COPYRIGHT AND USE OF THIS THESIS

This thesis must be used in accordance with the provisions of the Copyright Act 1968.

Reproduction of material protected by copyright may be an infringement of copyright and copyright owners may be entitled to take legal action against persons who infringe their copyright.

Section 51 (2) of the Copyright Act permits an authorized officer of a university library or archives to provide a copy (by communication or otherwise) of an unpublished thesis kept in the library or archives, to a person who satisfies the authorized officer that he or she requires the reproduction for the purposes of research or study.

The Copyright Act grants the creator of a work a number of moral rights, specifically the right of attribution, the right against false attribution and the right of integrity.

You may infringe the author’s moral rights if you:

- fail to acknowledge the author of this thesis if you quote sections from the work
- attribute this thesis to another author
- subject this thesis to derogatory treatment which may prejudice the author’s reputation

For further information contact the University’s Director of Copyright Services

sydney.edu.au/copyright
DOCREP: Document Representation for Natural Language Processing

Tim Dawborn

Supervisor: James R. Curran

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in the School of Information Technologies at The University of Sydney

School of Information Technologies

2016
Abstract

The field of natural language processing (NLP) revolves around the computational interpretation and generation of natural language. The language typically processed in NLP occurs in paragraphs or documents rather than in single isolated sentences. Despite this, most NLP tools operate over one sentence at a time, not utilising the context outside of the sentence nor any of the metadata associated with the underlying document. One pragmatic reason for this disparity is that representing documents and their annotations through an NLP pipeline is difficult with existing infrastructure.

Representing linguistic annotations for a text document using a plain text markup-based format is not sufficient to capture arbitrarily nested and overlapping annotations. Despite this, most linguistic text corpora and NLP tools still operate in this fashion.

A document representation framework (DRF) supports the creation of linguistic annotations stored separately to the original document, overcoming this nesting and overlapping annotations problem. Despite the prevalence of pipelines in NLP, there is little published work on, or implementations of, DRFs. The main DRFs, GATE and UIMA, exhibit usability issues which have limited their uptake by the NLP community.

This thesis aims to solve this problem through a novel, modern DRF, DOCREP; a portmanteau of document representation. DOCREP is designed to be efficient, programming language and environment agnostic, and most importantly, easy to use. We want DOCREP to be powerful and simple enough to use that NLP researchers and language technology application developers would even use it in their own small projects instead of developing their own ad hoc solution.

This thesis begins by presenting the design criteria for our new DRF, extending upon existing requirements from the literature with additional usability and efficiency requirements that should lead to greater use of DRFs. We outline how our new DRF, DOCREP, differs from existing DRFs in terms of the data model, serialisation strategy, developer interactions, support for rapid prototyping, and the expected runtime and environment requirements. We then describe our provided implementations of
DOCREP in Python, C++, and Java, the most common languages in NLP; outlining their efficiency, idiomaticity, and the ways in which these implementations satisfy our design requirements.

We then present two different evaluations of DOCREP. First, we evaluate its ability to model complex linguistic corpora through the conversion of the OntoNotes 5 corpus to DOCREP and UIMA, outlining the differences in modelling approaches required and efficiency when using these two DRFs. Second, we evaluate DOCREP against our usability requirements from the perspective of a computational linguist who is new to DOCREP. We walk through a number of common use cases for working with text corpora and contrast traditional approaches against their DOCREP counterpart. These two evaluations conclude that DOCREP satisfies our outlined design requirements and outperforms existing DRFs in terms of efficiency, and most importantly, usability.

With DOCREP designed and evaluated, we then show how NLP applications can harness document structure. We present a novel document structure-aware tokenization framework for the first stage of full-stack NLP systems. We then present a new structure-aware NER system which achieves state-of-the-art results on multiple standard NER evaluations. The tokenization framework produces its tokenization, sentence boundary, and document structure annotations as native DOCREP annotations. The NER system consumes DOCREP annotations and utilises many components of the DOCREP runtime.

We believe that the adoption of DOCREP throughout the NLP community will assist in the reproducibility of results, substitutability of components, and overall quality assurance of NLP systems and corpora, all of which are problematic areas within NLP research and applications. This adoption will make developing and combining NLP components into applications faster, more efficient, and more reliable.
Acknowledgements

First and foremost, I would like to thank my supervisor for his guidance, advice, discipline, thoroughness, stubbornness, and kindness. James has been an invaluable supervisor, mentor, co-founder, colleague, and friend over the last decade, and I look forward to working with him again in the future.

Next, I would like to thank all members of \textit{e}-lab, both past and present. These friends and colleagues have helped shape this thesis in many different ways, through draft paper and thesis readings, sitting through half-baked practice presentations, and the countless discussions and whinging sessions which occurred during the walks to and from the coffee machine. Specific thanks must go to Joel Nothman for his enthusiasm and commitment, with many hours of chats, discussions, prototypes, and bug reports to help shape DOCREP to what it is today.

This thesis is also the by-product of numerous games of tennis and squash with other \textit{e}-lab members. Thanks to Will Radford, Ben Hachey, Glen Pink, and Tim O’Keefe for the early morning dedication to tennis, and to Dominick Ng and Glen for advice on improving my squash game. No doubt these games helped all of us to stay sane.

\textit{e}-lab member Kellie Webster should also get a specific mention for putting up with me during our conference trip to COLING and the University of Edinburgh. It is always nice having a familiar face on the other side of the globe.

As anyone who has completed a PhD can attest to, writing up can be a long and drawn out process. I was fortunate enough to have Dom as a writing up companion
over the last couple of months. I think we both benefited from being able to vent to someone in the same situation.

Thanks should also go to the numerous housemates over the years who have had to live with a PhD student. This includes fellow α-lab member and close friend, James “Jcon” Constable. There have been many thoughtful discussions and thoughtless antics which have occurred in that apartment. Zac Cohan should get a special mention here. Zac, Jcon, and I spent what probably summed to months in philosophical, existential, and political debates and discussions. These were a fantastic source of PhD procrastination.

Other members of SIT who should get an explicit mention include Bernhard Scholz for continually pestering me to complete my candidature, and Georgina Wilcox for many cups of tea and ridiculous stories in the early years.

Any project as large or as long as a PhD thesis encounters many bumps along the way. Unfortunately, these often result in unexpected costs and sacrifices, including time, money, sanity, and people. Some of these are easier to recover from than others. Saudade.
Statement of compliance

I certify that:

- I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

- I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

- this Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Tim Dawborn

Signature: Date: 21st April 2016
# Contents

1 Introduction .................................................. 1
   1.1 Outline .................................................. 3
   1.2 A call to arms ........................................... 6
   1.3 Summary of contributions .............................. 7
   1.4 Publications and collaboration ...................... 8

2 Background ............................................... 11
   2.1 Representing annotations ............................. 12
   2.2 Annotation standards .................................. 16
      2.2.1 Corpus Encoding Standard .................... 16
      2.2.2 Annotation Graphs .............................. 19
      2.2.3 ISO SC4 TC37 ................................... 21
      2.2.4 Linguistic Annotation Format ................ 23
      2.2.5 Graph Annotation Framework ................. 27
   2.3 Document Representation Frameworks ............... 28
      2.3.1 ATLAS ........................................... 29
      2.3.2 GATE .......................................... 33
      2.3.3 UIMA ......................................... 36
   2.4 Use of annotation standards and DRFs for corpora . 41
   2.5 Interacting with annotations ........................ 42
      2.5.1 Flat files .................................... 42
CONTENTS

4 DOCREP: Implementation

4.1 Data model ................................................. 76
  4.1.1 Annotations ........................................ 76
  4.1.2 Documents .......................................... 78
  4.1.3 Stores ............................................. 78
  4.1.4 Primitive-typed fields ............................ 79
  4.1.5 Pointer and self-pointer fields ..................... 80
  4.1.6 Slice fields ....................................... 81
  4.1.7 Overview ......................................... 82

4.2 Serialisation ............................................ 84
  4.2.1 Wire protocol design ............................... 84
  4.2.2 Serialisation format ................................ 90
  4.2.3 Evaluation ........................................ 97
  4.2.4 Serialising with MessagePack ....................... 99

4.3 Implementing the DOCREP APIs ......................... 101
  4.3.1 The Python API .................................... 101
  4.3.2 The Java API ...................................... 105
  4.3.3 The C++ API ...................................... 107
  4.3.4 Consistency and idiomaticity ....................... 113

4.4 The DOCREP runtime ................................ 114
  4.4.1 Lazy serialisation .................................. 115
  4.4.2 Configurable wire to schema mappings ............. 118
  4.4.3 Decorators ....................................... 120

4.5 Summary ................................................. 124

5 Evaluating DOCREP on OntoNotes ...................... 127

5.1 The OntoNotes 5 corpus ............................... 128
  5.1.1 Flat files ......................................... 130
  5.1.2 Database ......................................... 132
5.2 Modelling OntoNotes in DOCREP and UIMA ............... 135
  5.2.1 Overview ........................................ 135
  5.2.2 Modelling decisions ............................... 137

5.3 Evaluation via corpus representation ....................... 140
  5.3.1 Corpus conversion .................................. 141
  5.3.2 Empirical evaluation ............................... 142
  5.3.3 Quality assurance .................................. 144

5.4 Evaluation against our design requirements .............. 147

5.5 Summary .............................................. 147

6 Evaluating DOCREP as a user .............................. 149
  6.1 Starting out using a DRF ................................ 150
  6.2 Working with DOCREP streams ......................... 154
    6.2.1 Already-familiar utilities ........................ 155
    6.2.2 Working with DOCREP streams on the command-line . 158
    6.2.3 Evaluating UNIX tools against the DOCREP command-line . 164
    6.2.4 Discussion ....................................... 169
  6.3 Testimonials .......................................... 169
  6.4 Evaluation against our design requirements ............ 172
  6.5 Summary .............................................. 173

