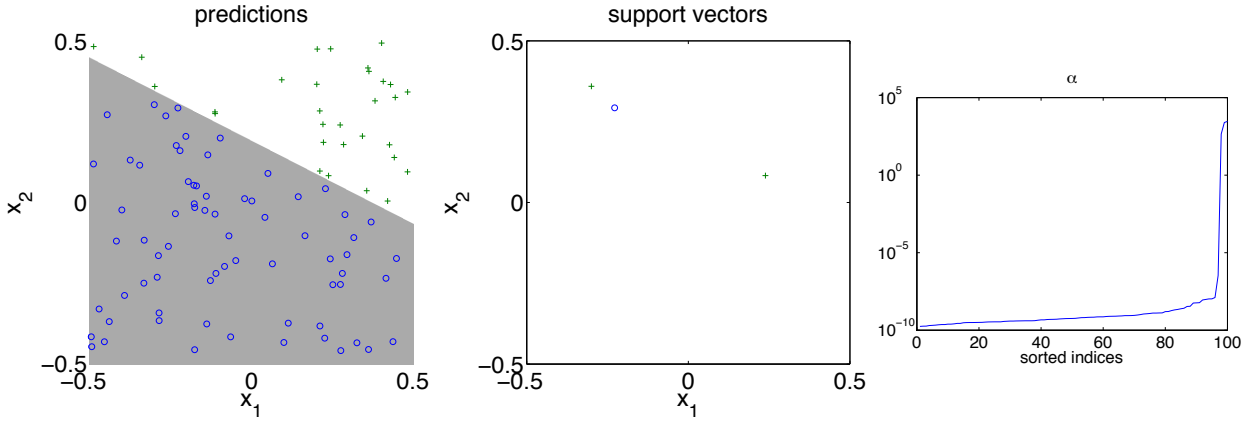
Dataset A, $c = 10$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



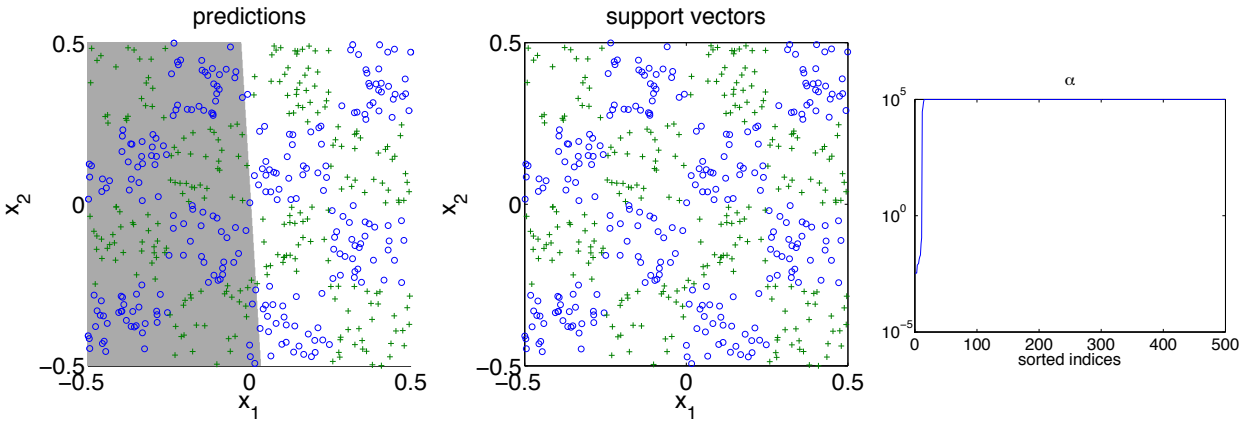
Dataset A, $c = 10^3$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



