Neural Pipeline for Zero-Shot Data-to-Text Generation

Anonymous ACL submission

Abstract

In data-to-text (D2T) generation, training on in-domain data leads to overfitting to the data representation and repeating training data noise. We examine how to avoid finetuning the pretrained language models (PLMs) on D2T generation datasets while still taking advantage of surface realization capabilities of PLMs. Inspired by pipeline approaches, we propose to generate text by rephrasing single-item templates using a sequence of modules trained on general-domain text-based operations—ordering, aggregation, and paragraph compression. We train PLMs for performing these operations on a synthetic corpus WIKIFLUENT which we build from English Wikipedia. Our experiments on two major triple-to-text datasets—WebNLG and E2E—show that our approach enables D2T generation from RDF triples in zero-shot settings.\footnote{The anonymized version of our code and data is available at https://anonymous.4open.science/r/zeroshot-d2t-pipeline/.

1 Introduction

The aim of data-to-text (D2T) generation is to produce natural language descriptions of structured data (Gatt and Kraher, 2018; Reiter and Dale, 1997). Although pipelines of rule-based D2T generation modules are still used in practice (Dale, 2020), end-to-end approaches based on PLMs recently showed superior benchmark performance (Ke et al., 2021; Chen et al., 2020a; Ferreira et al., 2020; Kale and Rastogi, 2020b; Ribeiro et al., 2020), surpassing pipeline systems (Ferreira et al., 2019) in both automatic and human evaluation metrics.

Finetuning PLMs on human-written references is widely accepted as a standard approach for adapting PLMs to the D2T generation objective and achieving good performance on a given benchmark (Agarwal et al., 2021; Ke et al., 2021). Nevertheless, this approach brings issues: Most obviously, finetuning the model for the domain-specific
data distribution leads to overfitting on the particular benchmark, decreasing performance on out-of-distribution data (Laha et al., 2020). Moreover, collecting a large set of references for a particular domain is costly and time-consuming, as the data are usually collected using crowdsourcing (Dušek et al., 2020). Few-shot approaches are an alternative, requiring only several tens or hundreds of annotated examples (Chen et al., 2020c; Ke et al., 2021; Su et al., 2021a). However, robustness of these approaches is questionable—selecting a representative set of examples which would improve performance is difficult (Chang et al., 2021a), and the limited sample is often noisy, increasing the chance of hallucinations and omissions (Dušek et al., 2019; Harkous et al., 2020; Rebuffel et al., 2021).

In this paper, we provide an alternative to this traditional paradigm by formulating the D2T generation from RDF triples as a sequence of general-domain operations over text in natural language. We start by transforming individual triples to text using trivial templates, which we subsequently or-
der, aggregate, and compress on the paragraph level to produce the resulting description of the data. All the pipeline modules operate over natural language text and are built upon PLMs trained on our WIKIFLUENT corpus. WIKIFLUENT contains 934k examples of first paragraphs from the English Wikipedia, each supplied with a synthesized set of simple template-like sentences conveying the same meaning. Our approach allows generating natural language descriptions from triples with a minimum amount of domain-specific rules or knowledge and without using training data from the D2T datasets. We show that our approach can yield large improvements upon simple baselines and match older supervised systems in terms of fluency, while bringing potential for further improvements and advantages with respect to controllability.

Our contributions are the following:

1. We propose an alternative D2T generation approach based on general-domain text-to-text operations (ordering, aggregation, and paragraph compression).
2. We introduce a synthetic WIKIFLUENT corpus containing 934k sentences based on English Wikipedia, providing training data for the operations in (1).
3. We apply our system on two D2T datasets and evaluate its performance both automatically and manually, including the contribution of individual pipeline modules.
4. We release our code, data, pretrained models, and system outputs to ease future research.

2 Related Work

D2T Generation with PLMs Large neural language models pretrained on self-supervised tasks (Lewis et al., 2020; Liu et al., 2019; Devlin et al., 2019) have recently gained a lot of traction in D2T generation research (Ferreira et al., 2020). Following Chen et al. (2020c), other works adopt PLMs for few-shot D2T generation (Chang et al., 2021b; Su et al., 2021a). Kale and Rastogi (2020b) and Ribeiro et al. (2020) showed that PLMs using linearized representations of data can outperform graph neural networks on graph-to-text datasets, recently surpassed again by graph-based models (Ke et al., 2021; Chen et al., 2020a). Although the models make use of general-domain pretraining tasks, all of them are eventually finetuned on domain-specific data.

