Spectral Learning of Latent-Variable PCFGs: High-Performance Implementation

Haoran Peng

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2021

Abstract

This work is a Python implementation of spectral learning of latent-variable PCFGs (Cohen et al., 2014, 2013). With code vectorization and a two-stage state splitting procedure, we demonstrate that this implementation is 64 times faster to train and uses 46 times less space (than the previous implementation) while reaching a similar F1 score of 87.80. The testing code is parallelized and JIT-compiled, which makes testing highly efficient. We also stated a way to cache computed data which makes automated hyperparameter tuning possible. The codebase, implemented in 1 language (instead of 4 previously) is much more user-friendly, compact, and maintainable. Finally, this work introduces spectral learning of latent-variable PCFGs from an engineering perspective, which helps inexperienced readers understand this subject quickly.

Acknowledgements

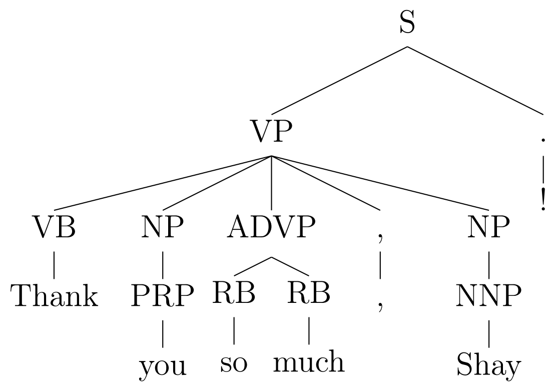
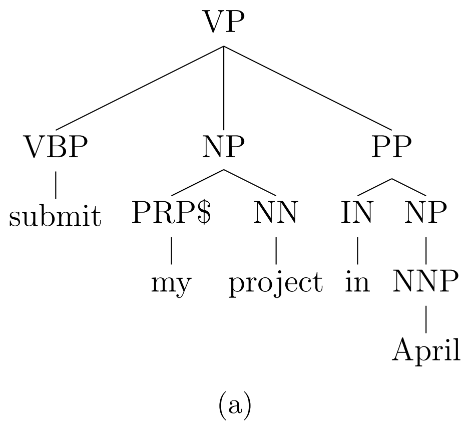


Table of Contents

Chapter 1

Introduction

Probabilistic context-free grammars (PCFGs) has been one of the most important probabilistic models in natural language processing. In the parsing problem, given a sentence, we attempt to group elements in the sentence into syntactic categories and form a parse tree. For example, in Figure 1.1(a), given the sentence *submit my project in April*: the whole sentence is labelled verb phrase (VP), the constituent *my project* is labelled noun phrase (NP), and the constituent *in April* is labelled prepositional phrase (PP), etc.



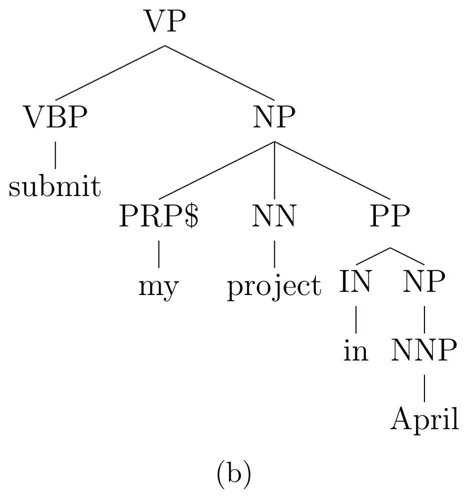


Figure 1.1: Two example parse trees of the sentence *submit my project in April.* The left one is the intuitively correct parse, while the right one exhibits PP-attachment error.

A PCFG in its most basic form (Chapter 2.1), trained on the Penn Treebank Wall Street Journal Dataset (Marcus et al., 1993) (PTB WSJ), can only achieve around 65-75 F1 score on the test set. Its poor accuracy is largely due to its strong independence assumptions:

- **Structural Independence**: the production probability at each node in the tree does not depend on its surrounding context. For example, the probability of PP producing IN NP is dependent **only** on PP itself and not on its context such as its parent VP. But it is known that in natural languages, production probabilities do depend on its context. For example, NP in a subject position is much more likely to produce a pronoun than NP in an object position.

- **Lexical Independence**: the production probabilities do not depend on the words seen.[1] For example, in the sentence *submit my project in April*, *in April* refers to *submit*, instead of *project.* But this dependency is not captured in a PCFG and thus it can make mistakes such as the one shown in Figure 1.1(b) - the tree has very similar productions compared to the correct tree but with PP attached to the incorrect position.

To combat the independence assumptions, researchers have tried to augment PCFGs by splitting each label into different states, with each state corresponding to some context. This way, contextual information can be incorporated into the grammar without sacrificing its context-freeness.

Charniak (1997) and Collins (2003) annotated each node with lexical information, illustrated in Figure 1.2(a). Lexicalized grammars achieved significantly improved F1 scores (87.5-88.5), but an disadvantage was that the resulting grammars were very sparse and required sophisticated smoothing.

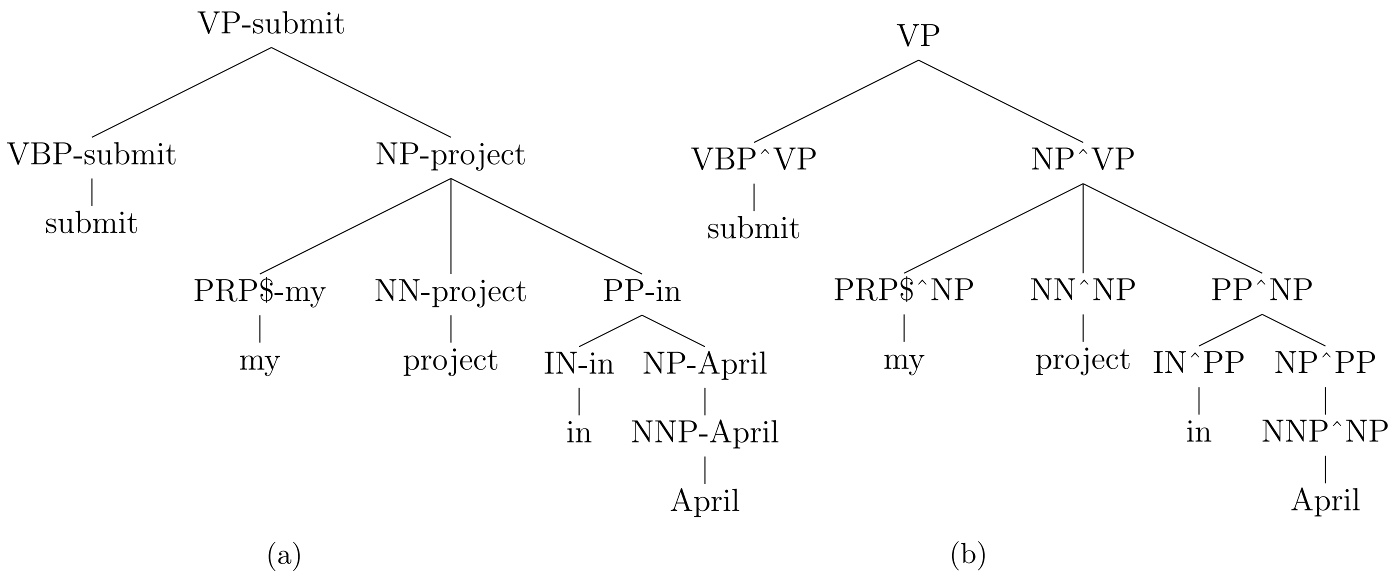


Figure 1.2: (a) Each node is annotated with its head word (b) Each node is annotated with its parent's label

Johnson (1998) showed that the accuracy can be much improved from the baseline if we simply annotate each node with its parent's/siblings' label(s), illustrated in Figure 1.2(b). Klein and Manning (2003) further annotated the nodes with linguistically motivated features and achieved an F1 score of 86.3.

Both lexicalized and unlexicalized grammars required manual annotations. Matsuzaki et al. (2005) proposed a latent-variable grammar where states are split au-

[1]Except for the productions right above the words.

tomatically, illustrated in Figure 1.3. Petrov et al. (2006); Petrov and Klein (2007) improved the F1 score to 90.1 using a hierarchical split-merge procedure.

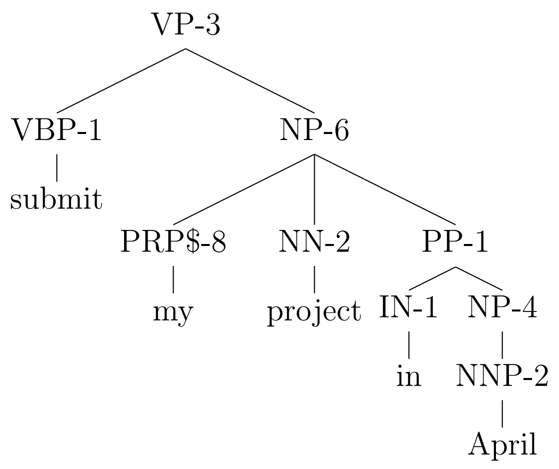


Figure 1.3: Parse tree with latent annotation.

Previous latent-variable grammars were learnt using expectation-maximization which required long training time and did not provide any consistency guarantee. Cohen et al. (2014) showed that a latent-variable grammar can be learnt using spectral learning which is highly efficient and has a PAC-style guarantee. Cohen et al. (2013) achieved an F1 score of 88.05 using spectral learning.

This work is largely based on (Cohen et al., 2014, 2013). The main contributions are as follows:

- A re-implementation of Cohen et al. (2013) in 1 programming language (Python) instead of 4 languages (Python, Matlab, Java, C++). Readers can find MIT licensed source code at `https://github.com/GavinPHR/Spectral-Parser`.
- Chapter 3 and Chapter 6: A step-by-step introduction to spectral learning and parser evaluation, from an engineering perspective. The resources currently available on these topics are quite mathematical and often pose a challenge to newcomers interested in the subject. These chapters should help readers understand how each component of the implementation fits together.
- Chapter 4: The procedure to vectorize the training algorithm which results in a significant reduction in training time.
- Chapter 5: A two-stage state-splitting method that is more efficient in regards to time and space complexity.
- Chapter 8: Code acceleration with Numba (Lam et al., 2015) which JIT-compiles Python code.
- Chapter 9: Caching for efficient automated hyperparameter tuning methods such as Bayesian optimization.

