Day 4

Syntax and Parsing

In this lab we will implement some exercises related with *parsing*.

4.1 Phrase-based Parsing

4.1.1 Context Free Grammars

Let $\mathcal{T}$ be an *alphabet* (*i.e.*, a finite set of symbols), and denote by $\mathcal{T}^*$ its Kleene closure, *i.e.*, the infinite set of strings produced with those symbols:

$$\mathcal{T}^* = \varnothing \cup \mathcal{T} \cup \mathcal{T}^2 \cup \dots$$

A *language* L is a subset of $\mathcal{T}^*$. The "complexity" of a language L can be accessed in terms of how hard it is to construct a machine (an *automaton*) capable of distinguishing the words in L from the elements of $\mathcal{T}^*$ which are not in L.[1] If L is finite, a very simple automaton can be built which just memorizes the strings in L. The next simplest case is that of *regular languages*, which are recognizable by *finite state machines*. These are the languages that can be expressed by regular expressions. An example (where $\mathcal{T} = \{a, b\}$) is the language $L = \{ab^n aa^n \mid n \in \mathbb{N}\}$, which corresponds to the regular expression $ab * a+$. *Hidden Markov models* (studied in previous lectures) can be seen as a stochastic version of finite state machines.

A step higher in the hierarchy of languages leads to *context-free languages*, which are more complex than regular languages. These are languages that are generated by *context-free grammars*, and recognizable by *push-down automata* (which are slightly more complex than finite state machines). In this section we describe context-free grammars and how they can be made probabilistic.

[1]We recommend the classic book by Hopcroft et al. (1979) for a thorough introduction on the subject of automata theory and formal languages.

This will yield models that are more powerful than hidden Markov models, and are specially amenable for modeling the syntax of natural languages.[2]

A *context-free grammar* (CFG) is a tuple $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \texttt{S} \rangle$ where:

1. $\mathcal{N}$ is a finite set of *non-terminal* symbols. Elements of $\mathcal{N}$ are denoted by upper case letters $(A, B, C, \ldots)$. Each non-terminal symbol is a syntactic category: it represents a different type of phrase or clause in the sentence.

2. $\mathcal{T}$ is a finite set of *terminal* symbols (disjoint from $\mathcal{N}$). Elements of $\mathcal{T}$ are denoted by lower case letters $(a, b, c, \ldots)$. Each terminal symbol is a surface word: terminal symbols make up the actual content of sentences. The set $\mathcal{T}$ is called the *alphabet* of the language defined by the grammar G.

3. $\mathcal{R}$ is a set of *production rules*, *i.e.*, a finite relation from $\mathcal{N}$ to $(\mathcal{N} \cup \mathcal{T})^*$. G is said to be in Chomsky normal form (CNF) if production rules in $\mathcal{R}$ are either of the form $A \to BC$ or $A \to a$.

4. $\texttt{S}$ is a *start symbol*, used to represent the whole sentence. It must be an element of $\mathcal{N}$.

Any CFG can be transformed to be in CNF without loosing any expressive power in terms of the language it generates. Hence, we henceforth assume that G is in CNF without loss of generality.

To see how CFGs can model the syntax of natural languages, consider the following sentence,

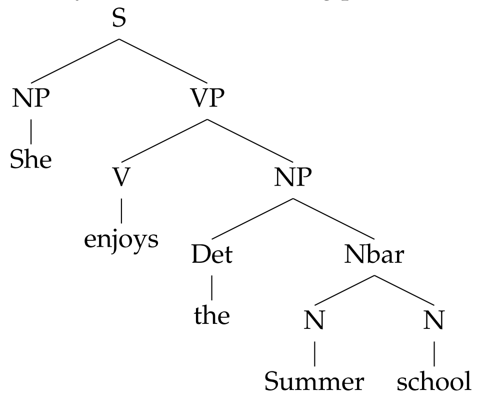
```
She enjoys the Summer school.
```

along with a grammar (in CNF) with the following production rules:

```
S --> NP VP
NP --> Det N
NP --> She
VP --> V NP
V --> enjoys
Det --> the
Nbar --> N N
N --> Summer
N --> school
```

[2]This does not mean that natural languages are context free. There is an immense body of work on grammar formalisms that relax the "context-free" assumption, and those formalisms have been endowed with a probabilistic framework as well. Examples are: lexical functional grammars, head-driven phrase structured grammars, combinatorial categorial grammars, tree adjoining grammars, etc. Some of these formalisms are *mildly context sensitive*, a relaxation of the "context-free" assumption which still allows polynomial parsing algorithms. There is also equivalence in expressive power among several of these formalisms.

