Dependency parsing and logistic regression

Sharon Goldwater

19 October 2017
Recap: transition-based dependency parsing

Basic transition-based parser:

- Configuration: consisting of the stack state, input buffer, and dependency relations found so far.

- Actions: e.g., Shift, LeftArc, RightArc.

- Assume we have an oracle that tells us $P(\text{action}|\text{configuration})$.

- Choosing the best action at each step (greedy parsing) produces a parse in linear time.
Today’s lecture

• Can we get better performance by not being quite so greedy?

• How do we evaluate a dependency parser?

• But really, how do we get $P(\text{action}|\text{configuration})$?
  – What do we need for training data?
  – What kind of model can we use?
Recap: parsing as search

Parser is searching through a very large space of possible parses.

- Greedy parsing is a depth-first strategy.
- **Beam search** is a limited breadth-first strategy.
Beam search: basic idea

• Instead of choosing only the best action at each step, choose a few of the best.

• Extend previous partial parses using these options.

• At each time step, keep a fixed number of best options, discard anything else.

Advantages:

• May find a better overall parse than greedy search,

• While using less time/memory than exhaustive search.
The agenda

An ordered list of configurations (parser state + parse so far).

• Items are ordered by score: how good a configuration is it?

• Implemented using a priority queue data structure, which efficiently inserts items into the ordered list.

• In beam search, we use an agenda with a fixed size (beam width). If new high-scoring items are inserted, discard items at the bottom below beam width.

Won’t discuss scoring function here; but beam search idea is used across NLP (e.g., in best-first constituency parsing, NNet models.)
Evaluating dependency parsers

- How do we know if beam search is helping?

- As usual, we can evaluate against a gold standard data set. But what evaluation measure to use?
Evaluating dependency parsers

• By construction, the number of dependencies is the same as the number of words in the sentence.

• So we do not need to worry about precision and recall, just plain old accuracy.

• **Labelled Attachment Score** (LAS): Proportion of words where we predicted the correct head and label.

• **Unlabelled Attachment Score** (UAS): Proportion of words where we predicted the correct head, regardless of label.
Building a classifier for next actions

We said:

- Assume we have an oracle that tells us $P(\text{action}|\text{configuration})$.

Where does that come from?
Classification for action prediction

We’ve seen **text classification**:

- Given (features from) text document, predict the class it belongs to.

Generalized classification task:

- Given features from observed data, predict one of a set of classes (labels).

Here, **actions** are the labels to predict:

- Given (features from) the current configuration, predict the next action.
Our goal is:

- Given (features from) the current configuration, predict the next action.

Our corpus contains annotated sentences such as:

Is this sufficient to train a classifier to achieve our goal?
Creating the right training data

Well, not quite. What we need is a sequence of the correct (configuration, action) pairs.

- Problem: some sentences may have more than one possible sequence that yields the correct parse. (see tutorial exercise)

- Solution: JM3 describes a procedure for converting each annotated sentence to a unique sequence of (configuration, action) pairs.

OK, finally! So what kind of model will we train?
Logistic regression

• Actually, we could use any kind of classifier (Naive Bayes, SVM, neural net...)

• Logistic regression is a standard approach that illustrates a different type of model: a discriminative probabilistic model.
  – So far, all our models have been generative.

• Even if you have seen it before, the formulation often used in NLP is slightly different from what you might be used to.
Generative probabilistic models

- Model the joint probability $P(\vec{x}, \vec{y})$
  - $\vec{x}$: the observed variables (what we’ll see at test time).
  - $\vec{y}$: the latent variables (not seen at test time; must predict).

<table>
<thead>
<tr>
<th>Model</th>
<th>$\vec{x}$</th>
<th>$\vec{y}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Bayes</td>
<td>features</td>
<td>classes</td>
</tr>
<tr>
<td>HMM</td>
<td>words</td>
<td>tags</td>
</tr>
<tr>
<td>PCFG</td>
<td>words</td>
<td>tree</td>
</tr>
</tbody>
</table>
Generative models have a “generative story”

- a probabilistic process that describes how the data were created
  - Multiplying probabilities of each step gives us $P(\vec{x}, \vec{y})$.

