Dynamic Feature Selection for Dependency Parsing

He He **Hal Daumé III**
Department of Computer Science
University of Maryland
College Park, MD 20740
{hhe,hal}@cs.umd.edu

Jason Eisner
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
jason@cs.jhu.edu

Abstract

Feature computation and exhaustive search have significantly restricted the speed of graph-based dependency parsing. We propose a faster framework of *dynamic feature selection*, where features are added sequentially as needed, edges are pruned early, and decisions are made online for each sentence. We model this as a sequential decision-making problem and solve it by imitation learning techniques. We test our method on 7 languages. Our dynamic parser can achieve accuracies comparable or even superior to parsers using a full set of features, while computing fewer than 30% of the feature templates.

1 Introduction

Graph-based dependency parsing usually consists of two stages. In the scoring stage, we score all possible edges (or other small substructures) using a learned function; in the decoding stage, we use combinatorial optimization to find the dependency tree with the highest *total* score.

Generally linear edge-scoring functions are used for speed. But they use a large set of features, derived from feature templates that consider different conjunctions of the edge's attributes. As a result, parsing time is dominated by the scoring stage—computing edge attributes, using them to instantiate feature templates, and looking up the weights of the resulting features in a hash table. For example, McDonald et al. (2005a) used on average about 120 first-order feature templates on each edge, built from attributes such as the edge direction and length, the two words connected by the edge, and the parts of speech of these and nearby words.

We therefore ask the question: can we use fewer features to score the edges, while maintaining the effect that the true dependency tree still gets a higher score? Motivated by recent progress on dynamic feature selection (Benbouzid et al., 2012; He et al., 2012), we propose to add features one group at a time to the dependency graph, and to use these features together with interactions among edges (as determined by intermediate parsing results) to make hard decisions on some edges before all their features have been seen. Our approach has a similar flavor to cascaded classifiers (Viola and Jones, 2004; Weiss and Taskar, 2010) in that we make decisions for each edge at every stage. However, in place of relatively simple heuristics such as a global relative pruning threshold, we learn a featurized decision-making policy of a more complex form. Since each decision can affect later stages, or later decisions in the same stage, we model this problem as a sequential decision-making process and solve it by Dataset Aggregation (DAgger) (Ross et al., 2011), a recent iterative imitation learning technique for structured prediction.

Previous work has made much progress on the complementary problem: speeding up the decoding stage by pruning the search space of tree structures. In Roark and Hollingshead (2008) and Bergsma and Cherry (2010), pruning decisions are made locally as a preprocessing step. In the recent vine pruning approach (Rush and Petrov, 2012), significant speedup is gained by leveraging structured information via a coarse-to-fine projective parsing cas-

Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, pages 1455–1464,
Seattle, Washington, USA, 18-21 October 2013.

cade (Charniak et al., 2006). These approaches do not directly tackle the feature selection problem. Although pruned edges do not require further feature computation, the pruning step must itself compute similar high-dimensional features just to decide which edges to prune. For this reason, Rush and Petrov (2012) restrict the pruning models to a smaller feature set for time efficiency. We aim to do feature selection and edge pruning dynamically, balancing speed and accuracy by using only as many features as needed.

In this paper, we first explore standard static feature selection methods for dependency parsing, and show that even a few feature templates can give decent accuracy (Section 3.2). We then propose a novel way to dynamically select features for each edge while keeping the overhead of decision making low (Section 4). Our present experiments use the Maximum Spanning Tree (MST) parsing algorithm (McDonald et al., 2005a; McDonald and Pereira, 2006). However, our approach applies to other graph-based dependency parsers as well—including non-projective parsing, higher-order parsing, or approximations to higher-order parsing that use stacking (Martins et al., 2008), belief propagation (Smith and Eisner, 2008), or structured boosting (Wang et al., 2007).

2 Graph-based Dependency Parsing

In graph-based dependency parsing of an n-word input sentence, we must construct a tree $\boldsymbol{y}$ whose vertices $0, 1, \ldots n$ correspond to the root node (namely 0) and the ordered words of the sentence. Each directed edge of this tree points from a head (parent) to one of its modifiers (child).

Following a common approach to structured prediction problems, the score of a tree $\boldsymbol{y}$ is defined as a sum of local scores. That is, $s_{\boldsymbol{\theta}}(\boldsymbol{y}) = \boldsymbol{\theta} \cdot \sum_{E \in \boldsymbol{y}} \phi(E) = \sum_{E \in \boldsymbol{y}} \boldsymbol{\theta} \cdot \phi(E)$, where E ranges over small connected subgraphs of $\boldsymbol{y}$ that can be scored individually. Here $\phi(E)$ extracts a high-dimensional feature vector from E together with the input sentence, and $\boldsymbol{\theta}$ denotes a weight vector that has typically been learned from data.

