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

Supervised approaches to sentence compression have previously been shown to benefit when the dependency structure of the output sentence is included in the inference objective, but prior formulations of this inference task do not admit efficient algorithms. We present a polynomial-time approach to recover dependency trees for compressed sentences in which dynamic programming is used to produce a valid parse tree over a subset of the tokens in the input in order to satisfy a compression rate. In addition to first-order dependency structure, this approach also recovers second-order dependencies as well as a bigram factorization of the output sentence with no asymptotic overhead. Experiments on two standard corpora for sentence compression show improved performance against previous work

1 Introduction

Text-to-text generation problems such as paraphrase generation, sentence fusion and text simplification involve monolingual transformations of text for specific goals such as reducing verbosity or assisting comprehension. The most popular of these problems is the task of sentence *compression* which involves the production of well-formed output sentences that are shorter than their corresponding input sentences while nevertheless preserving their meaning. Compression problems have received significant attention in recent years due to their usefulness in document summarization (**?**) as well as the increasing number of sources of compression data (**?**; **?**; **?**; **?**) for training and evaluation.

Sentence compression is usually formulated as a *token deletion* problem[1] in which the compressed output is synthesized using only the tokens in the input sentence without any reordering or paraphrasing, as seen in the following example from the corpus of **?**).

> **Original:** In 1967 Chapman, who had cultivated a conventional image with his ubiquitous tweed jacket and pipe, by his own later admission stunned a party attended by his friends and future Python colleagues by coming out as a homosexual.
>
> **Compressed:** In 1967 Chapman, who had cultivated a conventional image, stunned a party by coming out as a homosexual.

While a wide variety of structured inference techniques have been proposed to assemble compressed sentences in this setting, the production of compressed dependency trees is prevalent in most recent work for standalone sentence compression (**?**; **?**; **?**; **?**; **?**; **?**) as well as joint compression and summarization (**?**; **?**; **?**). However, each of these approaches is limited by one or more of the following: (i) a restriction that the output tree can only use edges from the input parse, resulting in unreachable gold compressions and increased sensitivity to parser errors, (ii) an intractable formulation which necessitates the use of expensive integer linear programs (ILPs) or algorithms for approximate solutions, and (iii) a reliance on hand-crafted rules or constraints which can be brittle and domain-dependent.

In this work, we present polynomial-time algorithms for *compressive parsing* which generate optimal dependency trees from all possible parse trees over output compressions without any need for heuristics or hand-crafted rules. These dynamic programming strategies extend the well-known Eisner algorithm for projective parsing (**?**) in order to drop a fixed or variable number of tokens in the output parse tree. Our formulation also scores a bigram factorization of the compressed sentence with no asymptotic overhead, thereby of-

[1] Following the terminology used for text summarization, this setting is also referred to as *extractive* compression by **?**) and **?**).

fering an efficient alternative to the ILP for joint n-gram and dependency structured inference proposed by **?**). Finally, this dynamic program can easily be extended to richer second-order compressive parsing (**?**; **?**) in which scores can be defined over consecutive parsing decisions without any increase in runtime.

The contributions of this paper include:

- An $O(n^3)$ time dynamic programming algorithm to jointly recover the optimal compressed dependency tree and bigram factorization over an input sentence of length n when no compression rate is specified.
- An $O(n^3m^2)$ time algorithm to recover the optimal compressed tree and bigram sequence covering exactly $m < n$ tokens.
- A bisection-based first-pass strategy to decrease runtime by recovering a fraction of compressions with m tokens in $O(n^3k)$ where $k << m^2$.

2 Parsing for Compression Inference

A diverse array of inference strategies have been employed in recent years towards standalone sentence compression and compression within extractive summarization. In particular, the production of arc-factored dependency trees to represent compressed sentences has gained appeal by balancing the straightforwardness of inference approaches for n-gram subsequences (**?**; **?**) and the syntactic perspective of tree substitution grammars (**?**; **?**) while performing competitively in experimental evaluations (**?**; **?**; **?**).

