RDFJSREALB: a Symbolic Approach for Generating Text from RDF Triples

Guy Lapalme
RALI-DIRO / Université de Montréal
CP. 6128, Succ. Centre-Ville
Montréal, Québec, Canada
lapalme@iro.umontreal.ca

Abstract

This paper describes the Resource Description Framework (RDF) triples verbalizer developed for the WEBNLG CHALLENGE 2020 shared task. After reviewing representative works in Natural Language Generation in the context of the Semantic Web, the task is then described. We then sketch the symbolic approach we used for verbalizing RDF triples: once the triples are grouped by subject, each group is realized as one or more sentences using templates written in Python whose output is feed to an English realizer written in Javascript. The system was developed using the test data of the previous edition of the task and the train and development data of this year's task. The automatic scores for this year's test data are quite competitive. We conclude with a critical review of the data and discuss the suitability of this competition results in a wider Natural Language Generation setting.

1 Introduction

This paper describes our system developed for participating to the *RDF-to-text generation for English* subtask of the WEBNLG CHALLENGE 2020 (Castro-Ferreira et al., 2020) as a follow-up to 2017 edition (Gardent et al., 2017b). The WEBNLG CHALLENGE 2020 data was developed for pushing the development of Resource Description Framework (RDF) verbalizers for realizing short texts while dealing with micro-planning problems such as sentence segmentation and ordering, referring expression generation and aggregation. The first edition of the data featuring 15 DBpedia categories was created in 2017 (Gardent et al., 2017a). The 2020 challenge covers more categories and an additional language, Russian and a new task: Text-to-RDF semantic parsing for converting a text into the corresponding set of RDF triples.

Our English RDF verbalizer is based on a symbolic approach using Python: each RDF triple corresponds to a sentence in which the subject and the object of a triple are mapped almost verbatim as subject and object of the sentence. The predicate of the triple corresponds to a verb phrase which determines the structure of the sentence. The predicates are ordered to create a meaningful story and parts of sentences are merged when they share subjects or predicates. The final realization is performed using JSREALB[1], a French-English realizer that we upgraded in recent years.

After presenting some representative natural language generation works dealing with Semantic Web information, we describe our approach and analyze the scores of the automatic evaluation on WEBNLG CHALLENGE 2020 test data. We then give a personal appraisal on the data for WEBNLG CHALLENGE 2020 and how it is representative of real Semantic Web data and conclude by giving ideas for future development.

2 Related work

Generating text from ontologies and RDF has a long history, because it is a typical case of computer-oriented data about entities that need to be explained to a human in order to be understood or modified. Bouayad-Agha et al. (2014) present a comprehensive discussion of Semantic Web concepts and its relation with Natural Language Generation. One challenge of RDF verbalization is the fact that the information is spread over a graph linking many entities that must be linearized as a coherent text. RDF triples cannot be *lexicalized* directly as they need links with other information in order to be conveyed correctly. Fortunately, most well-organized RDF triples depend on an ontology,

[1] See https://github.com/rali-udem/jsRealB for documentation, a tutorial and examples of use.

3rd International Workshop on Natural Language Generation from the Semantic Web (WebNLG+), Dublin, Ireland (Virtual), 18 December 2020, pages 144–153,

a hierarchy of concepts, that allows inferencing to help drive the generation process.

Some systems target the generation of texts for explaining ontologies while other use ontologies for explaining facts expressed in RDF which reflects two trends in the Semantic Web area: ontology construction using the Web Ontology Language (OWL) with an emphasis on the logical consistency, or the publication of a large number of linked data without necessarily ensuring consistency. WEBNLG CHALLENGE 2020 targets the latter because it is limited to the verbalization of a few RDF triples, but we think it is interesting to briefly look at the former, because it gives a broader view of potential applications for future developments.

2.1 Explaining or Using Ontologies

Aguado et al. (1998) motivate the use of text generation for explaining ontologies to help their reuse. They illustrate their approach by generating Spanish explanations in the domain of chemical substances. They combine the *General Upper Model* (GUM) (Bateman et al., 1995) approach with the KPML (Bateman, 1997) text realizer. Wilcock and Jokinen (2003) use the information in the ontology as background information for a dialogue system that provides information about a public transportation system. The ontology serves both as a source of information and for identifying misconceptions and suggesting alternative reasonable questions. Bontcheva and Wilks (2004) show how to generate reports from domain ontologies; they present a use case in the area of breast cancer in which the concepts of the ontology were manually mapped to words of a specialized lexicon.