Chapter 2

Background

There are three basic components to spectral learning of latent-variable probabilist context-free grammars, i.e.

1. Probabilistic context-free grammars (PCFGs)
2. Latent-variable
3. Spectral learning

We will briefly summarise each of them in the following three sections.

2.1 Probabilistic Context-Free Grammars

A probabilistic context-free grammar (PCFG) is a 7-tuple $(\mathcal{N}, \mathcal{I}, \mathcal{P}, n, t, q, \pi)$ where:

- $\mathcal{N}$ is the set of nonterminals in the grammar. $\mathcal{I} \subset \mathcal{N}$ is the set of interminals and $\mathcal{P} \subset \mathcal{N}$ is the set of preterminals. Note that $\mathcal{I} \cup \mathcal{P} = \mathcal{N}$ and $\mathcal{I} \cap \mathcal{P} = \emptyset$.
- n is the set of terminals.
- For all $a \in \mathcal{I}$, $b \in \mathcal{N}$, $c \in \mathcal{N}$, we have a context-free rule $a \to b\ c$ and its associated probability $t(a \to b\ c \mid a)$.
- For all $a \in \mathcal{P}$, $x \in n$, we have a context-free rule $a \to x$ its associated probability $q(a \to x \mid a)$.
- For all $a \in \mathcal{I}$, we have a probability that a appears at the root of a tree $\pi(a)$.

This definition of a PCFG differs slightly (but is equivalent to) from the standard definition in many literature, but this definition serves as a stepping stone towards the definition of a latent-variable PCFG in Chapter 3.

A PCFG is trained by aggregating statistics in a given corpus of parse trees. $\mathcal{N}, \mathcal{I}, \mathcal{P}, n$ are found by simple inspection; t, q, π are calculated by maximum likelihood estimation (MLE), for example:

$$t(a \rightarrow b\ c \mid a) = \frac{|a \rightarrow b\ c|}{|a|} \tag{2.1}$$

where $|\cdot|$ denotes of the number of occurrence of $\cdot$ (i.e. $a \rightarrow b\ c$ and a). q, π are calculated in similar ways.

Now given a parse tree, its probability can be calculated as the product of probabilities of its rules. For a tree with n nodes, if we label each node in (any) order from 1 to n and each node's associated rule from r_1 to r_n, then its probability is:

$$p(\text{tree}) = \pi(a_{\text{root}}) \prod_{i: i \in \mathcal{I}} t(r_i \mid a_i) \prod_{i: i \in \mathcal{P}} q(r_i \mid a_i) \tag{2.2}$$

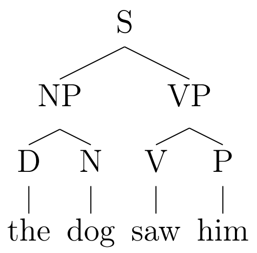


Figure 2.1: An example tree where $r_1 = S \rightarrow NP\ VP$, $r_2 = NP \rightarrow D\ N$, etc.

Take the tree in Figure 2.1 for a concrete example, its probability is:

$$\begin{aligned} p(\text{tree}) =& \pi(S) \times \\ & q(S \rightarrow NP\ VP \mid S) \times q(NP \rightarrow D\ N \mid NP) \times q(VP \rightarrow V\ P \mid VP) \times \\ & t(D \rightarrow \text{the} \mid D) \times t(N \rightarrow \text{dog} \mid N) \times t(V \rightarrow \text{saw} \mid V) \times t(P \rightarrow \text{him} \mid P) \end{aligned} \tag{2.3}$$

At test time, where we are only given a sentence and want to build a parse tree, we can enumerate all possible trees and return the *best* tree[1].

2.2 Latent-Variable Models

A latent variable is a variable that is unobserved but can be inferred from the variables that are observed. It might sound counter-intuitive to be able infer something that is unobserved, but let us illustrate it with an example.

In Figure 2.2, Bob is saying “How are you?" to Alice. Bob knows exactly the words he is trying to convey, but all Alice receives are some sound waves, where she needs to infer what words generated them. For Alice, the sound waves are observed, but the concrete words are latent. It should be clear that it indeed is possible for Alice

[1]The best tree *can* be the most likely tree, but we will use a different metric as discussed in Chapter 7.

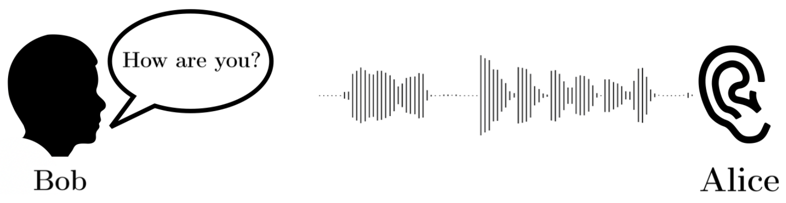


Figure 2.2: Bob asks Alice "How are you?"; Alice receives the sound waves and infers the words spoken.

to infer the words spoken. Thus this scenario can be modelled as a latent-variable model to enable machines to recognize speech.

Often, latent-variable models are trained using the expectation-maximization (EM) algorithm. Previously work by Matsuzaki et al. (2005) and Petrov et al. (2006) are both trained using the EM algorithm. The EM algorithm, though, has problems with local-optima and long training time. Spectral learning is another method to train latent-variable models with strong guarantees and is highly efficient.

2.3 Spectral Learning

The word *spectral* in spectral learning refers to the singular value decomposition (SVD). SVD can factor any matrix $\Omega \in \mathbb{R}^{m\times n}$ into 3 matrices U, Σ, V:

$$\Omega \overset{\text{SVD}}{=} U\Sigma V^T \tag{2.4}$$

where $U \in \mathbb{R}^{m\times m}$ (left singular vectors) and $V \in \mathbb{R}^{n\times n}$ (right singular vectors) are orthogonal matrices, and $\Sigma \in \mathbb{R}^{m\times n}$ (singular values[2]) is a rectangular-diagonal matrix.

For various reasons, a full SVD might not be desirable and a *truncated* SVD is applied where only the top-k largest singular values and their corresponding singular vectors are calculated:

$$\Omega \overset{\text{Truncated SVD}}{\approx} U_k\Sigma_k V_k^T \tag{2.5}$$

where $U \in \mathbb{R}^{m\times k}, \Sigma \in \mathbb{R}^{k\times k}, V \in \mathbb{R}^{n\times k}$. Usually k is chosen to be much smaller than Ω's rank, thus $U_k\Sigma_k V_k^T = \Omega_k$ is also called a low-rank approximation to Ω. See Figure 2.3 for an illustration.

A fact worth noting is that, Ω_k is the optimal rank-k approximation of Ω and there is no other rank-k matrix that is 'closer' to Ω in terms of Frobenius norm (Eckart and Young, 1936).

How does a truncated SVD help us learn the parameters of a latent-variable model? An intuition would be that: if our observables are of very high-dimension, the low-dimensional approximations of them can be seen as the latent representations that generated them. If we can put the observations into matrices, we can use

[2]More specifically, the values on the diagonal of Σ are called the singular values.

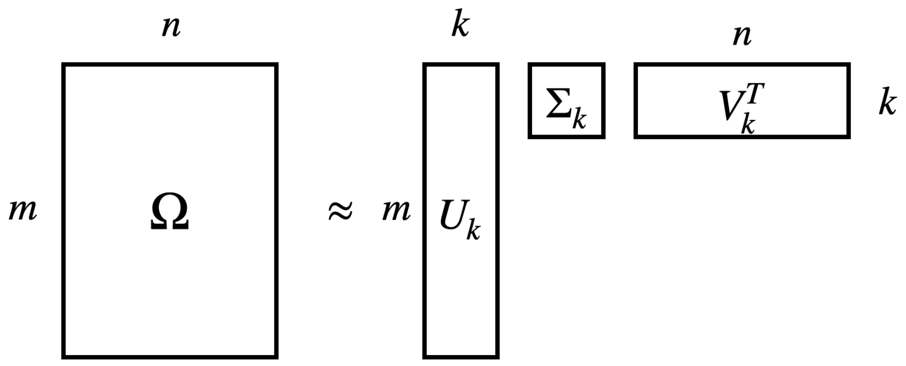


Figure 2.3: Rank-k approximation of matrix Ω using SVD.

the truncated SVD to get such low-dimensional representations. The mathematical proofs to exactly why this works are beyond the scope of this work, but readers are welcomed to consult Hsu et al. (2012) and Cohen et al. (2014) for proofs and statistical guarantees.

Chapter 3

Spectral Learning of Latent-Variable PCFGs

In this chapter, we first state the definition of a latent-variable PCFG from Cohen et al. (2014), then state the training algorithm in an engineering perspective.

3.1 Latent-Variable PCFGs

A latent-variable PCFG (L-PCFG) is a 8-tuple $(\mathcal{N}, \mathcal{I}, \mathcal{P}, m, n, t, q, \pi)$ where:

- $\mathcal{N}$ is the set of non-terminals in the grammar. $\mathcal{I} \subset \mathcal{N}$ is the set of in-terminals and $\mathcal{P} \subset \mathcal{N}$ is the set of pre-terminals. Note that $\mathcal{I} \cup \mathcal{P} = \mathcal{N}$ and $\mathcal{I} \cap \mathcal{P} = \emptyset$.
- m is the set of latent states.
- n is the set of terminals.
- For all $a \in \mathcal{I}$, $b \in \mathcal{N}$, $c \in \mathcal{N}$, $h_a, h_b, h_c \in m$, we have a context-free rule $a(h_a) \rightarrow b(h_b)\ c(h_c)$ and its associated likelihood $t(a \rightarrow b\ c, h_b, h_c \mid a, h_a)$.
- For all $a \in \mathcal{P}$, $x \in n$, $h \in m$, we have a context-free rule $a(h) \rightarrow x$ its associated likelihood $q(a \rightarrow x \mid a, h)$.
- For all $a \in \mathcal{I}$, $h \in m$, we have a probability that $a(h)$ appears at the root of a tree $\pi(a, h)$.

