Chapter 11

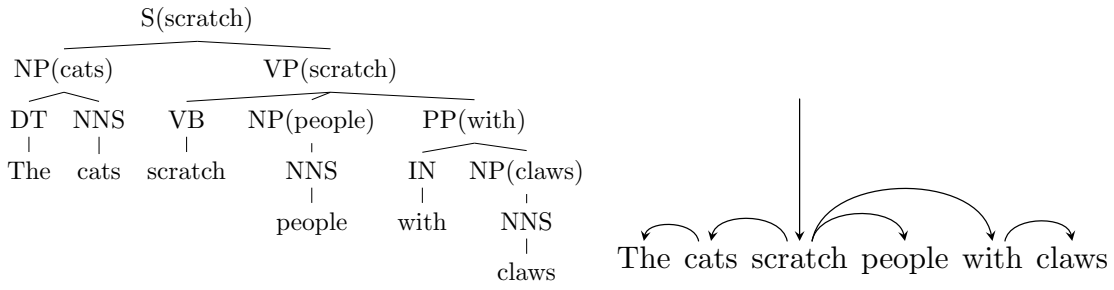
Dependency parsing

The previous chapter discussed algorithms for analyzing sentences in terms of nested constituents, such as noun phrases and verb phrases. However, many of the key sources of ambiguity in phrase-structure analysis relate to questions of **attachment**: where to attach a prepositional phrase or complement clause, how to scope a coordinating conjunction, and so on. These attachment decisions can be represented with a more lightweight structure: a directed graph over the words in the sentence, known as a **dependency parse**. Syntactic annotation has shifted its focus to such dependency structures: at the time of this writing, the **Universal Dependencies** project offers more than 100 dependency treebanks for more than 60 languages.[1] This chapter will describe the linguistic ideas underlying dependency grammar, and then discuss exact and transition-based parsing algorithms. The chapter will also discuss recent research on **learning to search** in transition-based structure prediction.

11.1 Dependency grammar

While **dependency grammar** has a rich history of its own (Tesnière, 1966; Kübler et al., 2009), it can be motivated by extension from the lexicalized context-free grammars that we encountered in previous chapter (§ 10.5.2). Recall that lexicalization augments each non-terminal with a **head word**. The head of a constituent is identified recursively, using a set of **head rules**, as shown in Table 10.3. An example of a lexicalized context-free parse is shown in Figure 11.1a. In this sentence, the head of the S constituent is the main verb, *scratch*; this non-terminal then produces the noun phrase *the cats*, whose head word is *cats*, and from which we finally derive the word *the*. Thus, the word *scratch* occupies the central position for the sentence, with the word *cats* playing a supporting role. In turn, *cats*

[1] `universaldependencies.org`



(a) lexicalized constituency parse (b) unlabeled dependency tree

Figure 11.1: Dependency grammar is closely linked to lexicalized context free grammars: each lexical head has a dependency path to every other word in the constituent. (This example is based on the lexicalization rules from § 10.5.2, which make the preposition the head of a prepositional phrase. In the more contemporary Universal Dependencies annotations, the head of *with claws* would be *claws*, so there would be an edge *scratch* → *claws*.)

occupies the central position for the noun phrase, with the word *the* playing a supporting role.

The relationships between words in a sentence can be formalized in a directed graph, based on the lexicalized phrase-structure parse: create an edge (i, j) iff word i is the head of a phrase whose child is a phrase headed by word j. Thus, in our example, we would have *scratch* → *cats* and *cats* → *the*. We would not have the edge *scratch* → *the*, because although S(*scratch*) dominates DET(*the*) in the phrase-structure parse tree, it is not its immediate parent. These edges describe **syntactic dependencies**, a bilexical relationship between a **head** and a **dependent**, which is at the heart of dependency grammar.

Continuing to build out this **dependency graph**, we will eventually reach every word in the sentence, as shown in Figure 11.1b. In this graph — and in all graphs constructed in this way — every word has exactly one incoming edge, except for the root word, which is indicated by a special incoming arrow from above. Furthermore, the graph is *weakly connected*: if the directed edges were replaced with undirected edges, there would be a path between all pairs of nodes. From these properties, it can be shown that there are no cycles in the graph (or else at least one node would have to have more than one incoming edge), and therefore, the graph is a tree. Because the graph includes all vertices, it is a **spanning tree**.

11.1.1 Heads and dependents

A dependency edge implies an asymmetric syntactic relationship between the head and dependent words, sometimes called **modifiers**. For a pair like *the cats* or *cats scratch*, how

do we decide which is the head? Here are some possible criteria:

- The head sets the syntactic category of the construction: for example, nouns are the heads of noun phrases, and verbs are the heads of verb phrases.
- The modifier may be optional while the head is mandatory: for example, in the sentence *cats scratch people with claws*, the subtrees *cats scratch* and *cats scratch people* are grammatical sentences, but *with claws* is not.
- The head determines the morphological form of the modifier: for example, in languages that require gender agreement, the gender of the noun determines the gender of the adjectives and determiners.
- Edges should first connect content words, and then connect function words.

These guidelines are not universally accepted, and they sometimes conflict. The Universal Dependencies (UD) project has attempted to identify a set of principles that can be applied to dozens of different languages (Nivre et al., 2016).[2] These guidelines are based on the universal part-of-speech tags from chapter 8. They differ somewhat from the head rules described in § 10.5.2: for example, on the principle that dependencies should relate content words, the prepositional phrase *with claws* would be headed by *claws*, resulting in an edge *scratch* → *claws*, and another edge *claws* → *with*.

One objection to dependency grammar is that not all syntactic relations are asymmetric. One such relation is coordination (Popel et al., 2013): in the sentence, *Abigail and Max like kimchi* (Figure 11.2), which word is the head of the coordinated noun phrase *Abigail and Max*? Choosing either *Abigail* or *Max* seems arbitrary; fairness argues for making *and* the head, but this seems like the least important word in the noun phrase, and selecting it would violate the principle of linking content words first. The Universal Dependencies annotation system arbitrarily chooses the left-most item as the head — in this case, *Abigail* — and includes edges from this head to both *Max* and the coordinating conjunction *and*. These edges are distinguished by the labels CONJ (for the thing begin conjoined) and CC (for the coordinating conjunction). The labeling system is discussed next.

11.1.2 Labeled dependencies

Edges may be **labeled** to indicate the nature of the syntactic relation that holds between the two elements. For example, in Figure 11.2, the label NSUBJ on the edge from *like* to *Abigail* indicates that the subtree headed by *Abigail* is the noun subject of the verb *like*; similarly, the label OBJ on the edge from *like* to *kimchi* indicates that the subtree headed by

[2] The latest and most specific guidelines are available at `universaldependencies.org/guidelines.html`

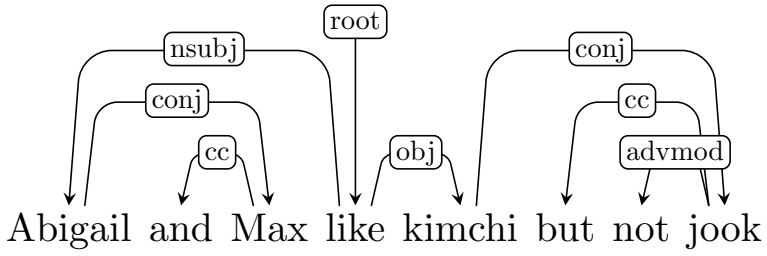


Figure 11.2: In the Universal Dependencies annotation system, the left-most item of a coordination is the head.