Many recent techniques produce compressed dependency trees by pruning edges from a dependency parse of the input sentence or transformations thereof. However, this problem is NP-hard in the general case[2] and has consequently been addressed with expensive integer linear programs (**?**; **?**) or approximations based on reranking (**?**; **?**). Furthermore, a strong reliance on the input parse makes these techniques sensitive to parse errors as well as prone to unreachable reference compressions.

Further constraining the head of the output tree to match that of the input permits only subtrees to be dropped from the input parse; in this case, the best compressed tree can be found in linear time using the Viterbi algorithm for trees. This approach is favored by joint compression and summarization systems (**?**; **?**; **?**; **?**; **?**) but the head restriction further exacerbates the fraction of unreachable reference sentences in a standalone sentence compression setting. Many of these approaches also define hand-crafted rules to ensure that important subtrees like the subject of the head aren't removed.

The joint n-gram and dependency compression approach of **?**) does not restrict output parse trees to lie within input parses but inference remains NP-hard (**?**) because the objective is defined over non-projective trees. However, since tokens cannot be reordered in extractive sentence compression, the recovery of linguistically-motivated[3] *projective* trees becomes tractable. With the exception of non-projectivity, the dynamic programming approach described in the following section fully generalizes over these joint models.

The rest of this section is organized as follows:

3 Related Work

(**?**)

(**?**)

4 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.

[2] A polynomial-time algorithm is viable when the edge weights are integers (**?**).

[3] English is mostly projective and even canonical non-projective languages such as Czech, Danish and Turkish have a low rate (1-2%) of non-projective arcs in their respective treebanks (**?**).

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

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

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

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

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.

5 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.

5.1 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 $\{\triangle, \triangleright, \square, \square\!\!\triangleright, \square\!\!\square\}$ (informally referred to as triangles, trapezoids, and boxes). These items indicate partial structures that may be combined with logical rules. For instance, the rules

$$\frac{(\triangleright, i, k) \quad (\triangle, k+1, j)}{(\square\!\!\triangleright, i, j)} \; S_z(i,j) \qquad \forall \, i \leq k < j$$

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.

5.2 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.

$$\frac{(\triangleright\!\!\!\cdot, i, k) \quad (\triangleleft\!\!\!\cdot, k+1, j)}{(\square, i, j)} \; S_z(i,j) \otimes S_y(k, k+1)$$

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 $0 \leq i \leq k < l < j \leq n$

$$\frac{(\triangleright\!\!\!\cdot, i, k) \quad (\triangleleft\!\!\!\cdot, l, j)}{(\square, i, j)} \; S_z(i,j) \otimes S_y(k, l)$$

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 $(\triangleright\!\!\!\cdot, i, i)$ with new items $(\triangleright\!\!\!-, i, i)$. These special items are only allowed to skip words to their right before becoming standard $\triangleright\!\!\!\cdot$ items

$$\frac{(\triangleright\!\!\!-, i, i)}{(\triangleright\!\!\!-, i, j)} \qquad \forall\, 0 \leq i < j \leq n$$

$$\frac{(\triangleright\!\!\!-, i, j)}{(\triangleright\!\!\!\cdot, i, j)} \; S_y(i, j+1) \qquad \forall\, 0 \leq i \leq j \leq n$$

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 **??**.

5.3 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

$$\frac{(\triangleright\!\!\!\cdot, i, i) \quad (\triangleleft\!\!\!\cdot, i+1, j)}{(\square, i, j)} \; S_z(i, \epsilon, j) \quad \forall i < j$$

which creates the initial right-modifier of i, (i, ϵ, j). This rule implicitly assumes that all items $(\triangleright\!\!\!\cdot, i, j)$ with $i \neq j$ have already taken at least one modifier. However, with the new rules, it is possible that $(\triangleright\!\!\!\cdot, i, j)$ 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.

$$\frac{(\triangleright\!\!\!-, i, k) \quad (\triangleleft\!\!\!\cdot, k+1, j)}{(\square, i, j)} \; S_z(i, \epsilon, j) \quad \forall i \leq k < j$$

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.