Templates in Data-Driven D2T Generation Using simple handcrafted templates for individual keys or predicates is an efficient way of introducing domain knowledge while preventing text-to-text models from overfitting to a specific data format (Heidari et al., 2021; Kale and Rastogi, 2020a; Kasner and Dušek, 2020). Transforming individual triples to text is also used in Laha et al. (2020) whose work is the most similar to ours. They also build a three-step pipeline for zero-shot D2T generation, but they use handcrafted rules for producing the output text and do not address content planning.

Content Planning in D2T Generation Content planning, i.e. ordering input facts and aggregating them into individual sentences, is a traditional part of the D2T generation pipeline (Ferreira et al., 2019; Gatt and Krahmer, 2018; Reiter and Dale, 1997). As previously demonstrated, using a content plan in neural D2T generation has important impact on the overall text quality (Moryossef et al., 2019a,b; Puduppully et al., 2019; Zhao et al., 2020; Trisedya et al., 2020). Recently, Su et al. (2021b) have shown that using a content plan leads to improved quality of PLM outputs. All the aforementioned models plan directly using predicates or keys in the D2T datasets representing the corresponding data item. Unlike these works, our planner is trained on ordering sentences in natural language.

Sentence Ordering Sentence ordering is the task of organizing a set of natural language sentences to increase the coherence of a text (Barzilay et al., 2001; Lapata, 2003). Several neural methods for this task were proposed, using either interactions between pairs of sentences (Chen et al., 2016; Li and Jurafsky, 2017), global interactions (Gong et al., 2016; Wang and Wan, 2019), or combination of both (Cui et al., 2020). We base our ordering module (§5.1) on the recent work of Calizzano et al. (2021), who use a pointer network (Wang and Wan, 2019; Vinyals et al., 2015) on top of a PLM.

Fact Aggregation The compact nature of the target text description results in aggregating multiple facts in a single sentence. Previous works (Wisman et al., 2018; Shao et al., 2019; Shen et al., 2020; Xu et al., 2021) capture the segments which correspond to individual parts of the input as latent variables. Unlike these works, we adopt a simpler scenario using an already ordered sequence of facts, in which we selectively insert delimiters marking sentence boundaries.
We assume that we can transform each triple \( t \) into a natural language description \( Y \) for a set of \( n \) RDF triples \( X = \{x_1, \ldots, x_n\} \). Each triple \( x_i = \{s_i, p_i, o_i\} \) consists of subject \( s_i \), predicate \( p_i \), and object \( o_i \).

We assume that we can transform each triple \( x_i \) to a fact \( f_i \) (where \( f_i \) is a sentence in natural language describing \( x_i \)) by filling the single-triple template \( t_{p_i} \in T \) for the predicate \( p_i \): \( t_{p_i}(s_i, o_i) \rightarrow f_i \).

We proceed as follows – given an input \( X \), we:

1. apply the templates to transform the set of triples \( X \) to the set of facts: \( F = T(X) = \{f_1, \ldots, f_n\} \) (§3.2),
2. sort the facts \( F \) using an ordering module which outputs an ordered sequence of facts \( F_o = O(F) = \{f_{o_1}, \ldots, f_{o_n}\} \) (§3.3),
3. obtain sentence delimiters by inputting the ordered facts \( F_o \) into an aggregation module \( A(F_o) = \{\delta_{o_1}, \delta_{o_2}, \ldots, \delta_{o_{n-1}}\} \) where \( \delta_{o_i} \) is a delimiter, i.e., that the sentences with facts \( f_{o_i} \) and \( f_{o_{i+1}} \) should not be fused (§3.4),
4. input the ordered sequence with delimiters \( F_a = \{f_{o_1}, \delta_{o_1}, f_{o_2}, \ldots, \delta_{o_{n-1}}, f_{o_n}\} \) into the paragraph compression module which generates the final description \( P(F_a) = Y \) (§3.5).

3.2 Templates

The first step in our pipeline involves transforming each of the input triples \( X \) into a set of facts \( F \) in natural language by using a template \( t_{p_i} \) for each predicate \( p_i \). We need at least one template for each predicate. Typically, the template will include placeholders which are filled with \( s_i \) and \( o_i \).