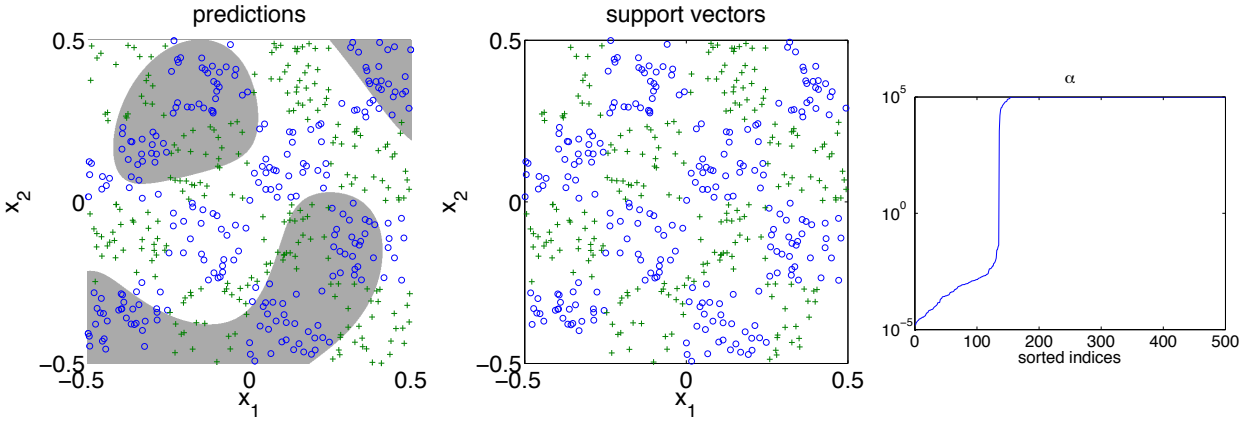
Dataset A, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



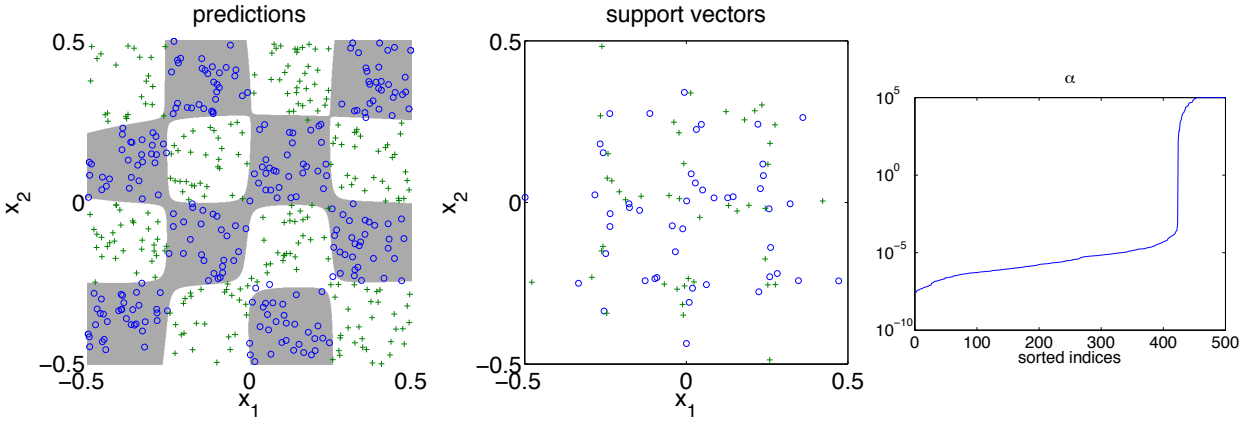
Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



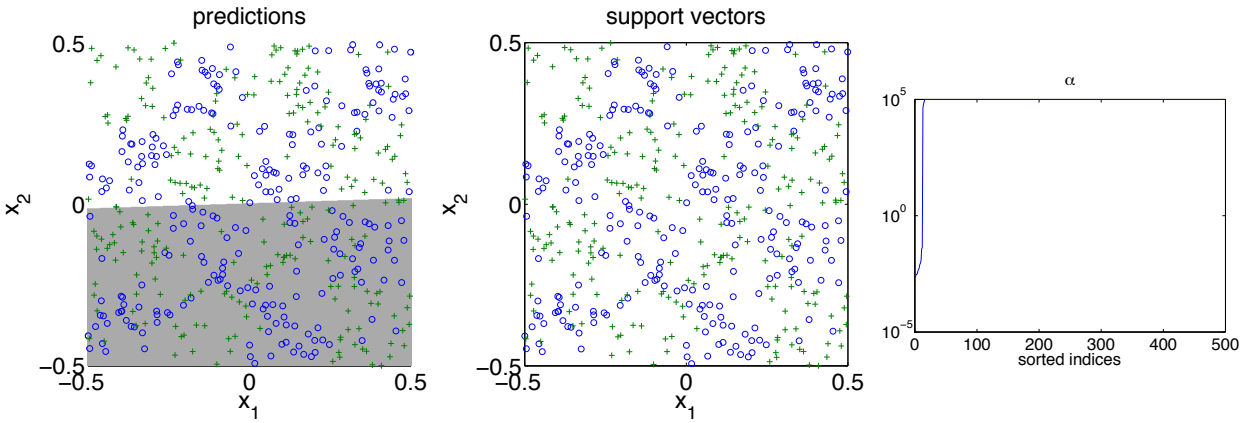
Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^5$.



