Dependency Parsing

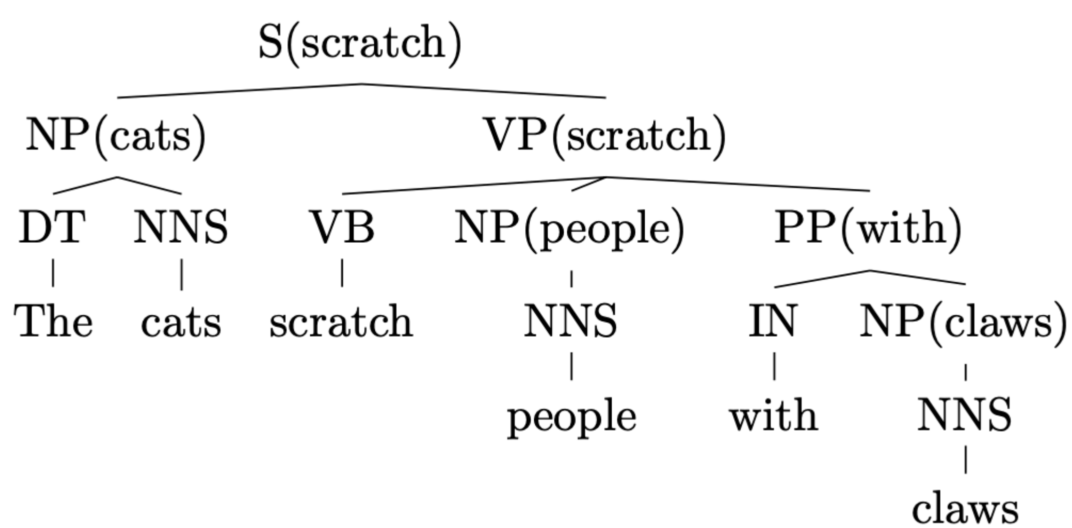
Chapter 11

Felix Dietrich 2020/06/19

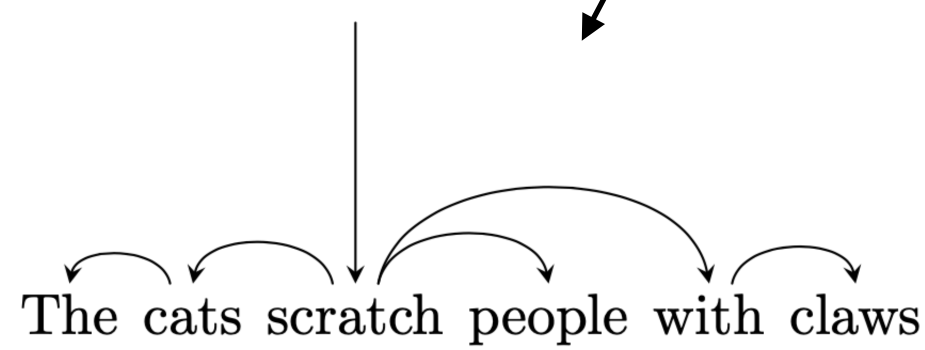
Dependency Grammar

- Motivated by lexicalized context-free grammars
- Tries to improve attachment ambiguities
- More light-weight structure
- **Universal Dependencies** project
  - More than 100 dependency treebanks for more than 60 languages

Dependency parse



(a) lexicalized constituency parse

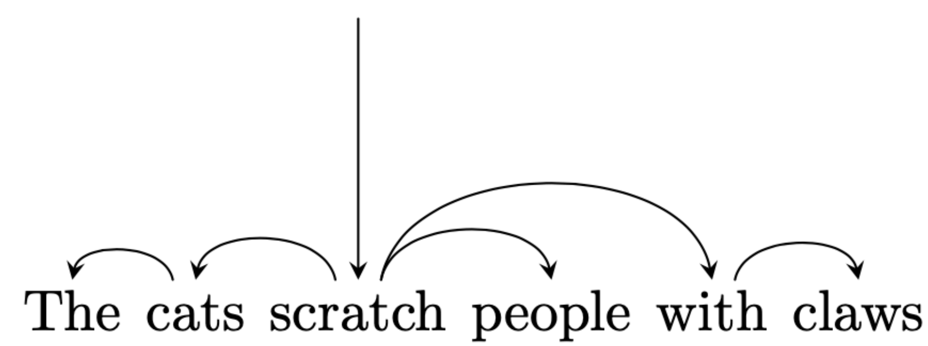


(b) unlabeled dependency tree

Dependency Grammar (cont.)

- Relationships modeled as directed graph (**dependency graph**)
- Vertices are the words and a special root vertex
- Edge (i, j) from **head** i to **dependent** j
  - Describes **syntactic dependencies**
  - Can be derived from a lexicalized constituency parse
- Exactly one incoming edge for each word (root has no incoming edge)

- Properties of the dependency graph
  - Weakly connected
  - No cycles
  - **Spanning tree** if directed edges are replaced by undirected edges



(b) unlabeled dependency tree

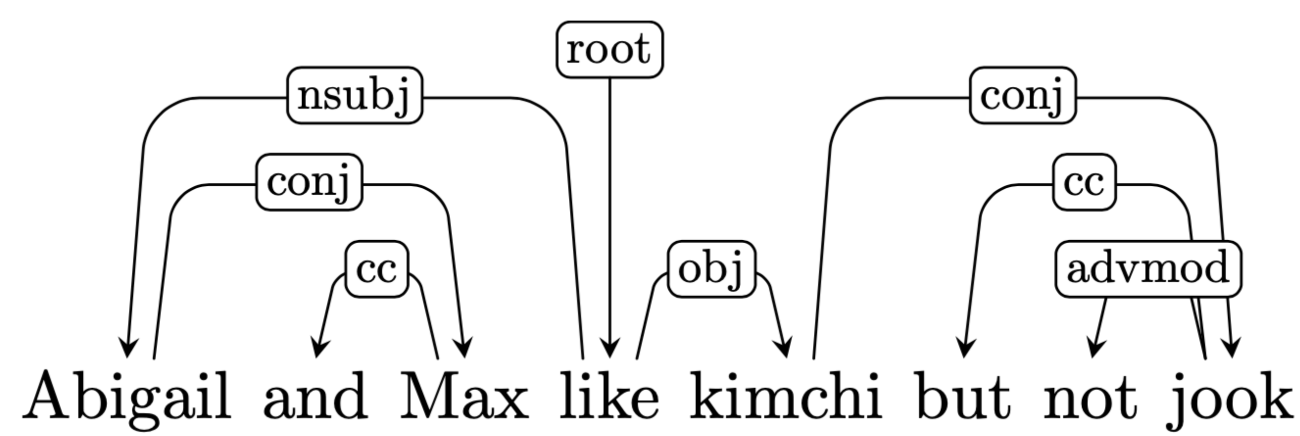
Heads and Dependents

<u>How to choose the head? (possible criteria)</u>

- Head sets syntactic category of the construction
  - E.g. Nouns as heads of noun phrases, verbs are heads of verb phrases
- **Modifier** (dependent) may be optional while head is mandatory
  - E.g. *cats scratch people with claws*, subtrees *cats scratch* and *cats scratch people* are grammatical sentences, but *with claws* is not
- Head determines the morphological form of the modifier
  - E.g. in languages with gender agreements, the gender of the noun determines the gender of the adjectives and determiners
- Edges should first connect content words, and then connect function words

