Support Vector Machines: Kernels

Carlo Tomasi

August 22, 2019

As we saw in the previous note, a Support Vector Machine (SVM) is a linear classifier with decision rule

\[ \hat{y} = h(x) = \text{sign}(w^T x + b^*) . \]  

(1)

The parameters \( w^*, b^* \) of this rule are the solution to the quadratic program

\[ \min_{w,b,\xi} f(w,\xi) \quad \text{where} \quad f(w,\xi) = \frac{1}{2} \|w\|^2 + C \sum_{n=1}^{N} \xi_n \]

subject to the constraints

\[ y_n(w^T x_n + b) - 1 + \xi_n \geq 0 \]
\[ \xi_n \geq 0 . \]

By the representer theorem, the vector \( w^* \) in the unique solution of this problem can be written as follows:

\[ w^* = \sum_{n \in A(w^*,b^*)} \alpha_n y_n x_n \quad \text{with} \quad \alpha_j^*, \beta_k^* \geq 0 \]  

(2)

where \( A(w^*,b^*) \) is the set of support vector indices. These are training samples that the resulting classifier 1 either classifies incorrectly or classifies correctly but with a margin no greater than the reference margin \( \mu^* = 1/\|w^*\| \).

From this expression for \( w^* \) we obtain

\[ \|w^*\|^2 = w^T w^* = \sum_{m \in A(w^*,b^*)} \sum_{n \in A(w^*,b^*)} \alpha_m^* \alpha_n^* y_m y_n x_m^T x_n \]

and

\[ w^T x_n = \sum_{m \in A(w^*,b^*)} \alpha_m^* y_m x_m^T x_n . \]

Therefore, we will not miss the optimal solution if we rewrite the SVM quadratic program above in the following form:

\[ \min_{w,b,\xi} f(w,\xi) \quad \text{where} \quad f(w,\xi) = \frac{1}{2} \sum_{m \in A(u)} \sum_{n \in A(u)} \alpha_m \alpha_n y_m y_n x_m^T x_n + C \sum_{n=1}^{N} \xi_n \]  

(3)
subject to the constraints

\[
y_n \left( \sum_{m \in A(u)} \alpha_m y_m x_m^T x_n + b \right) - 1 + \xi_n \geq 0 \quad (4)
\]

\[
\xi_n \geq 0 . \quad (5)
\]

The decision rule 1 can similarly be rewritten as follows:

\[
\hat{y} = h(x) = \text{sign} \left( \sum_{m \in A(w^*, b^*)} \alpha_m^* y_m x_m^T x_n + b^* \right) . \quad (6)
\]

As a consequence, both training and testing involve the data samples \( x_n \) exclusively through their inner products \( x_m^T x_n \). We now explore why this fact is so important.

We saw in an earlier note that whether the classes in a binary classification problem are linearly separable depends on the representation of the data space \( X \). For instance, consider a problem in which \( X = \mathbb{R}^2 \) and the separating boundary is the parabola \( x_2 = x_1^2 \). Then, the classification rule is

\[
h(x) = \begin{cases} 
1 & \text{if } x_2 \geq x_1^2 \\
-1 & \text{otherwise.}
\end{cases}
\]

These classes are of course not linearly separable. However, one can transform the data by replacing every point \( x = (x_1, x_2) \) to a new point \( z = (z_1, z_2) = (x_1^2, x_2) \), and now the decision boundary becomes \( z_2 = z_1 \), which is a straight line. The new problem is separable.

An alternative to changing the representation, as we did above, is to augment it instead. For instance, we could have defined \( z = (z_1, z_2, z_3) = (x_1, x_2, x_1^2) \). Here, we added the feature \( x_1^2 \), rather than replacing \( x_1 \) with it.

Augmentation seems wasteful: The new representation is redundant, and computations become more complex. In addition, augmentation increases the dimensionality of \( X \), and we are now courting the curse of dimensionality. In other words, we increased both the computational complexity (how many operations to perform) and the sample complexity (how many training samples are needed) of the problem. Why think of augmentation, then? The reason is that, when faced with a new classification problem, we do not know the decision boundary, so it is generally hard to guess a transformation that will make the problem linearly separable if it is not. Instead, we can add many transformations of the variables, in the hope that a combination of some of those might make the problem linearly separable.