7 DOCREP for Tokenization and SBD ...................... 175
  7.1 Motivation ........................................... 177
  7.2 The tokenization framework ............................ 179
    7.2.1 Input transcoding ................................ 180
    7.2.2 Document structure interpretation ............... 181
    7.2.3 Tokenization and SBD ............................. 184
  7.3 Evaluation and comparison ................................ 187
  7.4 DOCREP models ....................................... 192
CONTENTS

7.4.1 Discontiguous spans .................................. 192
7.5 Summary ................................................. 193

8 DOCREP for NER ............................................ 195

8.1 Named Entity Recognition .................................. 195
8.1.1 Sequence tagging ..................................... 196
8.1.2 Shared tasks and data .................................. 201
8.1.3 Evaluation ........................................... 209
8.1.4 Learning method ....................................... 212
8.1.5 External resources .................................... 214
8.2 Features used in NER systems ......................... 217
8.2.1 Morphosyntactic features .............................. 217
8.2.2 Other current-token features ......................... 217
8.2.3 Contextual features .................................. 218
8.3 State-of-the-art English NER ......................... 220
8.3.1 CoNLL 2003 ......................................... 220
8.3.2 OntoNotes ........................................... 221
8.4 Building a NER system with document structure .... 223
8.4.1 Implementation ....................................... 223
8.4.2 Results and evaluation .............................. 231
8.5 Summary ................................................. 235

9 Conclusion .................................................. 237

9.1 Future Work ............................................. 240
9.2 Summary ................................................. 241

Appendices .................................................. 243

A NER datasets: CoNLL 2003 .............................. 245
A.1 Category distribution .................................. 245
A.2 Sentence boundary errors ........................................ 246

B NER datasets: OntoNotes 5 ........................................ 247
B.1 Category distribution for the official splits .................. 248
B.2 Category distribution for the Passos et al. (2014) splits .... 249
B.3 Generating the Passos et al. (2014) splits .................... 250

Bibliography ............................................................ 253
List of Figures

2.1 An example raw sentence without any linguistic annotation. . . . . . . 11
2.2 Inline annotations for segmentation and linguistic information. . . . . . . 13
2.3 An example of stand-off annotations. . . . . . . . . . . . . . . . . 14
2.4 An example XCES stand-off annotation. . . . . . . . . . . . . . . . . 17
2.5 An example sentence in its PTB and corresponding XCES formats. . . . . 18
2.6 The overall architecture of LAF. . . . . . . . . . . . . . . . . . . . . . . . 23
2.7 Pivot data formats aid with format translation. . . . . . . . . . . . . . . 24
2.8 A visualisation of a segmentation and annotation graph in LAF. . . . . 25
2.9 LAF dump format serialisation of a primary segmentation. . . . . . . . . 26
2.10 LAF dump format serialisation of a feature structure. . . . . . . . . . . . 26
2.11 LAF dump format of phrase structure information over multiple arcs. . . . 26
2.12 GRAF represents alternate segmentations in a single graph structure. . . 28
2.13 The overall structure of ATLAS. . . . . . . . . . . . . . . . . . . . . . . . 29
2.14 The Annotation Graph and ATLAS object models. . . . . . . . . . . . . 30
2.15 Identified regions in an ATLAS signal. . . . . . . . . . . . . . . . . . . . 30
2.16 Creating an ATLAS annotation over a region. . . . . . . . . . . . . . . 31
2.17 Linking ATLAS annotations together. . . . . . . . . . . . . . . . . . . . 31
2.18 An example of the ATLAS XML representation for annotation graphs. . . 32
2.19 A screenshot of the GATE Developer user interface. . . . . . . . . . . . . 34
2.20 The GATE Embedded APIs. . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.21 The annotation graph structure used in GATE. . . . . . . . . . . . . . . 35
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.22</td>
<td>UIMA high-level architecture.</td>
<td>37</td>
</tr>
<tr>
<td>2.23</td>
<td>Conceptual view of the UIMA CAS.</td>
<td>37</td>
</tr>
<tr>
<td>2.24</td>
<td>The structure of a UIMA feature on the heap.</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>An example schema to be modelled in a DRF.</td>
<td>58</td>
</tr>
<tr>
<td>3.2</td>
<td>Retrieving annotations of a given type in GATE.</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of the DOCREP data modelling components.</td>
<td>83</td>
</tr>
<tr>
<td>4.2</td>
<td>The different ways a UTF-8 string can be encoded in MessagePack.</td>
<td>96</td>
</tr>
<tr>
<td>4.3</td>
<td>The DOCREP wire protocol specification.</td>
<td>100</td>
</tr>
<tr>
<td>4.4</td>
<td>An example DOCREP schema definition using the Python API.</td>
<td>102</td>
</tr>
<tr>
<td>4.5</td>
<td>Reading and writing using the DOCREP Python API.</td>
<td>103</td>
</tr>
<tr>
<td>4.6</td>
<td>DOCREP slices used with Python slice semantics.</td>
<td>103</td>
</tr>
<tr>
<td>4.7</td>
<td>Using the automagic reader provided by the Python API.</td>
<td>104</td>
</tr>
<tr>
<td>4.8</td>
<td>An example DOCREP schema definition using the Java API.</td>
<td>106</td>
</tr>
<tr>
<td>4.9</td>
<td>Reading and writing using the DOCREP Java API.</td>
<td>106</td>
</tr>
<tr>
<td>4.10</td>
<td>An example DOCREP schema definition using the C++ API.</td>
<td>109</td>
</tr>
<tr>
<td>4.11</td>
<td>Options for laying out the annotation objects in memory.</td>
<td>110</td>
</tr>
<tr>
<td>4.12</td>
<td>Reading and writing using the DOCREP C++ API.</td>
<td>112</td>
</tr>
<tr>
<td>4.13</td>
<td>DOCREP annotation slices being used as an iterator.</td>
<td>113</td>
</tr>
<tr>
<td>4.14</td>
<td>The ParseNode annotation type in all three DOCREP APIs.</td>
<td>114</td>
</tr>
<tr>
<td>4.15</td>
<td>Code snippets to illustrate how laziness is supported.</td>
<td>116</td>
</tr>
<tr>
<td>4.16</td>
<td>Different field names potentially cause issues with automatic schemas.</td>
<td>118</td>
</tr>
<tr>
<td>4.17</td>
<td>An example use of the “reverse slices” decorator.</td>
<td>122</td>
</tr>
<tr>
<td>4.18</td>
<td>An example use of the “sequence tagger” decorator.</td>
<td>123</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of multiple annotation layers on a document.</td>
<td>129</td>
</tr>
<tr>
<td>5.2</td>
<td>Various snippets from annotation layer flat files.</td>
<td>131</td>
</tr>
<tr>
<td>5.3</td>
<td>An entity-relationship diagram for the OntoNotes 5 database.</td>
<td>133</td>
</tr>
<tr>
<td>5.4</td>
<td>An example use of the OntoNotes 5 database.</td>
<td>134</td>
</tr>
</tbody>
</table>
5.5 Defining a named entity annotation type in UIMA. .................. 136
5.6 Defining a named entity annotation type in DOCREP. .............. 136
5.7 DOCREP models for the syntax annotation layer. ..................... 137
5.8 DOCREP models for the speaker annotation layer. ..................... 138
5.9 DOCREP models for the coreference annotation layer. .............. 139
5.10 DOCREP models for the sentence annotation layer. .................. 139
5.11 A snippet of the speaker flat file format. ............................ 140
5.12 Annotation discrepancies between the flat files and the database. .. 145
5.13 Broken annotations and file format in the gold standard data. .... 146

6.1 Example screenshots of inspect DOCREP documents. ................ 162

7.1 An example HTML input stream encoded in Windows-1250. ......... 178
7.2 Transcoding into UTF-8 while maintaining offsets. .................. 180
7.3 Interpreting document structure while maintaining offsets. ......... 182
7.4 The resulting document segmentation with offsets maintained. ...... 184
7.5 A snippet of the Ragel tokenization rules. ............................ 186
7.6 The first 1987 WSJ document in the Tipster corpus. .................. 188
7.7 Comparing DOCREP and LingPipe tokenization and SBD. .......... 190
7.8 The Token and Sentence DOCREP models. ........................... 193

8.1 A comparison of different sequence tag encodings. .................. 198
8.2 An example sentence with IOB1, IOB2, and BMEWO NE tags ........ 199
8.3 NE spans can be natively represented in a DRF such as DOCREP.  .. 200
8.4 A snippet from the MUC-7 data. .................................... 201
8.5 A snippet from the CoNLL 2003 English data. ......................... 203
8.6 Possible valid NE annotations for the U.S. .......................... 210
8.7 An example of desired non-local constraints. ........................ 213
8.8 A document model eases the implementation of certain features. ... 224
8.9 The DOCREP NER Token model. ............................... 225
8.10 A snippet of the block structure used in our NER system. . . . . . . . . 228
8.11 How document structure is utilised in our NER system. . . . . . . . . 229
8.12 An example of ambiguity in document headings. . . . . . . . . . . . . . 229
List of Tables

4.1 High-level differences between binary serialisation formats. 93
4.2 Primitive data type support in binary serialisation formats. 95
4.3 Non-primitive and collection type support in binary serialisation formats. 96
4.4 Comparison of binary serialisations for stand-off linguistic annotations. 98

