A non-DNN Feature Engineering Approach to Dependency Parsing – FBAML at CoNLL 2017 Shared Task

Xian Qian xianqian@fb.com
Yang Liu yangli@fb.com
Facebook Applied Machine Learning / 1 Facebook Way

Abstract

For this year's multilingual dependency parsing shared task, we developed a pipeline system, which uses a variety of features for each of its components. Unlike the recent popular deep learning approaches that learn low dimensional dense features using non-linear classifier, our system uses structured linear classifiers to learn millions of sparse features. Specifically, we trained a linear classifier for sentence boundary prediction, linear chain conditional random fields (CRFs) for tokenization, part-of-speech tagging and morph analysis. A second order graph based parser learns the tree structure (without relations), and a linear tree CRF then assigns relations to the dependencies in the tree. Our system achieves reasonable performance – 67.87% official averaged macro F1 score.

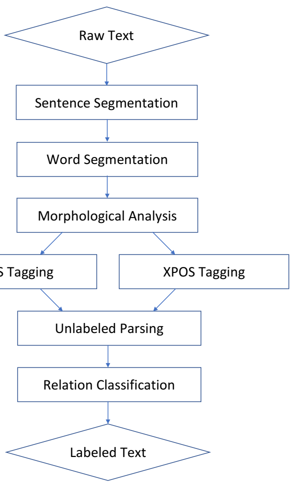

Figure 1: System architecture: pipeline components for universal dependency parsing.

1 Introduction

Our system for the universal dependency parsing shared task in CoNLL 2017 (Zeman et al., 2017) follows a typical pipeline framework.

The system architecture is shown in Figure 1, which consists of the following components : (1) sentence segmentor, which segments raw text into sentences, (2) tokenizer that tokenizes sentences into words, or performs word segmentation for Asian languages, (3) morphologic analyzer generates morphologic features, (4) part-of-speech (POS) tagger generates universal POS tags and language specific POS tags, (5) parser predicts tree structures without relations, (6) a relation predictor assigns relations to the dependencies in the tree.

For each component, we take a non deep learning based approach, that is the typical structured linear classifier that learns sparse features, but requires heavy feature engineering.

Sentence segmentation, tokenization, POS tagger and morphologic analyzer are based on linear chain CRFs (Lafferty et al., 2001), and the relation predictor is based on linear tree CRFs. We train the pipeline for each language independently using the training portion of the treebank and the official word embeddings for 45 languages provided by the organizers. Our system components are implemented in C++ with no third party toolkits. Due to the time limit, we did not optimize our system for speed or memory.

2 System Components

2.1 Sentence Segmentation

2.1.1 Task setup

We cast sentence segmentation as a classification problem at the character level, determining whether a character is the end of the sentence character (EOS). To obtain the gold labels, we aligned the raw text file with the conllu file with annotations.

Since most characters are not sentence boundaries, using all the characters will make the data very imbalanced. To address this problem, we only consider a character as a candidate trigger if it is labeled as EOS at least once in the training data. Intuitively this would prune many characters as EOS characters should be punctuation marks. However, we noticed that for English (possibly other languages too) many sentences in the data end without punctation, and thus the last character of the sentence will be added into the EOS character trigger set. To reduce the size of the triggers, we use a three label scheme for the characters.

- Label *N* for the character following the end of a sentence, and before the beginning of the next sentence. A typical example for this is the space between two sentences. Even for cases when punctuation marks are omitted, this applies to the space separating the two sentences.
- Label *E* represents a character is the end of a sentence, and its next character is the beginning of the next sentence. This category is introduced for sentences that are not split by space. For example, *past few years,...Great to have you on board!*, 'G' is the beginning of the second sentence, '.' before 'G' has a label of 'E'.
- Label *O* is used for all other cases. Note that a punctuation mark that ends a sentence will have a label of 'O' if there is a space following the sentence. In this case, EOS information is obtained by the 'N' label for the space.

Using this scheme, during testing, an EOS character is found if it is labeled as *E* or its next character is labeled as *N*. For training, we collect the characters labeled as *E* or *N* in the training set as candidates. Table 1 shows the number of candidates for each language. This significantly reduces the number of trigger candidates compared to considering all the characters.

languages	#trigger characters
cs_cltt, et, it, lv, pt, pt_br	3
en_lines, sl	4
no_nynorsk	5
no_bokmaal	6
ru_syntagrus	7
en	8
cs	14
zh	20
ja	23
others	2

Table 1: Number of trigger characters for EOS detection.