Galanis et al. (2009) describe NATURALOWL[2], a plug-in for the PROTÉGÉ[3] ontology editor that produces template-based descriptions of entities and classes from OWL ontologies that have been annotated with linguistic and user modeling information expressed in RDF. Given that it is open-source and embedded in a widely used ontology editor, it has been used as a baseline in many subsequent works.

2.2 Verbalizing RDF statements

The first step in generating texts from RDF is finding an appropriate subset of RDF statements. Duboue and McKeown (2003) were pioneers in determining relevant content for NLG using statistical methods for the extraction of facts from texts and coupling them with the associated data. This approach inspired most recent learning methods given the availability of a large number of texts and associated data in DBPedia and Wikipedia. Duboue and McKeown had to parse HTML pages and databases to find a large set of biographies. For WEBNLG CHALLENGE 2020, this essential but difficult step has already been done by the organizers of the competition (Gardent et al., 2017a).

Sun and Mellish (2006) take advantage of the fact that RDF representations are not only logical representations, but that they also contain rich linguistic information useful for generating text. After studying many published ontologies, they found systematic patterns in class and relation names that can be exploited for lexicalization without developing special purpose dictionaries.

Duma and Klein (2013) propose a system that can automatically learn sentence templates and document planning from parallel RDF data and text from the Simple English Wikipedia. They first match named entities in the sentence with a graph related to a specific entity in order to extract a template in which named entities are replaced by a variable. To prune entities and their dependents in the sentence not appearing in the graph, the sentences are first parsed and a few hand-written rules operating on the syntactic tree are applied as suggested by Gagnon and Da Sylva (2006) for summarization purposes. The content selection uses the method originally suggested by Duboue and McKeown. To determine relevant predicates, they determine a *prototypical* class by looking at the most frequent *subwords* in the class names which often use *camelCase*. Given the URI of an entity to be described, they determine the relevant class and its associated templates that are used for creating many sentences from which the best ones are chosen.

Ell and Harth (2014) show how to extract verbalization templates for RDF graphs from DBPedia and the corresponding Wikipedia documents about specific types of entities. Sentences that mention the entities are aligned with a data graph using language-independent transformations. The connected entities in the graph are then iteratively explored to find commonalities that allow some abstraction of the entities using variables. They thus

[2] https://protegewiki.stanford.edu/wiki/NaturalOWL

[3] https://protege.stanford.edu

obtain a set of abstracted sentences from which templates are created. They applied their technique on English and German with *promising* results.

Dong and Holder (2014) present *Natural Language Generation from Graphs* (NLGG) with three processing stages: model preparation and content determination, document structuring, and lexicalization, aggregation and realization to create English text. It uses templates that code linguistic information about each class such as its English label both singular and plural; the relations are also coded with templates that indicate the type of its subject and object and priority to drive the text organization. Model preparation uses an RDF reasoner to infer new triples and remove redundant ones. Document structuring consists in deciding the order of output: first classes, then attributes and finally relationship information. SIMPLENLG (Gatt and Reiter, 2009) is used for creating the English text. Our system follows a similar approach.

Vougiouklis et al. (2018) describe a statistical model for NLG by adapting the encoder–decoder framework to generate textual summaries for triples related to biographies. They use sequence to sequence methods to jointly perform content selection and surface realization without any rules or templates. Since triples are not sequentially correlated, they develop a feed-forward neural network that encodes each triple into a vector of fixed dimensionality in a continuous semantic space in which triples having similar semantic meaning have similar positions. This encoder is coupled with an RNN-based decoder that generates the textual summary one token at a time.

Gardent et al. (2017b) present the results of the WEBNLG CHALLENGE 2017 that compared the output produced by 8 submissions by 6 teams: three submissions used a template or grammar-based pipeline framework combined with a symbolic realizer similar to our approach. One system used a statistical machine translation framework and four submissions used an attention-based encoder-decoder architecture built using existing neural machine translation frameworks.

Zhu et al. (2019) propose a way of improving the quality of the text at the expense of diversity by optimizing the inverse KL divergence for conditional language generation. Their paper presents a detailed discussion on the fundamental problems of minimizing KL divergence in training for this problem and justify the inverse KL divergence as their optimization objective.

We now present the method we have designed to organize the text and determine lexicalization.

```
Alan_Shepard | mission | Apollo_14
Alan_Shepard | deathPlace | California
Alan_Shepard | birthPlace | New_Hampshire
Alan_Shepard | dateOfRetirement | "1974-08-01"
Apollo_14 | operator | NASA
Alan_Shepard | birthDate | "1923-11-18"
```

Alan Shepard was born on November 18, 1923 in New Hampshire and he was a crew member of Apollo 14 that is operated by NASA. He went into retirement on August 1, 1974 and passed away in California.