Note the similarity between this definition and that of a PCFG in section 2.1. The only difference is that every non-terminal is split into m (latent) states.

To calculate the probability of a given tree with n nodes, if we label each node in (any) order from 1 to n and each node's associated rule from r_1 to r_n, then its probability is:

$$p(\text{tree}) = \sum_{i=1}^{n} \pi(a_1, h_1) \prod_{i:i \in \mathcal{I}} t(r_i, h_{b(i)}, h_{c(i)} \mid a_i, h_i) \prod_{i:i \in \mathcal{P}} q(r_i \mid a_i, h_i) \tag{3.1}$$

where $b(i), c(i)$ return node-i's left child's index and right child's index, respectively. Again note the similarity between equation 3.1 and equation 2.2. The difference is that with the addition of latent variables, we need to marginalize them out for the probability of the tree.

At test time, we can still indeed enumerate all possible trees and return the best tree.

3.2 Spectral Learning Algorithm

Before we discuss the training algorithm, let us first put the desired parameters into matrix/vector forms as follows:

- For every binary context-free rule $r = a \to b\ c$, let $\mathbf{t} \in \mathbb{R}^{m \times m \times m}$ be a tensor where $[\mathbf{t}]_{i,j,k} = t(r, h_j, h_k \mid a, h_i)$.
- For every unary context-free rule $r = a \to x$, let $\mathbf{q} \in \mathbb{R}^m$ be a vector where $[\mathbf{q}]_i = t(r \mid a, h_i)$.
- For every $a \in \mathcal{I}$, let $\boldsymbol{\pi} \in \mathbb{R}^m$ be a vector where $[\boldsymbol{\pi}]_i = t(a, h_i)$.

Now we are ready to state the training algorithm, where each step is split into a subsection below.

3.2.1 Preprocessing

Penn Treebank Wall Street Journal (PTB) LDC99T42 (Marcus et al., 1993) dataset is the standard dataset used to train and evaluate syntactic parsers. PTB sections 02-21 is used for training, section 22 for developing, and section 23 for testing.

Is is also standard to clean up the data so that the trees:

- have no empty labels at the root.
- have no functional labels (e.g. `-LOC`, `-CLR`)
- have no `-NONE-`.
- have no `X` $\to$ `X` productions.

To fit our definition of L-PCFGs, we also need to binarize the parse trees.

Nonterminal Map	**Terminal Map**
JJ $\longleftrightarrow$ 0	earnings $\longleftrightarrow$ 696
NNS $\longleftrightarrow$ 1	Exchange $\longleftrightarrow$ 697
NP $\longleftrightarrow$ 2	trading $\longleftrightarrow$ 698
...	...

Figure 3.1: Mappings from nonterminals/terminals to integers. Operating with integers greatly improves performance.

For performance, it is also desirable to map all the non-terminals and terminals from strings to integers. We make one-to-one mappings like the ones shown in Figure 3.1.

3.2.2 Rule representation and PCFG Training

$\mathcal{N}, \mathcal{I}, \mathcal{P}, n$ can easily be found by inspection. Since non-terminals and terminals are represented using integers, we also create an one-to-one mapping from context-free rules to integers as follows:

- For each binary rule $a \to b\ c$, let its integer representation be:

```
(a << 40) ^ (b << 20) ^ c
```

- For each unary rule $a \to x$, let its integer representation be:

```
(a << 40) ^ (x << 20)
```

where << is the bit shift left operator, and ^ is the `XOR` operator. To recover $a \to b\ c$ (or $a \to x$), one can simply reverse these operations.

To complete training of the PCFG, we calculate likelihoods t, q, π using Equation 2.1.

3.2.3 Feature Extraction

For each node in a parse tree, we call the tree fragment the node spans **inside tree**, and the rest of the tree **outside tree**, see Figure 3.2.

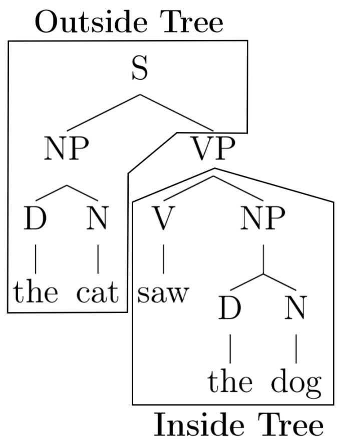


Figure 3.2: The inside and outside trees for the VP node.

We have two functions ϕ and ψ. When given a node in a tree, ϕ return features from the inside tree, and ψ returns features from the outside tree. For both ϕ and ψ, Cohen et al. (2013) selected features that consist of tree fragments, part-of-speech (POS) tags of head words, and span sizes. These features are all discrete and are one-hot encoded, and the returns from ϕ and ψ are (sparse) vectors. We detail the set of features used in this work in section 5.2.

It is worth noting that the features can be anything that is relevant and should be further explored as they affects the parsing accuracy significantly. We have experimented with many features, including different ways to encode inside/outside trees, word embeddings like GloVe (Pennington et al., 2014) and BERT (Devlin et al., 2019), span shapes (Hall et al., 2014), etc. Some yielded parsing accuracy worse than the PCFG baseline and we have yet to find a better set of features. We are confident that any feature which can improve the parsing accuracy must capture words' semantic meanings, as most of the incorrect parses have no problem with syntax.

3.2.4 Feature Scaling

The features are scaled after they are collected. Let inside feature-i of node-n be $[\phi(n)]_i$, we scale it to:

$$[\phi(n)]_i \times \sqrt{\frac{M}{\text{count}(i) + \kappa}} \tag{3.2}$$

where $\text{count}(i)$ is the number of appearance of feature-i, M is the number of examples in the training set, and κ is a small integer ($\kappa = 5$).

The same scaling is applied to outside features $[\psi(n)]_i$.

This simple scaling alone improves the F1 score by about 5. Cohen et al. (2013) offered some explanation to why this helps: the scaling is similar to **whitening** which standardize and decorrelate the data. Whitening is commonly used in data preprocessing to increase performance. Actually, we found that the value of M (100-100000) is surprisingly unimportant – any value from 100 to 100000 yields roughly the same results. A better understanding of the effect of scaling should contribute to higher accuracy and we shall explore this further.

In the following sections, we shall assume scaling has been applied when referring to the feature functions ϕ, ψ.

3.2.5 SVD and Projections

For each non-terminal a, calculate a (non-centered cross-covariance) matrix Ω^a:

$$\Omega^a = \frac{1}{|a|} \sum_{n:\text{label(n)}=a} \phi(n)\psi(n)^T \tag{3.3}$$

Note that ϕ, ψ can return high dimensional (10000+ dimensional) vectors with most of the entries being 0. Computationally, it is vital that these vectors, and subsequently the Ω matrices, are represented using sparse data structures in memory. We will discuss this in more details in section 4.2.

Perform a truncated SVD on each Ω^a:

$$\Omega^a \approx U^a \Sigma^a (V^a)^T \tag{3.4}$$

Let the inside projection matrix for a be $(U^a)^T$ and the outside projection matrix for a be $(\Sigma^a)^{-1}(V^a)^T$.

Then again for each node-n (which label a), project its feature vectors $\phi(n), \psi(n)$ to the latent representations using the projection matrices:

$$Y(n) = (U^a)^T \phi(n) \tag{3.5}$$
$$Z(n) = (\Sigma^a)^{-1}(V^a)^T \psi(n) \tag{3.6}$$

3.2.6 Constructing Parameters

Some technical details in this section are not entirely correct intentionally. In particular, the constructed parameters are not $\mathbf{t}, \mathbf{q}, \boldsymbol{\pi}$, but $\mathbf{t}, \mathbf{q}, \boldsymbol{\pi}$ up to a linear transformation. This information only adds an additional barrier to understanding this implementation, and everything will work as long as the parsing algorithm outlined in Chapter 7 is followed. Interested readers should consult Cohen et al. (2014) for more information.

For each rule $a \to b\ c$, construct a tensor $E^{3,a\to b\ c}_{i,j,k}$:

$$E^{3,a\to b\ c}_{i,j,k} = \frac{1}{|a \to b\ c|} \sum_{(n_a,n_b,n_c)} Z(n_a) \otimes Y(n_b) \otimes Y(n_c) \tag{3.7}$$

where (n_a, n_b, n_c) are tuples of nodes in which n_b, n_c are n_a's left and right child, $\otimes$ denotes the outer product. For readers who are familiar with `einsum`, $X \otimes Y \otimes Z$ corresponds to `einsum('i,j,k->ijk', X, Y, Z)`.

Then the (unsmoothed) parameter $\mathbf{t}$ for $a \to b\ c$ is:

$$\mathbf{t}(a \to b\ c) = \frac{|a \to b\ c|}{|a|} \times E^{3,a\to b\ c}_{i,j,k} \tag{3.8}$$

Similarly, parameters $\mathbf{q}$ for rules of the form $a \to x$ and the parameter $\boldsymbol{\pi}$:

$$\mathbf{q}(a \to x) = \frac{|a \to x|}{|a|} \times \frac{1}{|a \to x|} \sum_{(n_a,x)} Z(n_a) \tag{3.9}$$
$$\boldsymbol{\pi}(a) = \frac{|a|}{|\text{training set}|} \times \frac{1}{|a|} \sum_{\text{root } n_a} Y(n_a) \tag{3.10}$$

Figure 3.3, shows visually the vectors used to construct the parameters $\mathbf{t}, \mathbf{q}, \boldsymbol{\pi}$.