2.1.2 Features

We use a linear classifier for EOS detection. We tune the feature templates on the English development data, and apply to all the other languages. Detailed feature templates are described in Table 2. Features include the surrounding characters and their lower cases. For character types, we use digit and letters, and keep the other symbols. Take *12:00pm* as an example, it is represented as: *00:00aa*, where we replace all digits by '0' and all lower cased letters by 'a'. For languages that have spaces between words, we also use the surrounding 'words' split by spaces and the current character. For example, for the following example: *comes this story: President Bush* for character 'y', we have word features: $word_{-2}$=*this*, $word_{-1}$=*stor*, $word_1$=*:*, $word_2$=*President*.

2.2 Tokenization

2.2.1 Methodology

We use a sequence labeling model for tokenization. Each character will be labeled as one of the following tags:

- *B*: beginning of a multi-character token,
- *I*: inside a multi-character token,
- *E*: end of a multi-character token,
- *S*: single character token,
- *O*: other.

The labels are generated by aligning the raw text with the gold sentence segmentation with the word form column of the conllu table.

$char_i$, $-3 \le i \le +3$
$char_i char_{i+1}$, $-3 \le i \le +2$
$lowchar_i$, $-3 \le i \le +3$
$lowchar_i lowchar_{i+1}$, $-3 \le i \le +2$
$chartype_i$, $-3 \le i \le +3$
$chartype_i chartype_{i+1}$, $-3 \le i \le +2$
$word_i$, $i = -2, -1, 1, 2$
$wordtype_i$, $i = -2, -1, 1, 2$

Table 2: Feature templates for sentence segmentation. $char_i$ is the i^{th} character to the right of current character, $char_{-i}$ is the i^{th} character to the left of the current character. $lowchar$ is the lower cased character, $chartype$ is the character type, it can be digit, upper cased letter, lower cased letter or other. $word_i$ is surrounding 'words' splitted by spaces and the current character. $wordtype$ is the concatenation of character types

$chartype_0$, $chartype_{-1} chartype_0$
$word_{-1}$, $word_1$
$word_{-1} chartype_0$, $word_1 chartype_0$
$chartype_{-1} chartype_0$, $chartype_0 chartype_1$
transition feature

Table 3: Tokenization feature templates for languages with space between words (except Chinese and Japanese).

2.2.2 Features

Linear chain CRF is used to learn the model with character and word n-gram features. We used two sets of feature templates, one for languages having spaces between words including English, Arabic etc., the other for languages without spaces including Chinese and Japanese, as shown in Table 3 and 4. The first feature template set is tuned on English development set, the second one is tuned on Chinese development set.

2.3 POS Tagging and Morph Analysis

2.3.1 Methodology

For morphological analysis and POS tagging, we use the same model setup and features, therefore we group them together in this section. We used linear chain CRFs for these tasks (a sequence labeling task for each word in the sequence). As the morph features consist of several fields separated by a special symbol, we treat the prediction of each field as an independent task, and then combine the predictions from different models. For

$char_i char_{i+1}$, $-2 \le i \le 1$
$word_{-1}$, $word_1$
$char_0 word_{\text{left to current character}}$
$char_0 word_{\text{right to current character}}$
$word_{\text{left to current character}}$
$word_{\text{right to current character}}$
$word_{\text{left to left character}}$
$word_{\text{right to left character}}$
$word_{\text{left to right character}}$
$word_{\text{right to right character}}$
transition feature
transition feature + current character

Table 4: Tokenization feature templates for Chinese and Japanese. Words in these languages are obtained by maximum forward/backward matching.

POS tagging (both universal (UPOS) and language specific POS (XPOS) tagging), we use the same set of features as used for morph analysis, and the automatically predicted morph features. For languages that have multiple labels in XPOS tag, we use a similar strategy as for morph analysis, i.e., learning multiple taggers and combine the results.

2.3.2 Features

The list of feature templates are shown in Table 5. Note for POS tagging, as mentioned above, one additional feature is the morph feature, which comes from the automatic morph models.