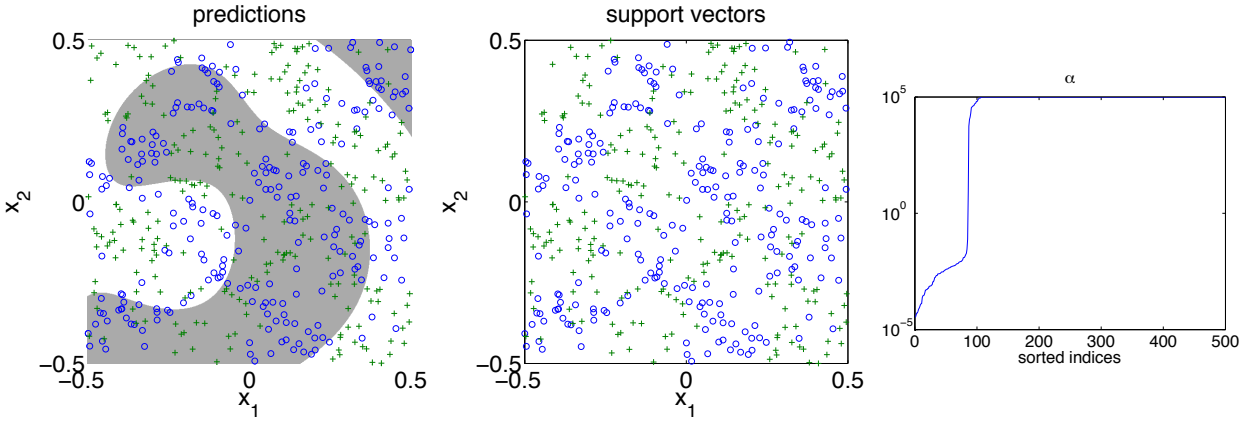
Dataset B, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^{10}$.



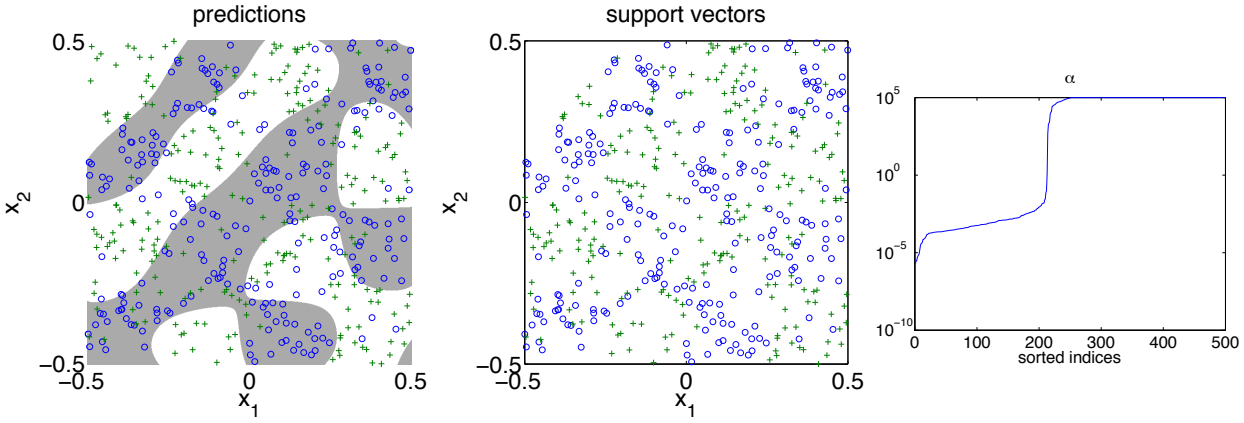
Dataset C (dataset B with noise), $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = 1 + \mathbf{x} \cdot \mathbf{v}$.