6 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.

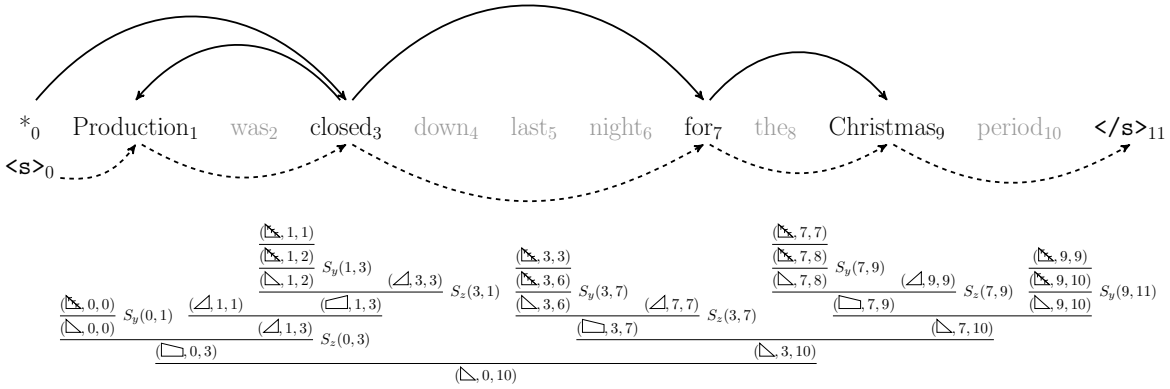

Figure 1: Example of a compressive parse with derivation. The compressed sentence is "Production closed for Christmas". The output structure has arc and bigrams $y_{0,3}, y_{3,1}, y_{3,7}, y_{7,9}, z_{0,1}, z_{1,3}, z_{3,7}, z_{7,9}$, and $z_{9,11}$ set to 1 and all others set to 0. The full first-order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

6.1 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

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

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 $O(n^3 \min\{m, n - m\}^2)$.

7 Features

7.1 Features and Learning

We based the features for our discriminative model on the features described by **?**) and **?**).

Dependency features Ω included the probability of an arc under a smoothed dependency grammar derived from the Penn Treebank, the *fidelity* of the output arc—an indicator of whether it is present in the input parse—conjoined with its dependency labels, and conjunctions of the following features: arc fidelity, fidelity of ancestral relations binned by the distance of the ancestor, direction of attachment, part-of-speech (POS) tags of the tokens incident to the edge, labels of constituent chunks containing these tokens and an indicator of whether they lie in the same chunk.

We also added token-specific features over the dependents of each arc. These included POS tag sequences of length at most 3 around the token, the label of the dependency arc to the token in the input parse conjoined with its POS tag, indicators for capitalized words and words in parentheses as

Premises:

$$(\triangle, i, i), (\triangleright, i, i) \;\; \forall\, 0 \leq i \leq j \leq n$$

Goal:

$$(\triangleright, 0, n)$$

Completion Rules:

$$\frac{(\square, i, k) \quad (\triangleright, k, j)}{(\triangleright, i, j)} \quad \forall\, 0 \leq i < k \leq j \leq n$$

$$\frac{(\triangle, i, k) \quad (\square, k, j)}{(\triangle, i, j)} \quad \forall\, 0 < i \leq k < j \leq n$$

First-Order Rules:

$$\frac{(\triangleright, i, k) \quad (\triangle, k+1, j)}{(\square, i, j)}\, S_z(i, j) \quad \forall\, i \leq k < j$$

$$\frac{(\triangleright, i, k) \quad (\triangle, k+1, j)}{(\square, i, j)}\, S_z(j, i) \quad \forall\, i \leq k < j$$

