docrep: A lightweight and efficient document representation framework

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

Abstract

Modelling linguistic phenomena requires highly structured and complex data representations. Document representation frameworks (DRFs) provide an interface to store and retrieve multiple annotation layers over a document. Researchers face a difficult choice: using a heavy-weight DRF or implement a custom DRF. The cost is substantial, either learning a new complex system, or continually adding features to a home-grown system that risks overrunning its original scope.

We introduce DOCREP, a lightweight and efficient DRF, and compare it against existing DRFs. We discuss our design goals and implementations in C++, Python, and Java. We transform the OntoNotes 5 corpus using DOCREP and UIMA, providing a quantitative comparison, as well as discussing modelling trade-offs. We conclude with qualitative feedback from researchers who have used DOCREP for their own projects. Ultimately, we hope DOCREP is useful for the busy researcher who wants the benefits of a DRF, but has better things to do than to write one.

1 Introduction

Computational Linguistics (CL) is increasingly a data-driven research discipline with researchers using diverse collections of large-scale corpora (Parker et al., 2011). Representing linguistic phenomena can require modelling intricate data structures, both flat and hierarchical, layered over the original text; e.g. tokens, sentences, parts-of-speech, named entities, coreference relations, and trees. The scale and complexity of the data demands efficient representations. A document representation framework (DRF) should support the creation, storage, and retrieval of different annotation layers over collections of heterogeneous documents. DRFs typically store their annotations as stand-off annotations, treating the source document as immutable and annotations "stand-off" with offsets back into the document.

Researchers may choose to use a heavy-weight DRF, for example GATE (Cunningham et al., 2002) or UIMA (Götz and Suhre, 2004), but this can require substantial investment to learn and apply the framework. Alternatively, researchers may "roll-their-own" framework for a particular project. While this is not inherently bad, our experience is that the scope of such smaller DRFs often creeps, without the benefits of the features and stability present in mature DRFs. Moreover, some DRFs are based on object serialisation, restricting the user to a specific language. In sum, while DRFs provide substantial benefits, they can come at an opportunity cost to valuable research time.

DOCREP aims to solve this problem by proving a light-weight DRF that does not get in the way. Using a language-agnostic storage layer enables reuse across different tasks in whatever tools and programming languages are most appropriate. Efficiency is our primary goal, and we emphasise compact serialisation and lazy loading. Our streaming design is informed by the pipeline operation of UNIX commands.

Section 2 compares existing DRFs and annotation schemes. We describe and introduce DOCREP in Section 3, outlining the design goals and the problems it aims to solve. We compare DOCREP to UIMA through a case study in Section 4, converting OntoNotes to both DRFs. Section 5 discusses real world uses of DOCREP within our research group and outlines experiences of its use by NLP researchers. DOCREP

Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers, pages 762–771, Dublin, Ireland, August 23-29 2014.

will be useful for any researcher who wants rapid development with multi-layered annotation that performs well at scale, but at minimal technical cost.

2 Background

Easily and efficiently storing and retrieving linguistic annotations over corpora is a core issue for data-driven linguistics. A number of attempts to formalise linguistic annotation formats have emerged over the years, including Annotation Graphs (AG) (Bird and Liberman, 1999), the Linguistic Annotation Format (LAF) (Ide and Romary, 2004, 2006), and more recently, the Graph Annotation Framework (GRAF) (Ide and Suderman, 2007). GRAF is a serialisation of the LAF model, using XML stand-off annotations to store layers of annotation. The GRAF representation is sufficiently abstract as to be used as a pivot format between other annotation schemes. Ide and Suderman (2009) use GRAF as an intermediate format to convert annotations between GATE and UIMA. The MASC corpus (Ide et al., 2010) has multiple layers of annotation which are distributed in GRAF. Neumann et al. (2013) provide insight into the effectiveness of GRAF as a format for corpus distribution when they import MASC into an annotation database. These linguistic annotation formalisations provide a useful set of requirements for DRFs. While these abstract formalisations are constructive from a theoretical perspective, they do not take into account the runtime performance of abstract representations, nor their ease of use for programmers.