The first-order model decomposes the tree into edges E of the form $\langle h, m \rangle$, where $h \in [0, n]$ and $m \in [1, n]$ (with $h \neq m$) are a head token and one of its modifiers. Finding the best tree requires first computing $\boldsymbol{\theta} \cdot \phi(E)$ for each of the n^2 possible edges.

Since scoring the edges independently in this way restricts the parser to a local view of the dependency structure, higher-order models can achieve better accuracy. For example, in the second-order model of McDonald and Pereira (2006), each local subgraph E is a triple that includes the head and two modifiers of the head, which are adjacent to each other. Other methods that use triples include grandparent-parent-child triples (Koo and Collins, 2010), or non-adjacent siblings (Carreras, 2007). Third-order models (Koo and Collins, 2010) use quadruples, employing grand-sibling and tri-sibling information.

The usual inference problem is to find the highest scoring tree for the input sentence. Note that in a valid tree, each token $1, \ldots, n$ must be attached to exactly one parent (either another token or the root 0). We can further require the tree to be projective, meaning that edges are not allowed to cross each other. It is well known that dynamic programming can be used to find the best projective dependency tree in $O(n^3)$ time, much as in CKY, for first-order models and some higher-order models (Eisner, 1996; McDonald and Pereira, 2006).[1] When the projectivity restriction is lifted, McDonald et al. (2005b) pointed out that the best tree can be found in $O(n^2)$ time using a minimum directed spanning tree algorithm (Chu and Liu, 1965; Edmonds, 1967; Tarjan, 1977), though only for first-order models.[2] We will make use of this fast non-projective algorithm as a subroutine in early stages of our system.

3 Dynamic Feature Selection

Unlike typical feature selection methods that fix a subset of selected features and use it throughout testing, in dynamic feature selection we choose features adaptively for each instance. We briefly introduce this framework below and motivate our algorithm from empirical results on MST dependency parsing.

[1] Although the third-order model of Koo and Collins (2010), for example, takes $O(n^4)$ time.

[2] The non-projective parsing problem becomes NP-hard for higher-order models. One approximate solution (McDonald and Pereira, 2006) works by doing projective parsing and then rearranging edges.

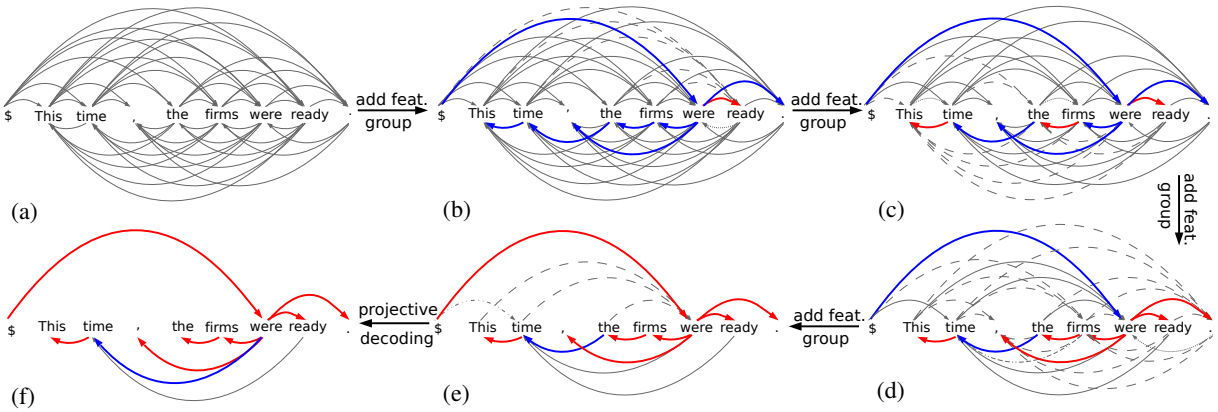

Figure 1: Dynamic feature selection for dependency parsing. (a) Start with all possible edges except those filtered by the length dictionary. (b) – (e) Add the next group of feature templates and parse using the non-projective parser. Predicted trees are shown as blue and red edges, where red indicates the edges that we then decide to lock. Dashed edges are pruned because of having the same child as a locked edge; 2-dot-3-dash edges are pruned because of crossing with a locked edge; fine-dashed edges are pruned because of forming a cycle with a locked edge; and 2-dot-1-dash edges are pruned since the root has already been locked with one child. (f) Final projective parsing.

3.1 Sequential Decision Making

Our work is motivated by recent progress on dynamic feature selection (Benbouzid et al., 2012; He et al., 2012; Grubb and Bagnell, 2012), where features are added sequentially to a test instance based on previously acquired features and intermediate prediction results. This requires sequential decision making. Abstractly, when the system is in some state $s \in S$, it chooses an action $a = \pi(s)$ from the action set A using its policy π, and transitions to a new state s', inducing some cost. In the specific case of dynamic feature selection, when the system is in a given state, it decides whether to add some more features or to stop and make a prediction based on the features added so far. Usually the sequential decision making problem is solved by reinforcement learning (Sutton and Barto, 1998) or imitation learning (Abbeel and Ng, 2004; Ratliff et al., 2004).

