Decoding Distributed Tree Structures

Ferrone Lorenzo[1], Fabio Massimo Zanzotto[1], and Xavier Carreras[2]

[1] Università degli studi di Roma "Tor Vergata"
[2] Xerox Research Centre Europe

Abstract. Encoding structural information in low-dimensional vectors is a recent trend in natural language processing that builds on distributed representations [13]. However, although the success in replacing structural information in final tasks, it is still unclear whether these distributed representations contain enough information on original structures. In this paper we want to take a specific example of a distributed representation, the distributed trees (DT) [16], and analyze the reverse problem: can the original structure be reconstructed given only its distributed representation? Our experiments show that this is indeed the case, DT can encode a great deal of information of the original tree, and this information is often enough to reconstruct the original object format.

1 Introduction

Typical Natural Language Processing methods are designed to produce a discrete, symbolic representation of the linguistic analysis. While this type of representation is natural to interpret, specially by humans, it has two main limitations. First, as a discrete structure, it is not immediately clear how to compute similarities between different structures. Second, it is not immediate to exploit such discrete linguistic representations in downstream applications. Thus, when dealing with discrete structures, one needs to define similarity measures and features that are appropriate for the application at hand.

Recently there has been a great deal of research on the so-called distributional representations [13]. These approaches represent linguistic annotations as vectors in d-dimensional spaces. The aim of these methods is that similarity across different annotations will be directly captured by the vector space: similar linguistic constructs should map to vectors that are close to each other. In other words, the Euclidean distance is meant to provide a similary measure. Because such d-dimensional vectors aim to capture the essence of the linguistic structure, one can use them directly as inputs in downstream classifiers, thus avoiding the need of feature engineering to map a discrete structure to a vector of features. Word embeddings [11] are a prominent example of this trend, with many succesful applications. Noticeably, in the recent years there has been a large body of work about modelingq semantic composition using distributional representations [14, 2, 12, 3, 4, 8, 18].

Stemming from distributed representations [13], real valued vectors have been also used to represent complex structural information such as syntactic trees.

Distributed Trees [16] are low-dimensional vectors that encode the feature space of syntactic tree fragments underlying tree kernels [5]. The main result behind this method is that the dot product in this low-dimensional vector space approximates tree kernels. In other words, the tree kernel between two syntactic trees (which computes tree similarity by looking at all subtrees) can be approximated by first mapping each tree into a low-dimensional distributed tree, and then computing the inner product. Thus, distributed trees can be seen as a compression of the space of all tree fragments down to a d-dimensional space, preserving tree similarity. Since the vector space representing distributed trees captures tree similarity (in the tree kernel sense), it must be that such vectors are good representations of syntactic structures. This hypothesis has been confirmed empirically in downstream applications for question classification and recozniging textual entailment [16].

However, a legitimate question is: what is exactly encoded in these vectors representing structural information? And, more importantly, is it possible to decode these vectors to recover the original tree? Auto-encoders from the neural network literature, such as those used to induce word embeddings, are designed to be able to encode words into vectors and decode them. Syntactic trees, however, are structures that are far more complex.

In this paper, we propose a technique to decode syntactic trees from a vector representation based on distributed trees. We pose the problem as a parsing problem: given a sentence, and a reference vector representing its syntactic tree, what is the syntactic tree that is most similar to the reference vector? Our solution is based on CKY, and the fact that we can compute the similarity between a partial tree and the reference vector representing the target tree.

In experiments we present a relation between the number of dimensions of the distributed tree representation and the ability to recover parse trees from vectors. We observe that, with sufficient dimensionality, our technique does in fact recover the correct trees with high accuracy. This confirms the idea that distributed trees are in fact a compression of the tree. Our methodology allows to direclty evaluate the compression rate at which trees can be encoded.

The rest of the paper is organized as follows: first we will give a brief background on Distributed Trees and then we will explore the process of going back from this representation to a symbolic one, we will introduce the famous CYK algorithm and then present our variant of it that deals with distributed trees. Finally, we will present an experimental investigation of this new algorithm, and we will then conclude with a few remarks and plans on possible direction of future works.

