Template Kernels for Dependency Parsing

Hillel Taub-Tabib
Hebrew University
Jerusalem, Israel

Yoav Goldberg
Bar-Ilan University
Ramat-Gan, Israel

{hillel.t,yoav.goldberg}@gmail.com

Amir Globerson
Hebrew University
Jerusalem, Israel
gamir@cs.huji.ac.il

Abstract

A common approach to dependency parsing is scoring a parse via a linear function of a set of indicator features. These features are typically manually constructed from templates that are applied to parts of the parse tree. The templates define which properties of a part should combine to create features. Existing approaches consider only a small subset of the possible combinations, due to statistical and computational efficiency considerations. In this work we present a novel kernel which facilitates efficient parsing with feature representations corresponding to a much larger set of combinations. We integrate the kernel into a parse reranking system and demonstrate its effectiveness on four languages from the CoNLL-X shared task.[1]

1 Introduction

Dependency parsing is the task of labeling a sentence x with a syntactic dependency tree $y \in \mathcal{Y}(x)$, where $\mathcal{Y}(x)$ denotes the space of valid trees over x. Each word in x is represented as a list of linguistic properties (e.g. word form, part of speech, base form, gender, number, etc.). In the graph based approach (McDonald et al., 2005b) parsing is cast as a structured linear prediction problem:

$$h_{\mathbf{v}}(x) = \underset{y \in \mathcal{Y}(x)}{\text{argmax}} \ \mathbf{v}^T \cdot \Phi(x, y) \quad (1)$$

where $\Phi(x, y) \in \mathbb{R}^d$ is a feature representation defined over a sentence and its parse tree, and $\mathbf{v} \in \mathbb{R}^d$ is a vector of parameters.

To construct an effective representation, $\Phi(x, y)$ is typically decomposed into local representations over parts p of the tree y:

$$\Phi(x, y) = \sum_{p \in y} \phi(x, p)$$

Standard decompositions include different types of parts: *arcs*, *sibling arcs*, *grandparent arcs*, etc. Feature templates are then applied to the parts to construct the local representations. The templates determine how the linguistic properties of the words in each part should combine to create features (see Section 2).

Substantial effort has been dedicated to the manual construction of feature templates (McDonald et al., 2005b; Carreras, 2007; Koo and Collins, 2010). Still, for both computational and statistical reasons, existing templates consider only a small subset of the possible combinations of properties. From a computational perspective, solving Eq. 1 involves applying the templates to y and calculating a dot product in the effective dimension of Φ. The use of many templates thus quickly leads to computational infeasibility (the dimensionality of $\mathbf{v}$, as well as the number of non-zero features in Φ, become very large). From a statistical perspective, the use of a large number of feature templates can lead to overfitting.

Several recent works have proposed solutions to the above problem. Lei et al., (2014) represented the space of all possible property combinations in an arc-factored model as a third order tensor and learned the parameter matrix for the tensor under a low rank assumption. In the context of transition parsers, Chen and Manning (2014) have implemented a neural network that uses dense representations of words and parts of speech as its input and implicitly considers combinations in its inner layers. Earlier work on transition-based dependency parsing used SVM classifiers with 2nd order polynomial kernels to achieve similar effects (Hall

[1]See https://bitbucket.org/hillel/templatekernels for implementation.

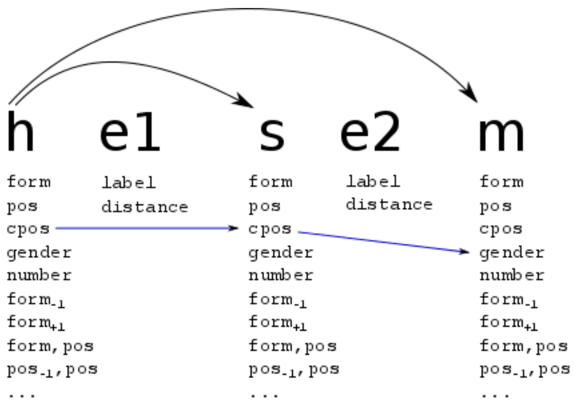

Figure 1: Feature template over the second order consecutive siblings part type. The part type contains slots for the head (h), sibling (s) and modifier (m) words, as well as for the two edges (e1 and e2). Each slot is associated with a set of properties. The directed path skips over the edge properties and defines the partial template <h-cpos=?; s-cpos=?; m-gender=?>.

et al., 2006). While training greedy transition-based parsers such as the ones used in (Chen and Manning, 2014) and (Hall et al., 2006) amounts to training a multiclass classifier, the graph-based parsing framework explored in (Lei et al., 2014) and in the present work is a more involved structured-learning task.