Table 1: The top part shows a triple set from `/dev/en/6triples/Astronaut.xml`. The bottom part shows the realized sentence produced by RDFJSREALB from this input.

3 Text Generation

To illustrate our NLG process, we use the set of triples shown in Table 1 with the corresponding generated English sentence.

The first step in text generation is determining the information to be conveyed in the text. In the context of WEBNLG CHALLENGE 2020, this is given: it is a set of at most 7 triples. As the predicate of a triple indicates a relation between its subject and object, in our case, it is mapped to a verb linking the subject and the object of the sentence realizing to this triple.

3.1 Microplanning

Triples being unordered, the first critical step is organizing them to build an *interesting story*. The triples are first grouped by their subject and the triples are sorted within their group. For example, to describe a person, the birth date and place could first be given, then some activities, finally the retirement and death; for a University or a company, first its creation date, then its activity. To achieve this ordering, we associate with each predicate a *priority* used for sorting the input triples. We also submitted for automatic scoring a version of the system that skipped this sorting process. Although the automatic scores were highly similar for both versions (see Section 4), we conjecture the sorting process is still useful to make the texts easier to follow.

The sorted groups are then processed in decreasing number of triples. We also query DBPedia (see Appendix A.1) to determine if the category of a group subject is the current one, if it is the case

```
Alan_Shepard
   birthDate "1923-11-18";
   birthPlace New_Hampshire;
   mission Apollo_14;
   dateOfRetirement "1974-08-01";
   deathPlace California.
Apollo_14
   operator NASA.
```

Table 2: `mtriples` from Table 1 sorted and grouped, shown as a Turtle-like formalism, used as input for RDFJSREALB. Predicates and objects sharing the same subject are shown indented and separated by semicolons.

then its score is increased so that the text will start with this subject similarly to what we saw in the lexicalizations in the training corpus. Each group forms a sentence as a coordination of *subsentences*. As a long coordinated sentence is often difficult to follow, groups of more than 3 triples are split into two sentences. In order to avoid very short sentences, groups with only one triple are combined using a subordinate when its subject is the object of another triple in a bigger group.

Table 2 shows the result of the sorting and grouping process on the example of Table 1. The five triples having `Alan_Shepard` as subject are grouped and sorted to form a coherent biography. This input will be used for realizing the two sentences shown in the bottom part of Table 1 using JSREALB.

3.2 Surface realization

For the final realization step, we use JSREALB (Molins and Lapalme, 2015), a surface realizer written in Javascript similar in principle to SIMPLENLG (Gatt and Reiter, 2009) in which programming language instructions create data structures corresponding to the constituents of the sentence to be produced. Once the data structure is built, it is traversed to produce the list of words in the sentence, dealing with conjugation, agreement, capitalization, all the *small* details that are important for easing the reading by the users and evaluators. Unfortunately, a large part of this hard work is not taken into account by the automatic evaluation process which often works with lowercased tokens.

For RDFJSREALB, we use Python to create the structure sent to a local JSREALB web server that returns the realized sentence. The data structure is built by calls to constructors whose names were chosen to be similar to the symbols typically used

```
S(Q("Alan_Shepard"),                  # quoted string
  VP(V("be").t("ps"), # auxiliary to the simple past
     V("born").t("pp"),     # verb to past participle
     PP(P("on"),              # prepositional phrase
        DT("1923-11-18").dOpt(...)),# date
     PP(P("in"),              # prepositional phrase
        Q("New_Hampshire"))))# another quoted string
```

Table 3: Top: Python/JavaScript functional notation for a JSREALB expression with comments at the right realized by JSREALB as: Alan Shepard was born on November 18, 1923 in New Hampshire.

for constituent syntax trees such as a *Terminal* (e.g. `N` (Noun), `V` (Verb), `A` (adjective), `D` (determiner), `Q` which quotes its parameter thus allowing *canned text*) or a *Phrase* (e.g. `S` (Sentence), `NP` (Noun Phrase), `VP` (Verb Phrase)).

Features added to the structures using the dot notation can modify their properties. For terminals, their person, number, gender can be specified. For phrases, the sentence may be negated or set to a passive mode; a noun phrase can be pronominalized, these features were not used here, but we use the automatic processing of coordinated phrases that insert appropriate commas and conjunction between coordinated elements. Table 3, shows the Python calls to create an internal structure that is serialized and sent to the JSREALB server to realize the English sentence.

