Published in: Bański, Piotr/Kupietz, Marc/Lüngen, Harald/Rayson, Paul/Biber, Hanno/Breiteneder, Evelyn/Clematide, Simon/Mariani, John/stevenson, Mark/Sick, Theresa (eds.): Proceedings of the Workshop on Challenges in the Management of Large Corpora and Big Data and Natural Language Processing (CMLC-5+BigNLP) 2017 including the papers from the Web-as-Corpus (WAC-XI) guest section. Birmingham, 24 July 2017. - Mannheim: Institut für Deutsche Sprache, 2017. Pp. 35-41

Keeping Properties with the Data CL-MetaHeaders - An Open Specification

John Vidler
j.vidler@lancaster.ac.uk

Stephen Wattam
steve@watt.am

School of Computing and Communications
Lancaster University

Abstract

Corpus researchers, along with many other disciplines in science are being put under continual pressure to show accountability and reproducibility in their work. This is unsurprisingly difficult when the researcher is faced with a wide array of methods and tools through which to do their work; simply tracking the operations done can be problematic, especially when toolchains are often configured by the developers, but left largely as a black box to the user. Here we present a scheme for encoding this 'meta data' inside the corpus files themselves in a structured data format, along with a proof-of-concept tool to record the operations performed on a file.

1 Introduction

Corpora are continually increasing in size, and as a side effect, the management of this data during processing continues to be a pressing issue. As researchers we are under pressure to be able to reproduce the findings that we publish, and as the data we use increases in magnitude this can quickly become an onerous task. As noted in "*A Note on Rigour and Replicability*" (Louridas and Gousios, 2012), "*Unfolding off-the-shelf IR systems for reproducibility*" (Di Buccio et al., 2015) and "*Who Wrote the Web? Revisiting Influential Author Identification Research Applicable to Information Retrieval*" (Potthast et al., 2016), tracking the transforms performed on an input set is difficult at the best of times with the best of intentions, but when confronted with an unfamiliar tool or tool chain, inexperienced users can be forgiven for accidentally performing operations that do not achieve what they intend.

On the one hand, as a discipline, difficulties arising from the existence of a broad tool suite are an excellent problem to have; a multitude of solutions are present to handle any number of problems we may face. However, on the other hand, having no standard interoperability level for these tools prevents some interactions across this space without a great deal of effort from the user. Being aware of the file type is seldom enough to correctly set up reader/writer operations in a tool, especially if we consider character encoding differences, and the overall engineering of the tool has become very important in some cases. Further, processes for management of text metadata are tightly bound to the format, resulting in the need for bespoke tooling just to manage text flow through an NLP toolchain. An extensive overview of the issues has been compiled in "*A critical look at software tools in corpus linguistics*" (Anthony, 2013).

Here we propose a format for metadata storage that is sufficiently concise and flexible to be included *within* existing formats, allowing for metadata storage and processing in a manner minimally coupled with the underlying text storage regime.

Our approach draws upon the design principles of UNIX' 'magic numbers', most commonly encountered in BASH (Project, 2017) scripts. In shell scripts we see the "hash-bang" prefixed first line of each script which identifies the interpreter that should be used to read the file.

Such a hashbang line is a special case of a code comment as delimited by a single hash symbol `#`, normally reserved to be used to have the interpreter simply ignore the rest of a line. In the special "hash-bang" case, the first line of the script includes as `#!` with a system path string indicating which binary should be invoked to

execute the script. A common example of this is BASH interpreted scripts using `#!/bin/bash`.

Without this included in the header of the file, implementation differences between shell interpreters would quickly render scripts useless as the behaviour would be undefined. In older loaders without the initial "hash-bang" recognising capability, the loader can skip over the line as normal as the line starts with a comment symbol, effectively rendering it invisible, and preserving legacy behaviour.

We can borrow from this design for our own implementations, providing a mechanism for metadata storage and file type disambiguation that is flexible and simple to parse. Herein, we address the high-level design concerns surrounding such a format (Section 2) and the container format selected (Section 3) before going on to detail the mandatory fields designed to aid processing tools (Section 4). This paper is intended to introduce a preliminary specification, and further steps towards standardisation and use are discussed in Section 6 prior to an appendix containing examples for common NLP file formats.

2 Design

The primary aims of this specification are to provide a mechanism for detailing text- and toolchain-related metadata, and providing additional information on existing files that may affect subsequent processing stages.