The basic features includes word and lower cased word n-grams, prefixes and suffixes. With these features, the baseline UPOS tagger achieves 94.78% accuracy on the English development set. Since we do not use deep learning based approaches, incorporating pretrained word embeddings is not straightforward for linear classifiers. In our system, we clustered the word vectors using k-means, where $k = 2048$ and 10000, and then used the cluster n-grams as features.

2.4 Unlabeled Dependency Parsing

2.4.1 Methodology

Our dependency parser consists of two components, one is the unlabeled parser which only predicts the tree structures, the other is relation type prediction that assigns dependency relations to the dependencies. Originally, we trained a third order parser with word/POS/morph n-gram features, but it is too slow to extract features, especially

$word_i, -2 \leq i \leq 2$
$word_i char_{i+1}, -2 \leq i \leq 1$
$cluster_0, cluster_i cluster_{i+1}, i = -1, 0$
$lowerCasedWord_0$
$prefix_{i,j}, i = -1, 0, 1, 1 \leq j \leq 6$
$suffix_{i,j}, i = -1, 0, 1, 1 \leq j \leq 6$
$word_0 prefix_{i,j}, i = -1, 1, 1 \leq j \leq 6$
$word_0 suffix_{i,j}, i = -1, 1, 1 \leq j \leq 6$
$morph$ (invalid for morph analysis)
transition features

Table 5: Feature templates for morph analysis and POS tagging, where $prefix_{i,j}$ is the length = j prefix of the i^{th} word to the right of current word, $cluster_i$ is the cluster id of $word_i$

the third order features. So we chose to build a second order parser to balance speed and performance. We developed two versions of dependency parsers, one is pseudo-projective parser that handles treebanks that are nearly projective (projective dependencies $\% > 95\%$), the other is the 1-endpoint-crossing parser (Pitler et al., 2013; Pitler, 2014) that processes treebanks with more non-projective dependencies (projective dependencies $\% < 95\%$), such as Dutch-LassySmall, Ancient_Greek, Ancient_Greek-PROIEL, Basque, Latin-PROIEL and Latin. We modified the original third order 1-endpoint-crossing parsing algorithm to guarantee the unique derivation of any parse tree, because we need the top k parse trees for training.

2.4.2 Features

Our original third order parser includes 1000+ feature templates, and generated more than 100 million features on English data. As the features consume too much memory, making the parser rather slow, we kept only 260 templates, and use second order parser instead, which generated 15 million features. Most of the feature templates come from the previous works (Koo and Collins, 2010; McDonald et al., 2005), including word, POS ngrams and their combinations. We also add some morphology and word cluster n-grams. Detailed feature templates are described in Table 6.

2.5 Relation Classification

2.5.1 Methodology

Once the tree structure of a parse tree is obtained, we train a linear tree CRF to assign the relation type to each arc in the tree. Given a tree represented as a collection of arcs: $\mathcal{T} = \{e\}$, the tree CRF represents the potential function of $\mathcal{T}$ as the sum of the potential functions of arcs and arc pair chains:

$$\phi(\mathcal{T}) = \sum_{e} \phi(\mathrm{e}) + \sum_{\mathrm{e} \to \mathrm{e}'} \phi(\mathrm{e} \to \mathrm{e}') \qquad (1)$$

where $\phi(e)$ is the linear combination of node features in the CRF and $\phi(e \to e')$ is the linear combination of transition features in the CRF.

2.5.2 Features

For each arc $p \to c$, we use the same feature templates as in Table 6 to generate node features. For transition features, we simply use the relation type bigrams, i.e., $\mathrm{relation(g \to p)relation(p \to c)}$.

3 CoNLL Shared Task Results

3.1 Implementation details

All the classifiers, including linear chain CRF, tree CRF and second order dependency parser, are trained using 10-best MIRA (McDonald et al., 2005). Parameters are averaged to avoid overfitting. We found that k best MIRA consistently outperforms averaged perceptron about $0.1 - 0.2\%$ for all tasks.

For CRFs and the parser, we used the lazy decoding algorithm (Huang and Chiang, 2005) for fast k-best candidate generation, the complexity is nearly the same as 1-best decoding. Specifically, the time complexity for CRFs is $O(nL^2 + nk\log(k))$, and $O(n^4 + nk\log(k))$ for the parser. where n is the length of sentence.