The transformation serves two purposes: (a) preparing the data for the subsequent text-to-text operations, (b) introducing in-domain knowledge about the semantics of individual predicates. Note that the filled templates are allowed to contain minor disfluencies since the text will be rephrased in the final step of the pipeline. See §5.5 for our approach to gathering the templates and Figure 2 for examples of the templates we use in our datasets.

We acknowledge that this step may be a bottleneck on datasets with an unconstrained (or very large) set of predicates, which is why we also discuss possibilities for automating this step in §7.

3.3 Ordering

We assume that the default order of triples \( X \) (and the respective facts \( F \)) is random. To maximize the coherency of the resulting description, we apply an ordering model \( O \) to get an ordered sequence of facts: \( F_o = \{f_{o_1}, \ldots, f_{o_n}\} \). The coherence of the final text will also depend on the paragraph compression step, but grouping related facts together (e.g. facts mentioning \( \text{birth date} \) and \( \text{birth place} \)) helps the paragraph compression module to focus only on fusing and rephrasing the neighboring sentences. We describe our ordering model in §5.1.
3.4 Aggregation

The aggregation model takes a sequence of ordered facts \( F_i \) as input and produces a sequence of sentence delimiters \( A(F_o) = \{ \delta_0, \delta_2, \ldots, \delta_{n-1} \} \); \( \delta_i \in \{0, 1\} \). The output \( \delta_i = 1 \) means that the neighboring facts are should be mentioned separately, serving as a hint for the paragraph compression model not to fuse the neighboring sentences. Conversely, \( \delta_i = 0 \) means that the facts should be aggregated and their corresponding sentences should be fused (see §5.2 and §5.3).

3.5 Paragraph Compression

The paragraph compression model (see §5.3 for simplified variants) takes as input the ordered sequence of facts with delimiters \( F_u = \{ f_0, \delta_0, f_2, \ldots, \delta_{n-1}, f_n \} \) and produces a resulting text \( Y \). The objectives of the model are two-fold: (a) fusing related sentences, i.e., sentences \( i \) and \( j \) in between which \( \delta_i = 0 \), and (b) rephrasing the text to improve its fluency, e.g. fixing minor disfluencies in the templates, replacing noun phrases with referring expressions, etc. The focus is on minor rephrasing since the goal is to preserve the semantics of the original text.

4 WIKIFLUENT Corpus

A key to our approach is building a large-scale synthetic corpus providing training data for the text operations in our pipeline. Our corpus needs to cover a broad range of domains while capturing the sentence style in D2T generation, both regarding the input templates and the target descriptions. In other words, we aim to build a corpus in which:

- the input is a set of simple, template-like sentences,
- the output is a fluent text in natural language preserving the semantics of the input.

As we describe below in detail, we achieve that by applying a split-and-rephrase model and a coreference resolution model on a set of human-written paragraphs in English Wikipedia. We consider the processed text as a source and the original text as the target. The process is illustrated in Figure 3; corpus statistics are included in Appendix A.

4.1 Data Source

For building the WIKIFLUENT corpus, we extracted 934k first paragraphs of articles from a Wikipedia dump\(^2\) using WikiExtractor (Attardi, 2015). The paragraphs contain mostly concise, fact-based descriptions from a wide range of domains. We selected paragraphs with length between 30–430 characters, filtering out lists, disambiguations, repeated and malformed paragraphs. To further ensure that the length of inputs is balanced, we selected 250k examples each from 4 equidistant length ranges (30–130 characters, etc.).

4.2 Split-and-Rephrase

For generating the target set of sentences, we divide each paragraph into sentences using NLTK (Bird, 2006) and apply a split-and-rephrase model on each sentence. Split-and-rephrase is a task of splitting a complex sentence into a meaning preserving sequence of shorter sentences (Narayan et al., 2017). We train our model on the large-scale WikiSplit corpus by Botha et al. (2018), containing human-made sentence splits from Wikipedia edit history. Following the setup in the rest of our experiments, we train the encoder-decoder PLM BART-base (Lewis et al., 2020) on the WikiSplit dataset in a sequence-to-sequence setting. We apply the trained split-and-rephrase model on each sentence, uniformly randomly choosing between 0-2 recursive calls to ensure that the splits are not deterministic. If the sentence cannot be meaningfully split, the model tends to duplicate the sentence on the output; in that case, we use only the original sentence and do not proceed with the splitting.