Describing text metadata is a task already competently handled by formats such as TEI, and is primarily concerned with a rich, structured storage format that can be mined for information algorithmically. The need for such structure must be balanced here with the need for *compatibility* — the ability for the data to be 'hidden' from other NLP tools within their comment fields — and *universality* — the need for data to be applicable to many different types of toolchain.

In line with other bottom-up approaches, here we follow the design of assembling a minimal set of smaller, optional, features into a common data format. Following the lead of other minimal formats (such as vCards (Dawson and Howes, 1998) and JSON Schema[1]), we take a structured approach that assembles these components into a key-value map.

This approach permits the variation of representations for even the most basic data formats, providing they cover a basic subset. Here we require only 4 data types in accordance with the JSON specification(ECMA, 2013)[2], namely:

String Defined as the ASCII-7 subset only for reasons for character set compatibility;

Decimal Numbers Base-10, ASCII representation;

Nil Analogous to the empty set, `null`, `None` etc.

Boolean A single bit of information, i.e. `true` or `false`

These basic data types may be composed into data structures of two types: *objects*, which are key-value stores with any basic type as the key and another as the value, and *arrays*: objects with implicit, ordered integer keys. Keys are to be specified using `snake_case`. Keys starting and ending with two underscores '`__key__`' are reserved, using a convention similar to Python's PEP8(van Rossum et al., 2001).

With these as building blocks, we can construct structures which encompass text metadata, such as author, source, or date information, in addition to the possibility of describing sequences of operations - such as the tool history that has been performed to generate the file. In order to make the latter of these applicable to many toolchains, we specify a subset of structures that may be used to fulfil certain roles: this is intended to standardise and simplify tooling.

2.1 Namespaces and Interfaces

The basic metaheader structure is a single key-value map (*Object*) containing a set of keys defined by the current version of the specification (these are detailed later in Section 4). Each key within this top-level object points to a value, which may itself be a simple type or another Object. A field containing an object at the top level is said to provide a *namespace*.

Namespaces are specified separately to the core set of fields, and allow information to be grouped by purpose or administrator—this is similar to the approach taken by many packaging systems such as RubyGems[3] and PyPI[4]. Namespaces may be

[1] `http://json-schema.org/`

[2] And, it should be noted, many other data formats such as messagepack, BSON and YAML

[3] `https://rubygems.org/`

[4] `https://pypi.python.org/pypi`

defined by third parties in order to provide tool-specific key-value pairings — this mechanism is intended to allow tool authors to store information in formats that remain compatible with other tooling, without resorting to standoff annotation.

Because decentralised definition may lead to these namespaces becoming difficult to integrate and process, we reserve the capability to define a set of *interfaces*. Interfaces define a set of keys that must be provided together within a namespace, in order to offer a given service. This is analogous to duck typing in object-oriented languages: a namespace providing all of the fields required is presumed to behave according to the interface specification.

We reserve keys beginning and ending with two underscores ('`__key__`') for this purpose, for example, any namespace wishing to implement semantic versioning may provide a `__version__` field containing a semantic versioning compatible string. Any tooling wishing to support versioning of namespaces may then detect this and process such namespaces consistently.

3 Container Format

As opposed to simpler formats such as shell scripts, which need only know a single parameter to be able to select the correct interpreter at start-up, we can leverage modern structured data formats to embed effectively any amount of data into the header of a file. Many modern configuration tools use formats such as YAML(Evans, 2011) and JSON(ECMA, 2013), to provide rich options for holding data. YAML is commonly used, for example, as a metadata header section for static site generators (MkDocs, Jekyll and Hugo).

We should note that our intention is to specify the data structure contained within any such format, rather than the format itself, and propose that a number of formats may be implemented if necessary to remain within comment fields of other files.

The examples listed here, and the tooling that accompanies this paper, use JSON as a container format.

JSON was selected due to the breadth of its software support (according to `json.org` JSON is supported by 63 programming and scripting languages), and ease/speed of parsing. These properties make it suitable for use in many NLP contexts. Additionally, its simple data model and whitespace-agnostic form make it particularly suitable for representation within the comment fields of many NLP text formats.

JSON's popularity has lead to the existence of binary equivalent formats already available such as BSON(Group, 2015) and MessagePack(Furuhashi, 2013) for cases where data storage is in binary form.

3.1 Representation

As interoperability is key, the representation of data within the format is unspecified as far as possible. This means that the format may be included in a header or standoff documentation whilst remaining logic-compatible with all processing and aggregation tools. Further, inclusion in existing data formats is possible by simply adding the format to the comments.