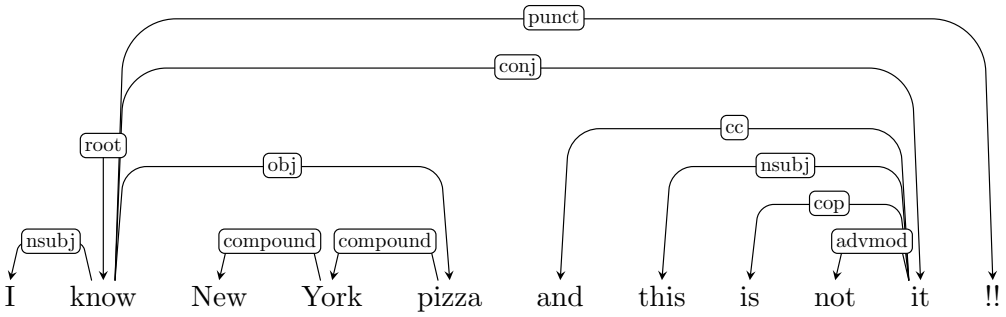


Figure 11.3: A labeled dependency parse from the English UD Treebank (reviews-361348-0006)

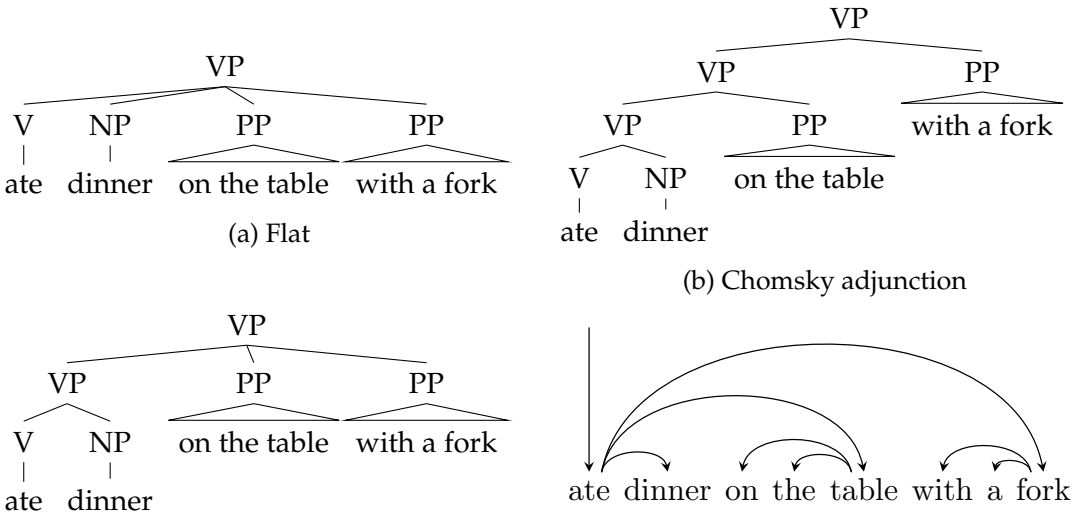
kimchi is the object.[3] The negation *not* is treated as an adverbial modifier (ADVMOD) on the noun *jook*.

A slightly more complex example is shown in Figure 11.3. The multiword expression *New York pizza* is treated as a "flat" unit of text, with the elements linked by the COMPOUND relation. The sentence includes two clauses that are conjoined in the same way that noun phrases are conjoined in Figure 11.2. The second clause contains a **copula** verb (see § 8.1.1). For such clauses, we treat the "object" of the verb as the root — in this case, *it* — and label the verb as a dependent, with the COP relation. This example also shows how punctuations are treated, with label PUNCT.

11.1.3 Dependency subtrees and constituents

Dependency trees hide information that would be present in a CFG parse. Often what is hidden is in fact irrelevant: for example, Figure 11.4 shows three different ways of

[3] Earlier work distinguished direct and indirect objects (De Marneffe and Manning, 2008), but this has been dropped in version 2.0 of the Universal Dependencies annotation system.



(a) Flat

(b) Chomsky adjunction

(c) Two-level (PTB-style)

(d) Dependency representation

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

representing prepositional phrase adjuncts to the verb *ate*. Because there is apparently no meaningful difference between these analyses, the Penn Treebank decides by convention to use the two-level representation (see Johnson, 1998, for a discussion). As shown in Figure 11.4d, these three cases all look the same in a dependency parse.

But dependency grammar imposes its own set of annotation decisions, such as the identification of the head of a coordination (§ 11.1.1); without lexicalization, context-free grammar does not require either element in a coordination to be privileged in this way. Dependency parses can be disappointingly flat: for example, in the sentence *Yesterday, Abigail was reluctantly giving Max kimchi*, the root *giving* is the head of every dependency! The constituent parse arguably offers a more useful structural analysis for such cases.

Projectivity Thus far, we have defined dependency trees as spanning trees over a graph in which each word is a vertex. As we have seen, one way to construct such trees is by connecting the heads in a lexicalized constituent parse. However, there are spanning trees that cannot be constructed in this way. Syntactic constituents are *contiguous* spans. In a spanning tree constructed from a lexicalized constituent parse, the head h of any constituent that spans the nodes from i to j must have a path to every node in this span. This is property is known as **projectivity**, and projective dependency parses are a restricted class of spanning trees. Informally, projectivity means that "crossing edges" are prohibited. The formal definition follows:

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

Table 11.1: Frequency of non-projective dependencies in three languages (Kuhlmann and Nivre, 2010)

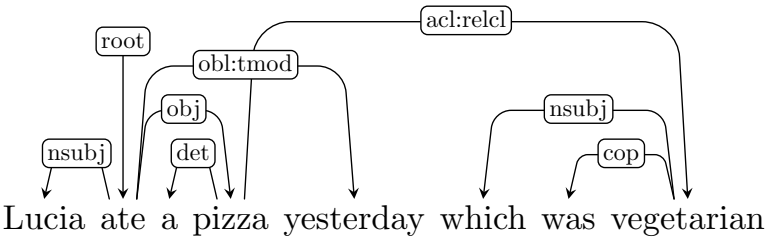


Figure 11.5: An example of a non-projective dependency parse. The "crossing edge" arises from the relative clause *which was vegetarian* and the oblique temporal modifier *yesterday*.

Definition 2 (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.*

Figure 11.5 gives an example of a non-projective dependency graph in English. This dependency graph does not correspond to any constituent parse. As shown in Table 11.1, non-projectivity is more common in languages such as Czech and German. Even though relatively few dependencies are non-projective in these languages, many sentences have at least one such dependency. As we will soon see, projectivity has important algorithmic consequences.