3.2.7 Smoothing

$E^{3,a\to b\ c}_{i,j,k}$ are 3rd-order moments, and to get good estimates of them, we need a large amount of data. In our training set, some rules do not appear very often at all and their estimated parameters are unreliable. Cohen et al. (2013) proposed to

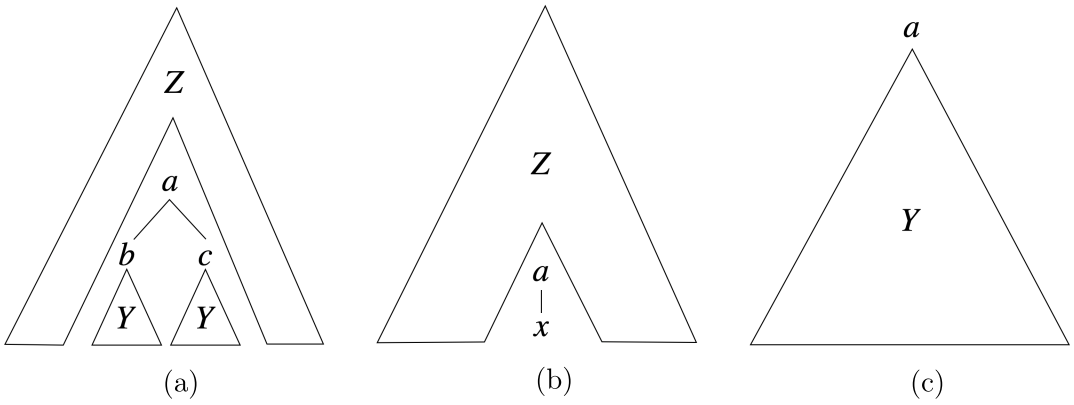


Figure 3.3: Projected vectors used to construct (a) $\boldsymbol{t}$ (b) $\boldsymbol{q}$ (c) $\boldsymbol{\pi}$

calculate lower-order estimates as well, and then interpolate estimates of different orders together.

Let:

$$E^{a\to b\ c}_{i,j,\cdot} = \frac{1}{|a\to b\ c|}\sum_{(n_a,n_b,n_c)} Z(n_a)\otimes Y(n_b) \qquad E^{a\to b\ c}_{\cdot,\cdot,k} = \frac{1}{|a\to b\ c|}\sum_{(n_a,n_b,n_c)} Y(n_c)$$

$$E^{a\to b\ c}_{i,\cdot,k} = \frac{1}{|a\to b\ c|}\sum_{(n_a,n_b,n_c)} Z(n_a)\otimes Y(n_c) \qquad E^{a\to b\ c}_{\cdot,j,\cdot} = \frac{1}{|a\to b\ c|}\sum_{(n_a,n_b,n_c)} Y(n_b)$$

$$E^{a\to b\ c}_{\cdot,j,k} = \frac{1}{|a\to b\ c|}\sum_{(n_a,n_b,n_c)} Y(n_b)\otimes Y(n_c) \qquad E^{a\to b\ c}_{i,\cdot,\cdot} = \frac{1}{|a\to b\ c|}\sum_{(n_a,n_b,n_c)} Z(n_a)$$

Then the 2nd-order moment would be:

$$E^{2,a\to b\ c}_{i,j,k} = \frac{1}{3}\left(E^{a\to b\ c}_{i,j,\cdot}\otimes E^{a\to b\ c}_{\cdot,\cdot,k} + E^{a\to b\ c}_{i,\cdot,k}\otimes E^{a\to b\ c}_{\cdot,j,\cdot} + E^{a\to b\ c}_{\cdot,j,k}\otimes E^{a\to b\ c}_{i,\cdot,\cdot}\right) \tag{3.11}$$

And the 1st-order moment would be:

$$E^{1,a\to b\ c}_{i,j,k} = E^{a\to b\ c}_{i,\cdot,\cdot}\otimes E^{a\to b\ c}_{\cdot,j,\cdot}\otimes E^{a\to b\ c}_{\cdot,\cdot,k} \tag{3.12}$$

Additionally we can calculate the 0th-order moment:

$$H^a_i = \frac{1}{|a|}\sum_{n_a} Z(n_a) \qquad F^a_i = \frac{1}{|a|}\sum_{n_a} Y(n_a)$$

$$E^{0,a\to b\ c}_{i,j,k} = H^a_i\otimes F^b_j\otimes F^c_k \tag{3.13}$$

Interpolate these estimates together and the smoothed parameters are:

$$E^{\text{smoothed},a\to b\ c}_{i,j,k} = \lambda E^{3,a\to b\ c}_{i,j,k} + \tag{3.14}$$

$$(1-\lambda)(\lambda E^{2,a\to b\ c}_{i,j,k} + \tag{3.15}$$

$$(1-\lambda)(\lambda E^{1,a\to b\ c}_{i,j,k} + (1-\lambda)E^{0,a\to b\ c}_{i,j,k})) \tag{3.16}$$

where $\lambda = \frac{\sqrt{|a \to b\ c|}}{C + \sqrt{|a \to b\ c|}}$ and C is chosen using the development set. This calculation of λ makes sure that when a rule appears frequently, it keeps its 3rd-order estimate; and conversely, when a rule appears infrequently, the lower-order estimates take on more weight.

Cohen et al. (2013) also smoothed unary rules, but we found that to have no effect on the parsing accuracy.

3.3 Training Pipeline

The whole spectral learning pipeline from section 3.2.1 to section 3.2.7 are visualized below in Figure 3.4.

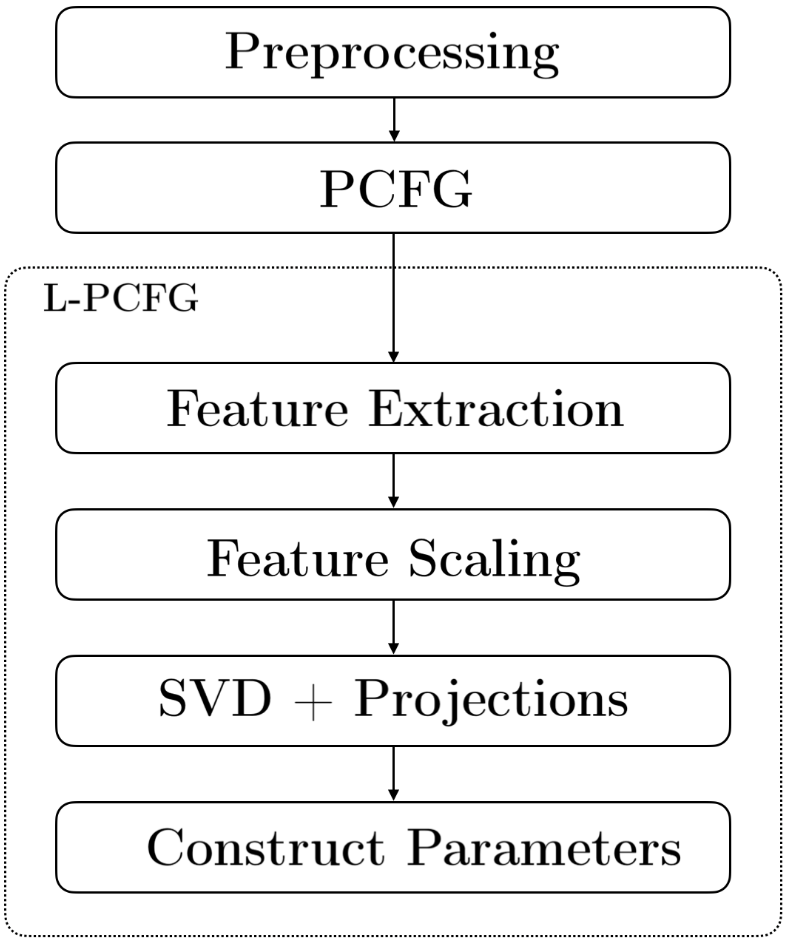


Figure 3.4: Every stage in the training pipeline for spectral learning of L-PCFG

The Python implementation also follows this pipeline rigorously, with some additional improvement made for performance.

Chapter 4

Vectorization

This chapter assumes the reader is familiar with Python, SciPy, and NumPy.

The training steps for L-PCFGs involve many matrix-vector operations. Chapter 3 states the steps in non-vectorized forms, vectorizing these steps can reduce the training time by 4-5 folds. The previous implementation, Rainbow Parser (Cohen and Narayan, 2017), was also vectorized, but ineffective because some care need to be taken when dealing with sparse data structures.

4.1 Node Ordering

A total ordering of tree nodes in our training set is the foundation behind vectorization.

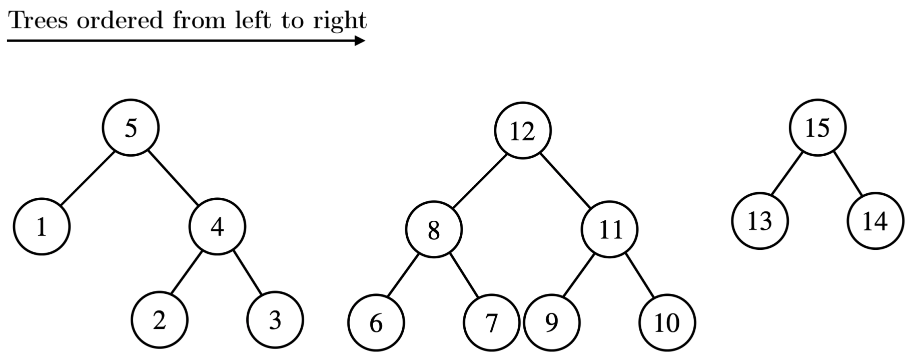


Figure 4.1: Ordering of nodes in a set of 3 example trees. Trees are ordered from left to right and nodes are ordered using post-order traversal.

Let the training set be $\mathcal{T}$ and let there be a total ordering on $\mathcal{T}$. In Python, this is the same as putting all the parse trees in a ordered data structure such as a list.

Then for each tree $t \in \mathcal{T}$, let the set of nodes in t be N_t and let there be a total ordering on N_t. In code, the ordering can be imposed using some ordered tree traversals (e.g. in/pre/post-order traversals).

Finally, for the set of all nodes N, where each node $n \in N$ is the j-th node of the i-th tree, total ordering is imposed by ordering the tuples (i, j). This is equivalent to how tuples are compared in most programming languages.

Figure 4.1 shows a possible ordering of nodes.

4.2 Augmented Feature Function

Remember that ϕ, ψ were feature functions that take in a node and return a vector. In this section, we propose augmented versions of them that return matrices.