Several DRFs have been developed and used within the CL community. GATE (Cunningham et al., 2002; Cunningham, 2002) has a focus on the human annotation of textual documents. While it has a large collection of extensions and plugins, it was not designed in a matter than suits web-scale corpus processing. Additionally, GATE is limited to Java, making integration with CL tools written in other languages difficult. UIMA (Götz and Suhre, 2004; Lally et al., 2008) is a Java framework for providing annotations over the abstract definition of documents, providing functionality to link between different views of the same document (e.g. translations of a document). UIMA calls these different views different "subjects of analysis" (SOFA). When UIMA was adopted into the Apache Software Foundation, a C++ version of the UIMA API was developed. However, it appears to lag behind behind the Java API in development effort and usefulness, with many undocumented components, numerous external dependencies, and with substantial missing functionality provided by the Java API. Additionally, the C++ API is written in an non-idiomatic manner, making it harder for developers to use.

Publicly available CL pipelining tools have emerged in recent years, providing a way to perform a wide range of CL processes over documents. The Stanford NLP pipeline[1] is one such example, but is Java only and must be run on a single machine. CURATOR (Clarke et al., 2012) provides a cross-language NLP pipeline using Thrift to provide cross-language communication and RPC. CURATOR requires a server to coordinate the components within the pipeline. Using pipelining functionality within a framework often the inspection of per-component contributions more difficult. We are not aware of any DRFs which use a streaming model to utilise UNIX pipelines, a paradigm CL researchers are already familiar with.

3 The docrep document representation framework

DOCREP (/dɒkrɛp/), a portmanteau of *doc*ument *rep*resentation, is a lightweight, efficient, and modern document representation framework for NLP systems that is designed to be simple to use and intuitive to work with. We use the term lightweight to compare it to the existing document representation systems used within the CL community, the main one being UIMA. The overhead of using DOCREP instead of a flat-file format is minimal, especially in comparison to large bulky frameworks.

Our research group has used DOCREP as its primary data storage format in both research projects and commercial projects since mid-2012. DOCREP has undergone an iterative design process during this time as limitations and issues arose, allowing modelling issues to be ironed out and a set of best practices to be established. These two years of solid use by CL researchers has resulted in a easy to use DRF we believe is suitable for most CL applications and researchers.

DOCREP was designed with streaming in mind, facilitating from the data storage layer upwards the ability for CL applications to utilise parallel processing. This streaming model is a model that many

[1] `http://nlp.stanford.edu/software/corenlp.shtml`

CL researchers are already familiar with from writing UNIX pipelines (Church, 1994; Brew and Moens, 2002), again reducing the overhead required to use DOCREP.

DOCREP is not a new language that researchers need to learn. Instead, it is a serialisation protocol and set of APIs to interact with annotations and documents. Using DOCREP is as simple as importing the package in ones favourite programming language and annotating class definitions appropriately. Neither a separate compilation step nor an external annotation definition file are required.

3.1 Idiomatic APIs

One of the motivations for constructing DOCREP was the lack of a good document representation framework in programming languages other than Java. We have implemented DOCREP APIs in three commonly used programming languages in the CL community: C++, Python, and Java. All of these APIs are open source and publicly available on GitHub,[2] released under the MIT licence. The C++ API is written in C++11, the Python API supports version 2.7 as well as versions $\geq$ 3.3, and the Java API supports versions $\geq$ 6. All three APIs are setup to use the standard build tools for the language.

When implementing these APIs, we aimed to make the interface as similar as possible between the three languages, while still feeling idiomatic within that language. Using the API should feel natural for that language. Figure 1 shows an example set of identical model definitions in C++, Python, and Java. This example defines a `Token` type, a `Sent` type spanning over a series of sequential `Token` annotations, and a `Doc` type. The `Token` and `Sent` types include some annotation attributes. Annotation instances are stored on the document in `Stores`. Apart from the missing implementations of the `Schema` constructors in the C++ example, these are complete and runnable definitions of annotation types in DOCREP. The `Schema` classes in the C++ example are automatically induced via runtime class introspection in the Python and Java APIs; functionality which C++ does not possess.

