10-315: Introduction to Machine Learning

Maria-Florina Balcan

1 Kernels Methods in Machine Learning

1.1 Recap of Perceptron, Margins, and Online learning

In the online learning model, examples arrive sequentially, one at a time. After observing the ith example, the algorithm makes a prediction of its label, and then it is told the true label of the example and whether its prediction was correct or not. The goal of an online learning algorithm is to make as few mistakes as possible. In particular, in the Mistake-Bound Model, we make no distributional assumptions on the examples (we do not assume they are drawn iid) and aim to bound the number of mistakes as some function of the sequence.

The *Perceptron Algorithm* is an algorithm for learning linear separators in this model. Without loss of generality, we assume the linear separators are homogeneous (go through the origin). The algorithm begins with an initial weight vector $w_1 = \mathbf{0}$. Given an example x it predicts positive iff the dot-product of its current weight vector with the example is greater than or equal to 0. That is, if its current weight vector is w_t then it predicts positive iff $w_t \cdot x \geq 0$. If it makes a mistake, predicting negative on a positive example, it then adds the example to its weight vector, setting $w_{t+1} = w_t + x$. If it makes a mistake predicting positive on a negative example, then it subtracts the example from its weight vector, setting $w_{t+1} = w_t - x$.

Notice that at each point in time, w_t is a weighted sum of incorrectly classified examples, with weight $+1$ if the example was positive and weight -1 if the example was negative.

We define the (geometric) *margin* of an example x with respect to a linear separator w as the distance of x to the plane $w \cdot x = 0$. Algebraically, if $||w|| = 1$, then the margin of x can be written as $|x \cdot w|$. The margin γ_w of a *set* of examples S with respect to a linear separator w is the smallest margin out of all points $x \in S$. Finally, the margin γ of a set of examples S is the maximum γ_w over all linear separators w. That is, it is the margin of the largest-margin separator for set S.

What relates all these concepts is that we can give a *mistake bound* for the *Perceptron algorithm* purely in terms of the *margin* of the sequence of examples S, and with no dependence on the length of the sequence or the dimension of the space that the data lies in.

Theorem 1 *If data sequence S is linearly separable by margin γ, and all point lie inside a ball of radius R, then the Perceptron algorithm makes $\leq (R/\gamma)^2$ mistakes.*

1.2 Kernels

The above analysis assumes data is linearly separable (or at least approximately so). For the case that we have a more complex decision boundary, one approach that allows us to still utilize algorithms such as Perceptron is to apply a *kernel*.

A *kernel* K is a legal definition of dot-product. Specifically, let X denote the input space. Then a function $K : X \times X \to \Re$ is called a kernel if there exists a function Φ (an implicit mapping of datapoints to a possibly much higher dimensional space) such that for any two datapoints x, z, we have $K(x, z) = \Phi(x) \cdot \Phi(z)$.

An example of a kernel over the instance space $X = \Re^n$ is the function $K(x, z) = (x \cdot z)^2$. You can verify that if $X = \Re^2$ then $K(x, z) = \Phi(x) \cdot \Phi(z)$ for the implicit mapping $\Phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$ (here we are using x_1 and x_2 to denote the coordinates of x).

What makes kernels important is that many algorithms can be written so they only interact with their data via taking dot-products. For example, the Perceptron algorithm can be written this way (see below), as can SVMs, linear regression and ridge regression, and even k-means clustering. So, if we replace the dot-product $x \cdot z$ with $K(x, z)$, then these algorithms act implicitly as if data was in the higher-dimensional space mapped to by Φ (the "Φ-space"). Moreover, if data is linearly separable *by a large margin* in the Φ-space, then algorithms such as Perceptron will perform well even if the dimension of that space is very high. In particular, the mistake-bound of the Perceptron algorithm depends only on the margin and not on the dimension of the space. Similarly this is true for the number of samples needed for good generalization for algorithms like SVM. Notice also that computing K may be much easier than computing Φ. For example, the function $K(x, z) = (x \cdot z)^d$ corresponds to a mapping from an n dimensional space to a $\binom{d+n-1}{d}$-dimensional space, which can be very large. For $d = 6, n = 100$ this works out to roughly 1.6 billion. Yet computing K is very efficient.

If an algorithm can be written so that it only interacts with its data via taking dot-products, then we say it is *kernelizable*. Let's now see why the Perceptron algorithm is kernelizable.

Recall that as noted above, each point in time, the weight vector w_t used by the Perceptron algorithm is a weighted sum of incorrectly classified examples, with weight $+1$ if the example was positive and weight -1 if the example was negative. So, we can write w_t as

$$w_t = a_{i_1}x_{i_1} + \ldots + a_{i_k}x_{i_k},$$

for some previously-seen examples $x_{i_1}, \ldots, x_{i_k}$ and some weights $a_{i_1}, \ldots, a_{i_k}$. This means that to predict on a new example x, we can rewrite the quantity $w_t \cdot x$ as $a_{i_1}(x_{i_1} \cdot x) + \ldots + a_{i_k}(x_{i_k} \cdot x)$. Since the prediction only requires taking dot-products of examples, this means the algorithm is kernelizable. Specifically, given kernel K, we would predict positive on example x iff $a_{i_1}K(x_{i_1}, x) + \ldots + a_{i_k}K(x_{i_k}, x) \geq 0$. This produces the exact same behavior *as if* we had mapped data into the Φ-space and had run Perceptron there, without needing to actually compute the mapping Φ.

