EDAN20
Language Technology
http://cs.lth.se/edan20/
Chapter 13: Dependency Parsing

Pierre Nugues

Lund University
Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

September 19, 2016

Parsing Dependencies

Generate all the pairs:

Which sentence root?

Bring

the

meal

to

the

table

Which head?

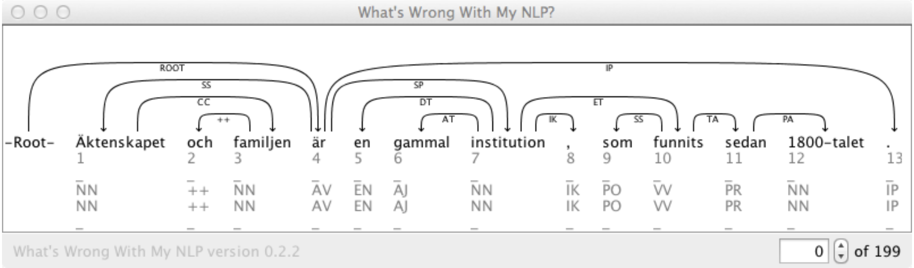
meal

Talbanken: An Annotated Corpus of Swedish

1	Äktenskapet	_	NN	NN	_	4	SS
2	och	_	++	++	_	3	++
3	familjen	_	NN	NN	_	1	CC
4	är	_	AV	AV	_	0	ROOT
5	en	_	EN	EN	_	7	DT
6	gammal	_	AJ	AJ	_	7	AT
7	institution	_	NN	NN	_	4	SP
8	,	_	IK	IK	_	7	IK
9	som	_	PO	PO	_	10	SS
10	funnits	_	VV	VV	_	7	ET
11	sedan	_	PR	PR	_	10	TA
12	1800-talet	_	NN	NN	_	11	PA
13	.	_	IP	IP	_	4	IP

Visualizing the Graph

Using *What's Wrong With My NLP*
(`https://code.google.com/p/whatswrong/`):



Parser Input

The words and their parts of speech obtained from an earlier step.

1	Äktenskapet	_	NN	NN	_
2	och	_	++	++	_
3	familjen	_	NN	NN	_
4	är	_	AV	AV	_
5	en	_	EN	EN	_
6	gammal	_	AJ	AJ	_
7	institution	_	NN	NN	_
8	,	_	IK	IK	_
9	som	_	PO	PO	_
10	funnits	_	VV	VV	_
11	sedan	_	PR	PR	_
12	1800-talet	_	NN	NN	_
13	.	_	IP	IP	_

Nivre's Parser

Joakim Nivre designed an efficient dependency parser extending the shift-reduce algorithm.
He started with Swedish and has reported the best results for this language and many others.

PP	NN	VB	PN	JJ	NN	HP	VB	PM	PM
På	60-talet	målade	han	djärva	tavlor	som	retade	Nikita	Chrusjtjov.
(In	the-60's	painted	he	bold	pictures	which	annoyed	Nikita	Chrustjev.)

His team obtained the best results in the CoNLL 2007 shared task on dependency parsing.

The Parser (Arc-Eager)

The first step is a POS tagging

The parser applies a variation/extension of the shift-reduce algorithm since dependency grammars have no nonterminal symbols

The transitions are:

1. **Shift**, pushes the input token onto the stack
2. **Right arc**, adds an arc from the token on top of the stack to the next input token and pushes the input token onto the stack.
3. **Reduce**, pops the token on the top of the stack
4. **Left arc**, adds an arc from the next input token to the token on the top of the stack and pops it.

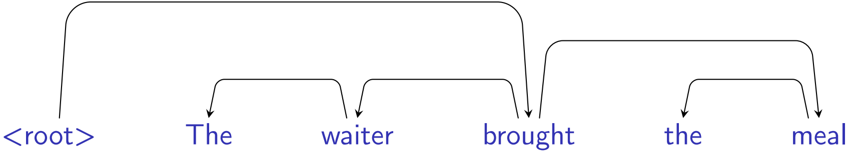
Transitions' Definition

Actions	Parser states	Conditions
Initialization	$\langle nil, W, \emptyset \rangle$	
Termination	$\langle S, [], A \rangle$	
Shift	$\langle S, [n \vert I], A \rangle \rightarrow \langle [S \vert n], I, A \rangle$	
Reduce	$\langle [S \vert n], I, A \rangle \rightarrow \langle S, I, A \rangle$	$\exists n'(n', n) \in A$
Left-arc	$\langle [S \vert n], [n' \vert I], A \rangle \rightarrow \langle S, [n' \vert I], A \cup \{(n \leftarrow n')\} \rangle$	$\nexists n''(n'', n) \in$
Right-arc	$\langle [S \vert n], [n' \vert I], A \rangle \rightarrow \langle [S \vert n, n'], I, A \cup \{(n \rightarrow n')\} \rangle$	

