Dependency Parsing

CMSC 473/673 Spring 2017
Bryan Wilkinson
Dependency Parsing

- Dependency Parses are represent the structure of a sentence as a set of binary relations directly between words
  - Because it is a binary relation, it forms a graph, where the words are the nodes and the relations are the edges
- There is no longer a concept of a constituent or phrase
Why Dependency Parsing

- Dependency parsing is usually much faster
- Dependency relations make good features for downstream tasks
  - Word embeddings using dependency contexts (we’ll talk about this in a few weeks)
- The binary relationships produced by dependency parsing often make excellent templates for information extraction, and more specifically relationship extraction

![Dependency Parsing Diagram]
One assumption we will make today is that the dependency parses are projective.

A projective parse is one where there is no crossing edges:
- This assumption makes processing a lot easier.
- It won’t mess up too many English sentences, but is a bigger problem in other languages.
Stanford Dependencies

- Consists of 56 binary relations describing dependencies
- Examples:
  - Adverbial modifier - advmod( often, less)
  - Adjectival modifier - amod(wall, blue)
  - Appositional modifier - appos(Sam, brother)
    - Sam, my brother, arrived
  - Coordination - cc(big, and)
    - Bill is big and honest
  - Conjunction - conj(big, honest)
    - Bill is big and honest
  - Direct Object - dobj(gave, raise)
  - Indirect Object - iobj(gave, me)
  - Nominal Subject - nsubj(taught, Bryan)
    - Bryan taught the class

2008, Marie-Catherine de Marneffe and Christopher D. Manning. [Stanford typed dependencies manual](#)
Universal Dependencies

- Presented in 2016
- Culmination of efforts to combine a lot of different annotation schemas
  - Core is based on Stanford Dependencies
- Because dependencies don’t rely on word orders, it is much easier to have a universal set of them that apply to almost every situation in every language
  - It is expected that specific languages may need specific features, and this is accounted for by allowed for specification of subtypes of a relation already in UD.
- The project has released dependency treebanks for about 47 languages.
  - They are always adding new ones!
  - See the current list at http://universaldependencies.org/
Recent work has started using enhanced dependencies, which do post processing of a dependency graph to add additional relations.

- The original dependencies are never deleted.

These new dependencies are meant to directly connect words that are connected through things like conjunctions and prepositions.

- Useful for information extraction.

2016, Sebastian Schuster and Christopher D. Manning. *Enhanced English Universal Dependencies*
Like constituency parsing, dependency parsing is usually done in a supervised manner.

The training data for dependency parsers in English usually comes from the Penn Treebank, still

- A dependency version of this exists, based on rules to transform the constituents into binary relations.

For other languages, a treebank constructed specifically for dependency parsing is often used.
● Dependency Treebanks are often stored in CONLL format.
  ○ CONLL is the Conference on Computational Natural Language Learning
  ○ For many years they have run shared tasks involving dependency parsing, and thus CONLL has become the default standard for dependency treebanks

● CONLL format actually is a general term for a group of file formats that vary in small ways
  ○ CONLL-X and CONLL-U are most common now

● The general premise of a CONLL file is that a sentence is written as one word per line
  ○ After the word, a series of columns encode several attributes about the sentence, such as lemma, part of speech, and head word, which is the word the current word “points” to
<table>
<thead>
<tr>
<th></th>
<th>Dependency</th>
<th>dependency</th>
<th></th>
<th></th>
<th></th>
<th>compound</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>parses</td>
<td>parse</td>
<td>NNS</td>
<td>O</td>
<td>6</td>
<td>nsubj</td>
</tr>
<tr>
<td>3</td>
<td>are</td>
<td>be</td>
<td>VBP</td>
<td>O</td>
<td>6</td>
<td>cop</td>
</tr>
<tr>
<td>4</td>
<td>very</td>
<td>very</td>
<td>RB</td>
<td>O</td>
<td>5</td>
<td>advmod</td>
</tr>
<tr>
<td>5</td>
<td>useful</td>
<td>useful</td>
<td>JJ</td>
<td>O</td>
<td>6</td>
<td>amod</td>
</tr>
<tr>
<td>6</td>
<td>tools</td>
<td>tool</td>
<td>NNS</td>
<td>0</td>
<td>0</td>
<td>ROOT</td>
</tr>
<tr>
<td>7</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>
Ways to do Dependency Parsing

- Transition-Based Methods
  - Today's Lecture
- Graph-Based
  - Next Lecture
- Mixed
Quick Review / Intro

Shift-reduce parsers are a bottom-up parser that is commonly used to parse programming languages.

The parser consists of:
- A parse stack and an input
  - The stack is initialized with a start symbol, such as $.
  - Depending on the end of the stack and the current front of the input, an action is taken, based on a table that specifies actions for a given parser.
  - The version taught in CMSC 331 commonly uses states and a goto table as well.

