Docria: Processing and Storing Linguistic Data with Wikipedia

Marcus Klang
marcus.klang@cs.lth.se
Lund University
Department of Computer Science
S-221 00 Lund, Sweden

Pierre Nugues
pierre.nugues@cs.lth.se
Lund University
Department of Computer Science
S-221 00 Lund, Sweden

Abstract

The availability of user-generated content has increased significantly over time. Wikipedia is one example of a corpus, which spans a huge range of topics and is freely available. Storing and processing such corpora requires flexible document models as they may contain malicious or incorrect data. Docria is a library which attempts to address this issue with a model using typed property hypergraphs. Docria can be used with small to large corpora, from laptops using Python interactively in a Jupyter notebook to clusters running map-reduce frameworks with optimized compiled code. Docria is available as open-source code at `https://github.com/marcusklang/docria`.

1 Introduction

The availability of user-generated content has increased significantly over time. Wikipedia is one example of a corpus, which spans a huge range of topics and is freely available. User-generated content tests the robustness of most tools as it may contain malicious or incorrect data. In addition, data often comes with valuable metadata, which might be semi-structured and/or incomplete. These kinds of resources require a flexible and robust data model capable of representing a diverse set of generic and domain-specific linguistic structures.

In this paper, we describe a document model which tries to fill the gap between fully structured and verifiable data models and domain-specific data structures. This model, called Docria, aims at finding a tradeoff between the rigidity of the former and the specificity of the latter. To show its merits, we contrast the application of fully structured data models to practical noisy datasets with the simplicity of Docria.

2 Related Work

Linguistically annotated data have been stored in many different formats, often developed to solve practical problems. We can group prior work into three categories:

Formats – the technical formats which are used to serialize the data;

Document models – conceptual descriptions of how the data is connected, often mapped to concrete software implementations;

Applications and tooling – user-facing applications for annotation, search, etc.

in this section, we will focus on the low-level formats and libraries to parse and access the data contained within.

Pustylnikov et al. (2008), in their work on unifying 11 treebanks, made a summary of formats typically used, which shows a dominance of XML variants and CoNLL-like formats. We examine some of them here.

Tabular annotation. The tabular annotation in plain text is one of the simplest formats: One token per line and white space separation for the data fields connected to the token followed by a double line separation to mark a sentence. This kind of format was used first in the CoNLL99 task on chunking (Osborne, 1999) and then on subsequent tasks. Its main merits are the ease of use with regards to writing parsers and its readability without documentation.

Universal Dependencies (Nivre et al., 2019) is an example of a recent project for multilingual corpora using this format. It defines a variant called CoNLL-U, an adaption of the format used in CoNLL-X shared task on multilingual dependency

Proceedings of the 22nd Nordic Conference on Computational Linguistics (NoDaLiDa), pages 400–405
Turku, Finland, 30 September – 2 October, ©2019 Linköping University Electronic Press

parsing (Buchholz and Marsi, 2006). CoNLL-U includes field descriptions at the start of a document using hashtag (#) comments, adds subword support, and a field, if used, would allow for untokenization by including information about spacing between tokens.

CoNLL-* formats are tightly connected to data used in the shared tasks. Variations of these plain-text formats in the wild have no real standard and are mostly ad-hoc development. The field separation is a practical aspect, which may vary: spaces or tabulations. Depending on the corpus, these are not interchangeable as the token field might include ordinary spaces as part of the data field.

Semi-structured formats. Semi-structured formats specify stricter rules and a frequent choice is to follow the XML syntax to implement them (Bray et al., 2008). XML is hierarchical and can support higher-order structures such as sections, paragraphs, etc. XML has been used successfully in the development of the TIGER Corpus (TIGER XML) (Brants et al., 2002) and the Prague Dependency Treebank (PML) (Hajič et al., 2018).

The XML annotation relies on a schema defining its content on which programs and users must agree. Aside from TIGER XML and PML, the Text Encoding Initiative (TEI) and FoLiA XML (van Gompel and Reynaert, 2013) are general purpose XML schema definitions focused on linguistic and text annotation. TEI and FoLiA provide extensive documentation and guidelines on how data should be represented in XML.

Graph formats. From primarily hierarchical formats, the NLP Interchange Format (NIF) provides a graph-oriented way of connecting information which builds on existing standards such as LAF/GrAF, RFC 5147, and RDF. The main innovation in NIF is a standardized way of referring to text with offsets also known as a stand-off annotation. NIF is similar to WIKIPARQ (Klang and Nugues, 2016).