3.3 Sentence Templates

The challenge is thus to transform the structure of Table 2 to the one in Table 3. We manually defined 200 templates associated with the most frequent predicates in the training set, those with 20 or more occurrences. A default template (described in Section 3.4) is used when no defined template is found which in less than 10% of cases in the training and development sets, but in 24% of cases in the test set.

A predicate `p` corresponds to a Python lambda expression whose parameter is the object `o`. The predicate is called to create a sentence with the subject `s`. The actual parameters are quoted strings of the subject or object of the triple, but replacing underscores by spaces with special cases for numbers and dates.

For example, given the two following Python definitions:

```
birthP = lambda o:VP(V("be").t("ps"),
                     V("born").t("pp"),
                     PP(P("in"),o))
sentence = lambda s,p,o: S(Q(s),
                           p(Q(o)))
```

```
"birthDate":(2,True,[
  lambda o:VP(V("be").t("ps"),
              V("born").t("pp"),PP(P("on"),o)),
]),
"birthPlace":(3,True,[
  lambda o:VP(V("be").t("ps"),
              V("born").t("pp"),PP(P("in"),o)),
]),
"birthYear":(2,True,"birthPlace"),
"dateOfRetirement":(90,True,[
  lambda o:VP(V("retire").t("ps"),PP(P("on"),o)),
  lambda o:VP(V("go").t("ps"),P("into"),
              N("retirement"),PP(P("on"),o)),
]),
"deathPlace":(100,True,[
  lambda o:VP(V("die").t("ps"),PP(P("in"),o)),
  lambda o:VP(V("pass").t("ps"),Adv("away"),
              PP(P("in"),o)),
]),
"mission":(22,True,[
  lambda o:VP(V("be").t("ps"),D("a"),
              N("crew"),N("member"),PP(P("of"),o))
                ,
  lambda o:VP(V("become").t("ps"),N("member"),
              PP(P("of"),o)),
]),
"operator":(51,False,[
  lambda o:VP(V("be"),V("operate").t("pp"),PP(P("
      by"),o)),
]),
```

Table 4: A few Python templates

the call

```
sentence("Alan␣Shepard",
         birthP,"New␣Hampshire")
```

creates the following structure:

```
S(Q("Alan␣Shepard"),
  VP(V("be").t("ps"),
     V("born").t("pp"),
     PP(P("in"),Q("New␣Hampshire"))))
```

which is verbalized as Alan Shepard was born in New Hampshire. by JSREALB. This is the basic mechanism for creating sentence structures that can be combined in various ways.

Templates are organized in a dictionary (see Table 4) having the name of the predicate as a key associated with a value which is a 3-tuple with the following elements: a priority (a number between 0 and 100) used for sorting, a boolean indicating if its subject can be a human and a list of lambda expressions that can verbalize this predicate, one of which is randomly chosen at the realization time.

Templates associated with predicates were developed by looking at `lex` elements in the training corpus. When two templates have the same realizations, the third element of the pair is the name of the original predicate (see `birthYear` in Table 4). Currently a template only depends on the name of the predicate, but it would be interesting to develop specialized templates according to the category of the set of triples; for example, the predicate `language` used for triples of the `WrittenWork` category should be verbalized as written in, but in categories `Artist` or a `Politician`, it could be speaks or sings in.

Once the above structure for the templates was settled after a few false starts, it became relatively easy to write them. Reading lexicalizations associated with a predicate, it takes less than minute to write a lambda defining a constituent expression to reproduce some of them. This is possible because we noticed that many lexicalizations are often very similar, having been created by crowdworkers who seemed to often rely on copy-pasting the subject and the object. We conjecture that it should be feasible to develop a learning algorithm to go from the lexicalizations to the lambdas, the final realization being left to JSREALB.

3.4 Default template

When an unknown predicate is encountered then a default template is created. By detecting case changes, the name of the predicate is split into *words*. and taken as subject of the be auxiliary, the object is used as attribute. For example,

```
elevationAboveTheSeaLevel =>
   Q("elevation␣above␣the␣sea␣level")
```

In the final sentence, the subject of the triple is taken as subject of the be auxiliary, the object of the triple is used as attribute. For example, the triple

```
Aarhus_Airport |
    elevationAboveTheSeaLevel | 25.0
```

is realized as Aarhus Airport elevation above the sea level is 25.0.