1. The first condition $\nexists n''(n'', n) \in A$, where n'' is the head and n, the dependent, is to enforce a unique head.
2. The second condition $\exists n'(n', n) \in A$, where n' is the head and n, the dependent, is to ensure that the graph is connected.

Nivre's Parser in Action

Input W = The waiter brought the meal.
The graph is:



$$\{\text{the} \leftarrow \text{waiter}, \text{waiter} \leftarrow \text{brought}, \text{ROOT} \rightarrow \text{brought}, \text{the} \leftarrow \text{meal}, \text{brought} \rightarrow \text{meal}\},$$

Let us apply the sequence:

`[sh, sh, la, sh, la, ra, sh, la, ra]`

Nivre's Parser in Action

[sh, sh, la, sh, la, ra, sh, la, ra]

Trans.	Stack	Queue	Graph
start	∅	[ROOT, the, waiter, brought, the, meal]	{}
sh			
	[*ROOT*]	[the, waiter, brought, the, meal]	{}
sh			
	[*the* *ROOT*]	[waiter, brought, the, meal]	{}
la			
	[*ROOT*]	[waiter, brought, the, meal]	{the ← waiter}
sh			
	[*waiter* *ROOT*]	[brought, the, meal]	{the ← waiter}
la			
	[*ROOT*]	[brought, the, meal]	{the ← waiter, waiter ← brought}

Nivre's Parser in Action (II)

[~~sh, sh, la, sh, la~~, ra, sh, la, ra]

Trans.	Stack	Queue	Graph
ra			
	brought *ROOT*	[the, meal]	{the ← waiter, waiter ← brought, ROOT → brought}
sh			
	the *brought* *ROOT*	[meal]	{the ← waiter, waiter ← brought, ROOT → brought}
la			
	brought *ROOT*	[meal]	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal}
ra			
end	*meal* *brought* *ROOT*	[]	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal, brought → meal}

Nivre's Parser in Python: Shift and Reduce

We use a `stack`, a `queue`, and a parser `state` that contains all the arcs.

```
def shift(stack, queue, state):
    stack = [queue[0]] + stack
    queue = queue[1:]
    return stack, queue, state
```

```
def reduce(stack, queue, state):
    return stack[1:], queue, state
```

Nivre's Parser in Python: Left-Arc

The state is a dictionary of dictionaries with the heads and the functions (deprels): `state['heads']` and `state['deprels']`
The `deprel` argument is is either to assign a function or to read it from the manually-annotated corpus.

```
def left_arc(stack, queue, state, deprel=False):
    state['heads'][stack[0]['id']] = queue[0]['id']
    if deprel:
        state['deprels'][stack[0]['id']] = deprel
    else:
        state['deprels'][stack[0]['id']] = stack[0]['deprel']
    return reduce(stack, queue, state)
```

Nivre's Parser in Prolog: Left-Arc

```
% shift_reduce(+Sentence, -Graph)
shift_reduce(Sentence, Graph) :-
  shift_reduce(Sentence, [], [], Graph).

% shift_reduce(+Words, +Stack, +CurGraph, -FinGraph)
shift_reduce([], _, Graph, Graph).
shift_reduce(Words, Stack, Graph, FinalGraph) :-
  left_arc(Words, Stack, NewStack, Graph, NewGraph),
  write('left arc'), nl,
  shift_reduce(Words, NewStack, NewGraph, FinalGraph).
```

Nivre's Parser in Prolog: Left-Arc (II)

```
% left_arc(+WordList, +Stack, -NewStack, +Graph, -NewGraph)

left_arc([w(First, PosF) | _], [w(Top, PosT) | Stack],
    Stack, Graph, [d(w(First, PosF),
    w(Top, PosT), Function) | Graph]) :-
  word(First, FirstPOS),
  word(Top, TopPOS),
  drule(FirstPOS, TopPOS, Function, left),
  \+ member(d(_, w(Top, PosT), _), Graph).
```

Gold Standard Parsing

Nivre's parser uses a sequence of actions taken in the set
`{la, ra, re, sh}`.
We have:

- A sequence of actions creates a dependency graph
- Given a projective dependency graph, we can find an action sequence creating this graph. This is gold standard parsing.

