Let’s get parsing!

- SpaCy default model includes tagger, parser and entity recognizer
  - nlp = spacy.load(‘en’) tells spaCy to use "en" with ["tagger", "parser", "ner"]

- Each component processes the Doc object, then passes it on

- doc.is_parsed attribute checks whether a Doc object has been parsed

NAME	COMPONENT	CREATES	DESCRIPTION
tokenizer	Tokenizer	Doc	Segment text into tokens.
tagger	Tagger	Doc[i].tag	Assign part-of-speech tags.
parser	DependencyParser	Doc[i].head, Doc[i].dep, Doc.sents, Doc.noun_chunks	Assign dependency labels.

SpaCy First Parse.py

```python
import spacy
import pandas as pd
import itertools as it
import codecs
import sys

nlp = spacy.load('en_core_web_sm')

# To load a file, change the below path to your actual path:
#doc = unicode(open('sample_review.txt').read().decode('utf8'))

#To parse that file, replace the below line with doc = nlp(doc).
doc = nlp(u'spaCy is designed to help you do real work. It will help you build real products and gather real insights.')

print('DOCUMENT IS PARSED?')
print(doc.is_parsed) #Checks whether the file is parsed. Will throw an exception if false.

#optionally print doc tokens
#for token in doc:
    #print(token.text)

#optionally print tokens to a document
#token_list = [token.text for token in doc]
#df1 = pd.DataFrame(zip(token_list),
                   #columns=['token'])
#df1.to_html('Parsed Token List.html')
```

Noun Chunks

TEXT	ROOT.TEXT	ROOT.DEP_	ROOT.HEAD.TEXT
Autonomous cars	cars	nsubj	shift
insurance liability	liability	dobj	shift
manufacturers	manufacturers	pobj	toward

Example parse: “Autonomous cars shift insurance liability toward manufacturers.”

Text: The original noun chunk text.
Root text: The original text of the word connecting the noun chunk to the rest of the parse.
Root dep: Dependency relation connecting the root to its head.
Root head text: The text of the root token's head.

SpaCy DP Chunks.py

```python
import spacy
import pandas as pd

nlp = spacy.load('en_core_web_sm')

doc = nlp(u'spaCy is designed to help you do real work. It will help you build real products and gather real insights.')

#print noun chunks to console
print('CHUNKS')
for chunk in doc.noun_chunks:
    print(chunk.text, chunk.root.text, chunk.root.dep_,
    chunk.root.head.text)

#alternatively, print in a table, to a document
chunk_text = [chunk.text for chunk in doc.noun_chunks]
chunk_root = [chunk.root.text for chunk in doc.noun_chunks]
chunk_root_dep = [chunk.root.dep_ for chunk in doc.noun_chunks]
chunk_root_head = [chunk.root.head.text for chunk in doc.noun_chunks]

df3 = pd.DataFrame(zip(chunk_text, chunk_root, chunk_root_dep, chunk_root_head),
                   columns=['Chunk_Text', 'Chunk Root', 'Chunk Root Dep', 'Chunk Root Head'])
df3.to_html('Chunker.html')
```

Dependency Relations

TEXT	DEP	HEAD TEXT	HEAD POS	CHILDREN
Autonomous	amod	cars	NOUN	
cars	nsubj	shift	VERB	Autonomous
shift	ROOT	shift	VERB	cars, liability
insurance	compound	liability	NOUN	
liability	dobj	shift	VERB	insurance, toward
toward	prep	liability	NOUN	manufacturers
manufacturers	pobj	toward	ADP	

Text: The original token text.
Dep: The syntactic relation connecting child to head.
Head text: The original text of the token head.
Head POS: The part-of-speech tag of the token head.
Children: The immediate syntactic dependents of the token.

Example parse: "Autonomous cars shift insurance liability toward manufacturers."

