# Week 6: Support Vector Machines

Instructor: Sergey Levine

## 1 Support vector machines recap

The support vector machine (SVM) optimization is defined as

$$\begin{aligned}&\min_{\mathbf{w},w_0,s_1,\ldots,s_N} \frac{1}{2}||\mathbf{w}||^2 + \lambda \sum_{i=1}^{N} s_i \text{ such that} \\ &y^i(\mathbf{w}\cdot h(\mathbf{x}^i) + w_0) + s_i \geq 1 \quad \forall i \in \{1,\ldots,N\} \\ &s_i \geq 0 \quad \forall i \in \{1,\ldots,N\}\end{aligned}$$

As we saw in the previous lecture, solving this optimization recovers a linear classifier of the form $y = \text{sign}(\mathbf{w}\cdot h(\mathbf{x}) + w_0)$ that minimizes the hinge loss for all misclassified points and maximizes the size of the margin (the distance to the closest point to the decision boundary). The term "support vector" refers to the vectors from the decision boundary to the closest points. Note that moving any point that is correct classified and further from the decision boundary than the margin will not affect the optimal weights, hence the term "support vector:" these vectors "support" the boundary, while all others do not.

## 2 Revisiting features

Note that SVMs are still linear classifiers, but just like we saw with logistic and linear regression, "linear" refers to the function being linear in the weights, not necessarily in the input $\mathbf{x}$, since we can use a complex feature vector $h(\mathbf{x})$ to create decision boundaries that are not linear in the input space. This idea is very powerful. For example, even low-order polynomial features can allow for very complex decision boundaries, and SVMs are great at learning good large-margin decision boundaries for high-dimensional features. This increases the risk of overfitting, but can allow us to learn classifiers that would be impossible otherwise.

Unfortunately, if the input is high-dimensional, incorporating highly expressive features like polynomials can dramatically increase the size of the feature vector. For example, if we add all monomial features of degree $D$ (i.e. $x_1x_2x_3^2$ is a monomial of degree 4, so is $x_1^4$ or $x_2^2x_3^2$), and the dimensionality of $\mathbf{x}$ is $M$, the total number of features is:

$$\left(\begin{array}{c} D+M-1 \\ D \end{array}\right) = \frac{(D+M-1)!}{D!(M-1)!}.$$

So even for relatively low-order polynomials, the number of features can be very large. For example, if $D = 6$ and $M = 100$, we get about 1.6 *billion* features. We could design our features more carefully to manually select just the ones that matter, but there is a better way. Note that in the SVM, every time we see $\mathbf{w}$ or $h(\mathbf{x})$, they always appear as a dot product: we have $\mathbf{w} \cdot h(\mathbf{x}^i)$ and $||\mathbf{w}||^2 = \mathbf{w} \cdot \mathbf{w}$. So if we had an efficient way to evaluate dot products, maybe we can avoid actually enumerating all 1.6 billion features!

Note that even if the number of features is huge, taking a dot product of two feature vectors might not be so bad. Say we have two-dimensional inputs $(u_1, u_2)$ and $(v_1, v_2)$. If our feature vector is $h(\mathbf{u}) = [u_1, u_2]$ – that is, all monomials of degree 1 – then clearly $h(\mathbf{u}) \cdot h(\mathbf{v}) = \mathbf{u} \cdot \mathbf{v}$. Now let's say we have $h(\mathbf{u}) = [u_1^2, u_1u_2, u_2u_1, u_2^2]$: that is, all monomials of degree 2. What is $h(\mathbf{u}) \cdot h(\mathbf{v})$? Well, we can evaluate it:

$$h(\mathbf{u}) \cdot h(\mathbf{v}) = \begin{bmatrix} u_1^2 \\ u_1u_2 \\ u_2u_1 \\ u_2^2 \end{bmatrix} \cdot \begin{bmatrix} v_1^2 \\ v_1v_2 \\ v_2v_1 \\ v_2^2 \end{bmatrix} = u_1^2v_1^2 + 2u_1u_2v_1v_2 + u_2^2v_2^2 = (u_1v_1 + u_2v_2)^2.$$

But the last expression is simply $(\mathbf{u} \cdot \mathbf{v})^2$. Indeed, we can show that for any feature vector that consists of monomials of degree $D$, $h(\mathbf{u}) \cdot h(\mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})^D$, which means that if we only ever take dot products of our feature vectors, we can have features with some huge degree $D$ and still have compute times that are *independent* of $D$.

