Appunti della tesi di PhD di Da San Martino

A.A. 2023/2024

Rosso Carlo

Indice

1 Abstract

For dealing with structured data, kernel methods seems to have strong theoretical background. They do not require explicit vectorial representation, instead it is defined a kernel function that measures the similarity between the two objects. This approach has two problems:

- kernel for trees should not be sparse: we use ML to group data, if they are sparse, we aren't able to group them;
- it is slow: the kernel function is used both in the prediction and in the training phase, so it needs to be fast.

This paper presents three alternative methods:

1. **kernel composition**: having representations too spare to be useful, a kernel function is applied to the tree structure, so the data are projected onto a lower dimensional space with the property that similar structures are mapped similarly. Then it is applied a second kernel function to the projected data, which is the "logic" function;
2. **convolutional kernel**: the kernel function which measures the similarity between two trees uses convolution;
3. **DAG**: the trees are exploded into a directed acyclic graph, instead of a forest of trees, thus the rapresentation is more dense.

2 Introduction

2.1 Issues in structured data representation

One of the main problem of the machine learning is the reduction of the dimensionality of the data. In fact, the smaller and more compact the data, the faster the calculations. One approach is to extract the most important features. This approach is discarded, because there is a limit to the number of features that can be extracted to maintain an acceptable accuracy. In fact, the greater the number of features, the greater the probability that the data are sparse, which means that it is not possible to group them. On the other hand it is desirable to make use of techinques able to directly handle structured data.

2.2 Kernel methods

Here we introduce the kernel methods. By definition, kernel methods look for linear relations in the feature space.

Basecally, input items are compared via dot products of their representation in the feature space (the most similar items have the highest dot product). Actually, the dot product is replaced by a kernel function, which needs to be symmetric positive and semidefinite, so that it is applied directly in the original space (without the need to compute the feature space). So the computational complexity is not dependent on the size of the feature space but on the complexity of the kernel function (the generalization capability of a kernel method depends on the number of misclassified examples in the learning phase).

Briefly, the classification of a new example is computed via weighted sum of kernel evaluations between the example and a subset of the training instances (non ci sono basi, ma alcuni pattern di training sono usati come base). In this way the technique is able to directly handle structured data (if the kernel function is appropriate enough).
Kernel functions have two interesting properties:

- they are closed under **addition** and **linear combination**. This makes different inputs easy to combine;
- they scale on the number of different examples, not on the number of features, so they do not scale on the dimensionality of the data, but on the number of the data. This is very interesting if we consider words as single data.

Finally, kernel function are super cool, but it is hard to find fast to compute and expressive ones.

2.3 Thesis motivation

Completely expressive kernels for graphs are NP-hard to compute (so not feasible).
If the data are too sparse, the kernel function is going to give sparse results as well (it's not going to be accurate enough). So we think about a way to make data more dense (using other kernels).
Another interesting thing might be keeping the information about the position of the substructures in the trees for a better classification. Kinda making it more convolutional and considering more the context of each node.
Finally, since the kernel function works on raw data, the model needs to store those raw data, which can be a problem if the data are too big. Indeed, if the kernel function doesn't have enough raw data to compare the new example with, it looses accuracy; in fact, the accuracy of the classifier improves with the size of the training set.

3 Notation

Since I already studied most of the notation, I am going to write only the new stuff.

3.1 Statistical Learning Theory

The VC (Vapnik-Chervonenkis) dimension of a family of functions H is defined as the cardinality of the largest subset of points of the domain that can be labelled arbitrarily by choosing a function $h \in H$. For example for $H = ax, VC(H) = 2$ (I think).
There is a cool theorem which uses the VC dimension: let v be the VC dimension of the family of functions H. Then $\forall \delta > 0, h \in H$ dependent from a set of parameters Θ, the upper bound

$$R(h(\Theta)) \leq R_e(h(\Theta)) + \Omega\left(\frac{VC(h)}{n}\right) \tag{1}$$

where R_e is the empirical risk (space under the loss curve) and n is the size of the training set, holds with probability of at least $1 - \delta$ for $n > VC(h)$. Note that $\Omega\left(\frac{VC(h)}{n}\right)$ is a monotonic increasing function and it is called the confidence interval.

Given the reported theorem, we can see that if the VC dimension increases (and n remain constant), then the expected risk gets a bigger upper bound, which means that H might generalize poorly. On the other hand, the bigger n the lower the upper bound gets.
When a function is able to correctly classify the training set but has a large error on the rest of the distribution, then the function is told to overfit the data.