SpaCy DP Dependency Relations.py

```python
import spacy
import pandas as pd

nlp = spacy.load('en_core_web_sm')

doc = nlp(u'spaCy is designed to help you do real work. It will help you build real products and gather real insights.')

print('DEPENDENCY RELATIONS')
print('Key:')
print('TEXT, DEP, HEAD TEXT, HEAD POS, CHILDREN')
for token in doc: #prints dependencies
    print(token.text, token.dep_, token.head.text, token.head.pos_,
          [child for child in token.children])

#alternatively, print in a table, to a document
token_text = [token.text for token in doc]
token_dep = [token.dep_ for token in doc]
token_head_text = [token.head.text for token in doc]
token_head_pos = [token.head.pos_ for token in doc]
token_child = ([child for child in token.children]for token in doc)

df1 = pd.DataFrame(zip(token_text, token_dep, token_head_text, token_head_pos, token_child),
                   columns=['Token Text', 'Token Dep', 'Token Head Text', 'Token Head Pos', 'Token Child'])
df1.to_html('Dependencies.html')
```

Matching Arcs of Interest

- You can iterate over the arcs by iterating over the words in the sentence.
- Best method to match an arc of interest — from below and moving back up the tree
- Matching from above and down the tree requires two iterations
- Notice the different order of results

SpaCy DP Finding Arcs of Interest.py

```python
import spacy

nlp = spacy.load('en_core_web_sm')

doc = nlp(u'spaCy is designed to help you do real work. It will help you build real products and gather real insights.')

#finding a verb with a subject from below - preferred direction
print('FINDING A VERB WITH SUBJECT FROM BELOW - GOOD')
from spacy.symbols import nsubj, VERB
verbs = set()
for possible_subject in doc:
    if possible_subject.dep == nsubj and possible_subject.head.pos == VERB:
        verbs.add(possible_subject.head)
print('VERBS:')
print(verbs)

#finding a verb with a subject from above - dispreferred direction
print('\n')
print('FINDING A VERB WITH SUBJECT FROM ABOVE - NOT GOOD')
verbs = []
for possible_verb in doc:
    if possible_verb.pos == VERB:
        for possible_subject in possible_verb.children:
            if possible_subject.dep == nsubj:
                verbs.append(possible_verb)
                break
print('VERBS')
print(verbs)
```

Working with Syntactic Phrases

- Token.lefts and Token.rights provide sequences of syntactic children before/after the token.

- Token.n_rights and Token.n_lefts give the number of left and right children.

- Token.subtree returns an ordered sequence of tokens
  - Note: guaranteed to be contiguous, as English model is projective
  - Cf. German model, which is non-projective

  - See example *#Get phrases by syntactic head*

Working with Syntactic Phrases (continued)

- Token.ancestors allows you to walk up the tree: returns a sequence of ancestor tokens such that ancestor.is_ancestor(self).

- Token.is_ancestor() checks whether the token is dominant.

  - See example *#Get phrases by syntactic head*

Working with Syntactic Phrases (continued)

- .left_edge and .right_edge* return the first/last token of the subtree.
  - See example *#Create a Span object for a syntactic phrase.*

- Note: Syntactic parse tree also calculates sentence boundaries

*.right_edge gives a token within the subtree, so if using as the end-point of a range, remember to +1

