Dependency parsing and logistic regression

Shay Cohen
(based on slides by Sharon Goldwater)

21 October 2019

Last class

Dependency parsing:
- a fully lexicalized formalism; tree edges connect words in the sentence based on head-dependent relationships.
- a better fit than constituency grammar for languages with free word order; but has weaknesses (e.g., conjunction).
- Gaining popularity because of move towards multilingual NLP.

Today’s lecture

- How do we evaluate dependency parsers?
- Discriminative versus generative models
- How do we build a probabilistic model for dependency parsing?

Example

Parsing Kim saw Sandy:

<table>
<thead>
<tr>
<th>Step</th>
<th>bot. Stack</th>
<th>Word List</th>
<th>Action</th>
<th>Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[root]</td>
<td>[Kim,saw, Sandy]</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[root, Kim]</td>
<td>[saw, Sandy]</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[root, Kim, saw]</td>
<td>[Sandy]</td>
<td>LeftArc</td>
<td>saw → Sandy</td>
</tr>
<tr>
<td>3</td>
<td>[root, saw]</td>
<td>[Sandy]</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[root, saw, Sandy]</td>
<td>[]</td>
<td>RightArc</td>
<td>saw → Sandy</td>
</tr>
<tr>
<td>5</td>
<td>[root, saw]</td>
<td>[]</td>
<td>RightArc</td>
<td>root → saw</td>
</tr>
<tr>
<td>6</td>
<td>[root]</td>
<td>[]</td>
<td>(done)</td>
<td></td>
</tr>
</tbody>
</table>

- Here, top two words on stack are also always adjacent in sentence. Not true in general! (See longer example in JM3.)
Labelled dependency parsing

- These parsing actions produce **unlabelled** dependencies (left).
- For **labelled** dependencies (right), just use more actions:
  - LeftArc(NSUBJ), RightArc(NSUBJ), LeftArc(DOBJ), …

\[\text{Kim saw Sandy} \quad \text{NSUBJ} \quad \text{DOBJ} \quad \text{ROOT}\]

Differences to constituency parsing

- Shift-reduce parser for CFG: not all sequences of actions lead to valid parses. Choose incorrect action → may need to backtrack.
- Here, all valid action sequences lead to valid parses.
  - Invalid actions: can’t apply LeftArc with root as dependent; can’t apply RightArc with root as head unless input is empty.
  - Other actions may lead to *incorrect* parses, but still *valid*.
- So, parser doesn’t backtrack. Instead, tries to greedily predict the correct action at each step.
  - Therefore, dependency parsers can be very fast (linear time).
  - But need a good way to predict correct actions (coming up).

Notions of validity

- In constituency parsing, valid parse = grammatical parse.
  - That is, we first define a grammar, then use it for parsing.
- In dependency parsing, we don’t normally define a grammar. Valid parses are those with the properties mentioned earlier:
  - A single distinguished root word.
  - All other words have exactly one incoming edge.
  - A unique path from the root to each other word.

Summary: Transition-based Parsing

- **arc-standard** approach is based on simple shift-reduce idea.
- Can do labelled or unlabelled parsing, but need to train a **classifier** to predict next action, as we’ll see.
- Greedy algorithm means time complexity is linear in sentence length.
- Only finds **projective** trees (without special extensions)
- Pioneering system: Nivre’s **MaltParser**.
Alternative: Graph-based Parsing

- Global algorithm: From the fully connected directed graph of all possible edges, choose the best ones that form a tree.
- Edge-factored models: Classifier assigns a nonnegative score to each possible edge; maximum spanning tree algorithm finds the spanning tree with highest total score in $O(n^2)$ time.
- Pioneering work: McDonald’s MSTParser
- Can be formulated as constraint-satisfaction with integer linear programming (Martins’s TurboParser).
- Details in JM3, Ch 14.5 (optional).

Graph-based vs. Transition-based vs. Conversion-based

- TB: Features in scoring function can look at any part of the stack; no optimality guarantees for search; linear-time; (classically) projective only.
- GB: Features in scoring function limited by factorization; optimal search within that model; quadratic-time; no projectivity constraint.
- CB: In terms of accuracy, sometimes best to first constituency-parse, then convert to dependencies (e.g., Stanford Parser). Slower than direct methods.

Choosing a Parser: Criteria

- Target representation: constituency or dependency?
- Efficiency? In practice, both runtime and memory use.
- Incrementality: parse the whole sentence at once, or obtain partial left-to-right analyses/expectations?
- Accuracy?

Probabilistic transition-based dep’y parsing

At each step in parsing we have:

- Current configuration: consisting of the stack state, input buffer, and dependency relations found so far.
- Possible actions: e.g., Shift, LeftArc, RightArc.

Probabilistic parser assumes we also have a model that tells us $P(\text{action}|\text{configuration})$. Then,

- Choosing the most probable action at each step (greedy parsing) produces a parse in linear time.
- But it might not be the best one: choices made early could lead to a worse overall parse.
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:
- Probabilistic parser assumes we also have a model 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.

Training data

Our goal is:
- Given (features from) the current configuration, predict the next action.
Our corpus contains annotated sentences such as:

```
A hearing on the issue is scheduled today
```

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 rules to convert each annotated sentence to a unique sequence of (configuration, action) pairs.\(^1\)

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

---
\(^1\)This algorithm is called the training oracle. An oracle is a fortune-teller, and in NLP it refers to an algorithm that always provides the correct answer. Oracles can also be useful for evaluating certain aspects of NLP systems, and we may say a bit more about them later.

