Dependency Parsing

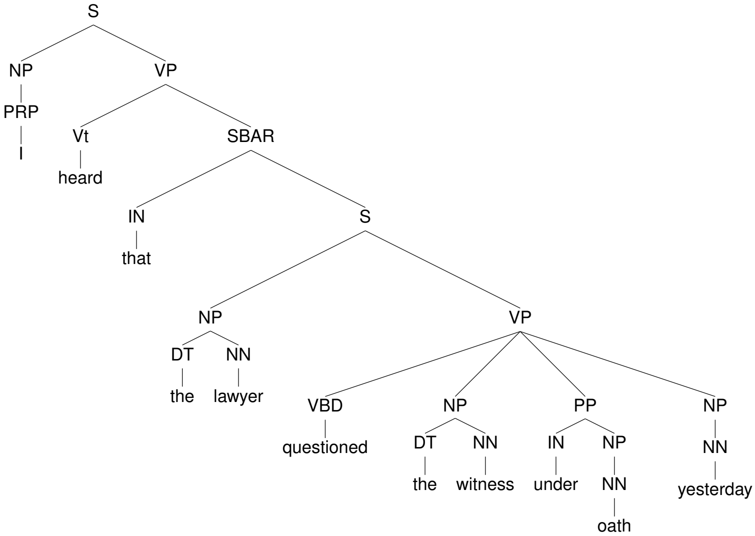
Yoav Goldberg

Bar Ilan University

(with slides from Michael Collins, Sasha Rush)

Reminder

Constituency Trees



Reminder

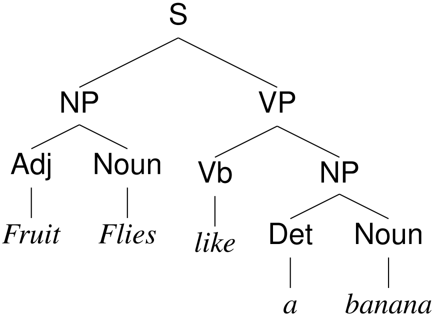
PCFG Parsing

- Assume trees are generated by a (P)CFG.
- Extract grammar rules from treebank.
- Each rule in a derivation has a score.
- **Parsing**: find the tree with the overall best score.
  - Using the CKY algorithm

Extracting CFG from Trees

- The leafs of the trees define Σ
- The internal nodes of the trees define N
- Add a special S symbol on top of all trees
- Each node an its children is a rule in R

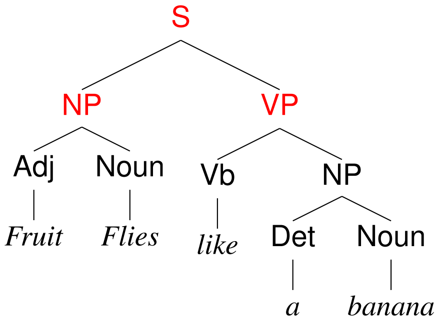
Extracting Rules



Extracting CFG from Trees

- The leafs of the trees define Σ
- The internal nodes of the trees define N
- Add a special S symbol on top of all trees
- Each node an its children is a rule in R

Extracting Rules

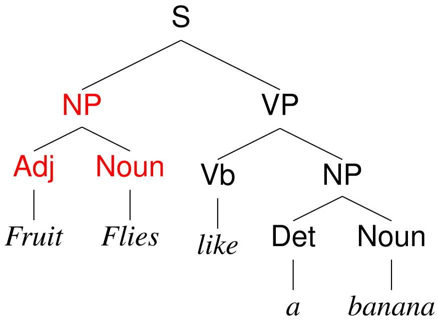


S $\rightarrow$ NP VP

Extracting CFG from Trees

- The leafs of the trees define Σ
- The internal nodes of the trees define N
- Add a special S symbol on top of all trees
- Each node an its children is a rule in R

Extracting Rules

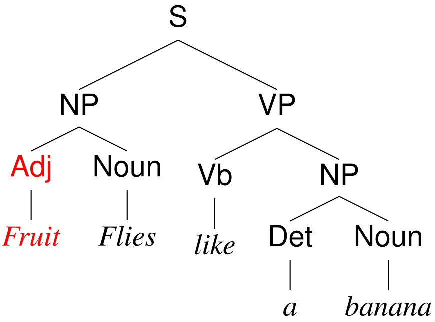


S $\rightarrow$ NP VP
NP $\rightarrow$ Adj Noun

Extracting CFG from Trees

- The leafs of the trees define Σ
- The internal nodes of the trees define N
- Add a special S symbol on top of all trees
- Each node an its children is a rule in R

Extracting Rules



S $\rightarrow$ NP VP
NP $\rightarrow$ Adj Noun
Adj $\rightarrow$ fruit

From CFG to PCFG

- English is NOT generated from CFG $\Rightarrow$ It's generated by a PCFG!

From CFG to PCFG

- English is NOT generated from CFG $\Rightarrow$ It's generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.

From CFG to PCFG

- English is NOT generated from CFG $\Rightarrow$ It's generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.
- Multiplying all the rule probs in a derivation gives the probability of the derivation.
- We want the tree with maximum probability.

From CFG to PCFG

- English is NOT generated from CFG $\Rightarrow$ It's generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.
- Multiplying all the rule probs in a derivation gives the probability of the derivation.
- We want the tree with maximum probability.

More Formally

$$P(tree, sent) = \prod_{l \to r \in deriv(tree)} q(l \to r)$$

From CFG to PCFG

- English is NOT generated from CFG $\Rightarrow$ It's generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.
- Multiplying all the rule probs in a derivation gives the probability of the derivation.
- We want the tree with maximum probability.

More Formally

$$P(tree, sent) = \prod_{l \to r \in deriv(tree)} q(l \to r)$$

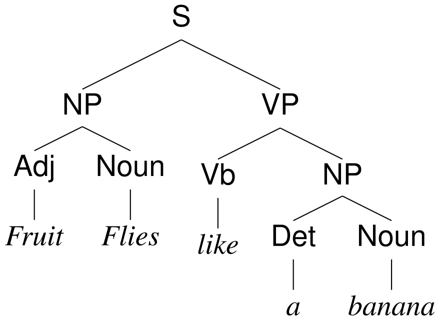
$$tree = \arg\max_{tree \in trees(sent)} P(tree|sent) = \arg\max_{tree \in trees(sent)} P(tree, sent)$$

PCFG Example

a simple PCFG

1.0 S $\rightarrow$ NP VP
0.3 NP $\rightarrow$ Adj Noun
0.7 NP $\rightarrow$ Det Noun
1.0 VP $\rightarrow$ Vb NP
-
0.2 Adj $\rightarrow$ fruit
0.2 Noun $\rightarrow$ flies
1.0 Vb $\rightarrow$ like
1.0 Det $\rightarrow$ a
0.4 Noun $\rightarrow$ banana
0.4 Noun $\rightarrow$ tomato
0.8 Adj $\rightarrow$ angry

Example



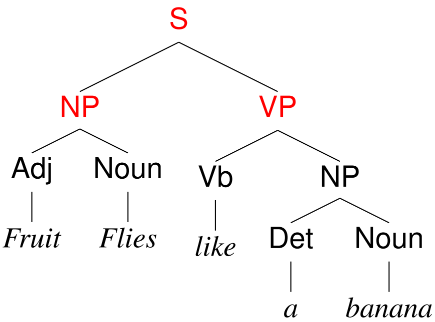
$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033$

PCFG Example

a simple PCFG

1.0 S → NP VP
0.3 NP → Adj Noun
0.7 NP → Det Noun
1.0 VP → Vb NP
-
0.2 Adj → fruit
0.2 Noun → flies
1.0 Vb → like
1.0 Det → a
0.4 Noun → banana
0.4 Noun → tomato
0.8 Adj → angry

Example



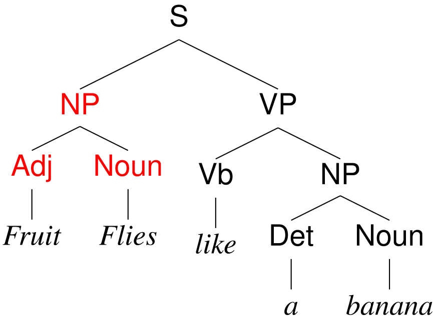
$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033$

PCFG Example

a simple PCFG

1.0 S → NP VP
0.3 NP → Adj Noun
0.7 NP → Det Noun
1.0 VP → Vb NP
-
0.2 Adj → fruit
0.2 Noun → flies
1.0 Vb → like
1.0 Det → a
0.4 Noun → banana
0.4 Noun → tomato
0.8 Adj → angry

Example



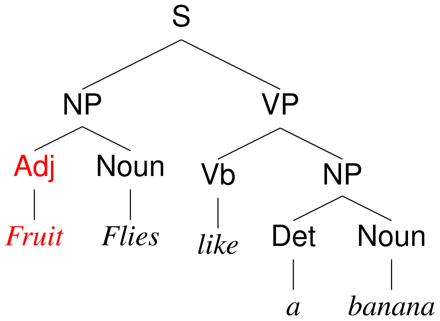
$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033$

PCFG Example

a simple PCFG

1.0 S → NP VP
0.3 NP → Adj Noun
0.7 NP → Det Noun
1.0 VP → Vb NP
-
0.2 Adj → fruit
0.2 Noun → flies
1.0 Vb → like
1.0 Det → a
0.4 Noun → banana
0.4 Noun → tomato
0.8 Adj → angry

Example



$1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 =$
0.0033

Parsing with PCFG

- Parsing with a PCFG is finding the most probable derivation for a given sentence.
- This can be done quite efficiently with dynamic programming (the CKY algorithm)

Parsing with PCFG

- Parsing with a PCFG is finding the most probable derivation for a given sentence.
- This can be done quite efficiently with dynamic programming (the CKY algorithm)

Obtaining the probabilities

- We estimate them from the Treebank.
- $q(LHS \rightarrow RHS) = \frac{count(LHS \rightarrow RHS)}{count(LHS \rightarrow \Diamond)}$
- We can also add smoothing and backoff, as before.
- Dealing with unknown words - like in the HMM

The big question

Does this work?

