Command-line utilities for managing and exploring annotated corpora

Joel Nothman and **Tim Dawborn** and **James R. Curran**
ə-lab, School of Information Technologies
University of Sydney
NSW 2006, Australia
{joel.nothman,tim.dawborn,james.r.curran}@sydney.edu.au

Abstract

Users of annotated corpora frequently perform basic operations such as inspecting the available annotations, filtering documents, formatting data, and aggregating basic statistics over a corpus. While these may be easily performed over flat text files with stream-processing UNIX tools, similar tools for structured annotation require custom design. Dawborn and Curran (2014) have developed a declarative description and storage for structured annotation, on top of which we have built generic command-line utilities. We describe the most useful utilities – some for quick data exploration, others for high-level corpus management – with reference to comparable UNIX utilities. We suggest that such tools are universally valuable for working with structured corpora; in turn, their utility promotes common storage and distribution formats for annotated text.

1 Introduction

Annotated corpora are a mainstay of language technology, but are often stored or transmitted in a variety of representations. This lack of standardisation means that data is often manipulated in an *ad hoc* manner, and custom software may be needed for even basic exploration of the data, which creates a bottleneck in the engineer or researcher's workflow. Using a consistent storage representation avoids this problem, since generic utilities for rapid, high-level data manipulation can be developed and reused.

Annotated text processing frameworks such as GATE (Cunningham et al., 2002) and UIMA (Lally et al., 2009) provide a means of implementing and combining processors over collections of annotated documents, for which each framework defines a serialisation format. Developers using these frameworks are aided by utilities for basic tasks such as searching among annotated documents, profiling processing costs, and generic processing like splitting each document into many. Such utilities provide a means of quality assurance and corpus management, as well as enabling rapid prototyping of complex processors.

The present work describes a suite of command-line utilities – summarised in Table 1 – designed for similar ends in a recently released document representation and processing framework, DOCREP (Dawborn and Curran, 2014). DOCREP represents annotation layers in a binary, streaming format, such that process pipelining can be achieved through UNIX pipes. The utilities have been developed organically as needed over the past two years, and are akin to UNIX utilities (`grep`, `head`, `wc`, etc.) which instead operate over flat file formats. The framework and utilities are free and open source (MIT Licence) and are available at `https://github.com/schwa-lab/libschwa`.

Some of our tools are comparable to utilities in UIMA and GATE, while others are novel. A number of our tools exploit the evaluation of user-supplied Python functions over each document, providing great expressiveness while avoiding engineering overhead when exploring data or prototyping.

Our utilities make DOCREP a good choice for UNIX-style developers who would prefer a quick scripting language over an IDE, but such modalities should also be on offer in other frameworks. We believe that a number of these utilities are applicable across frameworks and would be valuable to researchers and engineers working with manually and automatically annotated corpora. Moreover, we argue, the availability of tools for rapid corpus management and exploration is an important factor in encouraging users to adopt common document representation and processing frameworks.

Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT, pages 60–65,
Dublin, Ireland, August 23rd 2014.

2 Utilities for managing structured data

Data-oriented computing tends to couple data primitives with a declarative language or generic tools that operate over those primitives. UNIX provides tools for operating over paths, processes, streams and textual data. Among them are `wc` to count lines and words, and `grep` to extract passages matched by a regular expression; piped together, such utilities accomplish diverse tasks with minimal development cost. Windows PowerShell extends these notions to structured .NET objects (Oakley, 2006); Yahoo! Pipes (Yahoo!, 2007) provides equivalent operations over RSS feeds; SQL transforms relational data; and XSLT (Clark, 1999) or XQuery (Chamberlin, 2002) make XML more than mere markup.

Textual data with multiple layers of structured annotation, and processors over these, are primitives of natural language processing. Such nested and networked structures are not well represented as flat text files, limiting the utility of familiar UNIX utilities. By standardising formats for these primitives, and providing means to operate over them, frameworks such as GATE and UIMA promise savings in development and data management costs. These frameworks store annotated corpora with XML, so users may exploit standard infrastructure (e.g. XQuery) for basic transformation and aggregation over the data. Generic XML tools are limited in their ability to exploit the semantics of a particular XML language, such that expressing queries over annotations (which include pointers, spatial relations, etc.) can be cumbersome. LT-XML (Thompson et al., 1997) implements annotators using standard XML tools, while Rehm et al. (2008) present extensions to an XQuery implementation specialised to annotated text.