3 Docria

Docria is a document model based on typed property hypergraphs. We designed it to solve scalability and tooling problems we faced with the automatic processing and annotation of Wikipedia. This corresponds notably to:

- The lack of document models and storage solutions that could fit small and large corpora and that could be compatible with research practices;
- The impossibility to use the same document model with potentially costly large-scale extraction algorithms on a cluster with a map-reduce computing framework such as Apache Spark.

Motivation. These aspects were dominant in the construction of Docria, for which we set a list of requirements:

Openness – release the library as open source[1]; share processed corpora such as Wikipedia in formats used by this library; invite others to use the library for various tasks;

Scalability – from small corpora using a few lines of code to show a concept on a laptop to large-scale information extraction running on multiple computers in a cluster with optimized code;

Low barrier – progressive learning curve, sensible defaults, no major installations of services or configurations. Specifically, we wanted to reduce barriers when we shared larger corpora with students for use in project courses;

Flexibility – capable of representing a diverse set of linguistic structures, adding information and structures progressively, changing structure as needed;

Storage – reducing disk-space and bandwidth requirements when distributing larger corpora.

Design. To meet these goals, we implemented Docria in both Python and Java with a shared conceptual model and storage format. One of the user groups we had in mind in the design step was students in computer science carrying a course project. As our students have programming skills, we elected a programmer-first approach with a focus on common tasks and algorithms and a tooling through an API.

Python with Jupyter notebooks provides an interactive Read-Evaluate-Print-Loop (REPL) with rich presentation possibilities. We created extensions for it to reduce the need for external tooling and so that with a few lines of code, a programmer can inspect the contents of any Docria

[1]https://www.github.com/marcusklang/docria

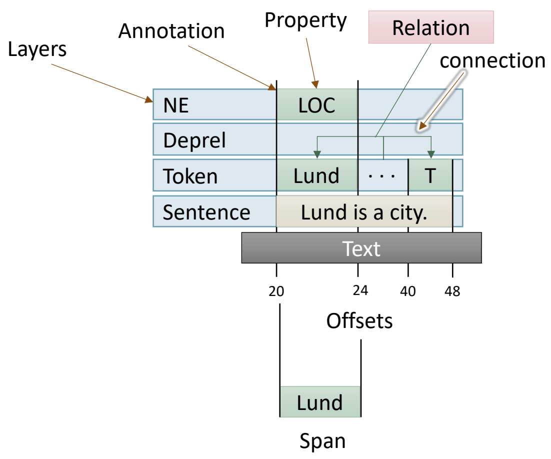

Figure 1: Docria data model

document. Through a matching implementation in Java, Docria provides a path to scale up when needed, as specific tasks can be orders of magnitudes faster than with a CPython implementation.

Docria documents consist of text collections and layers, shown in Figure 1. Text collections allow for multiple representations of a single text. A layer is a collection of nodes. These nodes can have fields which refer to the text collections. One particular restriction we impose is that a user must define a schema per layer. This is essential for introspection and verification of the data contained in documents. The schema defines the available fields and their data type with support for metadata.

Datatypes. The datatypes include basic types such as Boolean, integer, float, and string. Advanced types include text spans, node spans, node references, and node array references, which enable a programmer to represent graph structures. Field types, which are node references, must specify a target layer. In addition, this restriction results in well-defined dependencies between layers, which can be used in the future for partial document reconstruction when reading.

Using a relational database analogy, layers correspond to tables; they contain nodes which are equivalent to rows with fields, which are typed columns with specialized support for references to other nodes in other layers.

Stand-off references. Docria uses stand-off references in which we separate text from linguistic layers. These layers refer to ranges in the original text. To simplify the implementation and reduce sources of common bugs, the text string is split into pieces according to the offsets and stores text as a list of substrings, which is reconstructed without a loss by a join. Offsets, when serialized, only refer to spans of substrings. Software implementations can reconstruct offsets by computing the actual substring length and creating a lookup table. This will generate correct offsets even if the in-memory representation of a string differs, which is the case with standard strings in Java and Python 3.

Binary format. For the binary format, we selected MessagePack. MessagePack is self-describing, has an open well-defined specification, and has multiple open-source implementations in a diverse set of programming languages. The binary format can be used on a per document basis or in an included collection container, which writes multiple binary documents in sequence. This binary format was also designed to allow for a quicker content listing by separating content into compartments which can be read independently: document properties, schema, text, and layer data.