Such conventions allow for multiple possible paths of information flow through NLP toolchains (an example of which is depicted in Figure 1): tooling that supports comments may retain the metaheaders in-place, or headers may be explicitly stripped out and processed in between text processing stages.

This approach mirrors that of the UNIX pipeline philosophy — small scripting tools may form the 'glue' around NLP toolchain components by reading headers and directing data as appropriate. This processing may be sufficiently generic to be handled by off-the-shelf tools (for example, the compilation of an audit trail including timestamps and processing arguments), or a custom processing stage using the metaheaders as a storage format.

This approach means that, for the first time, toolchain management code would be transferable throughout the community and between resources. In turn this enables the creation of interoperable tooling for documenting processing stages, aiding replicability.

4 Data Structure & Current Specification

Here we present an overview of the draft field specification designed to provide a minimal subset of fields to aid basic parsing and processing. Draft version '1.0.2' of the specification mandates no fields, but has 4

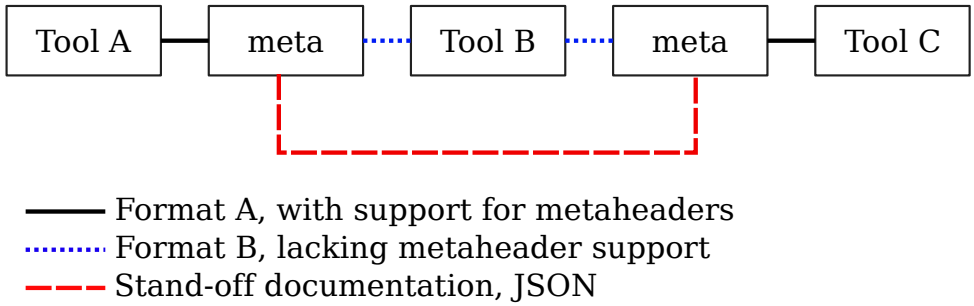

Figure 1: A sample workflow using the 'meta' tool to transfer annotations through a format that does not support commenting.

optional ones, some with default values, as described as follows:

`__version__` *(Optional, String)* - The specification version that this header complies with, if missing, assumes the latest draft. Version numbers are specified following Semantic Versioning (Preston-Werner,), making it easy to determine compatibility.

`encoding` *(Optional, String)* - Character encoding of current text, as defined by the IANA list of preferred text encoding names(Freed and Dürst, 2013). While optional, it is strongly advised that this field be present to avoid any ambiguity in parsing. If absent, this implicitly defaults to UTF-8.

`mime` *(Optional, String)* - The extended MIME(IANA, 2015) type used to describe what this file is.

`group` *(Optional, String or Object)* - Used to track which files belong to collections, defined as an object conforming to the `group` namespace specification.

`history` *(Optional, Object)* - A top-level namespace conforming to the `history` namespace specification that describes the processing history of this file. Complies with the interface specification and thus has an inner `__version__` field.

For the purposes of definition we draw the distinction between a *field*, where a key is assigned a simple value, and *namespace*, holding an object containing fields that themselves conform to a sub-format. Optional namespaces may then be assembled whilst retaining some guarantee of compatibility.

In the case of the above, `group` is specified as a namespace, which may contain:

`text_id` *(Required, String)* - A unique text identifier assigned to this text

`*` *(Optional, any)* - Further arbitrary key-value fields appropriate to the corpus

One simple example of where the non-string form of the `group` field could be used is that of parallel corpora; inner fields specifying a `collection` and `language` (See Figure 2) could be used to form the following structure to completely identify a part of the larger corpus.

```
...
"group": {
    "collection": "OPUS",
    "language": "en-gb"
},
...
```

Figure 2: An example of the `group` field being used as an object to identify this file as being part of the OPUS(`http://opus.lingfil.uu.se/`) UK English set.

The `history` namespace is intended to describe the history of a particular text's processing to form an audit trail of actions in the form of a list of actions. It is specified as:

`binary` *(Required, String)* - The program executed.

`time` *(Required, String)* - ISO8601 format datetime string describing the date and time at which the tool was run.

`args` *(Optional, String)* - The program arguments used.

platform *(Optional, String)* - The dot-delimited platform and architecture (ex. "Linux.x64").

md5 *(Optional, String)* - The md5 hash of the binary, used to ensure the correct version is used.