3.5 Text aggregation

In some cases, dealing with related information (e.g., birth date and place), combining templates using only their complements (i.e., their last element) will simplify the text. For this we define groups of predicates that can be combined at realization time. For example, `birthDate` and `birthPlace` in Table 4 or `numberOfStudents`, `academicStaffSize` and `numberOfPostgraduateStudents`. To this list are added, the *equivalent* templates (e.g. `birthYear` will be combined with `birthDate`). A similar process is used for combining objects sharing their subject and predicate.

When two or three triples are merged into a single sentence, the subject is used at the start but a pronoun is used for the following references. Currently, a very simple system is used for choosing the pronoun: if the predicate is coded as being applicable to a human and the gender of the subject

```
S(CP(C("and"),  # coordination
     S(Q("Alan␣Shepard"),
       VP(V("be").t("ps"),
          V("born").t("pp"),
          PP(P("on"), # combine objects
             DT("1923-11-18")),
          PP(P("in"),
             Q("New␣Hampshire")))),
     S(Pro("I").g("m"),
       VP(V("be").t("ps"),
          D("a"),
          N("crew"),
          N("member"),
          PP(P("of"),
             NP(Q("Apollo␣14"),
                SP(Pro("that"),
                   VP(V("be"),
                      V("operate").t("pp"),
                      PP(P("by"),
                         Q("NASA")))))))))
# Alan Shepard was born on November 18,
# 1923 in New Hampshire and he was a crew
# member of Apollo 14 that is operated by NASA.
```

```
S(CP(C("and"),
     S(Pro("I").g("m"),
       VP(V("retire").t("ps"),
          PP(P("on"),
             DT("1974-08-01")))),
     S(Q(""),
       VP(V("pass").t("ps"),
          Adv("away"),
          PP(P("in"),
             Q("California"))))))
# He retired on August 1, 1974 and
# passed away in California.
```

Table 5: Two indented structures produced by aRDFJSREALB Python program for the example of Table 1 annotated here with Python comments. Each sentence structure is followed by a comment showing the structure realization produced by JSREALB.

obtained by querying DBPedia is `male`, he is used, if it is `female` then she [4] is chosen, otherwise it is used. When a single triple whose subject is used as object of another, it is combined with the subordinate using a pronoun: who if the predicate applies to a human, otherwise that.

Table 5 shows the result of combining all these processes on the input of Table 2.

While during development, the default template was used less than 10% of the time, it was used 24% of the time during the test. There were 78 unseen templates for the tests of which 30 were used more than 10 times. It would have taken about 30 minutes to develop new templates for dealing with these new cases and improve the results. The competition rules did not allow for this type of modification, although it would probably be done in a production setting for improving the output.

[4]There are very few references to a woman in the training set. A rough estimate: she occurs 119 times in the `lex` elements, one of these being a ship, while he has 2,808 occurrences (24 times more). A classical case of gender-bias inferred from the data. This ratio is quite different in the test set in which there are 669 references to a female *against* 768 to a male. Counting occurrences in the same way in the training set, there are 8,855 males and 900 females, while in the development set there are 1086 males and 130 females.

4 Results

4.1 Automatic evaluation

During development, RDFJSREALB has been applied to the test set of WEBNLG CHALLENGE 2017 and to the dev and train set of WEBNLG CHALLENGE 2020. The system is very fast: it processes about 1 000 triple sets per second on Mac laptop; this is much faster than the evaluation which takes many seconds to process a single category with a given number of triples. Table 6 shows the scores on the WEBNLG CHALLENGE 2020 test set. Only values for the METEOR score are shown, the others being strongly correlated.

We submitted three versions of our system for the automatic scoring WEBNLG CHALLENGE 2020:

1. RDFJSREALB: as described in the preceding sections;
2. RDFJSREALB-unsorted: but skipping sorting the predicates that share a subject. We wanted to see the effect of ignoring the priorities which might be considered a bit artificial;
3. RDFJSREALB-baseline: a 10 line Python program shown in Appendix A.2 that creates a sentence by concatenating the subject, the words contained in the predicate split as we do for the default template (Section 3.4) and

System	All		Seen-Cat		Unseen-Cat		Unseen-Ent	
	MET	rank	MET	rank	MET	rank	MET	rank
2020-best	0.42	1	0.44	1	0.40	1	0.42	1
RDFJSREALB `id12`	0.39	11	0.39	23	0.38	12	0.40	12
RDFJSREALB-unsorted `id11`	0.38	12	0.39	25	0.37	14	0.40	11
2020-baseline	0.37	16	0.39	28	0.36	15	0.38	15
RDFJSREALB-baseline `id10`	0.33	27	0.33	34	0.33	19	0.32	23
2020-worst	0.22	35	0.33	35	0.13	35	0,30	35

Table 6: METEOR scores and rank among the 35 submissions to WEBNLG CHALLENGE 2020 for three versions of RDFJSREALB. We only report METEOR scores as other metrics seem quite correlated with them.

the object of a triple. These *sentences* are then concatenated to create the text.