3.2 Serialisation protocol

We chose to reuse an existing serialisation format for DOCREP. This allows developers to use existing serialisation libraries for processing DOCREP streams in languages we do not provide a DOCREP API for.

One of our design considerations when creating DOCREP was a desire for the protocol to be self-describing. With a self-describing protocol, no external files need to be associated with a serialised stream in order to know how to interpret the serialised data. This requires an efficient serialisation protocol because including the definition of the type system with each document comes at a cost. This is different to UIMA which requires its XML type definition files in order to deserialise the serialised data.

The four main competitors in the web-scale binary serialisation format space are BSON,[3] MessagePack,[4] Protocol Buffers,[5] and Thrift.[6] BSON and MessagePack are similar in their design. They both aim to provide a general purpose data serialisation format for common data types and data structures. BSON is used as the primary data representation within the MongoDB database. Protocol Buffers and Thrift work in a similar manner to one another. Their serialisation protocols are not self describing and require an external file which defines how to interpret the messages on the stream. In this external file, users define the structure of the messages they wish to serialise and deserialise, and use a provided tool to convert this external file into source code for their programming language of choice. Protocol Buffers and Thrift also provide RPC functionality, however this was not needed for our situation. Thrift is used by the CURATOR NLP pipeline (Clarke et al., 2012) to provide both serialisation and RPC functionality between cross-language disjoint components in the pipeline.

After designing the serialisation protocol for DOCREP, we implemented it on top of these binary serialisation formats in order to compare the size of the serialised data and the speed at which it could be compressed. As a simple stand-off annotation task, we chose to use the CoNLL 2003 NER shared task

[2] `https://github.com/schwa-lab/libschwa`
[3] `http://bsonspec.org/`
[4] `http://msgpack.org/`
[5] `http://code.google.com/p/protobuf/`
[6] `http://thrift.apache.org/`

```
struct Token : public dr::Ann {
  dr::Slice<uint64_t> span;
  std::string raw;
  std::string norm;
  class Schema;
};

struct Sent : public dr::Ann {
  dr::Slice<Token *> span;
  bool is_headline;
  class Schema;
};

struct Doc : public dr::Doc {
  dr::Store<Token> tokens;
  dr::Store<Sent> sents;
  class Schema;
};
```

```
struct Token::Schema : public dr::Ann::Schema<Token> {
  DR_FIELD(&Token::span) span;
  DR_FIELD(&Token::raw) raw;
  DR_FIELD(&Token::norm) norm;
  Schema(void);
};

struct Sent::Schema : public dr::Ann::Schema<Sent> {
  DR_POINTER(&Sent::span, &Doc::tokens) tokens;
  DR_FIELD(&Sent::is_headline) is_headline;
  Schema(void);
};

struct Doc::Schema : public dr::Doc::Schema<Doc> {
  DR_STORE(&Doc::tokens) tokens;
  DR_STORE(&Doc::sents) sents;
  Schema(void);
};
```

(a) C++ example

```
class Token(dr.Ann):
  span = dr.Slice()
  raw = dr.Text()
  norm = dr.Text()

class Sent(dr.Ann):
  span = dr.Slice(Token)
  is_headline = dr.Field()

class Doc(dr.Doc):
  tokens = dr.Store(Token)
  sents = dr.Store(Sent)
```

(b) Python example

```
@dr.Ann
public class Token extends AbstractAnn {
  @dr.Field public ByteSlice span;
  @dr.Field public String raw;
  @dr.Field public String norm;
}

@dr.Ann
public class Sent extends AbstractAnn {
  @dr.Pointer public Slice<Token> span;
  @dr.Field public bool isHeadline;
}

@dr.Doc
public class Doc extends AbstractDoc {
  @dr.Store public Store<Token> tokens;
  @dr.Store public Store<Sent> sents;
}
```

(c) Java example

Figure 1: Examples of identical type definitions using the DOCREP API in C++, Python, and Java.