Define the augmented feature functions as follows:

$$\Phi(a) = [\phi(n_{a1}), \phi(n_{a2}), \ldots, \phi(n_{ak})] \tag{4.1}$$
$$\Psi(a) = [\psi(n_{a1}), \psi(n_{a2}), \ldots, \psi(n_{ak})] \tag{4.2}$$

where $n_{a1}, n_{a2}, \ldots, n_{ak}$ are the nodes that have label a and that $n_{a1} \prec n_{a2} \prec \cdots \prec n_{ak}$.

Conceptually, it is very simple to construct these matrices; while practically, the approach can vary in speed considerably. Recall that the returned vectors from ϕ, ψ are sparse, and as the result, the returned matrices from Φ, Ψ are sparse as well. The previous implementation did the following (for Φ, similarly for Ψ):

1. Prepare an empty sparse matrix for $\Phi(a)$.
2. Retrieve each ϕ in dense representation.
3. Convert the dense vectors into sparse vectors.
4. Insert the sparse vectors one-by-one into the sparse matrix.

It is wasteful to convert dense vectors to sparse ones because one would have to 'look through' all those 0 entries (i.e. most of the entries). It is also wasteful to modify sparse matrices because they are often stored in a way that optimizes for future computation, and modification to them is (often) inefficient.

To fix these issues, we do the following in this work:

1. First retrieve each ϕ and store their indices and corresponding values only.
2. Make a dok (dictionary of keys) matrix (`scipy.sparse.dok_matrix`) - a sparse structure that allows fast modification.
3. Insert the indices and values into the dok matrix.
4. Convert the dok matrix into a csr (compressed sparse row) matrix (`scipy.sparse.csr_matrix`) - a sparse structure that allows fast computation.

We do not convert between dense and sparse data structures, and we only do modifications to a sparse matrix that is designed to be modified. It should be intuitively clear that this is a much faster way to construct these matrices than the previous implementation. We have also experimented with other less effective (but still much faster than before) ways to construct these matrices, but the details are omitted here.

4.3 Revised SVD and Projections

With Φ, Ψ, let us update our SVD and projections step outlined in section 3.2.5.

For each non-terminal $a \in \mathcal{N}$, calculate matrix Ω^a:

$$\Omega^a = \frac{1}{|a|}\Phi(a)\Psi(a)^T \tag{4.3}$$

Perform SVD as before $\Omega^a = U^a\Sigma^a(V^a)^T$, then the inside projection matrix is $(U^a)^T$ and the outside projection matrix is $(\Sigma^a)^{-1}(V^a)^T$.

Finally do projections:

$$\mathbf{Y}(a) = (U^a)^T\Phi(a) \tag{4.4}$$

$$\mathbf{Z}(a) = (\Sigma^a)^{-1}(V^a)^T\Psi(a) \tag{4.5}$$

$\mathbf{Y}(a)$ and $\mathbf{Z}(a)$ are the matrices that contain the original $Y(n)$s and $Z(n)$s. We can then proceed to construct the smoothed parameters as before, outlined in sections 3.2.6 and 3.2.7.

Chapter 5

Two-Stage Splitting

Before we discuss what *two-stage splitting* is and why it can be helpful, it is important we establish the time and space complexity lower bound of the spectral learning algorithm.

Remember that in a L-PCFG grammar, $\mathcal{N}$ is the set of nonterminals and m is the set of latent states, let their sizes be $|\mathcal{N}|$ and $|m|$ respectively. We state that the time and space lower bound are both[1]:

$$\Omega(|N||m|^3) \tag{5.1}$$

To establish this, we note that

- There are at least as many rules as there are non-terminals.
- Every binary rule $a \to b\ c$ has parameter $t(a \to b\ c, h_b, h_c \mid a, h_a)$, a tensor with $|m|^3$ elements. The parameter is an outer product of 3 $|m|$-dimensional vectors, which takes $\Omega(|m|^3)$ time to construct.

To construct all of the parameters for every rule, it takes $\Omega(|\mathcal{N}||m|^3)$ time. All of the parameters need to be saved, thus taking $\Omega(|\mathcal{N}||m|^3)$ space.

In the following sections, we discuss how to reduce the runtime and storage without losing parsing accuracy, by increasing $|\mathcal{N}|$ and decreasing $|m|$.

5.1 Markovization

As we have mentioned in Chapter 1, we can increase a PCFG's parsing accuracy by simply annotating each node with its parent's and siblings' labels. This process is called Markovization and it makes the grammar more fine-grained. But if too much Markovization is applied, the grammar can become too fragmented and result in no parse.

Markovization can be seen as explicitly splitting each-nonterminals into different states. We would hope that, if we apply Markovization to the trees, we would need

[1]Assuming no degenerate condition where every tree in the training set only has 1 node.

less latent states to achieve similar parsing accuracy. This was indeed true, but some care was required. It is very easy to oversplit because we are essentially splitting the grammar twice: first with Markovization and then with latent states.

In the literature, different orders of Markovization are denoted using a tuple. For example $(v = 1, h = 1)$ denotes that 1 vertical (parent) and 1 horizontal (siblings) Markovization. For a PCFG grammar, one can reach a maximum parsing score with a $(v = 2, h = 2)$ Markovization. But if we want to split the grammar further in L-PCFG, a $(v = 2, h = 2)$ Markovization will make the grammar too fragmented. As noted by Petrov (2012), even a $(v = 1, h = 1)$ Markovization would be too much. Klein and Manning (2003) achieved a good accuracy with a $(v < 2, h < 2)$ Markovized grammar, but their approach to construct such an "in-between" grammar is quite convoluted. Here, we propose a simple $(v < 1, h = 1)$ Markovization that worked well in this work.

We assume that the parse trees are preprocessed and binarized. Then there exist 2 types of nodes:

- Original nodes – nodes that exist before binarization
- New nodes – nodes that are induced from binarization

During binarization, we apply a $(v = 0, h = 1)$ Markovization (horizontal Markovization only affects new nodes). Once that is done, we apply a $(v = 1, h = 0)$ Markovization to the original nodes only. The resulting grammar would have a $(v < 1, h = 1)$ Markovization. Table 5.1 compares the F1 score across different Markovizations, but the conclusion is that the $(v < 1, h = 1)$ is the best.

5.2 Simpler Feature Functions

This section details the features the functions ϕ, ψ collect. An added benefit of Markovization is that we do not need to collect as many features in this stage, thus making the feature functions much simpler than that in Cohen et al. (2013).

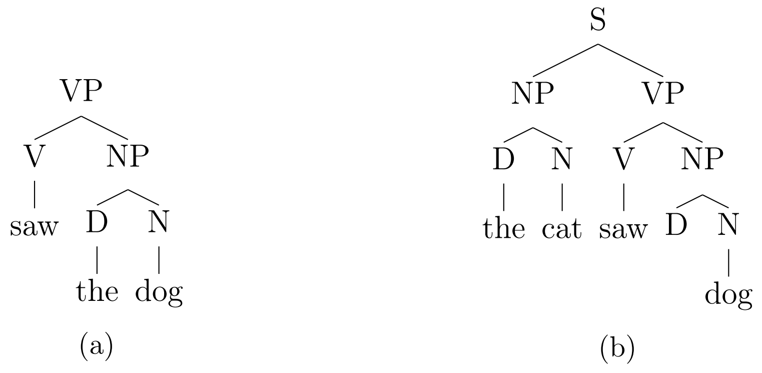


Figure 5.1: Example trees from Cohen et al. (2013)

The inside feature function ϕ collects 2 features:

1. Left child's rule + with the right child.
2. Right child's rule + with the left child.

If we collect the inside features of the inside tree in Figure 5.1(a), these 2 features would be:

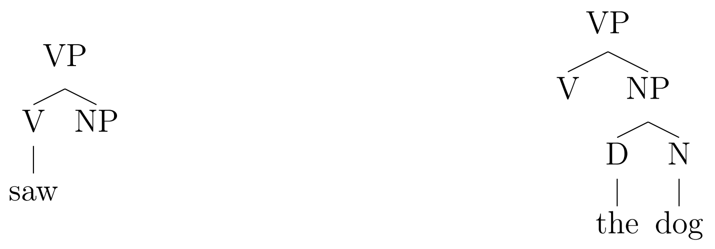


Figure 5.2

The outside feature function ψ collects 3 features:

1. Parent's rule
2. Grandparent's rule + with parent's rule
3. Grand-grandparent's rule + grand-parent's rule + parent's rule

If we collect the outside features of the outside tree in Figure 5.1(b), these 3 features would be:

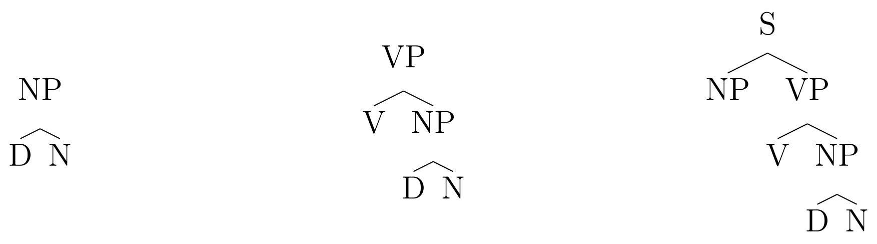


Figure 5.3

In Cohen et al. (2014), ϕ and ψ collected 7 features each. In this work, with only 2 inside and 3 outside features, we can get very similar parsing accuracy. Also note that extracting the features and putting them into a sparse data structure actually takes up a significant portion of the training time. By reducing the number of features extracted, we reduce the feature extraction time by almost 3 folds.

5.3 Latent Split and Complexity

Previously, Cohen et al. (2014) achieved a 88.05 F1 on the test set with $m = 32$ (i.e. 32 splits). In this work where we split twice (once with Markovization and once with latent annotations), we achieved 87.80 F1 on the test set with $|m_{\mathcal{I}}| = 13, |m_{\mathcal{P}}| = 16$

(i.e. 13 splits for in-terminal and 16 splits for pre-terminals). Markovization made $|\mathcal{N}|$ 4 times bigger, while the number of latent states $|m|$ needed is 2 times smaller. As we established, the complexity lower bound is $\Omega(|\mathcal{N}||m|^3)$. Thus we would expect about half the training time and storage needed than before, and this indeed was empirically true.