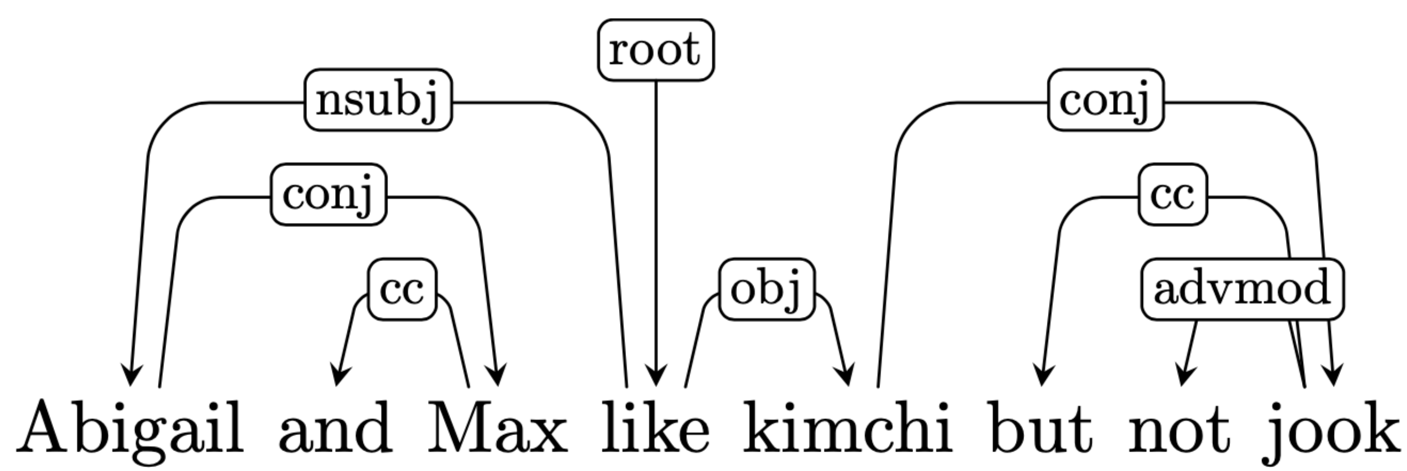
Heads and Dependents (cont.)

- Relationships are modeled as asymmetric
- Not all relations are asymmetric
  - **Example: Coordination (symmetric)**
  - *Abigail and Max like kimchi* with coordinated noun phrase *Abigail and Max*
  - How to choose the heads in the coordinated noun phrase?
  - Choosing *Abigail* or *Max* would be arbitrary
  - Choosing *and* goes against the principle of linking content words first
  - Universal Dependencies arbitrarily chooses the left-most item as head and uses of *conj* (=conjoined) and *cc* (=coordinating conjunction) labels



Labeled Dependencies

- **Labels** of edges indicate the nature of the syntactic relations
- What are the children of *like*? Who likes what?
  - *Abigail and Max* is *nsubj* (=noun subject) of verb *like*
  - *kimchi but not jook* is *obj* (=object) from verb *like*
  - Negation *not* is *advmod* (=adverbial modifier) on the noun *jook*



Another Example Dependency Parse

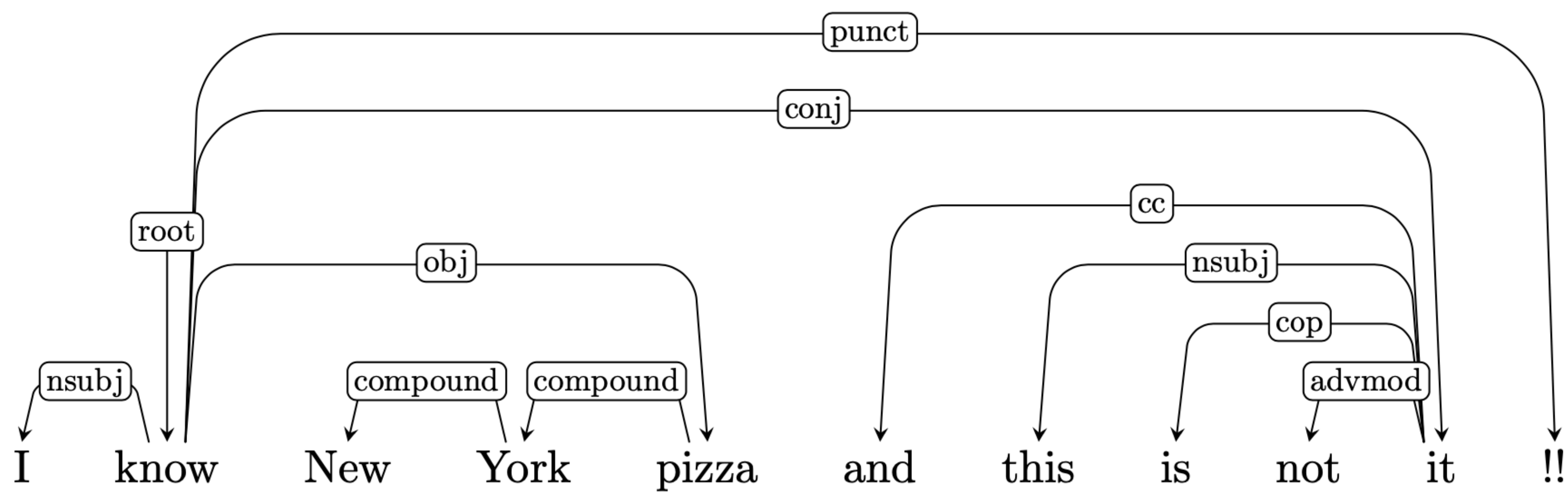
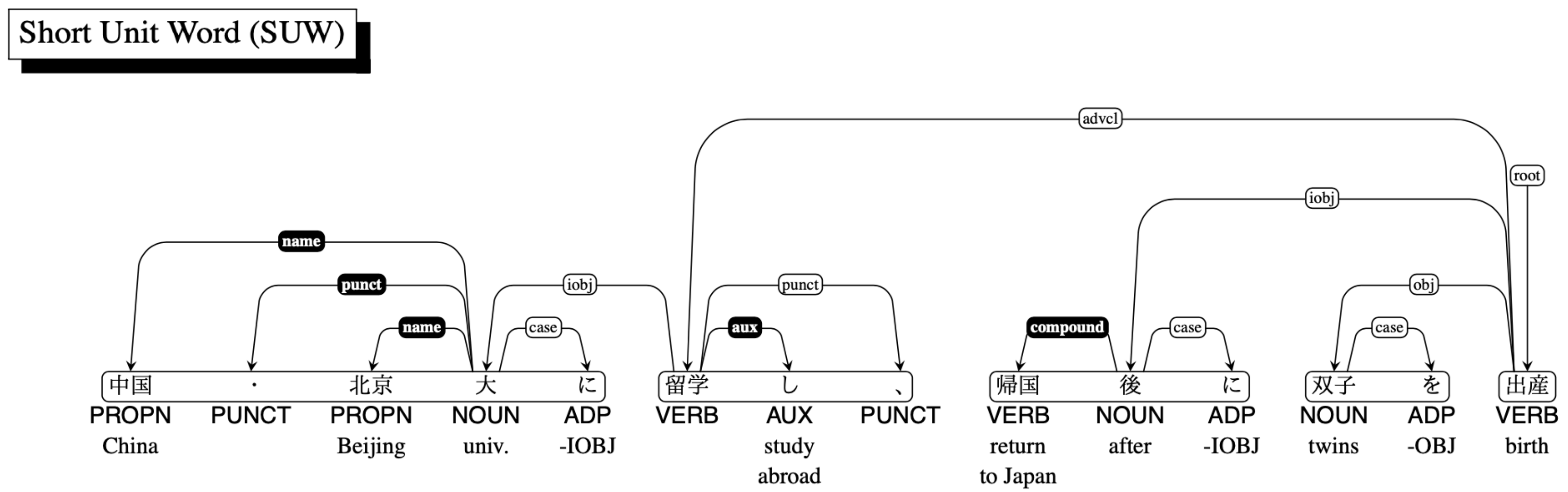
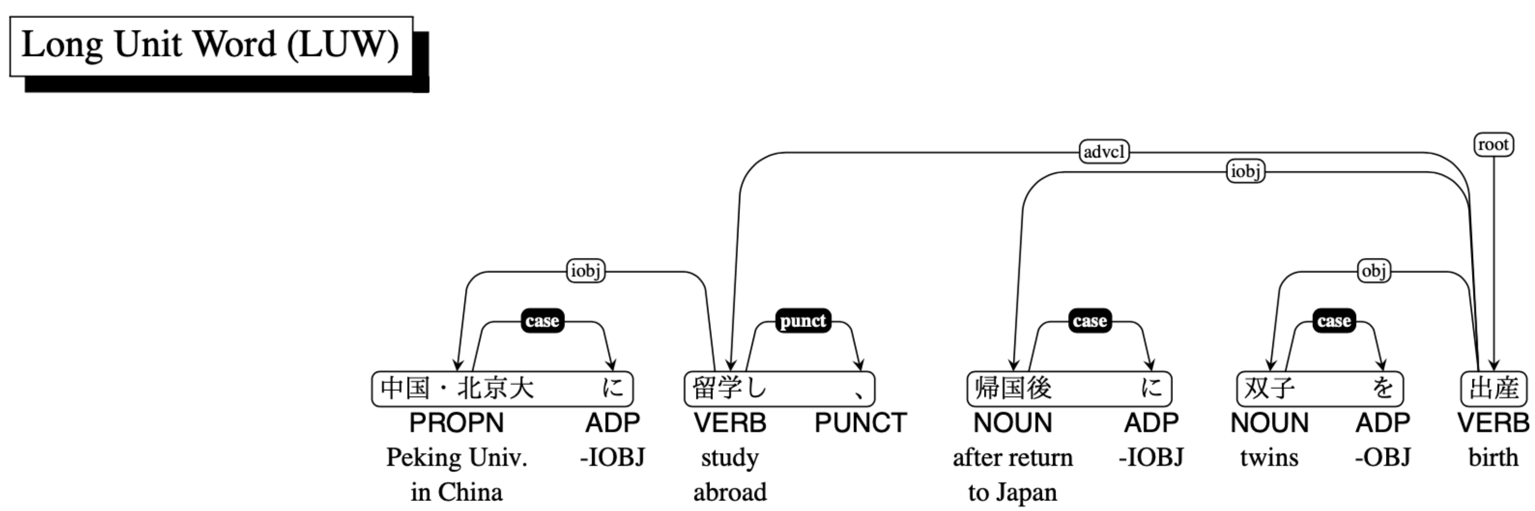
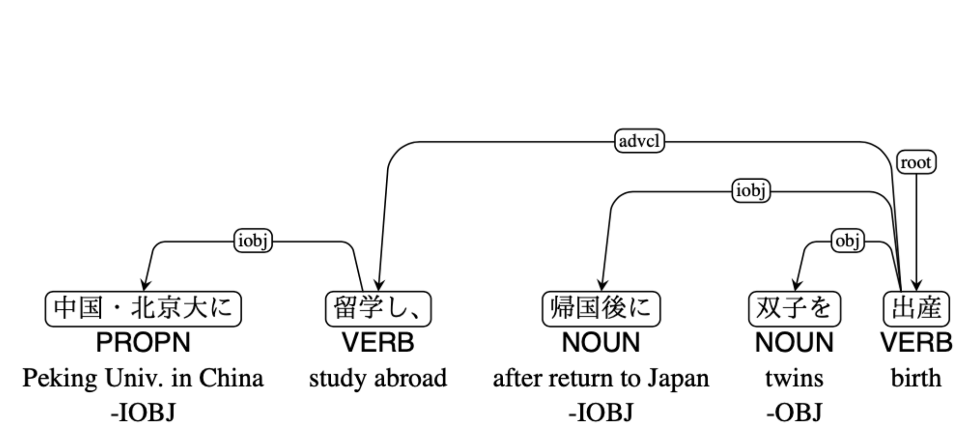