	Self-describing	Uncompressed		DEFLATE		Snappy		LZMA	
		Time	Size	Time	Size	Time	Size	Time	Size
Original data	–	–	31.30	1.0	5.95	0.1	9.81	39	0.39
BSON	✓	2.5	188.42	5.3	30.32	0.6	56.36	441	16.22
MessagePack	✓	1.6	52.15	3.2	16.61	0.3	24.82	61	4.36
Protocol Buffers	×	1.4	51.51	3.5	18.52	0.3	29.31	67	5.13
Thrift	×	1.0	126.12	3.5	20.64	0.4	33.69	224	10.99

Table 1: A comparison of binary serialisation libraries being used as the DOCREP serialisation format. Times are reported in seconds and sizes in MB. MessagePack and BSON include the full type system definition on the stream for each document whereas Protocol Buffers and Thrift do not.

data, randomly sampling around 50 MB worth of sentences from the English training data. The serialisation stores the documents, sentences, and tokens, along with the POS and NER tags for the tokens. The appropriate message specification files were written for Protocol Buffers and Thrift, and the type system was serialised as a header for BSON and MessagePack.

Table 1 shows the results of this experiment. The reported size of the original data is smaller than the sample size as we chose to output it in a more concise textual representation than the data was originally distributed in. BSON performs noticeably worse than the others, in terms of both size and speed. While serialising slightly faster, the size of the serialised data produced by Thrift is more then double the size of both MessagePack and Protocol Buffers, and does not compress quite as well. MessagePack compressed slightly better than Protocol Buffers and was on par in terms of speed, while being self-describing on the stream. The result of this experiment and some similar others lead us to conclude that MessagePack was the best serialisation format for DOCREP to use.

At the time of writing, the Python and Java DOCREP APIs use the official MessagePack libraries for those languages. We implemented our own C++ MessagePack library to facilitate laziness.

3.3 Laziness

The serialisation protocol was designed such that we could make the streaming aspect of DOCREP as efficient as possible. Before each collection of annotation objects appears in the serialised data, the number of bytes used to store the serialised annotations is stored. If the current application is not interested in the particular annotation types that are about to be read in, it can simply skip over the correct number of bytes without having to deserialise the internal MessagePack structure.

All three of our APIs implement this laziness. Only the types of annotations that the application specifies interest in will be deserialised at runtime. The other types of annotations will simply be kept in their serialised format and written back out to the output stream unmodified. This is also true for attributes on annotations that the current application is not interested in. The Python API provides an option to fully instantiate each of the types at runtime, even if you have not defined classes for them. Unknown annotation types will have classes created at runtime based on the schema of the types described in the serialisation protocol.

3.4 Processing tools

We trade-off performance against easy inspection of files. We provide a set of command-line tools for manipulating, filtering, and distributing DOCREP streams. The command-line tools mimic the standard set of UNIX tools used to process textual files as well as some other stream introspection and statistics gathering tools. All of these tools and their uses are documented on the DOCREP website.[7] Our provided toolbox for processing DOCREP streams contains tools for counting, visualising, filtering, ordering, partitioning, and exporting DOCREP streams. Due to space limitations in this paper, we are unable to go into these tools in detail.

Below are two examples of some of the tools in action. The first example filters the documents by a regular expression comparison against their ID attribute, and then outputs the ID of the document with the most number of tokens. The second randomly chooses 10 documents from a stream, passing them to another tool, and then opens the first returned document in the stream visualiser.

```
$ dr grep 'doc.id ~ /x-\d+/' corpus.dr | dr count -s tokens | sort -rn | head -n 1
$ dr sample -n 10 corpus.dr | ./my-tool | dr head -n 1 | dr less
```

3.5 Streaming model

Emphasising the fact that the DOCREP protocol is a streaming protocol, combining multiple DOCREP files together is as simple as concatenating the files together. The DOCREP deserialisers expect an input stream to contain zero or more serialised documents. Being able to easily distribute all documents in a corpus along with their annotation layers as a single file is very attractive.