Evaluation

Parsing Evaluation

- Let's assume we have a parser, how do we know how good it is?

$\Rightarrow$ Compare output trees to gold trees.

Parsing Evaluation

- Let's assume we have a parser, how do we know how good it is?

$\Rightarrow$ Compare output trees to gold trees.

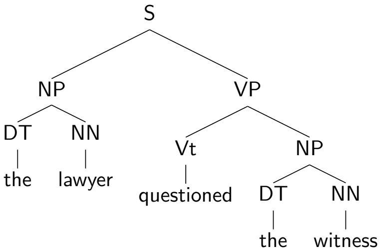
  - But how do we compare trees?
  - Credit of 1 if tree is correct and 0 otherwise, is too harsh.

Parsing Evaluation

- Let's assume we have a parser, how do we know how good it is?

⇒ Compare output trees to gold trees.
  - But how do we compare trees?
  - Credit of 1 if tree is correct and 0 otherwise, is too harsh.
- Represent each tree as a set of labeled spans.
  - NP from word 1 to word 5.
  - VP from word 3 to word 4.
  - S from word 1 to word 23.
  - ...
- Measure Precision, Recall and F_1 over these spans, as in the segmentation case.

Evaluation: Representing Trees as Constituents



Label	Start Point	End Point
NP	1	2
NP	4	5
VP	3	5
S	1	5

Precision and Recall

Label	Start Point	End Point
NP	1	2
NP	4	5
NP	4	8
PP	6	8
NP	7	8
VP	3	8
S	1	8

Label	Start Point	End Point
NP	1	2
NP	4	5
PP	6	8
NP	7	8
VP	3	8
S	1	8

- G = number of constituents in gold standard = 7
- P = number in parse output = 6
- C = number correct = 6

$$\text{Recall} = 100\% \times \frac{C}{G} = 100\% \times \frac{6}{7} \qquad \text{Precision} = 100\% \times \frac{C}{P} = 100\% \times \frac{6}{6}$$

Parsing Evaluation

- Is this a good measure?
  - Why? Why not?

Parsing Evaluation

How well does the PCFG parser we learned do?

Not very well: about 73% F_1 score.

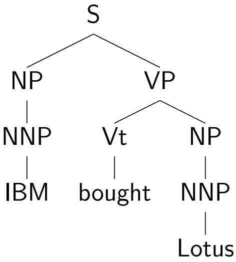
Problems with PCFGs

Weaknesses of Probabilistic Context-Free Grammars

Michael Collins, Columbia University

Weaknesses of PCFGs

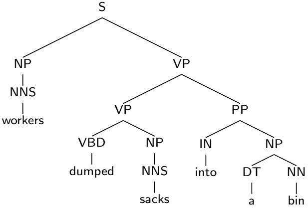
- Lack of sensitivity to lexical information
- Lack of sensitivity to structural frequencies



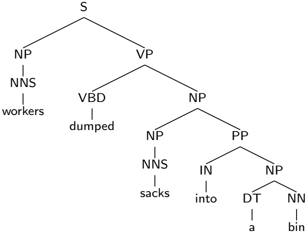
$$
\begin{aligned}
\mathsf{p(t)} = \quad & q(\mathsf{S} \rightarrow \mathsf{NP\ VP}) & \times q(\mathsf{NNP} \rightarrow IBM) \\
& \times q(\mathsf{VP} \rightarrow \mathsf{V\ NP}) & \times q(\mathsf{Vt} \rightarrow bought) \\
& \times q(\mathsf{NP} \rightarrow \mathsf{NNP}) & \times q(\mathsf{NNP} \rightarrow Lotus) \\
& \times q(\mathsf{NP} \rightarrow \mathsf{NNP}) &
\end{aligned}
$$

Another Case of PP Attachment Ambiguity

(a)



(b)



(a)

Rules
S $\rightarrow$ NP VP
NP $\rightarrow$ NNS
VP $\rightarrow$ VP PP
VP $\rightarrow$ VBD NP
NP $\rightarrow$ NNS
PP $\rightarrow$ IN NP
NP $\rightarrow$ DT NN
NNS $\rightarrow$ workers
VBD $\rightarrow$ dumped
NNS $\rightarrow$ sacks
IN $\rightarrow$ into
DT $\rightarrow$ a
NN $\rightarrow$ bin

(b)

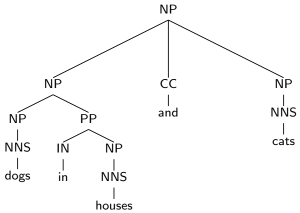
Rules
S $\rightarrow$ NP VP
NP $\rightarrow$ NNS
NP $\rightarrow$ NP PP
VP $\rightarrow$ VBD NP
NP $\rightarrow$ NNS
PP $\rightarrow$ IN NP
NP $\rightarrow$ DT NN
NNS $\rightarrow$ workers
VBD $\rightarrow$ dumped
NNS $\rightarrow$ sacks
IN $\rightarrow$ into
DT $\rightarrow$ a
NN $\rightarrow$ bin

If $q(\text{NP} \rightarrow \text{NP PP}) > q(\text{VP} \rightarrow \text{VP PP})$ then (b) is more probable, else (a) is more probable.

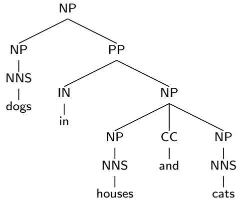
Attachment decision is completely independent of the words

A Case of Coordination Ambiguity

(a)



(b)



(a)

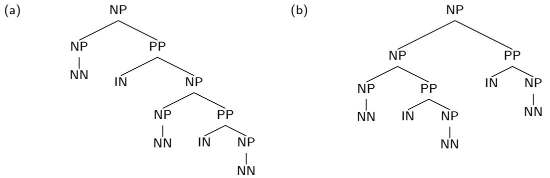
Rules
NP $\rightarrow$ NP CC NP
NP $\rightarrow$ NP PP
NP $\rightarrow$ NNS
PP $\rightarrow$ IN NP
NP $\rightarrow$ NNS
NP $\rightarrow$ NNS
NNS $\rightarrow$ dogs
IN $\rightarrow$ in
NNS $\rightarrow$ houses
CC $\rightarrow$ and
NNS $\rightarrow$ cats

(b)

Rules
NP $\rightarrow$ NP CC NP
NP $\rightarrow$ NP PP
NP $\rightarrow$ NNS
PP $\rightarrow$ IN NP
NP $\rightarrow$ NNS
NP $\rightarrow$ NNS
NNS $\rightarrow$ dogs
IN $\rightarrow$ in
NNS $\rightarrow$ houses
CC $\rightarrow$ and
NNS $\rightarrow$ cats

Here the two parses have identical rules, and therefore have identical probability under any assignment of PCFG rule probabilities

Structural Preferences: Close Attachment



- Example: president of a company in Africa
- Both parses have the same rules, therefore receive same probability under a PCFG
- "Close attachment" (structure (a)) is twice as likely in Wall Street Journal text.

Lexicalized PCFGs

PCFG Problem 1

Lack of sensitivity to lexical information (words)

Solution

- Make PCFG aware of words (*lexicalized* PCFG)
- Main Idea: **Head Words**

Head Words

Each constituent has one words which captures its “essence”.

Head Words

Each constituent has one words which captures its “essence”.

- (S John saw the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP saw the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young boy with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large **hat**)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large **hat**)
- (PP with the large hat)

Head Words

Each constituent has one words which captures its "essence".

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large **hat**)
- (PP **with** the large hat)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large **hat**)
- (PP **with** the large **hat**)

Head Words

Each constituent has one words which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large **hat**)
- (PP **with** the large hat)
  - **hat** is the “semantic head”
  - **with** is the “functional head”
  - (it is common to choose the functional head)

Heads in Context-Free Rules

Add annotations specifying the "head" of each rule:

S	$\Rightarrow$	NP	VP
VP	$\Rightarrow$	Vi	
VP	$\Rightarrow$	Vt	NP
VP	$\Rightarrow$	VP	PP
NP	$\Rightarrow$	DT	NN
NP	$\Rightarrow$	NP	PP
PP	$\Rightarrow$	IN	NP

Vi	$\Rightarrow$	sleeps
Vt	$\Rightarrow$	saw
NN	$\Rightarrow$	man
NN	$\Rightarrow$	woman
NN	$\Rightarrow$	telescope
DT	$\Rightarrow$	the
IN	$\Rightarrow$	with
IN	$\Rightarrow$	in

More about Heads

- Each context-free rule has one "special" child that is the head of the rule. e.g.,

S	$\Rightarrow$	NP	VP		(VP is the head)
VP	$\Rightarrow$	Vt	NP		(Vt is the head)
NP	$\Rightarrow$	DT	NN	NN	(NN is the head)

- A core idea in syntax (e.g., see X-bar Theory, Head-Driven Phrase Structure Grammar)
- Some intuitions:
  - The central sub-constituent of each rule.
  - The semantic predicate in each rule.