- **Naive Bayes:** For each item $i$ to be classified, (e.g., document)
  - Generate its class $c_i$ (e.g., SPORT)
  - Generate its features $f_{i1} \ldots f_{in}$ conditioned on $c_i$ (e.g., ball, goal, Tuesday)
Generative models have a “generative story”

• a probabilistic process that describes how the data were created
  – Multiplying probabilities of each step gives us $P(\vec{x}, \vec{y})$.

• Naive Bayes: For each item $i$ to be classified, (e.g., document)
  – Generate its class $c_i$ (e.g., SPORT)
  – Generate its features $f_{i1} \ldots f_{in}$ conditioned on $c_i$ (e.g., ball, goal, Tuesday)

Result:

$$P(\vec{c}, \vec{f}) = \prod_i \left[ P(c_i) \prod_j P(f_{ij}|c_i) \right]$$
Other generative stories

• **HMM:** For each position $i$ in sentence,
  
  – Generate its tag $t_i$ conditioned on previous tag $t_{i-1}$
  – Generate its word $w_i$ conditioned on $t_i$

• **PCFG:**
  
  – Starting from $S$ node, recursively generate children for each phrasal category $c_i$ conditioned on $c_i$, until all unexpanded nodes are pre-terminals (tags).
  – For each pre-terminal $t_i$, generate a word $w_i$ conditioned on $t_i$. 
Inference in generative models

• At test time, given only $\vec{x}$, infer $\vec{y}$ using Bayes’ rule:

$$P(\vec{y}|\vec{x}) = \frac{P(\vec{x}|\vec{y})P(\vec{y})}{P(\vec{x})}$$

• So, notice we actually model $P(\vec{x}, \vec{y})$ as $P(\vec{x}|\vec{y})P(\vec{y})$.
  - You can confirm this for each of the previous models.
Discriminative probabilistic models

- Model $P(\vec{y}|\vec{x})$ directly
  - No model of $P(\vec{x}, \vec{y})$
  - No generative story
  - No Bayes’ rule

- One big advantage: we can use arbitrary features and don’t have to make strong independence assumptions.

- But: unlike generative models, we can’t get $P(\vec{x}) = \sum_{\vec{y}} P(\vec{x}, \vec{y})$. 
Discriminative models more broadly

- Trained to **discriminate** right v. wrong value of $\vec{y}$, given input $\vec{x}$.

- Need not be probabilistic.

- Examples: support vector machines, (some) neural networks, decision trees, nearest neighbor methods.

- Here, we consider only multinomial logistic regression models, which are probabilistic.
  
  - *multinomial* means more than two possible classes
  - otherwise (or if lazy) just **logistic regression**
  - In NLP, also known as **Maximum Entropy** (or **MaxEnt**) models.
Example task: word sense disambiguation

Remember, logistic regression can be used for any classification task. The following slides use an example from lexical semantics:

- Given a word with different meanings (senses), can we classify which sense is intended?

I visited the Ford **plant** yesterday.
The farmers **plant** soybeans in spring.
This **plant** produced three kilos of berries.
WSD as example classification task

• disambiguate three senses of the target word plant
  – $\vec{x}$ are the words and POS tags in the document the target word occurs in
  – $y$ is the latent sense. Assume three possibilities:

\[
\begin{array}{ll}
y = & \text{sense} \\
1 & \text{Noun: a member of the plant kingdom} \\
2 & \text{Verb: to place in the ground} \\
3 & \text{Noun: a factory}
\end{array}
\]
Defining a MaxEnt model

• Define features \( f_i(\vec{x}, y) \) that depend on both observed and latent variables.

\[
\begin{align*}
  f_1 &: \text{POS(tgt)} = \text{NN} & y = 1 \\
  f_2 &: \text{POS(tgt)} = \text{NN} & y = 2 \\
  f_3 &: \text{preceding_word(tgt)} = \text{‘chemical’} & y = 3 \\
  f_4 &: \text{doc_contains(‘animal’)} & y = 1
\end{align*}
\]

where \( \text{tgt} \) is the target word

• Each feature \( f_i \) has a real-valued weight \( w_i \) (learned in training).