[7] https://github.com/schwa-lab/libschwa

This kind of streaming model makes distributed processing very easy using a typical work queue model. A distributed pipeline "source" can serve the documents from the DOCREP stream by reading them off the input stream without having to deserialise them (subsection 3.3) and a "sink" can simply concatenate the received documents together to the output stream, again without having to deserialise them. We provide a DOCREP source and sink distributed processing tool along with APIs for easily writing worker clients. The distribution is achieved through ØMQ[8] which allows for both scale-up and scale-out distributed processing out of the box without the need for a separate controller process to manage communication between client processes.

4 Case study: OntoNotes 5

The OntoNotes 5 corpus (Pradhan et al., 2013) is a large corpus of linguistically annotated documents from multiple genres in three different languages. This 5th release covers newswire, broadcast news, broadcast conversation, and web data in English and Chinese, a pivot corpus in English, and newswire data in Arabic. Roughly half of the broadcast conversation data is parallel data, with some of the documents providing tree-to-tree alignments. Of the 15 710 documents in the corpus, 13 109 are in English, 2002 are in Chinese, and 599 are in Arabic.

Each of the documents in the OntoNotes 5 corpus contain multiple layers of syntactic and semantic annotations. It builds upon the Penn Treebank for syntax and PropBank for predicate-argument structure, adding named entities, coreference, and word sense disambiguation layers to some documents.

The annotations in the OntoNotes 5 corpus are provided in two different formats: as a series of flat files (340 MB) per document with each file containing one annotation layer, and as a relational database in the form of a SQL file (5812 MB). Both of these data formats have usability issues. Working with the flat files requires parsing each of the different file formats and aligning the data between the files for the same document. Working with the database requires working out how the tables are related to one another, as well as knowledge of SQL, or having access to an efficient API for querying the database.

To outline the effectiveness of document representation frameworks, and in particular the efficiency of DOCREP, we provide code to convert the OntoNotes 5 corpus into both DOCREP and UIMA representations, comparing the conversion time, resultant size on disk, and ease of doing this conversion. We provide conversion scripts in all three languages for DOCREP and in Java and C++ for UIMA. Additionally, we also provide a verification script, reproducing the original OntoNotes 5 flat files from the document representation form, ensuring that no data was lost in the conversion.

4.1 Modelling decisions

The choices made on how to model the different annotation layers were almost identical in UIMA and DOCREP. The main difference occurs when you have an annotation over a sequential span of other annotations. UIMA has no way to model this directly. The most common way users choose to model this is as a normal `Annotation` subtype with its `begin` offset set to the `begin` offset of the first covered annotation and its `end` offset set to the `end` offset of the last covered annotation. An example of this situation is named entity annotations. In OntoNotes, named entities are represented as annotations over a sequence of token annotations. How this is represented in UIMA is shown in the XML snippet in Figure 2. The main disadvantage in this modelling approach is that there is then no direct representation that the named entity annotation is an annotation over a sequence of token annotations. In DOCREP, named entity annotation is directly modelled as a sequence of token annotations. The DOCREP definition for the named entity type is shown on the right hand side of Figure 2.

DOCREP does not allow for the direct modelling of cross-document information. This occurs in the OntoNotes 5 corpus in the form of the parallel document and parallel tree information. Because DOCREP is a streaming protocol, the documents are thought of as independent from one another and as such, no formal relationships between the documents can be made at the framework level. This parallel document information can still be be stored as metadata on the documents. This situation is dealt with in UIMA by the SOFA.