Beyond generic XML transformation, UIMA and GATE and their users provide utilities with broad application for data inspection and management, ultimately leading to quality assurance and rapid development within those frameworks. Both GATE and UIMA provide sophisticated graphical tools for viewing and modifying annotations; for comparing parallel annotations; and for displaying a concordance of contexts for a term across a document collection (Cunningham et al., 2002; Lally et al., 2009). Both also provide means of profiling the efficiency of processing pipelines. The Eclipse IDE serves as a platform for tool delivery and is comparable to the UNIX command-line, albeit more graphical, while providing further opportunities for integration. For example, UIMA employs Java Logical Structures to yield corpus inspection within the Eclipse debugger (Lally et al., 2009).

Generic processors in these frameworks include those for combining or splitting documents, or copying annotations from one document to another. The community has further built tools to export corpora to familiar query environments, such as a relational database or Lucene search engine (Hahn et al., 2008). The uimaFIT library (Ogren and Bethard, 2009) simplifies the creation and deployment of UIMA processors, but to produce and execute a processor for mere data exploration still has some overhead.

Other related work includes utilities for querying or editing treebanks (e.g. Kloosterman, 2009), among specialised annotation formats; and for working with binary-encoded structured data such as Protocol Buffers (e.g. protostuff[1]). Like these tools and those within UIMA and GATE, the utilities presented in this work reduce development effort, with an orientation towards data management and evaluation of arbitrary functions from the command-line.

3 Common utility structure

For users familiar with UNIX tools to process textual streams, implementing similar tools for working with structured DOCREP files seemed natural. Core utilities are developed in C++, while others are able to exploit the expressiveness of interpreted evaluation of Python. We describe a number of the tools according to their application in the next section; here we first outline common features of their design.

Invocation and API Command-line invocation of utilities is managed by a dispatcher program, `dr`, whose operation may be familiar from the `git` versioning tool: an invocation of `dr` *`cmd`* delegates to a command named `dr-`*`cmd`* where found on the user's `PATH`. Together with utility development APIs in both C++ and Python, this makes it easy to extend the base set of commands.[2]

[1] `https://code.google.com/p/protostuff/`

[2] Note that DOCREP processing is not limited in general to these languages. As detailed in Dawborn and Curran (2014) APIs are currently available in C++, Java and Python.

Command	Description	Required input	Output	Similar in UNIX
`dr count`	Aggregate	stream or many	tabular	`wc`
`dr dump`	View raw annotations	stream	JSON-like	`less` / `hexdump`
`dr format`	Excerpt	stream + expression	text	`printf` / `awk`
`dr grep`	Select documents	stream + expression	stream	`grep`
`dr head`	Select prefix	stream	stream	`head`
`dr sample`	Random documents	stream + proportion	stream	`shuf -n`
`dr shell`	Interactive exploration	stream + commands	mixed	`python`
`dr sort`	Reorder documents	stream + expression	stream	`sort`
`dr split`	Partition	stream + expression	files	`split`
`dr tail`	Select suffix	stream	stream	`tail`

Table 1: Some useful commands and comparable UNIX tools, including required input and output types.

Streaming I/O As shown in Table 1, most of our DOCREP utility commands take a single stream of documents as input (defaulting to `stdin`), and will generally output either plain text or another DOCREP stream (to `stdout` by default). This parallels the UNIX workflow, and intends to exploit its constructs such as pipes, together with their familiarity to a UNIX user. This paradigm harnesses a fundamental design decision in DOCREP: the utilisation of a document *stream*, rather than storing a corpus across multiple files.