There are two actions we can take:
- Shift the next input symbol onto the stack.
- Reduce the symbol on the top of the stack and the next input symbol to another symbol.
  - This is equivalent to applying a rule in the grammar.
Shift-Reduce Practice

- Using the grammar:
  1. $E \rightarrow E + T$
  2. $E \rightarrow T$
  3. $T \rightarrow T \ast F$
  4. $T \rightarrow F$
  5. $F \rightarrow ( E )$
  6. $F \rightarrow \text{id}$

- And the table to the right,
  Parse the expression: $( \text{id} + \text{id} )$
Arc-standard Parsing

- The most common style of shift-reduce parsing
- Still uses a stack and an input, and words are still shifted onto the stack
- Rather than a single reduce option, there are now two
  - Reduce is usually defined as using the top 2 symbols from the stack rather than the top of the stack and the input symbol
  - The two reduce operations of LEFTARC and RIGHTARC, which correspond to the direction of the arrow we are drawing between the top two words on the stack
  - Rather than reduce to a simpler rule, we remove one of the words
    - Remove the second from the top word for the stack for LEFTARC
    - Remove the top word for RIGHTARC
- A lookup table for this type of thing would be huge
  - An NxN matrix
- Instead we use something called an oracle that just “magically” tells us what to do
  - SPOILER ALERT: The oracle is just a classifier
Basic Transition-Based Parser Pseudocode

- The oracle bases its decision off the stack, the remaining input, as well as the dependency relations discovered so far.
- The pseudocode is very simple because the oracle is the main trick to this.
- Pseudocode:

```
#Initialize the state:
state = {stack:[ROOT], input:[words], deps:[]}

#Iterate through stack and input until everything has a dependency
while state != {stack:[ROOT], input:[], deps:[...]}:
    action = ORACLE(state)
    state = APPLY(action, state)
return state[deps]
```
The oracle uses the parser to learn what to do
  ○ Don’t worry if this is confusing right now

Let’s treat the oracle as a magic black box for now so we can get familiar with the mechanisms of the shift and reduce operations

What is the dependency graph for the sentence:

**Book me the morning flight**

We will use the example from SLP3, so for now that will be our oracle.
● **Train on corpus**
  ○ Pretend to parse something we know the answer to. So we know what move to make
  ○ Use this as the label to predict
  ○ Train a classifier using a bunch of (state, move) pairs

● **To generate the action given a dependency parse, we have to make a few more constraints explicit**
  ○ LEFTARC can not be used when the second element of the stack is the ROOT
  ○ You cannot apply a RIGHTARC action if the head word still has more dependents left in the input
    ■ If this allowed we would get stuck when the next dependent came up in the input
Let \( \text{DEP} \) be list of all dependency pairs in a parse
Let \( \text{CUR} \) be the list of dependency pairs we have learned an instance for
While \( \text{DEP} \neq \text{CUR} \)
    IF \( S_1S_2 \) in \( \text{DEP} \),
        output state, LEFTARC
        do LEFTARC action
    IF \( S_2S_1 \) in \( \text{DEP} \) and all([\( R \) in \( \text{CUR} \) for \( R \) in \( \text{DEP} \) if \( R \) matches /\( S_1 \) [←→] .*/])
        output state, RIGHTARC
        do RIGHTARC action
    ELSE
        output state, SHIFT
        do SHIFT
The Oracle Classifier

- Given a set of instances that might look something like:

  STACK
  [root, canceled, flights]

  WORDS
  [to, Houston]

  DEPS
  [canceled $\rightarrow$ United, flights $\rightarrow$ morning, flights $\rightarrow$ the]

  LABEL
  SHIFT

- We could create a feature vector like

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>flights</td>
<td>canceled</td>
<td>NNS</td>
<td>VBD</td>
<td>to</td>
<td>TO</td>
<td>SHIFT</td>
</tr>
</tbody>
</table>
The Oracle Classifier

- Using that feature vector, we can train a classifier using whichever algorithm we like
  - Neural Networks have been really good at this
  - The entire architecture can be implemented in one single neural network, including the stack
- We have been talking about unlabeled dependency parsing algorithms
  - To make it labeled, just add the arc label to the classifier, so the label is now the pair (action, dependency type)
Evaluation

- Dependency parsers are labeled on their accuracy
  - Number of Correct Heads / # of Words
- There are two commonly reported versions
  - Unlabeled (UAS)
    - For each word, if the head is correct, increment correct by one
    - UAS = correct / # of words
  - Labeled (UAS)
    - For each word, if the head AND the dependency type is correct, increment correct
    - UAS = correct / # of words
- Some currently state-of-the-art scores can be seen at the [spacy.io benchmark site](https://spacy.io)