4.3 Coreference Replacement

Next, we concatenate the split sentences and apply a coreference resolution model (Gardner et al., 2018) in order to replace referring expressions with their antecedents (e.g., pronouns with noun phrases). This allows to better follow the style of

\(^2\)enwiki-20210401-pages-articles-multistream
4.4 Filtering

To assert that the generated sentences convey the same semantics as the original paragraph, we use a pretrained RoBERTa model\(^3\) (Liu et al., 2019) trained on the MultiNLI dataset (Williams et al., 2018) for checking the semantic accuracy of the generated text. Following Dušek and Kasner (2020), we test if the original paragraph entails each of the synthesized sentences (checking for omissions), and if the set of concatenated synthesized sentences entails the original paragraph (checking for hallucinations). In a filtered version of the WIKI-FLUENT corpus, we include only the examples without omissions or hallucinations (as computed by the model), reducing it to approximately 3/4 of the original size.

5 Experiments

We show how we build our pipeline (§5.1-5.4) and discuss the D2T generation datasets which we use for our experiments (§5.5). The details of our training setup are included in Appendix B.

5.1 Ordering Model

For our ordering model (see §3.3), we use the Simple Pointer model from Calizzano et al. (2021). The model is based on a pretrained BART-base extended with a pointer network from Wang and Wan (2019). We provide a short description of the model here; for details see Calizzano et al. (2021).

In the encoding phase, facts \(F\) are concatenated and tokenized. Each fact is surrounded by special tokens denoting the beginning (<s>) and the end (</s>) of the fact. The sequence is processed by the BART encoder, generating a sequence of encoder states \(E\) for each end token </s> representing the preceding fact.

The decoding proceeds autoregressively. To bootstrap the decoding process, the pair of tokens <s>/<s> is fed into the decoder, producing the decoder state \(d_1\). The pointer network (attending to \(d_1\) and \(E\)), selects the first ordered fact \(f_{0}\), which is fed into the decoder in the next step. The process is repeated until the all the facts are decoded in a particular order.

The pointer network computes the probability of a fact to be on the \(j\)-th position, using the encoder output \(E\) and the decoder output \(d_j\). The network is based on the scaled dot product attention, where \(d_j\) is the query and encoder outputs \(E_i\) are the keys:

\[
Q = d_j W_Q \\
K = E W_K \\
P_j = \text{softmax} \left( \frac{QK^T}{\sqrt{b}} \right). 
\]

Here \(W_Q\) and \(W_K\) are matrices in \(\mathbb{R}^{b \times b}\), \(b\) is the dimension of BART hidden states, and \(P_j \in \mathbb{R}^{n+1}\) is the probability distribution for the \(j\)-th position (i.e., \(P_j\) is the probability that fact \(f_i\) is on the \(j\)-th position).

We train the model using the split sentences in the WIKI-FLUENT corpus, randomly shuffling the order of the sentences and training the model to restore their original order.

5.2 Aggregation Model

We base our aggregation model (cf. §3.4) on RoBERTa-large (Liu et al., 2019) with a token classification head.\(^4\) Similarly to the ordering model (§5.1), we input the sequence of facts \(F\) into the model, separating each pair of facts \(f_{0}\) with a special token </s> (used by the model as a separator). Subsequently, the token classification layer classifies each separator </s> position into two classes \(\{0, 1\}\) corresponding to the delimiter \(\delta_i\). We ignore the outputs for the non-separator tokens while computing the cross-entropy loss.

We create the training examples using the split sentences in the WIKI-FLUENT corpus, in which we set \(\delta_i = 0\) for the sentences \(i, i+1\) which were originally aggregated (i.e., are the result of splitting a single sentence) and \(\delta_i = 1\) otherwise.

5.3 Paragraph Compression Model

We adopt BART-base for our paragraph compression model. We train the model in a sequence-to-sequence setting on the WIKI-FLUENT corpus, concatenating the split sentences on the input. We add delimiters between sentences \(i\) and \(i+1\) where \(\delta_i = 1\) using a special token <sep>, which we add to the model vocabulary. As shown in Keskar et al. (2019), including control codes for training

\(^3\)https://huggingface.co/roberta-large-mnli

\(^4\)https://huggingface.co/transformers/model_doc/roberta.html#robertafortokenclassification
A dam obstructs flowing water.