For instance, we could add all monomials of the original variables up to some degree \( k \). If \( k = 3 \) and \( x = (x_1, x_2) \), we know from earlier theory that we have

\[
d' = \binom{d + k}{d} = \binom{2 + 3}{2} = 10
\]

monomials:

\[
1, \ x_1, \ x_2, \ x_1^2, \ x_1 x_2, \ x_2^2, \ x_1^3, \ x_1^2 x_2, \ x_1 x_2^2, \ x_2^3 . \quad (7)
\]

From Taylor’s theorem, we know that if the degree \( k \) is large enough, we can approximate any hypersurface to any degree of accuracy. Therefore, it is more than a hope that augmentation will help: It’s a theorem!

But what about sample complexity and computational complexity? We will see in the next two sections that SVMs help address both sample and computational complexity.
1 Sample Complexity

This Section conveys the gist of what would take several lectures to work out in detail, so there will be no proofs. The goal of this Section is to explain what “sample complexity” means for a machine learning problem. If you are interested, you can find a good treatment of these topics in a recent book [3].

Intuitively, the more training samples we have, the better a given training algorithm should be able to do in terms of generalization, because a large training set is a more faithful and detailed proxy for all the data we might encounter in the future. Sample complexity measures how fast the size $N$ of a training set needs to grow as we make more and more exacting demands on the ability of the classifier to generalize. Formalizing this notion requires some care, as we now see.

Recall that we assume that all data, both that in the training set and that we encounter when we deploy the algorithm, comes from a given but unknown probability distribution $p(x, y)$, which we called the model of the problem. At training time, we are given a training set $T$ of size $N$, and the training algorithm finds the classifier $\hat{h}$ in the hypothesis space $H$ that minimizes the empirical risk $L_T(h)$:

$$\hat{h} = \text{ERM}_T(H) \in \arg\min_{h \in H} L_T(h).$$

The empirical risk achieved by $\hat{h}$ on $T$ is the smallest one on $H$, so that

$$L_T(\hat{h}) = L_T(H) = \min_{h \in H} L_T(h).$$

When deployed, that classifier will achieve a statistical risk

$$L_p(\hat{h}) = \mathbb{E}_p[\ell(y, \hat{h}(x))]$$

that is generally higher than the empirical risk $L_T(\hat{h})$.

More importantly, the statistical risk $L_p(\hat{h})$ achieved by training our classifier on $T$ will also be higher than the smallest achievable statistical risk in $\mathcal{H}$ and under the given model $p(x, y)$,

$$L_p(\mathcal{H}) = \min_{h \in \mathcal{H}} L_p(h) \quad \text{where} \quad L_p(h) = \mathbb{E}_p[\ell(y, h(x))].$$

The statistical risk $L_p(\hat{h})$ is higher than $L_p(\mathcal{H})$ because $\hat{h}$ was trained on $T$, and $T$ is a poor proxy for $p(x, y)$.

**Notation:** There are at least three risks here, so let us keep them straight. To summarize: The classifier $\hat{h}$ is found by minimizing the empirical risk on $T$, resulting in a value $L_T(\hat{h})$ for the empirical risk. This classifier achieves a statistical risk $L_p(\hat{h})$ when deployed, and this statistical risk is typically greater than both the empirical risk $L_T(\hat{h})$ on $T$ and the smallest achievable statistical risk $L_p(\mathcal{H})$ for the given model $p$ and hypothesis space $\mathcal{H}$, because $T$ represents $p$ poorly.

The smallest achievable statistical risk $L_p(\mathcal{H})$ is generally not equal to zero. For instance, if $\mathcal{H}$ is the set of linear classifiers and the data is not linearly separable, then there is no way to achieve zero risk. However, the closer the classes are to being linearly separable, the smaller $L_p(\mathcal{H})$ will be for that $\mathcal{H}$.

Thus, in general, all we can hope to achieve is some statistical risk $L_p(\hat{h})$ that is equal to $L_p(\mathcal{H}) + \epsilon$ for some $\epsilon > 0$:

$$L_p(\hat{h}) = L_p(\mathcal{H}) + \epsilon \geq 0.$$
We would like $\epsilon$ to be small, and the price we pay to make $\epsilon$ smaller is to make the training set $T$ bigger, so that it is a more faithful representation of all the data we have not yet seen.