RDFJSREALB is quite competitive: while being far from the best, it gives very good results, being in the first third. It is roughly at par with the competition baseline which is also a symbolic system. We see that the sorting process improves scores by a very small margin for automatic scoring. We are curious to see how it will influence the human scoring. As expected, our baseline performs badly, but surprisingly it is not the worst of all submissions, it is even in the middle of the pack for the unseen categories. We further discuss the interest of this submission in the next section.

4.2 Human Evaluation

As shown in Table 7, the output of RDFJSREALB (`RALI/id12` in the published results) was judged excellent (always in the first group of participants) for coverage, relevance and correctness. This is understandable as a great care is given to realize all information given the triples. This is true for almost all combination of seen and unseen domains. The text structure and fluency was judged less good (in the second and third group) which shows that the use of language model would probably be useful to improve the fluency of the output.

Categories	Cov	Rel	Cor	Str	Fl.
All	1	1	1	3	4
Seen domains	1	1	2	3	3
Unseen entities	1	1	1	2	2
Unseen domains	1	1	1	3	3

Table 7: Rank given to the output of RDFJSREALB by the human evaluation on five different aspects: coverage, relevance, correctness, text structure and fluency.

5 Comments on the task data

We are very thankful of the task organizers who spent an enormous amount of time and energy building this corpus and collecting human lexicalizations. We also recognize the fact that they explicitly say that their goal was to provide enough data so that it becomes feasible for learning algorithms to develop micro-planners.

We now take a step back to reflect on how this task corresponds to the original motivation: providing verbalizers for *real* RDF set of triples, a goal stated in almost all papers cited in Section 2. We think that WEBNLG CHALLENGE 2020 has so greatly simplified the task that many of the *interesting* problems have been more or less bypassed.

As the data already identifies the triples to verbalize, the challenging step of searching an RDF triple store to select the appropriate statements is short-circuited. The problem is thus greatly simplified, but there is still plenty of interesting work to do, so this is not a fundamental criticism, as this step could be performed by another system or could be the subject of another shared task.

An important problem in NLG is lexicalization, i.e., finding the appropriate words to use for an utterance. In WEBNLG CHALLENGE 2020, this problem is greatly simplified by the fact that the URIs for the subject and object have been replaced by their labels. So, in almost all cases, it is sufficient just to copy the subject and the object in the output like we do in RDFJSREALB. The only lexicalization problem left is thus finding an appropriate wording for the predicate.

To illustrate how the reference sentences are similar to content of the triples, we can look at the result of our baseline which merely copies the input triples with a slight formatting. For 1-Triples, their is a surprisingly high similarity (BLEU scores of

more than 30) which of course decreases as the number of triples increases. The BLEU score for the whole WEBNLG CHALLENGE 2020 test set is 0.19. This shows the input is probably too close to the expected output to be a real NLG challenge.

A commonly used convention for naming a predicate is to use a conjugated verb (Uschold, 2018, p. 187) (e.g. `works` or `speaks`) or a verbal locution with words joined using *camelCase* (e.g. `isPartOf` or `hasChild`). Unfortunately, very few of the predicates in WEBNLG CHALLENGE 2020 data follow this convention which is widely used in ontology works. For example, here are the most frequent predicates in the train corpus: `country`, `leader`, `location`, `birthPlace`, `isPartOf`, `club`, `ethnicGroup`, `language`, `genre`, `capital`.

There are also many *long-winded* names for predicates such as

```
addedToTheNationalRegisterOfHistoricPlaces
elevationAboveTheSeaLevelInMetres
wasGivenTheTechnicalCampusStatusBy
```

which should have been divided into two or three triples to separate the operation (`added` or `isElevated`) from the destination or the comparison and the units. A similar remark can be done for `hasToItsNorth`, `hasToItsWest`, `hasToItsSoutheast`... or `isbnNumber`, `issnNumber`, `LCCN_Number` (sic) or `oclcNumber` although this is the kind of regularity that a learning algorithm should be able to take advantage of.