5.4 Effect of Markovization

In this section, we compare the parsing accuracy with grammars of different Markovization orders. Note that we collect the set of features introduced in section 5.2 and set the number of latent state splits to $|m_{\mathcal{I}}| = 13, |m_{\mathcal{P}}| = 16$. These settings are tuned for this work, and we shall emphasize that other settings will give very different numbers.

	$(v = 0, h = 0)$	$(v = 1, h < 1)$	$(v = 1, h = 1)$
PCFG	68.16	80.83	70.67
L-PCFG	84.30	88.10	86.30

Table 5.1: F1 scores on dev set when different Markovization orders are applied, with $|m_{\mathcal{I}}| = 13, |m_{\mathcal{P}}| = 16$.

Clearly, in this benchmark, $(v = 1, h < 1)$ Markovization is much better than the other two options. But the numbers also suggest that our grammar is still fragmented because of Markovization. Because the F1 score of the PCFG increased by >12 going from $(v = 0, h = 0)$ to $(v = 1, h < 1)$ Markovization; while the F1 score only increased by <4 for the L-PCFG. This is much lower than we have wished for and we shall try to find better ways to do Markovization in future work.

Chapter 6

Unknown Word Handling

Unknown work handling is an important and difficult problem we have not addressed. At test time, if we encounter a word that we have not seen in the training set, how would we deal with it?

Many researchers have tried different ways to address this. In particular, Cohen et al. (2013) did the following:

1. During preprocessing, replace words that appear infrequently with their part-of-speech (POS) tags.
2. Then train the grammar as normal.
3. At inference time, we use a POS tagger to tag the sentence.
4. Replace any unknown words with its POS tag (generated by the tagger).

This solution is very convenient because pre-trained POS taggers are widely available and it is also simple to train a custom POS tagger. But there are some potential problems with using POS taggers:

- If we use an externally trained POS tagger, the tagger might have been trained on a much larger vocabulary than the WSJ vocabulary and we gain an advantage when testing. This makes an unfair comparison between different parsers.
- The parser would not be very portable because the tagger must also be distributed for others to re-produce the result.

In this work, we implement a different solution to handling unknown words. It was originally written for the Berkeley Parser (Petrov, 2015). At training time, every infrequent word is mapped to one of more than 30 categories, for example:

- **19th-century** is mapped to **UNK-LC-NUM-DASH** - an unknown word that is in lower case that contains numeric characters and dashes.
- **Rolls-Royce** is mapped to **UNK-CAPS-DASH** - an unknown word that has capital letters and contains dashes.

- **oldest** is mapped to **UNK-LC-est** - an unknown word that is in lower case and ends in est.
- = is mapped to **UNK** - an unknown word with no special attributes.

At test time, the unknown words are mapped to these categories in the same way and we can then proceed to parse as normal.

The parsing accuracy did not increase (nor decrease) using this method for handling unknown words. But without the need for a POS tagger, this method is more portable. Also since many parsers have used this exact method for unknown word handling, it makes for a fairer comparison between this work and other related work.

Chapter 7

Evaluation Metric and Parsing Algorithm

This chapter assumes the reader is familiar with Python syntax.

So far, we have spent most of the time discussing the training algorithm for L-PCFG. In this chapter, we will discuss how we evaluate the parse trees our parser produces and more importantly, how the parse trees are produced by our parser in the first place.

7.1 Labelled Bracket Score

We now introduce the labelled bracket F1 score, the standard metric used for comparing syntactic parsers.

Each parse tree T consists of a set of nodes N. Each node has a label a and covers a **span** from the i-th word to the j-th word, represented using a 3-tuple (a, i, j). A parse tree T can then be characterized as a set of labelled spans:

$$T = \{(a, i, j) \mid \text{a node in } T \text{ has label } a \text{ that covers span } (i, j)\} \tag{7.1}$$

Given a sentence, let the parse tree a parser produces be T_c (c stands for candidate) and the gold parse tree be T_g. Then the labelled bracket recall, precision, and F1 score are as follows:

$$\text{Recall} = \frac{|T_c \cap T_g|}{|T_g|} \tag{7.2}$$

$$\text{Precision} = \frac{|T_c \cap T_g|}{|T_c|} \tag{7.3}$$

$$\text{F1} = 2 \cdot \frac{\text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}} \tag{7.4}$$

F1 scores are reported because both precision and recall can be artificially inflated. EVALB (Sekine and Collins, 1997) is the default software used to calculate the above metrics and is also used in this work (with the `new.prm` parameter file).

7.2 Labelled Recall Algorithm

Assuming that we have the marginal probabilities $\mu(a, i, j)$ (probability of non-terminal a covers the span (i, j)) for all non-terminals a and all possible spans (i, j) (we will state the algorithm to calculate the marginals in section 7.3).

Goodman (1996) introduced a bottom-up dynamic programming algorithm that chooses a parse tree that maximizes labelled bracket recall:

$$\arg\max_{T_c} \sum_{(a,i,j)\in T_c} \mu(a, i, j) \tag{7.5}$$

The pseudo-code of the algorithm is reproduced using Python syntax in Figure 7.1.

```python
for length in range(2, N + 1):  # N is the length of the yield
    for i in range(N - length + 1):
        j = i + length - 1
        for a in nonterminals:
            max_marginal = max(max_marginal, marginal(a, i, j))
        for k in range(i, j):
            best_split = max(best_split, maxc[i, k], maxc[k + 1, j])
        maxc[i, j] = max_marginal + best_split
```

Figure 7.1: Labelled recall algorithm

`maxc[1, N]` contains the maximum labelled bracket recall score and it is not hard to retrieve the parse tree that achieves this score by tracing back.

We found that it is insufficient to use the algorithm as stated, because the resulting parse is almost always syntactically incorrect. In this work, we extended the algorithm so that the every rule in the final parse tree must have been seen in the training set. Readers should consult the source code for details, but the basic idea is that for each split, we check whether the resulting rule has been seen.

7.3 Inside-Outside Algorithm for PCFG

In section 7.2, we assumed that we have obtained the marginal probabilities $\mu(a, i, j)$. Now we describe the algorithm to obtain the marginals **under a PCFG**[1] using the inside-outside algorithm. Michael Collins has a fantastic tutorial on this algorithm Collins, some of the materials here are adapted from his work.

[1]Instead of a L-PCFG, which will be in section 7.4.

First state the definition of the marginal:

$$\mu(a, i, j) = \sum_{T \in \mathcal{T} : (a,i,j) \in T} p(T)$$

where $\mathcal{T}$ is the set of all possible parse trees for the sentence and $p(T)$ is the probability of the tree T. In words, we are computing the sum probability of parse trees that contain nonterminal a that spans (i, j).

Next, we claim that $\mu(a, i, j)$ can be computed from two functions $\alpha(a, i, j)$ and $\beta(a, i, j)$:

$$\mu(a, i, j) = \alpha(a, i, j)\beta(a, i, j) \tag{7.6}$$

$\alpha(a, i, j)$ returns the sum probability of all the possible inside trees that contain nonterminal a that spans (i, j); and $\beta(a, i, j)$ returns the sum probability of all the possible outside trees that contain nonterminal a that spans (i, j). Figure 7.2 shows the inside and outside trees for (a, i, j).

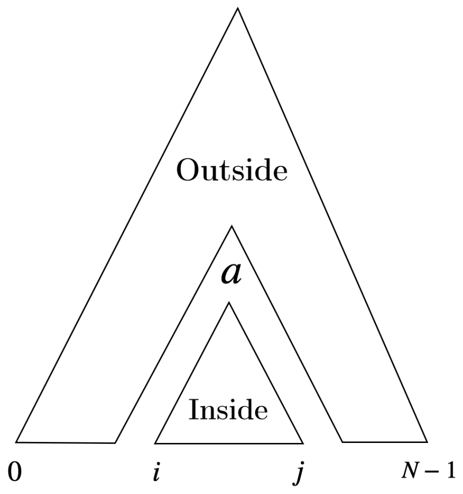


Figure 7.2: Inside tree of node a covering span (i, j) and outside tree of node a covering spans $(0, i - 1), (j + 1, N - 1)$.

It should be intuitive that, if the probability of all possible inside trees and the probability of all possible outside trees are known, their product would be the probability of all possible trees. Collins gave a proof in his tutorial, and we shall not repeat it here. In the next two sections, we state and visualize the dynamic programming algorithms to compute αs and βs.

7.3.1 Computing $\alpha(a, i, j)$

αs are computed using dynamic programming, identically to the Cocke–Younger–Kasami (CYK) (Kasami, 1965; Younger, 1967) algorithm. See Figure 7.3 for a code snippet (base case omitted).

```
for length in range(2, N + 1):
    for i in range(N - length + 1):
        j = i + length - 1
        for k in range(i, j):
            for $b$ in inside[i][k]:
                for $c$ in inside[k + 1][j]:
                    for $a \to b\ c$ in possible_rules($b$, $c$):
                        inside[i][j][$a$] += $p(a \to b\ c)$ * \
                                             inside[i][k][$b$] * \
                                             inside[k + 1][j][$c$]
```

Figure 7.3: Inside pass of the inside-outside algorithm

The value in entry inside[i][j][a] corresponds to $\alpha(a, i, j)$. Figure 7.4 is the mental image you should have for understanding the code.

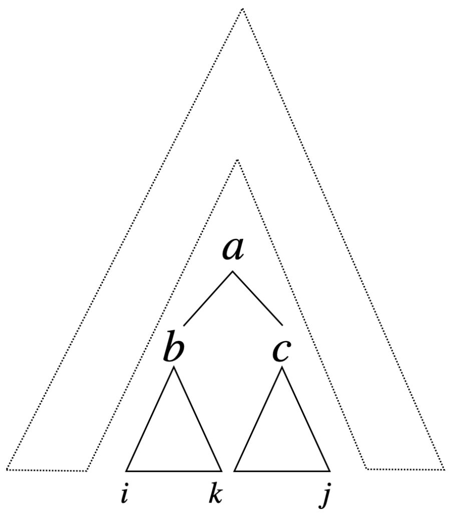


Figure 7.4: $\alpha(a, i, j)$ is calculated using $\alpha(b, i, k)$ and $\alpha(b, k + 1, j)$.

