On the internals of disco-dop

How to implement a state-of-the-art LCFRS parser

Kilian Gebhardt

Grundlagen der Programmierung, Fakultät Informatik, TU Dresden

November 16, 2018

Motivation

- LCFRS parsing is hard ($O(n^{m*k})$ where n, m, and k are sentence length, maximum numbers of nonterminals in a rule, and the fanout of the grammar, respectively.)

Motivation

- LCFRS parsing is hard ($O(n^{m*k})$ where n, m, and k are sentence length, maximum numbers of nonterminals in a rule, and the fanout of the grammar, respectively.)
- Exact inference with real world LCFRS might feasible up to length 30 (see Angelov and Ljunglöf 2014)?

Motivation

- LCFRS parsing is hard ($O(n^{m*k})$ where n, m, and k are sentence length, maximum numbers of nonterminals in a rule, and the fanout of the grammar, respectively.)
- Exact inference with real world LCFRS might feasible up to length 30 (see Angelov and Ljunglöf 2014)?
- We want to parse longer sentences and short sentences faster!

disco-dop

- Parsing framework developed by Andreas van Cranenburgh (cf. Cranenburgh, Scha, and Bod 2016)

disco-dop

- Parsing framework developed by Andreas van Cranenburgh (cf. Cranenburgh, Scha, and Bod 2016)
- Uses discontinuous data-oriented model (discontinuous tree-substitution grammar) at its core.

disco-dop

- Parsing framework developed by Andreas van Cranenburgh (cf. Cranenburgh, Scha, and Bod 2016)
- Uses discontinuous data-oriented model (discontinuous tree-substitution grammar) at its core.
- Employs a coarse-to-fine pipeline for parsing:
  1. PCFG stage
  2. LCFRS stage
  3. DOP stage

The coarse-to-fine pipeline (grammars)

- The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS)
  (see Goodman 2003 for continuous case).

[1] See `unknownword6` and `unknownword4` in
https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py

The coarse-to-fine pipeline (grammars)

- The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS)
  (see Goodman 2003 for continuous case).
- The original treebank t_1 is binarized/Markovized ($= t_2$) and a coarse prob. LCFRS is induced. (Grammar is binarized, simple, ordered, may contain chain rules)

[1] See `unknownword6` and `unknownword4` in
https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py

The coarse-to-fine pipeline (grammars)

- The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS) (see Goodman 2003 for continuous case).
- The original treebank t_1 is binarized/Markovized ($= t_2$) and a coarse prob. LCFRS is induced. (Grammar is binarized, simple, ordered, may contain chain rules)
- Discontinuity in t_2 is resolved by splitting categories. After binarizing again, we obtain t_3 and induce a PCFG. (Grammar is binarized, simple, may contain chain rules.)

[1] See `unknownword6` and `unknownword4` in https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py

The coarse-to-fine pipeline (grammars)

- The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS) (see Goodman 2003 for continuous case).
- The original treebank t_1 is binarized/Markovized ($= t_2$) and a coarse prob. LCFRS is induced. (Grammar is binarized, simple, ordered, may contain chain rules)
- Discontinuity in t_2 is resolved by splitting categories. After binarizing again, we obtain t_3 and induce a PCFG. (Grammar is binarized, simple, may contain chain rules.)
- Some preprocessing is applied to lexical rules to handle unknown words. (Stanford signatures[1])

[1] See `unknownword6` and `unknownword4` in https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py

The coarse-to-fine pipeline (application)

- Parse with stage s resulting in chart.

The coarse-to-fine pipeline (application)

- Parse with stage *s* resulting in chart.
- If successful, obtain a whitelist of items from chart:

The coarse-to-fine pipeline (application)

- Parse with stage s resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation

The coarse-to-fine pipeline (application)

- Parse with stage s resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item i, where $\alpha(i) \cdot \beta(i) \geq k$

The coarse-to-fine pipeline (application)

- Parse with stage s resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item i, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in k-best derivations

The coarse-to-fine pipeline (application)

- Parse with stage s resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item i, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in k-best derivations

(For PCFG $\rightarrow$ PLCFRS $k = 10,000$ is the default.)

