Recap: dependency syntax

- 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

- What’s the relationship between constituency (phrase structure) trees and dependency trees?
- How do we parse sentences to dependency trees?
- How do we evaluate the result?

How to get a dependency treebank?

Two options:
1. Annotate dependencies directly (e.g., Prague Dependency Treebank, many others).
2. Convert phrase structure annotations to dependencies (if we already have a phrase structure treebank.)

Conversion procedure is shown in slides at end of previous lecture; assumes we have lexical heads for each phrasal category.
Constituency tree with lexical heads
(Also used for lexicalized PCFG parsing.)

```
S-saw
  NP-kids
    kids
  VP-saw
    VP-saw
      NP-birds
        birds
      PP-binoculars
        P-with
          NP-binoculars
            P-with
              NP-binoculars
                NP-binoculars
```

- But how can we find each phrase’s head in the first place?

Head Rules

Standard solution: use head rules: for every non-unary (P)CFG production, designate one RHS nonterminal as containing the head.

```
S → NP VP, VP → VP PP, PP → P NP (content head), etc.
```

Heuristics to scale this to large grammars: e.g., within an NP, last immediate N child is the head.

```
S
  NP
    kids
  VP
    VP-saw
      NP-birds
        birds
      PP-binoculars
        P-with
          NP-binoculars
            NP-binoculars
```

Then, propagate heads up the tree:

```
S
  NP-kids
    kids
  VP
    VP-saw
      NP-birds
        birds
      PP-binoculars
        P-with
          NP-binoculars
            NP-binoculars
```

Sharon Goldwater logistic regression 4
Then, propagate heads up the tree:

**Head Rules**

S

NP-kids

kids

VP

S

NP-kids

kids

VP

VP-saw

PP-binoculars

V-saw

saw

NP-birds

birds

P-with

with

NP-binoculars

binoculars

Then, propagate heads up the tree:

**Head Rules**

S

NP-kids

kids

VP-saw

PP-binoculars

V-saw

saw

NP-birds

birds

P-with

with

NP-binoculars

binoculars

Projectivity

If we convert constituency parses to dependencies, all the resulting trees will be **projective**.

- Every subtree (node and all its descendants) occupies a contiguous span of the sentence.
- = the parse can be drawn over the sentence w/ no crossing edges.
Nonprojectivity

But some sentences are nonprojective.

\[ A \quad \text{hearing} \quad \text{is} \quad \text{scheduled} \quad \text{on} \quad \text{the} \quad \text{issue} \quad \text{today} \]

- We’ll only get these annotations right if we directly annotate the sentences (or correct the converted parses).
- Nonprojectivity is rare in English, but common in many languages.
- Nonprojectivity presents problems for parsing algorithms.

Recall: shift-reduce parser with CFG

Consider a very simple example:

- Grammar contains only these rules:
  \[ S \rightarrow \text{NP} \quad \text{VP} \quad \text{VP} \rightarrow \text{V} \quad \text{NN} \rightarrow \text{bit} \quad \text{V} \rightarrow \text{bit} \]
  \[ \text{NP} \rightarrow \text{DT} \quad \text{NN} \quad \text{DT} \rightarrow \text{the} \quad \text{NN} \rightarrow \text{dog} \quad \text{V} \rightarrow \text{dog} \]
- The input sequence is the dog bit

Dependency Parsing

Some of the algorithms you have seen for PCFGs can be adapted to dependency parsing.

- **CKY** can be adapted, though efficiency is a concern: obvious approach is \( O(Gn^5) \); Eisner algorithm brings it down to \( O(Gn^3) \)
- **Shift-reduce**: more efficient, doesn’t even require a grammar!
Recall: shift-reduce parser with CFG

- Operations:
  - Reduce (R)
  - Shift (S)
  - Backtrack to step \( n \) (\( B_n \))
- Note that at 9 and 11 we skipped over backtracking to 7 and 5 respectively as there were actually no choices to be made at those points.

<table>
<thead>
<tr>
<th>Step</th>
<th>Op.</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>root</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>the</td>
<td>dog bit</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>DT</td>
<td>dog bit</td>
</tr>
<tr>
<td>3</td>
<td>S</td>
<td>DT dog</td>
<td>bit</td>
</tr>
<tr>
<td>4</td>
<td>R</td>
<td>DT V</td>
<td>bit</td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td>DT VP</td>
<td>bit</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>DT VP bit</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>R</td>
<td>DT VP V</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>R</td>
<td>DT VP VP</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>B6</td>
<td>DT VP bit</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>R</td>
<td>DT VP V</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>B4</td>
<td>DT V</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>S</td>
<td>DT VP NN</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>R</td>
<td>DT V</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>R</td>
<td>DT V VP</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>B3</td>
<td>DT dog</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>R</td>
<td>DT NN</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>R</td>
<td>NP</td>
<td></td>
</tr>
</tbody>
</table>

Sharon Goldwater logistic regression 16

Transition-based Dependency Parsing

The **arc-standard** approach parses input sentence \( w_1 \ldots w_N \) using two types of reduce actions (three actions altogether):

- **Shift**: Read next word \( w_i \) from input and push onto the stack.
- **LeftArc**: Assign head-dependent relation \( s_2 \leftarrow s_1 \); pop \( s_2 \)
- **RightArc**: Assign head-dependent relation \( s_2 \rightarrow s_1 \); pop \( s_1 \)

where \( s_1 \) and \( s_2 \) are the top and second item on the stack, respectively. (So, \( s_2 \) preceded \( s_1 \) in the input sentence.)

- Unlike CFG parsing, reduce actions don’t rely on a grammar!

Example

Parsing *Kim saw Sandy*:

<table>
<thead>
<tr>
<th>Step</th>
<th>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>Kim→saw</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></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), . . .

```
  ROOT
  /    \
Kim   Sandy
  |    /  \
    NSUBJ  DOBJ
```

Sharon Goldwater logistic regression 17
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?
- Retrainable system?
- 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:

\[ \text{A hearing on the issue is scheduled today} \]

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 $f_{i1} \cdots f_{in}$ 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 $f_{i1} \cdots f_{in}$ conditioned on $c_i$ (e.g., ball, goal, Tuesday)

Result:

$P(\vec{c}, \vec{f}) = \prod_i P(c_i) \prod_j P(f_{ij}|c_i)$
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.
Summary

We’ve discussed

• Dependency annotations, heads, and constituency \(\rightarrow\) dependency conversion.

• Transition-based dependency parsing.

• Training data and evaluation for probabilistic dependency parsing.

On Thursday, details of the model for \(P(\text{action}|\text{configuration})\).

Announcements

• EdIntelligence is hosting mini-EMNLP on Wed, 5-7pm
  – See info on Piazza or EdIntelligence Facebook page.

• Tomorrow’s lecture:
  1. Information about the exam and how to prepare for it.
  2. Mid-semester course questionnaire: please bring a device you can use to fill it in online.