• \( P(y|\vec{x}) \) is a monotonic function of \( \sum_i w_i f_i(\vec{x}, y) \): they go up/down together.
Classification with MaxEnt

• Choose the class that has highest probability according to

\[ P(y | \vec{x}) = \frac{1}{Z} \exp \left( \sum_i w_i f_i(\vec{x}, y) \right) \]

where

− \( \exp(x) = e^x \) (the monotonic function)
− \( \sum_i w_i f_i \) is the *dot product* of \( \vec{w} \) and \( \vec{f} \), also written \( \vec{w} \cdot \vec{f} \).
− The normalization constant \( Z = \sum_{y'} \exp(\sum_i w_i f_i(\vec{x}, y')) \)
Which features are active?

- Example doc:
  
  
  \[ \ldots \text{ animal/NN } \ldots \text{ chemical/JJ plant/NN } \ldots \]

  
  
  \[
P(y = 1|\vec{x}) \quad \text{will have} \quad f_1, f_4 = 1 \quad \text{and} \quad f_2, f_3 = 0 \\
P(y = 2|\vec{x}) \quad f_2 = 1 \quad f_1, f_3, f_4 = 0 \\
P(y = 3|\vec{x}) \quad f_3 = 1 \quad f_1, f_2, f_4 = 0
\]

- Notice that zero-valued features have no effect on the final probability

- Other features will be multiplied by their weights, summed, then exp.
Feature weights: intuition

- $P(y|\bar{x})$ increases when $\sum_i w_i f_i(\bar{x}, y)$ increases.

- Using the same binary features of the target word $tgt$:
  
  $f_1$: $\text{POS}(tgt) = \text{NN} \& y = 1$
  $f_2$: $\text{POS}(tgt) = \text{NN} \& y = 2$
  $f_3$: $\text{preceding\_word}(tgt) = \text{'chemical'} \& y = 3$
  $f_4$: $\text{doc\_contains}('\text{animal}') \& y = 1$

- For senses \{1: member of plant kingdom; 2: put in ground; 3: factory\},
  which weights are likely to be positive? Negative?
Feature templates

• In practice, features are usually defined using templates:

\[
\begin{align*}
\text{POS}(\text{tgt}) &= t \text{ & } y \\
\text{preceding\_word}(\text{tgt}) &= w \text{ & } y \\
\text{doc\_contains}(w) &\text{ & } y
\end{align*}
\]

– instantiate with all possible POSs \( t \) or words \( w \) and classes \( y \)
– usually filter out features occurring very few times
– templates can also define real-valued or integer-valued features

• NLP tasks often have a few templates, but 1000s or 10000s of features
### Features for dependency parsing

Extract various combinations of words/tags from stack/input:

<table>
<thead>
<tr>
<th>Source</th>
<th>Feature templates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One word</strong></td>
<td></td>
</tr>
<tr>
<td>$s_1.w$</td>
<td>$s_1.t$</td>
</tr>
<tr>
<td>$s_2.w$</td>
<td>$s_2.t$</td>
</tr>
<tr>
<td>$b_1.w$</td>
<td>$b_1.w$</td>
</tr>
</tbody>
</table>

| **Two word** |                          |
| $s_1.w \circ s_2.w$ | $s_1.t \circ s_2.t$ |
| $s_1.t \circ s_2.wt$ | $s_1.w \circ s_2.w \circ s_2.t$ |
| $s_1.w \circ s_1.t \circ s_2.t$ | $s_1.w \circ s_1.t$ |

**Figure 14.9** Standard feature templates for training transition-based dependency parsers. In the template specifications $s_n$ refers to a location on the stack, $b_n$ refers to a location in the word buffer, $w$ refers to the wordform of the input, and $t$ refers to the part of speech of the input.
MaxEnt for n-best re-ranking

- So far, we’ve used logistic regression for classification.
  - Fixed set of classes, same for all inputs.

