A Generative Model for Punctuation in Dependency Parsing

Xiang Lisa Li*, Dingquan Wang*, and Jason Eisner

* Equal Contribution
NLP has neglected punctuation

- Treebanks treat punctuation marks as ordinary tokens, but ARE THEY?

Punctuation is useful

Punctuation marks are correlated with prosody and the syntactic tree structure.
Hail the king Arthur Pendragon who wields Excalibur...
Hail the king Arthur Pendragon, who wields "Excalibur".
Summary...

- Punctuation marks are not words.
  - Just as prosody is not words.
- They do not belong in the tree.
- Only underlying punctuation marks are in the tree, where they surround certain phrases.
The **surface** punctuation has a non-obvious correlation with the tree.

The **underlying** punctuation has a more direct correlation with the tree.
Let’s exploit the underlying punctuation under a generative model!
Hail the king Arthur Pendragon who wields Excalibur.
P(punctuated tree | unpunctuated tree) = P(ε | root, unpunctuated tree)
* P( , | appos, unpunctuated tree)
P(punctuated tree | unpunctuated tree)
= P(ε | root, unpunctuated tree)
  * P( , | appos, unpunctuated tree)
  * P( , | rel-clause, unpunctuated tree)

Hail the king, Arthur Pendragon, who wields Excalibur.

Attach
$P(\text{punctuated tree} \mid \text{unpunctuated tree}) = P(\epsilon \mid \text{root, unpunctuated tree})$

* $P(\text{root} \mid \text{appos, unpunctuated tree})$
* $P(\text{appos} \mid \text{rel-clause, unpunctuated tree})$
* $P(\text{rel-clause} \mid \text{dobj, unpunctuated tree})$

Hail the king, Arthur Pendragon, who wields Excalibur.
Hail the king Arthur Pendragon, who wields “Excalibur”,.
\[ P(\text{appos}, \text{unpunctuated tree}) = 0.4 \]

<table>
<thead>
<tr>
<th>left punct</th>
<th>right punct</th>
<th>Prob</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>,</td>
<td>0.4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>0.3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>€</td>
<td>,</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>,</td>
<td>€</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>€</td>
<td>€</td>
<td>0.2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **f1**: indicator feature for no punctuation (€ €).
- **f2**: indicator feature that checks for symmetry.
- **f3**: span length feature for attaching commas.
- **f4**: indicator feature that checks for internal comma.
Hail the king, Arthur Pendragon, who wields Excalibur.
Rewriting — A Sliding Window Model

Left-absorb: $P(\text{[ , ]} \rightarrow \text{[ ]})$
Left-absorb: $P(\text{[ ]} \rightarrow \text{[ ]})$
Left-absorb: $P(\text{)} \rightarrow \text{)}$
identity: $P(\text{)} \rightarrow \text{)}$
swap: $P(\text{)} \rightarrow \text{)}$
Summary

generative story

• Underlying punctuation is generated at each tree node (not quite independently).
• Total underlying punctuation at each slot between words is rewritten into surface punctuation (independently).
In training, we observe the tree and surface punct. Want to recover the underlying punctuation.

\[ P(\text{underlying punct} \mid \text{tree, surface punct}) \]
Method 1. A Sampling Approach

\[ P(\text{underlying punct}| \text{tree}, \text{surface punct}) \propto P(\text{underlying} | \text{tree}) \times P(\text{surface} | \text{underlying}) \]

Hail the king Arthur Pendragon who wields Excalibur.

Hail the king Arthur Pendragon who wields "Excalibur".
Method 1. A Sampling Approach

\[ P(\text{underlying punct} | \text{tree, surface punct}) \times P(\text{underlying} | \text{tree}) \times P(\text{surface} | \text{underlying}) \]
Method 1. A Sampling Approach

\[ P(\text{underlying punct}| \text{tree}, \text{surface punct}) \propto P(\text{underlying}| \text{tree}) \times P(\text{surface}| \text{underlying}) \]
Method 2. $O(n)$ Dynamic Programming

Idea: Make a (weighted) CFG that generates just the possible underlyingly punctuated trees for this sentence

- Soft constraint 1: Underlying punctuation fits the tree structure
- Soft constraint 2: Underlying punctuation fits surface punctuation

Intersect 1 2 → our CFG
Method 2. O(n) Dynamic Programming

Idea: Make a (weighted) CFG that generates just the possible underlyingly punctuated trees for this sentence.

Soft constraint 1: Underlying punctuation fits the tree structure.

Soft constraint 2: Underlying punctuation fits surface punctuation.

Intersect 1 2 → our CFG
Method 2. O(n) Dynamic Programming

Idea: Make a (weighted) CFG that generates just the possible underlyingly punctuated trees for this sentence

Soft constraint 1: Underlying punctuation fits the tree structure

Soft constraint 2: Underlying punctuation fits surface punctuation

Hail the king Arthur Pendragon who wields Excalibur.

Intersect 1 2 → our CFG
Method 2. O(n) Dynamic Programming

Idea: Make a (weighted) CFG that generates just the possible underlyingly punctuated trees for *this* sentence

- Soft constraint 1: Underlying punctuation fits the tree structure
- Soft constraint 2: Underlying punctuation fits surface punctuation

Intersect 1 2  →  our CFG

\[ \text{PCFG} \bigcap \text{weighted FSA} = \text{weighted CFG} \]
Method 2. O(n) Dynamic Programming

Idea: Make a (weighted) CFG that generates just the possible underlyingly punctuated trees for this sentence

Soft constraint 1: Underlying punctuation fits the tree structure

Soft constraint 2: Underlying punctuation fits surface punctuation

Intersect 1 2 → our CFG

Now we can find the best underlyingly punctuated tree

• or sum over all of them for the model likelihood in training

For our CFG, this can be done in O(n) time
Results

Rules Learned from the Noisy Channel
Analysis of Attachment model
Punctuation Restoration
Syntactic Rephrasing
We learn high probability for Nunberg’s English Rules
Earlier, Kerry said, "in fact, answered the question."
Viterbi recovers good underlying punctuation

Real Examples (simplified for slide)

Oxford Comma
What if we don’t have gold parses?

Punctuation Restoration
average edit distance [the lower the better]

BiLSTM-CRF tagger
Always Period tagger
Ours
but, if true, the caper failed.
but the caper failed, if true, obviously ugly, and high perplexity under an LM

Syntactically Transform Sentence from Treebank
but, if true, the caper failed.
but if true, the caper failed.
but the caper failed, if true

Syntactically Transform Sentence with our Annotation
Let’s slow down for a second here.

We better just stop right now.
Implemented by Finite State Transducer

\[ ab \rightarrow ab \]
\[ ab \rightarrow a \]
\[ ab \rightarrow b \]
\[ ab \rightarrow ba \]
Implemented by Finite State Transducer

\[ ab \rightarrow ab \]
\[ ab \rightarrow a \]
\[ ab \rightarrow b \]
\[ ab \rightarrow ba \]
\( ab \rightarrow ab \)

\( ab \rightarrow a \)

\( ab \rightarrow b \)

\( ab \rightarrow ba \)
Implemented by Finite State Transducer

\[
\begin{align*}
ab & \rightarrow ab \\
ab & \rightarrow a \\
ab & \rightarrow b \\
ab & \rightarrow ba
\end{align*}
\]