In this paper we present a kernel based approach to automated feature generation in the context of graph-based parsing. Compared to tensors and neural networks, kernel methods have the attractive properties of a convex objective and well understood generalization bounds (Shawe-Taylor and Cristianini, 2004). We introduce a kernel that allows us to learn the parameters for a representation similar to the tensor representation in (Lei et al., 2014) but without the low rank assumption, and without explicitly instantiating the exponentially many possible features.

In contrast to previous works on parsing with kernels (Collins and Duffy, 2002), in which the kernels are defined over trees and count the number of shared subtrees, our focus is on feature combinations. In that sense our work is more closely related to work on tree kernels for relation extraction (Zelenko et al., 2003; Culotta and Sorensen, 2004; Reichartz et al., 2010; Sun and Han, 2014), but the kernel we propose is designed to generate combinations of properties within selected part types and does not involve the all-subtrees representation.

2 Template Kernels

For simplicity, we begin with the case where parts p correspond to head modifier pairs (h, m) (i.e. all parts belong to the ”arc” part type). The features in $\phi(x, p)$ can then depend on any property of h, m and the sentence x. We denote properties related to h using the prefix h- (e.g., h-pos corresponds to the part-of-speech of the head), and similarly for m-. We also use e- to denote properties related to the triplets h, m, x (e.g., the surface distance between h, m in x is denoted by e-*dist*).

Templates defined over the ”arc” part type will then combine different properties of h, m and e, to create features. e.g. the template <h-form=?; e-dist=?; m-form=?,m-pos=?>, when applied to a dependency arc, may yield the feature: <h-form=dog;e-dist=1;m-form=black,m-pos=JJ>.

More generally, a parse tree *part* can be seen as ordered lists of slots that contain properties (different part types will contain different lists of slots). The feature templates defined over them select one property from each slot (possibly skipping some slots to produce partial templates). A template can thus be thought of as a directed path between the properties it selects in the different slots. Clearly, the number of possible templates in a given part type is exponential in the number of its slots. Figure 1 depicts the process for sibling parts.

As discussed in Section 1, manually constructed feature templates consider only a small subset of the combinations of properties (i.e. a small number of “good” paths is manually identified and selected). Our goal is to introduce a kernel that allows us to represent all possible paths for a part type in polynomial time.

Formally, let $\Phi(x, y) = \sum_{p \in y} \phi(x, p)$ be a feature representation which associates a feature with any distinct combination of properties in any of the tree parts in the training set. For a given part p, the effective dimensionality of $\phi(x, p)$ is thus $O(m^s)$ where s is the number of slots in p, and m is the maximal number of properties in a slot.

Explicitly representing $\Phi(x, y)$ is therefore often impractical. However, the well known “kernel trick” (Shawe-Taylor and Cristianini, 2004) implies that linear classifiers depend only on dot products between feature vectors and not on the feature vectors themselves. In the context of reranking (see Section 3), it means we can learn classifiers if we can calculate dot products $K(y_1, y_2) = \Phi^T(x_1, y_1) \cdot \Phi(x_2, y_2)$ for two sentences and candidate parses.[2]

We first note that such dot products can be expressed as sum of dot products over parts:

$$K(y_1, y_2) = \sum_{p \in y_1} \sum_{p' \in y_2} k(p, p')$$

where $k(p, p') = \phi(x_1, p) \cdot \phi(x_2, p)$.