3.2 Self Organizing Maps

The aim of the Self Organizing Maps (SOM) learning arlgorithm is to learn a feature map

$$\mathcal{M} : \mathcal{I} \to \mathcal{A} \tag{2}$$

This is obtained by associating each point in $\mathcal{A}$ to a different neuron. High dimentional input vectors are projected into the two (actually $n \in \mathbb{N}$) dimensional coordinates of the lattice, with the aim of preserving, as much as possible, the topological relationships among the input vectors (which means that close input vectors are associated to neurons which are close on the lattice).

An example

When the input space is a structured domain with labels in $\mathcal{U}$, we redefine Equation 2 to be:

$$\mathcal{M}^{\#} : \mathcal{U}^{\#[i,o]} \to \mathcal{A} \tag{3}$$

Where we define $\mathcal{M}^{\#}$ recursively as:

$$\mathcal{M}^{\#}(G) = \begin{cases} nil_{\mathcal{A}} & \text{if } G = \xi \\ \mathcal{M}_{node}(u_s, \mathcal{M}^{\#}(G^{(1)}), \ldots, \mathcal{M}^{\#}(G^{(o)})) & otherwise \end{cases} \tag{4}$$

Where $s = source(G), G^{(1)}, \ldots, G^{(o)}$ are the subgraphs pointed by the outgoing edges leaving from s, $nil_{\mathcal{A}}$ represents the 0 and u_s is the label of the s node; finally, $\mathcal{M}_{node}$ is a SOM, defined on a generic node, which takes in input the label of the node and the "encoding" of the subgraphs $(G^{(1)}, \ldots, G^{(o)})$ according to the $M^{\#}$ map. So

$$\mathcal{M}_{node} : \mathcal{U} \times \mathcal{A} \times \cdots \times \mathcal{A} \to \mathcal{A} \tag{5}$$

Training lagorithm

The weights associated with each neuron in the q dimensional lattice $\mathcal{M}_node$ can be trained as follow:

Step Competitive step. The winnign neuron, at iteration t, with the closest weight vector is selected:

$$y_{i^*}(t) = \arg\min_{c_i} ||\Lambda(x_v(t) - m_{c_i}(t))|| \tag{6}$$

where Λ is used to balance the importance of labels;

Step Comparative step. The weight vector $m_{y_{i^*}}$, as well as the weight vector of neurons in the topological neighborhood of the winning neuron, are moved closer to the input vector:

$$m_{c_r}(t+1) = m_{c_r}(t) + \eta(t) f(\Delta_{i^* r})(x_v(t) - m_{c_r}(t)) \tag{7}$$

where $\Delta_{i^* r}$ is the topological distance between c_r and c_{i^*} in the lattice. Usually $f(\cdot)$ takes the form of a Gaussian function to update very near neurons and almost ignore far away ones. For example:

$$f(x) = \exp\left(-\frac{x^2}{2\sigma(t)^2}\right) \tag{8}$$

Usually the neighborhood radius $\sigma(t)$ decreases to zero along with the training.

The described model (SOM-SD) allows the processing of undirected graphs, and non-positional graphs where the order of edges is not relevant. The heuristic nature of the model can not formally guarantee to preserve the topology of the items in the input space.

3.3 Kernel Methods

The class of kernel methods comprises all those algorithms that do not require an explicit representation of the examples but only information aobu the similarities among them. Any kernel method can be decomposed into two modules:

- a problem specific kernel function (to get the differences between the different inputs);
- a general purpose learning algorithm.

The modularity of the approach allows to study representation and optimization independently. Wahba's representer theorem states that the solution of certain optimization problems involving an empirical risk term and a quadratic regularizer can be written in terms of an expansion of the training examples. Thus, given a dataset $S = \{(x_i, y_i) : i = 1, \ldots, n\}$ and a kernel function K, the solution w of the problem can be expressed as:

$$w = \sum_i^n \alpha_i y_i \phi(x_i) \tag{9}$$

Now let's introduce the score function:

$$S = x) = \sum_i^n \alpha_i y_i \phi(x_i)\phi(x) = \sum_i^n \alpha_i y_i K(x_i, x) \tag{10}$$

Note tbat the score function can be expressed as a weighted linear combination of kernel function evaluations between examples in the dataset and x. Here follows the classification function:

$$c(x) = sign(S(x)) = sign\left(\sum_i^n \alpha_i y_i K(x_i, x)\right) \tag{11}$$