Figure 11.3: A labeled dependency parse from the English UD Treebank (reviews-361348-0006)

Example Japanese Dependency Parse







She studied in Peking University, and delivered twins when she returned to Japan.

Labels are not that important
in Japanese bunsetsu parsing

Source: Asahara, Masayuki, et al. "Universal Dependencies Version 2 for Japanese."

Dependency Subtrees and Constituents

- Dependency trees hides information present of CFG parse
- Often no meaningful difference between analyses
- Dependency parses can be flat:
  - E.g. *Abigail was reluctantly giving Max kimchi* with head *giving*

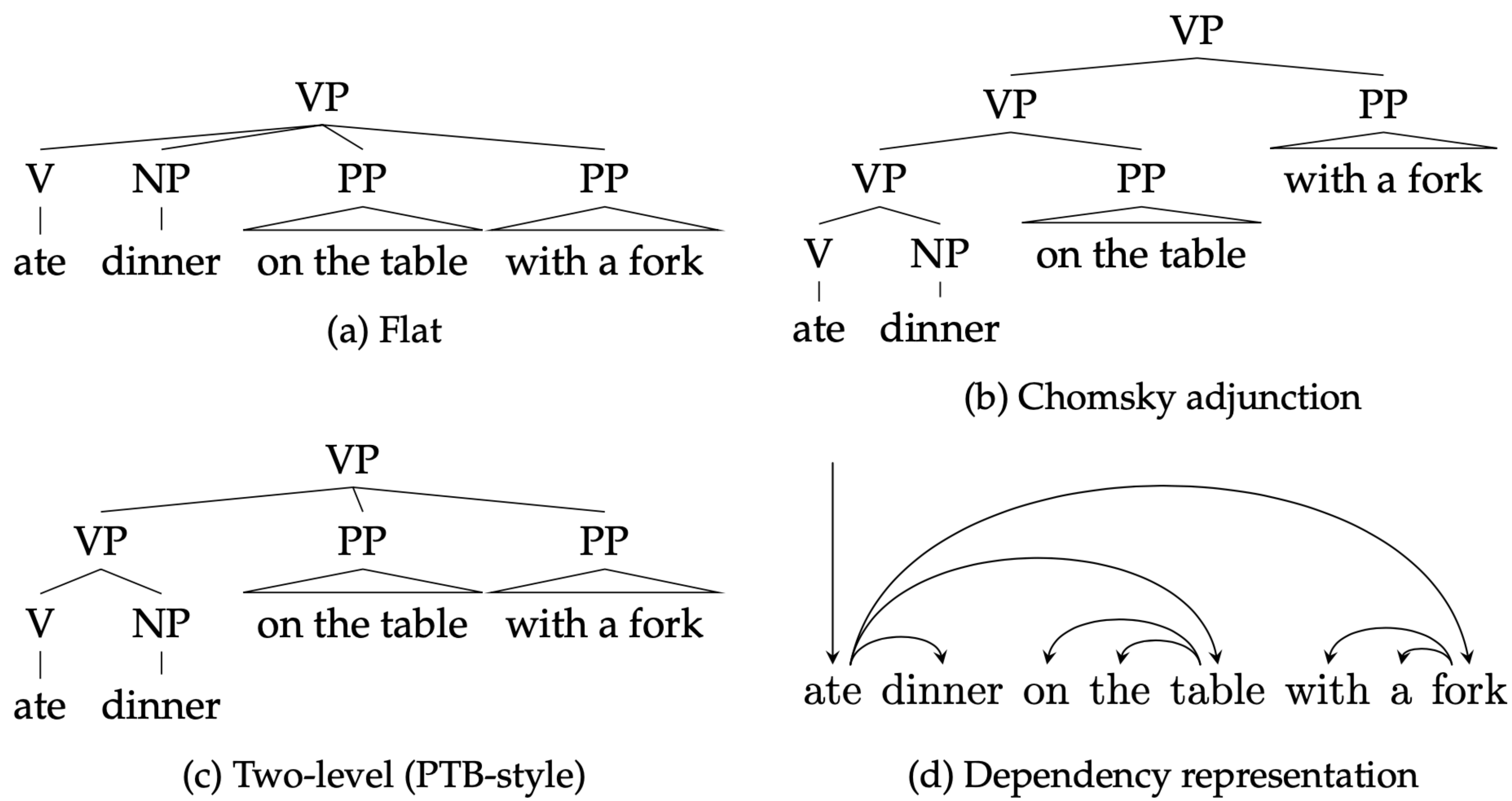


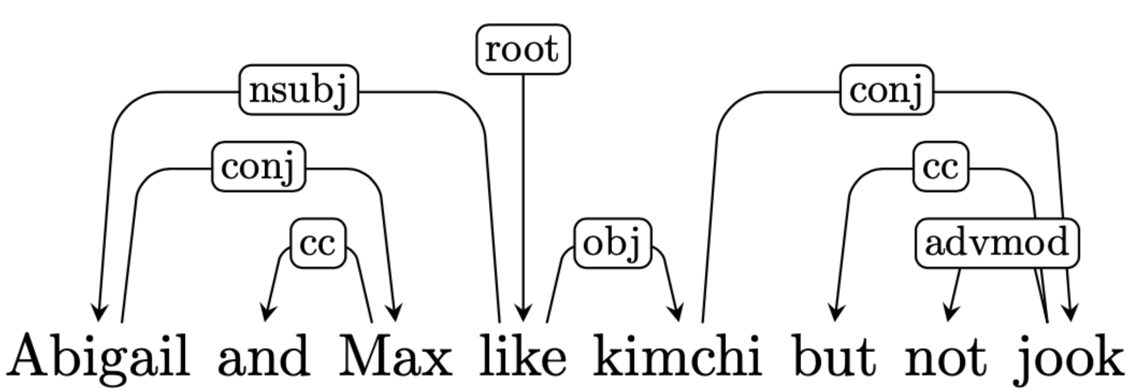
Figure 11.4: The three different CFG analyses of this verb phrase all correspond to a single dependency structure.