2 Background: Encoding Structures with Distributed Trees

Encoding Structures with Distributed Trees [16] (DT) is a technique to embed the structural information of a syntactic tree into a dense, low-dimensional vector of real numbers. DT were introduced in order to allow one to exploit the

modelling capacity of tree kernels [5] but without their computational complexity. More specifically for each tree kernel TK [1, 6, 15, 9] there is a corresponding distributed tree function [16] which maps from trees to vectors:

$$\begin{aligned} \mathrm{DT}\colon T &\to \mathbb{R}^d \\ t &\mapsto \mathrm{DT}(t) = \mathbf{t} \end{aligned}$$

such that:

$$\langle \mathrm{DT}(t_1), \mathrm{DT}(t_2) \rangle \approx TK(t_1, t_2) \tag{1}$$

where $t \in T$ is a tree, $\langle \cdot, \cdot \rangle$ indicates the standard inner product in $\mathbb{R}^d$ and $TK(\cdot, \cdot)$ represents the original tree kernel. It has been shown that the quality of the approximation depends on the dimension d of the embedding space $\mathbb{R}^d$.

To approximate tree kernels, distributed trees use the following property and intuition. It is possible to represent subtrees $\tau \in S(t)$ of a given tree t in distributed tree fragments $\mathrm{DTF}(\tau) \in \mathbb{R}^d$ such that:

$$\langle \mathrm{DTF}(\tau_1), \mathrm{DTF}(\tau_2) \rangle \approx \delta(\tau_1, \tau_2) \tag{2}$$

Where δ is the Kronecker's delta function. Hence, distributed trees are sums of distributed tree fragments of trees, that is:

$$\mathrm{DT}(t) = \sum_{\tau \in S(t)} \sqrt{\lambda}^{|\mathcal{N}(\tau)|} \mathrm{DTF}(\tau)$$

where λ is the classical decaying factor in tree kernels and $\mathcal{N}(\tau)$ is the set of the nodes of the subtree τ. With this definition, the property in Eq. 1 holds.

Distributed tree fragments are defined as follows. To each node label n we associate a random vector $\mathbf{n}$ drawn randomly from the d-dimensional hypersphere. Random vectors of high dimensionality have the property of being quasi-orthonormal (that is, they obey a relationship similar to 2). The following functions are then defined:

$$\mathrm{DTF}(\tau) = \bigodot_{n \in \mathcal{N}(\tau)} \mathbf{n}$$

where $\odot$ indicates the shuffled circular convolution operation [3], which has the property of preserving quasi-orthonormality between vectors.

To compute distributed trees, there is a last function $\mathrm{SN}(n)$ for each node n in a tree t that collects all the distributed tree fragments of t where n is the head. This is recursively defined as follows:

$$\mathrm{SN}(n) = \begin{cases} \mathbf{0} \text{ if } n \text{ is terminal} \\ \mathbf{n} \odot \bigodot_i \sqrt{\lambda} \left[\mathbf{n_i} + SN(n_i) \right] \text{otherwise} \end{cases}$$

[3] The circular convolution between $\mathbf{a}$ and $\mathbf{b}$ is defined as the vector $\mathbf{c}$ with component $c_i = \sum_j a_j b_{i-j \bmod d}$. The shuffled circular convolution is the circular convolution after the vectors have been randomly shuffled.

where n_i are the direct children of n in the tree t. Hence, distributed trees can also be computed with the more efficient equation:

$$\mathrm{DT}(t) = \sum_n \mathrm{SN}(n)$$

We now have all the equations to develop our idea of reconstructing trees from distributed trees. The fact that the approximation can get arbitrarily good with the increasing of the dimension of the vector space suggests that the vector $\mathrm{DT}(t)$ encodes in fact all the information about the tree t,. However, it is not immediately obvious how one could go about using this encoded information to recreate the symbolic version of the parse tree.

3 Going back: reconstructing symbolic trees from vectors

Our aim here is to investigate if the information stored in the distributed tree correctly represent trees. The hypothesis is that this is a correct representation if we can reconstruct original trees from distributed trees.

We treated the problem of reconstructing trees as a parsing problem. Hence, given the distributed tree $\mathrm{DT}(t)$ representing a syntactic tree of a given sentence s, we want to show that starting from s and $\mathrm{DT}(t)$ is possible to reconstruct t More formally, given a sentence s and its parse tree t, we use the following information in order to try to reconstruct t:

- the sentence s itself;
- the distributed vector of the tree: $\mathbf{t} = \mathrm{DT}(t)$;
- a big context-free grammar G for English that generates also t.