The dynamic feature selection framework has been successfully applied to supervised classification and ranking problems (Benbouzid et al., 2012; He et al., 2012; Gao and Koller, 2010). Below, we design a version that avoids overhead in our *structured prediction* setting. As there are n^2 possible edges on a sentence of length n, we wish to avoid the overhead of making many individual decisions about specific features on specific edges, with each decision considering the current scores of all other edges. Instead we will batch the work of dynamic feature selection into a smaller number of coarse-grained steps.

3.2 Strategy

To speed up graph-based dependency parsing, we first investigate time usage in the parsing process on our development set, section 22 of the Penn Treebank (PTB) (Marcus et al., 1993). In Figure 2, we observe that (a) feature computation took more than 80% of the total time; (b) even though non-projective decoding time grows quadratically in terms of the sentence length, in practice it is almost negligible compared to the projective decoding time, with an average of 0.23 ms; (c) the second-order projective model is significantly slower due to higher asymptotic complexity in both the scoring and decoding stages.

At each stage of our algorithm, we need to decide whether to use additional features to refine the edge scores. As making this decision separately for each of the n^2 possible edges is expensive, we instead propose a version that reduces the number of decisions needed. We show the process for one short sentence in Figure 1. The first step is to parse using the current features. We use the fast first-order non-projective parser for this purpose, since given observations (b) and (c), we cannot afford to run projective parsing multiple times. The single resulting tree (blue and red edges in Figure 1) has only

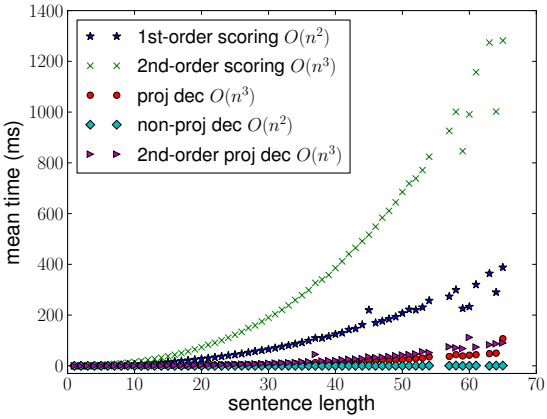

Figure 2: Time comparison of scoring time and decoding time on English PTB section 22.

n edges, and we use a classifier to decide which of these edges are reliable enough that we should "lock" them—i.e., commit to including them in the final tree. This is the only decision that our policy π must make. Locked (red) edges are definitely in the final tree. We also do constraint propagation: we rule out all edges that conflict with the locked edges, barring them from appearing in the final tree.[3] Conflicts are defined as violation of the projective parsing constraints:

- Each word has exactly one parent
- Edges cannot cross each other[4]
- The directed graph is non-cyclic
- Only one word is attached to the root

For example, in Figure 1(d), the dashed edges are removed because they have the same child as one of the locked (red) edges. The 2-dot-3-dash edge *time* ← *firms* is removed because it crosses the locked edge *(comma)* ← *were* (whereas we ultimately seek a projective parse). The fine dashed edge *were* ← *(period)* is removed because it forms a cycle with *were* → *(period)*. In Figure 1(e), the 2-dot-1-dash edge *(root)* → *time* is removed since we allow the root to have only one modifier.

[3]Constraint propagation also automatically locks an edge when all other edges with the same child have been ruled out.

[4]A reviewer asks about the cost of finding edges that cross a locked edge. Naively this is $O(n^2)$. But at most n edges will be locked during the entire algorithm, for a total $O(n^3)$ runtime—the same as *one* call to projective parsing, and far faster in practice. With cleverness this can even be reduced to $O(n^2 \log n)$.

Once constraint propagation has finished, we visit all edges (gray) whose fate is still unknown, and update their scores in parallel by adding the next group of features.

As a result, most edges will be locked in or ruled out without needing to look up all of their features. Some edges may still remain uncertain even after including all features. If so, a final iteration (Figure 1 (f)) uses the slower projective parser to resolve the status of these maximally uncertain edges. In our example, the parser does not figure out the correct parent of *time* until this final step. This final, accurate parser can use its own set of weighted features, including higher-order features, as well as the projectivity constraint. But since it only needs to resolve the few uncertain edges, both scoring and decoding are fast.

If we wanted our parser to be able to produce non-projective trees, then we would skip this final step or have it use a higher-order non-projective parser. Also, at earlier steps we would not prune edges crossing the locked edges.