Projectivity

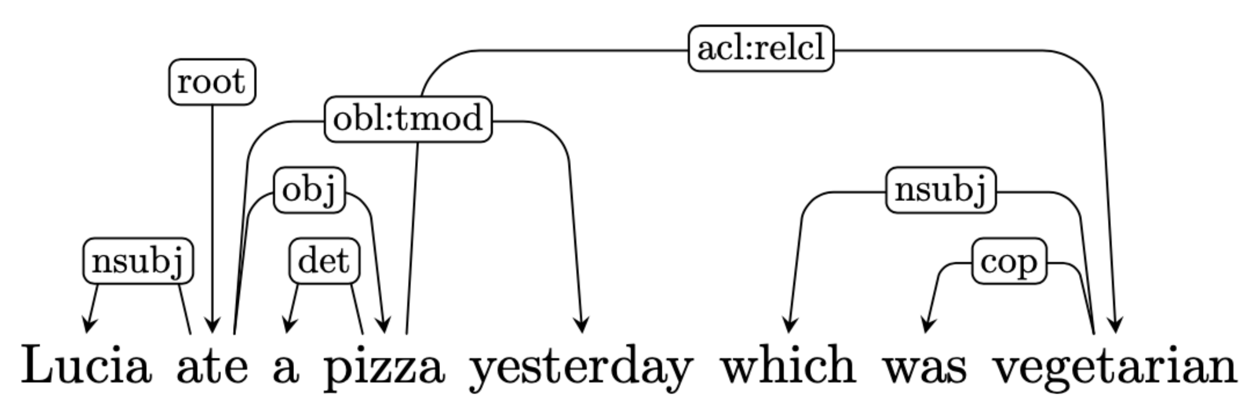
Definition (Projectivity). *An edge from i to j is projective iff all k between i and j are descendants of i. A dependency parse is projective iff all its edges are projective.*

- Dependency parses derived from lexicalized CFG parses are projective $\implies$ restricted class of spanning trees
- Syntactic constituents are *continues* spans

Projective Example



Non-Projective Example



Projectivity (cont.)

	% non-projective edges	% non-projective sentences
Czech	1.86%	22.42%
English	0.39%	7.63%
German	2.33%	28.19%

- The frequency of projectivity is language dependent
- Projectivity has algorithmic consequences
    - Transition-based parsing
        - Simple/efficient but mostly allows for projectivity
    - Graph-based parsing
        - Complex/inefficient but also allows non-projectivity

Graph-Based Dependency Parsing

- Dependency graph: $\boldsymbol{y} = \{(i \xrightarrow{r} j)\}$
- Relation r
- Head word $i \in \{1, 2, \ldots, M, \textsc{Root}\}$
- Modifier $j \in \{1, 2, \ldots, M\}$
- M is length of input $|\boldsymbol{w}|$
- Scoring function: $\Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta})$

Optimal parse: $\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\operatorname{argmax}} \Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta})$

- $\mathcal{Y}(\boldsymbol{w})$ is the set of valid dependency parses on input $\boldsymbol{w}$
- $|\mathcal{Y}(\boldsymbol{w})|$ is exponential in the length of the input

Arc-Factored Assumption

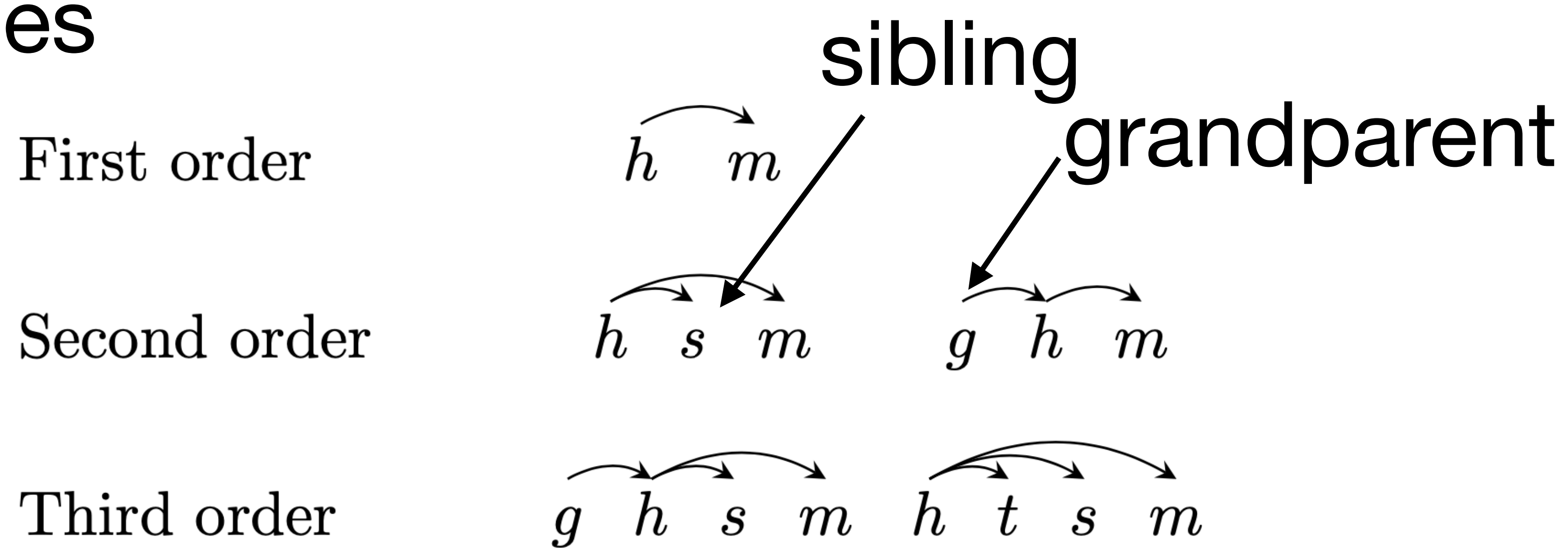
- Decompose the exponential search space into a sum of local feature vectors.
- This assumes that the score is independent of other edges

- **Arc-factored:**

$$\Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta}) = \sum_{i \xrightarrow{r} j \in \boldsymbol{y}} \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta})$$

Higher-Order Dependency Parsing

- Relax arc-factored decomposition to allow higher-order dependencies



$$
\begin{aligned}
\Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta}) = & \sum_{i \xrightarrow{r} j \in \boldsymbol{y}} \psi_{\text{parent}}(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) \\
& + \sum_{k \xrightarrow{r'} i \in \boldsymbol{y}} \psi_{\text{grandparent}}(i \xrightarrow{r} j, k, r', \boldsymbol{w}; \boldsymbol{\theta}) \\
& + \sum_{\substack{i \xrightarrow{r'} s \in \boldsymbol{y} \\ s \neq j}} \psi_{\text{sibling}}(i \xrightarrow{r} j, s, r', \boldsymbol{w}; \boldsymbol{\theta}).
\end{aligned}
$$

Efficient algorithms exist for 1st, 2nd, and 3rd order projective parsing, but it is NP-hard to do higher-order non-projective parsing