To calculate $k(p, p')$ we'll assume for simplicity that p and p' are of the same type (otherwise $k(p, p') = 0$). Let p_{ij} and p'_{ij} be the values of the i'th property in the j'th slot in p, p' (e.g., for a second order sibling part as in Figure 1, $p_{1,4}$ will correspond to the label of the edge e2 in p) , and let $C_{p \leftrightarrow p'} \in \{0, 1\}^{m \times s}$ be a binary matrix comparing p and p' such that $\left[C_{p \leftrightarrow p'}\right]_{ij} = 1$ when $p_{ij} = p'_{ij}$ and 0 otherwise. Simple algebra yields that:

$$k(p, p') = \prod_j \vec{1}^T \cdot \left[C_{p \leftrightarrow p'}\right]_{:,j}$$

That is, calculating $k(p, p')$ amounts to multiplying the sums of the columns in C.[3] The runtime of $k(p, p')$ is then $O(m \times s)$ which means the overall runtime of $K(y_1, y_2)$ is $O(|y_1| \times |y_2| \times |s| \times |m|)$, where $|y_1|, |y_2|$ are the number of parts in y_1 and y_2.

Finally, note that adding 1 to one of the column counts of C corresponds to a slot that can be skipped to produce a partial template (this simulates a wild card property that is always on).

3 Kernel Reranker

We next show how to use the template kernels within a reranker. In the reranking approach (Collins and Koo, 2005; Charniak and Johnson, 2005), a base parser produces a list of k-best candidate parses for an input sentence and a separately trained reranking model is used to select the best one.

Features: Our feature vector will have two parts. One, $\Phi_g(x, y) \in \mathbb{R}^{d_1}$, consists of features obtained from manually constructed templates. The other, $\Phi_k(x, y) \in \mathbb{R}^{d_2}$, corresponds to our kernel features. We will not evaluate or store it, but rather use the kernel trick for implicitly learning with it, as explained below. The score of a candidate parse y for sentence x is calculated via the following linear function:

$$\begin{aligned} \Phi(x, y) &= [\Phi_g(x, y), \Phi_k(x, y)] \\ h_{\mathbf{v}}(x, y) &= \mathbf{v} \cdot \Phi(x, y) \end{aligned} \qquad (2)$$

Learning For learning we use the passive-aggressive algorithm (Crammer et al., 2006; McDonald et al., 2005a), and adapt it to use with kernels. Formally, let $S = \{(x_i, \mathcal{K}(x_i))\}_{i=1}^n$ be a training set of size n such that $\mathcal{K}(x_i) = \{y_{i1}, \ldots, y_{ik}\}$ is the set of k-best candidate trees produced for the sentence x_i. Assume that y_{i1} is the optimal tree in terms of Hamming distance to the gold tree.

A key observation to make is that the $\mathbf{v}$ generated by the PA algorithm will depend on two parameters. One is a weight vector $\mathbf{w} \in \mathbb{R}^{d_1}$, in the manually constructed Φ_g feature space. The other is a set of weights α_{ij} with $i = 1, \ldots, n$ and $j = 1, \ldots, k$ corresponding to the j^{th} candidate for the i^{th} sample.[4] The score is then given by:

$$f_{\mathbf{w},\alpha}(x, y) = \mathbf{v} \cdot \Phi(x, y) = \mathbf{w} \cdot \Phi_g(x, y) + f_\alpha(x, y)$$

where:

$$f_\alpha(x, y) = \sum_{i,j} \alpha_{ij} \cdot (K(y_{i1}, y) - K(y_{ij}, y))$$

We can now rewrite the updates of the PA algorithm using $\mathbf{w}, \alpha$, as described in Alg 1.[5]

4 Implementation

The classifier depends on parameters α_{ij}, which are updated using the PA algorithm. In the worst case, all nk of these may be non-zero. For large datasets, this may slow down both learning and prediction.

[2] For brevity we'll omit x from the kernel parameters and use $K(y_1, y_2)$ instead of $K((x_1, y_1), (x_2, y_2))$.

[3] We omit the proof, but intuitively, the product of column sums is equal to the number of 1 valued paths between elements in the different columns of C. Each such path corresponds to a path in p and p' where all the properties have identical values. i.e. it corresponds to a feature that is active in both $\phi(x_1, p)$ and $\phi(x_2, p')$ and thus contributes 1 towards the dot product.