With this grammar, we may derive the following parse tree:



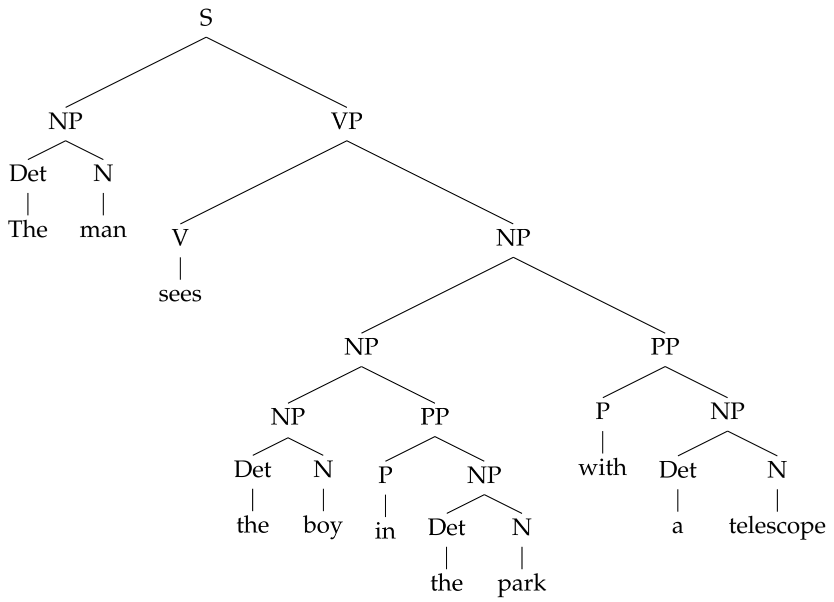
4.1.2 Ambiguity

A fundamental characteristic of natural languages is *ambiguity*. For example, consider the sentence

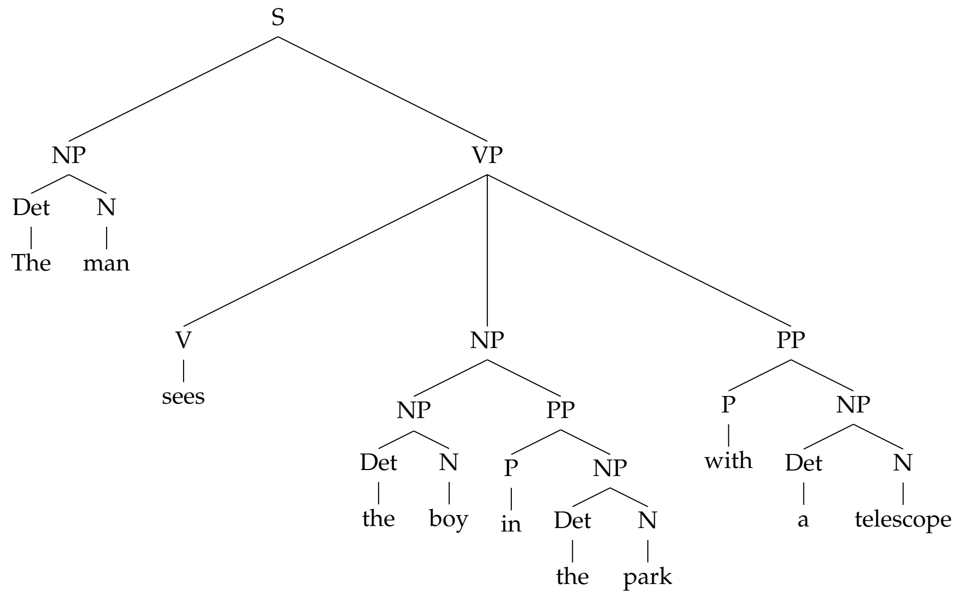
```
The man sees the boy in the park with a telescope.
```

for which all the following parse trees are plausible interpretations:

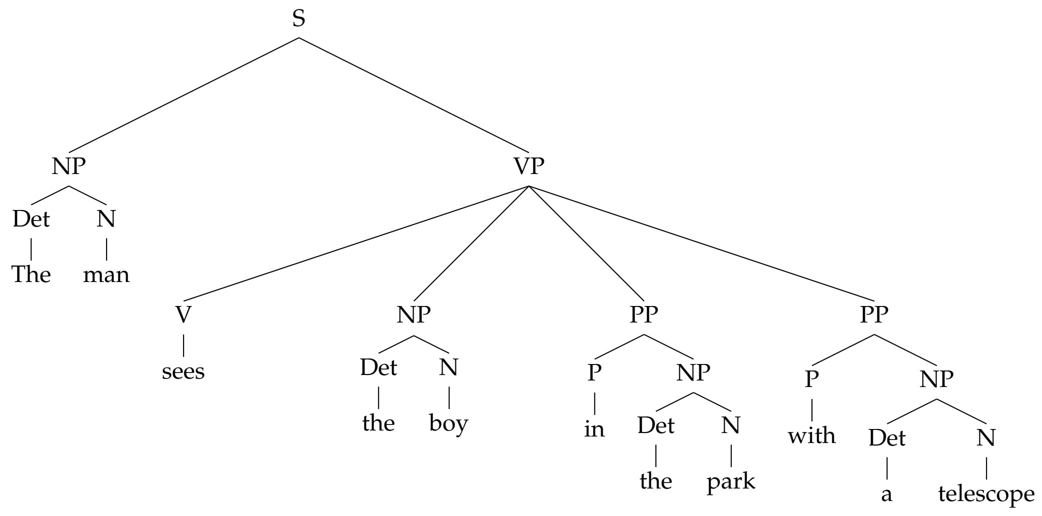
1. The boy is in a park and he has a telescope:



2. The boy is in a park, and the man sees him using a telescope as an instrument:



3. The man is in the park and he has a telescope, through which he sees a boy somewhere:



The ambiguity is caused by the several places to each the prepositional phrase could be attached. This kind of syntactical ambiguity (*PP-attachment*) is one of the most frequent in natural language.

4.1.3 Probabilistic Context-Free Grammars

A *probabilistic context-free grammar* is a tuple $G_{\boldsymbol{\theta}} = \langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \mathtt{S}, \boldsymbol{\theta} \rangle$, where $\langle \mathcal{N}, \mathcal{T}, \mathcal{R}, \mathtt{S} \rangle$ is a CFG and $\boldsymbol{\theta}$ is a vector of parameters, one per each production rule in $\mathcal{R}$. Assuming that the grammar is in CNF, each rule of the kind $Z \to XY$ is endowed a conditional probability

$$\theta_{Z\to XY} = P_{\boldsymbol{\theta}}(XY|Z),$$

and each unary rule of the kind $Z \to w$ is endowed with a conditional probability

$$\theta_{Z\to w} = P_{\boldsymbol{\theta}}(w|Z).$$

For these conditional probabilities to be well defined, the entries of $\boldsymbol{\theta}$ must be non-negative and need to normalize properly for each $Z \in \mathcal{N}$:

$$\sum_{X,Y\in\mathcal{N}} \theta_{Z\to XY} + \sum_{w\in\mathcal{T}} \theta_{Z\to w} = 1.$$

Let s be a string and t a parse tree derivation for s. For each $r \in \mathcal{R}$, let $n_r(t,s)$ be the number of times production rule r appears in the derivation. According to this generative model, the joint probability of t and s factors as the product of the conditional probabilities above:

$$P(t,s) = \prod_{r\in\mathcal{R}} \theta_r^{n_r(t,s)}.$$

For example, for the sentence above (*She enjoys the Summer school*) this probability would be

$$\begin{aligned} P(t,s) \quad = \quad & P(\texttt{NP VP}|\texttt{S}) \times P(\texttt{She}|\texttt{NP}) \times P(\texttt{V NP}|\texttt{VP}) \times P(\texttt{enjoys}|\texttt{V}) \\ & \times P(\texttt{Det Nbar}|\texttt{NP}) \times P(\texttt{the}|\texttt{Det}) \times P(\texttt{N N}|\texttt{Nbar}) \\ & \times P(\texttt{Summer}|\texttt{N}) \times P(\texttt{school}|\texttt{N}). \end{aligned} \tag{4.1}$$

When a sentence is ambiguous, the most likely parse tree can be obtained by maximizing the conditional probability $P(t|s)$; this quantity is proportional to $P(t,s)$ and therefore the latter quantity can be maximized. The number of possible parse trees, however, grows exponentially with the sentence length, rendering a direct maximization intractable. Fortunately, a generalization of the Viterbi algorithm exists which uses dynamic programming to carry out this computation. This is the subject of the next section.

4.1.4 The CKY Parsing Algorithm

One of the most widely algorithm for parsing natural language sentences is the Cocke-Kasami-Younger (CKY) algorithm. Given a grammar in CNF with $|\mathcal{R}|$ production rules, its runtime complexity for parsing a sentence of length N is $O(N^3|\mathcal{R}|)$. We present here a simple extension of the CKY algorithm that