Self-description for inspection Generic command-line tools require access to the *schema* as well as the data of an annotated corpus. DOCREP includes a description of the data schema along with the data itself, making such tools possible with minimal user input. Thus by reading from a stream, fields and annotation layers can be referenced by name, and pointers across annotation layers can be dereferenced.[3]

Extensions to this basic schema may also be useful. For many tools, a Python class (on file) can be referenced that provides *decorations* over the document: in-memory fields that are not transmitted on the stream, but are derived at runtime. For example, a document with pointers from dependent to governor may be decorated with a list of dependents on each governor. Arbitrary Python accessor functions (*descriptors*) may similarly be added. Still, for many purposes, the self-description is sufficient.

Custom expression evaluation A few of our utilities rely on the ability to evaluate a function given a document. The suite currently supports evaluating such functions in Python, providing great flexibility and power.[4] Their input is an object representing the document, and its offset (0 for the first document in a stream, 1 for the second, etc.). Its output depends on the purpose of the expression, which may be for displaying, filtering, splitting or sorting a corpus, depending on the utility in use, of which examples appear below. Often it is convenient to specify an anonymous function on the command-line, a simple Python expression such as `len(doc.tokens) > 1000`, into which local variables `doc` (document object) and `ind` (offset index) are injected as well as built-in names like `len`. (Python code is typeset in blue.) In some cases, the user may want to predefine a library of such functions in a Python module, and may then specify the path to that function on the command-line instead of an expression.

4 Working with DOCREP on the command-line

Having described their shared structure, this section presents examples of the utilities available for working with DOCREP streams. We consider three broad application areas: quality assurance, managing corpora, and more specialised operations.

[3] This should be compared to UIMA's Type System Descriptions, and uimaFIT's automatic detection thereof.

[4] `dr grep` was recently ported to C++ with a custom recursive descent parser and evaluator, which limits expressiveness but promises faster evaluation. In terms of efficiency, we note that some of the Python utilities have performed more than twice as fast using the JIT compilation of PyPy, rather than the standard CPython interpreter.

4.1 Debugging and quality assurance

Validating the input and output of a process is an essential part of pipeline development in terms of quality assurance and as part of a debugging methodology. Basic quality assurance may require viewing the raw content contained on a stream: schema, data, or both. This could ensure, for instance, that a user has received the correct version of a corpus from a correspondent, or that a particular field was used as expected. Since DOCREP centres on a binary wire format, `dr dump` (cf. UNIX's `hexdump`) provides a decoded textual view of the raw content on a stream. Optionally, it can provide minimal interpretation of schema semantics to improve readability (e.g. labelling fields by name rather than number), or can show schema details to the exclusion of data.

For an aggregate summary of the contents of a stream, `dr count` is a versatile tool. It mirrors UNIX's `wc` in providing the basic statistics over a stream (or multiple files) at different granularities. Without any arguments, `dr count` outputs the total number of documents on standard input, but the number of annotations in each store (total number of tokens, sentences, named entities, etc.) can be printed with flag `-a`, or specific stores with `-s`. The same tool can produce per-document (as distinct from per-stream) statistics with `-e`, allowing for quick detection of anomalies, such as an empty store where annotations were expected. `dr count` also doubles as a progress meter, where its input is the output of a concurrent corpus processor, as in the following example:

```
my-proc < /path/to/input | tee /path/to/output | dr count -tacv 1000
```

Here, the options `-acv`*n* output cumulative totals over all stores every *n* documents, while `-t` prefixes each row of output with a timestamp.

Problems can often be identified from only a small sample of documents. `dr head` (cf. UNIX's `head`) extracts a specified number of documents from the beginning of a stream, defaulting to 1. `dr sample` provides a stochastic alternative, employing reservoir sampling (Vitter, 1985) to efficiently draw a specified fraction of the entire stream. Its output can be piped to processing software for smoke testing, for instance. Such tools are obviously useful for a binary format; yet it is not trivial to split on document boundaries even for simple text representations like the CONLL shared task format.

4.2 Corpus management

Corpora often need to be restructured: they may be heterogeneous, or need to be divided or sampled from, such as when apportioning documents to manual annotators. In other cases, corpora or annotations from many sources should be combined.