Both CRFs are optimized for fast tagging: strings like words, POS tags are mapped to bit strings for efficient concatenation to generate feature strings, while the parser is not optimized. The actual running time for 1-endpoint-crossing parser is about 1.8 times of projective parser, though theoretically it should be 50x times slower. The main reason is that feature generating is much more slower than decoding, which is actully the same for both parsers. For fast training, we use hogwild strategy to update the parameters using 30 threads. Empirical results on English development data showed that compared with standard MIRA that only used single thread, the hogwild strategy get 5x speedup, the parser can be trained within 2.5 hours. While the performance is very competitive, only lost 0.1% UAS.

$\mathrm{p}_i.word$, $\mathrm{c}_i.word$, $-2 \le i \le 2$
$\mathrm{p}_i.word\ \mathrm{p}_{i+1}.word$, $\mathrm{c}_i.word\ \mathrm{c}_{i+1}.word$, $-2 \le i \le 1$
$\mathrm{p}_i.word\ \mathrm{c}_j.word$, $\mathrm{c}_i.word\ \mathrm{s}_j.word$, $\mathrm{g}_i.word\ \mathrm{c}_j.word$, $-1 \le i, j \le 1$
$\mathrm{g}_i.word\ \mathrm{p}_j.word\ \mathrm{c}_k.word$, $\mathrm{p}_i.word\ \mathrm{c}_j.word\ \mathrm{s}_k.word$, $-1 \le i, j, k \le 1$
$\mathrm{p}_i.word\ \mathrm{c}_j.word\ \mathrm{c}_{j+1}.word$, $-1 \le i \le 1$, $-1 \le j \le 0$
$\mathrm{p}_i.word\ \mathrm{p}_{i+1}.word\ \mathrm{c}_j.word$, $-1 \le i \le 0$, $-1 \le j \le 1$
$\mathrm{g}_0.word\ \mathrm{p}_i.word\ \mathrm{c}_j.word\ \mathrm{c}_{j+1}.word$, $-1 \le i \le 1$, $-1 \le j \le 0$
$\mathrm{g}_0.word\ \mathrm{p}_i.word\ \mathrm{p}_{i+1}.word\ \mathrm{c}_j.word$, $-1 \le i \le 0$, $-1 \le j \le 1$
$\mathrm{p}_i.word\ \mathrm{c}_j.word\ \mathrm{c}_{j+1}.word\ \mathrm{s}_0.word$, $-1 \le i \le 1$, $-1 \le j \le 0$
$\mathrm{p}_i.word\ \mathrm{p}_{i+1}.word\ \mathrm{c}_j.word\ \mathrm{s}_0.word$, $-1 \le i \le 0$, $-1 \le j \le 1$
replace $word$ above by $upos$, $xpos$, $lowCasedWord$, $wordCluster$, $morph$
combine the templates above with distance and direction of arcs

Table 6: Feature templates for unlabeled dependency parsing, where p_i, c_i, g_i, s_i are the i^{th} token right to the parent, child, grand parent, sibling. (to the left, if $i < 0$)

To cluster word vectors, we implemented fast k means using triangle inequality. We let k means run 20 iterations using 45 threads to quickly generate clusters. For languages without pretrained word vectors, such as *en_lines*, we use word vectors from *en* instead.

For surprised languages, we trained POS tagger, morphological analyzer and parser using the example data. The word cluster features are derived by running word2vec on the unlabeled dataset, and k-means clustering. For sentence segmentation and tokenization, we just used the models trained on English data, since the example dataset is quite limited.

3.2 Results on development data

The feature sets are tuned on English development data, except some languages specific tasks such as Chinese word segmentation. Table 7 shows the results on development dataset. We have the following observation regarding feature effect.

- Character type features are useful for sentence segmentation, which made 13% absolute F1 score improvement.
- Morphological features help the parser, resulting in an UAS 0.5% absolute F1 score improvement.
- For tokenization, word features i.e., $word_{-1}$ and $word_1$ in Table 3 are useful, which made 1% absolute F1 score improvement.
- Lemma features do not have a big effect on parsing. We compared using the gold lemma features vs. the automatically generated ones, with about 0.3% improvement from the former, and only 0.1% using the latter. Because of this our system did not do lemmartization for all the languages.
- Word cluster features have limited gains. We tried two different ways to convert the pretrained word vectors to binary features:

  (1) find the k nearest neighbors ($k = 3$ in experiments) in the embedding space, and use these neighors as features;

  (2) cluster the words into k clusters, ($k = 8, 16, \ldots, 2048, 10000, 100000$), and used the cluster features.