Second-Order Rules:

$$\frac{(\triangleright, i, k) \quad (\triangle, k+1, j)}{(\square, i, j)} \quad \forall\, i \leq k < j$$

$$\frac{(\triangleright, i, i) \quad (\triangle, i+1, j)}{(\square, i, j)}\, S_z(i, \epsilon, j) \quad \forall\, i < j$$

$$\frac{(\triangleright, i, j-1) \quad (\triangle, j, j)}{(\square, i, j)}\, S_z(j, \epsilon, i) \quad \forall\, i < j$$

$$\frac{(\square, i, k) \quad (\square, k, j)}{(\square, i, j)}\, S_z(i, k, j) \quad \forall\, i \leq k < j$$

$$\frac{(\square, i, k) \quad (\square, k, j)}{(\square, i, j)}\, S_z(j, k, i) \quad \forall\, i \leq k < j$$

Revised Premises:

$$(\triangle, i, i), (\triangleright\!\!\!\!\times, i, i) \;\; \forall\, 0 \leq i \leq j \leq n$$

Additional Rules:

$$\frac{(\triangleright\!\!\!\!\times, i, i)}{(\triangleright\!\!\!\!\times, i, j)} \quad \forall\, 0 \leq i < j \leq n$$

$$\frac{(\triangleright\!\!\!\!\times, i, j)}{(\triangleright, i, j)}\, S_y(i, j+1) \quad \forall\, 0 \leq i \leq j \leq n$$

$$\frac{(\triangleright\!\!\!\!\times, i, k) \quad (\triangle, k+1, j)}{(\square, i, j)}\, S_y(i, k+1) \otimes S_z(i, \epsilon, j) \quad \forall\, 0 \leq i < k < j \leq n$$
[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.

well as lexical features covering verb stems, symbols and negations.

Second-order features were comprised of the fidelity of the second-order dependency conjoined with its label in the input parse. Although we experimented with a wide range of second-order features on a development dataset, we found that they were prone to overfitting, likely due to the small size of the **?**) datasets used in our evaluation.[4]

Finally, bigram features θ include the likelihood under a language model (LM) contructed from the Gigaword corpus, the coarse and fine POS tags of its tokens and the labels of dependency arcs incident to these words in the input sentence.

For the experiments in the following section, we trained models using a variant of the structured perceptron (**?**) which incorporates minibatches (**?**) for easy parallelization and faster convergence.[5] Overfitting was avoided by averaging parameters and monitoring performance against a held-out development set during training.

8 Experiments

9 Experiments

This section describes the experimental setup used to evaluate the utility and performance of our proposed inference approach.

9.1 Datasets

Compression experiments were conducted over the standard newswire (NW) and broadcast news transcription (BN) corpora compiled by **?**). These datasets include gold compressions which were produced by human annotators who were restricted to only delete tokens. We filtered the instances as per **?**) and used the same training/dev/test splits as **?**), resulting in 953/63/603 instances for the NW corpus and 880/78/404 for the BN corpus. Reference dependency parses were approximated by parsing reference compressions with the Stanford dependency parser.[6] Following **?**), which revealed a strong correlation between system compression rate and human judgments of compression quality, all systems were constrained to produce compressed output at a fixed compression rate determined by the the median reference compressions for each instance.

[4] We expect these features to be valuable when training on larger compression datasets such as the one proposed by **?**).

[5] We used a minibatch size of 4 in all experiments.

[6] `http://nlp.stanford.edu/software/`

9.2 Evaluation measures

We evaluated system performance using F_1 metrics over n-grams and dependencies—produced by parsing system output with RASP (**?**) and the Stanford parser—following previous work (**?**; **?**; **?**; **?**; **?**; **?**). In the case of the BN corpus which has 3 reference compressions per instance, F_1 scores were averaged over all references.