As with the UNIX utility of the same name, `dr grep` has many uses. Here it might be used to extract a particular document, or to remove problematic documents. The user provides a function that evaluates to a Boolean value; where true, an input document is reproduced on the output stream. Thus it might extract a particular document by its identifier, all documents with a minimum number of words, or those referring to a particular entity. Note, however, that like its namesake, it performs a linear search, while a non-streaming data structure could provide fast indexed access; each traversed document is (at least partially) deserialised, adding to the computational overhead.[5] `dr grep` is often piped into `dr count`, to print the number of documents (or sub-document annotations) in a subcorpus.

`dr split` moves beyond such binary filtering. Like UNIX's `split`, it partitions a file into multiple, using a templated filename. In `dr split`, an arbitrary function may determine the particular output paths, such as to split a corpus whose documents have one or more category label into a separate file for each category label. Thus `dr split -t /path/to/{key}.dr py 'doc.categories'` evaluates each document's `categories` field via a Python expression, and for each key in the returned list will write to a path built from that key. In a more common usage, `dr split k 10` will assign documents by round robin to files named `fold000.dr` through `fold009.dr`, which is useful to derive a *k*-fold cross-validation strategy for machine learning. In order to stratify a particular field across

[5] Two commands that are not featured in this paper may allow for faster access: `dr subset` extends upon `dr head` to extract any number of documents with minimal deserialisation overhead, given their offset indices in a stream. `dr offsets` outputs the byte offset b of each document in a stream, such that C's `fseek(`b`)` or UNIX's `tail -c+`$(b+1)$ can be used to skip to a particular document. An index may thus be compiled in conjunction with `dr format` described below. Making fast random access more user friendly is among future work.

the partitions for cross-validation, it is sufficient to first sort the corpus by that field using `dr split k`. This is one motivation for `dr sort`, which may similarly accept a Python expression as the sort key, e.g. `dr sort py doc.label`. As a special case, `dr sort random` will shuffle the input documents, which may be useful before manual annotation or order-sensitive machine learning algorithms.

The inverse of the partitioning performed by `dr split` is to concatenate multiple streams. Given DOCREP's streaming design, UNIX's `cat` suffices; for other corpus representations, a specialised tool may be necessary to merge corpora.

While the above management tools partition over documents, one may also operate on portions of the annotation on each document. Deleting annotation layers, merging annotations from different streams (cf. UNIX's `cut` and `paste`), or renaming fields would thus be useful operations, but are not currently available as a DOCREP command-line utility. Related tasks may be more diagnostic, such as identifying annotation layers that consume undue space on disk; `dr count --bytes` shows the number of bytes consumed by each store (cf. UNIX's `du`), rather than its cardinality.

4.3 Exploration and transformation

Other tools allow for more arbitrary querying of a corpus, such as summarising each document. `dr format` facilitates this by printing a string evaluated for each document. The tool could be used to extract a concordance, or enumerate features for machine learning. The following would print each document's `id` field and its first thirty tokens, given a stream on standard input:

```
dr format py '"{}\t{}".format(doc.id, " ".join(tok.norm for tok in
                                                doc.tokens[:30]))'
```

We have also experimented with specialised tools for more particular, but common, formats, such as the tabular format employed in CoNLL shared tasks. `dr conll` makes assumptions about the schema to print one token per line with sentences separated by a blank line, and documents by a specified delimiter. Additional fields of each token (e.g. part of speech), or fields derived from annotations over tokens (e.g. IOB-encoded named entity recognition tags) can be added as additional columns using command-line flags. However, the specification of such details on the command-line becomes verbose and may not easily express all required fields, such that developing an *ad hoc* script to undertake this transformation may often prove a more maintainable solution.