  The results on the English development set showed that the two approaches performed quite the same, both achieving 94.92% UPOS accuracy, 0.15% improvement over the baseline. In addition, we noticed that the word cluster features did not help when k is small. In our system submission, we used $k = 2048, 10000$ to generate the clusters.

  It is worth pointing out that such improvement from using the cluster features is quite limited compared to using embeddings in deep learning based methods. For example, using stacked word and character bi-LSTM-CRFs (Lample et al., 2016) achieved 95.75% POS tagging accuracy, and 96.00% using word+prefix/suffix embedding. We suspect that the converting real valued features to binary features (cluster features) loses too much information.

Language	Sentence	Words	UPOS	XPOS	Feats	UAS	LAS
ar	87.89%	92.15%	87.51%	81.08%	78.83%	70.33%	65.03%
bg	91.32%	99.77%	97.36%	75.72%	89.78%	87.75%	83.30%
ca	99.30%	99.73%	97.89%	97.92%	95.63%	88.69%	85.41%
cs	96.09%	99.96%	98.71%	92.11%	88.65%	89.46%	86.11%
cs_cac	99.50%	100.00%	99.00%	88.60%	83.94%	87.47%	84.26%
cs_cltt	74.36%	99.33%	89.20%	70.17%	65.97%	71.70%	68.37%
da	87.39%	100.00%	95.65%	0.00%	89.51%	79.36%	75.40%
de	93.68%	99.91%	93.39%	95.74%	79.83%	84.16%	79.76%
el	94.37%	99.78%	94.76%	94.80%	84.85%	81.99%	78.60%
en	81.43%	98.92%	93.89%	92.22%	92.23%	81.63%	78.11%
en_partut	95.73%	99.37%	94.37%	93.76%	88.81%	80.60%	76.35%
es	98.75%	99.70%	96.24%	0.00%	95.00%	86.95%	83.71%
es_ancora	96.66%	99.74%	97.93%	97.77%	95.64%	87.73%	84.74%
et	93.21%	99.07%	89.27%	91.13%	74.46%	71.24%	59.57%
eu	100.00%	99.99%	94.95%	0.00%	83.97%	76.87%	70.12%
fa	98.74%	99.48%	95.75%	95.52%	94.33%	83.27%	78.84%
fi	89.33%	99.77%	94.93%	96.07%	88.01%	78.73%	74.26%
fi_ftb	85.02%	100.00%	92.96%	0.00%	88.60%	78.78%	73.34%
fr	97.73%	99.16%	96.27%	0.00%	94.87%	87.67%	85.03%
fr_sequoia	90.58%	98.83%	96.05%	0.00%	92.82%	83.03%	80.43%
gl	96.68%	99.96%	96.83%	95.32%	99.79%	83.17%	79.89%
got	26.87%	100.00%	94.29%	95.23%	80.63%	70.46%	62.89%
grc	99.34%	100.00%	88.35%	77.68%	84.01%	68.13%	60.82%
grc_proiel	42.42%	100.00%	96.20%	96.54%	86.50%	74.26%	68.52%
he	99.49%	84.12%	80.55%	80.66%	75.41%	62.52%	57.81%
hi	98.55%	100.00%	96.33%	95.11%	87.43%	92.20%	88.09%
hr	97.48%	99.89%	96.64%	0.00%	81.14%	82.20%	76.22%
hu	98.19%	99.95%	93.40%	0.00%	59.68%	73.99%	64.48%
it	97.02%	99.44%	96.92%	96.28%	95.44%	87.65%	85.36%
it_partut	96.62%	99.16%	95.16%	94.89%	92.60%	82.55%	79.02%
ko	90.61%	97.91%	91.54%	86.50%	97.57%	62.80%	55.05%
la_ittb	77.03%	99.87%	96.75%	0.00%	88.37%	75.85%	70.70%
la_proiel	22.81%	100.00%	95.62%	95.37%	84.94%	68.66%	62.19%
lv	94.30%	99.66%	91.76%	26.01%	69.57%	72.57%	64.15%
nl	93.25%	99.66%	94.49%	0.00%	89.96%	82.82%	77.86%
nl_lassysmall	81.01%	99.81%	96.32%	0.00%	93.48%	79.35%	74.41%
no_bokmaal	96.08%	99.88%	97.28%	0.00%	91.16%	86.64%	83.22%
no_nynorsk	93.90%	99.94%	96.54%	0.00%	91.27%	84.78%	81.28%
pl	99.56%	99.17%	95.54%	0.00%	77.10%	84.51%	79.17%
pt	90.71%	99.53%	96.70%	0.00%	92.86%	87.45%	84.66%
pt_br	96.71%	99.82%	97.65%	97.57%	99.70%	89.23%	87.01%
ro	97.47%	99.63%	96.58%	9.64%	89.17%	86.94%	81.59%
ru	93.09%	99.79%	95.48%	94.35%	79.82%	82.12%	77.38%
ru_syntagrus	97.29%	99.72%	98.10%	0.00%	90.25%	88.91%	86.14%
sk	76.09%	99.93%	94.94%	0.87%	69.04%	82.74%	77.20%
sl	99.59%	99.94%	97.38%	14.87%	81.19%	87.41%	84.31%
sv	96.17%	99.88%	95.66%	0.00%	89.63%	80.57%	76.05%
sv_lines	87.19%	99.97%	94.89%	66.27%	99.97%	80.30%	75.22%
tr	96.98%	96.61%	89.31%	88.25%	77.59%	60.47%	52.49%
ur	99.10%	99.99%	93.46%	91.47%	77.62%	84.81%	78.02%
vi	97.40%	85.98%	78.32%	75.69%	85.84%	49.67%	44.20%
zh	98.50%	93.81%	87.21%	87.39%	92.25%	67.97%	62.75%