The set of features identified for inclusion in the first draft have been selected to allow the identification of key features of the subsequent texts, and allow them to be correctly loaded by software. They are common to all machine-readable text representations, describe necessary-yet-uninteresting features of the dataset, and are generally useful across many tool types.

These fields provide a level of process-accounting that so far has been absent from many NLP toolchains, and allows us to replay the processing that created the files in use.

In addition to the namespaces above, we define only a single interface designed to offer version reporting on third-party namespace specifications. Note that all fields are implicitly required for an interface to apply:

__**version**__ *(String)* - A semantic versioning compliant version string describing the version of the namespace that is being used.

4.1 Tooling

In addition to the format specification here, a proof-of-concept tool was developed, '`meta`' (source available at `http://ucrel.github.io/CL-metaheaders/`) which wraps existing commands and records their use in the files they generate, and can be used to validate existing meta headers in source files. Figure 3 shows how this command can be used to wrap existing tools to record their actions.

```
$> meta tagger \
    --input corpus.xml \
    --output result.xml
```

Figure 3: Running the '`meta`' tool on the program '`tagger`' with some arguments included. This would record the command in the `history` block inside `output.xml` metadata. Backslashes indicate a continued line.

Once the tool has been used to produce this history in the headers, the same tool can be used to extract the commands for later execution by the user.

The obvious use case for this is in cases where the user may have forgotten the precise commands they used, but is also useful for a second user to process other files in the same way as the first user, especially as part of a validation process.

Other included features of this tool are the ability to initialise a file with the basic metadata fields in a user-friendly way, and generate a list of dependencies for a given file. By reading the metadata of the file and getting the command history we can walk the list looking for any files that are required to generate the output given.

Furthermore, this can recurse through any recognised file that *also* has metaheaders included to create a full dependency tree which in turn can be used as part of the packaging process when files are to be distributed. This should aid researchers in producing correct source packages for distribution.

4.2 Extensions and Custom Namespaces

Further specifications and versions will be maintained and released in an open-source manner via the project's website at `http://ucrel.github.io/CL-metaheaders/`.

In addition to the specification (and tooling) provided here, we have designed the namespacing system to allow for other developers and tools to insert arbitrary data below the top level data structure. Proprietary fields are expected to use nested namespaces to keep the top-level clean, and allow developers the freedom to add their own variants for their own purposes - we do not expect to be able to predict all use cases for these headers.

Figure 4 demonstrates a TEI file with an additional 'software' sub-object to contain developer specific information (See Appendix 6 for further examples). The use of 'per-tool' namespaces in this manner allows for the use of standard file formats by various tools without loss of information that otherwise would have to be discarded after execution (or output in a difficult-to-track and proprietary standoff annotation format).

Because the software reading the files cannot know about all possible extensions to this format, we mandate that tools supporting the metadata specification must pass all unknown headers from

```
<?xml version="1.0"
   encoding="UTF-8"
   standalone="no" ?>
<TEI
  xmlns="http://www.tei-c.org/ns/1.0">
<!-- meta {
   "version": 1.0,
   "encoding": "UTF-8",
   "mime": "text/xml-tei",
   "software": {
      "author": "Joe Bloggs",
      "tool": "Jo's Awesome Software",
      "window": "+-5 words",
      "stoplist": true
   }
} -->
<teiHeader>
<fileDesc>
  ...
```

Figure 4: JSON data including an additional software description fields. Note that this is still version '1.0' compliant, as there is no restriction on additional data in the meta header. Newlines presented here are for the benefit of the reader, and can be entirely omitted for a single-line meta entry.

input to output without modification. They are, of course, free to change the fields and namespaces that account for any change applied to the text.

5 Further Work

Stated in Section 4, the standard is intended to provide only the barest minimum set to enable better communication between tools, and we fully expect to extend the format with additional data as future tools develop.

What we define here forms a 'core' field set, forming a number of reserved keys and associated namespace definitions. It is the authors' intent to allow developers the freedom to extend the standard with their own additions and as such we welcome any comments, suggestions regarding these extensions for inclusion in later standard releases.

One organisational addition is the creation of a registry of top-level namespaces. This will eliminate any potential issues with collision of third-party namespace definitions leading to incompatible implementations by offering a single versioned canonical list of namespace allocations.

In addition, we expect to produce further specification details for including meta data with archived corpus files, providing a mechanism for creating hierarchies of files.