[4] This follows from tracing the steps of PA and noting their dependence on dot products.

[5] The denominator in line 5 is equal to $\|\Phi_g(x_i, y_{ij}) - \Phi_g(x_i, y_{i1})\|^2 + K(y_{ij}, y_{ij}) - 2K(y_{ij}, y_{i1}) + K(y_{i1}, y_{i1})$ so it can be calculated efficiently using the kernel. $\|y_{i1} - y_{ij}\|_1$ is the hamming distance between y_{i1} and y_{ij}. The updates for $\bar{\alpha}$ are equivalent to averaging over all alphas in iterations $1, ..., T$. We use this form to save space.

Below we discuss implementation techniques to mitigate this problem. To facilitate the discussion we rewrite the dot-product computation as follows:

$$f_\alpha(x,y) = \sum_{p' \in y} \hat{f}_\alpha(x,p') \tag{3}$$

where:

$$\hat{f}_\alpha(x,p') = \sum_{i,j} \alpha_{ij} \left(\sum_{p \in y_{i1}} k(p,p') - \sum_{p \in y_{ij}} k(p,p') \right)$$

Reducing Prediction Runtime From Equation 3 we note several facts. First, prediction involves calculating $k(p,p')$ for every combination of a part p from the support instances (i.e., those for which $\alpha_{ij} > 0$) and part p' from the instances in the k-best list. Our implementation thus maintains its support as a set of parts rather than a set of instances.

Second, parts that appear in both y_{i1} and y_{ij} do not affect the result of $f_\alpha(x,y)$ since they cancel each other out. Our implementation thus only updates the support set with parts that belong exclusively to either y_{i1} or y_{ij}. This improves performance significantly since the number of non-overlapping parts in y_{i1} and y_{ij} is typically much smaller than the total number of parts therein.

Another important performance gain is obtained by caching the results of $\hat{f}_\alpha(x,p')$ when calculating $f_\alpha(x,y)$ for the different instances in the k-best list. This avoids recalculating the summation for parts that occur multiple times in the k-best list. Once again, this amounts to a considerable gain, as the number of distinct parts in the k-best list is much smaller than the total number of parts therein.

Reducing Training Runtime We greatly improve training speed by caching the results of $f_\alpha(x_i, y_{ij})$ between training iterations so that on each repeating invocation of the function, only the support parts added since the previous iteration need to be considered. Since the predictions of the learning algorithm become increasingly more accurate, the number of added support parts decreases sharply between iterations[6], and so does the runtime. In practice, all iterations from the 3rd onwards have negligible runtime compared to the first and second iterations. This technique allows us to comfortably train the kernel predictor on large datasets.

[6]On correct predictions, τ_t at line 5 of Alg 1 is 0, so no update is taking place and no support parts are added.

Algorithm 1 PA Algorithm for Template Kernels

```
Input: $S = \{(x_i, \mathcal{K}(x_i))\}_{i=1}^{n}$, $NumIters$, Aggressiveness parameter $C$
 1: $\forall i,j\ \alpha_{ij} \leftarrow 0,\ \bar{\alpha}_{ij} \leftarrow 0; T \leftarrow n \times NumIters$
 2: for $t = 1$ to $T$ do
 3:     $i \leftarrow t \mod n$
 4:     $j \leftarrow \operatorname{argmax}_{j:\, y_{ij} \in \mathcal{K}(x_i)} f_{\mathbf{w},\alpha}(x_i, y_{ij})$
 5:     $\tau_t \leftarrow \min\left\{C, \frac{f_{\mathbf{w},\alpha}(x,y_{ij}) - f_{\mathbf{w},\alpha}(x,y_{i1}) + \|y_{i1} - y_{ij}\|_1}{\|\Phi(x_i,y_{ij}) - \Phi(x_i,y_{i1})\|^2}\right\}$
 6:     $\alpha_{ij} \leftarrow \alpha_{ij} + \tau_t$
 7:     $\bar{\alpha}_{ij} \leftarrow \bar{\alpha}_{ij} + \tau_t (T - t + 1)$
 8:     $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \tau_t (\Phi_g(x_i, y_{i1}) - \Phi_g(x_i, y_{ij}))$
 9: end for
10: $\forall i,j,\ \bar{\alpha}_{ij} \leftarrow \frac{\bar{\alpha}_{ij}}{T},\ \bar{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{w}^{(t)}$
Output: predictor: $\operatorname{argmax}_{y \in \mathcal{K}(x)} (f_{\bar{\mathbf{w}},\bar{\alpha}}(x,y))$
```