Table 7: Performance of our system on development dataset. XPOS accuracy for some languages are quite low due to the format issue.

Language	Sentence	Word	UPOS	XPOS	Feats	UAS	LAS
ar	85.69%	91.45%	86.59%	80.83%	78.27%	70.16%	64.89%
bg	91.5%	99.82%	97.67%	76.65%	90.18%	87.96%	83.89%
ca	99.35%	99.77%	97.68%	97.68%	95.35%	88.48%	85.02%
cs_cac	99.76%	99.94%	98.43%	87.99%	83.95%	87.54%	83.27%
cs_cltt	91.99%	99.59%	96.65%	83.34%	78.42%	80.26%	76.08%
cs	95.1%	99.99%	98.53%	91.45%	87.96%	88.14%	84.43%
cu	37.46%	100%	94.8%	95.02%	80.78%	73.68%	66.91%
da	81.41%	100%	95.84%	0%	90.55%	79.48%	75.59%
de	78.78%	99.61%	92.42%	96.74%	78.28%	79.17%	74.26%
el	28.28%	100%	95.75%	95.9%	84.99%	67.8%	61.53%
en_lines	86.95%	99.96%	95.4%	63.43%	99.96%	79.27%	74.67%
en_partut	98.1%	99.31%	94.2%	93.43%	88.49%	80.83%	76.68%
en	78.01%	98.98%	94.09%	93.46%	92.9%	80.8%	77.57%
es_ancora	98.67%	99.72%	97.87%	97.87%	95.66%	87.13%	83.81%
es	87.08%	99.98%	94.88%	62.88%	99.98%	80.33%	74.89%
et	92.63%	99.28%	89.9%	92.01%	75.16%	69.98%	58.48%
eu	99.75%	99.99%	94.59%	0%	83.32%	77.28%	70.76%
fa	99.25%	99.44%	95.86%	95.74%	94.4%	82.11%	77.74%
fi_ftb	86.46%	99.98%	93.11%	0%	89.1%	78.16%	72.08%
fi	89.48%	99.6%	95.15%	96.15%	88.07%	78.87%	74.51%
fr_sequoia	82.97%	99.19%	96.4%	0%	93.33%	82.91%	80.3%
fr	92.49%	98.84%	95.68%	0%	94.27%	83.83%	80.38%
gl	96.14%	99.98%	96.98%	95.82%	99.78%	83%	79.79%
got	28.35%	100%	94.85%	95.7%	81.73%	70.13%	62.64%
grc	98.96%	100%	87.25%	75.43%	81.13%	66.23%	58.42%
he	99.49%	80.93%	77.34%	77.34%	71.84%	58.54%	54.25%
hi	99.11%	99.99%	96.4%	95.8%	87.71%	92.31%	88.15%
hr	95.92%	99.88%	96.13%	0%	78.71%	82.95%	76.63%
hu	94.1%	99.75%	92.63%	0%	59.13%	74.22%	64.37%
id	92.14%	99.96%	93.41%	0%	99.46%	82.27%	75.74%
it	98.76%	99.56%	97.29%	97.17%	96.02%	88.42%	86.01%
ja	94.64%	93.32%	91.04%	0%	93.3%	80.71%	79.25%
ko	90.99%	98.24%	92.51%	88.35%	97.9%	68.14%	61.14%
la_ittb	92.91%	99.97%	97.49%	0%	91.59%	82.08%	77.62%
la_proiel	28.28%	100%	95.75%	95.9%	84.99%	67.8%	61.53%
lv	98.8%	99.45%	90.09%	26.52%	68.6%	69.15%	60.94%
nl_lassysmall	82.8%	99.93%	97.74%	0%	95.74%	84.92%	81.81%
nl	76.83%	99.79%	92.04%	0%	87.45%	77.93%	72%
no_bokmaal	96.26%	99.85%	96.64%	0%	90.78%	85.8%	82.6%
no_nynorsk	80.51%	96.99%	28.19%	0%	25.88%	23.91%	6.57%