Rules which Recover Heads: An Example for NPs

If the rule contains NN, NNS, or NNP:
    Choose the rightmost NN, NNS, or NNP

Else If the rule contains an NP: Choose the leftmost NP

Else If the rule contains a JJ: Choose the rightmost JJ

Else If the rule contains a CD: Choose the rightmost CD

Else Choose the rightmost child

e.g.,

NP	$\Rightarrow$	DT	NNP	NN
NP	$\Rightarrow$	DT	NN	NNP
NP	$\Rightarrow$	NP	PP	
NP	$\Rightarrow$	DT	JJ	
NP	$\Rightarrow$	DT		

Rules which Recover Heads: An Example for VPs

If the rule contains Vi or Vt: Choose the leftmost Vi or Vt

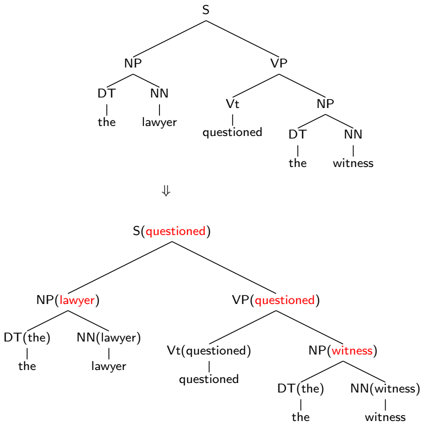
Else If the rule contains an VP: Choose the leftmost VP

Else Choose the leftmost child

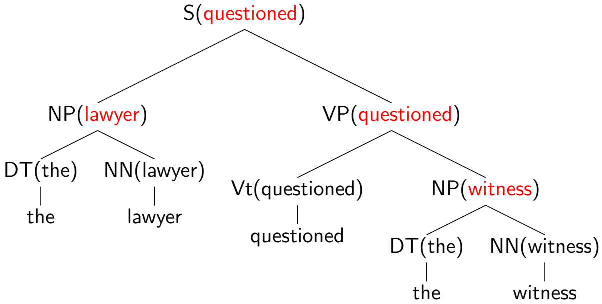
e.g.,

$$\begin{array}{llll} \text{VP} & \Rightarrow & \text{Vt} & \text{NP} \\ \text{VP} & \Rightarrow & \text{VP} & \text{PP} \end{array}$$

Adding Headwords to Trees



Adding Headwords to Trees (Continued)



- A constituent receives its headword from its **head child**.

S	$\Rightarrow$	NP	VP		(S receives headword from VP)
VP	$\Rightarrow$	Vt	NP		(VP receives headword from Vt)
NP	$\Rightarrow$	DT		NN	(NP receives headword from NN)

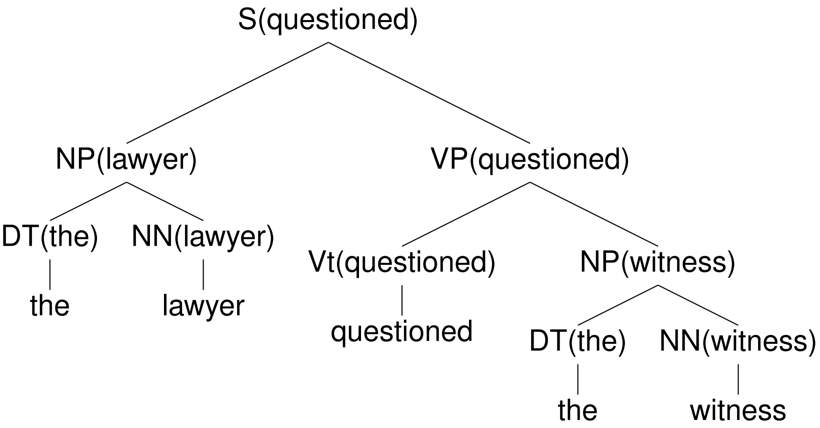
Dependency Representation

Dependency Representation

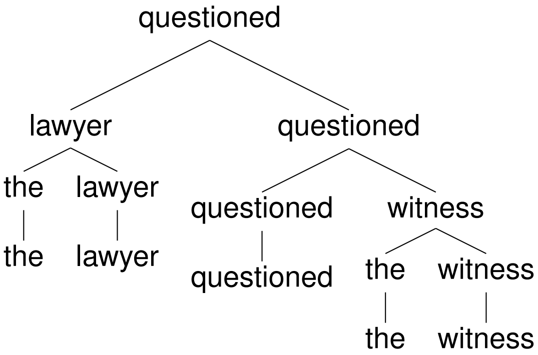
If we take the head-annotated trees and “forget” about the constituents, we get a representation called “dependency structure”.

Dependency structure capture the relation between words in a sentence.

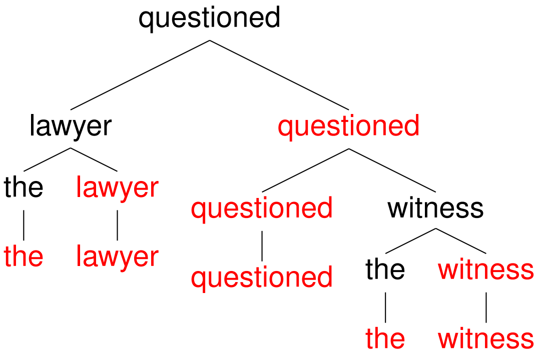
Dependency Representation



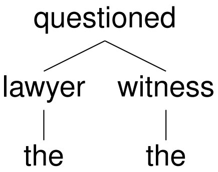
Dependency Representation



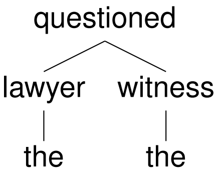
Dependency Representation

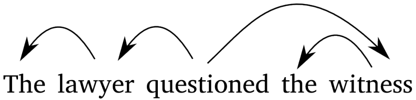


Dependency Representation

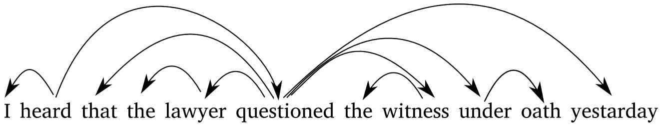


Dependency Representation

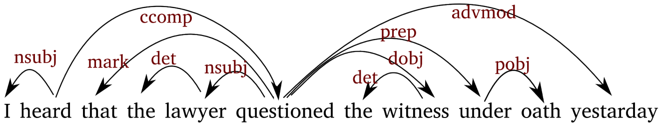




Dependency Representation



Dependency Representation



Dependency Representations

There are many different dependency representations

- Different choice of heads.
- Different set of labels.
- Each language usually has its own treebank, with own choices
- A common (and good) one for English: **Stanford Dependencies**
  - Prefer relations between words as heads.
  - About 50 labels.
- Recently, Trees in Stanford-Dependencies available for different languages.
  - Google's Universal Dependency Treebank

Universal Dependencies

- A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.

Universal Dependencies

- A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.
- Abstract over linguistic differences.
- Same set of parts-of-speech and morphology features.
- Same dependency relations.
- Same choice of heads.

Universal Dependencies

- A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.
- Abstract over linguistic differences.
- Same set of parts-of-speech and morphology features.
- Same dependency relations.
- Same choice of heads.
- Why is this good? why is this interesting?

Universal Dependencies

- A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.
- Abstract over linguistic differences.
- Same set of parts-of-speech and morphology features.
- Same dependency relations.
- Same choice of heads.
- Why is this good? why is this interesting?
- Interesting project/research idea: are the annotations really consistent across languages? do languages differ only in word order?

Let's analyze!

John saw Mary .

Let's analyze!

John saw Mary .

a yellow garbage can

Let's analyze!

He said that the boy who was wearing the blue shirt with

the white pockets has left the building .

Let's analyze!

a large pile of carrots and peas was closely guarded by dogs .

Let's analyze!

They wanted to buy cakes and eat them on the road .

Some tricky cases

I bought soda and pizza for John and Mary .

Some tricky cases

I bought soda and pizza for 4 and 57 cents.

Some tricky cases

I ordered five books but received four.

Some tricky cases

While Sue has many toys, Alice doesn't have any.

Some tricky cases

Cut, chop and peel the tomatoes.

Some tricky cases

Cut the tomatoes. Put in a bowl.

- Coordination is interesting and important.
- Missing elements are interesting and important.
  - on the border of syntax and discourse.
- Lots of work to do!

Dependency Parsing

Evaluation Measures

- UAS. Unlabeled Attachment Scores
  (% of words with correct head)
- LAS. Labeled Attachment Scores
  (% of words with correct head **and label**)
- Root
  (% of sentences with correct root)
- Exact
  (% of sentences with exact correct structure)

Evaluation Measures

- UAS. Unlabeled Attachment Scores — 90-94 (Eng, WSJ)
  (% of words with correct head)
- LAS. Labeled Attachment Scores — 87-92 (Eng, WSJ)
  (% of words with correct head **and label**)
- Root — $\sim$90 (Eng, WSJ)
  (% of sentences with correct root)
- Exact — 40-50 (Eng, WSJ)
  (% of sentences with exact correct structure)

Three main approaches to Dependency Parsing

Conversion

- Parse to constituency structure.
- Extract dependencies from the trees.