5 Experimental Setup

Datasets We test our system on 4 languages from the CoNLL 2006 shared task, all with rich morphological features.[7] The properties provided for each word in these datasets are its *form*, part of speech (*pos*), coarse part of speech (*cpos*), *lemma* and *morph* features (number, gender, person, etc. around 10-20 feats in total). We use 20-fold jackknifing to create the k-best lists for the reranker (Collins and Duffy, 2002).

Base Parser The base parser used in experiments was the sampling parser of Lei et al. (2014), augmented to produce the k-best trees encountered during sampling. The parser was set to use feature templates over third order part types, but its tensor component and global templates were deactivated.

Features The manual features Φ_g were based on first to third order templates from Lei et al. (2014). For the kernel features Φ_k we annotated the nodes and edges in each tree with the properties in Table 1. We used a first order template kernel to train a model using all the the possible combinations of head, edge and modifier properties. Our kernel also produces all the property combinations of the head and modifier words (disregarding the edge properties).

[7]Our property combination approach is less relevant for treebanks that do not specify morphological properties. This is the

Node Unigram properties:		
form	form_{-1}	form_{+1}
pos	pos_{-1}	pos_{+1}
cpos	cpos_{-1}	cpos_{+1}
$\forall i$ morph_i		
Node Bigram properties:		**Edge prop:**
pos_{-1}, pos		label
pos, pos_{+1}		len, dist
pos, form		always on
$\forall i$ pos, morph_i		

Table 1: Linguistic properties for nodes and edges.

Results For each language we train a *Kernel Reranker* by running Alg 1 for 10 iterations over the training set, using k-best lists of size 25 and C set to infinity. As baseline, we train a *Base Reranker* in the same setup but with kernel features turned off. Table 2 shows the results for the two systems. Even though they use the same feature set, the base-reranker lags behind the base-parser. We attribute this to the fact that the reranker explores a much smaller fraction of the search space, and that the gold parse tree may not be available to it in either train or test time. However, the kernel-reranker significantly improves over the base-reranker. In Bulgarian and Danish, the kernel-reranker outperforms the base-parser. This is not the case for Slovene and Arabic, which we attribute to the low oracle accuracy of the k-best lists in these languages. As is common in reranking (Jagarlamudi and Daumé III, 2012), our final system incorporates the scores assigned to sentences by the base parser: i.e. $score_{final}(x, y) = \beta score_{base}(x, y) + score_{reranker}(x, y)$. β is tuned per language on a development set.[8] Our final system outperforms the base parser, as well as TurboParser (Martins et al., 2013), a parser based on manually constructed feature templates over up to third order parts. The system lags slightly behind the sampling parser of Zhang et al. (2014) which additionally uses global features (not used by our system) and a tensor component for property combinations. Another important difference between the systems is that our search is severely restricted by the use of a reranker. It is likely that using our kernel in a graph-based parser will further improve its accuracy.

reason we did not select the English treebank.

[8]To obtain a development set we further split the reranker training sets into tuning training and a development sets (90/10). We then tune β per language on the respective development sets by selecting the best value from a list of $\{0,\ 0.05, \ldots, 3\}$

	Arabic	Slovene	Danish	Bulgarian
Base Parser	80.15	86.13	90.76	92.98
Base Reranker	79.46	84.61	90.36	92.27
Kernel Reranker	79.48	85.25	91.04	93.28
Final System	80.19	86.44	91.56	93.4
Turbo Parser	79.64	86.01	91.48	93.1
Zhang et al.	80.24	86.72	91.86	93.72

Table 2: System Performance (UAS excluding punctuation). TurboParser is (Martins et al., 2013), Zhang et al. is (Zhang et al., 2014)

	Arabic	Slovene	Danish	Bulgarian
Sentences	1,460	1,534	5,190	12,823
Avg. Sent Len	37	19	18	15
Support Parts	15,466	10,101	31,627	58,842
Training Time	6m	7m	31m	57m
tokens/sec	551	432	223	99

Table 3: Runtime statistics, measured on a standard Macbook Pro 2.8 GHz Core i7 using 8 threads.