We test our approach on two English D2T datasets, WebNLG and E2E. They differ in domain, size, text-based representation, and number of predicates (see Appendix A for details).

The model can steer the model towards producing certain outputs. We evaluate our model’s behavior with respect to ordering and aggregation in §6.3.

5.4 Ablation Study

In order to evaluate individual components of our pipeline, we train three versions of the PC model (see §5.3). The models share the same architecture and targets, but differ in their inputs:

- **PC**: the model takes as an input ordered facts with delimiters (as described in §3.5),
- **PC+AGG**: the model takes as an input ordered facts without delimiters (i.e., the aggregation is left implicitly to the model),
- **PC+ORD+AGG**: the model takes as an input facts in random order and without delimiters (i.e., both ordering and aggregation are left implicitly to the model).

Subsequently, we test three versions of the pipeline (see Figure 4):

- **3-STAGE**: a full version of the pipeline consisting of the ordering model, the aggregation model and the PC model (following the full pipeline from §3),
- **2-STAGE**: a pipeline consisting of the ordering model and the PC+AGG model,
- **1-STAGE**: a single stage consisting of the PC+ORD+AGG model.

We evaluate all versions of the pipeline with PC models trained on the full and filtered versions of the WikiFluent dataset (see §4).

5.5 D2T Datasets

We test our approach on two English D2T datasets, WebNLG and E2E. They differ in domain, size, textual style, and number of predicates (see Appendix A for details).

**WebNLG**: The WebNLG dataset (Gardent et al., 2017) contains RDF triples from DBPedia (Auer et al., 2007) and their crowdsourced descriptions. The dataset was extended for the WebNLG+ Challenge (Ferreira et al., 2020), but we use the version 1.4 for comparability to prior work. Templates for WebNLG could be extracted from the training data by delexicalizing single-triple examples. However, the examples are noisy and such data would not be available in a zero-shot setup. Therefore, we handcrafted templates for all 354 predicates, including unseen predicates in the test set.\(^5\)

**E2E**: The E2E dataset (Novikova et al., 2017; Dušek et al., 2020) contains restaurant recommendations in the form of attribute-value pairs. We use the cleaned version of the dataset (Dušek et al., 2019). Following previous work, we transformed the attribute-value pairs into RDF triples (using the restaurant name as a subject) and then applied the same setup as for WebNLG. We created a template for each of the 8 attributes manually.

6 Evaluation

We evaluate outputs from the \{1,2,3\}-STAGE variants of our pipeline automatically (§6.1) and manually (§6.2). Further, we evaluate the performance of the content planning modules and the ability of the PC module to follow the content plan (§6.3).

6.1 Automatic Metrics

Following prior work, we use BLEU (Papineni et al., 2002) and METEOR (Banerjee and Lavie, 2005) to evaluate the outputs against the human references.\(^6\) We also evaluate the number of omission and hallucination errors (i.e., facts missing or added, respectively) using a metric from Dušek and Kasner (2020) based on a RoBERTa model (Liu et al., 2019) pretrained on natural language inference (NLI).\(^7\)

---

\(^5\)The templates are single-sentence and mostly clear-cut verbalizations of the predicates. We did not use human references from the dataset when creating the templates.

\(^6\)We use the implementation from https://github.com/tuetschek/e2e-metrics.

\(^7\)We additionally evaluated the outputs on the E2E dataset using the provided pattern-based slot error script. See Ap-
Table 1: Example outputs of our model (3-STAGE, filtered). See Appendix D for more examples.

<table>
<thead>
<tr>
<th>Input</th>
<th>Templ.</th>
<th>Human</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Allen Forrest; background; solo singer), (Allen Forrest; genre; Pop music), (Allen Forrest; birthPlace; Dothan, Alabama)</td>
<td>Allen Forrest is a solo singer. Allen Forrest performs Pop music. Allen Forrest was born in Dothan, Alabama.</td>
<td>Born in Dothan, Alabama, Allen Forrest has a background as a solo singer and was a pop artist.</td>
</tr>
</tbody>
</table>

Table 2: Automatic metrics on WebNLG. B = BLEU, M = METEOR, O = omissions / # facts, H = hallucinations / # examples. The systems marked with asterisk (*) are copied from the respective papers. We manually evaluated 100 outputs of the models (especially in E2E) and produces frequent hallucinations (incorrect fact merging, redundancies) as well as grammatical errors. The results for the systems marked with † are copied from the respective papers.