Our most versatile utility makes it easy to explore or modify a corpus in an interactive Python shell. This functionality is inspired by server development frameworks (such as Django) that provide a shell specially populated with data accessors and other application-specific objects. `dr shell` reads documents from an input stream and provides a Python iterator over them named `docs`. If an output path is specified on the command-line, a function `write_doc` is also provided, which serialises its argument to disk. The user would otherwise have the overhead of opening input and output streams and initialising the de/serialisation process; the time saved is small but very useful in practice, since it makes interactive corpus exploration cheap. This in turn lowers costs when developing complex processors, as interactive exploration may validate a particular technique. The following shows an interactive session in which the user prints the 100 lemmas with highest document frequency.

```
$ dr shell /path/to/input.dr
>>> from collections import Counter
>>> df = Counter()
>>> for doc in docs:
...     doc_lemmas = {tok.lemma for tok in doc.tokens}
...     df.update(doc_lemmas)
...
>>> for lemma, count in df.most_common(100):
...     print("{:5d}\t{}".format(count, lemma))
```

Finally, `dr shell -c` can execute arbitrary Python code specified on the command-line, rather than interactively. This enables rapid development of *ad hoc* tools employing the common read-process-write paradigm (whether writing serialised documents or plain text), in the vein of `awk` or `sed`.

5 Discussion

DOCREP's streaming model allows the reuse of existing UNIX components such as `cat`, `tee` and pipes. This is similar to the way in which choosing an XML data representation means users can exploit standard XML tools. The specialised tools described above are designed to mirror the functionality if not names of familiar UNIX counterparts, making it simple for UNIX users to adopt the tool suite.

No doubt, many users of Java/XML/Eclipse find command-line tools unappealing, just as a "UNIX hacker" might be put off by monolithic graphical interfaces and unfamiliar XML processing tools. Ideally a framework should appeal to a broad user base; providing tools in many modalities may increase user adoption of a document processing framework, without which it may seem cumbersome and confining.

In this vein, a substantial area for future work within DOCREP is to provide more graphical tools, and utilities such as concordancing or database export that are popular within other frameworks. Further utilities might remove existing fields or layers from annotations; select sub-documents; set attributes on the basis of evaluated expressions; merge annotations; or compare annotations.

We have described utilities developed in response to a new document representation and processing framework, DOCREP. Similar suites deserve attention as an essential adjunct to representation frameworks, and should encompass expressive tools and various paradigms of user interaction. When performing basic corpus manipulation, these utilities save substantial user effort, particularly by adopting a familiar interface. Such boons highlight the benefit of using a consistent annotated corpus representation.

References

Donald D. Chamberlin. 2002. XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615.

James Clark. 1999. XSL transformations (XSLT). W3C Recommendation, 16 November.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 168–175.

Tim Dawborn and James R. Curran. 2014. docrep: A lightweight and efficient document representation framework. In *Proceedings of the 25th International Conference on Computational Linguistics*.

Udo Hahn, Ekaterina Buyko, Rico Landefeld, Matthias Mühlhausen, Michael Poprat, Katrin Tomanek, and Joachim Wermter. 2008. An overview of JCoRe, the JULIE lab UIMA component repository. In *Proceedings of LREC'08 Workshop 'Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP'*, pages 1–7.

Geert Kloosterman. 2009. *An overview of the Alpino Treebank tools*. `http://odur.let.rug.nl/vannoord/alp/Alpino/TreebankTools.html`. Last updated 19 December.

Adam Lally, Karin Verspoor, and Eoric Nyberg. 2009. *Unstructured Information Management Architecture (UIMA) Version 1.0*. OASIS Standard, 2 March.

Andy Oakley. 2006. *Monad (AKA PowerShell): Introducing the MSH Command Shell and Language*. O'Reilly Media.

Philip Ogren and Steven Bethard. 2009. Building test suites for UIMA components. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*, pages 1–4.

Georg Rehm, Richard Eckart, Christian Chiarcos, and Johannes Dellert. 2008. Ontology-based XQuery'ing of XML-encoded language resources on multiple annotation layers. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*.

Henry S. Thompson, Richard Tobin, David McKelvie, and Chris Brew. 1997. LT XML: Software API and toolkit for XML processing. `http://www.ltg.ed.ac.uk/software/`.

Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57.

Yahoo! 2007. Yahoo! Pipes. `http://pipes.yahoo.com/pipes/`. Launched 7 February.