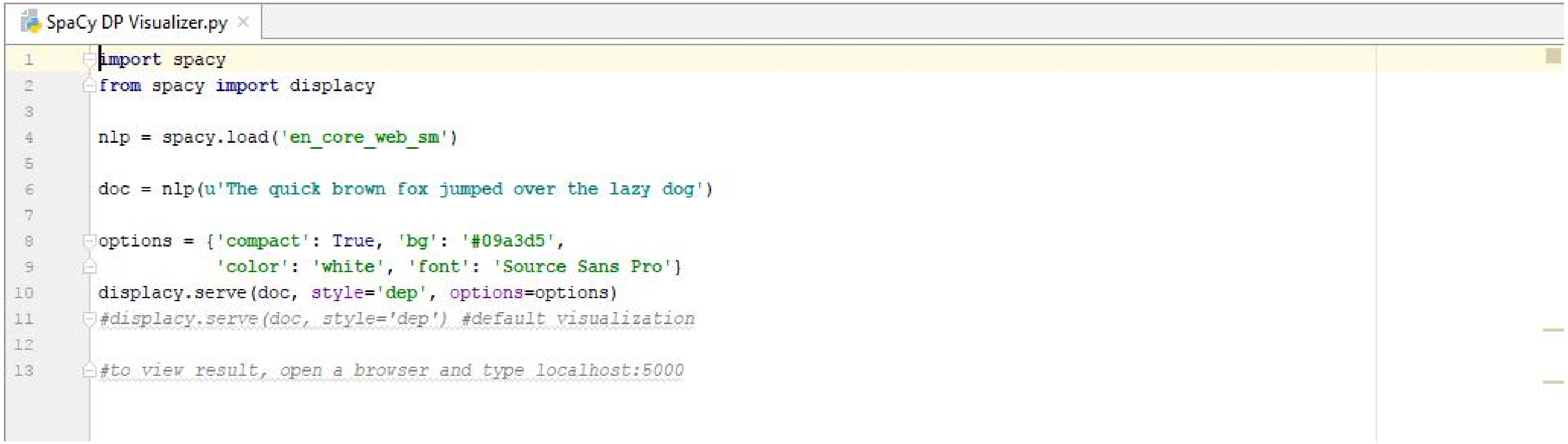
SpaCy DP Working With Phrases.py

```python
import spacy

nlp = spacy.load('en_core_web_sm')

#Get a phrase by its syntactic head
doc = nlp(u'Credit and mortgage account holders must submit their requests')
root = [token for token in doc if token.head == token][0]#finds the root token, here = submit
print('ROOT:')
print(root)
subject = list(root.lefts)[0] #provide sequences of syntactic children before the token.
for descendant in subject.subtree: #iterates through descendents
    if subject.is_ancestor(descendant):
        print('\n DESCENDENT TEXT, DESCENDENT DEP, NO. LEFT CHILDREN, NO. RIGHT CHILDREN, ANCESTORS')
    print(descendant.text, descendant.dep_, descendant.n_lefts, descendant.n_rights,
          [ancestor.text for ancestor in descendant.ancestors])

#Create a Span object for a syntactic phrase
doc = nlp(u'Credit and mortgage account holders must submit their requests')
span = doc[doc[4].left_edge.i : doc[4].right_edge.i+1]
span.merge()
for token in doc:
    print(token.text, token.pos_, token.dep_, token.head.text)
```

Visualizing Dependencies

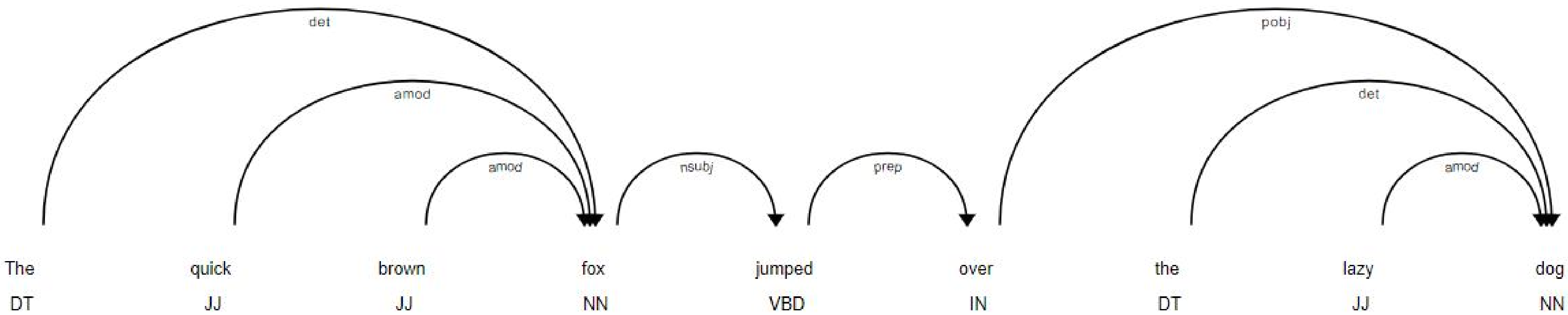
- displacy.serve runs the web server
- displacy.render generates the raw markup
  - eg. html = displacy.render([doc1, doc2], style='dep', page=True).