The Wikipedia corpus. We used the official REST API provided by Wikimedia and a page listing from the official dump page to collect the Wikipedia corpus. We downloaded all the pages in HTML format from this page listing in October 2018. This HTML format was processed and converted into a DOM using JSoup. Using recursive rules, we transformed the DOM into a flat text representation with structural layers referring to ranges such as section, paragraph, and anchors. Furthermore, we linked anchors to Wikidata by translating page targets to Q-numbers where available. We also retained formatting, such as bold and italics. We stored all this information using Docria.

In this dump, there are 5,405,075 pages excluding redirections.

4 Evaluation

We applied the spaCy library[2] to annotate all the English Wikipedia pages with parts of speech, entities, and dependency graphs, and we made the result available at `http://fileadmin.cs.lth.se/papers/nodalida2019/`. On average, each page of the corpus, after annotation,

[2]`https://spacy.io/`

contains 72.2 sentences, 901.8 tokens, 144.8 entities, and 4,383 characters.

We used this annotated corpus to evaluate the technical aspects of Docria and compare them to XML. We chose XML as it is pervasive in the literature and capable of representing all the structures present in Wikipedia.

We selected FoLiA as the XML format. FoLiA is well-defined, has good tooling, defines a diverse set of structural annotations which covers most, if not all, aspects of Wikipedia. FoLiA also has an official Python library, which we used to read documents.

Millions of XML files can be stored uncompressed in a file system. However, this often results in considerable overhead in terms of access times and reading and is therefore not practical for efficient processing. In addition, XML is verbose and contains redundant information. All this makes compression and streaming a necessity when storing and processing millions of documents.

To compare FoLiA XML with Docria, we chose to use a sequential tarball format with a bzip2 compression. We chose this format as it provided the most similar way to store documents in sequence applicable to both FoLiA XML and Docria. We created one XML file per article in-memory and saved them in a sequence using the tarfile API of Python. The structures we included for the comparison were section, paragraph, entities, tokens with their part of speech and lemma, and dependency relations.

5 Benchmark

We stored the Wikipedia corpus in 432 parts, containing on average 12,512 pages per part. Due to time constraints, the metrics below are computed using only 64 of the 432 parts.

First, we measured the difference in size when compressed: FoLiA XML files are on average 2.47 times larger than the matching Docria files. The compressed Docria parts have a mean size of 85.0 MB[3] compared to 209.8 MB for the compressed FoLiA XML parts. This translates to a compressed size of 6.8 kB resp. 16.8 kB on average per page.

Secondly, we measured the cost of decompressing the files in memory. Reading a single bzip2 Docria compressed file without any processing and a 1 MB buffer requires, on an Intel Xeon at 3.40 GHz, 16.3 sec ± 18.9 ms compared to 104 seconds ± 136 ms to read FoLiA XML, both averaged over 7 runs. Reading compressed FoLiA XML over binary Docria tar-files is on average 6.4 times slower.

[3] 1 MB = 1,000,000 bytes

Uncompressed Folia XML documents are on average 9.5 times larger per document with a mean size of a page of 314.5 kB vs. 32.1 kB for Docria. For comparison, the mean average size of raw UTF-8 encoded text is of 4.4 kB per page. Put another way, using the plain text as starting point, Docria has an annotation overhead of 7.6 times vs. 69.6 times for XML.

6 Programming Examples

In this section, we show programs for three basic operations:

1. Create a new document and add a token with part-of-speech annotation.

2. Read a sequential tarball and print all the tokens of all the sentences of the corpus;

3. Read a sequential tarball and extract the entities of type person.

Create a document and add a part of speech. We first create a document from a string and we add a token layer. We then add a node to this layer, spanning the 0..4 range and we annotate it with a part of speech using the `add()` method as this:

```
# Initial include
from docria import Document, \
     DataTypes as T

# Create a document
doc = Document()

# Add main text
doc.maintext = "Lund University"

# Create a token layer with two fields
doc.add_layer("token",
      pos=T.string, text=T.span)

# The token layer, when displayed
# in a Jupyter notebook, will be
# rendered as a HTML table.
tokens = doc["token"]

# Adding a token node
# referencing range 0:4
token = tokens.add(
  pos="PROPN",
  text=doc.maintext[0:4]
)
```