5.1 Size of each annotation layer in OntoNotes 5. 129
5.2 Representing OntoNotes 5 in UIMA and DOCREP. 142
5.3 Compression of OntoNotes 5 in various representations. 144

6.1 DOCREP tools and their UNIX counterparts. 156

7.1 Tokenization statistics for the WSJ section of the Tipster corpus. 189

8.1 Size breakdown of CoNLL shared task data splits. 204
8.2 Size breakdown of the two OntoNotes 5 splits. 206
8.3 Discrepancies in reported entities in the CoNLL 2012 test set. 207
8.4 Reported performance on CoNLL 2003 English. 221
8.5 Reported performance on Finkel and Manning (2009). 222
8.6 Our state-of-the-art performance on CoNLL 2003 English. 232
8.7 Performance breakdown of our CoNLL 2003 English dev results. 233
8.8 Performance breakdown of our CoNLL 2003 English test results. 233
8.9 Our state-of-the-art performance on Finkel and Manning (2009). 235
8.10 Our state-of-the-art performance on Passos et al. (2014). 235
A.1  NE category distribution across the CoNLL 2003 English splits. . . . . 245
A.2  Incorrect sentence boundaries in the CoNLL 2003 English data. . . . . 246
B.1  Category distribution across the official OntoNotes 5 splits. . . . . . . . 248
B.2  Category distribution across the Passos et al. (2014) splits. . . . . . . . 249
1 Introduction

The more you know, the more you realise you know nothing.

__________________________
Socrates

The field of natural language processing (NLP) or computational linguistics (CL) revolves around the computational interpretation and generation of natural language. NLP is broken down into many subfields and subtasks, where the output from one task often feeds into the next task. A computer program to perform one or more of these tasks is referred to as an NLP tool or NLP application. This composition of multiple NLP tools to perform a sequence of these tasks is referred to as an NLP pipeline. For example, a full-stack NLP pipeline might consist of tokenization, sentence boundary detection, named entity recognition, parsing, coreference resolution, and named entity linking, executed in that order; each task building upon the linguistic information provided by the previous task.

The language typically processed in NLP occurs in paragraphs or documents of text rather than in single isolated sentences. Despite this, NLP tools typically operate over one sentence of text at a time, not utilising the context outside of the sentence nor any of the metadata associated with the underlying document. This disparity exists for two main reasons. First, in a machine learning context, it is often unclear how to incorporate non-local information into the sentence-level optimisation efficiently. The second pragmatic reason is that representing documents and their annotations through each component in an NLP pipeline is difficult with existing infrastructure.
Historically, both unannotated and annotated NLP corpora consisted of only plain text documents, usually from newswire services, e.g. the Penn Treebank (Marcus et al., 1993), Tipster (Harman and Liberman, 1993), and English Gigaword (Parker et al., 2011). This, amongst other factors, has resulted in the majority of NLP tools only operating over plain text inputs. Linguistic annotations in NLP corpora are typically represented in an *ad hoc* markup format; a format that is unambiguous, simple, and easy to work with. This markup normally occurs in an inline fashion, with the linguistic annotations being inserted into the underlying document. The disadvantages of inline annotation are that it cannot represent arbitrarily nested or overlapping annotations and it disrupts any existing structure present in the original document.

An alternative to an inline representation is to store the linguistic annotations in a separate location. In this stand-off annotation approach, annotations can refer back to the original document, stating which segment they apply to. The limitation on arbitrarily nested or overlapping annotations is bypassed since the original document is not being modified. Additionally, stand-off annotations do not interfere with existing document structure in the original document.

A number of more structured and formally defined linguistic annotation formats have been proposed over the years. These formats, most of which involve the use of a structured markup language for serialisation, are plain text formats which overcome this nesting limitation via piggybacking technologies built on top of the structured markup language. The most common structured markup languages used are SGML and XML. While these provide a solution to the representation of linguistic annotations, they are not computationally efficient or easy to work with. As a result, these defined standards have not had widespread use in the NLP community.

A document representation framework (DRF) supports the creation of stand-off annotations over collections of documents. DRFs provide an API for systems to interact with the annotations, and in doing so, allows different systems to interact through the annotations. Despite the prevalence of pipelining within NLP, there is little published
work on, or implementations of, DRFs. The three main DRFs that have been developed over the last 15 years are ATLAS, GATE, and UIMA.

Any kind of NLP pipeline which operates over documents rather than separate sentences will need some form of DRF, albeit well defined or *ad hoc*. The use of these existing DRFs within the NLP community has been limited. There are a number of reasons for this, including usability issues, resource requirements, specific development workflows, and the fact they are not programming language agnostic.

The field of NLP has often focussed on conceptual, abstract annotation frameworks, rather than implementation details of such tools. With the size of corpora increasing, and the increased demand for practical NLP applications, the efficiency of these tools can no longer be ignored.

This thesis aims to solve this problem. **DOCREP**, a portmanteau of *document representation*, is our novel DRF presented in this thesis. The **DOCREP** framework is designed to be efficient, programming language and environment agnostic, and most importantly, easy to use. We want **DOCREP** to be powerful and simple enough to use that NLP researchers and language technology (LT) application developers would even use it in their own small projects instead of rolling their own *ad hoc* solution.

### 1.1 Outline

In Chapter 2 we present the existing approaches to document representation, describing these existing annotation formats and DRFs. This chapter also includes a discussion on existing design criteria and proposals for adequately representing linguistic annotations. We also outline the interaction between annotation formats and DRFs, and the ways in which existing NLP pipelining frameworks have solved this problem. We highlight issues with each of the existing approaches. We conclude that the field is lacking a lightweight, efficient, elegant, and modern DRF that is programming language agnostic, easy to learn, and lightweight in design.
In Chapter 3, we outline the requirements for this new DRF, making explicit the use cases that the current DRFs fail to satisfy. This new DRF, DOCREP, is a primary contribution of this thesis. Our outlined design requirements are compared to the requirements proposed in the literature for a general linguistic annotation format. We find that these sets of requirements are mostly in agreement, except we include additional pragmatic requirements.

Chapter 4 goes on to describe our implementations of DOCREP, one in each of the main programming languages used within the NLP community. We go through the formal specifications of DOCREP, outlining how annotations are represented at runtime and in their serialised form. These definitions are made in a programming language agnostic manner so that a DOCREP API can be easily implemented in a new language. One of our design goals presented in Chapter 3 is that DOCREP APIs should be consistent across languages but idiomatic within each language. Chapter 4 demonstrates how each implementation satisfies these goals.

With an implementation in place, Chapters 5 and 6 evaluate this DRF from two different perspectives — its ability to model diverse linguistic annotations efficiently and elegantly, and the ability for a new user to pick up and use the DRF.

In Chapter 5, we show how a diverse multilayered corpus, in particular OntoNotes 5, can be represented in a DRF, and why the existing corpus distribution approaches are inferior both for usability and quality assurance. This chapter compares the representation and interface provided by DOCREP, UIMA, and other corpus distribution formats when modelling this corpus. Experiments performed in this chapter conclude that DOCREP provides a significantly more runtime and space efficient solution than the UIMA DRF, and outperforms existing corpus distribution strategies in performance and usability.

Chapter 6 presents an evaluation of DOCREP from the perspective of a user, showing how it meets our design requirements presented in Chapter 3 — providing users with a lightweight, programming language agnostic, easy to use DRF for working with
linguistic annotations. This chapter demonstrates how operations which computational linguistics commonly perform on text corpora map to DOCREP equivalents through the DOCREP command-line tools. These tools are designed with the UNIX philosophy of chainable single-task tools which can be composed to solve larger tasks. Chapter 6 concludes with testimonials from DOCREP users from within our research lab and from the LT application development community.

With DOCREP designed, implemented, and evaluated, Chapters 7 and 8 move in a different direction. In these chapters, we utilise the document structure provided by DOCREP to improve two different NLP tasks.

Chapter 7 describes our newly developed document-aware tokenization framework which maintains byte and Unicode code point offsets back into the original document during document structure interpretation, input transcoding, tokenization, and sentence boundary detection (SBD). The tokenization and document structure are produced natively as DOCREP annotations. Users of our framework’s output are able to map linguistic units back to their location in the original document while simultaneously having access to the document’s internal structure.

Tokenization and sentence boundary detection are precursor tasks to almost all NLP pipelines, so providing document structure and offset information at the start of the pipeline allows all downstream applications to utilise this rich information. With a high-quality English tokenizer and SBD tool producing DOCREP for text, SGML, and HTML, NLP researchers and developers can easily access document structure and annotations throughout their NLP pipelines.

Chapter 8 demonstrates the use of DOCREP primarily as a consumer of linguistic annotations and document structure. In this chapter, we present a new named entity recognition (NER) system which harnesses document structure information. Our presented document-level features allow us to achieve state-of-the-art results on multiple NER datasets.
Finally, Chapter 9 summarises and reflects upon the contributions presented in this thesis, and provides an outline of the future work.

DOCREP is a software framework and API designed to be used by NLP researchers and application developers, not directly by end users such as annotators. It is designed to be language and operating system agnostic. As such, it deliberately does not supply components such as annotation tools or a graphical user interface. We expect that developers of annotation tools etc. will use the DOCREP APIs to add DOCREP support to their existing tools, in the process adding the ability to represent the annotations directly with respect to the original byte offsets in the original document encoding.

The applications that we discuss in Chapters 7 and 8 are just two examples used to evaluate DOCREP on authentic NLP example tasks, there is no limitations on what NLP tools and components that DOCREP can be used for. We expect that in the future, other tools, for example, the Stanford NLP pipeline, will incorporate DOCREP so that the tools in the pipeline can exploit information about the raw documents, and associate the annotations back to the original document at byte level regardless of the encoding.