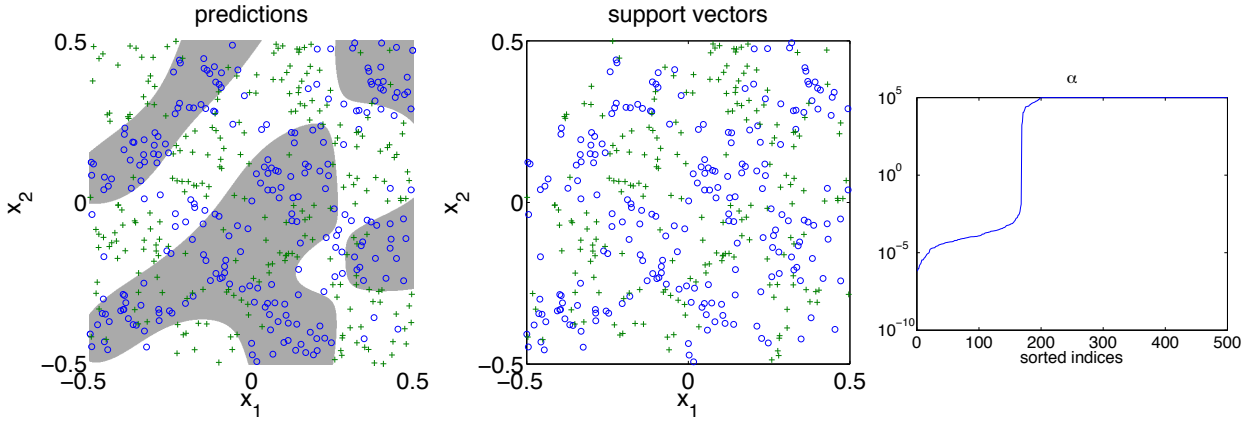
Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^5$.



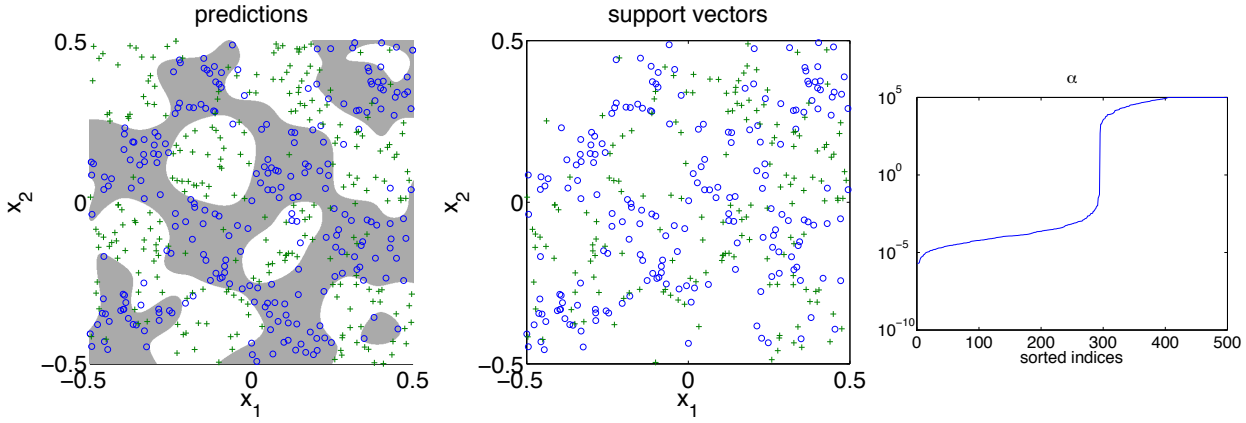
Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = (1 + \mathbf{x} \cdot \mathbf{v})^{10}$.



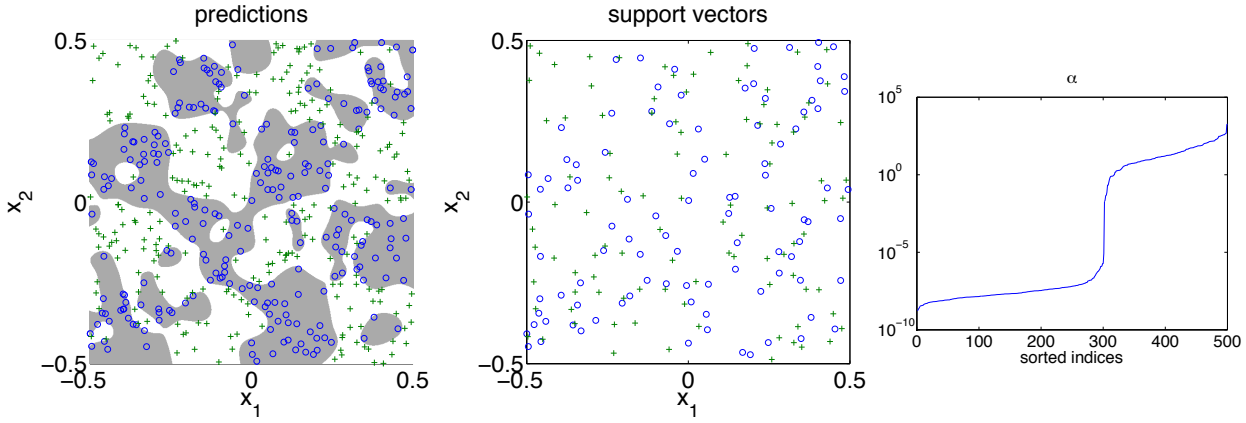
Dataset C (dataset B with noise), $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \exp\big(-2||\mathbf{x} - \mathbf{v}||^2\big)$.



Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \exp\big(-20||\mathbf{x} - \mathbf{v}||^2\big)$.



Dataset C, $c = 10^5$, $k(\mathbf{x}, \mathbf{v}) = \exp\big(-200||\mathbf{x} - \mathbf{v}||^2\big)$.



6 Kernel Theory

We now return to the issue of what makes a valid kernel $k(\mathbf{x}, \mathbf{v})$ where "valid" means there exists some feature space $\boldsymbol{\phi}$ such that

$$k(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v}).$$

6.1 Kernel algebra

We can construct complex kernel functions from simple ones, using an "algebra" of composition rules[3]. Interestingly, these rules can be understood from parallel compositions in feature space. To take an example, suppose we have two valid kernel functions k_a and k_b. If we define a new kernel function by

$$k(\mathbf{x}, \mathbf{v}) = k_a(\mathbf{x}, \mathbf{v}) + k_b(\mathbf{x}, \mathbf{v}),$$

k will be valid. To see why, consider the feature spaces $\boldsymbol{\phi}_a$ and $\boldsymbol{\phi}_b$ corresponding to k_a and k_b. If we define $\boldsymbol{\phi}$ by just concatenating $\boldsymbol{\phi}_a$ and $\boldsymbol{\phi}_b$, by

$$\boldsymbol{\phi}(\mathbf{x}) = (\boldsymbol{\phi}_a(\mathbf{x}), \boldsymbol{\phi}_b(\mathbf{x})),$$

then $\boldsymbol{\phi}$ is the feature space corresponding to k. To see this, note

$$\begin{aligned} \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v}) &= (\boldsymbol{\phi}_a(\mathbf{x}), \boldsymbol{\phi}_b(\mathbf{x})) \cdot (\boldsymbol{\phi}_a(\mathbf{v}), \boldsymbol{\phi}_b(\mathbf{v})) \\ &= \boldsymbol{\phi}_a(\mathbf{x}) \cdot \boldsymbol{\phi}_a(\mathbf{v}) + \boldsymbol{\phi}_b(\mathbf{x}) \boldsymbol{\phi}_b(\mathbf{v}) \\ &= k_a(\mathbf{x}, \mathbf{v}) + k_b(\mathbf{x}, \mathbf{v}) \\ &= k(\mathbf{x}, \mathbf{v}). \end{aligned}$$

We can make a table of kernel composition rules, along with the "dual" feature space composition rules.

kernel composition	**feature composition**
a) $k(\mathbf{x}, \mathbf{v}) = k_a(\mathbf{x}, \mathbf{v}) + k_b(\mathbf{x}, \mathbf{v})$	$\boldsymbol{\phi}(\mathbf{x}) = (\boldsymbol{\phi}_a(\mathbf{x}), \boldsymbol{\phi}_b(\mathbf{x}))$,
b) $k(\mathbf{x}, \mathbf{v}) = f k_a(\mathbf{x}, \mathbf{v})$, $f > 0$	$\boldsymbol{\phi}(\mathbf{x}) = \sqrt{f} \boldsymbol{\phi}_a(\mathbf{x})$
c) $k(\mathbf{x}, \mathbf{v}) = k_a(\mathbf{x}, \mathbf{v}) k_b(\mathbf{x}, \mathbf{v})$	$\phi_m(\mathbf{x}) = \phi_{ai}(\mathbf{x}) \phi_{bj}(\mathbf{x})$
d) $k(\mathbf{x}, \mathbf{v}) = \mathbf{x}^T A \mathbf{v}$, A positive semi-definite	$\boldsymbol{\phi}(\mathbf{x}) = L^T \mathbf{x}$, where $A = LL^T$.
e) $k(\mathbf{x}, \mathbf{v}) = \mathbf{x}^T M^T M \mathbf{v}$, M arbitrary	$\boldsymbol{\phi}(\mathbf{x}) = M\mathbf{x}$

[3]This material is based on class notes from Michael Jordan.

We have already proven rule (a). Let's prove some of the others.