Let TOP be the top of the stack and $FIRST$, the first token of the input list, and A the dependency graph.

1. **if** $arc(TOP, FIRST) \in A$, **then** right-arc;
2. **else if** $arc(FIRST, TOP) \in A$, **then** left-arc;
3. **else if** $\exists k \in Stack, arc(FIRST, k) \in A$ or $arc(k, FIRST) \in A$, **then** reduce;
4. **else** shift.

Parsing a Sentence

When parsing an unknown sentence, we do not know the dependencies yet

The parser will use a "guide" to tell which transition to apply in the set `{la, ra, re, sh}`.

The parser will extract a context from its current state, for instance the part of speech of the top of the stack and the first in the queue, and will ask the guide.

D-rules are a simply way to implement this

Dependency Rules

D-rules are possible relations between a head and a dependent.
They involve part-of-speech, mostly, and words

1. `determiner` $\leftarrow$ `noun`.
2. `adjective` $\leftarrow$ `noun`.
3. `preposition` $\leftarrow$ `noun`.
4. `noun` $\leftarrow$ `verb`.
5. `preposition` $\leftarrow$ `verb`.
6. `verb` $\leftarrow$ `root`.

$$\begin{bmatrix} \mathit{category} : \mathit{noun} \\ \mathit{number} : N \\ \mathit{person} : P \\ \mathit{case} : \mathit{nominative} \end{bmatrix} \leftarrow \begin{bmatrix} \mathit{category} : \mathit{verb} \\ \mathit{number} : N \\ \mathit{person} : P \end{bmatrix}$$

Parsing Dependency Rules in Prolog

```
%drule(Head, Dependent, Function).

drule(noun, determiner, determinative).
drule(noun, adjective, attribute).
drule(verb, noun, subject).
drule(verb, noun, object).
```

D-Rules may also include a direction, for instance a determiner is always to the left

```
%drule(Head, Dependent, Function, Direction).
```

Tracing Nivre's Parser in Python

1	Jag	_	PO	PO	_	2	SS	_	_
2	tycker	_	VV	VV	_	0	ROOT	_	_
3	det	_	PO	PO	_	2	OO	_	_
4	inte	_	AB	AB	_	2	NA	_	_
5	.	_	IP	IP	_	2	IP	_	_

Transitions:

```
['sh', 'sh', 'la.SS', 'ra.ROOT', 'ra.OO', 're', 'ra.NA',
  're', 'ra.IP']
```

Parser state:

```
{'heads': {'4': '2', '5': '2', '3': '2', '2': '0', '1': '2',
  '0': '0'},
'deprels': {'4': 'NA', '5': 'IP', '3': 'OO', '2': 'ROOT', '1':
  'SS', '0': 'ROOT'}}
```

Tracing Nivre's Parser (Prolog)

```
shift_reduce([w(the, 1), w(waiter, 2), w(brought, 3),
w(the, 4), w(meal, 5)], G).
shift
left arc
shift
left arc
shift
shift
left arc
right arc
G = [d(w(brought, 3), w(meal, 5), object),
d(w(meal, 5), w(the, 4), determinative),
d(w(brought, 3), w(waiter, 2), subject),
d(w(waiter, 2), w(the, 1), determinative)]
```

Using Features

D-rules consider a limited context: the part of speech of the top of the stack and the first in the queue

We can extend the context:

- Extracts more features (attributes), for instance two words in the stack, three words in the queue
- Use them as input to a four-class classifier and determine the next action

Training a Classifier

Gold standard parsing of a manually annotated corpus produces training data

Stack	**Queue**	**Stack**		**Queue**		**Trans.**
POS(T_0)	POS(Q_0)	POS(T_0)	POS(T_{-1})	POS(Q_0)	POS(Q_{+1})	
nil	ROOT	nil	nil	ROOT	DT	sh
ROOT	DT	ROOT	nil	DT	NN	sh
DT	NN	DT	ROOT	NN	VBD	la
ROOT	NN	ROOT	nil	NN	VBD	sh
NN	VBD	NN	ROOT	VBD	DT	la
ROOT	VBD	ROOT	nil	VBD	DT	ra
VBD	DT	VBD	ROOT	DT	NN	sh
DT	NN	DT	VBD	NN	nil	la
VBD	NN	VBD	ROOT	NN	nil	ra

Using Talbanken and CoNLL 2006 data, you can train decision trees and implement a parser.