Furthermore, as a living body of work, we intend to continue to integrate more document formats in to the standard as 'officially recognised types' and provide further examples of integration.

6 Summary

We have presented a simple method for including the properties of a file along with the file itself in a way that is backwards compatible with many existing text storage formats and tools. The basic design of this method is extensible in order to allow tool authors to annotate files, and to allow those building toolchains to use such data to manage existing tools. Using these capabilities, we present a proof-of-concept toolchain auditing application.

The specification outlined here is being actively developed, and a canonical reference (along with proof-of-concept and production tooling) are available at `http://ucrel.github.io/CL-metaheaders/`. Continued discussion of the specification, including bugs and feature requests can be done via the Github issues page at `https://github.com/UCREL/CL-metaheaders/issues`.

References

Laurence Anthony. 2013. A critical look at software tools in corpus linguistics. August.

F Dawson and T Howes. 1998. RFC 2426 - vCard MIME directory profile. `https://www.ietf.org/rfc/rfc2426.txt`, September.

Emanuele Di Buccio, Giorgio Maria Di Nunzio, Nicola Ferro, Donna Harman, Maria Maistro, and Gianmaria Silvello. 2015. Unfolding off-the-shelf IR systems for reproducibility. In *Proc. SIGIR Workshop on Reproducibility, Inexplicability, and Generalizability of Results (RIGOR 2015)*.

ECMA. 2013. Ecma-404 - the json data interchange format. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`, October.

Clark C. Evans. 2011. Yaml - yaml ain't markup language. http://yaml.org/, November.

Ned Freed and Martin Dürst. 2013. Iana, character sets. http://www.iana.org/assignments/character-sets/character-sets.xhtml, 12.

Sadayuki Furuhashi. 2013. Messagepack - it's like json, but fast and small. http://msgpack.org.

BSON Group. 2015. Bson - binary json. http://bsonspec.org/.

IANA. 2015. Media types. http://www.iana.org/assignments/media-types/media-types.xhtml, 10.

Jinho Choi, Universal Dependencies contributors. 2014. CoNLL-U Format. http://universaldependencies.org/format.html. Online; accessed June 2017.

Panos Louridas and Georgios Gousios. 2012. A note on rigour and replicability. *SIGSOFT Softw. Eng. Notes*, 37(5):1–4, September.

Martin Potthast, Sarah Braun, Tolga Buz, Fabian Duffhauss, Florian Friedrich, Jörg Marvin Gülzow, Jakob Köhler, Winfried Lötzsch, Fabian Müller, Maike Elisa Müller, Robert Paßmann, Bernhard Reinke, Lucas Rettenmeier, Thomas Rometsch, Timo Sommer, Michael Träger, Sebastian Wilhelm, Benno Stein, Efstathios Stamatatos, and Matthias Hagen, 2016. *Who Wrote the Web? Revisiting Influential Author Identification Research Applicable to Information Retrieval*, pages 393–407. Springer International Publishing, Cham.

Tom Preston-Werner. Semantic versioning 2.0.0. http://semver.org/.

The GNU Project. 2017. Bash - the gnu bourne-again shell. https://tiswww.case.edu/php/chet/bash/bashtop.html.

Guido van Rossum, Barry Warsaw, and Nick Coghlan. 2001. Pep 8: Style guide for python code. https://www.python.org/dev/peps/pep-0008/#id50.

A Further Examples For Common Formats

A.1 ARFF

```
% {"version":"1.0","encoding":"utf-8",\
  "mime":"application/x-weka"}
% 1. Title: Iris Plants Database
%
% 2. Sources:
%      (a) Creator: R.A. Fisher
%      (b) Donor: Michael Marshall
%      (c) Date: July, 1988
%
@RELATION iris

    @ATTRIBUTE sepallength   NUMERIC
    @ATTRIBUTE sepalwidth    NUMERIC
    @ATTRIBUTE petallength   NUMERIC
    @ATTRIBUTE petalwidth    NUMERIC
    @ATTRIBUTE class
{Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA ...
```

A.2 CoNLL-U

Note that implementations dealing with the CoNLL-U format are required to pass the contents of comments through their processing pipelines unaltered(Jinho Choi, Universal Dependencies contributors, 2014).

```
# {"version": "1.0","encoding":"utf-8",\
  "mime":"text/csv"}
# sent_id = 1
# text = Sue likes coffee and Bill \
  likes books
1       Sue         Sue
2       likes       like
3       coffee      coffee
4       and         and
5       Bill        Bill
...
```