Global Optimization (Graph based)

- **Define** a scoring function over <sentence,tree> pairs.
- **Search** for best-scoring structure.
- Simpler scoring $\Rightarrow$ easier search.
- (Similar to how we do tagging, constituency parsing.)

Greedy decoding (Transition based)

- Start with an unparsed sentence.
- Apply **locally-optimal** actions until sentence is parsed.

Three main approaches to Dependency Parsing

Conversion

- Parse to constituency structure.
- Extract dependencies from the trees.

Global Optimization

`argmax over combinatorial space`

- **Define** a scoring function over <sentence,tree> pairs.
- **Search** for best-scoring structure.
- Simpler scoring $\Rightarrow$ easier search.
- (Similar to how we do tagging, constituency parsing.)

Greedy decoding

`while (!done) { do best thing }`

- Start with an unparsed sentence.
- Apply **locally-optimal** actions until sentence is parsed.

Graph-based parsing (Global Search)

Dependency Parsing

Alexander Rush
srush@csail.mit.edu
NYU CS 3033

Arcs

Dependency parsing is concerned with head-modifier relationships.

Definitions:

- **head**; the main word in a phrase
- **modifier**; an auxiliary word in a phrase

Meaning depends on underlying linguistic formalism.

Common to use head$\rightarrow$modifier arc notation

* Millions on the coast face freak storm

Input Notation

Input:

- $x = (w, t)$
- $w_1 \ldots w_n$; the words of the sentence
- $t_1 \ldots t_n$; the tags of the sentence
- Special symbol $w_0 = *$; the pseudo-root

Note: Unlike in CFG parsing, we assume tags are given.

Output Notation

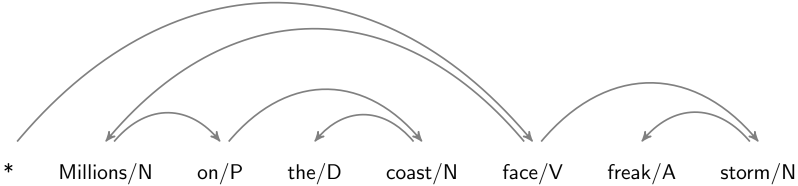
Output:

- set of possible dependency arcs

$$\mathcal{A} = \{(h, m) : h \in \{0 \ldots n\}, m \in \{1 \ldots n\}\}$$

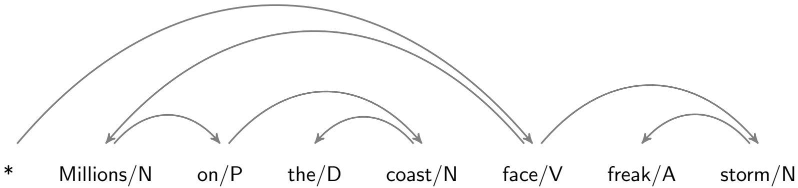
- $\mathcal{Y} \subset \{0, 1\}^{|\mathcal{A}|}$; set of all valid dependency parses
- $y \in \mathcal{Y}$; a valid dependency parse

Example



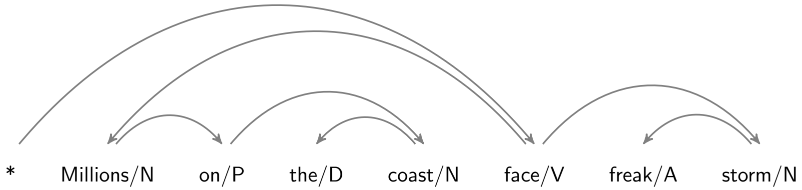
- $w_0 = *,\ w_1 = \text{Millions},\ w_2 = \text{on},\ w_3 = \text{the},\ \ldots$

Example



- $w_0 = *,\ w_1 = \text{Millions},\ w_2 = \text{on},\ w_3 = \text{the},\ \ldots$
- $t_0 = *,\ t_1 = \text{N},\ t_2 = \text{P},\ t_3 = \text{D},\ \ldots$

Example

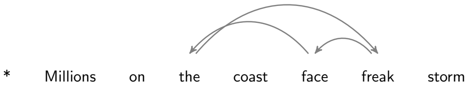


- $w_0 = *,\ w_1 = \mathrm{Millions},\ w_2 = \mathrm{on},\ w_3 = \mathrm{the},\ \ldots$
- $t_0 = *,\ t_1 = \mathrm{N},\ t_2 = \mathrm{P},\ t_3 = \mathrm{D},\ \ldots$
- $y(0,5) = 1,\ y(5,1) = 1,\ y(1,2) = 1\ \ldots$

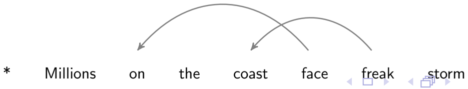
Forbidden Structures

- Each (non-root) word must modify exactly one word.

- Arcs must form a tree.



- (Projective) Arcs may not cross each other.



Main Idea

- Define a scoring function $g(y; x, \theta)$
- This function will tell us, for every x (sentence) and y (tree) pair, how good the pair is.
- θ are the parameters, or weights (we called them w before)
- For example: $g(y; x, \theta) = \sum_i \Phi_i(x, y)\theta_i = \Phi(x, y) \cdot \theta$
  - (a linear model)
- Look for the best y for a given sentence $\arg\max_y g(y; x, \theta)$

at is a good dependency parse?

$$y^* = \arg\max_{y \in \mathcal{Y}} g(y; x, \theta)$$

Method:

- Define features for this problem.
- Learn parameters θ from corpus data.
- Maximize objective to find best parse y^*.

First-order Scoring Function

Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores

*	Millions	on	the	coast	face	freak	storm

$$g(y; x, \theta) \quad =$$

First-order Scoring Function

Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores

* Millions on the coast face freak storm

$$g(y; x, \theta) \quad = \quad \mathit{score}(\text{coast} \rightarrow \text{the})$$

First-order Scoring Function

Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores

* Millions on the coast face freak storm

$$
\begin{aligned}
g(y; x, \theta) &= \mathit{score}(\text{coast} \rightarrow \text{the}) \\
&+ \mathit{score}(\text{on} \rightarrow \text{coast})
\end{aligned}
$$

First-order Scoring Function

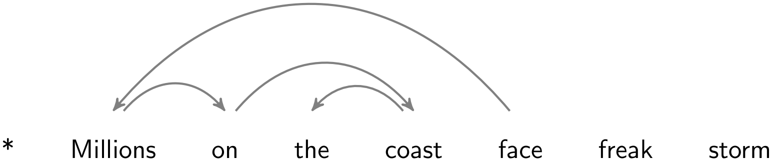
Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores

* Millions on the coast face freak storm

$$
\begin{aligned}
g(y; x, \theta) &= \mathit{score}(\text{coast} \rightarrow \text{the}) \\
&+ \mathit{score}(\text{on} \rightarrow \text{coast}) \\
&+ \mathit{score}(\text{Millions} \rightarrow \text{on})
\end{aligned}
$$

First-order Scoring Function

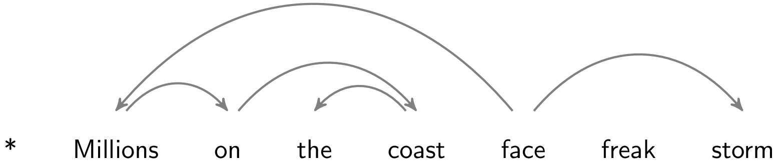
Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores



$$
\begin{aligned}
g(y; x, \theta) &= \mathit{score}(\text{coast} \rightarrow \text{the}) \\
&+ \mathit{score}(\text{on} \rightarrow \text{coast}) \\
&+ \mathit{score}(\text{Millions} \rightarrow \text{on}) \\
&+ \mathit{score}(\text{face} \rightarrow \text{millions})
\end{aligned}
$$

First-order Scoring Function

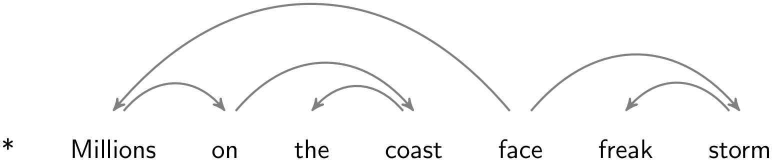
Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores



$$
\begin{aligned}
g(y; x, \theta) &= \mathit{score}(\text{coast} \rightarrow \text{the}) \\
&+ \mathit{score}(\text{on} \rightarrow \text{coast}) \\
&+ \mathit{score}(\text{Millions} \rightarrow \text{on}) \\
&+ \mathit{score}(\text{face} \rightarrow \text{millions}) \\
&+ \mathit{score}(\text{face} \rightarrow \text{storm})
\end{aligned}
$$