## 3 Kernels

Kernels are basically functions of the form $K(\mathbf{u}, \mathbf{v})$ that compute the dot product in some high-dimensional feature space of the vectors $h(\mathbf{u})$ and $h(\mathbf{v})$ without ever forming the feature vectors explicitly.

**Question.** Can we define a kernel $K(\mathbf{u}, \mathbf{v})$ for the case where $h(\mathbf{u})$ contains all monomials of degree $D$?

**Answer.** Based on the derivation above, we can define $K(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})^D$. Note that $D$ here is a *hyperparameter* of the kernel, and we can choose it either using our intuition or using cross-validation (in the context of a model like kernalized SVMs, which we'll discuss next).

Here are a few examples of commonly used kernels:

**1.** $K(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})^D$: as we saw above, this corresponds to $h(\mathbf{u})$ containing all monomials of degree $D$.

**2.** $K(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v} + 1)^D$: this corresponds to $h(\mathbf{u})$ containing all monomials *up to* degree $D$.

**3.** $K(\mathbf{u},\mathbf{v}) = \exp\left(-\frac{||\mathbf{u}-\mathbf{v}||^2}{2\ell}\right)$: this kernel, called the squared exponential or Gaussian kernel, is very commonly used and corresponds to an infinite feature space $h(\mathbf{u})$. This infinite feature space is formed by using features of the following form:

$$h_c(\mathbf{u}) = \exp\left(-\frac{||\mathbf{u}-c||^2}{2\ell}\right)$$

This is a radial basis function feature with a center at $c$. It is 1 if $\mathbf{u} = c$, and falls off as $\mathbf{u}$ gets further away. If we tile the centers $c$ in a regular grid pattern, we can think of these features as a kind of soft discretization of the space. The squared exponential kernel can be shown to be the dot product of two feature vectors consisting of these radial basis function features, with an *infinite* number of centers that densely tile the entire space on which $\mathbf{u}$ is defined!

## 4 Kernalized SVMs

Now that we know how to evaluate dot products of feature vectors efficiently in feature spaces that are extremely large (or even infinite in size), how can we use this "kernel trick" inside the SVM? First, we have to figure out how to represent the weights $\mathbf{w}$, since the size of the weights vector is equal to the number of features (which is now huge or even infinite).

To derive an idea for how to do this, let's briefly go back to the perceptron algorithm (and forget about the margin for now). Remember that for the perceptron, we trained the weights $\mathbf{w}$ by incrementing for them for all incorrectly classified datapoints according to:

$$\mathbf{w} \leftarrow \mathbf{w} + y^i h(\mathbf{x}^i).$$

What if instead of storing $\mathbf{w}$ directly, we simply store a weight on each datapoint, denoted $\alpha_i$? Then we can always recover $\mathbf{w}$ as:

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i h(\mathbf{x}^i),$$

and the perceptron algorithm simply increments the weight on each incorrectly classified datapoint according to

$$\alpha_i \leftarrow \alpha_i + y^i.$$

That way, when we recover $\mathbf{w} = \sum_{i=1}^{N} \alpha_i h(\mathbf{x}^i)$ we get exactly the same answer. We can do the same thing for the SVM (I won't derive this, but it's called the Representer Theorem), and then all we have to do when we take a dot product with the feature weights is:

$$\mathbf{w} \cdot h(\mathbf{x}^j) = \sum_{i=1}^{N} \alpha_i (h(\mathbf{x}^i) \cdot h(\mathbf{x}^j)) = \sum_{i=1}^{N} \alpha_i K(\mathbf{x}^i, \mathbf{x}^j),$$