11.2 Graph-based dependency parsing

Let $\boldsymbol{y} = \{(i \xrightarrow{r} j)\}$ represent a dependency graph, in which each edge is a relation r from head word $i \in \{1, 2, \ldots, M, \text{ROOT}\}$ to modifier $j \in \{1, 2, \ldots, M\}$. The special node ROOT indicates the root of the graph, and M is the length of the input $|\boldsymbol{w}|$. Given a scoring function $\Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta})$, the optimal parse is,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}}\, \Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta}), \qquad [11.1]$$

where $\mathcal{Y}(\boldsymbol{w})$ is the set of valid dependency parses on the input $\boldsymbol{w}$. As usual, the number of possible labels $|\mathcal{Y}(\boldsymbol{w})|$ is exponential in the length of the input (Wu and Chao, 2004).

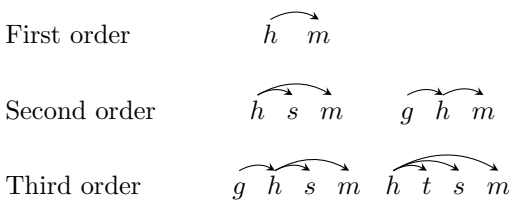


Figure 11.6: Feature templates for higher-order dependency parsing

Algorithms that search over this space of possible graphs are known as **graph-based dependency parsers**.

In sequence labeling and constituent parsing, it was possible to search efficiently over an exponential space by choosing a feature function that decomposes into a sum of local feature vectors. A similar approach is possible for dependency parsing, by requiring the scoring function to decompose across dependency arcs:

$$\Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta}) = \sum_{i \xrightarrow{r} j \in \boldsymbol{y}} \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}). \quad [11.2]$$

Dependency parsers that operate under this assumption are known as **arc-factored**, since the score of a graph is the product of the scores of all arcs.

Higher-order dependency parsing The arc-factored decomposition can be relaxed to allow higher-order dependencies. In **second-order dependency parsing**, the scoring function may include grandparents and siblings, as shown by the templates in Figure 11.6. The scoring function is,

$$\begin{aligned}\Psi(\boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta}) = \sum_{i \xrightarrow{r} j \in \boldsymbol{y}} & \psi_{\text{parent}}(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) \\ & + \sum_{k \xrightarrow{r'} i \in \boldsymbol{y}} \psi_{\text{grandparent}}(i \xrightarrow{r} j, k, r', \boldsymbol{w}; \boldsymbol{\theta}) \\ & + \sum_{\substack{i \xrightarrow{r'} s \in \boldsymbol{y} \\ s \neq j}} \psi_{\text{sibling}}(i \xrightarrow{r} j, s, r', \boldsymbol{w}; \boldsymbol{\theta}). \end{aligned} \quad [11.3]$$

The top line scores computes a scoring function that includes the grandparent k; the bottom line computes a scoring function for each sibling s. For projective dependency graphs, there are efficient algorithms for second-order and third-order dependency parsing (Eisner, 1996; McDonald and Pereira, 2006; Koo and Collins, 2010); for non-projective dependency graphs, second-order dependency parsing is NP-hard (McDonald and Pereira, 2006). The specific algorithms are discussed in the next section.

11.2.1 Graph-based parsing algorithms

The distinction between projective and non-projective dependency trees (§ 11.1.3) plays a key role in the choice of algorithms. Because projective dependency trees are closely related to (and can be derived from) lexicalized constituent trees, lexicalized parsing algorithms can be applied directly. For the more general problem of parsing to arbitrary spanning trees, a different class of algorithms is required. In both cases, arc-factored dependency parsing relies on precomputing the scores $\psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta})$ for each potential edge. There are $\mathcal{O}(M^2R)$ such scores, where M is the length of the input and R is the number of dependency relation types, and this is a lower bound on the time and space complexity of any exact algorithm for arc-factored dependency parsing.

Projective dependency parsing

Any lexicalized constituency tree can be converted into a projective dependency tree by creating arcs between the heads of constituents and their parents, so any algorithm for lexicalized constituent parsing can be converted into an algorithm for projective dependency parsing, by converting arc scores into scores for lexicalized productions. As noted in § 10.5.2, there are cubic time algorithms for lexicalized constituent parsing, which are extensions of the CKY algorithm. Therefore, arc-factored projective dependency parsing can be performed in cubic time in the length of the input.

Second-order projective dependency parsing can also be performed in cubic time, with minimal modifications to the lexicalized parsing algorithm (Eisner, 1996). It is possible to go even further, to **third-order dependency parsing**, in which the scoring function may consider great-grandparents, grand-siblings, and "tri-siblings", as shown in Figure 11.6. Third-order dependency parsing can be performed in $\mathcal{O}(M^4)$ time, which can be made practical through the use of pruning to eliminate unlikely edges (Koo and Collins, 2010).

Non-projective dependency parsing

In non-projective dependency parsing, the goal is to identify the highest-scoring spanning tree over the words in the sentence. The arc-factored assumption ensures that the score for each spanning tree will be computed as a sum over scores for the edges, which are precomputed. Based on these scores, we build a weighted connected graph. Arc-factored non-projective dependency parsing is then equivalent to finding the spanning tree that achieves the maximum total score, $\Psi(\boldsymbol{y}, \boldsymbol{w}) = \sum_{i \xrightarrow{r} j \in \boldsymbol{y}} \psi(i \xrightarrow{r} j, \boldsymbol{w})$. The **Chu-Liu-Edmonds algorithm** (Chu and Liu, 1965; Edmonds, 1967) computes this **maximum directed spanning tree** efficiently. It does this by first identifying the best incoming edge $i \xrightarrow{r} j$ for each vertex j. If the resulting graph does not contain cycles, it is the maximum spanning tree. If there is a cycle, it is collapsed into a super-vertex, whose incoming and outgoing edges are based on the edges to the vertices in the cycle. The algorithm is

then applied recursively to the resulting graph, and process repeats until a graph without cycles is obtained.

The time complexity of identifying the best incoming edge for each vertex is $\mathcal{O}(M^2R)$, where M is the length of the input and R is the number of relations; in the worst case, the number of cycles is $\mathcal{O}(M)$. Therefore, the complexity of the Chu-Liu-Edmonds algorithm is $\mathcal{O}(M^3R)$. This complexity can be reduced to $\mathcal{O}(M^2N)$ by storing the edge scores in a Fibonnaci heap (Gabow et al., 1986). For more detail on graph-based parsing algorithms, see Eisner (1997) and Kübler et al. (2009).

Higher-order non-projective dependency parsing Given the tractability of higher-order projective dependency parsing, you may be surprised to learn that non-projective second-order dependency parsing is NP-Hard. This can be proved by reduction from the vertex cover problem (Neuhaus and Bröker, 1997). A heuristic solution is to do projective parsing first, and then post-process the projective dependency parse to add non-projective edges (Nivre and Nilsson, 2005). More recent work has applied techniques for approximate inference in graphical models, including belief propagation (Smith and Eisner, 2008), integer linear programming (Martins et al., 2009), variational inference (Martins et al., 2010), and Markov Chain Monte Carlo (Zhang et al., 2014).