First-order Scoring Function

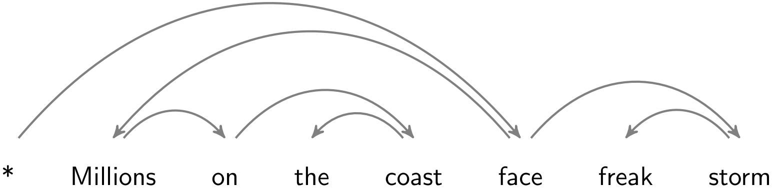
Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores



$$
\begin{aligned}
g(y; x, \theta) \quad = \quad & \mathit{score}(\text{coast} \rightarrow \text{the}) \\
+ \quad & \mathit{score}(\text{on} \rightarrow \text{coast}) \\
+ \quad & \mathit{score}(\text{Millions} \rightarrow \text{on}) \\
+ \quad & \mathit{score}(\text{face} \rightarrow \text{millions}) \\
+ \quad & \mathit{score}(\text{face} \rightarrow \text{storm}) \\
+ \quad & \mathit{score}(\text{storm} \rightarrow \text{freak})
\end{aligned}
$$

First-order Scoring Function

Scoring function $g(y; x, \theta)$ is the sum of first-order arc scores



$$
\begin{aligned}
g(y; x, \theta) \quad = \quad & \mathit{score}(\text{coast} \rightarrow \text{the}) \\
+ \quad & \mathit{score}(\text{on} \rightarrow \text{coast}) \\
+ \quad & \mathit{score}(\text{Millions} \rightarrow \text{on}) \\
+ \quad & \mathit{score}(\text{face} \rightarrow \text{millions}) \\
+ \quad & \mathit{score}(\text{face} \rightarrow \text{storm}) \\
+ \quad & \mathit{score}(\text{storm} \rightarrow \text{freak}) \\
+ \quad & \mathit{score}(* \rightarrow \text{face})
\end{aligned}
$$

Generative Model

One Possibility: $score(w_h \rightarrow w_m) = p(w_m | w_h)$

where:

- p; a multinomial distribution over words.

Intuition: A bigram-like model for arcs.

Note: Not often used (except unsupervised parsing)

Conditional Model (e.g. CRF)

Define:

$$score(w_h \rightarrow w_m) = \phi(x, \langle h, m \rangle) \cdot \theta$$

where:

- $\phi(x, \langle h, m \rangle) : \mathcal{X} \times \mathcal{A} \rightarrow \{0, 1\}^p$; a feature function
- $\theta \in R^p$; a parameter vector (assume given)
- p; number of features

Features

- Features are critical for dependency parsing performance.
- Specified as a vector of indicators.

$$\phi_{\text{NAME}}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } t_m = u \\ 0, & \text{o.w.} \end{cases}$$

- Each feature has a corresponding real-value weight.

$$\theta_{\text{NAME}} = 9.23$$

Features: Tags

$$\forall u \in \mathcal{T} \qquad \phi_{\text{TAG:M:}u}(\langle t, w \rangle, \langle h, m \rangle) = \begin{cases} 1, & \text{if } t_m = u \\ 0, & \text{o.w.} \end{cases}$$

$$\forall u \in \mathcal{T} \qquad \phi_{\text{TAG:H:}u}(\langle t, w \rangle, \langle h, m \rangle) = \begin{cases} 1, & \text{if } t_h = u \\ 0, & \text{o.w.} \end{cases}$$

$$\forall u, v \in \mathcal{T} \qquad \phi_{\text{TAG:H:M:}u:v}(\langle t, w \rangle, \langle h, m \rangle) = \begin{cases} 1, & \text{if } t_h = u \text{ and } t_m = v \\ 0, & \text{o.w.} \end{cases}$$

* Millions/N on/P the/D coast/N face/V freak/A storm/N

Features: Words

$$\forall u \in \mathcal{W} \qquad \phi_{\text{WORD:M:}u}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } w_m = u \\ 0, & \text{o.w.} \end{cases}$$

$$\forall u \in \mathcal{W} \qquad \phi_{\text{WORD:H:}u}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } w_h = u \\ 0, & \text{o.w.} \end{cases}$$

$$\forall u, v \in \mathcal{W} \qquad \phi_{\text{WORD:H:M:}u:v}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } w_h = u \text{ and } w_m = v \\ 0, & \text{o.w.} \end{cases}$$

* Millions/N on/P the/D coast/N face/V freak/A storm/N

Features: Context Tags

$$\forall u \in \mathcal{T}^4 \qquad \phi_{\text{CON}:-1:-1:u}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } t_{h-1} = u_1 \text{ and } t_h = u_2 \\ & \text{and } t_{m-1} = u_3 \text{ and } t_m = u_4 \\ 0, & \text{o.w.} \end{cases}$$

$$\forall u \in \mathcal{T}^4 \qquad \phi_{\text{CON}:1:-1:u}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } t_{h+1} = u_1 \text{ and } t_h = u_2 \\ & \text{and } t_{m-1} = u_3 \text{ and } t_m = u_4 \\ 0, & \text{o.w.} \end{cases}$$

* Millions/N on/P the/D coast/N face/V freak/A storm/N

Features: Between Tags

$$\forall u \in \mathcal{T} \qquad \phi_{\mathrm{BET}:u}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } t_i = u \text{ for } i \text{ between } h \text{ and } m \\ 0, & \text{o.w.} \end{cases}$$

*   Millions/N   on/P   the/D   coast/N   face/V   freak/A   storm/N

Features: Direction

$$\phi_{\text{RIGHT}}(\langle t, w \rangle, \langle h, m \rangle) = \begin{cases} 1, & \text{if } h > m \\ 0, & \text{o.w.} \end{cases}$$

$$\phi_{\text{LEFT}}(\langle t, w \rangle, \langle h, m \rangle) = \begin{cases} 1, & \text{if } h < m \\ 0, & \text{o.w.} \end{cases}$$

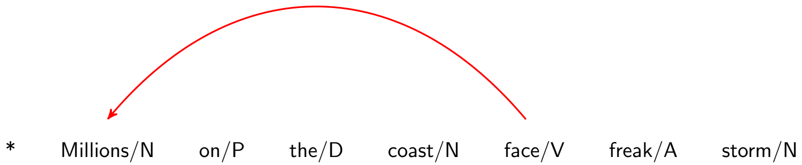
* Millions/N on/P the/D coast/N face/V freak/A storm/N

Features: Length

$$\phi_{\mathrm{LEN:2}}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } |h - m| > 2 \\ 0, & \text{o.w.} \end{cases}$$

$$\phi_{\mathrm{LEN:5}}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } |h - m| > 5 \\ 0, & \text{o.w.} \end{cases}$$

$$\phi_{\mathrm{LEN:10}}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } |h - m| > 10 \\ 0, & \text{o.w.} \end{cases}$$



Features: Backoffs and Combinations

- Additionally include backoff.

$$\forall u \in \mathcal{T}^3 \qquad \phi_{\text{CON:}-1:u}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if } t_{h-1} = u_1 \text{ and } t_h = u_2 \\ & \text{and } t_m = u_3 \\ 0, & \text{o.w.} \end{cases}$$

- As well as combination features.

$$\forall u \in \mathcal{W} \qquad \phi_{\text{LEN:2:DIR:LEFT:TAG:M:}u}(\langle t, w\rangle, \langle h, m\rangle) = \begin{cases} 1, & \text{if all on} \\ 0, & \text{o.w.} \end{cases}$$

First-Order Results

Model	Accuracy
NoPOSContextBetween	86.0
NoEdge	87.3
NoAttachmentOrDistance	88.1
NoBiLex	90.6
Full	90.7

From McDonald (2006)

What's left

- Define features for this problem.
- Learn parameters θ from corpus data.
- Maximize objective to find best parse y^*.

What's left

- Define features for this problem.
- Learn parameters θ from corpus data.
- Maximize objective to find best parse y^*.

Downside: Higher-order models make inference more difficult

$$y^* = \arg\max_{y \in \mathcal{Y}} g(y; x, \theta)$$

Parsing

Goal: Finding the best parse.

$$y^* = \arg\max_{y \in \mathcal{Y}} g(y; x, \theta)$$

Graph Algorithms

Algorithm 2: Use graph algorithms for parsing.

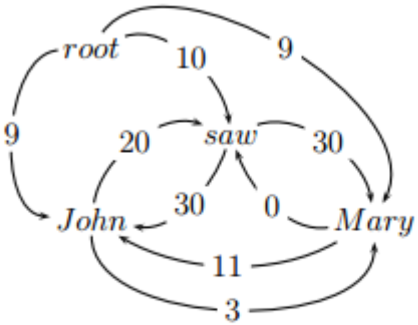
Graph Algorithms

Algorithm 2: Use graph algorithms for parsing.

Find the maximum *directed* spanning tree.

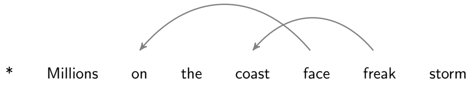
- Chou-Liu-Edmonds Algorithm $O(n^3)$
- Tarjan's Extension $O(n^2)$

Maximum Directed Spanning Tree Algorithm



Issues with MST Algorithm

- Allows non-projective parses.



  - Good for some languages.

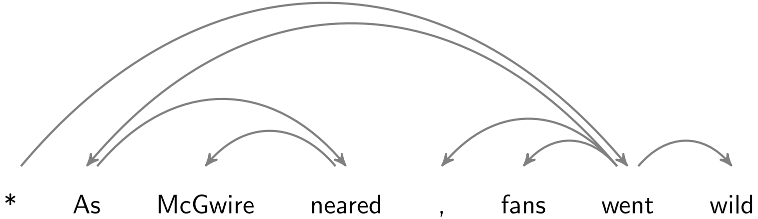
- Cannot incorporate higher-order parts.
  - Problem becomes NP-Hard.