pl	98.73%	98.99%	95.66%	0%	77.12%	84.05%	78.61%
pt_br	96.63%	99.83%	97.42%	97.42%	99.7%	87.57%	85.41%
pt	91.67%	99.34%	96.51%	0%	92.24%	85.15%	82.03%
ro	94.79%	99.64%	96.71%	9.43%	89.06%	86.68%	81.19%
ru_syntagrus	97.97%	99.69%	98.2%	0%	90.38%	89.36%	86.83%
ru	95.75%	99.81%	95.49%	95.24%	80.24%	81.58%	76.53%
sk	82.04%	99.98%	94.22%	0.9%	68.49%	81.58%	76.23%
sl	99.24%	99.93%	96.92%	14.55%	81.73%	85.74%	82.19%
sv_lines	87.08%	99.98%	94.88%	62.88%	99.98%	80.33%	74.89%
sv	94.92%	99.84%	96.05%	0%	89.81%	82.31%	77.7%
tr	96.32%	97.13%	91.3%	90.45%	78.62%	61.69%	53.08%
ur	97.67%	100%	93.17%	91.23%	78.21%	85.22%	78.61%
vi	92.44%	83.8%	75.84%	72.94%	83.56%	46.16%	40.89%
zh	98.5%	94.57%	88.36%	88.4%	92.9%	70.35%	65.15

Table 8: Official performance of our system on big treebanks. For language no_nynorsk, we used the model trained on another language, thus got very poor result.

Language	Sentence	Word	UPOS	XPOS	Feats	UAS	LAS
fr_partut	98.5%	98.88%	95.26%	95.04%	90.07%	84.2%	80.06%
ga	94.75%	99.64%	90.79%	89.9%	71.95%	76.51%	66.49%
gl_treegal	86.74%	97.91%	91.89%	86.48%	83.83%	73.44%	67.97%
kk	71.84%	94.45%	56.76%	57.13%	36.46%	44.67%	23.99%
la	98.14%	99.99%	88.57%	68.96%	67.45%	59.83%	48.33%
sl_sst	17.58%	100%	91.78%	13.97%	75.6%	56.73%	49.53%
ug	67.13%	96.94%	74.76%	76.87%	96.94%	54.54%	34.57%%
uk	92.49%	99.82%	90.08%	8.95%	62.26%	73.52%	65.17%
ar_pud	99.4%	89.08%	69.85%	0%	21.16%	53.74%	44.31%
cs_pud	96.13%	98.48%	96.32%	87.99%	83.56%	85.12%	79.76%
de_pud	88.4%	96.28%	83.82%	20.29%	30.75%	75.61%	69.19%
en_pud	98.06%	99.59%	94.46%	93.43%	91.38%	83.56%	79.88%
es_pud	94.88%	99.24%	88.01%	0%	54.03%	84.27%	77.09%
fr_pud	96.55%	96.63%	87.56%	0%	57.52%	78.74%	73.67%
hi_pud	1.17%	92.38%	79.04%	34.5%	13.38%	54.7%	43.46%
it_pud	93.97%	99.08%	93.19%	2.48%	57.18%	87.72%	84.41%
ja_pud	96.6%	93.57%	91.7%	0%	54.72%	82.04%	81.25%
pt_pud	97.32%	98.51%	88%	0%	58.8%	78.75%	72.85%
ru_pud	97.23%	97.15%	85.67%	79.23%	37.33%	77.5%	69.46%
sv_pud	94.11%	98.39%	92.54%	0%	69.4%	75.94%	70.76%
tr_pud	93%	94.94%	68.8%	0%	22.41%	52.75%	31.53%
bxr	90.62%	97.46%	49.11%	0%	39.9%	36.42%	17.08%
hsb	72.93%	94.32%	63.89%	0%	35.05%	37.58%	24.58%
kmr	92.91%	91.45%	58.03%	56.45%	31.56%	33.98%	25.85%
sme	98.09%	96.52%	53.98%	57.7%	30.7%	31.22%	17.1%