1.2 A call to arms

Why use DOCREP?

DOCREP facilitates the reproducibility of results, aids in quality assurance of corpora and their annotations, and promotes the reuse, substitutability, and extrinsic evaluation of NLP components. All of these are problematic areas within our field and we implore the community to reconsider practices in this regard. We have provided the DOCREP DRF, a lightweight, easy to use, programming language agnostic DRF for storing multiple linguistic annotation layers on documents; and a tokenization and SBD framework for importing corpora and their document structure into DOCREP. Additionally, we have shown that the use of document structure information can improve state-of-the-art performance on NER, a well-studied task in NLP.
1.3 Summary of contributions

Even if an NLP tool does not utilise DOCREP models and treats DOCREP simply as an I/O black box, by allowing the tool to consume and produce DOCREP annotations, it provides all downstream applications access to any annotation layers present on the documents. Once a tool in a pipeline decides to communicate via plain text markup formats, this rich structured information is lost for all components further downstream. Even if DOCREP is only used to pass a structured representation of documents and their sentences from one NLP component to another, this is a vast improvement to the current state of NLP pipelines.

In addition to NLP and LT application developers, we also encourage the creators of corpora to consider distributing their annotations in DOCREP. Doing so enhances reproducibility of results and provides quality assurance of corpus modifications — annotations are provided as runtime objects rather than as a text format which developers are required to parse and interpret. Our experiments in Chapter 5 show that DOCREP is a superior representation for the OntoNotes 5 corpus in terms of ease of use and quality assurance.

1.3 Summary of contributions

• Chapter 3 presents an abstract and pragmatic set of design criteria for a document representation framework to store and manipulate modern corpora, building upon existing work in the literature; going beyond theoretical models to efficient, practical applications.

• Chapter 4 provides DOCREP definitions for our outlined design requirements, and the definition and implementation of a serialisation protocol for DOCREP. This chapter provides Python, C++, and Java implementations of our DOCREP API and also presents DOCREP decorators for performing runtime denormalisation.
• Chapter 5 evaluates DOCREP’s ability to model a linguistically-rich corpus, demonstrating that DOCREP can do so, and also outperforms existing approaches.

• Chapter 6 evaluates DOCREP’s usability through use case analysis and testimonials from DOCREP users in the NLP and LT application community.

• Chapter 7 presents a new document-aware tokenization framework. This framework maintains byte and Unicode offsets of its produced tokens and sentence bounds, throughout document structure interpretation, transcoding, tokenization, and SBD. This approach to maintaining offset information throughout these precursor stages is a novel contribution of this thesis. An additional contribution is the implementation of a high quality English tokenizer and sentence boundary detector.

• Chapter 8 presents a new named entity recognizer which harnesses document-level information provided by DOCREP. This new document-aware NER system is a contribution of this thesis. The document structure features we present allow us to achieve a new state-of-the-art result for the standard CoNLL 2003 NER dataset, demonstrating that downstream NLP applications can benefit from document structure information. This system has only scratched the surface of possible document-aware features for NER.

The DOCREP APIs, the tokenization framework, and the NER system are all released as open source and are available under the MIT licence.

1.4 Publications and collaboration

Assorted parts of Chapters 3 through 6 appear in docrep: A lightweight and efficient document representation framework (Dawborn and Curran, 2014). This was the initial release publication of the developed DOCREP document representation framework, and as such, presents a very abridged version of the design and implementation
details of DOCREP, as well as the presented representation and usability evaluations. Parts of Chapter 6 also appear in *Command-line utilities for managing and exploring annotated corpora* (Nothman et al., 2014). While not the first author, Joel Nothman and I collaborated on this paper and the development of the DOCREP tools discussed in Section 6.2, which is the focus of this paper.

Much of the initial brainstorming and prototyping of the DOCREP decorators concept discussed in Section 4.4.3 was also performed by Joel Nothman as a fellow DOCREP beta tester and user within our research group.
2 Background

In this chapter, we provide background on the way in which linguistic annotations are represented and what tools computational linguists have at their disposal to work with them. In Section 2.1 we describe the two main methods for placing linguistic annotations onto documents. Section 2.2 goes on to describe linguistic annotation standards that have been developed, and discusses their adoption within the NLP community. Section 2.3 introduces the concept of a document representation framework (DRF); a core concept of this thesis. This section describes existing DRFs, and outlines their advantages, disadvantages, and limitations. Section 2.4 discusses the uptake of linguistic annotation standards and DRFs within the NLP community. Section 2.5 describes the ways in which NLP researchers typically interact with linguistic annotations. Section 2.6 and Section 2.7 outline the way in which disjoint NLP systems can exchange structured linguistic information, and how these technologies have been utilised in web-based NLP services.

The annotation of corpora is the process of adding linguistic information to a language resource such as text, speech transcriptions, images, audio or video clips, etc. There are many different ways to classify linguistic annotation, but one common high-level distinction is segmentation versus labelling. Segmentation involves delimiting

Dream Theater is an American progressive metal/rock band formed in 1985 under the name Majesty (until 1986).

Figure 2.1: An example raw sentence without any linguistic annotation.
the document into linguistic elements, including continuous segments, super- and sub-segments, discontinuous segments (linked continuous segments), and landmarks (e.g. timestamps). Common examples of segmentation in text documents is tokenization and sentence boundary detection, where the document is broken up into linguistic tokens, after which, sentence boundaries are identified. *Labelling* annotates the segments with linguistic information, such as marking segments as tokens or sentences, or adding POS tag information to tokens. In this thesis, we focus on text documents, but the approaches we describe generally work across most language resources. Figure 2.1 shows a small example document. Segmentation involves breaking this document into sentences and tokens, after which linguistic information could be annotated on the segments.

### 2.1 Representing annotations

There are two main ways in which annotations can be added to a document: inline or stand-off. *Inline annotation* consists of altering the original document (which is almost always plain text) by inserting segmentation and linguistic information in-place. Figure 2.2 shows an example of a parse tree annotation over Figure 2.1. Here, the original document has been modified, with the text being segmented into tokens and a LISP-style bracketing inserted to represent the parse tree structure. As well as the bracketing structure, internal parse node have labels associated with them, and the leaves of the tree have POS tags.

*Stand-off annotation* (Ide, 1994; Thompson and McKelvie, 1997) consists of creating a new document, separate to the original document being annotated, in which the annotations are stored. The annotations in this document point back to the original document by some notion of offsets, normally byte or Unicode code point offsets from the beginning of the document. Figure 2.3 shows an example stand-off annotation version of Figure 2.2.
2.1. Representing annotations

Figure 2.2: The example from Figure 2.1 with inline annotations representing both segmentation (tokens) and linguistic information (POS tags and a parse tree node labels). The LISP-style bracketing denotes the nested structure of the parse tree.

The markup used here was made up for ease of readability, but is similar to many existing stand-off annotation markup schemes. The <tok> nodes define token segmentations in terms of byte offsets over the original document. The nested nature of this SGML-style markup represents the parse tree structure, with attributes used to store the additional linguistic information. <leaf> nodes refer back to the token node they map to via the ref attribute referring to the ID of a <tok> node.

There are many disadvantages of using inline annotations compared to stand-off annotations. As the annotations are stored inside the original document, an appropriate level of escaping is required to unambiguously parse the inline annotations. An example of this escaping is the open and close parenthesis characters in Figure 2.1 that get translated into escape values -LRB- and -RRB- in Figure 2.2, highlighted in orange. If this escaping was not performed, the LISP-style bracketing could not be unambiguously parsed as a parenthesis could be interpreted as linguistic structure or as linguistic content. This particular mapping has caused countless small bugs in NLP systems where models trained on text containing -LRB- and -RRB- have been used on unescaped text, and visa versa.

Another disadvantage to inline annotations is that overlapping annotation layers normally cannot be represented. Easily representing the parse tree derivations pro-
<doc ref="raw.txt">
  <tokens>
    <tok id="t0" begin="0" end="5" />
    <tok id="t1" begin="6" end="16" />
    <tok id="t2" begin="17" end="28" />
    <tok id="t3" begin="29" end="37" />
    <tok id="t4" begin="38" end="47" />
    <tok id="t5" begin="48" end="56" />
    <tok id="t6" begin="57" end="66" />
    <tok id="t7" begin="67" end="76" />
    <tok id="t8" begin="77" end="87" />
    <tok id="t9" begin="88" end="103" />
    <tok id="t10" begin="104" end="125" />
    <tok id="t11" begin="126" end="136" />
    <tok id="t12" begin="137" end="146" />
    <tok id="t13" begin="147" end="157" />
    <tok id="t14" begin="158" end="166" />
    <tok id="t15" begin="167" end="171" />
    <tok id="t16" begin="172" end="183" />
    <tok id="t17" begin="184" end="189" />
    <tok id="t18" begin="190" end="200" />
    <tok id="t19" begin="201" end="211" />
  </tokens>
  <parseTree>
    <node label="S">
      <node label="NP">
        <leaf pos="NNP" ref="t0" />
        <leaf pos="NNP" ref="t1" />
      </node>
      <node label="VP">
        <leaf pos="VBZ" ref="t2" />
        <node label="NP">
          <node label="DT">
            <leaf pos="DT" ref="t3" />
          </node>
          <node label="JJ">
            <leaf pos="JJ" ref="t4" />
          </node>
          <leaf pos="JJ" ref="t5" />
          <leaf pos="NN" ref="t6" />
        </node>
      </node>
    </node>
    <node label="VP">
      <leaf pos="VBN" ref="t7" />
    </node>
    <node label="PP">
      <leaf pos="IN" ref="t8" />
      <node label="NP">
        <node label="CD">
          <leaf pos="CD" ref="t9" />
        </node>
      </node>
    </node>
    <node label="PP">
      <leaf pos="IN" ref="t10" />
      <node label="NP">
        <leaf pos="IN" ref="t11" />
      </node>
    </node>
    <node label="PRN">
      <leaf pos="(" ref="t12" />
      <node label="PP">
        <leaf pos="IN" ref="t13" />
      </node>
      <node label="PRN">
        <leaf pos=")" ref="t14" />
      </node>
    </node>
    <node label="NP">
      <leaf pos="DT" ref="t15" />
    </node>
    <node label="NP">
      <leaf pos="DT" ref="t16" />
    </node>
    <node label="NP">
      <leaf pos="CD" ref="t17" />
    </node>
    <node label="NP">
      <leaf pos="." ref="t18" />
    </node>
  </parseTree>
</doc>