11.2.2 Computing scores for dependency arcs

The arc-factored scoring function $\psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta})$ can be defined in several ways:

$$\text{Linear} \quad \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(i \xrightarrow{r} j, \boldsymbol{w}) \tag{11.4}$$

$$\text{Neural} \quad \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \text{Feedforward}([\boldsymbol{u}_{w_i}; \boldsymbol{u}_{w_j}]; \boldsymbol{\theta}) \tag{11.5}$$

$$\text{Generative} \quad \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \log \text{p}(w_j, r \mid w_i). \tag{11.6}$$

Linear feature-based arc scores

Linear models for dependency parsing incorporate many of the same features used in sequence labeling and discriminative constituent parsing. These include:

- the length and direction of the arc;
- the words w_i and w_j linked by the dependency relation;
- the prefixes, suffixes, and parts-of-speech of these words;
- 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.

Each of these features can be conjoined with the dependency edge label r. Note that features in an arc-factored parser can refer to words other than w_i and w_j. The restriction is that the features consider only a single arc.

Bilexical features (e.g., *sushi* $\rightarrow$ *chopsticks*) are powerful but rare, so it is useful to augment them with coarse-grained alternatives, by "backing off" to the part-of-speech or affix. For example, the following features are created by backing off to part-of-speech tags in an unlabeled dependency parser:

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

Regularized discriminative learning algorithms can then trade off between features at varying levels of detail. McDonald et al. (2005) take this approach as far as *tetralexical* features (e.g., $(w_i, w_{i+1}, w_{j-1}, w_j)$). Such features help to avoid choosing arcs that are unlikely due to the intervening words: for example, there is unlikely to be an edge between two nouns if the intervening span contains a verb. A large list of first and second-order features is provided by Bohnet (2010), who uses a hashing function to store these features efficiently.

Neural arc scores

Given vector representations $\boldsymbol{x}_i$ for each word w_i in the input, a set of arc scores can be computed from a feedforward neural network:

$$\psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta}) = \text{FeedForward}([\boldsymbol{x}_i; \boldsymbol{x}_j]; \boldsymbol{\theta}_r), \tag{11.7}$$

where unique weights $\boldsymbol{\theta}_r$ are available for each arc type (Pei et al., 2015; Kiperwasser and Goldberg, 2016). Kiperwasser and Goldberg (2016) use a feedforward network with a single hidden layer,

$$\boldsymbol{z} = g(\boldsymbol{\Theta}_r[\boldsymbol{x}_i; \boldsymbol{x}_j] + b_r^{(z)}) \tag{11.8}$$

$$\psi(i \xrightarrow{r} j) = \boldsymbol{\beta}_r \boldsymbol{z} + b_r^{(y)}, \tag{11.9}$$

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

The vector $\boldsymbol{x}_i$ can be set equal to the word embedding, which may be pre-trained or learned by backpropagation (Pei et al., 2015). Alternatively, contextual information can be incorporated by applying a bidirectional recurrent neural network across the input, as

described in § 7.6. The RNN hidden states at each word can be used as inputs to the arc scoring function (Kiperwasser and Goldberg, 2016).

Feature-based arc scores are computationally expensive, due to the costs of storing and searching a huge table of weights. Neural arc scores can be viewed as a compact solution to this problem. Rather than working in the space of tuples of lexical features, the hidden layers of a feedforward network can be viewed as implicitly computing feature combinations, with each layer of the network evaluating progressively more words. An early paper on neural dependency parsing showed substantial speed improvements at test time, while also providing higher accuracy than feature-based models (Chen and Manning, 2014).

Probabilistic arc scores

If each arc score is equal to the log probability $\log \text{p}(w_j, r \mid w_i)$, then the sum of scores gives the log probability of the sentence and arc labels, by the chain rule. For example, consider the unlabeled parse of *we eat sushi with rice*,

$$\boldsymbol{y} = \{(\text{ROOT}, 2), (2, 1), (2, 3), (3, 5), (5, 4)\} \quad [11.10]$$

$$\log \text{p}(\boldsymbol{w} \mid \boldsymbol{y}) = \sum_{(i \rightarrow j) \in \boldsymbol{y}} \log \text{p}(w_j \mid w_i) \quad [11.11]$$

$$= \log \text{p}(\textit{eat} \mid \text{ROOT}) + \log \text{p}(\textit{we} \mid \textit{eat}) + \log \text{p}(\textit{sushi} \mid \textit{eat}) + \log \text{p}(\textit{rice} \mid \textit{sushi}) + \log \text{p}(\textit{with} \mid \textit{rice}). \quad [11.12]$$

Probabilistic generative models are used in combination with expectation-maximization (chapter 5) for unsupervised dependency parsing (Klein and Manning, 2004).

11.2.3 Learning

Having formulated graph-based dependency parsing as a structure prediction problem, we can apply similar learning algorithms to those used in sequence labeling. Given a loss function $\ell(\boldsymbol{\theta}; \boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$, we can compute gradient-based updates to the parameters. For a model with feature-based arc scores and a perceptron loss, we obtain the usual structured perceptron update,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}} \, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}') \quad [11.13]$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}) \quad [11.14]$$

In this case, the argmax requires a maximization over all dependency trees for the sentence, which can be computed using the algorithms described in § 11.2.1. We can apply all the usual tricks from § 2.3: weight averaging, a large margin objective, and regularization. McDonald et al. (2005) were the first to treat dependency parsing as a structure

prediction problem, using MIRA, an online margin-based learning algorithm. Neural arc scores can be learned in the same way, backpropagating from a margin loss to updates on the feedforward network that computes the score for each edge.

A conditional random field for arc-factored dependency parsing is built on the probability model,

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\exp \sum_{i \xrightarrow{r} j \in \boldsymbol{y}} \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta})}{\sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp \sum_{i \xrightarrow{r} j \in \boldsymbol{y}'} \psi(i \xrightarrow{r} j, \boldsymbol{w}; \boldsymbol{\theta})} \quad [11.15]$$

Such a model is trained to minimize the negative log conditional-likelihood. Just as in CRF sequence models (§ 7.5.3) and the logistic regression classifier (§ 2.5), the gradients involve marginal probabilities $\mathrm{p}(i \xrightarrow{r} j \mid \boldsymbol{w}; \boldsymbol{\theta})$, which in this case are probabilities over individual dependencies. In arc-factored models, these probabilities can be computed in polynomial time. For projective dependency trees, the marginal probabilities can be computed in cubic time, using a variant of the inside-outside algorithm (Lari and Young, 1990). For non-projective dependency parsing, marginals can also be computed in cubic time, using the **matrix-tree theorem** (Koo et al., 2007; McDonald et al., 2007; Smith and Smith, 2007). Details of these methods are described by Kübler et al. (2009).