4 Methods

Our goal is to produce a faster dependency parser by reducing the feature computation time. We assume that we are given three increasingly accurate but increasingly slow parsers that can be called as subroutines: a first-order non-projective parser, a first-order projective parser, and a second-order projective parser. In all cases, their feature weights have already been trained using the *full* set of features, and we will not change these weights. In general we will return the output of one of the projective parsers. But at early iterations, the non-projective parser helps us rapidly consider interactions among edges that may be relevant to our dynamic decisions.

4.1 Feature Template Ranking

We first rank the 268 first-order feature templates by forward selection. We start with an empty list of feature templates, and at each step we greedily add the one whose addition most improves the parsing accuracy on a development set. Since some features may be slower than others (for example, the "between" feature templates require checking all tokens in-between the head and the modifier), we could in-

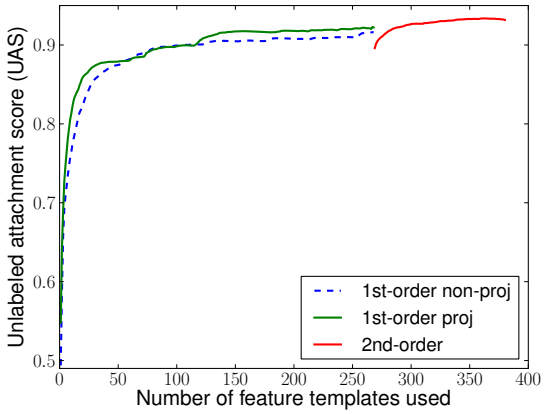

Figure 3: Forward feature selection result using the non-projective model on English PTB section 22.

stead select the feature template with the highest ratio of accuracy improvement to runtime. However, for simplicity we do not consider this: after grouping (see below), minor changes of the ranks within a group have no effect. The accuracy is evaluated by running the first-order non-projective parser, since we will use it to make most of the decisions. The 112 second-order feature templates are then ranked by adding them in a similar greedy fashion (given that all first-order features have already been added), evaluating with the second-order projective parser.

We then divide this ordered list of feature templates into K groups: $\{T_1, T_2, \ldots, T_K\}$. Our parser adds an entire group of feature templates at each step, since adding one template at a time would require too many decisions and obviate speedups. The simplest grouping method would be to put an equal number of feature templates in each group. From Figure 3 we can see that the accuracy increases significantly with the first few templates and gradually levels off as we add less valuable templates. Thus, a more cost-efficient method is to split the ranked list into several groups so that the accuracy increases by roughly the same amount after each group is added. In this case, earlier stages are fast because they tend to have many fewer feature templates than later stages. For example, for English, we use 7 groups of first-order feature templates and 4 groups of second-order feature templates. The sequence of group sizes is 1, 4, 10, 12, 47, 33, 161 and 35, 29, 31, 17 for first- and second-order parsing respectively.

4.2 Sequential Feature Selection

Similar to the length dictionary filter of Rush and Petrov (2012), for each test sentence, we first deterministically remove edges longer than the maximum length of edges in the training set that have the same head POS tag, modifier POS tag, and direction. This simple step prunes around 40% of the non-gold edges in our Penn Treebank development set (Section 6.1) at a cost of less than 0.1% in accuracy.

Given a test sentence of length n, we start with a complete directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{E} = \{\langle h, m\rangle : h \in [0, n], m \in [1, n]\}$. After the length dictionary pruning step, we compute T_1 for all remaining edges to obtain a pruned weighted directed graph. We predict a parse tree using the features so far (other features are treated as absent, with value 0). Then for each edge in this intermediate tree, we use a binary linear classifier to choose between two actions: $A = \{lock, add\}$. The *lock* action ensures that $\langle h, m\rangle$ appears in the final parse tree by pruning edges that conflict with $\langle h, m\rangle$.[5] If the classifier is not confident enough about the parent of m, it decides to *add* to gather more information. The *add* action computes the next group of features for $\langle h, m\rangle$ and all other competing edges with child m.

(Since we classify the edges one at a time, decisions on one edge may affect later edges. To improve efficiency and reduce cascaded error, we sort the edges in the predicted tree and process them as above in descending order of their scores.)

Now we can continue with the second iteration of parsing. Overall, our method runs up to $K = K_1 + K_2$ iterations on a given sentence, where we have K_1 groups of first-order features and K_2 groups of second-order features. We run $K_1 - 1$ iterations of non-projective first-order parsing (adding groups $T_1, \ldots, T_{K_1-1}$), then 1 iteration of *projective* first-order parsing (adding group T_{K_1}), and finally K_2 iterations of projective second-order parsing (adding groups $T_{K_1+1}, \ldots T_K$).

Before each iteration, we use the result of the previous iteration (as explained above) to prune some edges and add a new group of features to the rest. We

[5] If the conflicting edge is in the current predicted parse tree (which can happen because of non-projectivity), we forbid the model to prune it. Otherwise in rare cases the non-projective parser at the next stage may fail to find a tree.

then run the relevant parser. Each of the three parsers has a different set of feature weights, so when we switch parsers on rounds K_1 and $K_1 + 1$, we must also *change* the weights of the previously added features to those specified by the new parsing model.