Dynamic Programming for Parsing

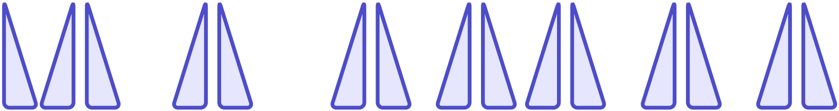
Algorithm 3: Use a specialized dynamic programming algorithm.

- The Eisner algorithm (1996) for bilexical parsing.
- Use split-head trick. Handle left and right dependencies separately.

Dependency Parsing New Example

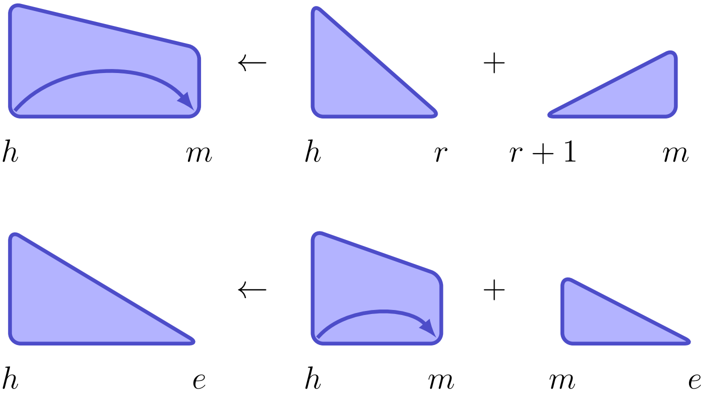


Base Case

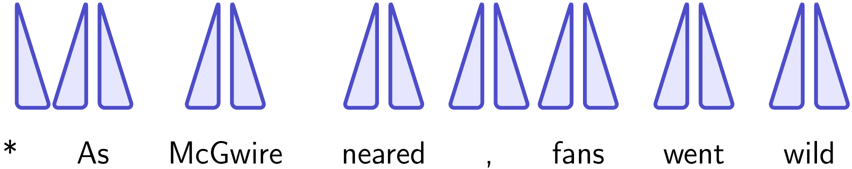


* As McGwire neared , fans went wild

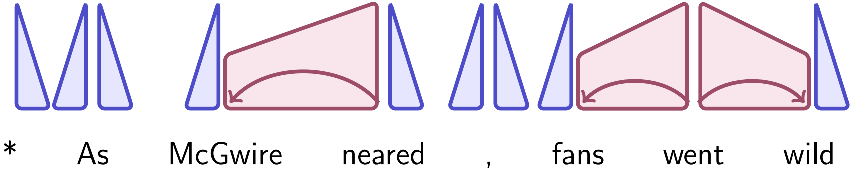
Dependency Parsing Algorithm - First-order Model



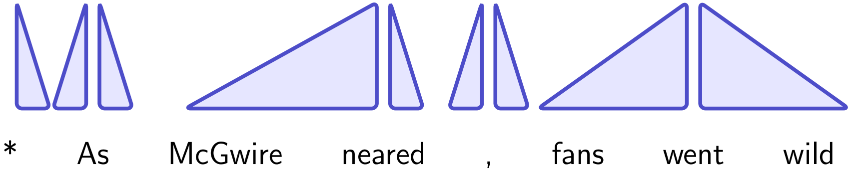
Parsing



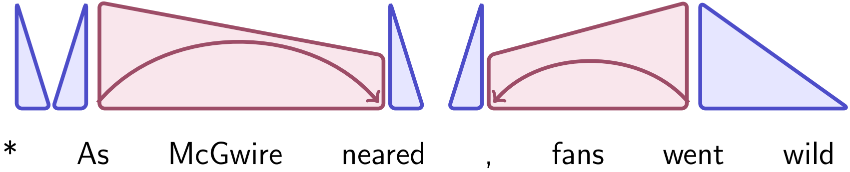
Parsing



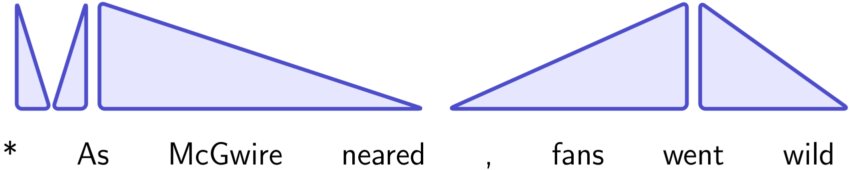
Parsing



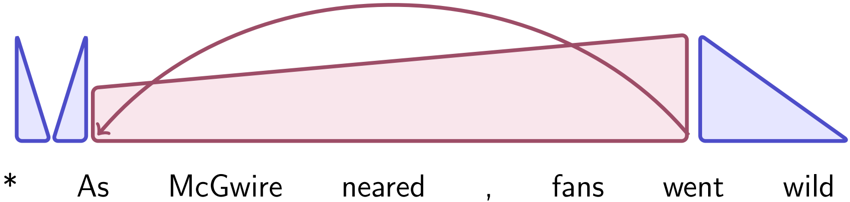
Parsing



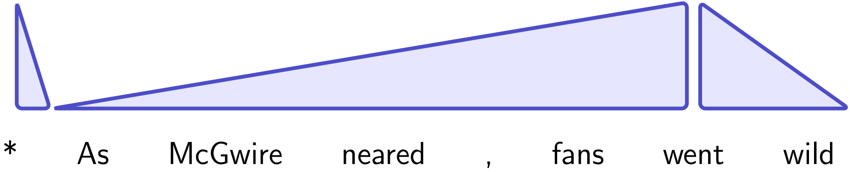
Parsing



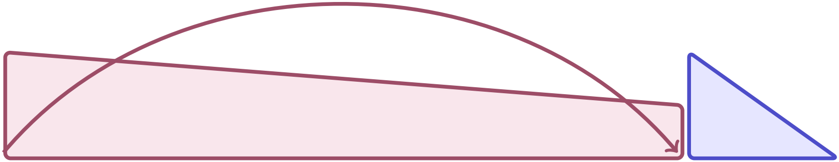
Parsing



Parsing

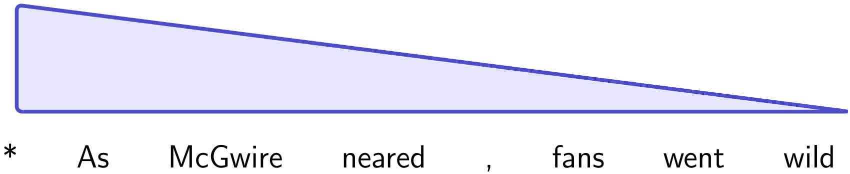


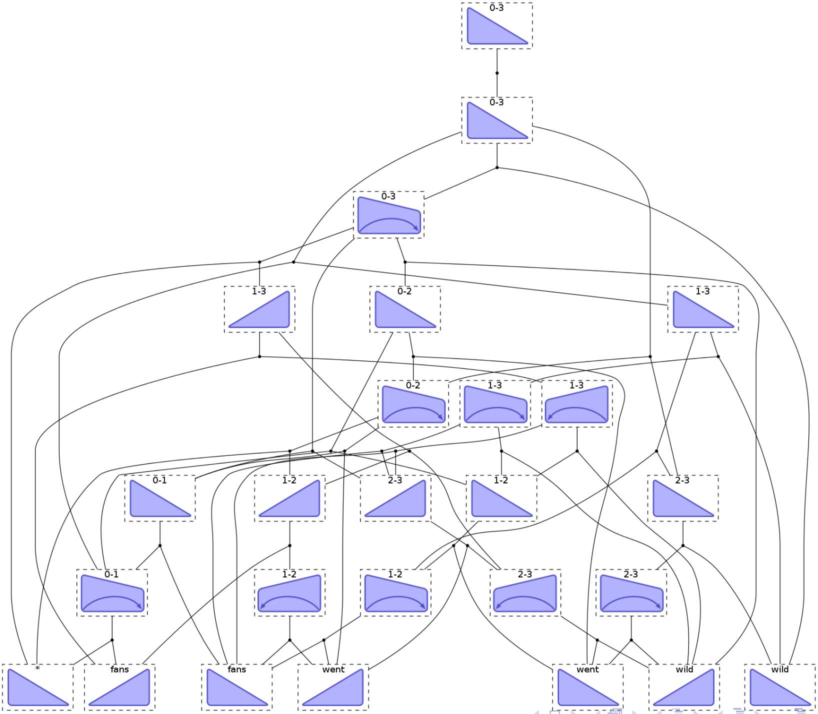
Parsing



* As McGwire neared , fans went wild

Parsing




0-3
0-3
0-3
1-3
0-2
1-3
0-2
1-3
1-3
0-1
1-2
2-3
1-2
2-3
0-1
1-2
1-2
2-3
2-3
*
fans
fans
went
went
wild
wild

Algorithm Key

- L; left-facing item
- R; right-facing item
- C; completed item (triangle)
- I; incomplete item (trapezoid)

Algorithm

Initialize:

for i in 0 . . . n **do**

$\pi[\mathrm{C}, \mathrm{L}, i, i] = 0$

$\pi[\mathrm{C}, \mathrm{R}, i, i] = 0$

$\pi[\mathrm{I}, \mathrm{L}, i, i] = 0$

$\pi[\mathrm{I}, \mathrm{R}, i, i] = 0$

Inner Loop:

for k in 1 . . . n **do**

for s in 0 . . . n **do**

$t \leftarrow k + s$

if $t \geq n$ **then** break

$\pi[\mathrm{I}, \mathrm{L}, s, t] = \max_{r \in s \ldots t-1} \pi[\mathrm{C}, \mathrm{R}, s, r] + \pi[\mathrm{C}, \mathrm{L}, r+1, t]$