To solve the problem we implemented a variant of the CYK algorithm that uses the information contained in the vector to guide the reconstruction of the original parser among all the possible parses of a given sentence. In the next section we will first give a brief recap on the CYK algorithm, and later we will present our variant.

3.1 Standard CYK algorithm

The CYK algorithm is one of the most efficient algorithms for parsing context-free grammars. It takes as input a sentence $s = a_1, a_2, \ldots, a_n$ and a context-free grammar G in Chomsky-Normal Form, that is, each rule is of one of these two forms:

$$A \to B\ C$$
$$A \to w$$

where A, B, C are non-terminal symbol and w is a terminal symbol (a word). The grammar itself is composed of a list of symbols $R_1\, R_2, \ldots, R_r$ among which there is a subset R_S of starting symbols.

In its most common form the CYK algorithm works by constructing a 3-dimensional table P, where the cell $P[i, j, k]$ contains a list of all the ways that the span from word i to word j could be obtained from the grammar symbol R_k. Moreover, each rule is also linked to its children nodes: in this way it is possible to navigate the table P and reconstruct a list of possible parse trees for the original sentence.

$$P_{ijk} = [R_k \to A_1\ B_1, R_k \to A_2\ B_2, \ldots]$$

Collectively, in the cell $P[i, j]$ are stored all the possible derivations that could span the sentence from word i to word j.

The CYK algorithm has subsequently been extended in various forms in order to deal with probabilistic grammars and be able thus to retrieve the k-best parse trees of a given sentence. Our proposal is similar in this intent, but instead of a k-best list of parse trees, it uses a beam search at each cell, guided by the distributed vector of the sentence that we want to reconstruct. In the following section we expose our algorithm.

3.2 CYK over Distributed Trees

In our variant of the algorithm we use the information encoded in the distributed vector ($\mathbf{t}$) of the parse tree of the sentence to guide the new parsing stage in the following manner: in each cell $P[i, j]$ we store not just a list of ways to obtain the rule active at the moment, instead, we store a list L of the m-best entire partial trees built up to that point, ordered by their similarity with $\mathbf{t}$.

More in detail, when we are in the cell $P[i, j]$ for all rules r in the grammar of the form $A \to B\ C$, we check for all trees that we already built in $P[k, j]$ and $P[i-k, j+k]$ that starts with $B \to \bullet\ \bullet$ and $C \to \bullet\ \bullet$, respectively. Let's call L_B and L_C these two lists, each composed of at most m trees. We create then the list L_{BC} composed of at most m^2 trees that have A as root, a subtree from L_B as left child and one coming from L_C as right child. For each one of these trees t_ℓ we compute:

$$\text{score} = \frac{\langle \text{SN}(t_\ell), \mathbf{t} \rangle}{\langle \text{DTF}(r), \mathbf{t} \rangle}$$

We then sort the list L_{BC} in decreasing order according to score and store only the first m trees.

The pseudocode for our algorithm (also including the more advanced extension presented in the next section) is presented in (alg. 1).

As we do not store each possible parse tree it may happen that this algorithm will fail to recognize a sentence as belonging to the grammar, when this happens we can either flag the result as a fail to reconstruct the original parse tree, or try again increasing the value of m. In our experiment we decided to flag the result as an error, leaving the possibilities of increasing m to further experimentation.

Algorithm 1 CYK, DTK variant

Input: sentence $s = w_1, w_2, \ldots, w_n$, grammar G, distributed vector $\mathbf{t}$