- Dependency Visualizer options: https://spacy.io/api/top-level#displacy_options

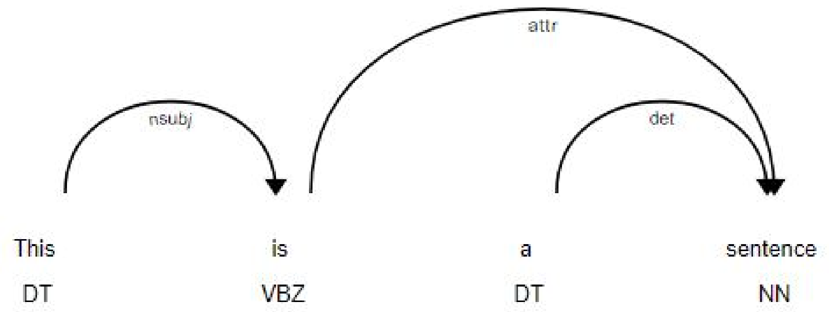


SpaCy DP Visualizer.py

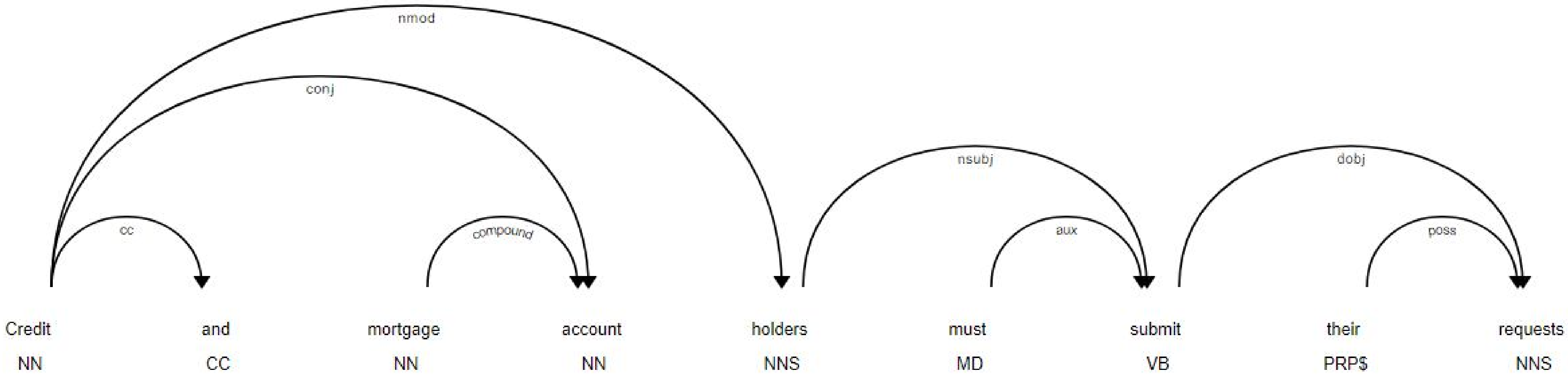
```
import spacy
from spacy import displacy

nlp = spacy.load('en_core_web_sm')

doc = nlp(u'The quick brown fox jumped over the lazy dog')

options = {'compact': True, 'bg': '#09a3d5',
           'color': 'white', 'font': 'Source Sans Pro'}
displacy.serve(doc, style='dep', options=options)
#displacy.serve(doc, style='dep') #default visualization

#to view result, open a browser and type localhost:5000
```



True Dependencies?



SpaCy website say this feature can be used for viewing dependencies, but notice these aren't true dependencies.



Parsing in SpaCy

- Development of the SpaCy parsing method:
  - (Background) Basic, transition-based dependency parsing
  - Arc-eager parsing
  - Non-monotonic parsing
  - “Unshift” method
  - SpaCy: non-monotonic actions and Unshift

Basic, transition-based parsing

stack buffer dependency arcs

Start: $\sigma = [\text{ROOT}],\ \beta = w_1, \ldots, w_n,\ A = \varnothing$

1. Shift $\quad \sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$

reduce operations

2. Left-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j, w_i)\}$
3. Right-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_i|\beta, A \cup \{r(w_i, w_j)\}$