Algorithm 12 CKY algorithm

```
1: input: probabilistic CFG $G_\theta$ in CNF, sentence $s = w_1 \dots w_N$
2:
3: {Initialization}
4: for $i = 1$ to $N$ do
5:   for each production rule $r \in \mathcal{R}$ of the form $Z \to w_i$ do
6:     $\delta(i, i, Z) = \theta_{Z \to w_i}$
7:   end for
8: end for
9:
10: {Induction}
11: for $i = 2$ to $N$ do {$i$ is length of span}
12:   for $j = 1$ to $N - i + 1$ do {$j$ is start of span}
13:     for each non-terminal $Z \in \mathcal{N}$ do
14:       Set partial probability:
            $$\delta(j, j+i-1, Z) = \max_{\substack{X,Y \\ j<k<j+i}} \delta(j, k-1, X) \times \delta(k, j+i-2, Y) \times \theta_{Z \to XY}$$
15:       Store backpointer:
            $$\psi(j, j+i-1, Z) = \arg\max_{\substack{X,Y \\ j<k<j+i}} \delta(j, k-1, X) \times \delta(k, j+i-2, Y) \times \theta_{Z \to XY}$$
16:     end for
17:   end for
18: end for
19:
20: {Termination}
21: $P(s, \hat{t}) = \delta(1, N, \mathtt{S})$
22: Backtrack through $\psi$ to obtain most likely parse tree $\hat{t}$
```

obtains the most likely parse tree of a sentence, along with its probability.[3] This is displayed in Alg. 12.

Exercise 4.1 *In this simple exercise, you will see the CKY algorithm in action. There is a Javascript applet that illustrates how CKY works (in its non-probabilistic form). Go to* `http://www.diotavelli.net/people/void/demos/cky.html`*, and observe carefully the several steps taken by the algorithm. Write down a small grammar in CNF that yields multiple parses for the ambiguous sentence* The man saw the boy in the park with a telescope*, and run the demo for this particular sentence. What*

[3]Similarly, the forward-backward algorithm for computing posterior marginals in sequence models can be extended for context-free parsing. It takes the name *inside-outside algorithm*. See Manning and Schütze (1999) for details.

would happen in the probabilistic form of CKY?

4.1.5 Learning the Grammar

There is an immense body of work on *grammar induction* using probabilistic models (see *e.g.*, Manning and Schütze (1999) and the references therein, as well as the most recent works of Klein and Manning (2002); Smith and Eisner (2005b); Cohen et al. (2008)): this is the problem of learning the parameters of a grammar from plain sentences only. This can be done in an EM fashion (like in sequence models), except that the forward-backward algorithm is replaced by inside-outside. Unfortunately, the performance of unsupervised parsers is far from good, at present days. Much better results have been produced by supervised systems, which, however, require expensive annotation in the form of *treebanks*: this is a corpus of sentences annotated with their corresponding syntactic trees. The following is an example of an annotated sentence in one of the most widely used treebanks, the *Penn Treebank* (`http://www.cis.upenn.edu/~treebank/`):

```
( (S
    (NP-SBJ (NNP BELL) (NNP INDUSTRIES) (NNP Inc.) )
    (VP (VBD increased)
      (NP (PRP$ its) (NN quarterly) )
      (PP-DIR (TO to)
        (NP (CD 10) (NNS cents) ))
      (PP-DIR (IN from)
        (NP
          (NP (CD seven) (NNS cents) )
          (NP-ADV (DT a) (NN share) ))))
    (. .) ))
```

Exercise 4.2 *This exercise will show you that real-world sentences can have complicated syntactic structures. There is a parse tree visualizer in* `http://www.ark.cs.cmu.edu/treeviz/`*. Go to your local* `data/treebanks` *folder and open the file* `PTB_excerpt.txt`*. Copy a few sentences from there, and examine their parse trees in the visualizer.*

A treebank makes possible to learn a parser in a supervised fashion. The simplest way is via a generative approach. Instead of counting transition and observation events of an HMM (as we did for sequence models), we now need to count production rules and symbol occurrences to estimate the parameters of a probabilistic context-free grammar. While performance would be much better than that of unsupervised parsers, it would still be rather poor. The reason is that the model we have described so far is oversimplistic: it makes too strong independence assumptions. In this case, the Markovian assumptions are:

1. Each terminal symbol w in some position i is independent of everything else given that it was derived by the rule $Z \to w$ (i.e., given its parent Z);
2. Each pair of non-terminal symbols X and Y spanning positions i to j, with split point k, is independent of everything else given that they were derived by the rule $Z \to XY$ (i.e., given their parent Z).

The next section describes some model refinements that complicate the problem of parameter estimation, but usually allow for a dramatic improvement on the quality of the parser.

4.1.6 Model Refinements

A number of refinements has been made that yield more accurate parsers. We mention just a few:

Parent annotation. This strategy splits each non-terminal symbol in the grammar (*e.g.* Z) by annotating it with all its possible parents (*e.g.* creates nodes $Z^X, Z^Y, \ldots$ every time production rules like $X \to Z\cdot$, $X \to \cdot Z$, $Y \to Z\cdot$, or $Y \to \cdot Z$ exist in the original grammar). This increases the vertical Markovian length of the model, hence weakening the independence assumptions. Parent annotation was initiated by Johnson (1998) and carried on in the unlexicalized parsers of Klein and Manning (2003) and follow-up works.