Initialization: fill an $n \times n$ table P with zeroes
for $i = 1$ to n **do**
    for all symbol $A \in G$ **do**:
        rule $\leftarrow (A \rightarrow w_i)$
        ruleScore $\leftarrow \frac{\langle \mathrm{SN(rule)}, \mathbf{t} \rangle}{\|\mathrm{SN(rule)}\|}$
        append (rule, ruleScore) to $P[i, 0]$
    end for
    sort $P[i, 0]$ decreasingly by **score**
    take the first m element of $P[i, 0]$
end for
for $i = 2$ to n **do**
    for $j = 1$ to $n - i + 1$ **do**
        for $k = 1$ to $i - 1$ **do**
            for all pairs $(B \rightarrow \bullet\ \bullet, C \rightarrow \bullet\ \bullet) \in P[k, j] \times P[i - k, j + k]$ **do**
                for all rule $(A \rightarrow B\ C) \in G$ **do**
                    ruleScore $\leftarrow \frac{\langle \mathrm{DTF(rule)}, \mathbf{t} \rangle}{\|\mathrm{DTF(rule)}\|}$
                    if ruleScore $\geq$ threshold **then**
                        tree $\leftarrow A \rightarrow [B \rightarrow \bullet\ \bullet, C \rightarrow \bullet\ \bullet]$
                        score $\leftarrow \langle \mathrm{DT(tree)}, \mathbf{t} \rangle$
                        append (tree, score) to $P[i, j]$
                    end if
                end for
            end for
        end for
        sort $P[i, j]$ decreasingly by **score**
        take the first m element of $P[i, j]$
    end for
end for
if $P[n, 1]$ is **not** empty **then**
    return $P[1, n]$
else
    return Not parsed
end if

3.3 Additional rules

In order to increase both efficiency and accuracy, we introduced a few more rules in our algorithm:

- a filter: in this way we don't cycle through all the rules in the grammar but only on those with a score more then a given threshold. Such score is computed again as a similarity between the rule r (viewed as a tree) and $\mathbf{t}$:

$$\text{rulescore} = \frac{\langle \text{DTF}(r), \mathbf{t} \rangle}{\|\text{DTF}(r)\|}$$

  the threshold is defined as:

$$\frac{\lambda^{\frac{3}{2}}}{p}$$

  The significance of this threshold is the following: first recall that DTK is an approximation of TK, which in turn is a weighted count of common subtrees between two trees. The numerator $\frac{\lambda^{\frac{3}{2}}}{p}$ is then the exact score (not an approximation) that a rule would get if it appeared exactly once in the whole tree, while the parameter $p > 1$ relaxes the requirement by lowering this threshold, in order to take in account the fact that we are dealing with approximated scores. In other words, the rules that pass this filter are those that should appear at least one time in t, with some room to account for approximation error.
- a reintroduction of new rules to avoid the algorithm getting stuck on the wrong choice: that is, after the sorting and trimming of the list of the first m trees in each cell, we also add another m trees (again, sorted by score) but for which the root node is different than the node of the first tree in the cell
- a final reranking, according to the dtk between each element in our list of final candidate, and the original tree.

4 Experiment

The pipeline of the experiment is the following:

1. Create a (non probabilistic) context-free grammar G from the input file. This is just a collection of all the production rules (either terminal or not) that appear in the input file
2. For each tree t in the testing set:
   (a) compute the distributed vector $\text{DT}(t)$;
   (b) parse the sentence using the CYK algorithm as explained in section (4) and (5);
   (c) check if the resulting parse tree is equal to the original one;
3. Compute average labeled precision, recall, and f-score for the entire dataset.

4.1 Setup

As testbed for our experiment we used the Wall Street Journal section of the PennTree Bank. Sections 00 to 22 have been used to generate the grammar. Note that the generated grammar is not a probabilistic one, nor there is any learning involved. Instead, it is just a collection of all the rules that appear in parse trees in those sections, more precisely the resulting grammar contains 95 (non-terminal) symbols and 1706 (non-terminal) production rules. The test has been performed on section 23, consisting of a total of 2389 sentences. As a preprocessing step we also first binarise each tree in the PTB so that the resulting grammar is in Chomsky Normal form.

There are four parameters in our model that can be changed: a first set of parameters pertains to the DTK encoding, while another set of parameters pertains to the CYK algorithm. The parameter relative to the DTK are the dimension d of the distributed trees, for which we tried the values of $1024, 2048, 4096, 8192$ and 16384, and the parameter λ for which we tried the values $0.2, 0.4, 0.6, 0.8$.

For the second set of parameter instead we have m, which is the size of the list of best trees that are kept at each stage and p which is the threshold of the filter. For the parameter m we tried the values up to 10 but found out that increasing this value doesn't increase the performance of the algorithm, while at the same time increasing significantly the computational complexity. For this reason we fixed the value of this parameter to 2, and only report results relative to this value. Finally, for the filter threshold we tried values of $p = 1.5, 2$ and 2.5. In the following section we report the results for the given parameters on the test set.