11.3 Transition-based dependency parsing

Graph-based dependency parsing offers exact inference, meaning that it is possible to recover the best-scoring parse for any given model. But this comes at a price: the scoring function is required to decompose into local parts — in the case of non-projective parsing, these parts are restricted to individual arcs. These limitations are felt more keenly in dependency parsing than in sequence labeling, because second-order dependency features are critical to correctly identify some types of attachments. For example, prepositional phrase attachment depends on the attachment point, the object of the preposition, and the preposition itself; arc-factored scores cannot account for all three of these features simultaneously. Graph-based dependency parsing may also be criticized on the basis of intuitions about human language processing: people read and listen to sentences *sequentially*, incrementally building mental models of the sentence structure and meaning before getting to the end (Jurafsky, 1996). This seems hard to reconcile with graph-based algorithms, which perform bottom-up operations on the entire sentence, requiring the parser to keep every word in memory. Finally, from a practical perspective, graph-based dependency parsing is relatively slow, running in cubic time in the length of the input.

Transition-based algorithms address all three of these objections. They work by moving through the sentence sequentially, while performing actions that incrementally update a stored representation of what has been read thus far. As with the shift-reduce

parser from § 10.6.2, this representation consists of a stack, onto which parsing substructures can be pushed and popped. In shift-reduce, these substructures were constituents; in the transition systems that follow, they will be projective dependency trees over partial spans of the input.[4] Parsing is complete when the input is consumed and there is only a single structure on the stack. The sequence of actions that led to the parse is known as the **derivation**. One problem with transition-based systems is that there may be multiple derivations for a single parse structure — a phenomenon known as **spurious ambiguity**.

11.3.1 Transition systems for dependency parsing

A **transition system** consists of a representation for describing configurations of the parser, and a set of transition actions, which manipulate the configuration. There are two main transition systems for dependency parsing: **arc-standard**, which is closely related to shift-reduce, and **arc-eager**, which adds an additional action that can simplify derivations (Abney and Johnson, 1991). In both cases, transitions are between **configurations** that are represented as triples, $C = (\sigma, \beta, A)$, where σ is the stack, β is the input buffer, and A is the list of arcs that have been created (Nivre, 2008). In the initial configuration,

$$C_{\text{initial}} = ([\text{ROOT}], \boldsymbol{w}, \varnothing), \tag{11.16}$$

indicating that the stack contains only the special node ROOT, the entire input is on the buffer, and the set of arcs is empty. An accepting configuration is,

$$C_{\text{accept}} = ([\text{ROOT}], \varnothing, A), \tag{11.17}$$

where the stack contains only ROOT, the buffer is empty, and the arcs A define a spanning tree over the input. The arc-standard and arc-eager systems define a set of transitions between configurations, which are capable of transforming an initial configuration into an accepting configuration. In both of these systems, the number of actions required to parse an input grows linearly in the length of the input, making transition-based parsing considerably more efficient than graph-based methods.

Arc-standard

The **arc-standard** transition system is closely related to shift-reduce, and to the LR algorithm that is used to parse programming languages (Aho et al., 2006). It includes the following classes of actions:

- SHIFT: move the first item from the input buffer on to the top of the stack,

$$(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A), \tag{11.18}$$

[4] Transition systems also exist for non-projective dependency parsing (e.g., Nivre, 2008).

where we write $i|\beta$ to indicate that i is the leftmost item in the input buffer, and $\sigma|i$ to indicate the result of pushing i on to stack σ.

- ARC-LEFT: create a new left-facing arc of type r between the item on the top of the stack and the first item in the input buffer. The head of this arc is j, which remains at the front of the input buffer. The arc $j \xrightarrow{r} i$ is added to A. Formally,

$$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, j|\beta, A \oplus j \xrightarrow{r} i), \quad [11.19]$$

  where r is the label of the dependency arc, and $\oplus$ concatenates the new arc $j \xrightarrow{r} i$ to the list A.

- ARC-RIGHT: creates a new right-facing arc of type r between the item on the top of the stack and the first item in the input buffer. The head of this arc is i, which is "popped" from the stack and pushed to the front of the input buffer. The arc $i \xrightarrow{r} j$ is added to A. Formally,

$$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, i|\beta, A \oplus i \xrightarrow{r} j), \quad [11.20]$$

  where again r is the label of the dependency arc.

Each action has preconditions. The SHIFT action can be performed only when the buffer has at least one element. The ARC-LEFT action cannot be performed when the root node ROOT is on top of the stack, since this node must be the root of the entire tree. The ARC-LEFT and ARC-RIGHT remove the modifier words from the stack (in the case of ARC-LEFT) and from the buffer (in the case of ARC-RIGHT), so it is impossible for any word to have more than one parent. Furthermore, the end state can only be reached when every word is removed from the buffer and stack, so the set of arcs is guaranteed to constitute a spanning tree. An example arc-standard derivation is shown in Table 11.2.

Arc-eager dependency parsing

In the arc-standard transition system, a word is completely removed from the parse once it has been made the modifier in a dependency arc. At this time, any dependents of this word must have already been identified. Right-branching structures are common in English (and many other languages), with words often modified by units such as prepositional phrases to their right. In the arc-standard system, this means that we must first shift all the units of the input onto the stack, and then work backwards, creating a series of arcs, as occurs in Table 11.2. Note that the decision to shift *bagels* onto the stack guarantees that the prepositional phrase *with lox* will attach to the noun phrase, and that this decision must be made before the prepositional phrase is itself parsed. This has been argued to be cognitively implausible (Abney and Johnson, 1991); from a computational perspective, it means that a parser may need to look several steps ahead to make the correct decision.

	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*.

Arc-eager dependency parsing changes the ARC-RIGHT action so that right dependents can be attached before all of their dependents have been found. Rather than removing the modifier from both the buffer and stack, the ARC-RIGHT action pushes the modifier on to the stack, on top of the head. Because the stack can now contain elements that already have parents in the partial dependency graph, two additional changes are necessary:

- A precondition is required to ensure that the ARC-LEFT action cannot be applied when the top element on the stack already has a parent in A.
- A new REDUCE action is introduced, which can remove elements from the stack if they already have a parent in A:

$$(\sigma|i, \beta, A) \Rightarrow (\sigma, \beta, A). \quad [11.21]$$

As a result of these changes, it is now possible to create the arc *like* $\rightarrow$ *bagels* before parsing the prepositional phrase *with lox*. Furthermore, this action does not imply a decision about whether the prepositional phrase will attach to the noun or verb. Noun attachment is chosen in the parse in Table 11.3, but verb attachment could be achieved by applying the REDUCE action at step 5 or 7.

Projectivity

The arc-standard and arc-eager transition systems are guaranteed to produce projective dependency trees, because all arcs are between the word at the top of the stack and the

	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*.

left-most edge of the buffer (Nivre, 2008). Non-projective transition systems can be constructed by adding actions that create arcs with words that are second or third in the stack (Attardi, 2006), or by adopting an alternative configuration structure, which maintains a list of all words that do not yet have heads (Covington, 2001). In **pseudo-projective dependency parsing**, a projective dependency parse is generated first, and then a set of graph transformation techniques are applied, producing non-projective edges (Nivre and Nilsson, 2005).