In practice, we can stop as soon as the fate of all edges is known. Also, if no projective parse tree can be constructed at round K_1 using the available unpruned edges, then we immediately fall back to returning the non-projective parse tree from round $K_1 - 1$. This FAIL case rarely occurs in our experiments (fewer than 1% of sentences).

We report results both for a first-order system where $K_2 = 0$ (shown in Figure 1 and Algorithm 1) and for a second-order system where $K_2 > 0$.

Algorithm 1 DynFS($\mathcal{G}(\mathcal{V}, \mathcal{E}), \pi$)

$\mathcal{E} \leftarrow \{\langle h, m\rangle : |h - m| \leq \text{lenDict}(h, m)\}$
Add T_1 to all edges in $\mathcal{E}$
$\hat{\boldsymbol{y}} \leftarrow$ non-projective decoding
for $i = 2$ to K **do**
  $\mathcal{E}_{\text{sort}} \leftarrow$ sort unlocked edges $\{E : E \in \hat{\boldsymbol{y}}\}$ in descending order of their scores
  for $\langle h, m\rangle \in \mathcal{E}_{\text{sort}}$ **do**
    if $\pi(\psi(\langle h, m\rangle)) == \textit{lock}$ **then**
      $\mathcal{E} \leftarrow \mathcal{E} \setminus \{\{\langle h', m\rangle \in \mathcal{E} : h' \neq h\} \bigcup$
      $\{\langle h', m'\rangle \in \mathcal{E} :$ crosses $\langle h, m\rangle\} \bigcup$
      $\{\langle h', m'\rangle \in \mathcal{E} :$ cycle with $\langle h, m\rangle\}\}$
      if $h == 0$ **then**
        $\mathcal{E} \leftarrow \mathcal{E} \setminus \{\langle 0, m'\rangle \in \mathcal{E} : m' \neq m\}$
      end if
    else
      Add T_i to $\{\langle h', m'\rangle \in \mathcal{E} : m' == m\}$
    end if
  end for
  if $i == K$ **then**
    $\hat{\boldsymbol{y}} \leftarrow$ projective decoding
  else if $i \neq K$ **or** FAIL **then**
    $\hat{\boldsymbol{y}} \leftarrow$ non-projective decoding
  end if
end for
return $\hat{\boldsymbol{y}}$

5 Policy Training

We cast this problem as an imitation learning task and use Dataset Aggregation (DAgger) Ross et al. (2011) to train the policy iteratively.

5.1 Imitation Learning

In imitation learning (also called apprenticeship learning) (Abbeel and Ng, 2004; Ratliff et al., 2004), instead of exploring the environment directed by its feedback (reward) as in typical reinforcement learning problems, the learner observes expert demonstrations and aims to mimic the expert's behavior. The expert demonstration can be represented as trajectories of state-action pairs, $\{(s_t, a_t)\}$ where t is the time step. A typical approach to imitation learning is to collect supervised data from the expert's trajectories to learn a policy (multiclass classifier), where the input is $\psi(s)$, a feature representation of the current state (we call these *policy features* to avoid confusion with the *parsing features*), and the output is the predicted action (label) for that state.

In the sequential feature selection framework, it is hard to directly apply standard reinforcement learning algorithms, as we cannot assign credit to certain features until the policy decides to stop and let us evaluate the prediction result. On the other hand, knowing the gold parse tree makes it easy to obtain expert demonstrations, which enables imitation learning.

5.2 DAgger

Since the above approach collects training data only from the expert's trajectories, it ignores the fact that the distribution of states at training time and that at test time are different. If the learned policy cannot mimic the expert perfectly, one wrong step may lead to states never visited by the expert due to cumulative errors. This problem of insufficient exploration can be alleviated by iteratively learning a policy trained under states visited by both the expert and the learner (Ross et al., 2011; Daumé III et al., 2009; Kääriäinen, 2006).

Ross et al. (2011) proposed to train the policy iteratively and aggregate data collected from the previous learned policy. Let π^* denote the expert's policy and s_{π_i} denote states visited by executing π_i. In its simplest parameter-free form, in each iteration, we first run the most recently learned policy π_i; then for each state s_{π_i} on the trajectory, we collect a training example $(\psi(s_{\pi_i}), \pi^*(s_{\pi_i}))$ by labeling the state with the expert's action. Intuitively, this step intends to correct the learner's mistakes and pull it back to the

expert's trajectory. Thus we can obtain a policy that performs well under its own induced state distribution.

5.3 DAgger for Feature Selection

In our case, the expert's decision is rather straightforward. Replace the policy π in Algorithm 1 by an expert. If the edge under consideration is a gold edge, it executes *lock*; otherwise, it executes *add*. Basically the expert "cheats" by knowing the true tree and always making the right decision. On our PTB dev set, it can get 96.47% accuracy[6] with only 2.9% of the first-order features. This is an upper bound on our performance.