[8] `http://www.zeromq.org/`

```
<typeDescription>
  <name>
ontonotes5.to_uima.types.NamedEntity
  </name>
  <description/>
  <supertypeName>
    uima.tcas.Annotation
  </supertypeName>
  <features>
    <featureDescription>
      <name>tag</name>
      <description>The NE tag.</description>
      <rangeTypeName>uima.cas.String</rangeTypeName>
    </featureDescription>
    <featureDescription>
      <name>startOffset</name>
      <description>Character offset into the start token.</description>
      <rangeTypeName>uima.cas.Integer</rangeTypeName>
    </featureDescription>
    <featureDescription>
      <name>endOffset</name>
      <description>Character offset into the end token.</description>
      <rangeTypeName>uima.cas.Integer</rangeTypeName>
    </featureDescription>
  </features>
  </typeDescription>
```

```
@dr.Ann
public class NamedEntity extends AbstractAnn {
  @dr.Pointer public Slice<Token> span;
  @dr.Field public String tag;
  @dr.Field public int startOffset;
  @dr.Field public int endOffset;
}
```

Figure 2: Defining the named entity annotation type in UIMA (left) and the DOCREP Java API (top-right).

	UIMA							DOCREP		
	Java XMI	Java XCAS	Java bin	Java cbin	C++ XMI	C++ XCAS	C++ bin	Java –	C++ –	Python –
Conversion time	25	25	25	25	77	77	77	12	12	27
Serialisation time	131	122	2103	76	630	611	695	61	23	32
Size on disk	1894	3252	1257	99	2141	3252	2135	371	371	371

Table 2: A comparison of the resources required to represent the OntoNotes 5 corpus in UIMA and DOCREP. Times are reported in seconds and sizes are reported in MB.

4.2 Empirical results

In these experiments, we first load all of the data into memory from the database for the current document we are processing. This data is stored in an object structure which knows nothing about document representation frameworks. We then convert this object representation into the appropriate UIMA and DOCREP annotations, recording how long the conversion took. The UIMA and DOCREP versions of the documents are then serialised to disk, recording how long the serialisation took and the resultant size on disk. All of these performance experiments were run on the same isolated machine, running 64-bit Ubuntu 12.04, using OpenJDK 1.7, CPython 2.7, and gcc 4.8.

In order to provide a fair comparison between UIMA and DOCREP, we perform the conversion using both the Java and C++ UIMA APIs, as well as using all three DOCREP APIs (Java, C++, and Python). The code to load the data from the database and construct the in-memory object structure was common between the UIMA and DOCREP conversions. For UIMA, we serialise in all available output formats: both the XMI and XCAS XML formats, the binary format (bin), and the compressed binary (cbin) format. The UIMA C++ API does not appear to support output in the compressed binary format.

The result of this conversion process can be seen in Table 2. The first row shows the accumulated time taken to convert all of the documents from their in-memory representation into UIMA and DOCREP annotations. As visible in the table, DOCREP performs this conversion twice as fast as UIMA in Java and six times as fast as UIMA in C++ The second row shows the accumulated time taken to serialise

	Flat files	DOCREP	UIMA XMI	UIMA XCAS	UIMA bin	UIMA cbin	SQL	MySQL -indices	MySQL +indices
Uncompressed	340	371	1894	3252	1257	99	4560	4303	5812
`gzip` (DEFLATE)	52	115	268	330	375	66	646	–	–
`xz` (LZMA)	30	69	144	185	150	65	262	–	–

Table 3: A comparison of the how well each of the annotation serialisation formats compress using standard compression libraries. All sizes are reported in MB.

all of the documents to disk. DOCREP serialises up to 34 times faster than UIMA in Java, depending on the UIMA output format, and up to 30 times faster in C++ Ṫhe third row in this table shows the accumulated serialisation size on disk. Apart from the compressed binary output format in UIMA (cbin), DOCREP serialisation requires up to nine times less space than UIMA Ẇe are unsure why the sizes for the different output formats in UIMA do not match up between the Java and C++ APIs Ẇe are also unsure why the UIMA Java binary serialisation is so slow, especially in comparison to the compressed binary serialisation.