Feature Vectors

You extract one feature (attribute) vector for each parsing action.

The most elementary feature vector consists of two parameters: `POS_TOP`, `POS_FIRST`

Nivre et al. (2006) used from 16 to 30 parameters and support vector machines.

As machine-learning algorithm, you can use decision trees, perceptron, logistic regression, or support vector machines.

Finding Dependencies using Constraints

Parts of speech	Possible governors	Possible functions
Determiner	noun	det
Noun	verb	object, iobject
Noun	prep	pcomp
Verb	root	root
Prep	verb, noun	mod, loc

Tagging

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	nil, root	3, det 6, det	4, pcomp 1, object 1, iobject	3, mod 1, loc	3, det 6, det	4, pcomp 1, object 1, iobject

A second step applies and propagates constraint rules.
Rules for English describe: projectivity – links must not cross –, function uniqueness – there is only one subject, one object, one indirect object –, topology

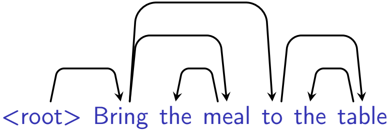
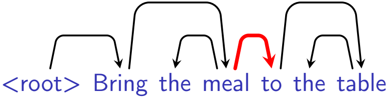
Constraints

- A determiner has its head to its right-hand side
- A subject has its head to its right-hand side when the verb is at the active form
- An object and an indirect object have their head to their left-hand side (active form)
- A prepositional complement has its head to its left-hand side

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	nil, root	3, det	1, iobject	3, mod	6, det	4, pcomp
			1, object	1, loc		

Evaluation of Dependency Parsing

Dependency parsing: The error count is the number of words that are assigned a wrong head, here 1/6.

Reference	Output
	

Parser Variant: Arc-Standard

Nivre's parser has two variants in addition to arc-eager: arc-standard (Yamada and Matsumoto) and swap
The first step is a POS tagging
The transitions are:

1. **Shift**, pushes the input token onto the stack
2. **Right arc**, adds an arc from the second token in the stack to the top of the stack and pops it.
3. **Left arc**, adds an arc from the top of the stack to the second in the stack and removes the second in the stack.

Transitions' Definition of Arc-Standard

Actions	Parser states	Conditions
Initialization	$\langle nil, W, \emptyset \rangle$	
Termination	$\langle [ROOT], [], A \rangle$	
Shift	$\langle S, [n\|I], A \rangle \rightarrow \langle [S\|n], I, A \rangle$	
Left-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n'], I, A \cup \{(n \leftarrow n')\} \rangle$	$n \neq ROOT$
Right-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n], I, A \cup \{(n \rightarrow n')\} \rangle$	

Arc Standard Parser in Action (I)

Trans.	Stack	Queue	Graph
start	[ROOT]	[I, booked, a, ticket, to, Google]	{}
sh			
	$\begin{bmatrix} \text{I} \\ \text{ROOT} \end{bmatrix}$	[booked, a, ticket, to, Google]	{}
sh			
	$\begin{bmatrix} \text{booked} \\ \text{I} \\ \text{ROOT} \end{bmatrix}$	[a, ticket, to, Google]	{}
la			
	$\begin{bmatrix} \text{booked} \\ \text{ROOT} \end{bmatrix}$	[a, ticket, to, Google]	$\{\text{I} \leftarrow \text{booked}\}$
sh			
	$\begin{bmatrix} \text{a} \\ \text{booked} \\ \text{ROOT} \end{bmatrix}$	[ticket, to, Google]	$\{\text{I} \leftarrow \text{booked}\}$

Arc Standard Parser in Action (II)

Trans.	Stack	Queue	Graph
sh			
	[ticket a booked ROOT]	[to, Google]	{I ← booked}
la			
	[ticket booked ROOT]	[to, Google]	{I ← booked, a ← ticket}
sh			
	[to ticket booked ROOT]	[Google]	{I ← booked, a ← ticket}