Rule (b) is quite easy to understand:

$$\begin{aligned}\boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v}) &= \sqrt{f}\boldsymbol{\phi}_a(\mathbf{x}) \cdot \sqrt{f}\boldsymbol{\phi}_a(\mathbf{v}) \\ &= f\boldsymbol{\phi}_a(\mathbf{x}) \cdot \boldsymbol{\phi}_a(\mathbf{v}) \\ &= f k_a(\mathbf{x}, \mathbf{v}) \\ &= k(\mathbf{x}, \mathbf{v})\end{aligned}$$

Rule (c) is more complex. It is important to understand the notation. If

$$\begin{aligned}\boldsymbol{\phi}_a(\mathbf{x}) &= (\phi_{a1}(\mathbf{x}), \phi_{a2}(\mathbf{x}), \phi_{a3}(\mathbf{x})) \\ \boldsymbol{\phi}_b(\mathbf{x}) &= (\phi_{b1}(\mathbf{x}), \phi_{b2}(\mathbf{x})),\end{aligned}$$

then $\boldsymbol{\phi}$ contains all six pairs.

$$\begin{aligned}\boldsymbol{\phi}(\mathbf{x}) = \big(&\phi_{a1}(\mathbf{x})\phi_{b1}(\mathbf{x}), \phi_{a2}(\mathbf{x})\phi_{b1}(\mathbf{x}), \phi_{a3}(\mathbf{x})\phi_{b1}(\mathbf{x}) \\ &\phi_{a1}(\mathbf{x})\phi_{b2}(\mathbf{x}), \phi_{a2}(\mathbf{x})\phi_{b2}(\mathbf{x}), \phi_{a3}(\mathbf{x})\phi_{b2}(\mathbf{x})\big)\end{aligned}$$

With that understanding, we can prove rule (c) via

$$\begin{aligned}\boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v}) &= \sum_m \phi_m(\mathbf{x})\phi(\mathbf{v})_m \\ &= \sum_i \sum_j \phi_{ai}(\mathbf{x})\phi_{bj}(\mathbf{x})\phi_{ai}(\mathbf{v})\phi_{bj}(\mathbf{v}) \\ &= \big(\sum_i \phi_{ai}(\mathbf{x})\phi_{ai}(\mathbf{v})\big)\big(\sum_j \phi_{bj}(\mathbf{x})\phi_{bj}(\mathbf{v})\big) \\ &= \big(\boldsymbol{\phi}_a(\mathbf{x}) \cdot \boldsymbol{\phi}_a(\mathbf{v})\big)\big(\boldsymbol{\phi}_b(\mathbf{x}) \cdot \boldsymbol{\phi}_b(\mathbf{v})\big) \\ &= k_a(\mathbf{x}, \mathbf{v})k_b(\mathbf{x}, \mathbf{v}) \\ &= k(\mathbf{x}, \mathbf{v}).\end{aligned}$$

Rule (d) follows from the well known result in linear algebra that a symmetric positive semi-definite matrix A can be factored as $A = LL^T$. With that known, clearly

$$\begin{aligned}\boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v}) &= (L\mathbf{x}) \cdot (L\mathbf{v}) \\ &= \mathbf{x}^T L^T L\mathbf{v} \\ &= \mathbf{x}^T A^T \mathbf{v} \\ &= \mathbf{x}^T A\mathbf{v} \\ &= k(\mathbf{x}, \mathbf{v})\end{aligned}$$

We can alternatively think of rule (d) as saying that $k(\mathbf{x}) = \mathbf{x}^T M^T M\mathbf{x}$ corresponds to the basis expansion $\boldsymbol{\phi}(\mathbf{x}) = M\mathbf{x}$ for any M. That gives rule (e).

6.2 Understanding Polynomial Kernels via Kernel Algebra

So, we have all these rules for combining kernels. What do they tell us? Rules (a),(b), and (c) essentially tell us that *polynomial combinations* of valid kernels are valid kernels. Using this, we can understand the meaning of polynomial kernels First off, for some scalar variable x, consider a polynomial kernel of the form

$$k(x, v) = (xv)^d.$$

To what basis expansion does this kernel correspond? We can build this up stage by stage.

$$\begin{aligned}k(x, v) = xv &\leftrightarrow \phi(x) = (x) \\ k(x, v) = (xv)^2 &\leftrightarrow \phi(x) = (x^2) \text{ by rule (c)} \\ k(x, v) = (xv)^3 &\leftrightarrow \phi(x) = (x^3) \text{ by rule (c).}\end{aligned}$$