Loosely speaking, we can view $N$, the size of $T$, as a monotonically increasing function of $1/\epsilon$, which we think of as the degree of exactitude we want to achieve: To obtain a large value of $1/\epsilon$ (that is, a small $\epsilon$), we need to make $N$ large enough. Then, we say that a machine learning problem has a low sample complexity if $N$ grows slowly with $1/\epsilon$, that is, as $\epsilon$ shrinks.

A key difficulty in formalizing this notion, that is, when trying to speak less loosely, is that it is not possible to have absolute guarantees: The $N$ samples in $T$ are drawn at random out of $p(x, y)$, and if we are really unlucky we may end up with a particularly bad $T$ (a “statistical fluke”). In that case, the classifier $\hat{h}$ will perform more poorly than if we were given a “more typical” set of $N$ random samples out of $p$.

Because of this, we cannot demand to be within $\epsilon$ from $L_p(\mathcal{H})$ with absolute certainty, but we must settle for a probabilistic guarantee. Specifically, we ask to fall below risk $L_p(\mathcal{H}) + \epsilon$ with high probability. More formally, in addition to a number $\epsilon$ with $0 < \epsilon$, we also give a positive number $\delta$ with $0 < \delta < 1$, and we ask how large $N$ needs to be for it to be unlikely (as measured by the probability $\delta$) that the actual statistical risk $L_p(\hat{h})$ achieved by the classifier $\hat{h}$ is greater than our target $L_p(\mathcal{H}) + \epsilon$. In a formula, we ask for the following inequality to hold:

$$P[L_p(\hat{h}) \geq L_p(\mathcal{H}) + \epsilon] \leq \delta.$$  

(8)

Note the strict inequalities on $\delta$: This number is assumed to be less than 1, because it bounds a probability, so with $\delta = 1$ the bound would be moot. This number is also strictly positive, because we can never be certain to achieve the desired bound.

We are now ready to define sample complexity: The sample complexity of a given machine learning problem is the function $N_H(\epsilon, \delta)$ that specifies the smallest number $N$ of samples that are necessary to satisfy the inequality 8 for the given hypothesis space $\mathcal{H}$ and regardless of the model $p$. That is, $N(\epsilon, \delta)$ must be so large that the bound 8 holds for all $p$.

As an example, suppose that $\epsilon = 0.1$ and $\delta = 0.01$. Then, if $N$ is large enough for the inequality above to hold, we know that the probability that the loss is more than 0.1 above the minimum possible loss $L_p(\mathcal{H})$ is less than 1 percent. Stated differently, the loss falls within 0.1 from optimal with 99 percent probability.

The requirement that the bound be independent of the model $p(x, y)$ is rather strong. As a result, we will typically be able to bound $N$ from below (“it takes at least this many samples”) and with an asymptotic bound, rather than giving an exact expression for it.

**Sample Complexity for Linear Classifiers and for SVMs** Now that we know how to define sample complexity, let us see, without proof, what the complexity is for binary linear classifiers in general and for SVMs in particular. We will see that SVMs come with much better guarantees under a weak assumption on $X$.

For binary linear classifiers, it turns out that $N$ grows at least as fast as the following function:

$$\frac{d + \log(1/\delta)}{\epsilon}.$$

Thus, $N$ must grow linearly with $1/\epsilon$ and with the logarithm of $1/\delta$ for the bound 8 to hold. More importantly, $N$ must grow linearly with the dimension $d$ of the data space $X$: The more dimensions,
the more data we need. This linear dependence is not too bad, given what we know about the curse of dimensionality, and is the main reason why linear classifiers are so successful.

For SVMs, on the other hand, the situation is even better, if the data space \( X \subset \mathbb{R}^d \) is bounded\(^1\), that is, if there exists a sufficiently large \( d \)-dimensional hypersphere that contains all of \( X \). A formula for the actual sample complexity is harder to give in this case, because it depends on the radius of the hypersphere that bounds \( X \), the parameter \( \gamma \) used for regularization of the SVM training risk, and the margin between the two classes. However, and most importantly, the sample complexity of SVMs does not depend on the dimensionality \( d \) of \( X \).