Projective Dependency Parsing

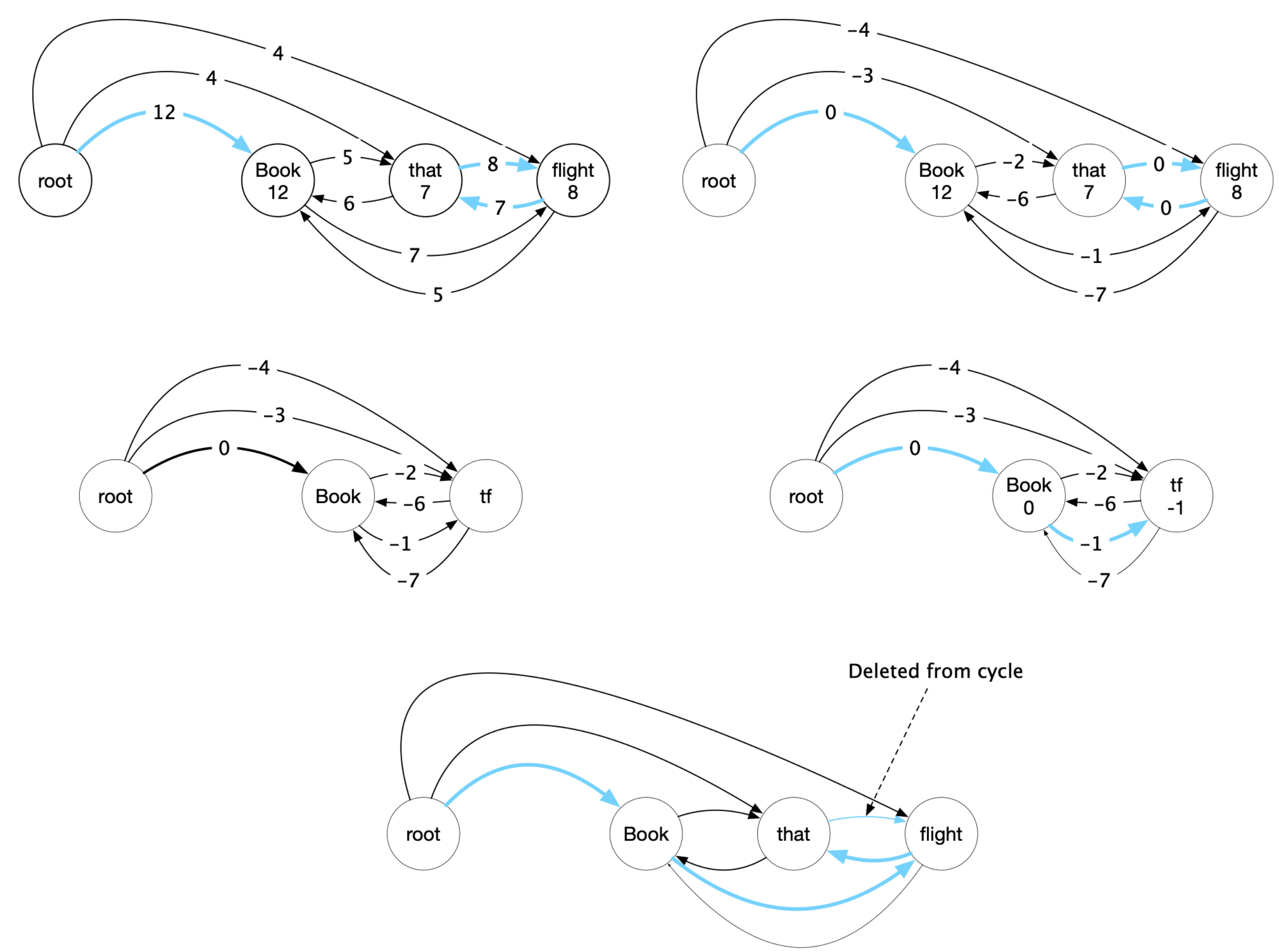
- Lexicalized CFG parsing algorithms can be applied directly to get a projective dependency parse
- Lower bound for scoring the edges $\mathcal{O}(M^2R)$
- There are cubic time algorithms for lexicalized constituent parsing
- Hence, arc-factored projective dependency parsing is in cubic time in the length of the input

- Second-order projective dependency parsing can also be performed in cubic time
- Third-order projective dependency parsing can be performed in $\mathcal{O}(M^4)$

Non-Projective Dependency Parsing

- Precompute scores for the edges
- Find **maximum directed spanning tree** to maximize the total score (**Chu-Liu-Edmonds algorithm**)

- Chu-Liu-Edmonds complexity $\mathcal{O}(M^3R)$
- Can be reduced to $\mathcal{O}(M^2N)$ by storing the edge scores in a Fibonacci heap



Chu-Liu-Edmonds algorithm
Jurafsky Figure 15.14

Computing Scores for Dependency Arcs

$$
\begin{aligned}
\text{Linear} \qquad & \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(i \xrightarrow{r} j, \boldsymbol{w}) \\
\text{Neural} \qquad & \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \text{Feedforward}([\boldsymbol{u}_{w_i}; \boldsymbol{u}_{w_j}]; \boldsymbol{\theta}) \\
\text{Generative} \qquad & \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \log \mathrm{p}(w_j, r \mid w_i).
\end{aligned}
$$

Linear Feature-Based Arc Scores

$$\text{Linear} \qquad \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(i \xrightarrow{r} j, \boldsymbol{w})$$

- Same features $\boldsymbol{f}$ possible as in sequence labeling and discriminative constituency parsing including:
  - the length and direction of the arc;
  - the words w_i and w_j linked by the dependency relation;
  - the neighbors of the dependency arc, $w_{i-1}, w_{i+1}, w_{j-1}, w_{j+1}$;
  - the prefixes, suffixes, and part-of-speech of these neighbor words
- Bilexical features (e.g. $\textit{sushi} \rightarrow \textit{chopsticks}$)
  - Useful but rare, backing off can be helpful

$$\begin{aligned}\boldsymbol{f}(3 \rightarrow 5, \textit{we eat sushi with chopsticks}) = \langle &\textit{sushi} \rightarrow \textit{chopsticks},\\ &\textit{sushi} \rightarrow \text{N}\textsc{ns},\\ &\text{N}\textsc{n} \rightarrow \textit{chopsticks},\\ &\text{N}\textsc{ns} \rightarrow \text{N}\textsc{n}\rangle.\end{aligned}$$

- Many more features are possible

Neural Arc Scores

- Given vector representation $\boldsymbol{x}_i$ for each word w_i

$$\psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \text{FeedForward}([\boldsymbol{x}_i; \boldsymbol{x}_j]; \boldsymbol{\theta}_r)$$

- Kiperwasser and Goldberg (2016) feed forward network:

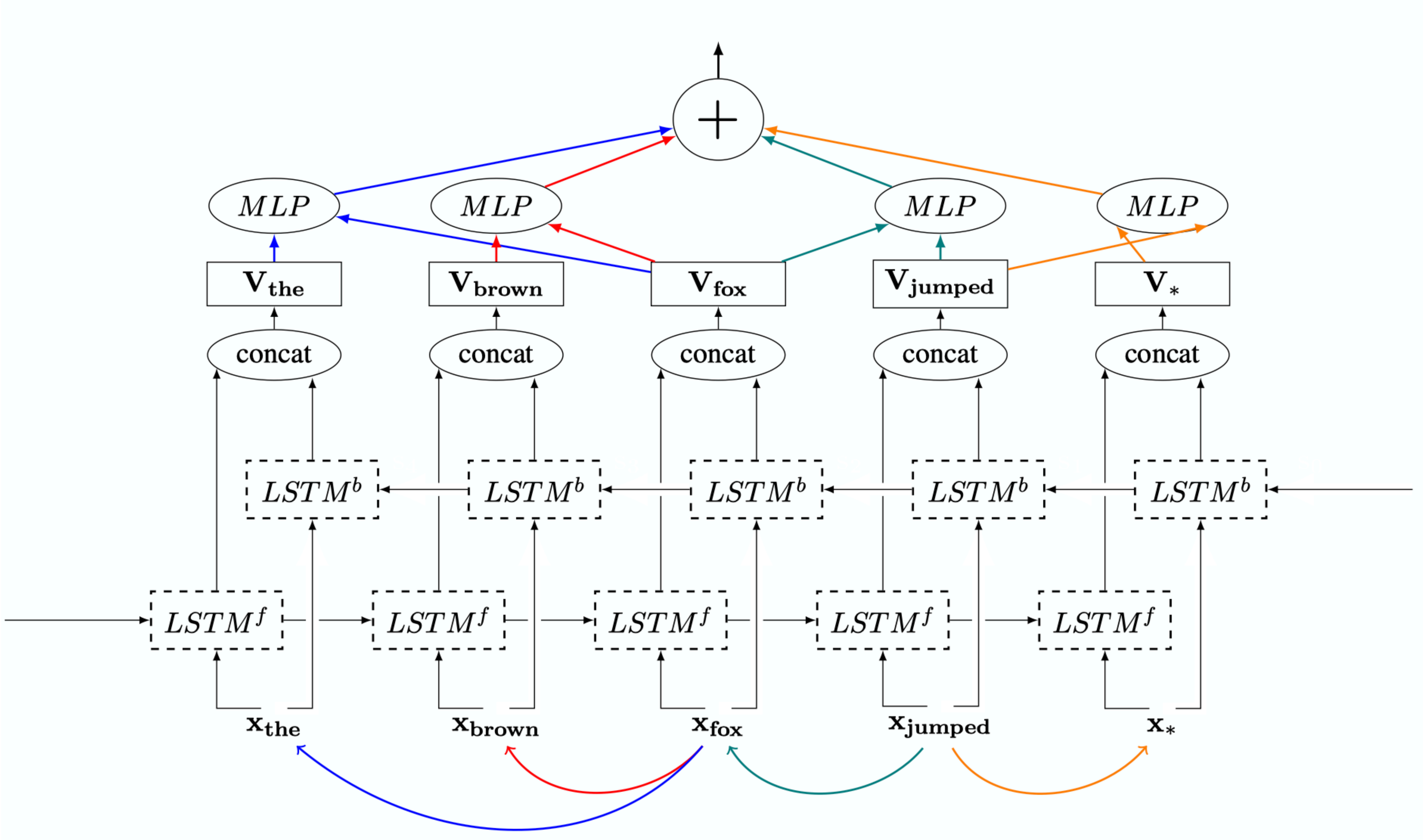
$$\boldsymbol{z} = g(\boldsymbol{\Theta}_r[\boldsymbol{x}_i; \boldsymbol{x}_j] + b_r^{(z)})$$
$$\psi(i \xrightarrow{r} j) = \boldsymbol{\beta}_r \boldsymbol{z} + b_r^{(y)}$$

$\boldsymbol{\Theta}_r$ is a matrix, $\boldsymbol{\beta}_r$ is a vector, each b_r is a scalar, g is an elementwise $\tanh$ activation

- $\boldsymbol{x}_i$ can be a word embedding or a vector incorporating context through a BiLSTM layer on the input word embeddings (Kiperwasser and Goldberg, 2016)

Kiperwasser and Goldberg (2016)

- No handcrafted lexical features
- Context captured through BiLSTM layer



Kiperwasser and Goldberg (2016)

Probabilistic Arc Scores

$$\text{Generative} \qquad \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \log \mathrm{p}(w_j, r \mid w_i)$$

- Unlabeled parse for: *we eat sushi with rice*

$$\begin{aligned}
\boldsymbol{y} &= \{(\text{ROOT}, 2), (2, 1), (2, 3), (3, 5), (5, 4)\} \\
\log \mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) &= \sum_{(i \to j) \in \boldsymbol{y}} \log \mathrm{p}(w_j \mid w_i) \\
&= \log \mathrm{p}(\textit{eat} \mid \text{ROOT}) + \log \mathrm{p}(\textit{we} \mid \textit{eat}) + \log \mathrm{p}(\textit{sushi} \mid \textit{eat}) \\
&\quad + \log \mathrm{p}(\textit{rice} \mid \textit{sushi}) + \log \mathrm{p}(\textit{with} \mid \textit{rice}).
\end{aligned}$$

- Used in combination with expectation-maximization for unsupervised dependency parsing

Learning

- We can apply similar learning algorithms to those used in sequence labeling

- We can update a feature-based arc scores perceptron with:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}')$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}})$$

argmax can be computed as previously described (Chu-Liu-Edmonds algorithm)

Transition-Based Dependency Parsing

- Graph-based dependency parsing
  - Offers exact inference (chooses always best-scoring parse)
  - Scoring is restricted to individual arcs (first-order features) for non-projective parsing
  - Conflict: some types of attachment require second-order features
  - Goes against intuitions of human language processing (sequential reading and listening)
  - Runs relatively slow, running in cubic time in the length of the input
- Transition-based dependency parsing tries to solve those issues
  - Sequential sentence processing
  - Build-up and update the parsing structure through a simple sequence of actions
  - Incorporate higher-order features by looking at this structure
  - Linear time complexity
- **Derivation:** the sequence of actions producing the parse
- Multiple derivations possible ⇒ **spurious ambiguity**

Transition Systems for Dependency Parsing

- **Transition system** consists of parser configuration and a set of transition actions manipulating the configuration
- Configuration $C = (\sigma, \beta, A)$
  - σ is the stack
  - β is the input buffer
  - A is the set of created arcs
- Initial configuration:

$$C_{\text{initial}} = ([\textsc{Root}], \boldsymbol{w}, \varnothing)$$

- Accepting configuration:

$$C_{\text{accept}} = ([\textsc{Root}], \varnothing, A)$$

Spanning tree over the input

Transition System: Arc-Standard

- Closely related to shift-reduce, and to the LR algorithm used to parse programming languages

- Actions
  - *SHIFT* (precond.: input buffer not empty)
    $$(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A)$$
  - *ARC-LEFT* (precond.: top of stack is not *ROOT*)
    $$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, j|\beta, A \oplus j \xrightarrow{r} i)$$
  - *ARC-RIGHT*
    $$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, i|\beta, A \oplus i \xrightarrow{r} j)$$

- Always results in a spanning tree

Transition System: Arc-Standard (Example)

- Actions are provided
- Notice the positions of *ARC-RIGHT* actions

	σ	β	action	arc added to $\mathcal{A}$
1.	[ROOT]	*they like bagels with lox*	SHIFT	
2.	[ROOT, *they*]	*like bagels with lox*	ARC-LEFT	(*they* $\leftarrow$ *like*)
3.	[ROOT]	*like bagels with lox*	SHIFT	
4.	[ROOT, *like*]	*bagels with lox*	SHIFT	
5.	[ROOT, *like, bagels*]	*with lox*	SHIFT	
6.	[ROOT, *like, bagels, with*]	*lox*	ARC-LEFT	(*with* $\leftarrow$ *lox*)
7.	[ROOT, *like, bagels*]	*lox*	ARC-RIGHT	(*bagels* $\rightarrow$ *lox*)
8.	[ROOT, *like*]	*bagels*	ARC-RIGHT	(*like* $\rightarrow$ *bagels*)
9.	[ROOT]	*like*	ARC-RIGHT	(ROOT $\rightarrow$ *like*)
10.	[ROOT]	$\varnothing$	DONE	

Table 11.2: Arc-standard derivation of the unlabeled dependency parse for the input *they like bagels with lox*.