Figure 2.3: The example sentence from Figure 2.1 with stand-off annotations denoting
tokenization, POS tags, and a parse tree.
2.1. Representing annotations

duced by two different parsers for the same document is impossible using the notation shown in Figure 2.2. An alternative bracketing of the tokens cannot be inserted in an unambiguous manner using the same parenthesised markup notation. One might agree that a different markup notation could be used for a second parse tree, such as square brackets instead of rounded brackets for the parse tree structure. This approach does not solve the underlying problem of the original document being modified and requires an unlimited number of different escaping procedures for each additional annotation layer placed over the document.

Stand-off annotations mostly solve both of these issues. Multiple incompatible annotations can easily be added without conflicts as the original document is not modified. Depending on how the linguistic is represented in the stand-off annotation storage layer, some escaping may be required, but this is normally a lot less than an inline annotation format. One disadvantage of stand-off annotations is that their serialisation is normally larger in size than an equivalent inline annotation representation. This is demonstrated by the relative sizes of Figures 2.2 and 2.3. With corpora constantly increasing in size, having the annotation layer serialisations exceed the size of the original documents will hinder the processing of large web-scale corpora.

Despite the problems with inline annotation formats, corpora have typically been distributed with inline annotations. One of the reasons for this is readability. It is significantly easier to get the overall picture of what the parse tree and POS tags look like in Figure 2.2 than in Figure 2.3. This is partially due to the token strings being inside the tree structure, providing the reader with a complete overall picture of the parse, and also due to there being no additional data in the file to distract the reading. Reading through the stand-off representation, the reader must interpret the SGML-style annotations, which adds significant cognitive load.
2.2 Annotation standards

Historically, the creation of corpora has often been paired with the creation of a new annotation file format to suit the needs of the new corpus. This has lead to a broad range of ad hoc file formats throughout the community. The diversity of formats causes an increase in the engineering effort required by consumers of these corpora, as custom I/O code is required for each new file format. Additionally, cross-corpora comparison of annotations becomes more difficult without consistency on annotation structure or attributes.

A number of attempts have been made to create standardised annotation formats, as well as create abstract annotation pivot formats. Pivot formats are designed to facilitate the translation (sometimes referred to as transliteration in the literature) of one annotation format to another format via a common intermediate format — the pivot. In this section, we outline some of the more notable attempts at annotation standardisation and pivot formats.

2.2.1 Corpus Encoding Standard

One of the first attempts at creating such a standardised corpus annotation format was the Corpus Encoding Standard (CES; Ide, 1994, 1998a,b). This work was done in conjunction with MULTITEXT and the Expert Advisory Group on Language Engineering Standards (EAGLES); a direct result of their 1996 recommendations report on the syntactic annotation of corpora (Leech et al., 1996). CES uses a Standard Generalised Markup Language (SGML; ISO8879, 1986) representation conforming to the Text Encoding Initiative (TEI) guidelines (Sperberg-McQueen and Burnard, 1994), and provides encoding conventions for linguistic corpora. CES was designed to be a “more practical format” for language engineering, while being linguistic theory and tagset independent. By more practical, Ide meant that the annotation format provides increased processability, validatability, and consistency over the original annotation format, as these properties...
2.2. Annotation standards

Figure 2.4: An example XCES stand-off annotation. Stand-off is achieved through the combination of XLink, Xpointer, and XPath. Taken from Ide et al. (2000).

References to other annotation (e.g. the canonical mention in coreference resolution) are achieved through Xpointer\(^1\) queries.

Ide et al. (2000) introduce XCES — an Extensible Markup Language\(^2\) (XML) instantiation of CES. XCES is based on the same data architecture as CES, with the original document and the linguistic annotations stored in separate stand-off files. The move from SGML to XML was motivated by the more powerful transformation and querying mechanisms provided by XML-based technologies, such as XPath\(^3\), XLink\(^4\), and Extensible Stylesheet Language Transformations (XSLT).\(^5\) Once an existing annotation format has been translated into XCES, an XSLT script can be used to export the annotations into a non-XML format.

Figure 2.4 shows an example XCES stand-off annotation of a token. Here, an Xpointer annotation is used in conjunction with an XPath query in an XLink attribute to specify a stand-off annotation over a different document, referenced via a URL. Figure 2.5 shows an example sentence in Penn Treebank (PTB) annotation format and a corresponding XCES stand-off annotation file. The tree structure of the original PTB-style annotation is represented by nested XML nodes in the stand-off annotation, with each node pointing back to its location in the original document via XLink and Xpointer queries.

Ide and Romary (2001) extends upon Ide et al. (2000), incorporating the recommendations from the EAGLES report (Leech et al., 1996), to include data category registry

\(^1\)http://www.w3.org/TR/WD-xptr  
\(^2\)http://www.w3.org/TR/REC-xml/  
\(^3\)http://www.w3.org/TR/xpath/  
\(^4\)http://www.w3.org/TR/xlink/  
\(^5\)http://www.w3.org/TR/xslt20/
Dream Theater is an American progressive metal/rock band formed in 1985 under the name Majesty (until 1986).

(a) The raw example sentence.

((S (NP-SBJ-1 Jones)
 (VP followed)
 (NP him)
 (PP-DIR into
 (NP the front room))
 ,
 (S-ADV (NP-SBJ +1)
 (VP closing
 (NP the door)
 (PP behind
 (NP him))))))

(b) PTB annotation of the sentence.

<struct id="s0" type="S"
 <struct id="s1" type="NP" rel="SBJ"
 xlink:href="xptr(/p/s[1]/text(),1,5)"
 >Jones</struct>
 <struct id="s2" type="VP"
 >followed</struct>
 <struct id="s3" type="NP"
 >him</struct>
 <struct id="s4" type="PP" rel="DIR"
 >the front room</struct>
 ,
 <struct id="s5" type="S-ADV"
 >closing</struct>
 <struct id="s6" type="NP"
 >the door</struct>
 <struct id="s7" type="PP" rel="DIR"
 >behind</struct>
 <struct id="s8" type="NP"
 >him</struct>)

(c) XCES stand-off annotation over the sentence.

Figure 2.5: An example sentence in its original PTB format and a separate stand-off annotation in XCES, annotating the parse tree structure and attributes. Note that the trace node and co-indexation in the PTB annotation are represented by an ID lookup via the ref attribute in XCES (highlighted in orange).
support. The Data Category Registry (DCR) is a centralised inventory of annotation types for syntactic annotation. These types are defined using Resource Description Framework\(^6\) (RDF) descriptions which formalise the hierarchy of types and the attributes they contain. By mapping corpus-specific annotation types into the reference types in the DCR, the authors aim to make explicit the equivalences between corpora.

### 2.2.2 Annotation Graphs

Bird and Liberman (2001) performed one of the first comparative surveys of the existing annotation formats with a focus on annotation formats which handle timestamped linguistic data. This work surveys existing annotation schemes including TIMIT,\(^7\) various LDC broadcast news and speech transcript formats, and the Emu speech database system (Cassidy and Harrington, 1996). The authors conclude that the existing surveyed formats have a common conceptual core, which they call an annotation graph. This common conceptual core is a directed acyclic graph with typed labels on some of the edges and time markers on some of the vertices. These graphs are referred to as Annotation Graphs (AG).

Unlike CES or XCES, the definition of an AG does not include a serialisation format — they are defined as an abstract model only. The formal description of an AG, taken from Bird and Liberman (2001), is as follows. Let \( T \) be a set of types, where each type in \( T \) has a (possibly open) set of contentful elements. The label space \( L \) is the union of all of these sets, with each label being written as a type-content pair. An annotation graph \( G \) over a label set \( L \) and node set \( N \) is a set of triples having the form \( \langle n_1, l, n_2 \rangle \), where \( l \in L \) and \( n_1, n_2 \in N \), which satisfy the following conditions:

1) \( \langle N, \{ \langle n_1, n_2 \rangle | \langle n_1, l, n_2 \rangle \in G \} \rangle \) is a directed acyclic graph;

2) \( \tau : N \rightarrow \mathbb{R} \) is an order preserving map assigning times to some of the nodes.

---

\(^6\)http://www.w3.org/RDF/

\(^7\)https://catalog.ldc.upenn.edu/LDC93S1
There is no requirement that the annotation graphs be connected or rooted, nor that they cover the whole timeline of the linguistic signal they describe. The set of annotation graphs is closed under union, intersection, and relative complement.