4.2 Results

In this section we report our results on the dataset. In table (1) we report the percentage of *exactly* reconstructed sentences. The parameter λ is kept fixed at 0.6, while the dimension d and filter p vary. The same results are also presented in figure (1).

In table (2) we report the average precision and recall on the entire dataset for different values of p, fixed $\lambda = 0.6$, and varying the dimension d. In figure (2) we graph the f-measure relative to those same parameter.

As we can see the number of correctly reconstructed sentences grows significantly with the increasing of the dimension (as expected) topping at 92.79% for $d = 16384, p = 2.5$ and $\lambda = 0.6$. On the other hand lower values of d while yielding a low percentage of reconstructed sentences, still can provide a high precision with value as low as 1024 resulting in a precision of 0.89.

In conclusion it seems that the main parameter to influence the algorithm is the dimension d, which was what we expected, because the quality of the approximation depends on d, and thus the amount of information that can be stored in $\mathrm{DT}(t)$ without too much distortion. The parameter p on the other hand does not seem to influence the final results nearly as much. As we can see in the

p	1.5	2	2.5
$d = 1024$	22.26%	23.5%	23.25%
$d = 2048$	48.8%	60.46%	52.32%
$d = 4096$	77.9%	81.39%	75.58%
$d = 8192$	91.86%	88.28%	87.5%
$d = 16384$	92.59%	92.54%	**92.79**%

Table 1: Percentage of correctly reconstructed sentences. $\lambda = 0.6$

p	1.5	2	2.5
$d = 1024$	0.89	0.78	0.71
$d = 2048$	0.964	0.912	0.85
$d = 4096$	0.984	0.967	0.951
$d = 8192$	0.994	0.994	0.99
$d = 16384$	**0.995**	**0.995**	0.994

(a) precision

p	1.5	2	2.5
$d = 1024$	0.285	0.43	0.477
$d = 2048$	0.58	0.754	0.78
$d = 4096$	0.846	0.923	0.929
$d = 8192$	0.959	0.959	0.967
$d = 16384$	0.965	0.965	**0.976**

(b) recall

Table 2: Average precision and recall. $\lambda = 0.6$

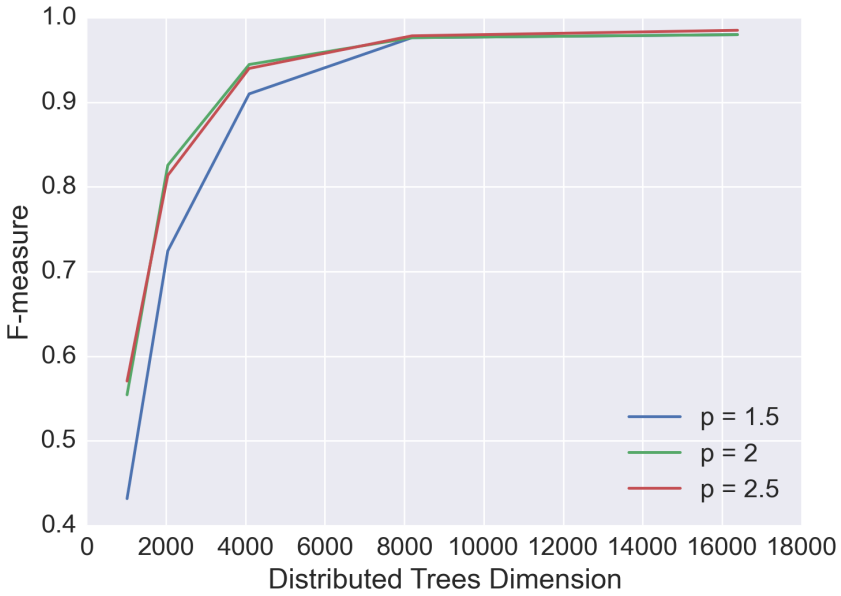

Fig. 1: Average F-measure on the entire dataset as the dimension increase. $\lambda = 0.6$

tables the results, especially for high dimension d are nearly the same for all the values of p that we tried.