The coarse-to-fine pipeline (application)

- Parse with stage s resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item i, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in k-best derivations

  (For PCFG $\rightarrow$ PLCFRS $k = 10,000$ is the default.)
- Next stage $s + 1$ prunes item i, if coarsify(i) is not in whitelist.

The coarse-to-fine pipeline (application)

- Parse with stage s resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item i, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in k-best derivations

  (For PCFG $\rightarrow$ PLCFRS $k = 10,000$ is the default.)
- Next stage $s + 1$ prunes item i, if coarsify(i) is not in whitelist.
- If unsuccessful, stop parsing and greedily/recursively select the largest possible items from chart as fallback strategy.

Representation of LCFRS rules I

$$A \rightarrow \langle x_1^{(1)} x_1^{(2)} x_2^{(1)}, x_2^{(2)} x_3^{(1)} x_4^{(1)} \rangle (B, C)$$

Representation of LCFRS rules I

$$A \rightarrow \langle \underbrace{x_1^{(1)}}_{\substack{0\\0}} \underbrace{x_1^{(2)}}_{\substack{1\\0}} \underbrace{x_2^{(1)}}_{\substack{0\\1}}, \underbrace{x_2^{(2)}}_{\substack{1\\0}} \underbrace{x_3^{(1)}}_{\substack{0\\0}} \underbrace{x_4^{(1)}}_{\substack{0\\1}} \rangle (B, C)$$

$i-1$ if $x_j^{(i)}$
1 if end of component

Representation of LCFRS rules I

$$A \rightarrow \langle \underbrace{x_1^{(1)}}_{\substack{0\\0}} \underbrace{x_1^{(2)}}_{\substack{1\\0}} \underbrace{x_2^{(1)}}_{\substack{0\\1}}, \underbrace{x_2^{(2)}}_{\substack{1\\0}} \underbrace{x_3^{(1)}}_{\substack{0\\0}} \underbrace{x_4^{(1)}}_{\substack{0\\1}} \rangle (B, C)$$

$i-1$ if $x_j^{(i)}$
1 if end of component

```
struct ProbRule {  // total: 32 bytes.
  double prob;  // 8 bytes
  uint32_t lhs;  // 4 bytes
  uint32_t rhs1;  // 4 bytes
  uint32_t rhs2;  // 4 bytes
  uint32_t args;  // 4 bytes => 32 max vars per rule
  uint32_t lengths;  // 4 bytes => same
  uint32_t no;  // 4 bytes
};
```

e.g. `args = 0b001010` and `lengths = 0b100100`.

Representation of LCFRS rules II

2. $A \rightarrow \langle x_1^{(1)}, x_2^{(1)} x_3^{(1)} \rangle (B)$ (same, with `rhs2 = 0`)

Representation of LCFRS rules II

2. $A \rightarrow \langle x_1^{(1)}, x_2^{(1)} x_3^{(1)} \rangle (B)$ (same, with `rhs2 = 0`)
3. $A \rightarrow \langle \alpha \rangle$

stored via a map $\Sigma \rightarrow$ `vector<uint32_t>` and a `vector<LexicalRule>` where:

```
struct LexicalRule {
  double prob;
  uint32_t lhs;
};
```

PCFG parsing I

bottom-up chart parsing (based on Bodenstab 2009's fast grammar loop)

```
populate_pos(chart, grammar, sentence)

for span in range(2, n+1):
  for left in range(1, n + 1 - span):
    right = left + span
    for lhs in grammar.nonts:
      for rule in grammar.rules[lhs]:
        for mid in range(left + 1, right):
          p1 = chart.getprob(left, mid, rule.rhs1)
          p2 = chart.getprob(mid, right, rule.rhs2)
          p_new = rule.prob + p1 + p2
          if chart.updateprob(left, right, p_new):
            chart.add_edge( ... )

    applyunary(left, right, chart, grammar)
```

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If $\texttt{span} \leq \delta$ and $\texttt{p\_new} < \eta \cdot \texttt{p\_best4cell}$, then prune.