Beam search

In "greedy" transition-based parsing, the parser tries to make the best decision at each configuration. This can lead to search errors, when an early decision locks the parser into a poor derivation. For example, in Table 11.2, if ARC-RIGHT were chosen at step 4, then the parser would later be forced to attach the prepositional phrase *with lox* to the verb *likes*. Note that the *likes* $\rightarrow$ *bagels* arc is indeed part of the correct dependency parse, but the arc-standard transition system requires it to be created later in the derivation.

Beam search is a general technique for ameliorating search errors in incremental decoding.[5] While searching, the algorithm maintains a set of partially-complete hypotheses, called a beam. At step t of the derivation, there is a set of k hypotheses, each of which

[5]Beam search is used throughout natural language processing, and beyond. In this text, it appears again in coreference resolution (§ 15.2.4) and machine translation (§ 18.4).

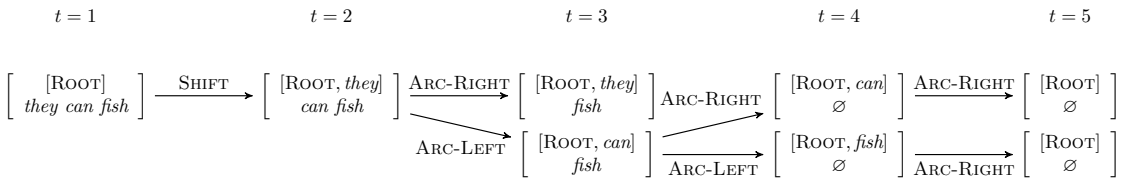


Figure 11.7: Beam search for unlabeled dependency parsing, with beam size $K = 2$. The arc lists for each configuration are not shown, but can be computed from the transitions.

includes a score $s_t^{(k)}$ and a set of dependency arcs $A_t^{(k)}$:

$$h_t^{(k)} = (s_t^{(k)}, A_t^{(k)}) \quad [11.22]$$

Each hypothesis is then "expanded" by considering the set of all valid actions from the current configuration $c_t^{(k)}$, written $\mathcal{A}(c_t^{(k)})$. This yields a large set of new hypotheses. For each action $a \in \mathcal{A}(c_t^{(k)})$, we score the new hypothesis $A_t^{(k)} \oplus a$. The top k hypotheses by this scoring metric are kept, and parsing proceeds to the next step (Zhang and Clark, 2008). Note that beam search requires a scoring function for action *sequences*, rather than individual actions. This issue will be revisited in the next section.

Figure 11.7 shows the application of beam search to dependency parsing, with a beam size of $K = 2$. For the first transition, the only valid action is SHIFT, so there is only one possible configuration at $t = 2$. From this configuration, there are three possible actions. The two best scoring actions are ARC-RIGHT and ARC-LEFT, and so the resulting hypotheses from these actions are on the beam at $t = 3$. From these configurations, there are three possible actions each, but the best two are expansions of the bottom hypothesis at $t = 3$. Parsing continues until $t = 5$, at which point both hypotheses reach an accepting state. The best-scoring hypothesis is then selected as the parse.

11.3.2 Scoring functions for transition-based parsers

Transition-based parsing requires selecting a series of actions. In greedy transition-based parsing, this can be done by training a classifier,

$$\hat{a} = \underset{a \in \mathcal{A}(c)}{\text{argmax}} \, \Psi(a, c, \boldsymbol{w}; \boldsymbol{\theta}), \quad [11.23]$$

where $\mathcal{A}(c)$ is the set of admissible actions in the current configuration c, $\boldsymbol{w}$ is the input, and Ψ is a scoring function with parameters $\boldsymbol{\theta}$ (Yamada and Matsumoto, 2003).

A feature-based score can be computed, $\Psi(a, c, \boldsymbol{w}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(a, c, \boldsymbol{w})$, using features that may consider any aspect of the current configuration and input sequence. Typical features for transition-based dependency parsing include: the word and part-of-speech of the top

element on the stack; the word and part-of-speech of the first, second, and third elements on the input buffer; pairs and triples of words and parts-of-speech from the top of the stack and the front of the buffer; the distance (in tokens) between the element on the top of the stack and the element in the front of the input buffer; the number of modifiers of each of these elements; and higher-order dependency features as described above in the section on graph-based dependency parsing (see, e.g., Zhang and Nivre, 2011).

Parse actions can also be scored by neural networks. For example, Chen and Manning (2014) build a feedforward network in which the input layer consists of the concatenation of embeddings of several words and tags:

- 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.

Let us call this base layer $\boldsymbol{x}(c, \boldsymbol{w})$, defined as,

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

where $\boldsymbol{v}_{w_{\sigma_1}}$ is the embedding of the first word on the stack, $\boldsymbol{v}_{t_{\beta_2}}$ is the embedding of the part-of-speech tag of the second word on the buffer, and so on. Given this base encoding of the parser state, the score for the set of possible actions is computed through a feedforward network,

$$\boldsymbol{z} = g(\Theta^{(x \to z)} \boldsymbol{x}(c, \boldsymbol{w})) \quad [11.24]$$

$$\psi(a, c, \boldsymbol{w}; \boldsymbol{\theta}) = \Theta_a^{(z \to y)} \boldsymbol{z}, \quad [11.25]$$

where the vector $\boldsymbol{z}$ plays the same role as the features $\boldsymbol{f}(a, c, \boldsymbol{w})$, but is a learned representation. Chen and Manning (2014) use a cubic elementwise activation function, $g(x) = x^3$, so that the hidden layer models products across all triples of input features. The learning algorithm updates the embeddings as well as the parameters of the feedforward network.

11.3.3 Learning to parse

Transition-based dependency parsing suffers from a mismatch between the supervision, which comes in the form of dependency trees, and the classifier's prediction space, which is a set of parsing actions. One solution is to create new training data by converting parse trees into action sequences; another is to derive supervision directly from the parser's performance.

Oracle-based training

A transition system can be viewed as a function from action sequences (derivations) to parse trees. The inverse of this function is a mapping from parse trees to derivations, which is called an **oracle**. For the arc-standard and arc-eager parsing system, an oracle can be computed in linear time in the length of the derivation (Kübler et al., 2009, page 32). Both the arc-standard and arc-eager transition systems suffer from spurious ambiguity: there exist dependency parses for which multiple derivations are possible, such as $1 \leftarrow 2 \rightarrow 3$.The oracle must choose between these different derivations. For example, the algorithm described by Kübler et al. (2009) would first create the left arc $(1 \leftarrow 2)$, and then create the right arc, $(1 \leftarrow 2) \rightarrow 3$; another oracle might begin by shifting twice, resulting in the derivation $1 \leftarrow (2 \rightarrow 3)$.