One of the goals of this data compilation was to develop enough training data for the development of learning algorithms, so it would have been convenient to merge some predicates in order to have more data for each. Following some of our suggestions, the organizers had produced a revised version of the learning data, merging many similarly named predicates. But there are still interesting cases left: `address` and `location` or `mission` and `crewMembers` (its inverse) are different predicates that could have been normalized. The same remark applies to `order` which is the inverse of `class` or `division`.

Contrarily, some predicates should be split because they are used in different contexts: for example, `language` can be used for the language spoken by a person or the language in which a book is written. This would not have appeared if the usual naming convention for predicates had been followed.

Of course, predicates being arbitrary URI, their *name* is not in principle important, but a system developed without caring for basic conventions would perhaps not be very appealing for Semantic Web enthusiasts.

Another suggestion to better reflect NLG challenges would be to split complex objects such as in the following `mtriples`:

```
Arros_negre | mainIngredients | "White␣
    rice,␣cuttlefish␣or␣squid,␣
    cephalopod␣ink,␣cubanelle␣peppers"
Bacon_sandwich | alternativeName | "
    Bacon␣butty,␣bacon␣sarnie,␣rasher␣
    sandwich,␣bacon␣sanger,␣piece␣'n␣
    bacon,␣bacon␣cob,␣bacon␣barm,␣bacon␣
    muffin"
```

Each of these triples should be split into many each with an object containing a single ingredient or name. Ideally, it would be an RDF collection of single ingredients or names. Currently, simply copying the object probably artificially inflates the similarity scores since these enumerations have many words in common with the references.

The task is still far from trivial but, given these caveats, we think that developers should not extrapolate too much on how the performance of their system on the WEBNLG CHALLENGE 2020 dataset could be replicated in *real-life* RDF contexts. Moreover RDF triples are usually linked with an ontology whose content should be integrated into the realizer, a context that is not taken into account in WEBNLG CHALLENGE 2020.

6 Conclusion

This paper has described a symbolic approach to solve the shared task WEBNLG CHALLENGE 2020. The automatic scores are quite competitive compared to the ones of the other participants which, we conjecture, most often used machine learning approaches. The system is very fast on a standard laptop. It can also be quite easy to adapt to a new domain: considering about one minute per new predicate, it would have taken about 3 hours to develop 200 new predicates. It would be interesting if machine learning could be applied for developing new templates, even though we doubt it would be as fast.

Acknowledgments

We thank our colleagues Fabrizo Gotti and Philippe Langlais for constructive comments on a first version of this paper and the referees for their suggestions.

References

Guadalupe Aguado, A. Bañón, John Bateman, Maria Bernardos Galindo, Mariano Fernández-López, Asuncion Gomez-Perez, E Nieto, A. Olalla, R. Plaza, and A Sánchez. 1998. ONTOGENERATION: Reusing domain and linguistic ontologies for spanish text generation. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI'98)*, Brighton, UK.

John Bateman. 1997. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, 3(1):15–55.

John A. Bateman, Renate Henschel, and Fabio Rinaldi. 1995. Generalized Upper Model 2.0: documentation. Technical report, GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany.

Kalina Bontcheva and Yorick Wilks. 2004. Automatic report generation from ontologies: The MIAKT approach. In *Natural Language Processing and Information Systems*, pages 324–335, Berlin, Heidelberg. Springer Berlin Heidelberg.

Nadjet Bouayad-Agha, Gerard Casamayor, and Leo Wanner. 2014. Natural language generation in the context of the semantic web. *Semantic Web*, 5:493–513.

Thiago Castro-Ferreira, Claire Gardent, Nikolai Ilinykh, Chris van der Lee, Simon Mille, Diego Moussalem, and Anastasia Shimorina. 2020. The 2020 Bilingual, Bi-Directional WebNLG+ Shared Task: Overview and Evaluation Results (WebNLG+ 2020). In *Proceedings of the 3rd WebNLG Workshop on Natural Language Generation from the Semantic Web (WebNLG+ 2020)*, Dublin, Ireland (Virtual). Association for Computational Linguistics.

Ngan T. Dong and Lawrence B. Holder. 2014. Natural language generation from graphs. *International Journal of Semantic Computing*, 08(03):335–384.

Pablo Ariel Duboue and Kathleen R. McKeown. 2003. Statistical acquisition of content selection rules for natural language generation. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, pages 121–128.

Daniel Duma and Ewan Klein. 2013. Generating natural language from linked data: Unsupervised template extraction. In *Proceedings of the 10th International Conference on Computational Semantics (IWCS 2013) – Long Papers*, pages 83–94, Potsdam, Germany. Association for Computational Linguistics.

