CONVOLUTION KERNELS FOR NATURAL LANGUAGE.

Michael Collins Nigel Duffy

In this paper it is shown how kernel methods are used in Natural Language Processing (NLP) problems.NLP make use of complex data like trees ,strings,graphs.These need to be converted into feature vectors.

It is difficult to process the complex data structures used in NLP.

Kernel Method avoids the need to convert data into vector form. The only information they require is a measurement of the similarity of each pair of items in a data set. This measurement is called a *kernel*, and the function for determining it is called the *kernel function*.

The advantage of using kernel functions is that a huge feature space can be analyzed with a computational complexity not dependent on the size of the feature space, but on the complexity of the kernel function, which means that kernel methods do not suffer the curse of dimensionality.
This paper tells how kernels can be used in NLP tasks like Parsing and tells that there is some dependency between structural information in different trees unlike the PCFGs and HMMs which made independence assumptions, disregarding substantial amounts of structural information.
This method considers all the fragments in a particular parse tree and finds better dependencies between grammar rules.

Here the kernel used is the Tree kernel which was proposed by Houssler.

Tree Kernel

Tree Kernel calculates the number of common subtrees in two trees by taking dot product between the feature vectors of subtrees in high dimensional space.

For two parse trees T and T' the tree kernel is given as

$$K_t(T,T') = \sum_{i=1}^{|\mathcal{T}|} \left(\sum_{n \in N_T} I_i(n) \cdot \sum_{n' \in N_{T'}} I_i(n') \right)$$
$$= \sum_{n \in N_T} \sum_{n' \in N_{T'}} \Lambda(n,n')$$

where Nt and Nt' refer to the node sets of the parse trees T and T' . Let T refer to the substructure space. Let Ii(n) refer to an indicator function which equals to 1 iff the corresponding subtree is rooted at the node n and 0 otherwise.

Let $\Lambda(n, n') = \sum_{i=1}^{|T|} (Ii(n).Ii(n'))$

evaluates the number of matched subtrees rooted at n and n' .

Λ(n, n') can be evaluated via dynamic programming as follows:

(1) If the production rule rooted at n and n' are different, Λ(n, n')=0;

(2) else if both n and n' are are same and they are pre terminals, Λ(n, n') = λ;

(3) else Λ(n, n') = λ nc(n) j=1 (1 + Λ(c(n, j), c(n' , j))). (recursive definition)

Here λ is the scaling factor.this downweights the contribution of tree fragments exponentially with their size.

where nc(n) is the number of children of node n. Let c(n, j) be the j-th child of node n. Let $\lambda \in [0, 1]$ be the decay factor for the depth of the subtree. Tree kernel can be evaluated in $O(|NT| \cdot |NT'|)$

The key idea here is that one may take a structured object and split it up into parts. If one can construct kernels over the parts then one can combine these into a kernel over the whole object. This idea can be extended recursively so that one only needs to construct kernels over the "atomic" parts of a structured object. The recursive combination of the kernels over parts of an object retains information regarding the structure of that object.

Experiment

The tree kernel was applied to the problem of parsing the penn treebank ATIS corpus.

It was found that voted perceptron algorithm with tree kernel performed much better than the PCFG ...results are given in below table

Scale	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Score	77 ± 1	78 ± 1	79 ± 1	79 ± 1	79 ± 1	79 ± 1	79 ± 1	79 ± 1	78 ± 1
Imp.	11 ± 6	17 ± 5	20 ± 4	21 ± 3	21 ± 4	22 ± 4	21 ± 4	19 ± 4	17 ± 5

Table 2: *Score* shows how the parse score varies with the scaling factor for deeper sub-trees is varied. *Imp.* is the relative reduction in error in comparison to the PCFG, which scored 74%. The numbers reported are the mean and standard deviation over the 10 development sets.

Tree kernels have been shown to be interesting approaches for the modeling of syntactic information in natural language tasks, e.g. syntactic parsing, relation extraction (Zelenko et al., 2003), Named Entity recognition (Cumby

and Roth, 2003; Culotta and Sorensen, 2004) and Semantic Parsing (Moschitti, 2004). The main tree kernel advantage is the possibility to generate a high number of syntactic features and let the learning algorithm select those most relevant for a specific application.

Unfortunately, Tree Kernels show

(a) an inherent super linear complexity and

(b) a lower accuracy than traditional attribute/value methods

In the tree kernel used by Duffy and Collins the substructure used was sub-trees as shown below.

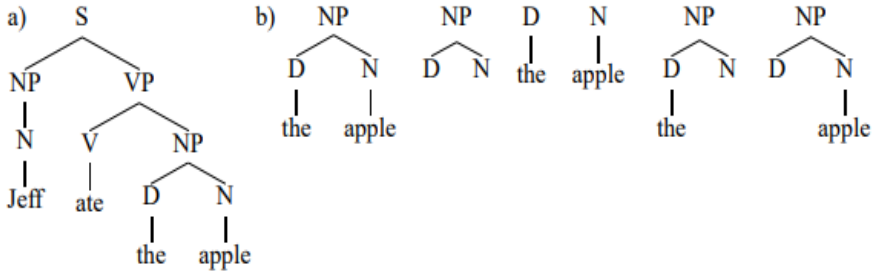

Figure 2: a) An example tree. b) The sub-trees of the NP covering *the apple*. The tree in (a) contains all of these sub-trees, and many others. We define a sub-tree to be any sub-graph which includes more than one node, with the restriction that entire (not partial) rule productions must be included. For example, the fragment `[NP [D the ]]` is excluded because it contains only part of the production `NP → D N`.