Given such an oracle, a dependency treebank can be converted into a set of oracle action sequences $\{A^{(i)}\}_{i=1}^{N}$. The parser can be trained by stepping through the oracle action sequences, and optimizing on an classification-based objective that rewards selecting the oracle action. For transition-based dependency parsing, maximum conditional likelihood is a typical choice (Chen and Manning, 2014; Dyer et al., 2015):

$$\mathrm{p}(a \mid c, \boldsymbol{w}) = \frac{\exp \Psi(a, c, \boldsymbol{w}; \boldsymbol{\theta})}{\sum_{a' \in \mathcal{A}(c)} \exp \Psi(a', c, \boldsymbol{w}; \boldsymbol{\theta})} \quad [11.26]$$

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{N} \sum_{t=1}^{|A^{(i)}|} \log \mathrm{p}(a_t^{(i)} \mid c_t^{(i)}, \boldsymbol{w}), \quad [11.27]$$

where $|A^{(i)}|$ is the length of the action sequence $A^{(i)}$.

Recall that beam search requires a scoring function for action sequences. Such a score can be obtained by adding the log-likelihoods (or hinge losses) across all actions in the sequence (Chen and Manning, 2014).

Global objectives

The objective in Equation 11.27 is **locally-normalized**: it is the product of normalized probabilities over individual actions. A similar characterization could be made of non-probabilistic algorithms in which hinge-loss objectives are summed over individual actions. In either case, training on individual actions can be sub-optimal with respect to global performance, due to the **label bias problem** (Lafferty et al., 2001; Andor et al., 2016).

As a stylized example, suppose that a given configuration appears 100 times in the training data, with action a_1 as the oracle action in 51 cases, and a_2 as the oracle action in the other 49 cases. However, in cases where a_2 is correct, choosing a_1 results in a cascade of subsequent errors, while in cases where a_1 is correct, choosing a_2 results in only a single

error. A classifier that is trained on a local objective function will learn to always choose a_1, but choosing a_2 would minimize the overall number of errors.

This observation motivates a global objective, such as the globally-normalized conditional likelihood,

$$\mathrm{p}(A^{(i)} \mid \boldsymbol{w}; \boldsymbol{\theta}) = \frac{\exp \sum_{t=1}^{|A^{(i)}|} \Psi(a_t^{(i)}, c_t^{(i)}, \boldsymbol{w})}{\sum_{A' \in \mathbb{A}(\boldsymbol{w})} \exp \sum_{t=1}^{|A'|} \Psi(a'_t, c'_t, \boldsymbol{w})}, \tag{11.28}$$

where the denominator sums over the set of all possible action sequences, $\mathbb{A}(\boldsymbol{w})$.[6] In the conditional random field model for sequence labeling (§ 7.5.3), it was possible to compute this sum explicitly, using dynamic programming. In transition-based parsing, this is not possible. However, the sum can be approximated using beam search,

$$\sum_{A' \in \mathbb{A}(\boldsymbol{w})} \exp \sum_{t=1}^{|A'|} \Psi(a'_t, c'_t, \boldsymbol{w}) \approx \sum_{k=1}^{K} \exp \sum_{t=1}^{|A^{(k)}|} \Psi(a_t^{(k)}, c_t^{(k)}, \boldsymbol{w}), \tag{11.29}$$

where $A^{(k)}$ is an action sequence on a beam of size K. This gives rise to the following loss function,

$$L(\boldsymbol{\theta}) = -\sum_{t=1}^{|A^{(i)}|} \Psi(a_t^{(i)}, c_t^{(i)}, \boldsymbol{w}) + \log \sum_{k=1}^{K} \exp \sum_{t=1}^{|A^{(k)}|} \Psi(a_t^{(k)}, c_t^{(k)}, \boldsymbol{w}). \tag{11.30}$$

The derivatives of this loss involve expectations with respect to a probability distribution over action sequences on the beam.

*Early update and the incremental perceptron

When learning in the context of beam search, the goal is to learn a decision function so that the gold dependency parse is always reachable from at least one of the partial derivations on the beam. (The combination of a transition system (such as beam search) and a scoring function for actions is known as a **policy**.) To achieve this, we can make an **early update** as soon as the oracle action sequence "falls off" the beam, even before a complete analysis is available (Collins and Roark, 2004; Daumé III and Marcu, 2005). The loss can be based on the best-scoring hypothesis on the beam, or the sum of all hypotheses (Huang et al., 2012).

For example, consider the beam search in Figure 11.7. In the correct parse, *fish* is the head of dependency arcs to both of the other two words. In the arc-standard system,

[6] Andor et al. (2016) prove that the set of globally-normalized conditional distributions is a strict superset of the set of locally-normalized conditional distributions, and that globally-normalized conditional models are therefore strictly more expressive.

this can be achieved only by using SHIFT for the first two actions. At $t = 3$, the oracle action sequence has fallen off the beam. The parser should therefore stop, and update the parameters by the gradient $\frac{\partial}{\partial \boldsymbol{\theta}} L(A^{(i)}_{1:3}, \{A^{(k)}_{1:3}\}; \boldsymbol{\theta})$, where $A^{(i)}_{1:3}$ is the first three actions of the oracle sequence, and $\{A^{(k)}_{1:3}\}$ is the beam.

This integration of incremental search and learning was first developed in the **incremental perceptron** (Collins and Roark, 2004). This method updates the parameters with respect to a hinge loss, which compares the top-scoring hypothesis and the gold action sequence, up to the current point t. Several improvements to this basic protocol are possible:

- As noted earlier, the gold dependency parse can be derived by multiple action sequences. Rather than checking for the presence of a single oracle action sequence on the beam, we can check if the gold dependency parse is *reachable* from the current beam, using a **dynamic oracle** (Goldberg and Nivre, 2012).

- By maximizing the score of the gold action sequence, we are training a decision function to find the correct action given the gold context. But in reality, the parser will make errors, and the parser is not trained to find the best action given a context that may not itself be optimal. This issue is addressed by various generalizations of incremental perceptron, known as **learning to search** (Daumé III et al., 2009). Some of these methods are discussed in chapter 15.

11.4 Applications

Dependency parsing is used in many real-world applications: any time you want to know about pairs of words which might not be adjacent, you can use dependency arcs instead of regular expression search patterns. For example, you may want to match strings like *delicious pastries*, *delicious French pastries*, and *the pastries are delicious*.

It is possible to search the Google n-grams corpus by dependency edges, finding the trend in how often a dependency edge appears over time. For example, we might be interested in knowing when people started talking about *writing code*, but we also want *write some code*, *write good code*, *write all the code*, etc. The result of a search on the dependency edge *write* $\rightarrow$ *code* is shown in Figure 11.8. This capability has been applied to research in digital humanities, such as the analysis of gender in Shakespeare Muralidharan and Hearst (2013).