If we work with vectors, we find that

$$k(\mathbf{x}, \mathbf{v}) = (\mathbf{x} \cdot \mathbf{v})$$

corresponds to $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$, while (by rule (c)) $k(\mathbf{x}, \mathbf{v}) = (\mathbf{x} \cdot \mathbf{v})^2$ corresponds to a feature space with all pairwise terms

$$\phi_m(\mathbf{x}) = x_i x_j,\ 1 \leq i, j \leq n.$$

Similarly, $k(\mathbf{x}, \mathbf{v}) = (\mathbf{x} \cdot \mathbf{v})^3$ corresponds to a feature space with all triplets

$$\phi_m(\mathbf{x}) = x_i x_j x_k,\ 1 \leq i, j, k \leq n.$$

More generally, $k(\mathbf{x}, \mathbf{v}) = (\mathbf{x} \cdot \mathbf{v})^d$ corresponds to a feature space with terms

$$\phi_m(\mathbf{x}) = x_{i_1} x_{i_2} ... x_{i_d},\ 1 \leq i_j \leq n \tag{6.1}$$

Thus, a polynomial kernel is equivalent to a polynomial basis expansion, with all terms of order d. This is pretty surprising even though the word "polynomial" is in front of both of these terms! Again, we should reiterate the computational savings here. In general, computing a polynomial basis expansion will take time $O(n^d)$. However, computing a polynomial kernel only takes time $O(n)$. Again, though, we have only defeated the *computational* issue with high-degree polynomial basis expansions. The statistical properties are unchanged.

Now, consider the kernel $k(\mathbf{x}, \mathbf{v}) = (r + \mathbf{x} \cdot \mathbf{v})^d$. What is the impact of adding the constant of r? Notice that this is equivalent to simply taking the vectors $\mathbf{x}$ and $\mathbf{v}$ and prepending a constant of $\sqrt{r}$ to them. Thus, this kernel corresponds to a polynomial expansion with constant terms added. One way to write this would be

$$\phi_m(\mathbf{x}) = x_{i_1} x_{i_2} ... x_{i_d},\ 0 \leq i_j \leq n \tag{6.2}$$

where we consider x_0 to be equal to $\sqrt{r}$. Thus, this kernel is equivalent to a polynomial basis expansion with all terms of all order less than or equal to d.

An interesting question is the impact of the constant r. Should we set it large or small? What is the impact of this choice. Notice that the lower-order terms in the basis expansion in Eq. 6.2 will have many terms x_0, and so get multiplied by a $\sqrt{r}$ to a high power. Meanwhile, high order terms will have have few or no terms of x_0, and so get multiplied by $\sqrt{r}$ to a low power. Thus, a large factor of r has the effect of making the low-order term larger, relative to high-order terms.

Recall that if we make the basis expansion larger, this has the effect of reducing the the regularization penalty, since the same classification rule can be accomplished with a smaller weight. Thus, if we make *part* of a basis expansion larger, those parts of the basis expansion will tend to play a larger role in the final classification rule. Thus, using a larger constant r had the effect of making the low-order parts of the polynomial expansion in Eq. 6.2 tend to have more impact.

6.3 Mercer's Theorem

One thing we might worry about is if the SVM optimization is convex in α. The concern is if

$$\sum_i \sum_j \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \tag{6.3}$$

is convex with respect to $\boldsymbol{\alpha}$. We can show that *if* k is a valid kernel function, then the kernel matrix K must be positive semi-definite.

$$\begin{aligned}
\mathbf{z}^T K \mathbf{z} &= \sum_i \sum_j z_i K_{ij} z_j \\
&= \sum_i \sum_j z_i \boldsymbol{\phi}(\mathbf{x}_i) \cdot \boldsymbol{\phi}(\mathbf{x}_j) z_j \\
&= \sum_i z_i \boldsymbol{\phi}(\mathbf{x}_i) \cdot \sum_j \boldsymbol{\phi}(\mathbf{x}_j) z_j \\
&= ||\sum_i z_i \boldsymbol{\phi}(\mathbf{x}_i)||^2 \\
&\geq 0
\end{aligned}$$

We can also show that, if K is positive semi-definite, then the SVM optimization is concave. The thing to see is that

$$\begin{aligned}
\sum_i \sum_j \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) &= \boldsymbol{\alpha}^T \text{diag}(\mathbf{y}) K \text{diag}(\mathbf{y}) \boldsymbol{\alpha} \\
&= \boldsymbol{\alpha}^T M \boldsymbol{\alpha},
\end{aligned}$$

where $M = \text{diag}(\mathbf{y}) K \text{diag}(\mathbf{y})$. It is not hard to show that M is positive semi-definite.

So this is very nice– if we use any valid kernel function, we can be assured that the optimization that we need to solve in order to recover the Lagrange multipliers $\boldsymbol{\alpha}$ will be concave. (The equivalent of convex when we are doing a maximization instead of a minimization.)