Because of this advantage, when learning SVMs we need not worry about \( d \), and we are effectively free from the curse of dimensionality, as long as the data space \( X \) is bounded! The implication for data augmentation is that we do not need to worry about sample complexity when we add features to the data points \( x \).

2 Computational Complexity: Kernels

As we saw in the previous Section, sample complexity is not an issue when we augment the features for SVMs, as long as the data space is bounded. What about computational complexity? We now see how the representer theorem comes to the rescue.

The idea of feature augmentation can be formulated abstractly by saying that instead of using the original feature vector \( x \) as a data point, we use some higher-dimensional feature vector \( \varphi(x) \) obtained by some transformation of \( x \). Thus, we can see \( \varphi \) as a feature mapping from \( \mathbb{R}^d \) to \( \mathbb{R}^{d'} \), where \( d' > d \), and often \( d' \gg d \). If we use these new features, equations 3 through 6 can be written as follows:

\[
\min_{w,b,\xi} f(w,\xi) \quad \text{where} \quad f(w,\xi) = \frac{1}{2} \sum_{m \in A(u)} \sum_{n \in A(u)} \alpha_m \alpha_n y_m y_n \varphi(x_m)^T \varphi(x_n) + C \sum_{n=1}^{N} \xi_n
\]

subject to the constraints

\[
y_n \left( \sum_{m \in A(u)} \alpha_m y_m \varphi(x_m)^T \varphi(x_n) + b \right) - 1 + \xi_n \geq 0
\]

\[
\xi_n \geq 0.
\]

The decision rule 1 can similarly be rewritten as follows:

\[
\hat{y} = h(x) = \text{sign} \left( \sum_{m \in A(w^*,b^*)} \alpha_m^* y_m \varphi(x_m)^T \varphi(x_n) + b^* \right).
\]

The key point now is that (i) the quantity

\[
K(x_m, x_n) \overset{\text{def}}{=} \varphi(x_m)^T \varphi(x_n)
\]

\(^1\)This assumption is useful for SVMs, but would not help for general linear classifiers.
is a single number, regardless of the dimensionality $d'$ of the codomain of $\varphi$; and (ii) all the optimization algorithm needs to know is $K(x_m, x_n)$, because the individual terms $\varphi(x_n)$ do not appear anywhere in isolation.

The first idea for exploiting this observation is that if we have a mapping $\varphi$ in mind and somehow we find an inexpensive way to compute the inner product $K(x_m, x_n)$, then we can save computation. For instance, it is not clear how to do this with the monomials listed in expression 7. However, let us replace these by the following:

$$\varphi(x) = \varphi(x_1, x_2) = (1, \sqrt{3}x_1, \sqrt{3}x_2, \sqrt{3}x_1^2, \sqrt{3}x_2^2, x_1^3, \sqrt{3}x_1^2x_2, \sqrt{3}x_1x_2^2, x_2^3)^T.$$ 

These monomials are scaled versions of the original ones, and therefore span the same space. With this new definition, however, we can write

$$\text{(x}^T z + 1)^3 = (x_1 z_1 + x_2 z_2 + 1)^3$$

$$= 1 + 3x_1 z_1 + 3x_2 z_2 + 3x_1^2 z_1^2 + 6x_1 x_2 z_1 z_2 + 3x_2^2 z_2^2 + x_1^3 z_1^3$$

$$+3x_1^2 x_2 z_1 z_2 + 3x_1 x_2^2 z_1 z_2 + x_2^3 z_2^3$$

$$= \varphi(x)^T \varphi(z) = K(x, z).$$

Thus, instead of computing the ten products and nine sums necessary to compute the inner product of $\varphi(x)$ and $\varphi(z)$, we just compute the four products and two sums needed to compute $(x^T z + 1)^3$. This is of course a modest saving. However, as both the degree $k$ and the number $d$ of variables increase, we know that the savings become exponentially large.

However, there is a much neater idea for exploiting the observations made earlier: We can just come up with some kernel $K(x, z)$, without knowing what $\varphi$ generates it. Of course, we cannot just use any function $K$ of two vectors $x, z \in \mathbb{R}^d$, because $K$ needs to be indistinguishable from an inner product. For instance, inner products are symmetric:

$$x^T z = z^T x$$

and satisfy the Cauchy-Schwartz inequality:

$$(x^T z)^2 \leq \|x\|^2 \|z\|^2,$$

and therefore a valid kernel $K(x, z)$ must satisfy

$$K(x, z) = K(z, x) \quad \text{and} \quad K(x, z) \leq K(x, x) \cdot K(z, z).$$

However, these conditions are not enough to guarantee that $K(x, z)$ is the inner product in some vector space.