1.2.1 More Examples

The following are a few examples of important kernels.

- **Linear:** $K(x, z) = x \cdot z$.
- **Polynomial:** $K(x, z) = (x \cdot z)^d$ or $K(x, z) = (1 + x \cdot z)^d$.
- **Gaussian:** $K(x, z) = e^{-||x-z||^2/(2\sigma^2)}$.
- **Laplace:** $K(x, z) = e^{-||x-z||/(2\sigma^2)}$.

One can also give kernels for other kinds of data, such as kernels that measure similarity between sequences. Intuitively, one can think of a kernel as a measure of similarity between objects that satisfies certain mathematical conditions.

1.2.2 Mercer's Theorem

Theorem 2 (Mercer) *A pairwise function K is a kernel if and only if:*

1. *K is symmetric, and*
2. *For any set of training points $x_1, \ldots, x_m$ and for any $a_1, \ldots, a_m \in \Re$ we have:*

$$\sum_{ij} a_i a_j K(x_i, x_j) \geq 0$$

or equivalently, $a^T K a \geq 0$.

That is, $K = (K(x_i, x_j))_{i,j=1,\ldots,n}$ is a positive semidefinite matrix.

1.2.3 Useful Properties of Kernels

Kernel functions have a number of nice features. One is that they offer great modularity. There is no need to change the underlying learning algorithm (such as Perceptron or SVMs) to accomodate a particular choice of kernel function. Similarly, one can easily keep the same kernel function and substitute a different learning algorithm. Another nice feature of kernel functions is it is easy to create new kernel functions from existing ones, using their *closure properties*. Specifically, we have the following quite useful facts:

Fact 1 *If K_1 and K_2 are kernels, then for any $c_1, c_2 \geq 0$, the function*

$$K(x, z) = c_1 K_1(x, z) + c_2 K_2(x, z)$$

is also a kernel.

Proof: Let $\Phi_1 : X \to \Re^{N_1}$ be the implicit mapping corresponding to kernel K_1 and let $\Phi_2 : X \to \Re^{N_2}$ be the implicit mapping corresponding to kernel K_2. Then $\Phi : X \to \Re^{N_1+N_2}$ defined as $\Phi(x) = (\sqrt{c_1}\Phi_1(x), \sqrt{c_2}\Phi_2(x))$ is an implicit mapping corresponding to kernel K. Specifically, $\Phi(x) \cdot \Phi(z) = c_1\Phi_1(x) \cdot \Phi_1(z) + c_2\Phi_2(x) \cdot \Phi_2(z) = c_1 K_1(x, z) + c_2 K_2(x, z)$. ■

We can also multiply kernels to form new kernels.

Fact 2 *If K_1 and K_2 are kernels, then the function*

$$K(x, z) = K_1(x, z) K_2(x, z)$$

is also a kernel.

Proof: Let $\Phi_1 : X \to \Re^{N_1}$ be the implicit mapping corresponding to kernel K_1 and let $\Phi_2 : X \to \Re^{N_2}$ be the implicit mapping corresponding to kernel K_2. Then $\Phi : X \to \Re^{N_1 N_2}$ defined as $\Phi(x) = (\Phi_{1,i}(x)\Phi_{2,j}(x))_{i\in\{1,\ldots,n\},j\in\{1,\ldots,m\}}$ is an implicit mapping corresponding to kernel K. Specifically,

$$\begin{aligned}
\Phi(x)\cdot\Phi(z) &= \sum_{i,j} \Phi_{i,1}(x)\Phi_{2,j}(x)\Phi_{1,i}(z)\Phi_{2,j}(z) \\
&= \sum_{i} \Phi_{i,1}(x)\Phi_{1,i}(z)\left(\sum_{j} \Phi_{2,j}(x)\Phi_{2,j}(z)\right) \\
&= \sum_{i} \Phi_{i,1}(x)\Phi_{1,i}(z)K_2(x,z) \\
&= K_1(x,z)K_2(x,z).
\end{aligned}$$

■

1.3 Summary

If a learning algorithm can be written so that all of its computations on data points are in terms of dot-products of examples with each other, then the algorithm can be kernelized, replacing the dot-product with a kernel. Conceptually, this allows the algorithm to work as if the data were in a much higher dimensional space, and its performance will then depend only on the linear separability of points in that space. In particular, if data is linearly separable *by a large margin* in the Φ-space, then algorithms such as Perceptron will have a good mistake bound, and algorithms such as SVMs will need only a small sample size for good generalization. Computationally, we do not need to explicity map points into that space – we only need to modify the algorithm by replacing each $x \cdot z$ with a kernel computation $K(x, z)$.

Many machine learning algorithms are kernelizable, including Perceptron, SVMs, linear regression, and k-means clustering.

One big question is how to choose what kernel to use for your learning problem. Conceptually, a good kernel is often one that represents a good measure of similarity for the type of data you have, so kernels often encode domain knowledge (such as string kernels that measure some notion of the similarity of two sequences). In cases where the kernel has parameters, one can use Cross-Validation to choose the best parameter setting, e.g., choosing σ for the Gaussian kernel $K(x, z) = e^{-||x-z||^2/(2\sigma^2)}$. There is also work more generally on learning good kernels from data.