$\pi[\mathrm{I}, \mathrm{R}, s, t] = \max_{r \in s \ldots t-1} \pi[\mathrm{C}, \mathrm{R}, s, r] + \pi[C, \mathrm{L}, r+1, t]$

$\pi[\mathrm{C}, \mathrm{L}, s, t] = \max_{r \in s \ldots t-1} \pi[\mathrm{C}, \mathrm{L}, s, r] + \pi[\mathrm{I}, \mathrm{L}, r, t]$

$\pi[\mathrm{C}, \mathrm{R}, s, t] = \max_{r \in s+1 \ldots t} \pi[\mathrm{I}, \mathrm{R}, s, r] + \pi[\mathrm{C}, \mathrm{R}, r, t]$

return $\pi[\mathrm{C}, \mathrm{R}, 0, n]$

Graph-based parsing algorithm

- Begin with a tagged sentence (can use a POS-tagger)

Graph-based parsing algorithm

- Begin with a tagged sentence (can use a POS-tagger)
- Extract a set of "parts"
  - For a first-order model, each part is a (h, m) pair ($O(n^2)$ parts)
  - For a second-order model, each part is a $(h, m1, m2)$ tuple ($O(n^3)$ parts)

Graph-based parsing algorithm

- Begin with a tagged sentence (can use a POS-tagger)
- Extract a set of "parts"
  - For a first-order model, each part is a (h, m) pair ($O(n^2)$ parts)
  - For a second-order model, each part is a $(h, m1, m2)$ tuple ($O(n^3)$ parts)
- Calculate a score for each part (using feature-extractor ϕ and parameters θ)

Graph-based parsing algorithm

- Begin with a tagged sentence (can use a POS-tagger)
- Extract a set of "parts"
  - For a first-order model, each part is a (h, m) pair ($O(n^2)$ parts)
  - For a second-order model, each part is a $(h, m1, m2)$ tuple ($O(n^3)$ parts)
- Calculate a score for each part (using feature-extractor ϕ and parameters θ)
- Find a valid parse tree that is composed of the best parts.
  - using Chu-Liu-Edmunds (for first-order non-projective) ($O(n^2)$)
  - using a dynamic-programming algorithm (for first- and second-order projective) ($O(n^3)$)

Graph-based parsing algorithm

- Begin with a tagged sentence (can use a POS-tagger)
- Extract a set of "parts"
  - For a first-order model, each part is a (h, m) pair ($O(n^2)$ parts)
  - For a second-order model, each part is a $(h, m1, m2)$ tuple ($O(n^3)$ parts)
- Calculate a score for each part (using feature-extractor ϕ and parameters θ)
- Find a valid parse tree that is composed of the best parts.
  - using Chu-Liu-Edmunds (for first-order non-projective) ($O(n^2)$)
  - using a dynamic-programming algorithm (for first- and second-order projective) ($O(n^3)$)

Does this remind you of anything?

Inference

- Full algorithms $O(n^3)$.
- Much faster than standard lexicalized parsing.
- Other ways to further improve speed.

Training - setting values for θ

Note: we need values such that $g(y; x, \theta)$ of gold tree y is larger than $g(y'; x, \theta)$ for all other trees y'.

Perceptron Sketch: Part 1

- $(x_1, y_1) \ldots (x_n, y_n)$; training data
- Gold features

$$\sum_{a \in \mathcal{A}: y(a)=1} \phi(x_i, a)$$

Idea: Increase value (in θ) of gold features.

Perceptron Sketch: Part 2

- Best-scoring structure

$$z_i = \arg\max_{z \in \mathcal{Y}} g(z; x, \theta)$$

- Best-scoring structure features

$$\sum_{a \in \mathcal{A} : z(a) = 1} \phi(x_i, a)$$

Idea: Decrease value (in θ) of *wrong* best-scoring features

Perceptron Algorithm

$\theta \leftarrow 0$

for t = 1 … T, i = 1 … n **do**

$$z_i = \arg\max_{y \in \mathcal{Y}} g(y; x_i, \theta)$$

$$gold \leftarrow \sum_{a \in \mathcal{A}: y_i(a)=1} \phi(x_i, a)$$

$$best \leftarrow \sum_{a \in \mathcal{A}: z_i(a)=1} \phi(x_i, a)$$

$$\theta \leftarrow \theta + gold - best$$

return θ

Theory

- If possible, perceptron will separate the correct structure from the incorrect structure.
- That is, it will find a θ that assigns y_i a higher score than other $y \in \mathcal{Y}$ for each example.

Practical Training Considerations

- Training requires solving inference many times.
- Often times computing feature values is time consuming.
- In practice, averaged perceptron variant preferred (Collins, 2002).

Conclusion

Method:

- Define features for this problem.
- Learn parameters θ from corpus data.
- Maximize objective to find best parse y^*.

Structured prediction framework, applicable to many problems.

Transition-based parsing

Transition-based (greedy) parsing

1. Start with an unparsed sentence.
2. Apply **locally-optimal** actions until sentence is parsed.

Transition-based (greedy) parsing

1. Start with an unparsed sentence.
2. Apply **locally-optimal** actions until sentence is parsed.
3. Use whatever features you want.
4. Surprisingly accurate.
5. Can be extremely fast.

Intro to Transition-based Dependency Parsing

An abstract machine composed of a **stack** and a **buffer**.

Machine is initialized with the words of a sentence.

A set of actions process the words by moving them from buffer to stack, removing them from the stack, or adding links between them.

A specific set of actions define a transition system.

The Arc-Eager Transition System

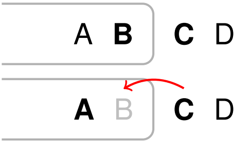
- **Shift** move first word from buffer to stack.
  (pre: Buffer not empty.)

A | **B** C D

A **B** | **C** D

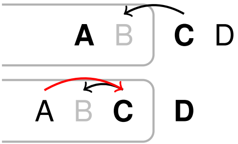
The Arc-Eager Transition System

- **Shift** move first word from buffer to stack.
  (pre: Buffer not empty.)
- **LeftArc**$_{label}$ make first word in buffer head of top of stack, pop the stack.
  (pre: Stack not empty. Top of stack does not have a parent.)



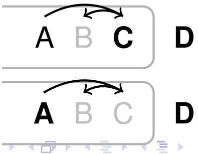
The Arc-Eager Transition System

- **SHIFT** move first word from buffer to stack.
  (pre: Buffer not empty.)
- **LEFTARC**$_{label}$ make first word in buffer head of top of stack, pop the stack.
  (pre: Stack not empty. Top of stack does not have a parent.)
- **RIGHTARC**$_{label}$ make top of stack head of first in buffer, move first in buffer to stack.
  (pre: Buffer not empty.)



The Arc-Eager Transition System

- **SHIFT** move first word from buffer to stack.
  (pre: Buffer not empty.)
- **LEFTARC**$_{label}$ make first word in buffer head of top of stack, pop the stack.
  (pre: Stack not empty. Top of stack does not have a parent.)
- **RIGHTARC**$_{label}$ make top of stack head of first in buffer, move first in buffer to stack.
  (pre: Buffer not empty.)
- **REDUCE** pop the stack
  (pre: Stack not empty. Top of stack has a parent.)



Parsing Example

She ate pizza with pleasure

Parsing Example

She ate pizza with pleasure

Parsing Example

She ate pizza with pleasure

Parsing Example

She ate pizza with pleasure

Parsing Example

She ate pizza with pleasure

Parsing Example

She ate pizza with pleasure

Parsing Example

She ate pizza with pleasure

Parsing Example

She ate pizza with pleasure

Parsing Example

Parsing Example

She ate pizza with pleasure

Parsing Example

What do we know about the arc-eager transition system?