Finish: $\beta = \varnothing$

1. Shift word from buffer onto the stack
2. Takes w*j* (right) as head to create left arc. Adds this to set of arcs
{r(w*j*, w*i*)} = {*amod*(children, happy)}. ‘happy’ depends on ‘children’
3. Takes w*i* (left) as root and adds to stack, leaves w*j* (its dependent) on buffer.
Finish: once buffer is exhausted

Problem

- Have to construct low level dependencies first, even when you can already see a higher one:

She tried to open the door in the cellar

- Inefficient! Means shifting through every word before assigning the first dependency.
- Requiremed solution: that a head can take a right dependent, before *its* dependents are found

Solution: Arc-eager Dependency Parsing

Start: $\sigma = [\text{ROOT}]$, $\beta = w_1, \ldots, w_n$, $A = \varnothing$

1. Left-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j,w_i)\}$

   Precondition: $(w_k, r', w_i) \notin A$, $w_i \neq \text{ROOT}$

2. Right-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma|w_i|w_j, \beta, A \cup \{r(w_i,w_j)\}$
3. Reduce $\quad \sigma|w_i, \beta, A \rightarrow \sigma, \beta, A$

   Precondition: $(w_k, r', w_i) \in A$

4. Shift $\quad \sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$

Finish: $\beta = \varnothing$

- Nivre (2003)
- Add preconditions to the Left-Arc and Reduce actions (thus system is monotonic)
  - candidate word can't already be a dependent (**1.**)
  - words can be left on the stack to get dependents later (**2.**)

Start: $\sigma = [\text{ROOT}], \beta = w_1, \ldots, w_n, A = \varnothing$

1. Left-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma, w_j|\beta, A \cup \{r(w_j, w_i)\}$

   Precondition: $(w_k, r', w_i) \notin A, w_i \neq \text{ROOT}$

2. Right-Arc$_r$ $\quad \sigma|w_i, w_j|\beta, A \rightarrow \sigma|w_i|w_j, \beta, A \cup \{r(w_i, w_j)\}$

3. Reduce $\quad \sigma|w_i, \beta, A \rightarrow \sigma, \beta, A$

   Precondition: $(w_k, r', w_i) \in A$

4. Shift $\quad \sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$

Finish: $\beta = \varnothing$

- Necessitates the **reduce** operation to remove words from the stack that have been assigned an arc.
  - Precondition: only permitted if word has been made a dependent

Improvement 1: Non-monotonic Parsing

- Edit preconditions, making actions more flexible
  - Left-Arc may delete existing arc
  - Reduce action may insert ‘missing’ arc

Improvement 2: “Unshift”

- Nivre & Fernandez-Gonzalez (2014): “Unshift” operation
  - Can repair false sequences
  - Designed for use only when buffer is empty but stack contains words without governors

Improvement 3: Combination

- Honnibal and Johnson (2015) combine these. Reasoning:
  - Shift and Right-arc pushes word from buffer to stack, right-arc adds an arc
  - Previously: presence or absence of arc determines whether Reduce or Left-arc is possible
  - Unnecessary when parser is trained on a dynamic oracle (See next slide)
  - Can thus use Unshift more broadly

Dynamic vs Static Oracles

- Normally train transition-based dependency parsers on static oracle
  - Predicts optimal transition sequence and gold tree
- Dynamic oracle provides set of optimal transitions for every possible parser configuration
- If gold tree isn’t reachable, will provide transitions leading to the best tree possible, also from non-optimal sequences
- Model can thus learn to recover from previous errors
- Example Goldberg and Nivre (2012): outperformed greedy parsers

Improved Transition System

- A word pushed onto stack with Shift won't have an arc
- Unshift then possible; pops word back onto buffer
  - Note: Sets a bit in a Boolean vector which will prevent a word being pushed and popped more than twice