<table>
<thead>
<tr>
<th>Model</th>
<th>Human</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPF-FORGe*</td>
<td>A amazing French restaurant is called the Wildwood. The restaurant is near the Raja Indian Cuisine in riverside.</td>
<td>name[Wildwood], eatType[restaurant], food[French], area[riverside], near[Raja Indian Cuisine]</td>
</tr>
<tr>
<td>Ke et al. (2021)*</td>
<td>Wildwood is a restaurant serving French food. It is in the riverside near Raja Indian Cuisine.</td>
<td>Templ.</td>
</tr>
<tr>
<td>Laha et al. (2020)*</td>
<td>Wildwood is a restaurant. Wildwood serves French food. Wildwood is in the riverside. Wildwood is near Raja Indian Cuisine.</td>
<td>Input</td>
</tr>
<tr>
<td>COPY</td>
<td>Wildwood is a restaurant serving French food. It is in the riverside near Raja Indian Cuisine.</td>
<td>Human</td>
</tr>
</tbody>
</table>

Table 3: Automatic metrics on E2E. B = BLEU, M = METEOR, O = omissions / # facts, H = hallucinations / # examples. The systems marked with asterisk (*) are trained on the E2E dataset. The results for Harkous et al. (2020) are copied from the paper.

6.2 Manual Evaluation

We manually evaluated 100 outputs of the models regarding factual errors (hallucinations, omissions, incorrect fact merging, redundancies) as well as grammatical errors. The results are listed in Table 4. The 1-STAGE model (which has to order the facts implicitly) tends to repeat the facts in the text (especially in E2E) and produces frequent hallucinations. These problems are only slightly reduced in the filtered version, but they are largely elim-
7 Discussion and Future Work

In the current form, our pipeline can be directly applied to generating text from RDF triples (or similarly structured data) which require no extra processing. Further extensions are needed for more complex D2T scenarios, e.g., datasets requiring content selection or common-sense and logical reasoning (Wiseman et al., 2017; Lin et al., 2019; Chen et al., 2020b).

Our approach regarding handcrafting a single template for each predicate is quite basic. Generating simple statements from the triples automatically, e.g., using the approach of Laha et al. (2020), could reduce the manual workload and allow applying our approach on datasets with a less constrained set of data attributes such as ToTTo (Parikh et al., 2020) or DART (Nan et al., 2021). Moreover, explicitly including a denoising task for the paragraph compression model could help to tackle the disfluencies in the templates.

More research is also needed on semantic errors stemming from merging of facts in improper ways. We suggest that explicitly controlling the semantics of sentence fusion (Ben-David et al., 2020) could help to mitigate this issue, while still keeping the advantages of a zero-shot approach.

Table 5: Evaluation of our zero-shot ordering model based on Calizzano et al. (2021). The results marked with † are copied from the respective papers.

<table>
<thead>
<tr>
<th></th>
<th>WebNLG</th>
<th>E2E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H I O R G</td>
<td>H I O R G</td>
</tr>
<tr>
<td>full 3-STAGE</td>
<td>3 39 2 2 16</td>
<td>0 1 0 0 17</td>
</tr>
<tr>
<td>2-STAGE</td>
<td>8 36 1 5 16</td>
<td>1 1 0 1 23</td>
</tr>
<tr>
<td>1-STAGE</td>
<td>28 27 6 10 20</td>
<td>17 0 1 79 45</td>
</tr>
<tr>
<td>filtered 3-STAGE</td>
<td>2 37 2 1 15</td>
<td>0 0 0 0 17</td>
</tr>
<tr>
<td>2-STAGE</td>
<td>5 32 1 2 14</td>
<td>0 0 0 0 11</td>
</tr>
<tr>
<td>1-STAGE</td>
<td>8 40 6 6 16</td>
<td>11 2 1 41 22</td>
</tr>
</tbody>
</table>

Table 4: Number of manually annotated errors on 100 examples: H = hallucinations, I = incorrect fact merging, O = omissions, R = redundancies, G = grammar errors or disfluencies.
References


A Dataset Statistics
Statistics for the datasets described in the paper are listed in Table 7.