We present the training procedure in Algorithm 2. We begin by partitioning the training set into N folds. To simulate parsing results at test time, when collecting examples on $\mathcal{T}^i$, similar to cross-validation, we use parsers trained on $\bar{\mathcal{T}}^i = \mathcal{T} \setminus \mathcal{T}^i$. Also note that we show only one pass over training sentences in Algorithm 2; however, multiple passes are possible in practice, especially when the training data is limited.

Algorithm 2 DAgger($\mathcal{T}$, π^*)

```
Split the training sentences T into N folds T^1, T^2, ..., T^N
Initialize D ← ∅, π_1 ← π*
for i = 1 to N do
  for G(V, E) ∈ T^i do
    Sample trajectories {(s_{π_i}, π_i(s_{π_i}))} by DynFS(G(V, E), π_i)
    D ← D ⋃ {(ψ(s), π*(s))}
  end for
end for
Train policy π_{i+1} on D
return Best π_i evaluated on development set
```

5.4 Policy Features

Our linear edge classifier uses a feature vector ψ that concatenates all previously acquired parsing features together with "meta-features" that reflect confidence in the edge. The classifier's weights are fixed across iterations, but ψ(edge) changes per iteration.

We standardize the edge scores by a sigmoid function. Let $\dot{s}$ denote the normalized score, defined by $\dot{s}_{\boldsymbol{\theta}}(\langle h, m\rangle) = 1/(1 + \exp\{-s_{\boldsymbol{\theta}}(\langle h, m\rangle)\})$. Our meta-features for $\langle h, m\rangle$ include

- current normalized score, and normalized score before adding the current feature group
- margin to the highest scoring competing edges, i.e., $\dot{s}(\boldsymbol{w}, \langle h, m\rangle) - \max_{h'} \dot{s}(\boldsymbol{w}, \langle h', m\rangle)$ where $h' \in [0, n]$ and $h' \neq h$
- index of the next feature group to be added

We also tried more complex meta-features, for example, mean and variance of the scores of competing edges, and structured features such as whether the head of e is locked and how many locked children it currently has. It turns out that *given all the parsing features*, the margin is the most discriminative meta-feature. When it is present, other meta-features we added do not help much, Thus we do not include them in our experiments due to overhead.

6 Experiment

6.1 Setup

We generate dependency structures from the PTB constituency trees using the head rules of Yamada and Matsumoto (2003). Following convention, we use sections 02–21 for training, section 22 for development and section 23 for testing. We also report results on six languages from the CoNLL-X shared task (Buchholz and Marsi, 2006) as suggested in (Rush and Petrov, 2012), which cover a variety of language families. We follow the standard training/test split specified in the CoNLL-X data and tune parameters by cross validation when training the classifiers (policies). The PTB test data is tagged by a Stanford part-of-speech (POS) tagger (Toutanova et al., 2003) trained on sections 02–21. We use the provided gold POS tags for the CoNLL test data. All results are evaluated by the unlabeled attachment score (UAS). For fair comparison with previous work, punctuation is included when computing parsing accuracy of all CoNLL-X languages but not English (PTB).

For policy training, we train a linear SVM classifier using Liblinear (Fan et al., 2008). For all languages, we run DAgger for 20 iterations and se-

[6]The imperfect performance is because the accuracy is measured with respect to the gold parse trees. The expert only makes optimal pruning decisions but the performance depends on the pre-trained parser as well.

Language	Method	First-order				Second-order			
		Speedup	Cost(%)	UAS(D)	UAS(F)	Speedup	Cost(%)	UAS(D)	UAS(F)
Bulgarian	DynFS	3.44	34.6	91.1	91.3	4.73	16.3	91.6	92.0
	VineP	3.25	-	90.5	90.7	7.91	-	91.6	92.0
Chinese	DynFS	2.12	42.7	91.0	91.3	2.36	31.6	91.6	91.9
	VineP	1.02	-	89.3	89.5	2.03	-	90.3	90.5
English	DynFS	5.58	24.8	91.7	91.9	5.27	49.1	92.5	92.7
	VineP	5.23	-	91.0	91.2	11.88	-	92.2	92.4
German	DynFS	4.71	21.0	89.2	89.3	6.02	36.6	89.7	89.9
	VineP	3.37	-	89.0	89.2	7.38	-	90.1	90.3
Japanese	DynFS	4.80	15.6	93.7	93.6	8.49	7.53	93.9	93.9
	VineP	4.60	-	91.7	92.0	14.90	-	92.1	92.0
Portuguese	DynFS	4.36	32.9	87.3	87.1	6.84	40.4	88.0	88.2
	VineP	4.47	-	90.0	90.1	12.32	-	90.9	91.2
Swedish	DynFS	3.60	37.8	88.8	89.0	5.04	22.1	89.5	89.8
	VineP	4.64	-	88.3	88.5	13.89	-	89.4	89.7

Table 1: Comparison of speedup and accuracy with the vine pruning cascade approach for six languages. In the setup, DynFS means our dynamic feature selection model, VineP means the vine pruning cascade model, UAS(D) and UAS(F) refer to the unlabeled attachment score of the dynamic model (D) and the full-feature model (F) respectively. For each language, the speedup is relative to its corresponding first- or second-order model using the full set of features. Results for the vine pruning cascade model are taken from Rush and Petrov (2012). The cost is the percentage of feature templates used per sentence on edges that are *not pruned by the dictionary filter*.

lect the best policy evaluated on the development set among the 20 policies obtained from each iteration.