Transition System: Arc-Eager

- Problem with arc-standard: we are not eager to apply *ARC-RIGHT* since right-branching is common in English
  - We tend to *SHIFT* everything onto the stack and assign arcs later to not remove words which still have dependents
- **Arc-eager dependency parsing** will use the *ARC-RIGHT* action more eagerly

- Modified *ARC-RIGHT* action:
  - Pushes the modifier onto the stack rather then removing it
- Additional *ARC-LEFT* precondition:
  - It can not be applied when the top of stack element already has a parent in A
- New *REDUCE* action:
  - Can remove elements from top of stack if it has a parent in A

$$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, i|\beta, A \oplus i \xrightarrow{r} j)$$

Transition System: Arc-Eager (Example)

	σ	β	action	arc added to $\mathcal{A}$
1.	[ROOT]	*they like bagels with lox*	SHIFT	
2.	[ROOT, *they*]	*like bagels with lox*	ARC-LEFT	(*they* $\leftarrow$ *like*)
3.	[ROOT]	*like bagels with lox*	ARC-RIGHT	(ROOT $\rightarrow$ *like*)
4.	[ROOT, *like*]	*bagels with lox*	ARC-RIGHT	(*like* $\rightarrow$ *bagels*)
5.	[ROOT, *like, bagels*]	*with lox*	SHIFT	
6.	[ROOT, *like, bagels, with*]	*lox*	ARC-LEFT	(*with* $\leftarrow$ *lox*)
7.	[ROOT, *like, bagels*]	*lox*	ARC-RIGHT	(*bagels* $\rightarrow$ *lox*)
8.	[ROOT, *like, bagels, lox*]	$\varnothing$	REDUCE	
9.	[ROOT, *like, bagels*]	$\varnothing$	REDUCE	
10.	[ROOT, *like*]	$\varnothing$	REDUCE	
11.	[ROOT]	$\varnothing$	DONE	

Table 11.3: Arc-eager derivation of the unlabeled dependency parse for the input *they like bagels with lox*.

Projectivity

- Arc-standard and arc-eager transition systems produce **projective** dependency trees
- Non-projective transition systems include actions which create arcs to words that are second or third in the stack
- **Pseudo-projective dependency parsing**
  - First do the projective dependency parse
  - Apply graph transformation techniques to produce a non-projective parse

Beam Search

- “greedy” transition-based parsing tries to do the best action at each configuration
  - Leads to search errors
  - Early wrong decisions propagate and can lock the parser in a poor derivation

- **Beam search** tries to correct search errors by keeping multiple partially-complete hypotheses around, called a **beam**

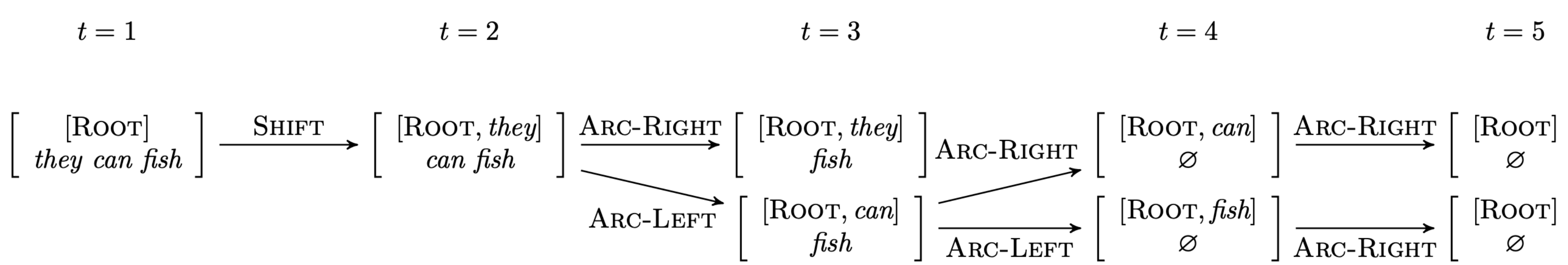


Figure 11.7: Beam search for unlabeled dependency parsing, with beam size $K = 2$. The arc lists for each configuration are not shown, but can be computed from the transitions.

Beam Search (cont.)

- At step t of the derivation there is a set of k hypotheses with score $s_t^{(k)}$ and a set of dependency arcs $A_t^{(k)}$

$$h_t^{(k)} = (s_t^{(k)}, A_t^{(k)})$$

- Keep the k best scoring configurations transitioned from step t at step $t+1$ around
- At the last step the highest scoring configuration is chosen as the parse

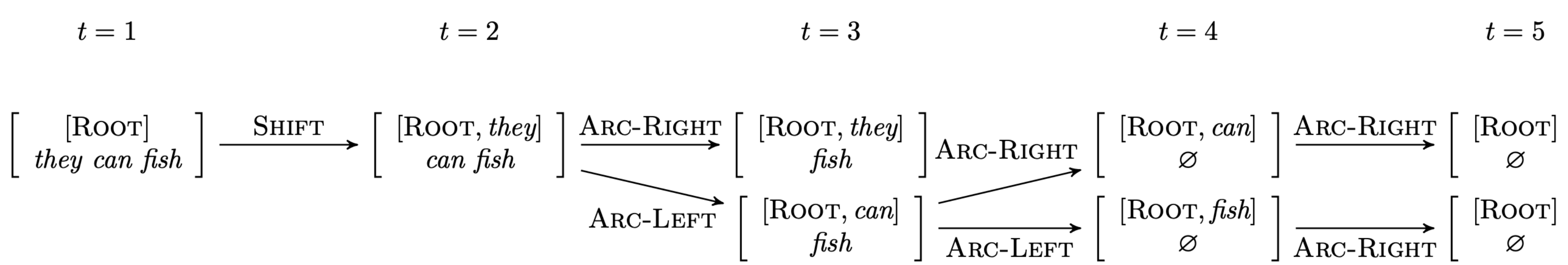


Figure 11.7: Beam search for unlabeled dependency parsing, with beam size $K = 2$. The arc lists for each configuration are not shown, but can be computed from the transitions.

Scoring Functions for Transition-Based Parsers

- In greedy transition-based parsing the current action can be chosen by training a classifier:

$$\hat{a} = \underset{a \in \mathcal{A}(c)}{\operatorname{argmax}} \, \Psi(a, c, \boldsymbol{w}; \boldsymbol{\theta})$$

  - $\mathcal{A}(c)$ is the set of admissible actions
  - c is the current configuration
  - $\boldsymbol{w}$ is the input
  - Ψ is the scoring function with parameters $\boldsymbol{\theta}$

- Feature-based scoring function:

$$\Psi(a, c, \boldsymbol{w}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(a, c, \boldsymbol{w})$$

  - Features $\boldsymbol{f}$ can be all sorts of features from the input buffer, stack, and already created arcs

Neural Scoring Function

- Chen and Manning (2014) feed forward network features:
  - the top three words on the stack, and the first three words on the buffer;
  - the first and second leftmost and rightmost children (dependents) of the top two words on the stack;
  - the leftmost and right most grandchildren of the top two words on the stack;
  - embeddings of the part-of-speech tags of these words

$$
\begin{aligned}
c =& (\sigma, \beta, A) \\
\boldsymbol{x}(c, \boldsymbol{w}) =& [\boldsymbol{v}_{w_{\sigma_1}}, \boldsymbol{v}_{t_{\sigma_1}} \boldsymbol{v}_{w_{\sigma_2}}, \boldsymbol{v}_{t_{\sigma_2}}, \boldsymbol{v}_{w_{\sigma_3}}, \boldsymbol{v}_{t_{\sigma_3}}, \boldsymbol{v}_{w_{\beta_1}}, \boldsymbol{v}_{t_{\beta_1}}, \boldsymbol{v}_{w_{\beta_2}}, \boldsymbol{v}_{t_{\beta_2}}, \ldots]
\end{aligned}
$$

- Feed forward network:

$$
\begin{aligned}
\boldsymbol{z} =& g(\Theta^{(x \to z)} \boldsymbol{x}(c, \boldsymbol{w})) \\
\psi(a, c, \boldsymbol{w}; \boldsymbol{\theta}) =& \Theta_a^{(z \to y)} \boldsymbol{z}
\end{aligned}
$$

  - Cubic elementwise activation function $g(x) = x^3$

Learning to Parse

- Mismatch between the supervision, the dependency trees, and the classifier's prediction space (set of parsing actions)

- Create new training data by converting parse trees into action sequences (often a deterministic algorithm)
- Alternatively, derive supervision directly from the parser's performance

Oracle-Based Training

- Transition system: action sequence $\longmapsto$ dependency tree
- **Oracle**: dependency tree $\longmapsto$ action sequence
- Oracle has to choose between multiple derivations in case of spurious ambiguity
- Convert dependency treebank to set of oracle action sequences $\{A^{(i)}\}_{i=1}^{N}$