Print the tokens. We assume we have a tarball of documents segmented into sentences and tokens, and annotated with the parts of speech. We read the tarball with `TarMsgpackReader` and we access and print the sentences, tokens, and parts of speech using the Python dictionary syntax.

```
from docria.storage \
    import TarMsgpackReader

with TarMsgpackReader(
    "enwiki00001.tar.bz2",
    mode="r|bz2") as reader:
  for rawdoc in reader:
    # Materialize document
    doc = rawdoc.document()

    # Lists all layers with field
    # types and metadata
    doc.printschema()

    # Print the original text
    # Equivalent to doc.text["main"]
    print(doc.maintext)

    for sentence in doc["sentence"]:
      # Print the full sentence
      print(sentence["tokens"].text())

      for tok in sent["tokens"]:
         # Form <TAB> part-of-speech
         print("%s\t%s" %
                (tok["text"], tok["pos"]))
```

Extract entities of a certain type. We assume here that the tarball is annotated with entities stored in an `ENTITY` layer. We read the tarball and access the entities. We then extract all the entities of category `PERSON`:

```
with TarMsgpackReader(
    "enwiki00001.tar.bz2",
    mode="r|bz2") as reader:
  for rawdoc in reader:
    # Materialize document
    doc = rawdoc.document()

    # Get the entity layer
    entities = doc["entity"]

    # Filter out PERSON in entity
    # layer having field label
    # equal to PERSON
    query = (entities["label"]
             == "PERSON")

    for person in entities[query]:
      # Tokens represents potentially
      # many tokens, text()
      # transforming it to a string
      # from the leftmost
      # to the rightmost token.
      print(person["tokens"].text())
```

7 Discussion

When converting the Wikipedia corpora to fit the FoLiA XML format, we had issues identifying a suitable span annotation for the Wikipedia anchor link. We decided to associate it with the FoLiA XML entity type.

In addition, when using stand-off annotations, some documents did not pass validation with offset errors, possibly due to normalization issues common to Wikipedia text. This gives an argument that these kinds of formats do not work reliably with noisy datasets. We instead included the sentences as text and used the nospace attribute to allow untokenization, which does increase verbosity slightly.

Initially, we used the official foliapy library, but we were unable to get a decent performance with it, potentially addressed in the future. We resorted to using the LXML DOM matching example documents with Folia. To ensure correctness, we verified samples of our XMLs using foliavalidator.

Acknowledgments

This research was supported by Vetenskapsrådet, the Swedish research council, under the *Det digitaliserade samhället* program, grant number 340-2012-5738.

References

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The tiger treebank. In *Proceedings of the workshop on treebanks and linguistic theories*, volume 168.

Tim Bray, Eve Maler, François Yergeau, Michael Sperberg-McQueen, and Jean Paoli. 2008. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C. Http://www.w3.org/TR/2008/REC-xml-20081126/.

Sabine Buchholz and Erwin Marsi. 2006. Conll-x shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X '06, pages 149–164, Stroudsburg, PA, USA. Association for Computational Linguistics.

Maarten van Gompel and Martin Reynaert. 2013. Folia: A practical xml format for linguistic annotation — a descriptive and comparative study. *Computational Linguistics in the Netherlands Journal*, 3:63–81.

Jan Hajič, Eduard Bejček, Alevtina Bémová, Eva Buráňová, Eva Hajičová, Jiří Havelka, Petr Homola, Jiří Kárník, Václava Kettnerová, Natalia Klyueva, Veronika Kolářová, Lucie Kučová, Markéta Lopatková, Marie Mikulová, Jiří Mírovský, Anna Nedoluzhko, Petr Pajas, Jarmila Panevová, Lucie Poláková, Magdaléna Rysová, Petr Sgall, Johanka Spoustová, Pavel Straňák, Pavlína Synková, Magda Ševčíková, Jan Štěpánek, Zdeňka Urešová, Barbora Vidová Hladká, Daniel Zeman, Šárka Zikánová, and Zdeněk Žabokrtský. 2018. Prague dependency treebank 3.5. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Marcus Klang and Pierre Nugues. 2016. WIKIPARQ: A tabulated Wikipedia resource using the Parquet format. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2016)*, pages 4141–4148, Portorož, Slovenia.

Joakim Nivre, Mitchell Abrams, Željko Agić, Lars Ahrenberg, Gabrielė Aleksandravičiūtė, et al. 2019. Universal dependencies 2.4. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

M. Osborne. 1999. *CoNLL-99. Computational Natural Language Learning. Proceedings of a Workshop Sponsored by The Association for Computational Linguistics*. Association for Computational Linguistics (ACL).

Olga Pustylnikov, Alexander Mehler, and Rüdiger Gleim. 2008. A unified database of dependency treebanks: Integrating, quantifying & evaluating dependency data. In *LREC 2008*.