Table 3 shows how well each of the serialisation formats compress using three standard compression libraries. Each of these compression libraries were run with their default settings. The files generated by UIMA as well as the "flat file" files were first placed into a tarball so that the compression algorithms could be run over the whole corpus instead of per document. The "flat files" used were the original OntoNotes 5 flat files containing the annotation layers that were converted. The SQL numbers are using the original OntoNotes 5 SQL file. The MySQL numbers are obtained after loading the original SQL into a MySQL database and obtaining table and index sizes from the `information_schema.tables` table. The MySQL database was not altered from the initial import. Unsurprisingly, the DOCREP binary representation does not compress as well as textual serialisation formats with lots of repetition, such as XML or the original stand-off annotation files. However, under all of these reported situations, apart from the UIMA compressed binary format, our DOCREP representation is two to five times smaller than its UIMA counterpart, and 15 times smaller than the representation in MySQL. The UIMA compressed binary (cbinary) format has already been compressed so it is unsurprising that compressing it further makes little difference.

5 Usability

We have primarily evaluated the usefulness of DOCREP from an efficiency perspective, reporting time and space requirements for a complex corpus conversion. In this section, we provide feedback from NLP researchers in our lab who have been using DOCREP over the past two years for a variety of NLP tasks. As researchers ourselves, we are aware of how valuable research time is. We provide these real-world examples of DOCREP's use to solidifying that DOCREP is a valuable tool for researchers.

Coreference DOCREP *is a great tool for this project as all we want to do is develop a good coreference system; we do not want to have to worry about the storage of data. Having an* API *in Python is super convenient, allowing us to write code that changes frequently as we try new ideas.* Related publication: Webster and Curran (2014)

Event Linking *Some work on Event Linking sought to work with gold annotations on one hand, and knowledge from web-based hyperlinks on the other. For some processes these data sources were to be treated identically, and for some differently.* DOCREP*'s extensibility easily supported this use-case, while providing a consistent polymorphic abstraction that made development straightforward, while incorporating many other layers of annotation such as extracted temporal relations. Separately, describing the relationship between a pair of documents in* DOCREP *was a challenging use-case that required more engineering and fore-thought than most* DOCREP *applications so far.* Related publication: Nothman et al. (2012).

Named Entity Linking *Our approach to* NEL *uses a pipeline of components and we initially wrote our own* DRF *using Python's object serialisation. While this worked well initially, we accrued technical debt as we added features with minimal refactoring. Before too long, a substantial part of our experiment runtime was devoted to dataset loading and storage.* DOCREP *made this easier and using* UNIX *pipelines over structured document objects is a productive workflow.* Related publications: Radford et al. (2012); Pink et al. (2013).

Quote Extraction and Attribution *For this task we performed experiments over four corpora, all with distinct data formats and assumptions. Our early software loaded each format into memory, which was a slow, error-prone, and hard-to-debug process. This approach became completely unusable when we decided to experiment with coreference systems, as it introduced even more unique data formats. Converting everything to* DOCREP *greatly simplified the task, as we could represent everything we needed efficiently, and within one representation system. We also gained a nice speed boost, and were able to write a simple set of tests that examined a given* DOCREP *file for validity, which greatly improved our code quality.* Related publication: O'Keefe et al. (2013).

Slot Filling *Being one of the last stages in an* NLP *pipeline, slot filling utilises all of the document information it can get its hands on. Being able to easily accept annotation layers from prior* NLP *components allows us to focus on slot filling instead of component integration engineering. Having access to a multi-language* API *means we are able to write efficiency-critical code in C++ and the more experimental and dynamic components in Python.*

6 Conclusion

We present a light-weight and easy-to-use document representation framework for the busy NLP researcher who wants to model document structure, but does not want to use a heavy-weight DRF. We provide empirical evidence of the efficiency of DOCREP, and provide insights into its use within our research group over the past two years. We believe NLP other researchers will benefit from DOCREP as they are now able to utilise the usefulness of a DRF without it getting in the way of their research time.

Acknowledgments

We would like to thank the anonymous reviewers for their useful feedback. We would also like to thank Will Radford and Joel Nothman for their contributions to this paper as well as to DOCREP itself over the past years. This work was supported by ARC Discovery grant DP1097291 and the Capital Markets CRC Computable News project.

References

Steven Bird and Mark Liberman. 1999. A formal framework for linguistic annotation. *Speech Communication*, 33:23–60.

