Corpus transformations
Praktikum Verarbeitung natürlicher Sprache

Richard Mörbitz

28.05.2019
Praktical problems in parsing

Our basic CFG parser has some issues:

- Only binary and unary rules can be processed, but the training corpus is $n$-ary

\[
A \rightarrow BC \\
A \rightarrow a 
\]

vs.

\[
A \quad B \quad C \quad D \\
e \quad f \quad g 
\]

- Training corpus does not contain all English words

He is a brilliant deipnosophist.  \(\not\rightarrow\) No parse!

- Sparse data problem: rare words in the training corpus may get atypical probabilities
Outline

1. Binarization and Markovization
2. Unknown/rare word handling
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar

Each occurrence of two consecutive nonterminals leads to a new nonterminal.

No information about context is preserved.

Number of right-hand side nonterminals is fixed.
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar

Problems:

- Each occurrence of two consecutive nonterminals leads to a new nonterminal $X_i$
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar

Problems:
- Each occurrence of two consecutive nonterminals leads to a new nonterminal $X_i$
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar

Problems:
- Each occurrence of two consecutive nonterminals leads to a new nonterminal $X_i$
- No information about context is preserved
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar

Problems:
- Each occurrence of two consecutive nonterminals leads to a new nonterminal $X_i$
- No information about context is preserved
Straight-forward binarization

Binarizing the training corpus vs. binarizing the grammar

![Diagram showing binarization process]

**Problems:**
- Each occurrence of two consecutive nonterminals leads to a new nonterminal $X_i$
- No information about context is preserved
- Number of right-hand side nonterminals is fixed
Vertical and horizontal Markovization

Previous slide Markovization with \( v = 1 \) and „\( h = \infty \)“

In general store \( v \in \mathbb{N}_+ \) nonterminals towards the root and \( h \in \mathbb{N}_+ \) nonterminals to the left

1: \textbf{function} MARKOVIZE\((t = \sigma(t_1, \ldots, t_k))\)
2: \textbf{if} \( t \) is preterminal \textbf{then}
3: \textbf{return} \( t \)
4: \textbf{else if} \( k \leq 2 \) \textbf{then}
5: \textbf{return} ANNOTATE\((\sigma)\left(\text{MARKOVIZE}(t_1), \ldots, \text{MARKOVIZE}(t_k)\right)\)
6: \textbf{else}
7: \( \sigma' \leftarrow \text{ORIGINALLABEL}(\sigma)|\langle\text{label of } t_2, \ldots, \text{label of } t_{h+1}\rangle \)
8: \textbf{return} ANNOTATE\((\sigma)\left(\text{MARKOVIZE}(t_1), \text{MARKOVIZE}(\sigma'(t_2, \ldots, t_k))\right)\)

\textbf{Note:}

- \( \text{ANNOTATE}(\sigma) = \sigma^\wedge\langle l_1, \ldots, l_{v-1}\rangle \), where the \( l_i \) are the labels of the ancestors of \( \sigma \) which occur in the original tree
- If \( v = 1 \) or there are no parents, leave out \( ^\wedge\langle \rangle \)
After parsing: debinarization

**Why?**

- Debinarized parse trees are linguistically relevant
- Comparison with (debinarized) gold corpus

```plaintext
1: function DEBINARIZE(t = \sigma(t_1, \ldots, t_k))
2: if \text{ROOT}(t_k) \text{ is Markovization node with children } t'_1, t'_2 \text{ then}
3: return DEBINARIZE(\sigma(t_1, \ldots, t_{k-1}, t'_1, t'_2))
4: else
5: return \sigma(\text{DEBINARIZE}(t_1), \ldots, \text{DEBINARIZE}(t_k))
```

Also: remove ancestor annotation from each node (not shown)!