Table 9: Official performance of our system on small treebanks, PUD treebanks and suprise languages.

3.3 Official Results and Analysis

Detailed numbers for official runs on the test set (Nivre et al., 2017) are listed in Table 8 and Table 9.

Our system ranked the 15^{th} among the 33 submissions. Unfortunately, we found that for one language (no_nynorsk), we used the model trained on another language, therefore the performance is poor. Changing to the correct model would change our results from 67.87% averaged macro F1 score to 68.78%. For two languages *la* and *grc_proiel*, we trained the 1-endpoint-crossing parser, but used the projective parser for testing due to memory issue. On the development dataset, we found that such strategy lost about 0.5% LAS due to the inconsistent decoding algorithms between training and testing. For PUD treebanks that have no corresponding training portion, we used the model trained on the non-PUD dataset, e.g., used the model trained on *en* to parse *en_pud*.

Regarding speed, our parser is not optimized for running time nor memory. It spent 67 hours to parse all the languages using 10 threads. The peak memory usage is about 89GB when parsing *grc_proiel*. The most time consuming part in our system is feature generation that has a complexity of $O(n^3T)$, where $T = 260$ is the number of templates.

4 Conclusion and Future Work

We described our system for the universal dependency parsing task that relies heavily on feature engineering for each component in the pipeline. Our system achieves reasonable performance. An important observation we have is regarding the pretrained word embeddings. Unlike neural net based parsers that can effectively use large unlabeled data by pretrained word embedding, pictures of semi-supervised learning approaches for feature engineering based systems are unclear. Though we tried different ways in our work, the improvement is quite limited. In our future work, we plan to combine our system with neural net based approaches and explore some other semi-superivsed learning techniques.

References

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*. Association for Computational Linguistics, Stroudsburg, PA, USA, Parsing '05, pages 53–64. http://dl.acm.org/citation.cfm?id=1654494.1654500.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Uppsala, Sweden, pages 1–11. http://www.aclweb.org/anthology/P10-1001.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ICML '01, pages 282–289. http://dl.acm.org/citation.cfm?id=645530.655813.

Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. *CoRR* abs/1603.01360. http://arxiv.org/abs/1603.01360.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Association for Computational Linguistics, Ann Arbor, Michigan, pages 91–98. https://doi.org/10.3115/1219840.1219852.

Joakim Nivre et al. 2017. Universal Dependencies 2.0 CoNLL 2017 shared task development and test data. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University, Prague, `http://hdl.handle.net/11234/1-2184`. http://hdl.handle.net/11234/1-2184.

Emily Pitler. 2014. A crossing-sensitive third-order factorization for dependency parsing. *Transactions of the Association for Computational Linguistics* 2:41–54. https://www.transacl.org/ojs/index.php/tacl/article/view/193.

Emily Pitler, Sampath Kannan, and Mitchell Marcus. 2013. Finding optimal 1-endpoint-crossing trees. *Transactions of the Association for Computational Linguistics* 1:13–24. https://www.transacl.org/ojs/index.php/tacl/article/view/23.

Daniel Zeman, Filip Ginter, Jan Hajič, Joakim Nivre, Martin Popel, Milan Straka, and et al. 2017. CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. Association for Computational Linguistics, pages 1–20.