Chris Brew and Marc Moens. 2002. Data-intensive linguistics. HCRC Language Technology Group, University of Edinburgh.

Kenneth Ward Church. 1994. Unix™ for poets. *Notes of a course from the European Summer School on Language and Speech Communication, Corpus Based Methods*.

James Clarke, Vivek Srikumar, Mark Sammons, and Dan Roth. 2012. An NLP curator (or: How I learned to stop worrying and love NLP pipelines). In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey.

Hamish Cunningham. 2002. GATE, a general architecture for text engineering. *Computers and the Humanities*, 36:223–254.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: an architecture for development of robust HLT applications. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 168–175. Association for Computational Linguistics, Philadelphia, Pennsylvania, USA.

T. Götz and O. Suhre. 2004. Design and implementation of the UIMA common analysis system. *IBM Systems Journal*, 43(3):476–489.

Nancy Ide, Collin Baker, Christiane Fellbaum, and Rebecca Passonneau. 2010. The manually annotated sub-corpus: A community resource for and by the people. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 68–73. Uppsala, Sweden.

Nancy Ide and Laurent Romary. 2004. International standard for a linguistic annotation framework. *Natural Language Engineering*, 10(3-4):211–225.

Nancy Ide and Laurent Romary. 2006. Representing linguistic corpora and their annotations. In *Proceedings of the Fifth Language Resources and Evaluation Conference LREC*.

Nancy Ide and Keith Suderman. 2007. GrAF: A graph-based format for linguistic annotations. In *Proceedings of the Linguistic Annotation Workshop*, pages 1–8. Association for Computational Linguistics, Prague, Czech Republic.

Nancy Ide and Keith Suderman. 2009. Bridging the Gaps: Interoperability for GrAF, GATE, and UIMA. In *Proceedings of the Third Linguistic Annotation Workshop*, pages 27–34. Association for Computational Linguistics, Suntec, Singapore.

Adam Lally, Karin Verspoor, and Eoric Nyberg. 2008. *Unstructured Information Management Architecture (UIMA) Version 1.0. Standards Specification 5, OASIS*.

Arne Neumann, Nancy Ide, and Manfred Stede. 2013. Importing MASC into the ANNIS linguistic database: A case study of mapping GrAF. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 98–102. Sofia, Bulgaria.

Joel Nothman, Matthew Honnibal, Ben Hachey, and James R. Curran. 2012. Event linking: grounding event reference in a news archive. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 228–232. Jeju, Korea.

Tim O'Keefe, James R. Curran, Peter Ashwell, and Irena Koprinska. 2013. An annotated corpus of quoted opinions in news articles. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 516–520. Association for Computational Linguistics, Sofia, Bulgaria.

Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. 2011. English Gigaword Fifth Edition. Technical report, Linguistic Data Consortium, Philadelphia.

Glen Pink, Will Radford, Will Cannings, Andrew Naoum, Joel Nothman, Daniel Tse, and James R. Curran. 2013. SYDNEY_CMCRC at TAC 2013. In *Proceedings of the Text Analysis Conference*. National Institute of Standards and Technology, Gaithersburg, MD USA.

Sameer Pradhan, Alessandro Moschitti, Nianwen Xue, Hwee Tou Ng, Anders Björkelund, Olga Uryupina, Yuchen Zhang, and Zhi Zhong. 2013. Towards Robust Linguistic Analysis using OntoNotes. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 143–152. Sofia, Bulgaria.

Will Radford, Will Cannings, Andrew Naoum, Joel Nothman, Glen Pink, Daniel Tse, and James R. Curran. 2012. (Almost) Total Recall – SYDNEY_CMCRC at TAC 2012. In *Proceedings of the Text Analysis Conference*. National Institute of Standards and Technology, Gaithersburg, MD USA.

Kellie Webster and James R. Curran. 2014. Low memory incremental coreference resolution. In *Proceedings of COLING 2014*. The COLING 2014 Organizing Committee, Dublin, Ireland. To appear.