Sentence Compression as Dependency Parsing

First Author
Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain

Second Author
Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain

Abstract

1 Introduction

2 Related Work

(?)

(?)

3 Background

We begin by presenting formal background for sentence compression. Given an input sentence as a sequence $s_1 \dots s_n$, a sentence compression is an ordered sub-sequence of tokens, i.e. a sequence $s_{p_1} \dots s_{p_m}$ for some $1 \leq p_1 < \dots < p_m \leq n$.

Instead of directly predicting the sub-sequence p, we model compression as a joint parsing and language modeling problem. We first give notation for these models and then describe their use for compression.

To incorporate a simple language model over sub-sequences, we define the index set $\mathcal{I}$ of possible compression bigrams

$$\mathcal{I} = \big\{(i,j) : 0 \leq i < j \leq n+1\big\}$$

The element (i,j) is used to indicate that $p_k = i$ and $p_{k+1} = j$ for some index $0 \leq k \leq m$ where for boundary cases define terms p_0 and p_{m+1} as the start/end symbols `<s>` and `</s>` respectively.

The set of valid compression bigram sequences $\mathcal{Y}$ is defined as

$$\mathcal{Y} = \left\{ \begin{array}{l} y \in \{0,1\}^{\mathcal{I}} : \displaystyle\sum_{i=1}^{n+1} y_{0,i} = 1,\ \sum_{i=0}^{n} y_{i,n+1} = 1, \\ \displaystyle\sum_{i=0}^{j-1} y_{ij} = \sum_{k=j+1}^{n+1} y_{jk}\ \forall\, 1 \leq j \leq n \end{array} \right.$$

And the score of each bigram is given by a score vector $\theta \in \mathbb{R}^{\mathcal{I}}$.

Next, to incorporate syntactic structure, define the index set of dependency arcs over the original sentence as

$$\mathcal{J} = \big\{(i,j) : \forall\, i \in \{0 \dots n\}, j \in \{1 \dots n\}\big\}$$

where the arc (i,j) will indicate that s_i is the head word of s_j, and $i = 0$ is a special *root* position $*$.

The set of *compressed* dependency parses is $\mathcal{Z}'$ where

$$\mathcal{Z}' = \{z \in \{0,1\}^{\mathcal{J}} : z \text{ is a directed tree rooted at } *\}$$

The score of a dependency arc is given by a scoring vector $\omega \in \mathbb{R}^{\mathcal{J}}$.

Note that standard dependency parsing requires z to be a directed *spanning* tree, i.e. $||z||_1 = n$. For compression we allow $0 \leq ||z||_1 \leq n$. Compression rate is discussed in Section 5.

Finally define the score of a sentence compression is the sum of the score of the compressed dependency parse and the corresponding language model score. The decoding problem is to find

$$\begin{array}{ll} \displaystyle\arg\max_{y \in \mathcal{Y}, z \in \mathcal{Z}'} & \theta^\top y + \omega^\top z \\ \text{s.t.} & \displaystyle\sum_{i=0}^{j-1} y_{ij} = \sum_{i=0}^{n} z_{ij}\ \ \forall\, 1 \leq j \leq n \end{array}$$

The compressed sub-sequence p can be directly read from y.

Unfortunately, solving this maximization problem has been shown to be NP-hard (). This is mainly because of the difficulty of constrained maximum directed spanning tree problems.

In this paper we focus on a variant form of dependency parsing known as *projective* parsing. Informally, a projective parse requires that for any $z_{i,j} = 1$ and $z_{i'j'} = 1$ the arcs do not cross each other. Define the set $\mathcal{Z}$ as all compressive projective parses. The projective assumption allows us to use dynamic programming for decoding and to vastly improve worst-case run-time.

4 Decoding

We now present the decoding algorithm for sentence compression. We begin by reviewing the standard dynamic programming algorithm for projective dependency parsing, and then give an extension for sentence compression.

4.1 Projective Parsing

The standard decoding algorithm for projective dependency parsing is known as Eisner's algorithm (**?**; **?**). The algorithm is specified through a set of deductive rules acting on items. Each item consists of a tuple (t, i, j) where i, j is a span $0 \leq i \leq j \leq n$ and t is symbol from the set $\{\text{◿}, \text{◺}, \text{⏥}, \text{⏢}, \square\}$ (informally referred to as triangles, trapezoids, and boxes). These items indicate partial structures that may be combined with logical rules. For instance, the rules

$$\frac{(\text{◺}, i, k) \quad (\text{◿}, k+1, j)}{(\text{⏢}, i, j)} \; S_z(i,j) \qquad \forall\, i \leq k < j$$

combine right- and left-facing triangles covering i, k and $k+1, j$ respectively to produce a right-facing trapezoids covering i, j. The full set of $O(n^3)$ rules is given in Figure 2.