Arc Standard Parser in Action (III)

Trans.	Stack	Queue	Graph
sh			
	[Google to ticket booked ROOT]	[]	{I ← booked, a ← ticket}
ra			
	[to ticket booked ROOT]	[]	{I ← booked, a ← ticket, to → Google}
ra			
	[ticket booked ROOT]	[]	{I ← booked, a ← ticket, to → Google, ticket → to}

Arc Standard Parser in Action (IV)

Trans.	Stack	Queue	Graph
ra			
	$\begin{bmatrix}\text{booked}\\ \text{ROOT}\end{bmatrix}$	[]	$\{\text{I} \leftarrow \text{booked}$, $\text{a} \leftarrow \text{ticket}$, $\text{to} \rightarrow \text{Google}$, $\text{ticket} \rightarrow \text{to}$, $\text{booked} \rightarrow \text{ticket}\}$

Transitions' Definition of Swap

The Swap variant enables the parser to parse nonprojective sentences

Actions	Parser states	Conditions
Initialization	$\langle nil, W, \emptyset \rangle$	
Termination	$\langle [ROOT], [], A \rangle$	
Shift	$\langle S, [n\|I], A \rangle \rightarrow \langle [S\|n], I, A \rangle$	
Left-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n'], I, A \cup \{(n \leftarrow n')\} \rangle$	$n \neq ROOT$
Right-arc	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n], I, A \cup \{(n \rightarrow n')\} \rangle$	
Swap	$\langle [S\|n, n'], I, A \rangle \rightarrow \langle [S\|n'], [n\|I], A \rangle$	$n \neq ROOT$ and $inx(n) < inx(n')$

Application: Dependency Parsing for Knowledge Extraction

IBM Watson, the question-answering system, uses dependency parsing to

- Analyze questions and
- Extract knowledge from text.

Given the question:

> *POETS & POETRY: He was a bank clerk in the Yukon before* ***he published “Songs of a Sourdough”*** *in 1907*

Watson extracts:

```
authorOf(he, 'Songs of a Sourdough')
```

A predicate–argument structure representation would be:

```
published(he, 'Songs of a Sourdough')
```

IBM Watson: Parsing the Question

Watson parses the question in the form of dependencies.

- In the CoNLL format:

Inx	Form	Lemma	POS	Head	Funct.
1	he	he	pronoun	2	subject
2	published	publish	verb	0	root
3	Songs of a Sourdough	Songs of a Sourdough	noun	2	object

- In the Watson format:

```
lemma(1, "he").                      partOfSpeech(1,pronoun).
lemma(2, "publish").                 partOfSpeech(2,verb).
lemma(3, "Songs of a Sourdough").    partOfSpeech(3,noun).

subject(2,1).
object(2,3).
```

IBM Watson: Inferences

Watson uses Prolog rules to detect author/composition relationships:

```
authorOf(Author,Composition) :-
    createVerb(Verb),
    subject(Verb,Author),
    author(Author),
    object(Verb,Composition),
    composition(Composition).

createVerb(Verb) :-
    partOfSpeech(Verb,verb),
    lemma(Verb,VerbLemma),
    member(VerbLemma, ["write", "publish",...]).
```

Eventually, the question is reduced to:

```
authorOf(he, 'Songs of a Sourdough')
```

IBM Watson: Evidence from Text

Watson parses large volumes of text, for instance Wikipedia and the *New York Times*.
From the excerpt:

> *Songs of a Sourdough by Robert W. Service*

Watson extracts:

```
authorOf('Robert W. Service', 'Songs of a Sourdough')
```

The classical predicate–argument structure representation of the same phrase is:

```
by('Songs of a Sourdough', 'Robert W. Service')
```

IBM Watson: Matching Evidences

The relation is extracted from text using the rule:

```
authorOf(Author,Composition) :-
    composition(Composition),
    argument(Composition,Preposition),
    lemma(Preposition, "by"),
    objectOfPreposition(Preposition,Author),
    author(Author).
```

Leading to:

```
authorOf('Robert W. Service', 'Songs of a Sourdough')
```

This predicate–argument structure can be compared to:

```
authorOf(he, 'Songs of a Sourdough')
```

Or unified with:

```
authorOf(X, 'Songs of a Sourdough')
```