Bird and Liberman (2001) define a set of three criteria which they believe should be used when evaluating different annotation standards. These criteria are:

**Generality, specificity, and simplicity** An annotation framework should be sufficiently expressive to encompass all commonly used forms of linguistic annotation. The framework should also be simple and well-defined so researchers can build special-purpose tools for unseen applications.

**Searchability and browsability** Annotations should be conveniently and efficiently searchable, regardless of their size and content.

**Maintainability and durability** Annotations stored in the framework should be easily modifiable as corpora are often corrected or adjusted, especially during creation.

The authors go on to discuss how their proposed annotation graphs satisfy, or at least place the foundations for satisfying, each of these criteria. We believe these evaluation criteria are sound, and provide a good basis upon which to evaluate our own annotation framework.

One criticism of Annotation Graphs has been their inability to represent annotations linked to other annotations, such as in a syntax tree (Ide and Suderman, 2007, 2009). An artefact of this is that multiple annotations cannot be viewed as a single graph in AGs. This restriction is quite limiting for a general purpose annotation standard, and does not entirely satisfy the *generality* evaluation criteria above.

Annotation Graphs have had some concrete uses. The ATLAS documentation representation framework (Section 2.3.1) utilises a concrete instantiation of a generalised AG model to store its linguistic annotations. This generalisation of the graphical model overcomes this annotation linkage problem. Another notable use of AGs has been in conjunction with databases and annotation retrieval. Cassidy and Bird (2000) defined
relational database schemas for annotation storage, mapping annotations defined in AGs and the Emu speech database (Cassidy and Harrington, 1996, 2001) to their corresponding relational form. A query language for efficiently querying annotations stored in AGs has also been developed (Bird et al., 2000a).

2.2.3 ISO SC4 TC37: Terminology and other language resources

In the early 2000’s, the International Organisation for Standardization (ISO) formed a sub-committee (SC4) under Technical Committee 37 (TC37, Terminology and Other Language Resources), devoted to language resource management. The main objective of this sub-committee was to prepare international standards and guidelines for effective language resource management.

One important contribution of this ISO sub-committee was the identification of a set of requirements which any linguistic annotation framework must adhere to if it is to be a general annotation framework. The requirements they identified were summarised by Ide et al. (2003)\(^8\) as:

**Expressive adequacy** The framework must provide means to represent all varieties of linguistic information, and possibly also other types of information. This includes representing the full range of information from the very general to the finest level of granularity.

**Media independence** The framework must be able to handle all potential media types, including text, audio, images, and video.

**Semantic adequacy** Representation structures must have a formal defined semantics, including definitions of logical operations. There must also exist a centralised method of sharing descriptors and information categories.

\(^8\)These are similar to those defined in Ide (1994).
Incrementality  The framework must support various stages of input interpretation and output generation, both during annotation and use. It must also support the merging of existing annotations.

Separability  Complementary to incrementality, it must be easy to separate the annotation layers, filtering out everything but the task at hand.

Uniformity  Representations must utilise the same building blocks and the same methods for combining them.

Openness  The framework must be independent of any linguistic theory.

Extensibility  The framework must provide an API.

Human readability  Representations must be human readable, at least for creating and editing.

Explicitness  Information in the annotation scheme must be explicit — that is, the burden of interpretation should not be left to the processing software.

Consistency  Different mechanisms should not be used to represent the same type of information.

TC37 went on to define a set of requirements for an annotation format data model which can satisfy all of these requirements. These requirements form the basis for the Linguistic Annotation Format, which is covered below.

These requirements are mostly sound, and overlap with many aspects of the evaluation requirements outlined by Bird and Liberman (2001). The requirement for human readability is questionable. If an annotation format provides appropriate tools for visualising and interacting with the annotations, it is not necessary for the underlying serialisation format to be in a human readable form.

Formats such as the PTB format (e.g. Figure 2.2) are very minimal in their markup of linguistic annotations. It is this minimalism, not the fact that there are inline annotations,
2.2. Annotation standards

![Diagram of annotation standards]

that makes the PTB format easily human readable. Formats like XML which often claim human readability do no have this same degree of minimalism. Briefly viewing PTB-formatted annotations is enough to get a general feel for what is going on; the same is not true for XML due to this lack of minimalism even though XML is readable by humans. The viewer is required to skim over a larger quantity of markup before finding the stored content. It is unclear to us that XML or stand-off XML counts as an (easily) human readable format, even though this is what the Linguistic Annotation Format uses for serialisation.

2.2.4 Linguistic Annotation Format

Within TC37, a working group (WG1) was formed to create a ISO-standardised Linguistic Annotation Format (LAF). A number of papers were published while this standard was being developed and finalised, releasing updated information in each subsequent publication. Ide and Romary (2003) and Ide et al. (2003) first introduce the work in progress specification for LAF, which was based on the prior work done with XCES. Ide and Romary (2006) later continues with more technical details about how LAF would be implemented.

The overall design of LAF is based on a few straightforward interacting principals:

**Separation of data and annotations** The original data should be separated off from the annotations within a section which should be considered read-only.

**Separation of user annotation formats and the exchange (“dump”) format** Users provide a mapping function between their own annotation formats and the LAF exchange format.
Chapter 2. Background

(a) Translating without a pivot.  
(b) Translating with a pivot.

Figure 2.7: A pivot data format reduces the number of required conversion scripts from $n^2 - n$ down to $2n$, from Ide and Suderman (2007).

Separation of structure and content in the dump format LAF requires that all annotation information be made explicit in the dump format, so that mapping from the dump format to another format is ensured.

The overall architecture of LAF is shown in Figure 2.6. A mapping process is used to automatically convert annotations from a particular user-defined data format into the LAF dump format. This automatic mapping process should be bidirectional so that the user-defined format in question can ingest new annotations via the dump format.

This core purpose of the central dump (pivot) format in LAF, from which existing annotations are mapped in and out of, is for data exchange and translation. This idea is visualised in Figure 2.7. The illustration shows the translation via the pivot format of data in six different annotation formats, labelled A through F. The existence of such a pivot format means for $n$ different data formats, only $2n$ conversion scripts need to be written for the data to be fully convertible between any of the formats, instead of the $n^2 - n$ conversion scripts otherwise needed. For the pivot to work, an existing annotation scheme must be isomorphic to the LAF abstract model.

The dump format represents an annotation as a directed graph referencing $n$-dimensional regions of the original document, as well as other annotations. Figure 2.8 shows an example of this graphical structure. The nodes of the graphical structure are
2.2. Annotation standards

Figure 2.8: A visualisation of segmentation and annotation graphs in the LAF dump format for the example sentence in Figure 2.1.

virtual, located between each of the characters in the original document. Each layer of arc edges is referred to as a segmentation in LAF, and each segmentation is stored in a separate file called a segmentation document. Primary segmentation documents contains no annotations, but instead serve to identify the base edge set for other annotation graphs build on. In text documents, this primary segmentation is typically token boundaries (the blue arcs in Figure 2.8). The serialisation of this primary segmentation is shown in Figure 2.9. An <edge> node with a unique identifier is instantiated for each segmentation arc, specifying the stand-off character offsets.

Any annotation document can be treated as a virtual segmentation document by another annotation, allowing its graph structure to be treated as a conjugate graph\(^9\) (Harary and Norman, 1960) for another segmentation document to attach to. That is, given a graph \(G\) over the original document, LAF creates an edge graph \(G'\) whose nodes can themselves be annotated, allowing for edges between the edges in \(G\). An example of this is the red annotation layer in Figure 2.8, treating the primary segmentation edges as nodes for its annotations. Figure 2.10 shows the serialisation of a morphosyntactic annotation layer over the primary segmentation. The annotations here refer to the edges in the primary segmentation document. Figure 2.11 shows the serialisation of a phrase structure annotation over the morphosyntactic annotation layer. The annotation in this layer are using the edge graph of the feature structure annotation layer as edge points for its annotation arcs.

\(^9\)The conjugate graph of a undirected graph \(G\) is another graph \(L(G)\) that represents adjacencies between the edges of \(G\).
Figure 2.9: Dump format serialisation of the LAF primary segmentation in Figure 2.8.

Figure 2.10: Dump format serialisation of a feature structure associated with a primary segmentation in Figure 2.9.

Figure 2.11: Dump format serialisation of phrase structure information over multiple arcs in the feature structure annotations in Figure 2.10. More than two targets can be specified for an <edge>, allowing for a hypergraph.
Like XCES, LAF does not provide specifications for the annotation types. Instead, like the XCES architecture, the LAF architecture includes a data category registry (DCR) — a centralised repository of annotation types that external annotation schemas can refer to (Ide and Romary, 2004). The DCR includes both annotation types and attribute values which may be referenced directly by user annotations, or to which a mapping from user-defined types can be defined.

### 2.2.5 Graph Annotation Framework

One limitation of the LAF model is that it is not capable of representing merged sets of annotations as a single graph. Ide and Suderman (2007) introduce an extension to LAF, the Graph Annotation Framework (GRAF). GRAF is an XML serialisation of the generic graph structure of annotations in LAF, allowing merged sets of annotations to be viewed as a graph. Generic graphical representations of annotations have been widely used since their description in Annotation Graphs (Bird and Liberman, 2001).

Within the LAF XML serialisation format, multiple annotation layers are represented independently of one another. One disadvantage of this is that it is not easy to discover when multiple annotation layers have subgraphs in common. There are many NLP applications where this is the case, such as having a parse and NER annotations over a document of tokenised text. In LAF, the tokens would be specified in both the annotation graph for the parser and the NER system. GRAF provides a LAF-compatible dump format serialisation which allows multiple annotation layers to be represented as a single graph.