Initial	$([\,], [1...n], \mathbf{A}(1) = 1)$		
Terminal	$([i], [\,], \mathbf{A})$		
Shift	$(\sigma, b\|\beta, \mathbf{A}, \mathbf{S}(b) = 0)$	$\Rightarrow$	$(\sigma\|b, \beta, \mathbf{A}, \mathbf{S}(b) = 1)$
Right-Arc	$(\sigma\|s, b\|\beta, \mathbf{A}, \mathbf{S})$	$\Rightarrow$	$(\sigma\|s\|b, \beta, \mathbf{A}(b) = s, \mathbf{S})$
Reduce	$(\sigma\|s, \beta, \mathbf{A}(s) \neq 0, \mathbf{S})$	$\Rightarrow$	$(\sigma, \beta, \mathbf{A}, \mathbf{S})$
Unshift	$(\sigma\|s, \beta, \mathbf{A}(s) = 0, \mathbf{S})$	$\Rightarrow$	$(\sigma, s\|\beta, \mathbf{A}, \mathbf{S})$
Left-Arc	$(\sigma\|s, b\|\beta, \mathbf{A}, \mathbf{S})$	$\Rightarrow$	$(\sigma, s\|\beta, \mathbf{A}(s) = b, \mathbf{S}$

$(\sigma, \beta, \mathbf{A}, \mathbf{S})$ = configuration where
$\sigma|s$ = stack with topmost element s
$b|\beta$ = buffer with first element b
A = vector of head indices
A$(i) = j$ = arc wj —> wi
S = bit-vector used to prevent Shift/Unshift cycles

Practical Implications

- Actions allowed by monotonic arc-eager systems:
  - Arcs from stack words to buffer words
  - Arcs from buffer words to headless stack words
  - Arcs between words in a buffer

- Honnibal at al (2013) extra non-monotonic actions:
  - Left-arc can “clobber” existing heads
  - A word *i* on the stack can reach an arc to or from a word *j* ahead on the stack if *j* has no head

Experiment

- Trained parser on OntoNotes corpus, converted into dependencies using ClearNLP 3.1 converter

- OntoNotes corpus:
  - Telephone conversations, broadcast news, weblogs and more
  - English, Chinese, Mandarin Chinese, Arabic
  - Uses Penn Treebank for syntax and Penn PropBank for predicate-argument structure

ka4 OntoNotes Includes syntax, predicate argument structure and shallow semantics (coreference, word sense disambiguation for nouns and verbs, and some word senses connected to an ontology)

katherine amabel; 09.12.2017

- Compared three arc-eager transition systems to theirs
  - Original Arc Eager (Nivre, 2003)
  - Previous Non-Monotic (Honnibal et al., 2013)
  - Tree Constrained (Unshift) (Nivre and Fernandez-Gonzalez, 2014)
  - Combination (Honnibal and Johnson 2015)
  - Plus four state-of-the-art systems for context

Transition System	Search	UAS	LAS
Orig. Arc Eager	Greedy	91.25	89.40
Tree Constrained	Greedy	91.40	89.50
Prev. Non-Monotonic	Greedy	91.36	89.52
This work	Greedy	91.85	89.91
Chen and Manning (2014)	Greedy	89.59	87.63
Goldberg and Nivre (2012)	Greedy	90.54	88.75
Choi and Mccallum (2013)	Branch	92.26	90.84
Zhang and Nivre (2011)	Beam_{32}	92.24	90.50
Bohnet (2010)	Graph	92.50	90.70

Hannibal and Johnson (2015)

Results

- Improved Unlabelled Accuracy Score (UAS) by 0.6%
- Improved Labelled Accuracy Score (LAS) by 0.51%
- Outperformed greedy- and non-greedy transition parsers

SpaCy Now - v2.0

- Neural network models (based on Kiperwasser & Goldberg 2016):
  - “**Each sentence token is associated with a BiLSTM vector representing the token in its sentential context, and feature vectors are constructed by concatenating a few BiLSTM vectors.** The BiLSTM is trained jointly with the parser objective, resulting in very effective feature extractors for parsing (Kiperwasser and Goldberg 2016).”

- Tokenization: OntoNotes5 ka5
- Lemmatization: WordNet
- POS Tagging: Google Universal POS tag set, plus:
  - English: OntoNotes5 version of Penn Treebank tag set
  - German: TIGER Treebank + Google Universal POS tag set.
- Syntactic Dependency Parsing:
  - English: CLEAR Style (Clear NLP)
  - German: TIGER Treebank annotation

Folie 31

ka5 WordNet: WordNet® is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated with the browser.
katherine amabel; 09.12.2017