6.2 Baseline Models

We use the publicly available implementation of MSTParser[7] (with modifications to the feature computation) and its default settings, so the feature weights of the projective and non-projective parsers are trained by the MIRA algorithm (Crammer and Singer, 2003; Crammer et al., 2006).

Our feature set contains most features proposed in the literature (McDonald et al., 2005a; Koo and Collins, 2010). The basic feature components include lexical features (token, prefix, suffix), POS features (coarse and fine), edge length and direction. The feature templates consists of different conjunctions of these components. Other than features on the head word and the child word, we include features on in-between words and surrounding words as well. For PTB, our first-order model has 268 feature templates and 76,287,848 features; the second-order model has 380 feature templates and 95,796,140 features. The accuracy of our full-feature models is comparable or superior to previous results.

[7] http://www.seas.upenn.edu/~strctlrn/MSTParser/MSTParser.html

6.3 Results

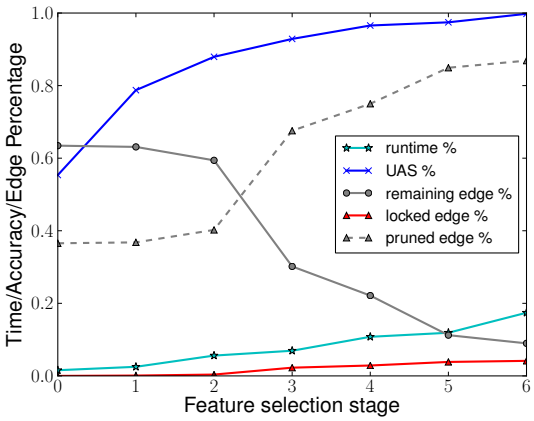

Figure 4: System dynamics on English PTB section 23. Time and accuracy are relative to those of the baseline model using full features. Red (locked), gray (undecided), dashed gray (pruned) lines correspond to edges shown in Figure 1.

In Table 1, we compare the dynamic parsing models with the full-feature models and the vine pruning cascade models for first-order and second-order

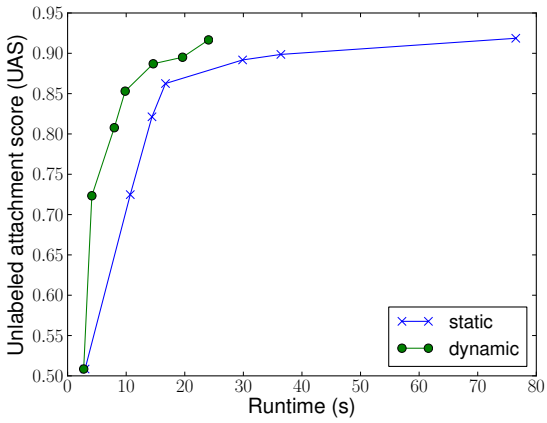

Figure 5: Pareto curves for the dynamic and static approaches on English PTB section 23.

parsing. The *speedup* for each language is defined as the speed relative to its full-feature baseline model. We take results reported by Rush and Petrov (2012) for the vine pruning model. As speed comparison for parsing largely relies on implementation, we also report the percentage of feature templates chosen for each sentence. The *cost* column shows the average number of feature templates computed for each sentence, expressed as a percentage of the number of feature templates if we had only pruned using the length dictionary filter.

From the table we notice that our first-order model's performance is comparable or superior to the vine pruning model, both in terms of speedup and accuracy. In some cases, the model with fewer features even achieves higher accuracy than the model with full features. The second-order model, however, does not work as well. In our experiments, the second-order model is more sensitive to false negatives, i.e. pruning of gold edges, due to larger error propagation than the first-order model. Therefore, to maintain parsing accuracy, the policy must make high-precision pruning decisions and becomes conservative. We could mitigate this by training the original parsing feature weights in conjunction with our policy feature weights. In addition, there is larger overhead during when checking non-projective edges and cycles.