Lexicalization. A particular weakness of PCFGs is that they ignore word context for interior nodes in the parse tree. Yet, this context is relevant in determining the production rules that should be invoked in the derivation. A way of overcoming this limitation is by *lexicalizing* parse trees, *i.e.*, annotating each phrase node with the lexical item (word) which governs that phrase: this is called the *head* word of the phrase. Fig. 4.1 shows an example of a lexicalized parse tree. To account for lexicalization, each non-terminal symbol in the grammar (*e.g.* Z) is split into many symbols, each annotated with a word that may govern that phrase (*e.g.* $Z^{w_1}, Z^{w_2}, \ldots$). This greatly increases the size of the grammar, but it has a significant impact in performance. A string of work involving lexicalized PCFGs includes Magerman (1995); Charniak (1997); Collins (1999).

Discriminative models. Similarly as in sequence models (where it was shown how to move from an HMM to a CRF), we may abandon our generative model and consider a discriminative one. An advantage of doing that is that it becomes much easier to adopt non-local input features (*i.e.*, features that depend arbitrarily on the surface string), for example the kind of features obtained via lexicalization, and much more. The CKY parsing algorithm can still be used for decoding, provided the feature vector decompose according to *non-terminal symbols* and *production rules*.

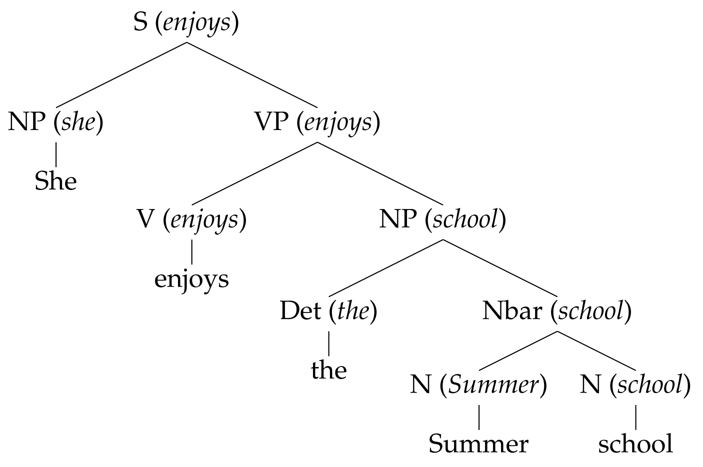


Figure 4.1: A lexicalized parse tree for the sentence *She enjoys the Summer school*.

In this case, productions and non-terminals will have a score which does not correspond to a log-probability; the partition function and the posterior marginals can be computed with the inside-outside algorithm. See Taskar et al. (2004) for an application of structured SVMs to parsing, and Finkel et al. (2008) for an application of CRFs.

Latent variables. Splitting the variables in the grammar by introducing latent variables appears as an alternative to lexicalization and parent annotation. There is a string of work concerning latent variable grammars, either for the generative and discriminative cases: Petrov and Klein (2007, 2008a,b). Some related work also considers coarse-to-fine parsing, which iteratively applies more and more refined models: Charniak et al. (2006).

History-based parsers. Finally, there is a totally different line of work which models parsers as a sequence of greedy shift-reduce decisions made by a push-down automaton (Ratnaparkhi, 1999; Henderson, 2003). When discriminative models are used, arbitrary conditioning can be done on past decisions made by the automaton, allowing to include features that are difficult to handle by the other parsers. This comes at the price of greediness in the decisions taken, which implies suboptimality in maximizing the desired objective function.

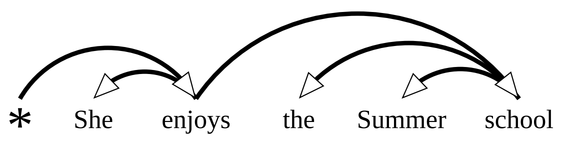


Figure 4.2: A dependency tree for the sentence *She enjoys the Summer school*. Note the additional dummy root symbol (*) which is included for convenience.

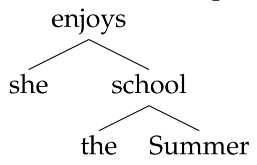
4.2 Dependency Parsing

4.2.1 Motivation

Consider again the sentence

She enjoys the Summer school.

along with the lexicalized parse tree displayed in Fig. 4.1. If we drop the phrase constituents and keep only the head words, the parse tree would become:



This representation is called a *dependency tree*; it can be alternatively represented as shown in Fig. 4.2. Dependency trees retain the lexical relationships involved in lexicalized phrase-based parse trees. However, they drop phrasal constituents, which render non-terminal nodes unnecessary. This has computational advantages (no grammar constant is involved in the complexity of the parsing algorithms) as well as design advantages (no grammar is necessary, and treebank annotations are way simpler, since no internal constituents need to be annotated). It also shifts the focus from internal syntactic structures and generative grammars (Chomsky, 1965) to lexical and transformational grammars (Tesnière, 1959; Hudson, 1984; Melčuk, 1988; Covington, 1990). Arcs connecting words are called *dependency links* or *dependency arcs*. In a arc $\langle h, m \rangle$, the source word h is called the *head* and the target word m is called the *modifier*.