B Experimental Setup

B.1 Our Models
We implemented the models for split-and-rephrase, aggregation, and paragraph compression in PyTorch Lightning (Paszke et al., 2019), using the PyTorch (Falcon, 2019) version of the BART and RoBERTa models from the Huggingface library (Wolf et al., 2019).

We use the Adam (Kingma and Ba, 2015) optimizer (β1 = 0.9, β2 = 0.997, ε = 1−9) with learning rate 2−5, linear scheduling and 0.1 warmup proportion; batches of size 8 and accumulating gradients with factor 4. We train the models for 1 epoch on a single GeForce RTX 3090 GPU with 24 GB RAM. We use greedy decoding in all our experiments.

B.2 Ordering
For training the ordering model, we used the implementation from Calizzano et al. (2021)8 including their training parameters. We plan to fully integrate the ordering model into our framework in the future.

C Additional Results

We provide evaluation of semantic accuracy on the E2E dataset as evaluated with the slot-error script based on matching regular expressions in Table 6.9

Note that our manual investigation of a sample of the data shows that the majority of the errors identified in our model outputs are false. For example, the following regular expression used in the slot-error script:

```
prices(?: range)?(?:w+)0,3 high
```

matches "(...) price range and high customer rating (...)\"" incorrectly classifying the presence of the extra slot priceRange[high]. This importance of this problem is exacerbated by the consistent

---

8https://github.com/airKlizz/passage-ordering

9https://github.com/tuetschek/e2e-cleaning/blob/master/slot_error.py
<table>
<thead>
<tr>
<th></th>
<th>miss</th>
<th>add</th>
<th>miss+add</th>
</tr>
</thead>
<tbody>
<tr>
<td>TGEN COPY</td>
<td>0.0060</td>
<td>0.0433</td>
<td>0.0016</td>
</tr>
<tr>
<td>full 3-STAGE</td>
<td>0.0238</td>
<td>0.0698</td>
<td>0.0060</td>
</tr>
<tr>
<td>full 2-STAGE</td>
<td>0.0054</td>
<td>0.0363</td>
<td>0.0000</td>
</tr>
<tr>
<td>full 1-STAGE</td>
<td>0.0043</td>
<td>0.0330</td>
<td>0.0000</td>
</tr>
<tr>
<td>full filtered</td>
<td>0.0444</td>
<td>0.0487</td>
<td>0.0076</td>
</tr>
<tr>
<td>filtered 3-STAGE</td>
<td>0.0043</td>
<td>0.0368</td>
<td>0.0000</td>
</tr>
<tr>
<td>filtered 1-STAGE</td>
<td>0.0043</td>
<td>0.0347</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

Table 6: Proportion of output examples with missed only, added only, and both missed and added facts, according to the regular-expression-based E2E slot error script.

outputs of our models, which tend to repeat certain patterns. However, we also manually identified several cases in which an error was found correctly, e.g. the model hallucinating "3 out of 4 customer rating" instead of "3 out of 5 customer rating".

D Example Outputs

Tables 8, 9, 10, and 11 show examples of behavior of our models on the WebNLG dataset. Tables 12 and 13 show examples of behavior of our models on the E2E dataset.

The green color marks the model outputs which are completely correct, the red color marks the errors. For better readability of the input format, we add numeric order identifiers for the individual facts (bold, in squared brackets). These are subsequently used as references in the Order and Aggregation rows. Note that zero-th input in E2E is used as a subject in the RDF triples.
Table 7: Number of examples (train / dev / test), average number of tokens per source and target, average number of sentences per source and target (after filling the templates for the D2T datasets).

<table>
<thead>
<tr>
<th>Dataset</th>
<th># train</th>
<th># dev</th>
<th># test</th>
<th># tok</th>
<th>tok/src</th>
<th>tok/tgt</th>
<th>sent/src</th>
<th>sent/tgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebNLG</td>
<td>18,102</td>
<td>870</td>
<td>1,862</td>
<td>26.8</td>
<td>22.6</td>
<td>3.0</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>Clean E2E</td>
<td>33,236</td>
<td>4,299</td>
<td>1,847</td>
<td>29.2</td>
<td>22.3</td>
<td>4.2</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td>WikiFluent-full</td>
<td>915,855</td>
<td>9,346</td>
<td>9,346</td>
<td>52.9</td>
<td>41.1</td>
<td>3.9</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>WikiFluent-filtered</td>
<td>700,517</td>
<td>7,149</td>
<td>7,149</td>
<td>45.6</td>
<td>35.4</td>
<td>3.4</td>
<td>1.8</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Correct behavior of the models on WebNLG. All the models (1, 2, 3-stage full/filtered) produce the same output.