- Every sequence of actions result in a valid projective structure.
- Every projective tree is derivable by (at least one) sequence of actions.
- Given a tree, finding a sequence of actions for deriving it. ("oracle")

we know these things also for the
arc-standard, arc-hybrid and other transition systems

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

This knowledge is quite powerful

Parsing with an oracle sequence

sequence $\leftarrow$ **oracle(sentence, tree)**
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() **do**
    action $\leftarrow$ sequence.next()
    configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

| She | ate pizza with pleasure |

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She	ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She | ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← **sequence.next()**
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

Sh Left Sh Right Re Right Right Re Re Re

She | ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence $\leftarrow$ oracle(sentence, tree)
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() **do**
    action $\leftarrow$ sequence.next()
    configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate | pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate | pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza | with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza | with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← **sequence.next()**
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

"She ate pizza with pleasure"

Sh Left Sh Right Re Right Right Re Re Re

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with | pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

"She ate pizza with pleasure"

Sh Left Sh Right Re Right Right Re Re Re

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

She ate pizza with pleasure

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

"She ate pizza with pleasure"

Sh Left Sh Right Re Right Right Re Re Re

This knowledge is quite powerful

Parsing without an oracle

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

This knowledge is quite powerful

Parsing without an oracle

start with weight vector w
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() **do**
    action $\leftarrow$ predict(w, ϕ(configuration))
    configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

This knowledge is quite powerful

Parsing without an oracle

summarize the configuration
as a feature vector

start with weight vector w
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() **do**
    action $\leftarrow$ predict(w, ϕ(configuration))
    configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

This knowledge is quite powerful

Parsing without an oracle

summarize the configuration
as a feature vector

start with weight vector w
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() **do**
    action $\leftarrow$ predict(w, ϕ(configuration))
    configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

predict the action based on the features

This knowledge is quite powerful

Parsing without an oracle

summarize the configuration as a feature vector

start with weight vector w
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() **do**
    action $\leftarrow$ predict(w, ϕ(configuration))
    configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

predict the action based on the features

need to learn the correct weights

This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()
    configuration ← configuration.apply(action)

This knowledge is quite powerful

Learning a parser (batch)

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() **do**
    action ← sequence.next()

    configuration ← configuration.apply(action)

This knowledge is quite powerful

Learning a parser (batch)

```
training_set ← []
for sentence,tree pair in corpus do
    sequence ← oracle(sentence, tree)
    configuration ← initialize(sentence)
    while not configuration.IsFinal() do
        action ← sequence.next()
        features ← φ(configuration)
        training_set.add(features, action)
        configuration ← configuration.apply(action)
train a classifier on training_set
```

This knowledge is quite powerful

Learning a parser (batch)

training_set $\leftarrow$ []
for sentence,tree pair in corpus **do**
    sequence $\leftarrow$ oracle(sentence, tree)
    configuration $\leftarrow$ initialize(sentence)
    while not configuration.IsFinal() **do**
        action $\leftarrow$ sequence.next()
        features $\leftarrow$ ϕ(configuration)
        training_set.add(features, action)
        configuration $\leftarrow$ configuration.apply(action)
train a classifier on training_set

This knowledge is quite powerful

Learning a parser (online)

```
training_set ← []
for sentence,tree pair in corpus do
    sequence ← oracle(sentence, tree)
    configuration ← initialize(sentence)
    while not configuration.IsFinal() do
        action ← sequence.next()
        features ← φ(configuration)
        training_set.add(features, action)
        configuration ← configuration.apply(action)
train a classifier on training_set
```

This knowledge is quite powerful

Learning a parser (online)

```
w ← 0
for sentence,tree pair in corpus do
    sequence ← oracle(sentence, tree)
    configuration ← initialize(sentence)
    while not configuration.IsFinal() do
        action ← sequence.next()
        features ← φ(configuration)
        predicted ← predict(w, φ(configuration))
        if predicted ≠ action then
            w.update(φ(configuration), action, predicted)
        configuration ← configuration.apply(action)
return w
```

This knowledge is quite powerful

Learning a parser (online)

$w \leftarrow 0$
for sentence,tree pair in corpus **do**
    sequence $\leftarrow$ oracle(sentence, tree)
    configuration $\leftarrow$ initialize(sentence)
    while not configuration.IsFinal() **do**
        action $\leftarrow$ sequence.next()
        features $\leftarrow$ ϕ(configuration)
        predicted $\leftarrow$ predict(w, ϕ(configuration))
        if predicted $\neq$ action **then**
            w.update(ϕ(configuration), action, predicted)
        configuration $\leftarrow$ configuration.apply(action)
return w

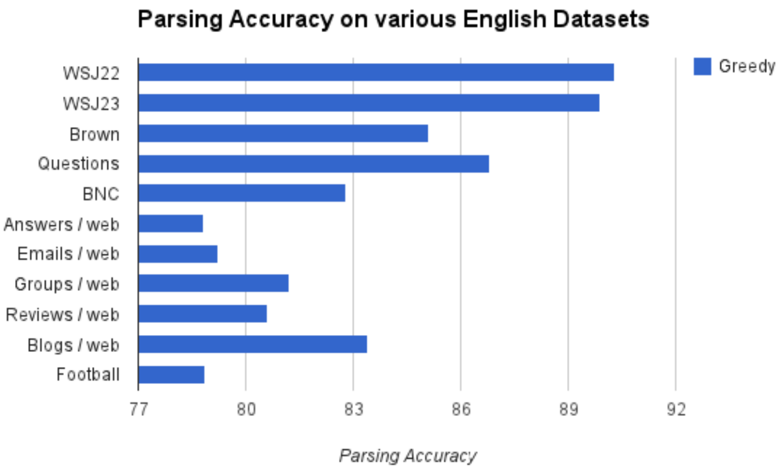
This knowledge is quite powerful

Parsing time

configuration $\leftarrow$ initialize(sentence)
while not configuration.isFinal() **do**
    action $\leftarrow$ predict(w, ϕ(configuration))
    configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

In short

- Summarize configuration by a set of features.
- Learn the best action to take at each configuration.
- Hope this generalizes well.



Parsing Accuracy on various English Datasets
Greedy
WSJ22
WSJ23
Brown
Questions
BNC
Answers / web
Emails / web
Groups / web
Reviews / web
Blogs / web
Football
77
80
83
86
89
92
Parsing Accuracy

Transition Based Parsing

- A different approach.
- Very common.
- Can be as accurate as first-order graph-based parsing.
  - Higher-order graph-based are still better.
- Easy to implement.
- Very fast. $(O(n))$
- Can be improved further:
  - Easy-first
  - Dynamic oracle
  - Beam Search

Neural Networks

Neural-network (deep learning) based approaches

- Both graph based and transition-based models benefit from the move to neural networks.
- Same over-all approach and algorithm as before, but:
  - Replace classifier from linear to MLP.
  - Use pre-trained word embeddings.
  - Replace feature-extractor with Bi-LSTM.
- Now exploring;

Neural-network (deep learning) based approaches

- Both graph based and transition-based models benefit from the move to neural networks.
- Same over-all approach and algorithm as before, but:
  - Replace classifier from linear to MLP.
  - Use pre-trained word embeddings.
  - Replace feature-extractor with Bi-LSTM.
- Now exploring;
  - Semi-supervised learning.
  - Multi-task learning objectives.

Neural-network (deep learning) based approaches

- Both graph based and transition-based models benefit from the move to neural networks.
- Same over-all approach and algorithm as before, but:
  - Replace classifier from linear to MLP.
  - Use pre-trained word embeddings.
  - Replace feature-extractor with Bi-LSTM.
- Now exploring;
  - Semi-supervised learning.
  - Multi-task learning objectives.
  - **Out of domain parsing.**

Hybrid Approaches

Hybrid-approaches

- Different parsers have different strengths.

$\Rightarrow$ Combine several parsers.

Hybrid-approaches

- Different parsers have different strengths.

$\Rightarrow$ Combine several parsers.

Stacking

- Run parser A.
- Use tree from parser A to add features to parser B.

Hybrid-approaches

- Different parsers have different strengths.

$\Rightarrow$ Combine several parsers.

Stacking

- Run parser A.
- Use tree from parser A to add features to parser B.

Voting

- Parse the sentence with k different parsers.
- Each parser "votes" on its dependency arcs.
- Run first-order graph-parser to find tree with best arcs according to votes.

Semi-supervised-approaches

- We only see very few words (and word-pairs) in training data.
- If we know (eat, carrot) is a good pair, what do we know about (eat, tomato)?
- Nothing, if the pair is not in our training data!

$\Rightarrow$ Use unlabeled data.

Semi-supervised-approaches

- We only see very few words (and word-pairs) in training data.
- If we know (eat, carrot) is a good pair, what do we know about (eat, tomato)?
- Nothing, if the pair is not in our training data!

$\Rightarrow$ Use unlabeled data.

Semi-supervised-approaches

- We only see very few words (and word-pairs) in training data.
- If we know (eat, carrot) is a good pair, what do we know about (eat, tomato)?
- Nothing, if the pair is not in our training data!

$\Rightarrow$ Use unlabeled data.

Cluster Features

- Represent words as context vectors.
- Define a similarity measure between vectors.
- Use a **clustering algorithm** to cluster the words.
- We hope that:
  - (eat, drink, devour,. . .) are in the same cluster.
  - (tomato, carrot, pizza, . . .) are in the same cluster.
- Use clusters as additional features to the parser.

Semi-supervised-approaches

- We only see very few words (and word-pairs) in training data.
- If we know (eat, carrot) is a good pair, what do we know about (eat, tomato)?
- Nothing, if the pair is not in our training data!

$\Rightarrow$ Use unlabeled data.

Cluster Features

- Represent words as context vectors.
- Define a similarity measure between vectors.
- Use a **clustering algorithm** to cluster the words.
- We hope that:
  - (eat, drink, devour,. . .) are in the same cluster.
  - (tomato, carrot, pizza, . . .) are in the same cluster.
- Use clusters as additional features to the parser.
  - This works well (better?) also for POS-tagging, NER.

Available Software

There are many parsers available for download, including:

Constituency (PCFG)

- Stanford Parser (can produce also dependencies)
- Berkeley Parser
- Charniak Parser
- Collins Parser

Dependency

- RBGParser, TurboParser (graph based)
- ZPar (transition+beam)
- ClearNLP (many variants)
- EasyFirst (my own)
- Bist Parser (from BGU lab, biLSTM, graph + transition)
- **SpaCy** (nice API, super fast!!)

Summary

Dependency Parsers

- Conversion from Constituency
- Graph-based
- Transition-based
- Hybrid / Ensemble
- Semi-supervised (cluster features)