Now, we still face the question– do there exist *invalid* kernel functions that also yield positive semi-definite kernel matrices? It turns out that the answer is *no*. This result is known as **Mercer's theorem**.

> A kernel function is valid *if and only if* the corresponding kernel matrix is positive semi-definite for all training sets $\{\mathbf{x}_i\}$.

This is very convenient– the valid kernel functions are exactly those that yield optimization problems that we can reliably solve. However, notice that Mercer's theorem refers to *all* sets of points $\{\mathbf{x}_i\}$. An invalid kernel can yield a positive semi-definite kernel matrix for some particular training set. All we know is that, for an invalid kernel, there is some training set that yields a non positive semi-definite kernel matrix.

7 Our Story so Far

There were a lot of technical details here. It is worth taking a look back to do a conceptual overview and make sure we haven't missed the big picture. The starting point for SVMs is is minimizing the hinge loss under ridge regression, i.e.

$$\mathbf{w}^* = \min_{\mathbf{w}} c \sum_i (1 - y_i \mathbf{w} \cdot \mathbf{x}_i)_+ + \frac{1}{2}||\mathbf{w}||^2.$$

Fundamentally, SVMs are just fitting an optimization like this. The difference is that they perform the optimization in a different way, and they allow you to work efficiently with powerful basis expansions / kernel functions.

Act 1. We proved that if $\mathbf{w}^*$ is the vector of weights that results from this optimization, then we could alternatively calculate $\mathbf{w}^*$ as

$$\mathbf{w}^* = \sum_i \alpha_i y_i \mathbf{x}_i,$$

where the α_i are given by the optimization

$$\max_{0 \leq \boldsymbol{\alpha} \leq c} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j. \tag{7.1}$$

With that optimization solved, we can classify a new point $\mathbf{x}$ by

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{w}^* = \sum_i \alpha_i y_i \mathbf{x} \cdot \mathbf{x}_i, \tag{7.2}$$

Thus, if we want to, we can think of the variables α_i as being the main things that we are fitting, rather than the weights $\mathbf{w}$.

Act 2. Next, we noticed that the above optimization (Eq. 7.3) and classifier (Eq. 7.4) only depend on the inner products of the data elements. Thus, we could replace the inner products in these expressions with kernel evaluations, giving the optimization

$$\max_{0 \leq \boldsymbol{\alpha} \leq c} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j), \tag{7.3}$$

and the classification rule

$$f(\mathbf{x}) = \sum_i \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i), \tag{7.4}$$

where $k(\mathbf{x}, \mathbf{v}) = \mathbf{x} \cdot \mathbf{v}$.

Act 3. Now, imagine that instead of directly working with the data, we wanted to work with some basis expansion. This would be easy to accomplish just by switching the kernel function to be

$$k(\mathbf{x}, \mathbf{v}) = \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{v}).$$

However, we also noticed that for some basis expansions, like polynomials, we could compute $k(\mathbf{x}, \mathbf{v})$ much more efficiently than explicitly forming the basis expansions and then taking the inner product. We called this computational trick the "kernel trick".

Act 4. Finally, we developed a "kernel algebra", which allowed us to understand how we can combine different kernel functions, and what this meant in feature space. We also saw Mercer's theorm, which tells us what kernel functions are and are not legal. Happily, this corresponded with the SVM optimization problem being convex.

8 Discussion

8.1 SVMs as Template Methods

Regardless of what we say, at the end of the day, support vector machines make their predictions through the classification rule

$$f(\mathbf{x}) = \sum_i \alpha_i y_i k(\mathbf{x}, \mathbf{x}_i).$$

Intuitively, $k(\mathbf{x}, \mathbf{x}_i)$ measures how "similar" $\mathbf{x}$ is to training example $\mathbf{x}_i$. This bears a strong resemblence to K-NN classification, where we use k as the distance metric, rather than something like the Euclidean distance. Thus, SVMs can be seen as glorified template methods, where the amount that each point $\mathbf{x}_i$ participates in predictions is reweighted in the learning stage. This is a view usually espoused by SVM skeptics, but a reasonable one. Remember, however, that there is absolutely nothing wrong with template methods.

8.2 Theoretical Issues

An advantage of SVMs is that rigorous theoretical guarantees can often be given for their performance. It is possible to use these theoretical bounds to do model selection, rather than, e.g., cross validation. However, at the moment, these theoretical guarantees are rather loose in practice, meaning that SVMs perform significantly *better* than the bounds can show. As such, one can often get better practical results by using more heuristic model selection procedures like cross validation. We will see this when we get to learning theory.