A classic application of dependency parsing is **relation extraction**, which is described

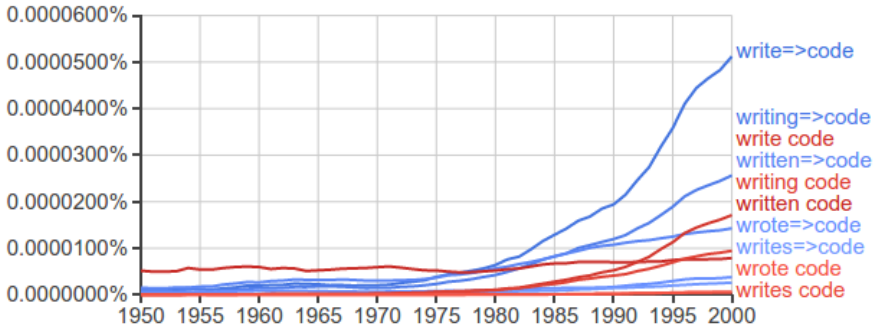


Figure 11.8: Google n-grams results for the bigram *write code* and the dependency arc *write* => *code* (and their morphological variants)

in chapter 17. The goal of relation extraction is to identify entity pairs, such as

(MELVILLE, MOBY-DICK)
(TOLSTOY, WAR AND PEACE)
(MARQUÉZ, 100 YEARS OF SOLITUDE)
(SHAKESPEARE, A MIDSUMMER NIGHT'S DREAM),

which stand in some relation to each other (in this case, the relation is authorship). Such entity pairs are often referenced via consistent chains of dependency relations. Therefore, dependency paths are often a useful feature in supervised systems which learn to detect new instances of a relation, based on labeled examples of other instances of the same relation type (Culotta and Sorensen, 2004; Fundel et al., 2007; Mintz et al., 2009).

Cui et al. (2005) show how dependency parsing can improve automated question answering. Suppose you receive the following query:

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

The corpus contains this sentence:

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

The location of *Wisconsin* in the surface form of this string makes it a poor match for the query. However, in the dependency graph, there is an edge from *produce* to *Wisconsin* in both the question and the potential answer, raising the likelihood that this span of text is relevant to the question.

A final example comes from sentiment analysis. As discussed in chapter 4, the polarity of a sentence can be reversed by negation, e.g.

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

By tracking the sentiment polarity through the dependency parse, we can better identify the overall polarity of the sentence, determining when key sentiment words are reversed (Wilson et al., 2005; Nakagawa et al., 2010).

Additional resources

More details on dependency grammar and parsing algorithms can be found in the manuscript by Kübler et al. (2009). For a comprehensive but whimsical overview of graph-based dependency parsing algorithms, see Eisner (1997). Jurafsky and Martin (2019) describe an **agenda-based** version of beam search, in which the beam contains hypotheses of varying lengths. New hypotheses are added to the beam only if their score is better than the worst item currently on the beam. Another search algorithm for transition-based parsing is **easy-first**, which abandons the left-to-right traversal order, and adds the highest-scoring edges first, regardless of where they appear (Goldberg and Elhadad, 2010). Goldberg et al. (2013) note that although transition-based methods can be implemented in linear time in the length of the input, naïve implementations of beam search will require quadratic time, due to the cost of copying each hypothesis when it is expanded on the beam. This issue can be addressed by using a more efficient data structure for the stack.

Exercises

1. The dependency structure $1 \leftarrow 2 \rightarrow 3$, with 2 as the root, can be obtained from more than one set of actions in arc-standard parsing. List both sets of actions that can obtain this parse. Don't forget about the edge $\text{ROOT} \rightarrow 2$.

2. This problem develops the relationship between dependency parsing and lexicalized context-free parsing. Suppose you have a set of unlabeled arc scores $\{\psi(i \rightarrow j)\}_{i,j=1}^{M} \cup \{\psi(\text{ROOT} \rightarrow j)\}_{j=1}^{M}$.

   a) Assuming each word type occurs no more than once in the input ($(i \neq j) \Rightarrow (w_i \neq w_j)$), how would you construct a weighted lexicalized context-free grammar so that the score of *any* projective dependency tree is equal to the score of some equivalent derivation in the lexicalized context-free grammar?

   b) Verify that your method works for the example *They fish*.

   c) Does your method require the restriction that each word type occur no more than once in the input? If so, why?

   d) *If your method required that each word type occur only once in the input, show how to generalize it.

3. In arc-factored dependency parsing of an input of length M, the score of a parse is the sum of M scores, one for each arc. In second order dependency parsing, the total score is the sum over many more terms. How many terms are the score of the parse for Figure 11.2, using a second-order dependency parser with grandparent and sibling features? Assume that a child of ROOT has no grandparent score, and that a node with no siblings has no sibling scores.

4. a) In the worst case, how many terms can be involved in the score of an input of length M, assuming second-order dependency parsing? Describe the structure of the worst-case parse. As in the previous problem, assume that there is only one child of ROOT, and that it does not have any grandparent scores.

   b) What about third-order dependency parsing?

5. Provide the UD-style unlabeled dependency parse for the sentence *Xi-Lan eats shoots and leaves*, assuming *shoots* is a noun and *leaves* is a verb. Provide arc-standard and arc-eager derivations for this dependency parse.

6. Compute an upper bound on the number of successful derivations in arc-standard shift-reduce parsing for unlabeled dependencies, as a function of the length of the input, M. Hint: a lower bound is the number of projective decision trees, $\frac{1}{M+1}\binom{3M-2}{M-1}$ (Zhang, 2017), where $\binom{a}{b} = \frac{a!}{(a-b)!b!}$.

7. The **label bias problem** arises when a decision is locally correct, yet leads to a cascade of errors in some situations (§ 11.3.3). Design a scenario in which this occurs. Specifically:

   - Assume an arc-standard dependency parser, whose action classifier considers only the words at the top of the stack and at the front of the input buffer.
   - Design two examples, which both involve a decision with identical features.
     - In one example, shift is the correct decision; in the other example, arc-left or arc-right is the correct decision.
     - In one of the two examples, a mistake should lead to at least two attachment errors.
     - In the other example, a mistake should lead only to a single attachment error.

For the following exercises, run a dependency parser, such as Stanford's CoreNLP parser, on a large corpus of text (at least 10^5 tokens), such as `nltk.corpus.webtext`.

8. The dependency relation NMOD:POSS indicates possession. Compute the top ten words most frequently possessed by each of the following pronouns: *his*, *her*, *our*, *my*, *your*, and *their* (inspired by Muralidharan and Hearst, 2013).

9. Count all pairs of words grouped by the CONJ relation. Select all pairs of words (i, j) for which i and j each participate in CONJ relations at least five times. Compute and sort by the **pointwise mutual information**, which is defined in § 14.3 as,

$$\text{PMI}(i, j) = \log \frac{\text{p}(i, j)}{\text{p}(i)\text{p}(j)}. \quad [11.31]$$

Here, $\text{p}(i)$ is the fraction of CONJ relations containing word i (in either position), and $\text{p}(i, j)$ is the fraction of such relations linking i and j (in any order).

10. In § 4.2, we encountered lexical semantic relationships such as **synonymy** (same meaning), **antonymy** (opposite meaning), and **hypernymy** (i is a special case of j). Another relevant relation is **co-hypernymy**, which means that i and j share a hypernym. Of the top 20 pairs identified by PMI in the previous problem, how many participate in synsets that are linked by one of these four relations? Use WORDNET to check for these relations, and count a pair of words if any of their synsets are linked.