7.3.2 Computing $\beta(a, i, j)$

Once we have all the αs, we can use them when computing the βs. It is again dynamic programming, but top-down instead of bottom-up. See Figure 7.5 for a code snippet (base cases omitted).

```
1   for length in range(N - 1, 0, -1):
2           for i in range(N - length + 1):
3               j = i + length - 1
4               for k in range(j + 1, N):
5                   for a in outside[i][k]:
6                       for c in outside[j + 1][k]:
7                           for a → b c in possible_rules(a, c):
8                               outside[i][j][b] += p(a → b c) * \
9                                                         outside[i][k][a] * \
10                                                        inside[j+1][k][c]
11              for k in range(i):
12                  for a in outside[k][j]:
13                      for b in outside[k][i - 1]:
14                          for a → b c in possible_rules(a, b):
15                              outside[i][j][c] += p(a → b c) * \
16                                                        outside[k][j][a] * \
17                                                        inside[k][i - 1][b]
```

Figure 7.5: Outside pass of the inside-outside algorithm

The outside tree of any node are comprised of two parts - its parent's outside tree and its sibling's inside tree. The node can itself be a left or a right sibling, and we need to take both cases into account (one case on lines 4-10 and another on lines 11-17). Figure 7.6 is the mental image you should have for understanding the code (the current node can either be at the b position or the c position).

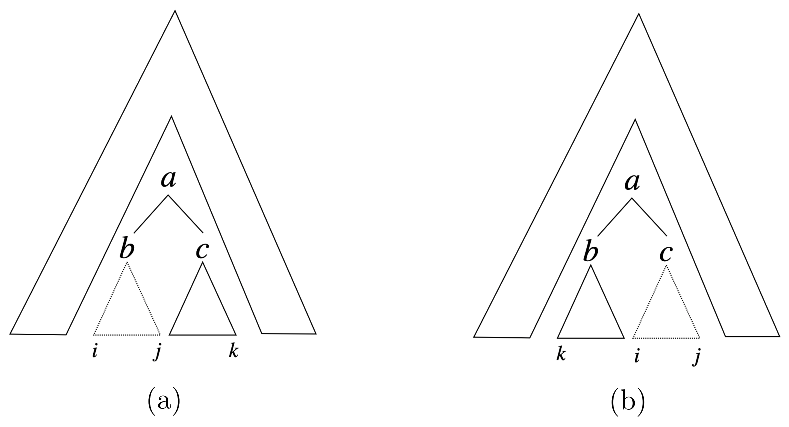


Figure 7.6: (a) Node is the left sibling, $\beta(b, i, j)$ is calculated using $\beta(a, i, k)$ and $\alpha(c, j + 1, k)$. (b) Node is the right sibling, $\beta(c, i, j)$ is calculated using $\beta(a, k, j)$ and $\alpha(b, k, i - 1)$.

7.4 Inside-Outside Algorithm for L-PCFG

Extending the inside-outside algorithm for PCFG to L-PCFG is simple. In a L-PCFG, parameters are tensors $\boldsymbol{t}$ (and vectors $\boldsymbol{q}, \boldsymbol{\pi}$). For convenience, we define 3 operators that take in a tensor and 2 vectors as arguments and return a vector:

$$\begin{aligned}
[O_{jk}(\boldsymbol{t}, \boldsymbol{q}^1, \boldsymbol{q}^2)]_i &= \sum_{j,k} \boldsymbol{t}_{i,j,k} \boldsymbol{q}^1_j \boldsymbol{q}^2_k \\
[O_{ik}(\boldsymbol{t}, \boldsymbol{q}^1, \boldsymbol{q}^2)]_j &= \sum_{i,k} \boldsymbol{t}_{i,j,k} \boldsymbol{q}^1_i \boldsymbol{q}^2_k \\
[O_{ij}(\boldsymbol{t}, \boldsymbol{q}^1, \boldsymbol{q}^2)]_k &= \sum_{i,j} \boldsymbol{t}_{i,j,k} \boldsymbol{q}^1_i \boldsymbol{q}^2_j
\end{aligned}$$

For readers who are familiar with `einsum`, these operators are equivalent to `einsum('ijk,j,k->i', t, q1, q2)`, `einsum('ijk,i,k->j', t, q1, q2)`, `einsum('ijk,i,j->k', t, q1, q2)`.

Substitute lines 8-10 in Figure 7.3 with:

```
inside[i][j][$a$] += $O_{jk}(\boldsymbol{t}(a \to b\ c)$,
                        inside[i][k][$b$],
                        inside[k + 1][j][$c$])
```

and substitute lines 8-10 in Figure 7.5 with:

```
outside[i][j][$b$] += $O_{ik}(\boldsymbol{t}(a \to b\ c)$,
                        outside[i][k][$a$],
                        inside[j+1][k][$c$])
```

and finally substitute lines 15-17 in Figure 7.5 with:

```
outside[i][j][$c$] += $O_{ij}(\boldsymbol{t}(a \to b\ c)$,
                        outside[k][j][$a$],
                        inside[k][i - 1][$b$])
```

Convince yourself that this new inside-outside algorithm in tensor form computes the right parameters. In practise, we first run the inside-outside algorithm for PCFG and prune the entries that have low probabilities and then run the algorithm for L-PCFG. Running the algorithm for L-PCFG directly would take significantly longer and would only achieve marginally better (or the same) parsing accuracy.

Once we have our $\boldsymbol{\alpha}$s and $\boldsymbol{\beta}$s, we compute the marginals using the dot product[2]:

$$\mu(a, i, j) = |\boldsymbol{\alpha}(a, i, j) \cdot \boldsymbol{\beta}(a, i, j)| \tag{7.7}$$

With the marginals, we can build the best parse using the labelled recall algorithm. In the next chapter, we discuss some implementation problems and solutions.

[2]We need to also take the absolute value of the dot product, as explained in Cohen et al. (2013).

Chapter 8

Code Acceleration with Numba

In Chapter 7 we introduced two important dynamic programming algorithms. Given a sentence of length N, the labelled recall algorithm runs in $O(N^3)$ time and the inside-outside algorithm runs in $O(|\mathcal{N}|N^3)$ time. These high complexities make high-performance code essential.

The problem was that we wished to implement everything in Python and native Python (CPython) for dynamic programming and numerical loops is known to be very slow because of its interpreted nature. Even with multi-processing, parsing the whole development set would take days.

We considered accelerating the code with custom C-extensions, Cython, PyPy, but finally settled on using Numba (Lam et al., 2015). Taken from Numba's website[1], "Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code". To put it simply, we can annotate our existing code and it would be compiled and executed (instead of interpreted) at runtime. There were some complications because only a subset of Python is supported and compromises were made to fit everything (data structures, matrix operations, multi-processing) into that subset. The details are hard to demonstrate in a written format, readers should consult the code if interested.

As part of this dissertation, I gave a talk on Numba to Dr Shay Cohen's group at Edinburgh. The slides can be found here: `https://github.com/GavinPHR/Numba-Talk`.

[1]Numba's official website: `https://numba.pydata.org/`

Chapter 9

Efficient Hyperparameter Tuning

Many hyperparameters need to be tuned in the training algorithm, notably:

- $m_{\mathcal{I}}$ – the number of latent states for in-terminals
- $m_{\mathcal{P}}$ – the number of latent states for pre-terminals
- C for calculating λ in parameter smoothing

We tune them by repeated running the training algorithm and evaluating on the development set, and finally picking the set of hyperparameters that achieve the highest F1 score.

With this high-performance implementation, it takes only 30 minutes[1] to run the training and evaluation code end-to-end. It is indeed possible for humans to manually tune the hyperparameters and achieve an acceptable parsing accuracy. But 30 minutes for 1 iteration is still prohibitively long for an automated routine, such as Bayesian optimization in section 9.2.

9.1 Caching

In an attempt to overcome the prohibitively long training-evaluation iteration, we noticed that in every such iteration, much of the same work is done. If we can cache everything that is unchanged from iteration to iteration, we can save a lot of time.

For the training pipeline, we can actually delay the application of all the hyperparameters to the final stage where we construct the parameters. Let $m_{\max}$ be the maximum number of states we would allow. During the truncated SVD stage, we truncate the SVD to rank $m_{\max}$, the projections are then also of size $m_{\max}$. Finally, the constructed parameters $\boldsymbol{t}$ have size $m_{\max} \times m_{\max} \times m_{\max}$, $\boldsymbol{q}, \boldsymbol{\pi}$ have size $m_{\max}$. Only now, we truncate the parameters further into the right dimensions (i.e. with $m_{\mathcal{I}}, m_{\mathcal{P}}$). Figure 9.1 illustrates how this truncation is done.

If we are applying the hyperparameters at the final stage, we can indeed just cache everything the final stage. Then in every training-evaluation iteration, we can run

[1]Training on 1 4-core CPU and evaluation on 4 8-core CPUs.

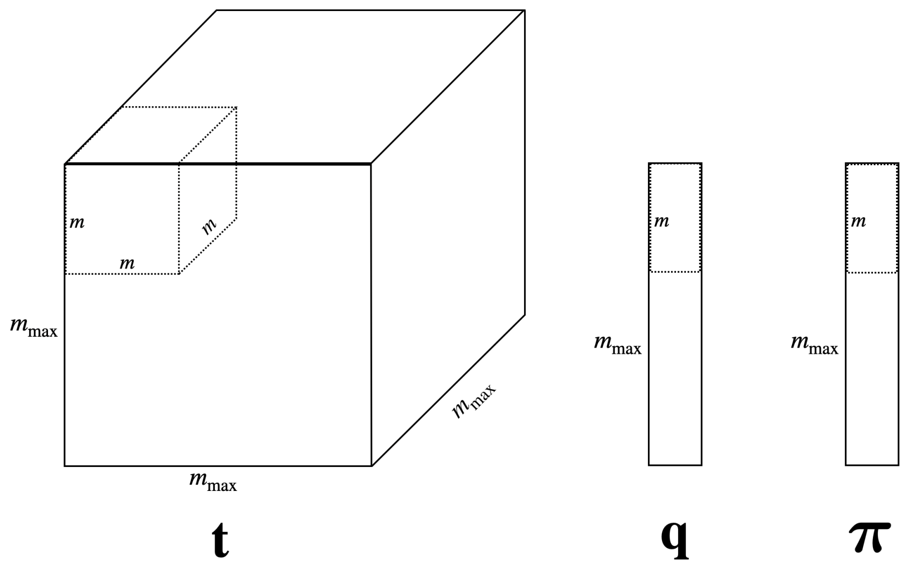


Figure 9.1: All the maximum parameters (solid lines) are cached and they are truncated (dotted lines) depending on the hyperparameters.

the final stage only with different hyperparameters and save 95% of the training time to about 30 seconds.