A deductive rule may also have an associated *consequence* notated on its right, for instance the consequence of the above rule is $S_z(i,j)$. The consequence is a member of a semiring, specified by a tuple $(\mathbb{S}, \oplus, \otimes, 0, 1)$ where $\mathbb{S}$ is a set of possible values, $\oplus$ and $\otimes$ are binary operators, and 0 and 1 are the annihilator and identity respectively. Define $S_z : \mathcal{J} \mapsto \mathbb{S}$ to map from dependency arcs to semiring values.

We can combine items and semiring values using dynamic programming. To find the score of the highest-scoring parse, we can use the max-semiring $(\mathbb{R}, \max, +, -\infty, 0)$ and let $S_z(i,j) = \omega_{ij}$. This specification can be used to construct an $O(n^3)$ CKY-style dynamic programming algorithm for dependency parsing. For other semirings, we can use this algorithm to find the semiring value of the goal item in $O((on)^3)$ time, where o is the max cost of the $\otimes$ and $\oplus$ operator.

4.2 Compressive Parsing

Next we turn to projective parsing that allows non-spanning trees and incorporates a language model, we refer to this technique as compressive parsing.

First, define a new function $S_y : \mathcal{I} \mapsto \mathbb{S}$ to map word bigrams to semiring values. For the max-semiring this function is $S_y(i,j) = \theta_{ij}$. For no-deletion parsing, we can incorporate this new value directly into the modification rules, e.g.

$$\frac{(\text{◺}, i, k) \quad (\text{◿}, k+1, j)}{(\text{⏢}, i, j)} \; S_z(i,j) \otimes S_y(k, k+1)$$

For compressive parsing, however, we also want to merge items that may have gaps, i.e. in-between words that are not descendants of either item. This requires introducing rules that skip intermediate words. These rules can simply ignore the intermediate words i.e. for all $0 \leq i \leq k < l < j \leq n$

$$\frac{(\text{◺}, i, k) \quad (\text{◿}, l, j)}{(\text{⏢}, i, j)} \; S_z(i,j) \otimes S_y(k, l)$$

These rules produce an (i,j) arc, leave out words $j+1$ through $l-1$, and produce a (k,l) bigram.

Unfortunately this modification results in an additional free variable l and gives an $O(n^4)$ parsing algorithm. Observe though that it is not necessary to produce the arc and bigram with the same rule. In fact, we can force k to predict if there will be gap from $k+1$ to $l-1$ before applying other rules, and then apply the standard Eisner's rules, as if $k+1$ through $l-1$ were descendants of k (and thus i as well).

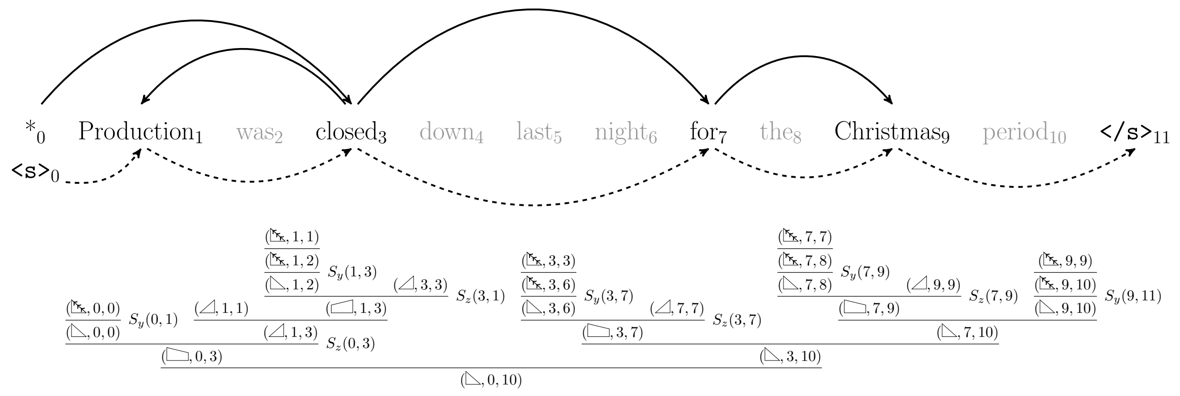
This optimization is known as the "hook trick" (). We implement this trick by replacing initial items $(\text{◺}, i, i)$ with new items $(\text{⧄}, i, i)$. These special items are only allowed to skip words to their right before becoming standard ◺ items

$$\frac{(\text{⧄}, i, i)}{(\text{⧄}, i, j)} \qquad \forall\, 0 \leq i < j \leq n$$

$$\frac{(\text{⧄}, i, j)}{(\text{◺}, i, j)} \; S_y(i, j+1) \qquad \forall\, 0 \leq i \leq j \leq n$$