| Input | [1] (Andrews County Airport; elevationAboveTheSeaLevel (in metres); 973.0) [2] (Andrews County Airport; runwayLength; 896.0) [3] (Andrews County Airport; location; Texas) | Templates | Andrews County Airport is 973.0 metres above the sea level. Andrews County Airport runway is 896.0 m long. Andrews County Airport is located in Texas. | Order | 3 2 1 | Aggregation | 3 <sep> 2 1 | Models (all) | Andrews County Airport is located in Andrews County, Texas. Its runway is 896.0 m long and 973.0 metres above the sea level. | Human | The runway length of Andrews County Airport (located in Texas and 973.0 above sea level) is 896. |

Table 9: Incorrect behavior on WebNLG: besides the minor disfluencies caused by the templates ("Akron, Ohio is from..."), the models (except for 3-stage filtered) tend to hallucinate and merge the facts incorrectly.

| Input | [1] (Akron, Ohio; country; United States) [2] (United States; language; English language) | Templates | Akron, Ohio is from United States. English language is spoken in United States. | Order | 1 2 | Aggregation | 1 <sep> 2 | 3-stage full | Akron, Ohio is from United States. English language is spoken in Akron, Ohio, United States, which is located in the U.S. state of Ohio. | 2-stage full | Akron, Ohio is from United States. English language is spoken in Akron, Ohio. | 1-stage full | English language is spoken in the United States. Akron, Ohio is from the state of Ohio in the U.S. state of Illinois. | 3-stage filtered | Akron, Ohio is from United States. English language is spoken in the United States of America. | 2-stage filtered | Akron, Ohio is from United States. English language is spoken in Akron, Ohio. | 1-stage filtered | English language is spoken in the United States. Akron, Ohio is from the United Kingdom. | Human | The country of Akron, Ohio, is United States, where English is the language spoken. |
Table 10: Incorrect behavior on WebNLG: by using a pronoun "He", the models tend to imply that Alberto Teisaire (instead of Juan Perón) belongs to the Labour Party.

Table 11: Incorrect behavior on WebNLG: the models tend to merge the country of vehicle’s assembly and its capital.
For a central family friendly venue serving cheap Chinese food, try The Cricketers restaurant. It has average customer ratings and is near All Bar One.

Table 12: Correct behavior of the models on E2E: the models can deal even with a higher amount of input values.
**Input**

- [0] name[The Phoenix]
- [1] area[riverside]
- [2] priceRange[cheap]
- [3] familyFriendly[yes]
- [4] near[Raja Indian Cuisine]
- [5] eatType[restaurant]

**Templates**

- The Phoenix is in the riverside.
- The Phoenix has cheap price range.
- The Phoenix is family-friendly.
- The Phoenix is near Raja Indian Cuisine.
- The Phoenix is a restaurant.

**Order**

- 5 1 4 <sep> 3 2

**Aggregation**

- 5 1 4 3 2

**3-stage full**

- The Phoenix is a restaurant in the riverside near Raja Indian Cuisine. It is family-friendly with cheap price range.

**2-stage full**

- The Phoenix is a restaurant in the riverside near Raja Indian Cuisine. It is family-friendly with cheap price range.

**1-stage full**

- The Phoenix is a restaurant near Raja Indian Cuisine in the riverside. It is family-friendly and has cheap price range. It has cheap food range.

**3-stage filtered**

- The Phoenix is a restaurant in the riverside near Raja Indian Cuisine. It is family-friendly with cheap price range.

**2-stage filtered**

- The Phoenix is a restaurant in the riverside near Raja Indian Cuisine. It is family-friendly with cheap price range.

**1-stage filtered**

- The Phoenix is a restaurant near Raja Indian Cuisine in the riverside. It is family-friendly and has cheap price range. It has cheap food.

**Human**

Cheap food and a family friendly atmosphere at The Phoenix restaurant. Situated riverside near the Raja Indian Cuisine.

Table 13: Incorrect behavior on E2E: the 1-STAGE models add redundant information to the output.