For the evaluation pipeline, we noticed that 90% of the time was spent pruning using the PCFG. Since the PCFG remains unchanged (because the hyperparameters only affect the L-PCFG), we can cache all the prune charts and reduce the parsing time by 90% to about 2 minutes.

With caching, 1 iteration of training-evaluation takes less than 3 minutes. It is now possible to use automated routines to tune the hyperparameters and we shall introduce one such routine in the next section.

9.2 Bayesian Optimization

Bayesian optimization is a method to optimize any black-box function. The details of Bayesian optimization is not the focus of this work, but it optimizes a function by exploring and exploiting the function's different regions. It is especially advantageous for optimizing functions that are not easily differentiable (like the one we have).

A common requirement for Bayesian optimization is that the parameters are continuous. Some parameters (i.e. the number of latent states) we are tuning are clearly not continuous. There are ways to make Bayesian optimization work with discrete parameters, but that is well beyond the scope of this work. We can 'hack' this limitation by inputting continuous parameters and cast them into integers 'inside the black-box'.

We used the `bayes_opt` module (Nogueira, 2014) to do Bayesian optimization and was successful at raising the parsing accuracy.

Chapter 10

Results and Evaluation

We have thus far described the training and evaluation pipelines and methods to improve their performance. In this chapter, we show some quantitative results.

10.1 Training Speed and Parameter Size

In Chapter 4, we described vectorization that makes efficient matrix-vector computations. In Chapter 5, we proposed a two-stage splitting procedure that is faster and takes less space, in terms of computational complexities.

Benchmarked on a laptop with an Intel i7-7700HQ (4-core CPU) and 16GB of RAM.

In this implementation, the whole training pipeline takes 5 minutes and the final parameters take 270MB disk space. The previous implementation, Rainbow Parser (Cohen and Narayan, 2017), takes 5 hours 20 minutes to train and the final parameters take 12.5GB disk space (in addition, 12GB disk space is used to store intermediate parameters). We achieved a **64x** improvement in speed and a **46x** improvement in space.

These results are much better than we anticipated. They prove that the spectral learning algorithm can be highly efficient and has huge potential. For completeness, Figure 10.1 shows a breakdown of training time.

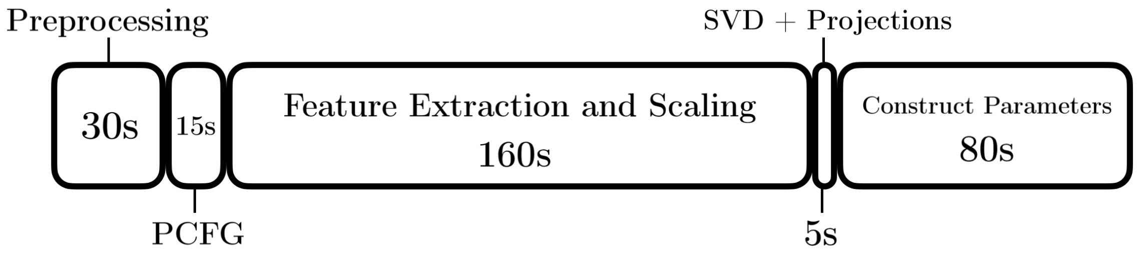


Figure 10.1: Breakdown of training time

10.2 Parsing Speed and Accuracy

Using 4 Intel Xeon E5-2630 (8-core CPU), parsing the development set (1700 sentences) takes 20 minutes and the test set (2416 sentences) takes 32 minutes.

Table 10.1 shows the full results on the development and test sets.

	Dev Set	Test Set
Precision	88.20	88.06
Recall	88.01	87.56
F1	88.10	87.81
EX	32.92	32.70
POS	97.09	97.25

Table 10.1: Results from development and test sets, as generated by EVALB.
EX = exact match POS = POS tagging accuracy

Table 10.2 compares this work to other related work and state-of-the-arts.

	Dev Set F1	Test Set F1
(Matsuzaki et al., 2005)	–	86.10
(Petrov and Klein, 2007)	91.20	90.10
(Cohen et al., 2013)	88.82	88.05
(Zhao et al., 2018)	91.24	91.02
This work	88.10	87.81
(Socher et al., 2013)	91.20	90.40
(Dyer et al., 2016)	–	92.40
(Vaswani et al., 2017)	–	91.30
(Kitaev and Klein, 2018)	95.21	95.13
(Mrini et al., 2020)	–	96.38

Table 10.2: Comparison between this work and other related work including some state-of-the-arts

It can be seen that this work achieves an F1 score similar to that in Cohen et al. (2013). Parsers in the upper section of Table 10.2 are all latent-variable models and Zhao et al. (2018) achieved a test F1 of 91.02. Much work still needs to be done for a spectral parser to match that F1 score. Parsers in the bottom section of Table 10.2 are all neural models. Socher et al. (2013) and Dyer et al. (2016) are based on recurrent neural networks and achieved slightly higher F1 than latent-variable models. Since the famous *Attention Is All You Need* paper (Vaswani et al., 2017), many state-of-the-art parsers use the attention mechanism, Berkeley Neural Parser (Kitaev and Klein, 2018) is one of the most cited such parsers. These parsers' accuracies greatly exceed this work and achieves human-level performance. In future work, we should keep exploring the possibilities of incorporating deep learning and attention into a spectral parser.

Chapter 11

Conclusion and Future Work

In this work, we re-implemented the spectral learning algorithm for L-PCFG. We used vectorization (Chapter 4) and two-stage splitting (Chapter 5) to drastically reduce the training time; we used Numba to JIT-compile and accelerate the high-complexity parsing algorithms; and we described a way to tune hyperparameters efficiently with caching. All of these are implemented efficiently in a single programming language. Additionally, the codebase is user-friendly, compact, and maintainable.

In this written work we have spelled out all the unspoken practises in the field of syntactic parsing – the common dataset used, how the dataset is processed, the evaluation metric and evaluation software etc. We have also introduced the spectral learning algorithm from an engineering perspective, and we believe that newcomers to this field would find this work easier to grasp than the other very mathematical papers.

This work can be continued in many ways, including but not limited to:

- adapting it to work with languages other than English
- making use of word embeddings to disambiguate, especially for PP, SBAR, and CC constituents.
- finding better ways to do Markovization and feature extraction to improve accuracy

There can be even more out-of-the-box extensions like making training-evaluation differentiable end-to-end and training with gradient-descent, but we shall leave this to the readers' imagination.

Overall, this work proves that spectral parser is highly efficient, helps newcomers understand how it works and how it is implemented, and serves as a stepping-stone towards a spectral parser that is also highly accurate.

References

Eugene Charniak. 1997. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 598–603. AAAI Press.

Shay B. Cohen and Shashi Narayan. 2017. The rainbow parser.

Shay B. Cohen, Karl Stratos, Michael Collins, Dean P. Foster, and Lyle Ungar. 2013. Experiments with spectral learning of latent-variable PCFGs. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 148–157, Atlanta, Georgia. Association for Computational Linguistics.

Shay B. Cohen, Karl Stratos, Michael Collins, Dean P. Foster, and Lyle Ungar. 2014. Spectral learning of latent-variable pcfgs: Algorithms and sample complexity. *Journal of Machine Learning Research*, 15(69):2399–2449.

Michael Collins. The inside-outside algorithm.

Michael Collins. 2003. Head-driven statistical models for natural language parsing. *Comput. Linguist.*, 29(4):589–637.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209, San Diego, California. Association for Computational Linguistics.

Carl Eckart and Gale Young. 1936. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218.

Joshua Goodman. 1996. Parsing algorithms and metrics. In *34th Annual Meet-*

ing of the Association for Computational Linguistics, pages 177–183, Santa Cruz, California, USA. Association for Computational Linguistics.

David Hall, Greg Durrett, and Dan Klein. 2014. Less grammar, more features. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 228–237, Baltimore, Maryland. Association for Computational Linguistics.

Daniel Hsu, Sham M. Kakade, and Tong Zhang. 2012. A spectral algorithm for learning hidden markov models. *Journal of Computer and System Sciences*, 78(5):1460 – 1480. JCSS Special Issue: Cloud Computing 2011.

Mark Johnson. 1998. PCFG models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632.

T. Kasami. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages.

Nikita Kitaev and Dan Klein. 2018. Constituency parsing with a self-attentive encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2676–2686, Melbourne, Australia. Association for Computational Linguistics.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan. Association for Computational Linguistics.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA. Association for Computing Machinery.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic CFG with latent annotations. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 75–82, Ann Arbor, Michigan. Association for Computational Linguistics.

Khalil Mrini, Franck Dernoncourt, Quan Hung Tran, Trung Bui, Walter Chang, and Ndapa Nakashole. 2020. Rethinking self-attention: Towards interpretability in neural parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 731–742, Online. Association for Computational Linguistics.

Fernando Nogueira. 2014. Bayesian Optimization: Open source constrained global optimization tool for Python.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical*

Methods in Natural Language Processing (EMNLP), pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Slav Petrov. 2012. *Coarse-to-Fine Natural Language Processing*.

Slav Petrov. 2015. Berkeley parser.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, Sydney, Australia. Association for Computational Linguistics.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 404–411, Rochester, New York. Association for Computational Linguistics.

Satoshi Sekine and Michael Collins. 1997. Evalb.

Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. 2013. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 455–465, Sofia, Bulgaria. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189–208.

Yanpeng Zhao, Liwen Zhang, and Kewei Tu. 2018. Gaussian mixture latent vector grammars. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1181–1189, Melbourne, Australia. Association for Computational Linguistics.