After skipping words $i+1$ to j we incorporate the bigram consequence $S_y(i, j+1)$.

These hook rules allow us to avoid the extra free variable and maintain the same efficiency as Eisner's algorithm. A full parse derivation is show in Figure **??**.



$*_0$ Production$_1$ was$_2$ closed$_3$ down$_4$ last$_5$ night$_6$ for$_7$ the$_8$ Christmas$_9$ period$_{10}$ </s>$_{11}$
<s>$_0$
$S_y(0,1)$ $S_y(1,3)$ $S_z(3,1)$ $S_z(0,3)$ $S_y(3,7)$ $S_z(3,7)$ $S_y(7,9)$ $S_z(7,9)$ $S_y(9,11)$

Figure 1: Example of a compressive parse with derivation. The compressed sentence is "Production closed for Christmas". The output structure has arc and bigrams $y_{0,3}, y_{3,1}, y_{3,7}, y_{7,9}, z_{0,1}, z_{1,3}, z_{3,7}, z_{7,9}$, and $z_{9,11}$ set to 1 and all others set to 0. The full first-order derivation tree is shown below. The final derivation has $||z||_1 = 4$.

4.3 Second-Order Compressive Parsing

A similar algorithm can be used for second-order dependency parsing (). For a second-order parser we replace the index set $\mathcal{I}$ with a new index set that also includes a *sibling* index j (the previous modifier on the same side as modifier k).

$$\mathcal{J} = \begin{cases} (i, j, k) : i \in \{0 \dots n\}, j \in \{1 \dots n\}, \\ \quad (i < k < j) \vee (i > k > j) \vee k = \epsilon \end{cases}$$

where ϵ indicates that k is the first modifier on its side.

The deductive rules for second-order parsing are shown in Figure 2. They result in an $O(n^3)$ parsing algorithm.

The main complication for second-order compressive parsing is the rule

$$\frac{(\llcorner, i, i) \quad (\angle, i+1, j)}{(\sqsubset, i, j)} \; S_z(i, \epsilon, j) \quad \forall i < j$$

which creates the initial right-modifier of i, (i, ϵ, j). This rule implicitly assumes that all items $(\llcorner, i, j)$ with $i \neq j$ have already taken at least one modifier. However, with the new rules, it is possible that $(\llcorner, i, j)$ may have skipped $i+1$ through j. Therefore it is necessary to also allow these items to take a first right-modifier as well, i.e.

$$\frac{(\llcorner\!\!\times, i, k) \quad (\angle, k+1, j)}{(\sqsubset, i, j)} \; S_z(i, \epsilon, j) \quad \forall i \leq k < j$$

The remaining second-order rules are identical parsing, and second-order compressive CKY can be run in $O(n^3)$ time. The full set of rules is show in Figure 2.

5 Compression Limits

Sentence compression problems often have a target range for the compressed sentence. That is we might have upper and lower bounds for the size m of the final sentence, i.e.

$$\begin{aligned} q(m, \theta, \omega) = \quad & \max_{y \in \mathcal{Y}, z \in \mathcal{Z}} \theta^\top y + \omega^\top z \\ \text{s.t} \quad & \sum_{i=0}^{j-1} y_{ij} = \sum_{i=0}^{n} z_{ij} \;\; \forall\, 1 \leq j \leq n, \\ & ||z||_1 = m \end{aligned}$$

In this section we discuss two methods for enforcing these limits, first by dynamic programming and the second by bisection.

5.1 Resource-Constrained Parsing

Resource-constrained problems require finding an optimal solution under a resource limit or requirement. Without loss of generality we will assume that a resource begins at 0 and is upper-bounded by d. We can modify a dynamic program to keep track of the amount of resources consumed.