Figure 2.12 shows an example of two different primary segmentations of the original document being represented in a single graphical structure. A parse tree annotation layer exists for both of the primary segmentations, represented in blue and orange in the figure. In GRAF, both of these alternate annotations are represented, merged, into a single graph, as indicated by the common parent node ADJP. The dashed edges coming from this common parent node labelled role: alt indicate that these edges are
alternative choices for the subtree for this node. This merging step, performed during
GRAF serialisation, minimises the on-disk representation of the annotation layers
while also allowing the identification of common tree fragments between alternative
annotation layers.

Ide and Bunt (2010) and Ide et al. (2011) demonstrate the ability for GRAF to rep-
resent a variety of different forms of linguistic annotations by outlining mappings
between GRAF and a number of existing linguistic data formats, including ISO-TimeML
(ISO24617-1, 2009), PropBank (Palmer et al., 2005), and FrameNet (Baker et al., 1998).

The final ISO specifications for LAF and GRAF were published as ISO24612 (2012).

2.3 Document Representation Frameworks

A document representation framework (DRF) supports the creation of stand-off annotation
layers over collections of documents, with the documents in these collections not
necessarily being homogeneous in nature. As well as supporting the creation of these
stand-off annotation layers, DRFs also provide an API for the user to interact with the
annotation layers. Any kind of NLP pipelining system will need some form of DRF,
albeit well defined or ad hoc. Despite the prevalence of pipelining within NLP, there
is little published work on, or implementations of, DRFs. The three main DRFs that have been developed over the last 15 years are ATLAS, GATE, and UIMA. Both GATE and UIMA are used in various parts of the CL and LT community, but their adoption has not been widespread. This section outlines each of these DRFs, along with their advantages, disadvantages, and limitations.

2.3.1 ATLAS

Architecture and Tools for Linguistic Analysis Systems (ATLAS; Bird et al., 2000b), was arguably the first formally specified DRF. ATLAS aims to provide an architecture for annotation, including a logical data format, an API and toolset, and a persistent data representation. The data model underlying ATLAS is an instantiation and generalisation of the Annotation Graph (AG) model described by Bird and Liberman (2001). Laprun et al. (2002) provide a summary of the ATLAS timeline after its initial release.

ATLAS provides an architecture for facilitating the development of linguistic annotation applications and is structured in three levels: application, logical, and physical. Figure 2.13 shows the overall structure of ATLAS. The physical level defines how data is accessed and where it is stored. The logical level consists of a linguistic formalism and an API. This formalism is a generalisation of AGs to higher dimensions, which...
Chapter 2. Background

(a) The AG object model.  
(b) The ATLAS object model.

Figure 2.14: The Annotation Graph and ATLAS object models, from Bird et al. (2000b).

Figure 2.15: Identified regions in an ATLAS signal, from Laprun et al. (2002).

The ATLAS calls “annotation sets”. Lastly, the application level provides common functionality which users of the system would have to implement themselves otherwise. Applications in this level utilise the public ATLAS API.

Figure 2.14 shows the object model defined by AGs and their generalisation used in ATLAS. In the AG object model, a graph object consists of zero or more arcs, where these arcs specify an identifier, two nodes, a type, and its content. A node is specified by an identifier, a timeline, and an offset into that timeline. Timeline here refers to the abstract notion defined in Bird and Liberman (2001). The main differences in the object model used by ATLAS are the generalisation away from one-dimensional annotations through the Region object, as well as some nomenclature changes such as the underlying document being referred to as a signal. This graph structure generalisation is necessary to represent annotations linked to other annotations, such as in a parse tree.

The ATLAS core object model is quite simple, yet still expressive. The ATLAS annotation process can be broken down into three main steps: 1) the identification of regions of interest in a signal, 2) the association of content with these regions, and
3) the linking of related annotations together. Figure 2.15 shows the first step in this process, identifying and marking the region of interest for an annotation. Anchors are fixed-offset markers into the original signal. A region requires two or more anchors into the signal. An annotation object can then be created for that region as shown in Figure 2.16. Annotations are typed and have key-value content pairs. Once the appropriate annotation objects exist, they can be linked together as Figure 2.17 illustrates.

ATLAS provides two storage options: the ATLAS Interchange Format (AIF), an XML interchange format; and a RDBMS backend for connecting to an ODBC-compliant database. Annotations in AIF are stored in a stand-off manner, facilitating the serialisation of multiple annotation layers over the same graph structure. Figure 2.18 shows an example of an annotation graph in XML. This example shows two different original documents (signals) being used in conjunction as a single annotation graph — a video of someone speaking in sign language, and the corresponding closed caption transcription of the signing. The offset information present in the signal segmentation nodes
Figure 2.18: An example of the ATLAS XML representation for annotation graphs, from Bird et al. (2000b).
(<AG_Node>) is more involved than what is available in the LAF and GRAF primary segmentations. As nodes V0 through V2 show, these offsets can be in richer units than byte offsets. An offset value in a unit of seconds requires interpretation of the video stream, using an appropriate codec in order to map these offsets to a location in the file. The unit of seconds here might make more sense than byte offsets as, depending on the nature of the encoding of the video stream, the data between two points in time might not be contained within a single contiguous region of bytes.

The ATLAS API was implemented in a number of different programming languages. Perl, C++, and Java were all supported from the original specification, allowing ATLAS users to take advantage of its functionality without being restricted to a single programming language. This programming language agnostic API was an important initiative, which was unfortunately not followed by the prominent DRFs released subsequently.

2.3.2 GATE

GATE (Cunningham, 2000, 2002; Cunningham et al., 2011, 2013), a General Architecture for Text Engineering, is a DRF which was first released in 1996 (Cunningham et al., 1997), but was not widely adopted. This was due to a number of factors, including non-extensibility, difficulty of installation, and problematic multilingual support. Learning from design decisions made by ATLAS as well as from their own mistakes, GATE was rewritten, redesign and then re-released in 2002 (Cunningham et al., 2002). GATE is open source and is available for download from their website.\(^\text{10}\)

These days, GATE is written entirely in Java, and is configured by XML files. Like ATLAS, it internally stores its annotations in a graph-based format based on Annotation Graphs (Bird and Liberman, 2001). An annotation graph in GATE consists of a series of arcs, each arc containing a set of key-value data pairs called features.

GATE has a focus on presenting a powerful graphical environment for composing NLP pipelines as well as creating and interacting with linguistic corpora and their

\(^{10}\text{http://gate.ac.uk}\)
Chapter 2. Background

Figure 2.19: A screenshot of the GATE Developer user interface, from Cunningham et al. (2013).

annotation layers. These GUIs maximising the accessibility of the GATE tools, allowing people with less technical backgrounds easier access to language technology applications. The GATE suite of tools has grown over the years to include a desktop application for developers, a collaborative workflow-based web application, an index server, and core Java library, and an architecture to process linguistic annotation. A screenshot of the desktop application, GATE Developer, is shown in Figure 2.19.

Powering the user interfaces, the GATE Embedded library provides a Java API to programatically access the GATE DRF functionality, its provided suite of NLP tools, and corpus processing components. The set of plugins that are integrated with GATE is called CREOLE, a Collection of REusable Objects for Language Engineering. These plugin components are defined as Java Beans bundled with XML configuration files. Figure 2.20 shows a summary of the APIs provided by the GATE Embedded library. These include a finite state transduction language (JAPE, a Java Annotations Pattern Language), pluggable machine learning implementations e.g. Weka (Witten and Frank, 2005), the widely-used ANNIE information extraction system, and many others.
2.3. Document Representation Frameworks

Corpus Layers (LRs)
- Language Resource Layer (LRs)
- DiffOntoIVR
- DocVR
- ANNE
- OBIE
- DepParStrong
- Protege
- WN
- Gaz
- MG4j
- ser

Processing Layers (PRs)
- NE
- Coref
- TEs
- TRs
- POS
- Concept
- Ontol
- DataStore and Index Layers
- XML
- OWL
- MG4j
- ser

Figure 2.20: The GATE Embedded APIs, from Cunningham et al. (2013).

Figure 2.21: The annotation graph structure used in GATE. Edge arcs attach to anchors in the original document, pointed to by the arcs. Arcs contain an annotation type and feature key-value content, such as dependency parse labels as shown here.

Figure 2.21 shows a visualisation of an instantiated GATE graph structure. Documents in GATE can have one or more annotation layers. An annotation layer is organised as a directed acyclic graph. Nodes for annotation layers are placed at particular locations (anchors) in the original document. Anchors in GATE correspond to the concept by the same name in ATLAS. Arcs are directed edges between anchors on which annotations reside. An annotation in GATE is the combination of an annotation type and a set of key-value data pairs called features. The value of a feature can be any Java object implementing the java.io.Serializable interface.
2.3.3 UIMA

UIMA (Ferrucci and Lally, 2004; Götz and Suhre, 2004; Ferrucci et al., 2009), the Unstructured Information Management Architecture, is a framework which aims to provide an interoperability mechanism for tools which process unstructured content. UIMA is a cross-domain DRF suitable for processing data contained in a heterogeneous set of documents. Here, content is considered to be structured when its format presents sufficient information to derive the meaning.

UIMA was originally developed by IBM in 2001 but was later migrated into the Apache Software Foundation in 2006.\textsuperscript{11} The original IBM implementation was written in Java, but upon migration into the Apache Software Foundation, was additionally implemented in C++. OASIS,\textsuperscript{12} the Organisation for the Advancement of Structured Information Standards, is a standards consortium which develops open standards for information management and representation. In 2009, UIMA was approved as an OASIS standard, demonstrating that the UIMA approach well accepted by the community (Ferrucci et al., 2009).