In material: both binarized and (debinarized) gold corpus
Outline

1. Binarization and Markovization

2. Unknown/rare word handling
Basic unking

Idea:
1. Replace words that are *unknown* to the parser by an „unknown token“ (here: UNK)
2. Assign some probability mass to rules that generate UNK
Basic unking

**Idea:**
1. Replace words that are *unknown* to the parser by an „unknown token“ (here: UNK)
2. Assign some probability mass to rules that generate UNK

**Implementation:**
1. Reflection in the parser: before parsing $w$ (requires words of grammar $\Sigma$)

   1. $wmap \leftarrow (w_i \mid i \in \{1, \ldots, |w|\})$
   2. $\text{for } i = 1, \ldots, |w| \text{ do}$
   3. $\text{if } w_i \notin \Sigma \text{ then}$
   4. $w_i \leftarrow \text{UNK}$

   and after parsing $w$ as $t$, restore original words

   1. $\text{for } i = 1, \ldots, |w| \text{ do}$
   2. $\text{LEAVES}(t)[i] \leftarrow wmap[i]$
Reflection in the grammar: replace rare words in the training corpus by UNK

**Require:** tree corpus `corpus` with terminal alphabet `Σ`, `threshold ∈ \mathbb{N}_+`

**Ensure:** every word occurring \(\leq threshold\) times in `corpus` is replaced by UNK

1: `wordcount ← (0 | i ∈ Σ)`
2: `for t ∈ corpus do`
3: `for i = 1, \ldots, |leaves(t)| do`
4: `wordcount[leaves(t)[i]] ← wordcount[leaves(t)[i]] + 1`
5: `for t ∈ corpus do`
6: `for i = 1, \ldots, |leaves(t)| do`
7: `if wordcount[leaves(t)[i]] ≤ threshold then`
8: `leaves(t)[i] ← UNK`

The grammar is induced from the modified corpus!
**Refinement**

**Observation:** each unknown word is assigned the same probability

- bad language model: certain words are more likely to occur
- may worsen the parsing of sentences with rare words

**Solution:** categorize unknown words based on their signature [KM03]

\[
\begin{align*}
    w_i &\leftarrow \text{UNK} \quad \mapsto \quad w_i \leftarrow \text{getSignature}(w_i, i) \\
    \text{LEAVES}(t)[i] &\leftarrow \text{UNK} \quad \mapsto \quad \text{LEAVES}(t)[i] \leftarrow \text{getSignature}(w_i, i)
\end{align*}
\]

- Some signatures can be found in the source code of the Berkely parser
- Here: „unknownLevel = 4“
- Heuristics, prone to overfitting
function getSignature(word, i)
if |word| = 0 then return UNK

letterSuffix ← isUpper(word[0]) ∧ none(isLower, word) ⇒ -AC
    isUpper(word[0]) ∧ i = 1 ⇒ -SC
    isUpper(word[0]) ⇒ -C
    any(isLower, word) ⇒ -L
    any(isLetter, word) ⇒ -U
    otherwise ⇒ -S

numberSuffix ← all(isDigit, word) ⇒ -N
    any(isDigit, word) ⇒ -n
    otherwise ⇒ ε

dashSuffix ← any((= '-'), word) ⇒ -H
    otherwise ⇒ ε

periodSuffix ← any((= '.'), word) ⇒ -P
    otherwise ⇒ ε

commaSuffix ← any((= ','), word) ⇒ -C
    otherwise ⇒ ε

wordSuffix ← |word| > 3 ⇒ toLower(word[|word|])
    otherwise ⇒ ε

return UNK · letterSuffix · numberSuffix · dashSuffix · periodSuffix · commaSuffix · wordSuffix
What to do?

Until 03.07.2019, 23:59,
- implement debinarization (3a)
- implement trivial unking for corpus and adapt parser (3b)
- implement 3 of
  - Markovization
  - smoothing
  - pruning
  - $n$-best parsing
  - heuristic search

$\{\text{next week}\}$

All tasks’ solutions are one submission!
You may send in your solutions earlier for feedback.
References I