4.2.2 Projective and Non-projective Parsing

Dependency trees constructed using the method just described (*i.e.*, lexicalization of context-free phrase-based trees) always satisfy the following properties:

1. Each word (excluding the dummy root symbol) has exactly one parent.

2. The dummy root symbol has no parents.

3. There are no cycles.

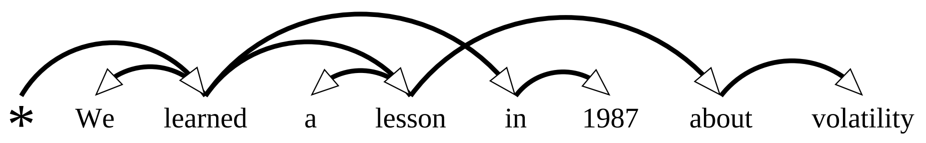


Figure 4.3: A non-projective parse tree.

4. The dummy root symbol has exactly one child.

5. All arcs are *projective*. This means that for any arc $\langle h, m \rangle$, all words in its span (*i.e.*, all words lying between h and m) are descendents from h (i.e. there is a directed path of dependency links from h to such word).

Conditions 1–3 ensure that the set of dependency links form a well-formed tree, rooted in the dummy symbol, which spans all the words of the sentence. Condition 4 requires that there is a single link departing from the root. Finally, a tree satisfying condition 5 is said *projective*: it implies that arcs cannot cross (*e.g.*, we cannot have arcs $\langle h, m \rangle$ and $\langle h', m' \rangle$ such that $h < h' < m < m'$).

In many languages (*e.g.*, those which have free-order) we would like to relax the assumption that all trees must be projective. Even in languages which have fixed word order (such as English) there are syntactic phenomena which are awkward to characterize using projective trees arising from the context-free assumption. Usually, such phenomena are characterized with additional linguistic artifacts (e.g., traces, Wh-movement, *etc.*). An example is the sentence (extracted from the Penn Treebank)

We learned a lesson in 1987 about volatility.

There, the prepositional phrase *in 1987* should be attached to the verb phrase headed by *learned* (since this is *when* we learned the lesson), but the other prepositional phrase *about volatility* should be attached to the noun phrase headed by *lesson* (since the *lesson* was about volatility). To explain such phenomenon, context-free grammars need to use additional machinery which allows words to be scrambled (in this case, via a movement transformation and the consequent insertion of a trace). In the dependency-based formalism, we can get rid of all those artifacts altogether by allowing *non-projective* parse trees. These are trees that satisfy conditions 1–3 above, but not necessarily conditions 4 or 5.[4] The dependency tree in Fig. 4.3 is non-projective: note that the arc $\langle \mathit{lesson}, \mathit{about} \rangle$ is not projective.

[4]It is also common to impose conditions 1–4, in which case the tree need not be projective, but it must have a single link departing from the root. The algorithms to be described below can be adapted for this case.

We end this section by mentioning that dependency trees can have their arcs labeled, to provide more detailed syntactic information. For example, the arc $\langle enjoys, She\rangle$ could be labeled as `SUBJ` to denote that the modifier *She* has a subject function, and the arc $\langle enjoys, school\rangle$ could be labeled as `OBJ` to denote that the modifier *school* has an object function. For simplicity, we resort to *unlabeled* trees, which just convey the backbone structure. The cope with the labels, one can use either a joint model that infers the backbone and labels altogether, or to have a two-stage approach that first gets the backbone structure, and then the arc labels.

4.2.3 Algorithms for Projective Dependency Parsing

We now turn our attention to *algorithms* for obtaining a dependency parse tree. We start by considering a simple kind of models which are called *arc-factored*. These models assign a score $s_{\boldsymbol{\theta}}(h, m)$ to each possible arc $\langle h, m\rangle$ connecting a pair of words; they then score a particular dependency tree t by summing over the individual scores of the arcs that are present in the tree:

$$\text{score}_{\boldsymbol{\theta}}(t) = \sum_{\langle h,m\rangle \in t} s_{\boldsymbol{\theta}}(h, m).$$

As usual, from the point of view of the parsing algorithm, it does not matter whether the scores come from a generative or discriminative approach, and which features were used to compute the scores. The three important inference tasks are:

1. Obtain the tree with the largest score,
$$\hat{t} = \arg\max_t \text{score}_{\boldsymbol{\theta}}(t).$$

2. Compute the partition function (for a log-linear model),
$$Z(s_{\boldsymbol{\theta}}) = \sum_t \exp(\text{score}_{\boldsymbol{\theta}}(t)).$$

3. Compute the posterior marginals for all the possible arcs (which for a log-linear model is the gradient of the log-partition function),
$$P_{\boldsymbol{\theta}}(\langle h, m\rangle \in t) = \frac{\partial \log Z(s_{\boldsymbol{\theta}})}{\partial s_{\boldsymbol{\theta}}(h, m)}.$$

Exercise 4.3 *In* projective *dependency parsing using arc-factored models, the three tasks above can be solved in time $O(N^3)$. Sketch how the most likely dependency tree can be computed by "adapting" the CKY algorithm. (Hint: note that the CKY algorithm builds larger spans by combining smaller spans, and multiplies their weights by* *the weight of the corresponding production rule. In dependency parsing, each "span" is not represented by a constituent, but rather by the position of its lexical head. Convince yourself that this can only be either the leftmost or the rightmost position, and work out how the two spans can be combined.)*

The instantiation of the CKY algorithm for projective dependency parsing is called Eisner's algorithm (Eisner, 1996). Analogously, the partition function and the posterior marginals can be computed by adapting the inside-outside algorithm.

4.2.4 Algorithms for Non-Projective Dependency Parsing

We turn our attention to *non-projective* dependency parsing. In that case, efficient solutions also exist for the three problems above; interestingly, they are based in combinatorial algorithms which are not related at all with dynamic programming:

- The first problem corresponds to finding the *maximum weighted directed spanning tree* in a directed graph. This problem is well known in combinatorics and can be solved in $O(N^3)$ using Chu-Liu-Edmonds' algorithm (Chu and Liu, 1965; Edmonds, 1967).[5] This has first been noted by McDonald et al. (2005).

- The second and third problems can be solved by invoking another important result in combinatorics, the *matrix-tree theorem* (Tutte, 1984). This fact has been noticed independently by Smith and Smith (2007); Koo et al. (2007); McDonald and Satta (2007). The cost is that of computing a determinant and inverting a matrix, which can be done in time $O(N^3)$. The procedure is as follows. We first consider the directed weighted graph formed by including all the possible dependency links $\langle h, m \rangle$ (including the ones departing from the dummy root symbol, for which $h = 0$ by convention), along with weights given by $\exp(s_{\boldsymbol{\theta}}(h, m))$, and compute its $(N+1)$-by-$(N+1)$ Laplacian matrix $\boldsymbol{L}$ whose entries are:

$$L_{hm} = \begin{cases} \sum_{h'=0}^{N} \exp(s_{\boldsymbol{\theta}}(h', m)), & \text{if } h = m, \\ -\exp(s_{\boldsymbol{\theta}}(h, m))), & \text{otherwise.} \end{cases} \tag{4.2}$$

  Denote by $\boldsymbol{r} = (r_m)_{m=1,\ldots,n}$ the first row of this matrix, transposed, with the first entry removed, i.e., $r_i \triangleq -\exp(s_{\boldsymbol{\theta}}(0, m)))$. Furthermore, denote by $\hat{L}$ the $(0,0)$-minor of L, *i.e.*, the matrix obtained from L by removing the first row and column. Consider the determinant $\det \hat{L}$ and the inverse matrix $[\hat{L}^{-1}]$. Then:

  – the partition function is given by

$$Z(s_{\boldsymbol{\theta}}) = \det \hat{L}; \tag{4.3}$$

[5]There is a asymptotically faster algorithm by Tarjan (1977) which solves the same problem in $O(N^2)$.

– the posterior marginals are given by

$$P_{\boldsymbol{\theta}}(\langle h, m\rangle \in t) = \begin{cases} \exp(s_{\boldsymbol{\theta}}(h,m)) \cdot ([\hat{L}^{-1}]_{mm} - [\hat{L}^{-1}]_{mh}) & \text{if } h \neq 0 \\ \exp(s_{\boldsymbol{\theta}}(0,m)) \cdot [\hat{L}^{-1}]_{mm} & \text{otherwise.} \end{cases} \tag{4.4}$$

Exercise 4.4 *In this exercise you are going to experiment with arc-factored non-projective dependency parsers.*

The CoNLL-X and CoNLL 2008 shared task datasets (Buchholz and Marsi, 2006; Surdeanu et al., 2008) contain dependency treebanks for 14 *languages. In this lab, we are going to experiment with the Portuguese and English datasets. We preprocessed those datasets to exclude all sentences with more than* 15 *words; this yielded the files:*

- `data/deppars/portuguese_train.conll`,
- `data/deppars/portuguese_test.conll`,
- `data/deppars/english_train.conll`,
- `data/deppars/english_test.conll`.