- Train transition based parser with:

$$\mathrm{p}(a \mid c, \boldsymbol{w}) = \frac{\exp \Psi(a, c, \boldsymbol{w}; \boldsymbol{\theta})}{\sum_{a' \in \mathcal{A}(c)} \exp \Psi(a', c, \boldsymbol{w}; \boldsymbol{\theta})}$$

log-likelihood loss

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \sum_{t=1}^{|A^{(i)}|} \log \mathrm{p}(a_t^{(i)} \mid c_t^{(i)}, \boldsymbol{w})$$

  - $|A^{(i)}|$ is length of the action sequence $A^{(i)}$
- Beam search: sequence score is obtained by summing action losses

Global Objective

- Objective $\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \sum_{t=1}^{|A^{(i)}|} \log \mathrm{p}(a_t^{(i)} \mid c_t^{(i)}, \boldsymbol{w})$ is **locally-normalized**
- Training on individual actions can be sub-optimal with respect to the global performance (**label bias problem**)
- Example:
  - Configuration appears *100* times in training data oracle action a_1 is used in *51* cases and a_2 in *49* cases. a_1 results in a cascading error whereas a_2 results in a single error
  - Local objective function prefers a_1, but choosing a_2 minimizes the overall number of errors

- Globally-normalized conditional likelihood:

$$\mathrm{p}(A^{(i)} \mid \boldsymbol{w}; \boldsymbol{\theta}) = \frac{\exp \sum_{t=1}^{|A^{(i)}|} \Psi(a_t^{(i)}, c_t^{(i)}, \boldsymbol{w})}{\sum_{A' \in \mathbb{A}(\boldsymbol{w})} \exp \sum_{t=1}^{|A'|} \Psi(a'_t, c'_t, \boldsymbol{w})}$$

Set of all possible action sequences $\mathbb{A}(\boldsymbol{w})$

Global Objective (cont.)

$$\mathrm{p}(A^{(i)} \mid \boldsymbol{w}; \boldsymbol{\theta}) = \frac{\exp \sum_{t=1}^{|A^{(i)}|} \Psi(a_t^{(i)}, c_t^{(i)}, \boldsymbol{w})}{\sum_{A' \in \mathbb{A}(\boldsymbol{w})} \exp \sum_{t=1}^{|A'|} \Psi(a'_t, c'_t, \boldsymbol{w})}$$

- Denominator can be approximated using Beam search:

$$\sum_{A' \in \mathbb{A}(\boldsymbol{w})} \exp \sum_{t=1}^{|A'|} \Psi(a'_t, c'_t, \boldsymbol{w}) \approx \sum_{k=1}^{K} \exp \sum_{t=1}^{|A^{(k)}|} \Psi(a_t^{(k)}, c_t^{(k)}, \boldsymbol{w})$$

$A^{(k)}$ is an action sequence on a beam of size K

- Resulting in the loss function:

$$L(\boldsymbol{\theta}) = -\sum_{t=1}^{|A^{(i)}|} \Psi(a_t^{(i)}, c_t^{(i)}, \boldsymbol{w}) + \log \sum_{k=1}^{K} \exp \sum_{t=1}^{|A^{(k)}|} \Psi(a_t^{(k)}, c_t^{(k)}, \boldsymbol{w})$$

Dependency Parsing on Penn Treebank

Model	POS	UAS	LAS	Paper / Source	Code
Label Attention Layer + HPSG + XLNet (Mrini et al., 2019)	97.3	97.42	96.26	Rethinking Self-Attention: Towards Interpretability for Neural Parsing	Official
BIST transition-based parser (Kiperwasser and Goldberg, 2016)	97.3	93.9	91.9	Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations	Official
BIST graph-based parser (Kiperwasser and Goldberg, 2016)	97.3	93.1	91.0	Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations	Official

http://nlpprogress.com/english/dependency_parsing.html

Applications - Digital Humanities Research

- Searching for pairs of words which might not be adjacent
- Search Google n-grams for: *write* → *code*
  - Results: *write some code*, *write good code*, *write all the code*, etc.
  - Plot use of this dependency arc over time

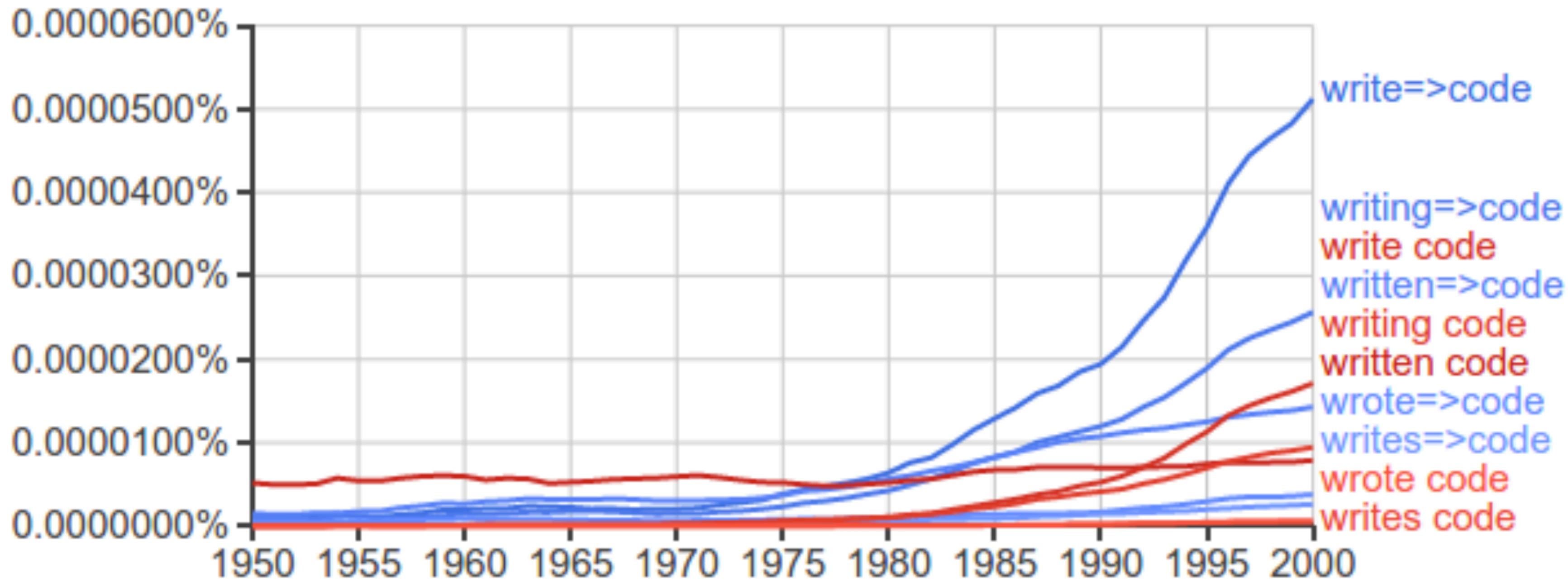


Figure 11.8: Google n-grams results for the bigram *write code* and the dependency arc *write* => *code* (and their morphological variants)

Applications - Relation Extraction

- **Relation extraction** of chapter 17
- Identify pairs of entities with relations to each other
  - Example authorship:

(Melville, Moby-Dick)
(Tolstoy, War and Peace)
(Marquéz, 100 Years of Solitude)
(Shakespeare, A Midsummer Night's Dream)

- Paris can often be identified by consistent chains of dependency relations
- Dependency parsing can help finding new instances of a relation based of other instances of the same type

Applications - Question Answering

- Dependency parsing can improve question answering
- Example query:

(11.1) What percentage of the nation's cheese does Wisconsin produce?

- Sentence in corpus:

(11.2) In Wisconsin, where farmers produce 28% of the nation's cheese, …

- In the dependency parses there is an edge from *produce* to *Wisconsin* in both the question and the potential answer
- Likelihood is increased that this span of text is relevant for the answer

Applications - Sentiment Analysis

- Is sentence positive or negative?
- Polarity can be reversed by negation

(11.3) *There is no reason at all to believe the polluters will suddenly become reasonable.*

- Through dependency parsing we can track the sentiment polarity to better identify the overall polarity

Questions?

Thank you for listening