To track resources, define the resource-constrained max-semiring as

$$(\mathbb{R}^{\{0 \dots d\}}, \max_{rc}, +_{rc}, -\infty, \mathbf{1})$$

where for an item with value $r \in \mathbb{R}^d$, r_i is the highest-score seen using i resources, and we define

$$\begin{aligned} \max_{rc}\{r, s\} &= \sum_{i=0}^{d} \delta_i \max\{r_i, s_i\} \\ r +_{rc} s &= \sum_{i=0}^{d} \delta_i \max_{j,k: j+k=i} r_j + s_k \end{aligned}$$

For this semiring $\otimes$ has runtime $O(d^2)$. The runtime for CKY-style projective parsing is therefore $O(n^3 d^2)$.

For sentence compression, one resource of interest would be the number of words kept. The upper bound for this resource is $d = m$, and the semirings at each item would be

$$S_y(i, j) = \delta_0 \theta_{ij}, \;\; S_z(i, j) = \delta_1 \omega_{ij}$$

If the goal item has value is r^* then $q(m) = r^*_m$.

Alternatively when $n - m < m$ it makes sense to reverse the resource to count words dropped. We can set $d = n - m$ and

$$S_y(i, j) = \delta_{j-i-1} \theta_{ij}, \;\; S_z(i, j) = \delta_0 \omega_{ij}$$

Picking the better of these resources gives a worst-case run-time of $O(n^3 \min\{m, n - m\}^2)$.

6 Training

7 Features

8 Experiments

Premises:

$$(\text{◿}, i, i), (\text{◺}, i, i)\ \forall\, 0 \le i \le j \le n$$

Goal:

$$(\text{◺}, 0, n)$$

Completion Rules:

$$\frac{(\text{⏢}, i, k) \quad (\text{◺}, k, j)}{(\text{◺}, i, j)} \qquad \forall\, 0 \le i < k \le j \le n$$

$$\frac{(\text{◿}, i, k) \quad (\text{⏥}, k, j)}{(\text{◿}, i, j)} \qquad \forall\, 0 < i \le k < j \le n$$

First-Order Rules:

$$\frac{(\text{◺}, i, k) \quad (\text{◿}, k+1, j)}{(\text{⏢}, i, j)}\ S_z(i, j) \qquad \forall\, i \le k < j$$

$$\frac{(\text{◺}, i, k) \quad (\text{◿}, k+1, j)}{(\text{⏥}, i, j)}\ S_z(j, i) \qquad \forall\, i \le k < j$$

Second-Order Rules:

$$\frac{(\text{◺}, i, k) \quad (\text{◿}, k+1, j)}{(\square, i, j)} \qquad \forall\, i \le k < j$$

$$\frac{(\text{◺}, i, i) \quad (\text{◿}, i+1, j)}{(\text{⏢}, i, j)}\ S_z(i, \epsilon, j) \qquad \forall\, i < j$$

$$\frac{(\text{◺}, i, j-1) \quad (\text{◿}, j, j)}{(\text{⏥}, i, j)}\ S_z(j, \epsilon, i) \qquad \forall\, i < j$$

$$\frac{(\text{⏢}, i, k) \quad (\square, k, j)}{(\text{⏢}, i, j)}\ S_z(i, k, j) \qquad \forall\, i \le k < j$$

$$\frac{(\square, i, k) \quad (\text{⏥}, k, j)}{(\text{⏥}, i, j)}\ S_z(j, k, i) \qquad \forall\, i \le k < j$$

Revised Premises:

$$(\text{◿}, i, i), (\xcancel{\text{◺}}, i, i)\ \forall\, 0 \le i \le j \le n$$

Additional Rules:

$$\frac{(\xcancel{\text{◺}}, i, i)}{(\xcancel{\text{◺}}, i, j)} \qquad \forall\, 0 \le i < j \le n$$

$$\frac{(\xcancel{\text{◺}}, i, j)}{(\text{◺}, i, j)}\ S_y(i, j+1) \qquad \forall\, 0 \le i \le j \le n$$

$$\frac{(\xcancel{\text{◺}}, i, k) \quad (\text{◿}, k+1, j)}{(\text{⏢}, i, j)}\ S_y(i, k+1) \otimes S_z(i, \epsilon, j) \qquad \forall\, 0 \le i < k < j \le n$$

[Second-Order only]

Figure 2: Deductive rules for standard dependency parsing and the extensions for compressive dependency parsing. First- and second-order parsing share the same premises, goal, and a set of completion rules. Compressive parsing has a revised set of premises and extends the standard rules with three additional types. S_z and S_y are functions mapping arcs and bigrams, respectively, to semiring values.