Basil Ell and Andreas Harth. 2014. A language-independent method for the extraction of RDF verbalization templates. In *Proceedings of the 8th International Natural Language Generation Conference (INLG)*, pages 26–34, Philadelphia, Pennsylvania, U.S.A. Association for Computational Linguistics.

Michel Gagnon and Lyne Da Sylva. 2006. Text compression by syntactic pruning. In *Proceedings of the 19th International Conference on Advances in Artificial Intelligence: Canadian Society for Computational Studies of Intelligence*, AI'06, pages 312–323, Berlin, Heidelberg. Springer-Verlag.

Dimitrios Galanis, George Karakatsiotis, Gerasimos Lampouras, and Ion Androutsopoulos. 2009. An open-source natural language generator for OWL ontologies and its use in Protégé and Second Life. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 17–20, Athens, Greece. Association for Computational Linguistics.

Claire Gardent, Anastasia Shimorina, Shashi Narayan, and Laura Perez-Beltrachini. 2017a. Creating training corpora for NLG micro-planners. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 179–188, Vancouver, Canada. Association for Computational Linguistics.

Claire Gardent, Anastasia Shimorina, Shashi Narayan, and Laura Perez-Beltrachini. 2017b. The WebNLG challenge: Generating text from RDF data. In *Proceedings of the 10th International Conference on Natural Language Generation*, pages 124–133, Santiago de Compostela, Spain. Association for Computational Linguistics.

Albert Gatt and Ehud Reiter. 2009. SimpleNLG: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation (ENLG 2009)*, pages 90–93, Athens, Greece. Association for Computational Linguistics.

Paul Molins and Guy Lapalme. 2015. JSrealB: A bilingual text realizer for web programming. In *Proceedings of the 15th European Workshop on Natural Language Generation (ENLG)*, pages 109–111, Brighton, UK. Association for Computational Linguistics.

Xiantang Sun and Christopher Stuart Mellish. 2006. Domain independent sentence generation from RDF representations for the Semantic Web. In *ECAI06 Combined Workshop on Language-Enabled Educational Technology and Development and Evaluation of Robust Spoken Dialogue Systems*.

Michael Uschold. 2018. Demystifying OWL for the enterprise. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 8(1):i–237.

Pavlos Vougiouklis, Hady Elsahar, Lucie-Aimée Kaffee, Christophe Gravier, Frédérique Laforest, Jonathon Hare, and Elena Simperl. 2018. Neural Wikipedian: Generating textual summaries from knowledge base triples. *Journal of Web Semantics*, 52-53:1 – 15.

Graham Wilcock and Kristiina Jokinen. 2003. Generating responses and explanations from RDF/XML and DAML+OIL. In *Knowledge and Reasoning in Practical Dialogue Systems*, pages 58–63. Volume: Proceeding volume:.

Yaoming Zhu, Juncheng Wan, Zhiming Zhou, Liheng Chen, Lin Qiu, Weinan Zhang, Xin Jiang, and Yong Yu. 2019. Triple-to-Text: Converting RDF triples into high-quality natural languages via optimizing an inverse KL divergence. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'19, pages 455–464, New York, NY, USA. Association for Computing Machinery.

A Appendices

A.1 External data

As the WEBNLG CHALLENGE 2020 competition allows external resources, we developed two functions to query the DBPEDIA SPARQL ENDPOINT[5] for :

Checking if a *subject* corresponds to a given *category* using :

```
ASK WHERE {
 <http://dbpedia.org/resource/subject>
 rdf:type
 <http://dbpedia.org/ontology/category>}
```

Getting the gender of a subject:

```
SELECT ?gender WHERE {
 <http://dbpedia.org/resource/subject>
 <http://xmlns.com/foaf/0.1/gender>
 ?gender}
```

The above SPARQL queries were sent to the DBPedia SPARQL endpoint using the Python interface `SPARQLWrapper`.

A.2 Baseline

```
def camel_case_split(s):
    return list(map(str.lower,
    re.findall(r'([A-Z0-9]+|[A-Z0-9]?'+
      '[a-z0-9]+)(?=[A-Z0-9]|\b)', s)))

def realizeTriple(triple):
    return cleanNode(triple.s)+"␣"+\
      "␣".join(map(str.lower,
          camel_case_split(triple.p)))\
      +"␣"+cleanNode(triple.o)

def cleanNode(n):
    return n.replace("_language","").\
      replace("_","␣").replace('"','')
```

[5] http://dbpedia.org/sparql