We demonstrate the dynamics of our system in Figure 4 on PTB section 23. We show how the runtime, accuracy, number of locked edges and undecided edges change over the iterations in our first-order dynamic projective parsing. From iterations 1 to 6, we obtain parsing results from the non-projective parser; in iteration 7, we run the projective parser. The plot shows relative numbers (percentage) to the baseline model with full features. The number of remaining edges drops quickly after the second iteration. From Figure 3, however, we notice that even with the first feature group which only contains one feature template, the non-projective parser can almost achieve 50% accuracy. Thus, ideally, our policy should have locked that many edges after the first iteration. The learned policy does not imitate the expert perfectly, either because our policy features are not discriminative enough, or because a linear classifier is not powerful enough for this task.

Finally, to show the advantage of making dynamic decisions that consider the interaction among edges on the given input sentence, we compare our results with a static feature selection approach on PTB section 23. The static algorithm does no pruning except by the length dictionary at the start. In each iteration, instead of running a fast parser and making decisions online, it simply adds the next group of feature templates to all edges. By forcing both algorithms to stop after each stage, we get the Pareto curves shown in Figure 5. For a given level of high accuracy, our dynamic approach (black) is much faster than its static counterpart (blue).

7 Conclusion

In this paper we present a dynamic feature selection algorithm for graph-based dependency parsing. We show that choosing feature templates adaptively for each edge in the dependency graph greatly reduces feature computation time and in some cases improves parsing accuracy. Our model also makes it practical to use an even larger feature set, since features are computed only when needed. In future, we are interested in training parsers favoring the dynamic feature selection setting, for example, parsers that are robust to missing features, or parsers optimized for different stages.

Acknowledgements

This work was supported by the National Science Foundation under Grant No. 0964681. We thank the anonymous reviewers for very helpful comments.

References

P. Abbeel and A. Y. Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of ICML*.

D. Benbouzid, R. Busa-Fekete, and B. Kégl. 2012. Fast classification using space decision DAGs. In *Proceedings of ICML*.

S. Bergsma and C. Cherry. 2010. Fast and accurate arc filtering for dependency parsing. In *Proceedings of COLING*.

S. Buchholz and E. Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *CoNLL*.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL*.

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel coarse-to-fine PCFG parsing. In *Proceedings of ACL*.

Y. J. Chu and T. H. Liu. 1965. On the shortest arborescence of a directed graph. *Science Sinica*, 14.

Koby Crammer and Yoram Singer. 2003. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3:951–991.

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

Hal Daumé III, John Langford, and Daniel Marcu. 2009. Search-based structured prediction. *Machine Learning Journal (MLJ)*.

J. Edmonds. 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards*, (71B):233–240.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: an exploration. In *Proceedings of COLING*.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

Tianshi Gao and Daphne Koller. 2010. Active classification based on value of classifier. In *Proceedings of NIPS*.

Alexander Grubb and J. Andrew Bagnell. 2012. SpeedBoost: Anytime prediction with uniform near-optimality. In *AISTATS*.

He He, Hal Daumé III, and Jason Eisner. 2012. Cost-sensitive dynamic feature selection. In *ICML Inferning Workshop*.

Matti Kääriäinen. 2006. Lower bounds for reductions. Talk at the Atomic Learning Workshop (TTI-C), March.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of ACL*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

André F. T. Martins, Dipanjan Das, Noah A. Smith, and Eric P. Xing. 2008. Stacking dependency parsers. In *Proceedings of EMNLP*.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of EACL*, pages 81–88.

Ryan McDonald, K. Crammer, and Fernando Pereira. 2005a. Online large-margin training of dependency parsers. In *Proceedings of ACL*.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005b. Non-projective dependency parsing using spanning tree algorithms. In *Proc. of EMNLP*.

N. Ratliff, D. Bradley, J. A. Bagnell, and J. Chestnutt. 2004. Boosting structured prediction for imitation learning. In *Proceedings of ICML*.

B. Roark and K. Hollingshead. 2008. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of COLING*.

Stéphane. Ross, Geoffrey J. Gordon, and J. Andrew. Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of AISTATS*.

Alexander Rush and Slav Petrov. 2012. Vine pruning for efficient multi-pass dependency parsing. In *Proceedings of NAACL*.

David A. Smith and Jason Eisner. 2008. Dependency parsing by belief propagation. In *EMNLP*.

Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning : An Introduction*. MIT Press.

R. E. Tarjan. 1977. Finding optimum branchings. *Networks*, 7(1):25–35.

Kristina Toutanova, Dan Klein, Christopher Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *NAACL*.

Paul Viola and Michael Jones. 2004. Robust feal-time face detection. *International Journal of Computer Vision*, 57:137–154.

Qin Iris Wang, Dekang Lin, and Dale Schuurmans. 2007. Simple training of dependency parsers via structured boosting. In *Proceedings of IJCAI*.

David Weiss and Ben Taskar. 2010. Structured prediction cascades. In *Proceedings of AISTATS*.

H. Yamada and Y. Matsumoto. 2003. Statistical dependency analysis with Support Vector Machines. In *Proceedings of IWPT*.