- Word sense disambiguation:
<table>
<thead>
<tr>
<th>Input</th>
<th>Possible outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>word in doc1</td>
<td>sense 1, sense 2, sense 3</td>
</tr>
<tr>
<td>word in doc2</td>
<td>sense 1, sense 2, sense 3</td>
</tr>
</tbody>
</table>

- Dependency parsing:
<table>
<thead>
<tr>
<th>Input</th>
<th>Possible outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>parser config1</td>
<td>action 1, . . . action $n$</td>
</tr>
<tr>
<td>parser config2</td>
<td>action 1, . . . action $n$</td>
</tr>
</tbody>
</table>
MaxEnt for n-best re-ranking

- In NLP, we often use it for picking the best of $n$ options, where the options depend on the input. For example, with $n = 2$:

  - Input: healthy dogs and cats
  - Possible outputs:

```plaintext
NP
  JJ  NP
  healthy NP
      NP  CC  NP
      dogs and cats
```

```
NP
  NP
  healthy NP
      CC  NP
      and cats
```

```
NP
  JJ  NP
  healthy dogs
```

```
NP
  CC  NP
  and cats
```
MaxEnt for n-best re-ranking

- In NLP, we often use it for picking the best of \( n \) options, where the options depend on the input. For example, with \( n = 2 \):
  - Input: ate pizza with cheese
  - Possible outputs:

MaxEnt for n-best re-ranking

• This scenario is why we defined our features as a function of both inputs and outputs: because the outputs may not be pre-defined.

• Typical usage: use a generative model to produce the options, then re-rank using a discriminative model.
  - Generative models typically faster to train and run, but can’t use arbitrary features.
  - In NLP, MaxEnt models may have so many features that extracting them from each example can be time-consuming, and training is even worse (see next lecture).
MaxEnt for constituency parsing

• Now we have $y =$ parse tree, $x =$ sentence.

• Features can mirror parent-annotated/lexicalized PCFG:
  – counts of each CFG rule used in $y$
  – pairs of words in head-head dependency relations in $y$
  – each word in $x$ with its parent and grandparent categories in $y$.

• Note these are no longer binary features.
Global features

- Features can also capture global structure. E.g., from Charniak and Johnson (2005):
  - length difference of coordinated conjuncts

```
NP
   JJ
  healthy
  NP
    NP
      dogs
    CC
    and
  NP
      cats

vs

NP
   NP
      NP
        JJ
      healthy
      CC
      and
    NP
      dogs
  NP
    cats
```
Global features

- Features can also capture global structure. E.g., from Charniak and Johnson (2005):
  - cat=c & len=ℓ & end_sent & before_punc
  - i.e., are heavy (long) noun phrases at the end of the sentence?
    
    I gave him the book about NLP
    
    vs
    
    I gave the book about NLP to him
Features for parsing

- Altogether, Charniak and Johnson (2005) use 13 feature templates
  - with a total of 1,148,697 features
  - and that is after removing features occurring less than five times

- One important feature not mentioned earlier: the log prob of the parse under the generative model!

- So, how does it do?
Parser performance

- $F_1$-measure (from precision/recall on constituents) on WSJ test:
  
  standard PCFG $\sim 80\%$ \footnote{Figure from (Charniak, 1996): assumes POS tags as input}
  lexicalized PCFG (Charniak, 2000) 89.7%
  re-ranked LPCFG (Charniak and Johnson, 2005) 91.0%
Parser performance

- $F_1$-measure (from precision/recall on constituents) on WSJ test:

  - standard PCFG \( \sim 80\% \)
  - lexicalized PCFG (Charniak, 2000) 89.7%
  - re-ranked LPCFG (Charniak and Johnson, 2005) 91.0%

- Best WSJ parser is now 93.8%, combining NNets and ideas from parsing, language modelling (Choe and Charniak, 2016)

- But as discussed earlier, other languages/domains are still much worse.
Summary of model types

• Probabilistic generative models:
  – model $P(x, y)$ by assuming a generative process
  – easy to train but independence assumptions can be problematic

• Probabilistic discriminative models:
  – model $P(y|x)$ only, have no generative process
  – can use arbitrary local/global features, including correlated ones
  – much slower to train; so may use $n$-best list

• Both kinds of model are used for a variety of NLP tasks.
  – Classification, parsing (discussed here)
  – Sequence models: e.g., taggers that use spelling features.
References