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 \( \hat{y} \), given input \( \hat{x} \).
- 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**
  - \( \hat{x} \) are the words and POS tags in the document the target word occurs in
  - \( y \) is the latent sense. Assume three possibilities:
    
    \[
    y = \text{sense} \\
    1 \quad \text{Noun: a member of the plant kingdom} \\
    2 \quad \text{Verb: to place in the ground} \\
    3 \quad \text{Noun: a factory}
    \]
  - We want to build a model of \( P(y|x) \).

Defining a MaxEnt model: intuition

- Start by defining a set of **features** that we think are likely to help discriminate the classes. E.g.:
  - the POS of the target word
  - the words immediately preceding and following it
  - other words that occur in the document
- During training, the model will learn how much each feature contributes to the final decision.
Defining a MaxEnt model

- Features $f_i(\mathbf{x}, y)$ depend on both observed and latent variables. E.g., if $\text{tgt}$ is the target word:
  - $f_1$: \(\text{POS(tgt)} = \text{NN} \& y = 1\)
  - $f_2$: \(\text{POS(tgt)} = \text{NN} \& y = 2\)
  - $f_3$: \(\text{preceding_word(tgt)} = \text{‘chemical’} \& y = 3\)
  - $f_4$: \(\text{doc\_contains}('\text{animal}') \& y = 1\)

- Each feature $f_i$ has a real-valued weight $w_i$ (learned in training).
- $P(y|\mathbf{x})$ is a monotonic function of $\mathbf{w} \cdot f$ (that is, $\sum_i w_i f_i(\mathbf{x}, y)$).

Example of features and weights

- Let’s look at just two features from the plant disambiguation example:
  - $f_1$: \(\text{POS(tgt)} = \text{NN} \& y = 1\)
  - $f_2$: \(\text{POS(tgt)} = \text{NN} \& y = 2\)
- Our classes are:
  \{1: member of plant kingdom; 2: put in ground; 3: factory\}
- Our example doc ($\mathbf{x}$):
  \[\ldots \text{animal/NN} \ldots \text{chemical/JJ} \text{plant/NN} \ldots\]
Two cases to consider

- Computing $P(y = 1|\vec{x})$:
  - Here, $f_1 = 1$ and $f_2 = 0$.
  - We would expect the probability to be relatively high.
  - Can be achieved by having a positive value for $w_1$.
  - Since $f_2 = 0$, its weight has no effect on the final probability.

- Computing $P(y = 2|\vec{x})$:
  - Here, $f_1 = 0$ and $f_2 = 1$.
  - We would expect the probability to be close to zero, because sense 2 is a verb sense, and here we have a noun.
  - Can be achieved by having a large negative value for $w_2$.
  - By doing so, $f_2$ says: "If I am active, do not choose sense 2!".

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:
  `[... animal/NN ... chemical/JJ plant/NN ...]`

  \[
P(y = 1|\vec{x}) \quad \text{will have} \quad f_1, f_4 = 1 \quad \text{and} \quad f_2, f_3 = 0
\]
  \[
P(y = 2|\vec{x}) \quad f_2 = 1 \quad f_1, f_2, f_4 = 0
\]
  \[
P(y = 3|\vec{x}) \quad f_3 = 1 \quad f_1, f_2, f_4 = 0
\]
  - Notice that zero-valued features have no effect on the final probability
  - Other features will be multiplied by their weights, summed, then exp.
Feature templates

▶ In practice, features are usually defined using templates
  \[
  \text{POS}(\text{tgt})=t \ \& \ y \\
  \text{preceding}\_\text{word}(\text{tgt})=w \ \& \ y \\
  \text{doc}\_\text{contains}(w) \ \& \ y
  \]
  ▶ 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

▶ We want the model to tell us \( P(\text{action}|\text{configuration}) \).
▶ So \( y \) is the action, and \( \vec{x} \) is the configuration.
▶ Features are various combinations of words/tags from stack/input:

<table>
<thead>
<tr>
<th>Source</th>
<th>Feature templates</th>
</tr>
</thead>
<tbody>
<tr>
<td>One word</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>( s_1,, w )</td>
<td>( s_1,, w t )</td>
</tr>
<tr>
<td>Two word</td>
<td></td>
</tr>
<tr>
<td>( s_1,, w \circ , s_2,, w )</td>
<td>( s_1,, u \circ , s_2,, t )</td>
</tr>
<tr>
<td>( s_1,, t \circ , s_2,, t )</td>
<td>( s_1,, w \circ , s_2,, t )</td>
</tr>
<tr>
<td>( s_1,, w \circ , s_1,, t \circ , s_2,, t )</td>
<td>( s_1,, w \circ , s_1,, t \circ , s_2,, t )</td>
</tr>
<tr>
<td>( s_1,, w \circ , s_1,, t \circ , s_2,, t )</td>
<td>( s_1,, w \circ , s_1,, t \circ , s_2,, t )</td>
</tr>
</tbody>
</table>

Summary

We’ve discussed
▶ Beam search.
▶ Evaluation for probabilistic dependency parsing.
▶ The logistic regression classifier.