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If $\texttt{span} \leq \delta$ and $\texttt{p\_new} < \eta \cdot \texttt{p\_best4cell}$, then prune.
- Only applied to binary rules.

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If $\texttt{span} \leq \delta$ and $\texttt{p\_new} < \eta \cdot \texttt{p\_best4cell}$, then prune.
- Only applied to binary rules.

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If `span` $\leq \delta$ and `p_new` $< \eta \cdot$ `p_best4cell`, then prune.
- Only applied to binary rules.

chart datastructures

- items are densely enumerated (`cellidx(start, stop, nonterminal)`)

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If `span` $\leq \delta$ and `p_new` $< \eta \cdot$ `p_best4cell`, then prune.
- Only applied to binary rules.

chart datastructures

- items are densely enumerated (`cellidx(start, stop, nonterminal)`)
- saves log-probabilities in vector (indexed by `cellidx`)

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If `span` $\leq \delta$ and `p_new` $< \eta \cdot$ `p_best4cell`, then prune.
- Only applied to binary rules.

chart datastructures

- items are densely enumerated (`cellidx(start, stop, nonterminal)`)
- saves log-probabilities in vector (indexed by `cellidx`)
- saves incoming edges for each item (`chart.parseforest`)

PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If `span` $\leq \delta$ and `p_new` $< \eta \cdot$ `p_best4cell`, then prune.
- Only applied to binary rules.

chart datastructures

- items are densely enumerated (`cellidx(start, stop, nonterminal)`)
- saves log-probabilities in vector (indexed by `cellidx`)
- saves incoming edges for each item (`chart.parseforest`)
- best derivation (or k-best derivations) retrieved afterwards by recursively selecting best edge

PCFG parsing III

mid filter = auxiliary data structure (size: $4 \cdot |N| \cdot n$) with entries

$$\begin{aligned}
\text{minleft}(A, j) &= \max\{\ i \mid [A, i, j] \in \text{chart}\} \\
\text{maxleft}(A, j) &= \min\ \{\ i \mid [A, i, j] \in \text{chart}\} \\
\text{minright}(A, j) &= \min\ \{\ j \mid [A, i, j] \in \text{chart}\} \\
\text{maxright}(A, j) &= \max\{\ j \mid [A, i, j] \in \text{chart}\}
\end{aligned}$$

replace “`for mid in range(left + 1, right)`” by

```
for mid in range(
        max(minright(B, left), maxleft(C, right)),
        min(maxright(B, left), minleft(C, right)))
```

LCFRS parsing

agenda driven LCFRS parser (with filter)

LCFRS parsing

agenda driven LCFRS parser (with filter)

```
populate_pos(...)

while not agenda.emtpy():
  item, prob = agenda.pop()
  chart.updateprob(item, prob)

  if item == goal and not exhaustive:
    break

  applyunaryrules(item, grammar, chart, agenda)
  for rule in lbinary[item.nont]:
    for item2 in chart.items[rule.rhs2]:
      process(rule, item, item2, chart, agenda, whitelist)
  for rule in rbinary[item.nont]:
    for item2 in chart.items[rule.rhs1]:
      process(rule, item2, item, chart, agenda, whitelist)
```

LCFRS parsing (heuristics)

- SX, SXlrgaps, etc. (Klein and Manning 2003 and Kallmeyer and Maier 2013)
- `score += length * MAX_LOGPROB`, i.e., smaller items are processed before larger items

LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length ≤ 64)

```
cdef cppclass SmallChartItem:
  uint32_t label
  uint64_t vec
```

LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length ≤ 64)

```
cdef cppclass SmallChartItem:
  uint32_t label
  uint64_t vec
```

- LCFRS Item (for sentences with length > 64)

```
cdef cppclass FatChartItem:
  uint32_t label
  uint64_t vec[SLOTS]
```

LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length ≤ 64)

```
cdef cppclass SmallChartItem:
  uint32_t label
  uint64_t vec
```

- LCFRS Item (for sentences with length > 64)

```
cdef cppclass FatChartItem:
  uint32_t label
  uint64_t vec[SLOTS]
```

- Combination of items based on algorithm in rparse's FastYFComposer

LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length ≤ 64)

  ```
  cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec
  ```

- LCFRS Item (for sentences with length > 64)

  ```
  cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
  ```

- Combination of items based on algorithm in rparse's FastYFComposer
- Items are indexed in the order they are found. Index is stored in a B-Tree map. Items are ordered by label (primary) and vec (secondary).

LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length ≤ 64)

  ```
  cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec
  ```

- LCFRS Item (for sentences with length > 64)

  ```
  cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
  ```

- Combination of items based on algorithm in rparse's FastYFComposer
- Items are indexed in the order they are found. Index is stored in a B-Tree map. Items are ordered by label (primary) and vec (secondary).
- Probabilities are stored in a vector, indexed by item index.

LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length ≤ 64)

  ```
  cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec
  ```

- LCFRS Item (for sentences with length > 64)

  ```
  cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
  ```

- Combination of items based on algorithm in rparse's FastYFComposer
- Items are indexed in the order they are found. Index is stored in a B-Tree map. Items are ordered by label (primary) and vec (secondary).
- Probabilities are stored in a vector, indexed by item index.
- Incoming edges are stored in a `vector[vector[Edge]]`, indexed by item index.

LCFRS Agenda

Agenda

- combines heap of (item, prob) and map: item $\rightarrow$ best probability

LCFRS Agenda

Agenda

- combines heap of (item, prob) and map: item $\rightarrow$ best probability
- while popping: check that best (item, prob) in heap satisfies map(item) $=$ prob, otherwise pop next

LCFRS Agenda

Agenda

- combines heap of (item, prob) and map: item $\rightarrow$ best probability
- while popping: check that best (item, prob) in heap satisfies map(item) $=$ prob, otherwise pop next
- on adding (item, prob): check that item $\notin$ map or map(item) $<$ prob, otherwise discard

References I

Krasimir Angelov and Peter Ljunglöf. "Fast Statistical Parsing with Parallel Multiple Context-Free Grammars". In: *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*. Gothenburg, Sweden: Association for Computational Linguistics, Apr. 2014, pp. 368–376. URL: https://www.aclweb.org/anthology/E14-1039.

Nathan Bodenstab. *Efficient Implementation of the CKY algorithm*. Tech. rep. 2009. URL: http://csee.ogi.edu/~bodensta/bodenstab_efficient_cyk.pdf.

References II

Andreas van Cranenburgh, Remko Scha, and Rens Bod. "Data-Oriented Parsing with discontinuous constituents and function tags". In: *Journal of Language Modelling* 4.1 (2016), pp. 57–111. DOI: 10.15398/jlm.v4i1.100.

Joshua Goodman. "Efficient parsing of DOP with PCFG-reductions". In: *Data-Oriented Parsing*. Ed. by Rens Bod, Khalil Sima'an, and Remko Scha. Stanford, CA, USA: CSLI Publications, 2003. Chap. 4. ISBN: 1575864355. URL: https://pdfs.semanticscholar.org/2943/16b9b0156eee9cd06c778e06966b77c20e83.pdf.

References III

Dan Klein and Christopher D Manning. “A parsing: fast exact Viterbi parse selection”. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*. Association for Computational Linguistics. 2003, pp. 40–47.

Laura Kallmeyer and Wolfgang Maier. “Data-driven Parsing using Probabilistic Linear Context-Free Rewriting Systems”. In: *Computational Linguistics* 39.1 (2013), pp. 87–119. DOI: 10.1162/COLI_a_00136.

Yue Zhang et al. “Chart Pruning for Fast Lexicalised-Grammar Parsing”. In: *Coling 2010: Posters*. Beijing, China: Coling 2010 Organizing Committee, Aug. 2010, pp. 1471–1479. URL: http://www.aclweb.org/anthology/C10-2168.