Fortunately, there is a characterization of all valid kernels in the literature [4]. If we have a finite set of vectors $x_n \in \mathbb{R}^d$ (as we do with a training set), then a symmetric function $K(x, z) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a kernel function on that set if and only if the matrix with entry $i, j$ equal to $K(x_i, x_j)$ is positive semi-definite.

There is also an analogous result for infinite sets. This is important, because we would like to know that a given $K(x, z)$ is a kernel without first having to choose a training set. In the continuous case, a continuous, symmetric function $K(x, z)$ is a kernel function, that is, it can be written as

$$K(x, z) = \varphi(x)^T \varphi(z).$$
for some mapping \( \varphi \) from \( \mathbb{R}^d \) to \( \mathbb{R}^{d'} \) if and only if for every function \( f : \mathbb{R}^d \to \mathbb{R} \) for which the integral 
\[
\int_{\mathbb{R}^d} f(x) \, dx \quad \text{is finite,}
\]
the following condition holds:
\[
\int_{\mathbb{R}^d \times \mathbb{R}^d} K(x, z) f(x) f(z) \, dx \, dz \geq 0.
\]

This condition is called Mercer’s condition, and you should be able to recognize it as an immediate extension of the notion of a positive semi-definite matrix to the continuous case. There is a whole theory that lets one check algorithmically whether this condition holds, and the theory is based on eigenfunctions, the functional equivalent of eigenvectors [4]. Suffice it to say that the Gaussian kernel
\[
K(x, z) = e^{-\frac{|x-z|^2}{\sigma^2}}
\]
satisfies Mercer’s condition, and is therefore a kernel. SVMs that use a Gaussian kernel are called Radial Basis Function (RBF) SVMs. Many ways to build new kernels can be found in the literature [1, 2]. You will see an example of using kernels (and Gaussian kernels in particular) in a homework assignment.

**Kernels and Support Vectors**  
As we know, the decision rule for an SVM is
\[
h(x) = \text{sign}(w^T x + b).
\]
and the separating hyperplane has equation
\[
w^T x + b = 0.
\]
In the transformed space \( \varphi(x) \), this reads
\[
w^T \varphi(x) + b = 0.
\]

We also saw that the representer theorem implies that the hyperplane parameters are a linear combination of the support vectors. Again, in the transformed space this reads
\[
w = \sum_{n=1}^{N} \alpha_n y_n \varphi(x_n),
\]
where the Lagrange multipliers \( \alpha_n \) are nonzero only for the support vectors.

Therefore, the separating hyperplane found by using kernel \( K(x, z) \) associated to the (generally unknown) transformation \( \varphi(x) \) can be written as follows:
\[
\sum_{n=1}^{N} \alpha_n y_n \varphi(x_n)^T \varphi(x) + b = 0.
\]
While we may not know what \( \varphi(x) \) is, we see that even in this equation features only show up through inner products, so we can write the equation above as follows:
\[
\sum_{n=1}^{N} \alpha_n y_n K(x_n, x) + b = 0,
\]
and we actually don’t need to know $\varphi(x)$ if we know the kernel. This expression is particularly instructive for Gaussian kernels, because the equation of the separating hyperplane then becomes

$$
\sum_{n=1}^{N} \alpha_n y_n e^{-\frac{||x-x_n||^2}{\sigma^2}} + b = 0.
$$

Geometrically, this amounts to placing a positive Gaussian, scaled by $\alpha_n$, at every support vector $x_n$ with a true positive label ($y_n = 1$), and a negative Gaussian, similarly scaled, at every support vector with a true negative label ($y_n = -1$). Thus, the positive support vectors are “pulling up” a function, and the negative support vectors are “pulling it down.” The signed Gaussians are added up, and the separating hyper-surface in the original space is the set of curves where this sum equals $-b$. Thus, with enough support vectors, an RBF SVM can follow arbitrarily complex boundaries, and the boundary weaves its way around the Gaussians between positive and negative support vectors.

References