5 Conclusion and future work

We showed under our setting that it is possible to reconstruct the original parse tree from the information included in its distributed representation. Together with the work on distributed representation parsing [17] we envision that it would be possible to create a symbolic parse tree of a sentence from *any* distributed representation, not necessarily derived directly from the correct parse tree, but which may be learned in some other way, for example as output of a neural network approach.

As for future work we plan to expand the experimental investigation on a more ample dataset, moreover we also want to use a more state-of-the art implementation of the CYK algorithm both to increase the speed of the algorithm and in order to lift the limitation that all the trees (including those in the grammar) should be in Chomsky Normal Form. Finally, we want to explore our approach in a task-based setting as [10] and [7].

References

1. Aiolli, F., Da San Martino, G., Sperduti, A.: Route kernels for trees. In: Proceedings of the 26th Annual International Conference on Machine Learning. pp. 17–24. ICML '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1553374.1553377
2. Baroni, M., Lenci, A.: Distributional memory: A general framework for corpus-based semantics. Comput. Linguist. 36(4), 673–721 (Dec 2010), http://dx.doi.org/10.1162/coli_a_00016
3. Baroni, M., Zamparelli, R.: Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space. In: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing. pp. 1183–1193. Association for Computational Linguistics, Cambridge, MA (October 2010), http://www.aclweb.org/anthology/D10-1115
4. Clark, S., Coecke, B., Sadrzadeh, M.: A compositional distributional model of meaning. Proceedings of the Second Symposium on Quantum Interaction (QI-2008) pp. 133–140 (2008)
5. Collins, M., Duffy, N.: Convolution kernels for natural language. In: NIPS. pp. 625–632 (2001)
6. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: Proceedings of ACL02 (2002)
7. Dagan, I., Glickman, O., Magnini, B.: The PASCAL RTE challenge. In: PASCAL Challenges Workshop. Southampton, U.K (2005)
8. Grefenstette, E., Sadrzadeh, M.: Experimental support for a categorical compositional distributional model of meaning. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing. pp. 1394–1404. EMNLP '11, Association for Computational Linguistics, Stroudsburg, PA, USA (2011), http://dl.acm.org/citation.cfm?id=2145432.2145580
9. Kimura, D., Kuboyama, T., Shibuya, T., Kashima, H.: A subpath kernel for rooted unordered trees. In: Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining - Volume Part I. pp. 62–74. PAKDD'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2017863.2017871

10. Li, X., Roth, D.: Learning question classifiers. In: Proceedings of the 19th international conference on Computational linguistics - Volume 1. pp. 1–7. COLING '02, Association for Computational Linguistics, Stroudsburg, PA, USA (2002), http://dx.doi.org/10.3115/1072228.1072378
11. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. CoRR abs/1301.3781 (2013), http://arxiv.org/abs/1301.3781
12. Mitchell, J., Lapata, M.: Vector-based models of semantic composition. In: Proceedings of ACL-08: HLT. pp. 236–244. Association for Computational Linguistics, Columbus, Ohio (June 2008), http://www.aclweb.org/anthology/P/P08/P08-1028
13. Plate, T.A.: Distributed Representations and Nested Compositional Structure. Ph.D. thesis (1994), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.5527
14. Turney, P.D., Pantel, P.: From frequency to meaning: Vector space models of semantics. J. Artif. Intell. Res. (JAIR) 37, 141–188 (2010)
15. Vishwanathan, S.V.N., Smola, A.J.: Fast kernels for string and tree matching. In: Becker, S., Thrun, S., Obermayer, K. (eds.) NIPS. pp. 569–576. MIT Press (2002)
16. Zanzotto, F.M., Dell'Arciprete, L.: Distributed tree kernels. In: Proceedings of International Conference on Machine Learning. pp. – (June 26July 1, 2012)
17. Zanzotto, F.M., Dell'Arciprete, L.: Transducing sentences to syntactic feature vectors: an alternative way to "parse"? In: Proceedings of the Workshop on Continuous Vector Space Models and their Compositionality. pp. 40–49 (8 August 2013), http://www.aclweb.org/anthology/W13-3205
18. Zanzotto, F.M., Korkontzelos, I., Fallucchi, F., Manandhar, S.: Estimating linear models for compositional distributional semantics. In: Proceedings of the 23rd International Conference on Computational Linguistics (COLING) (August, 2010)