and we never have to explicitly represent $\mathbf{w}$! So the parameters of our hypothesis class are now the weights $\alpha_i$, and our method is now *non-parametric*, which means we never explicitly store the parameters $\mathbf{w}$, but only a weight $\alpha_i$ on each datapoint. If the number of features is much larger than the size of the dataset, this makes our method more efficient. To recover the kernalized SVM, we now only need to evaluate

$$||\mathbf{w}||^2 = \mathbf{w} \cdot \mathbf{w} = \left[\sum_{i=1}^{N} \alpha_i h(\mathbf{x}^i)\right] \cdot \left[\sum_{j=1}^{N} \alpha_j h(\mathbf{x}^j)\right] = \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j K(\mathbf{x}^i, \mathbf{x}^j)$$

If we construct a matrix $\mathbf{K}$ such that $\mathbf{K}_{ij} = K(\mathbf{x}^i, \mathbf{x}^j)$, then we can write the above equation in matrix notation as

$$||\mathbf{w}||^2 = \alpha^T \mathbf{K} \alpha,$$

where $\alpha$ is the vector of weights. So the full kernalized SVM optimization is given by

$$\begin{aligned}
&\min_{\alpha, w_0, s_1, \ldots, s_N} \frac{1}{2}\alpha^T \mathbf{K}\alpha + \lambda \sum_{i=1}^{N} s_i \text{ such that} \\
&y^i \left( \left[ \sum_{j=1}^{N} \alpha_j K(\mathbf{x}^i, \mathbf{x}^j) \right] + w_0 \right) + s_i \geq 1 \quad \forall i \in \{1, \ldots, N\} \\
&s_i \geq 0 \quad \forall i \in \{1, \ldots, N\}
\end{aligned}$$

Note that the constraints are still linear in $\alpha$, $w_0$, and $s$, and the objective is still quadratic in all of the variables, so we can still solve this optimization problem using any standard QP solver, or specialized solvers for SVMs. We can also derive an unconstrained objective just like we did before, by noting that

$$s_i = \max \left( 0, 1 - y^i \left( \left[ \sum_{j=1}^{N} \alpha_j K(\mathbf{x}^i, \mathbf{x}^j) \right] + w_0 \right) \right),$$

giving the unconstrained problem

$$\min_{\alpha, w_0} \frac{1}{2}\alpha^T \mathbf{K}\alpha + \lambda \sum_{i=1}^{N} \max \left( 0, 1 - y^i \left( \left[ \sum_{j=1}^{N} \alpha_j K(\mathbf{x}^i, \mathbf{x}^j) \right] + w_0 \right) \right).$$

## 5 SVM Interactive Demo

See this page: `http://cs.stanford.edu/people/karpathy/svmjs/demo/`

Try both linear and RBF (squared exponential) kernels. Here, $C$ denotes $\lambda$, the weight on the slack variables.

## 6 Multiclass SVMs

Lastly, we'll briefly discuss how we can use SVMs when we have more than two classes. There are two main approaches we'll discuss: (1) one-against-all classifiers and (2) multiclass SVMs.

One-against-all classification is the simplest way to adapt SVMs to multiclass classification. In this scheme, instead of solving a single learning problem with $L_y$ classes, we instead solve $L_y$ binary problems, each of which requires us to classify the current class $j$ against *all* other classes. So we simply construct $L_y$ datasets, for each of which the label is $y^i_j = \delta(y^i = j)$, and we get $L_y$ weight vectors $\mathbf{w}_1, \ldots, \mathbf{w}_{L_y}$. Now we just need to figure out how to classify a new point $\mathbf{x}^\star$. The idea is very simple: the further the point is from the decision boundary in the "positive" direction, the more likely we think it is to belong to that class. So we simply choose the point for which the point is furthest from the boundary in the positive direction, and set the class according to:

$$y^\star = \arg\max_j h(\mathbf{x}^\star) \cdot \mathbf{w}_j.$$

One-against-all classification is reasonable and can work quite well, though it requires training multiple SVMs. Next week, we'll also talk about how we can build multiclass SVMs directly.