Performance Table 3 lists the performance metrics of our system on the four evaluation treebanks. While training times are reasonable even for large datasets, the increase in support size causes prediction to become slow for medium and large training sets. The number of support instances is a general problem with kernel methods. It has been addressed using techniques like feature maps (Rahimi and Recht, 2007; Lu et al., 2014) and bounded online algorithms (Dekel et al., 2008; Zhao et al., 2012). The application of these techniques to template kernels is a topic for future research.

6 Conclusions

We present a kernel approach to graph based dependency parsing. The proposed method facilitates globally optimal parameter estimation in a high dimensional feature space, corresponding to the full set of property combinations. We implemented our solution as part of a parse reranking system, demonstrating state of the art results. Future work will focus on performance improvements, using the kernel on higher order parts, and integrating the kernel directly into a graph based dependency parser.

Acknowledgments This work is supported by the US-Israel Binational Science Foundation (BSF, Grant No 2012330) and by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI). We thank the NAACL HLT reviewers for their comments.

References

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *EMNLP-CoNLL*, pages 957–961.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 173–180.

Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, volume 1, pages 740–750.

Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 263–270. Association for Computational Linguistics.

Michael Collins and Terry Koo. 2005. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *The Journal of Machine Learning Research*, 7:551–585.

Aron Culotta and Jeffrey Sorensen. 2004. Dependency tree kernels for relation extraction. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 423. Association for Computational Linguistics.

Ofer Dekel, Shai Shalev-Shwartz, and Yoram Singer. 2008. The forgetron: A kernel-based perceptron on a budget. *SIAM Journal on Computing*, 37(5):1342–1372.

Johan Hall, Joakim Nivre, and Jens Nilsson. 2006. Discriminative classifiers for deterministic dependency parsing. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 316–323.

Jagadeesh Jagarlamudi and Hal Daumé III. 2012. Low-dimensional discriminative reranking. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 699–709.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1–11. Association for Computational Linguistics.

Tao Lei, Yu Xin, Yuan Zhang, Regina Barzilay, and Tommi Jaakkola. 2014. Low-rank tensors for scoring dependency structures. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 1381–1391.

Zhiyun Lu, Avner May, Kuan Liu, Alireza Bagheri Garakani, Dong Guo, Aurélien Bellet, Linxi Fan, Michael Collins, Brian Kingsbury, Michael Picheny, et al. 2014. How to scale up kernel methods to be as good as deep neural nets. *arXiv preprint arXiv:1411.4000*.

André FT Martins, Miguel Almeida, and Noah A Smith. 2013. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 617–622.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005a. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 91–98.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005b. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 523–530. Association for Computational Linguistics.

Ali Rahimi and Benjamin Recht. 2007. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184.

Frank Reichartz, Hannes Korte, and Gerhard Paass. 2010. Semantic relation extraction with kernels over typed dependency trees. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 773–782.

John Shawe-Taylor and Nello Cristianini. 2004. *Kernel methods for pattern analysis*. Cambridge university press.

Le Sun and Xianpei Han. 2014. A feature-enriched tree kernel for relation extraction. In *The 52nd Annual Meeting of the Association for Computational Linguistics*, volume 2, pages 61–67.

Dmitry Zelenko, Chinatsu Aone, and Anthony Richardella. 2003. Kernel methods for relation extraction. *The Journal of Machine Learning Research*, 3:1083–1106.

Yuan Zhang, Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2014. Greed is good if randomized: New inference for dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1013–1024.

Peilin Zhao, Jialei Wang, Pengcheng Wu, Rong Jin, and Steven CH Hoi. 2012. Fast bounded online gradient descent algorithms for scalable kernel-based online learning. *arXiv preprint arXiv:1206.4633*.