A better kernel can be made using fragments of trees as ST and SST's. They can be explained as-

ST and SST

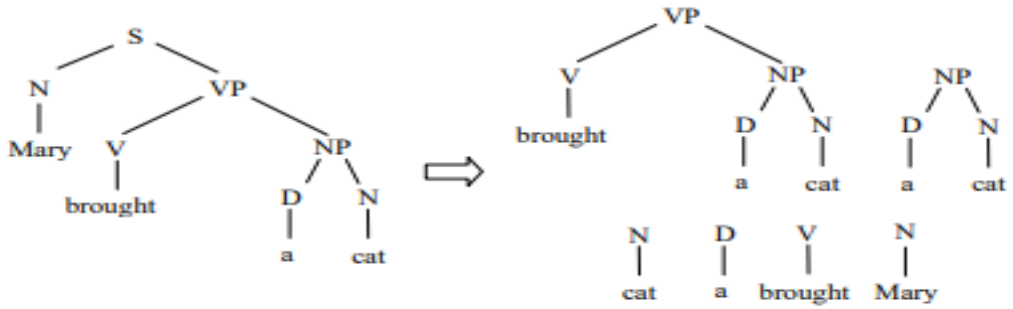

Figure 2: A syntactic parse tree with its subtrees (STs).

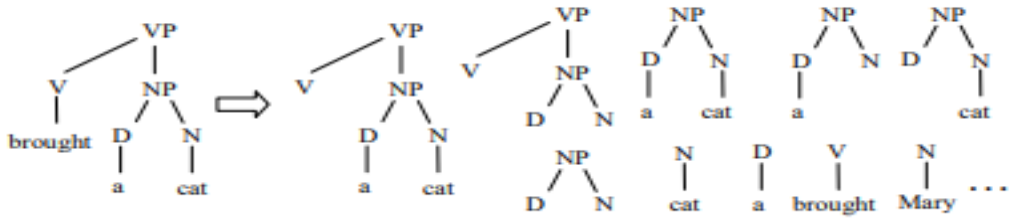

Figure 3: A tree with some of its subset trees (SSTs).

An algorithm for the evaluation of the ST and SST kernels which runs in linear average time can be made.

$$K(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2)$$

3. if the productions at n1 and n2 are the same, and n1 and n2 are not pre-terminals then

$$\Delta(n1, n2) = \text{ncY}(n1)\ j{=}1\ (\sigma + \Delta(c\ j\ n1\ ,\ cj\ n2\ ))\ (2)$$

where $\sigma \in \{0, 1\}$, nc(n1) is the number of the children of n1 and c j n is the j-th child of the node n.

Since the productions are the same, nc(n1) = nc(n2). When $\sigma = 0$, Δ(n1, n2) is equal 1 only if $\forall$ j Δ(c j n1 , cj n2) = 1, i.e. all the productions associated with the children are identical.

(This will count the total subtrees)

By recursively applying this property, it follows that the subtrees in n1 and n2 are identical. When $\sigma = 1$, Δ(n1, n2) evaluates the number of SSTs common to n1 and n2.(The one in Collin and Duffy's kernel)

Additionally, we study some variations of the above kernels which include the leaves in the fragment space. For this purpose, it is enough to add the condition

0. if n1 and n2 are leaves and their associated symbols are equal then $\Delta(n1, n2) = 1$,

Fast Tree Kernel

To compute the kernels defined in the previous section, we sum the Δ function for each pair $hn1, n2i \in NT1 \times NT2$. When the productions associated with n1 and n2 are different, we can avoid evaluating $\Delta(n1, n2)$ since it is 0.

Thus, we look for a node pair set $Np = \{hn1, n2i \in NT1 \times NT2 : p(n1) = p(n2)\}$, where p(n) returns the production rule associated with n. To efficiently build Np, we

(i) extract the L1 and L2 lists of the production rules from T1 and T2,

(ii) sort them in the alphanumeric order and

(iii) scan them to find the node pairs hn1, n2i such that $(p(n1) = p(n2)) \in L1 \cap L2$.

Step (iii) may require only O(|NT1 | + |NT2 |) time, but, if p(n1) appears r1 times in T1 and p(n2) is repeated r2 times in T2, we need to consider r1 × r2 pairs.

(a) The list sorting can be done only once at the data preparation time (i.e. before training) in O(|NT1 | × log(|NT1 |)).

(b) The algorithm shows that the worst case occurs when the parse trees are both generated using only one production rule, i.e. the two internal while cycles carry out |NT1 |×|NT2 | iterations. In contrast, two identical parse trees may generate a linear number of non-null pairs if there are few groups of nodes associated with the same production rule.

(c) Such an approach is perfectly compatible with the dynamic programming algorithm which computes Δ. In fact, the only difference with the original approach is that the matrix entries corresponding to pairs of different production rules are not considered. Since such entries contain null values they do not affect the application of the original dynamic programming. Moreover, the order of the pair evaluation can be

established at run time, starting from the root nodes towards the children.

The fast Tree kernel improves the speed of computation by a huge amount.

Ayush Badola

B20MT012