1. *After importing all the necessary libraries, load the Portuguese dataset:*

```
import sys
sys.path.append("parsing/" )

import dependency_parser as depp

dp = depp.DependencyParser()
dp.read_data("portuguese")
```

Observe the statistics which are plotted. How many features are there in total?

2. *We will now have a close look on the* features *that can be used in the parser. Examine the file* `code/parsing/dependency_features.py`*. The following method takes a sentence and computes a vector of features for each possible arc* $\langle h, m\rangle$*:*

```
def create_arc_features(self, instance, h, m, add=False):
    '''Creates features for arc h-->m.'''
```

We grouped the features in several subsets, so that we can conduct some ablation experiments:

- Basic *features that look only at the parts-of-speech of the words that can be connected by an arc;*

- Lexical *features that also look at these words themselves;*
- Distance *features that look at the length and direction of the dependency link* (i.e., *distance between the two words);*
- Contextual *features that look at the context (part-of-speech tags) of the words surrounding h and m.*

In the default configuration, only the basic features are enabled. The total number of features is the quantity observed in the previous question. With this configuration, train the parser by running 10 *epochs of the structured perceptron algorithm:*

```
dp.train_perceptron(10)
dp.test()
```

What is the accuracy obtained in the test set? (Note: the plotted accuracy is the fraction of words whose parent was correctly predicted.)

3. *Repeat the previous exercise by subsequently enabling the lexical, distance and contextual features:*

```
dp.features.use_lexical = True
dp.read_data("portuguese")
dp.train_perceptron(10)
dp.test()

dp.features.use_distance = True
dp.read_data("portuguese")
dp.train_perceptron(10)
dp.test()

dp.features.use_contextual = True
dp.read_data("portuguese")
dp.train_perceptron(10)
dp.test()
```

For each configuration, write down the number of features and test set accuracies. Observe the improvements obtained when more features were added. Feel free to engineer new features!

4. *Which of the three important inference tasks discussed above (computing the most likely tree, computing the partition function, and computing the marginals) need to be performed in the structured perceptron algorithm? What about a maximum entropy classifier, with stochastic gradient descent? Check your answers by looking at the following two methods in* `code/dependency_parser.py`*:*

```
def train_perceptron(self, n_epochs):
...

def train_crf_sgd(self, n_epochs, sigma, eta0 = 0.001):
...
```

Repeat the last exercise by training a maximum entropy classifier, with stochastic gradient descent, using $\lambda = 0.01$ *and a initial stepsize of* $\eta_0 = 0.1$*:*

```
dp.train_crf_sgd(10, 0.01, 0.1)
dp.test()
```

Compare the results with those obtained by the perceptron algorithm.

5. *Train a parser for English using your favourite learning algorithm:*

```
dp.read_data("english")
dp.train_perceptron(10)
dp.test()
```

The predicted trees are placed in the file `data/deppars/english_test.conll.pred`*. To get a sense of which errors are being made, you can check the sentences that differ from the gold standard (*`data/deppars/english_test.conll`*) and visualize those sentences,* e.g.*, in* `http://www.ark.cs.cmu.edu/treeviz/`*.*

4.2.5 Model Refinements

A number of refinements has been made that yield more accurate dependency parsers. We mention just a few:

Sibling and grandparent features. The arc-factored assumption fails to capture correlations between pairs of arcs. The dynamic programming algorithms for the *projective* case can be extended (at some additional cost) to handle features that look at consecutive sibling arcs on the same side of the head (*i.e.*, pairs of arcs of the form $\langle h, m\rangle$ and $\langle h, s\rangle$ with $h < m < s$ or $h > m > s$, such that no arc $\langle h, r\rangle$ exists with r between m and s. This has been done by Eisner and Satta (1999). Similarly, grandparents can also be accommodated with similar extensions (Carreras, 2007). These are called "second-order models."

For the non-projective case, however, any extension beyond the arc-factored model becomes NP-hard (McDonald and Satta, 2007). Yet, approximate algorithms have been proposed to handle "second-order models" that seem to work well: a greedy method (McDonald et al., 2006), loopy belief propagation (Smith and Eisner, 2008), a linear programming relaxation (Martins et al., 2009), and a dual decomposition method (Koo et al., 2010).

Third-order models. For the projective case, third order models have also been considered (Koo and Collins, 2010)

Transition-based parsers. Like in the phrase-based case, there is a totally different line of work which models parsers as a sequence of greedy shift-reduce decisions (Nivre et al., 2006; Huang and Sagae, 2010). These parsers seem to be very fast (expected linear time) and only slightly less accurate than the state-of-the-art. Solutions have been worked out for the non-projective case also (Nivre, 2009).