Kernel Methods and SVMs: A convex optimization point of view

Gael Lederrey Corentin Tallec

May 28, 2016

Abstract

Kernel methods, and more specifically support vectors machines, are well known, well theorized and efficient machine learning tools. For years, they were presenting state of the art performances in many machine learning fields, before being overtaken by deep learning methods. They have the enormous advantage of being well understood theoretically, and most interestingly to belong to the class of convex optimization problems, which is clearly not the case of the latter.

In this document, we aim at presenting the general framework of Kernel methods, to expose how they relate to convex optimization, and how their optimization can be undertaken in practice. Besides, we intend to give a more precise treatment of the support vectors machines special case.

1 Kernel methods

1.1 Generalities

The kernel methods are used in machine learning for pattern analysis. The support vector machine (SVM) is the best know kernel method. This class of algorithm is used to find and study some types of relations in datasets, *e.g.* classifications of the data, correlations between them, etc. For a usual algorithm solving this kind of task, the data needs to be transformed into feature vector representations which can be computationally costly. The kernel methods require only a **kernel**, *i.e.* a similarity function between pairs of data, specified by the user.
In the following sections, we will first give the definition of a kernel function, then we will present the feature mapping and finally we will present the idea of the kernel method

1.1.1 Kernel

First, we denote $\mathcal{X}$ an arbitrary input set and $\mathcal{Y}$ an output set. An input-output pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$ is called an example. A **kernel** is a similarity

function between these two sets. It's a real-valued function that quantifies the similarity between the input x and the output y. We define the function $K : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$, a **kernel** function if K is symmetric and if it is positive semidefinite: for any $x_1, \ldots, x_m \in \mathcal{X}$, the Gram matrix $G \in \mathbb{R}^{m \times m}$, defined by $G_{ij} = K(x_i, x_j) = K(x_j, x_i) = G_{ji}$, is positive semidefinite.

1.1.2 Feature Mapping

In order to make the computation simpler, the kernel needs to be written in the form of a "feature map". Mercer's theorem (or Aronszajn's theorem) gives the definition of a feature map:

Theorem 1 (Mercer's theorem) *The function K is a positive definite kernel on the $\mathcal{X}$ if and only if there exists a Hilbert space $\mathcal{H}$ with an inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ and a mapping*

$$\phi : \mathcal{X} \mapsto \mathcal{H}$$

such that, for any x, x' in $\mathcal{X}$:

$$K(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$$

The function ϕ is called the **feature mapping**.
In other words, the feature mapping is used to take the data in a space of small dimension $\mathcal{X}$ and map them into a Hilbert space of high dimension (even infinite). We do this because, it will be easier to perform a regression of the data in a space of higher dimension.

Je devrais peut etre definir ici les RKHS

1.1.3 Idea about the Kernel Methods

The **kernel methods** are using a set of pre-trained data. We can denote them by the pairs of example: $(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$. This means that for all the input x_i, $i = 1, \ldots, n$, we know the corresponding result y_i. Now, we can add an unlabelled value, let's say x'. The kernel function will be used to compare x' with all the pre-trained inputs x_i. The kernel will then say if x' is similar to x_i or not. With adding weights for each of the pre-trained examples, an output $\hat{y}$ depending on the unlabelled input x' can be computed.
We can use the example of a *kernelized binary classifier*. It computes a weighted sum of similarities:

$$\hat{y} = \operatorname{sgn} \sum_{i=1}^{n} w_i y_i K(x_i, x') \tag{1}$$

where

- $\hat{y} \in \{-1, +1\}$ is the predicted output for the unlabeled input x'
- $K : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ is the kernel function
- $\{(x_i, y_i)\}_{i=1}^{n}$ are the pre-trained examples with $y_i \in \{-1, +1\} \forall i$.
- $w_i \in \mathbb{R}$ are the weights for the pre-trained examples. They are determined by the larning algorithm.
- sgn is the sign function.

With this example, we can clearly see how a kernel function can be used to label a new input. The kernel method is the learning algorithm that will provide the weights w_i.

1.2 Kernel methods as convex optimization problems

From section 1.1.3, we already understand that the learning algorithm will have to find the best weights in order to have the most accurate prediction. The idea of the learning algorithm is to take a subset of the pre-trained examples, test the predictions, change the weight, etc. This can easily be seen as an optimization problem.

Je suis un peu perdu sur cette partie...

1.3 From infinite to finite

KERNEL TRICK + REPRESENTER THEOREM

We saw in the section 1.1.2, we can use a feature map to go from the input space $\mathcal{X}$ to a Hilbert space of a very high or infinite dimension $\mathcal{H}$. However, having the explicit representation of the mapping can be difficult and it is not convenient to work with it because of the high dimension of $\mathcal{H}$. Therefore, the idea is to work implicitly in the feature space $\mathcal{H}$. In order to do this, we must define the kernel trick.

Definition 1 (The Kernel trick) *We take any algorithm to process finite-dimensional vectors. If we can express this algorithms only in terms of pairwise inner products. Then this algorithm can be applied to potentially infinite-dimensional vectors in the feature space of a positive definite kernel if we replace each inner product evaluation by a kernel evaluation.*

Intuitively, the idea is that we will compute $K(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$ without even knowing the function $\phi(x)$ because it is often really complex.

2 Support vector machines: a special case

2.1 Generalities

2.2 Dual problem

2.3 A use case