Figure 2.22 shows a high-level overview of UIMA, highlighting roles, interfaces and communications of coarse-grained components which are essential within any unstructured information management applications. One key point which the UIMA authors emphasise, which is also true of GATE and ATLAS, is that these DRFs are abstract enough to support documents in any format, not just text.

UIMA stores its annotations in a structure called the Common Analysis Subsystem (CAS). The CAS is conceptually analogous to the annotation graphs used in ATLAS and GATE. Ide and Suderman (2009) use GRAF (Section 2.2.5) as an interchange format to convert annotations between GATE and UIMA, albeit with some caveats, demonstrating the common underlying conceptual data model.

\textsuperscript{11}http://uima.apache.org/
\textsuperscript{12}http://oasis-open.org
2.3. Document Representation Frameworks

Figure 2.22: UIMA high-level architecture, from Ferrucci and Lally (2004).

Figure 2.23: Conceptual view of the UIMA CAS, from Götz and Suhre (2004).
The CAS handles data exchange between different UIMA components. One kind of UIMA component, the Analysis Engines (AE), receive annotated documents, attaches its own annotations, and yields the potentially mutated document back to UIMA for further processing. UIMA components do not exchange any code; they communicate only the data stored in the CAS. A full technical outline of UIMA and the CAS can be found in the OASIS specification report (Ferrucci et al., 2009).

Figure 2.23 gives the conceptual overview of the CAS. It consists of four components: the original document stored in a read-only manner, the type system, the heap, and the index repository. The CAS can roughly be thought of as a database engine in terms of the functionality and API it provides. A data model defined by a type system corresponds to a database schema, and cursor-like iterators defined by indexes in the index repository provide access to the annotation instances in the heap.

UIMA uses a strict type system, with all annotation types and their allowed attributes being explicitly defined and loaded upon UIMA initialisation. The type system is defined using XML configuration files. To utilise a defined type, the user must convert this XML file into Java or C++ source code by running the jcasgen application. There have been many criticisms of the rigidity of the UIMA type system, and of UIMA itself. These criticisms have often come from the original authors of UIMA (Götz et al., 2014), citing the “stagnation in development regarding UIMA’s core since its initial release while the world around [it] has changed”.

UIMA does not provide any method for altering the type hierarchy at runtime; if the user wishes to add a new type to the system then the UIMA instance needs to be shut down, the XML configuration files altered appropriately, and the Java or C++ source code regenerated. Additionally, annotation instances are not safe across an annotation schema change. That is, if there exists annotation instances of type \( T \) and this type is altered to be \( T' \), these existing instances will fail to load when UIMA is re-initialised. This inability to alter the type system as the needs of an application change hinders rapid prototyping in both research and commercial environments.
UIMA types are defined in a hierarchical manner, with each type having a single parent type. UIMA provides a number of base types such as the `uima.cas.Annotation` type, which most user-defined types derive from. This base `Annotation` type provides the important `begin` and `end` attributes of an annotation span which indicate the start and end character offsets into the original document. These two attributes correspond to anchors in ATLAS and GATE. Like ATLAS and GATE, annotations can have key-value attributes. The values for these attributes can be of the fixed set of types provided by UIMA, or a reference to a (single) annotation instance of a user-defined type. Due to the offline process of converting XML type definitions into machine-generated Java or C++ class definitions, developers do not have control over the corresponding classes. That is, the developer cannot easily alter the class in any way such as adding additional member variables or methods. This inflexibility is frustrating for users of UIMA (Götz et al., 2014) and limits the usability of the annotation objects.

The CAS heap is a private component within UIMA, and is not directly accessible to the user via an API. Annotation instances reside inside the heap structure. All annotation instances are stored in the same heap structure, regardless of their annotation type. Figure 2.24a shows the logical representation of a `NameAnnotation` type, which has a nested complex type within it; an instance of a `CanonicalForm` type. Figure 2.24b
shows how this instance is allocated within the heap. Each of the attributes of the type are stored in sequential order.

The last component of the CAS is the index repository, which provides users with access to the annotation instances within the heap. Indexes are specialised containers which contain references to annotation instances. There are three different forms of index: sorted, set, and bag. A sorted index provides a user-defined sorting over the instances, a set index stores only unique annotation instances, and a bag index stores annotation instances in the order in which they were inserted into the index. To retrieve an index and its iterator, the index repository is queried with a type from the type system, and the appropriate index is returned. One limitation of the index repository is that these iterators are limited to iterating over annotation instances of a single type only. This problem is partially overcome though type inheritance, but this requires significant fore-thought into the design of the inheritance hierarchy as types cannot easily be modified once they are defined.

Unlike ATLAS and GATE, UIMA provides an additional abstraction over the unstructured entity being analysed. UIMA defines an entity as an artefact, of which, there can be one or more representations. Each representation is called a subject of analysis (SOFA). For example, an artefact might represent a conversation between two people, of which there are two SOFAs: an audio recording of the conversation and a transcription of the conversation. The notion of the SOFA allows these two related-yet-different representations of the same entity to be modelled as related to one another in UIMA. This functionality has use cases for multi-modal documents, but it is not apparent how useful or widely-used this abstraction is for text processing or corpus linguistics.

Some of the usability criticisms of UIMA have been partially addressed through the uimaFIT library (Ogren and Bethard, 2009), which provides abstractions and wrappers over parts of UIMA the uimaFIT developers dislike working with. For example, in regular UIMA, the configuration parameters for each Analysis Engine need to be defined in their own top-level XML file loaded upon initialisation. uimaFIT improves
the usability of UIMA by allowing Analysis Engine’s to be configured in code rather than these XML files. That being said, uimaFIT can only hide some of the underlying deficiencies in the UIMA framework rather than fixing them.

2.4 Use of annotation standards and DRFs for corpora

Recently released corpora are beginning to contain more than one annotation layer per document. In 2006, the first version of the OntoNotes corpus was released (Hovy et al., 2006; Pradhan et al., 2013). This corpus is a multilayer, multilingual corpus, consisting of several annotation layers per document across multiple domains in English, Chinese, and Arabic. While this first release of OntoNotes was five years after Bird and Liberman’s initial survey of annotation formats and call for consistency across the field, fourteen years later, OntoNotes still distributes its annotations in plain text formats. Exacerbating the issue, each annotation layer for an OntoNotes document is in a different file format. Hovy et al. (2006) mention they looked at using UIMA for the corpus, but they did not believe it was mature enough at the time.

The MASC corpus (Ide et al., 2010) contains documents with multiple annotation layers per document. The annotations are distributed in multiple formats, including XCES and GRAF. The corpus also comes with Java libraries and type definitions for importing the corpus into GATE and UIMA. 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 using the GRAF API. They found that this process was straightforward — the GRAF API was sufficient to successfully extract the MASC data from GRAF and insert it into the annotation database.

Other than the OntoNotes and MASC corpora, we are not aware of many more recently released multilayered corpora. Ngo et al. (2013) and Mille et al. (2013) both present new multilayered corpora, but both are distributed as multiple flat files instead of using an annotation format such as GRAF or a DRF.
Chapter 2. Background

2.5 Interacting with annotations

Linguistically annotated corpora are present throughout NLP, but the manner in which they are distributed as well as their internal representation varies greatly. This lack of standardisation means that data is often manipulated in an *ad hoc* manner. This requires custom scripts and workflows to be developed which are corpus specific, and can often become a bottleneck in a NLP pipeline. In this section, we aim to provide a broad coverage summary of the ways users interact with structured data, in particular focusing on the formats that have been used for corpus distribution.

2.5.1 Flat files

Corpora are still often distributed as one or more flat files of plain text, with inline annotations used to encode linguistic annotations. Researchers working with textual corpora often need to filter and extract instances from the corpora which match some given condition. This could be for testing a hypothesis, for statistics gathering, or for error analysis. UNIX provides many tools for operating over paths, processes, streams, and textual data. Among them are `wc` to count lines and words, and `grep` to extract segments matched by a regular expression. These two UNIX tools alone, when piped together, accomplish a great deal with minimal development cost. Tools of a similar philosophy exist for other formats. 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 and 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 tools. By standardising formats for these primitives, and providing means to operate over them, DRFs promise savings in development and data management costs.
2.5. Interacting with annotations

DRFs often store annotated corpora using XML. As such, users are free to utilise existing standardised tools for performing basic transformation, filtering, and aggregation over these annotations (e.g. XQuery). Generic XML tools are limited in their ability to exploit the semantics of a particular XML markup 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.

2.5.2 Standardised annotation formats and DRFs

A number of toolkits have emerged for working with Annotation Graphs (Section 2.2.2), facilitating the manipulation, visualisation, and exporting of existing graph structures, as well as the importing of corpora in other formats into AGs (Bird et al., 2001; Cotton and Bird, 2002; Maeda et al., 2002). GRAF provides a Java API\(^{13}\) for serialising and deserialising annotations, but does not provide a set of command-line tools or graphical interfaces specific to GRAF. Being an XML-backed serialisation format, standard XML tools might be able to suit the users needs (e.g. LT-XML).

Both GATE and UIMA provide sophisticated graphical tools for viewing and modifying annotations, for comparing parallel annotations, and for displaying an index of terms across a document collection (Cunningham, 2002; Ferrucci et al., 2009). Both also provide means of profiling the efficiency of processing pipelines. The Eclipse IDE\(^{14}\) serves as a platform for tool delivery and is comparable to the UNIX command-line, albeit graphical, while providing further opportunities for integration. For example, UIMA employs Java Logical Structures to yield corpus inspection within the Eclipse debugger (Ferrucci et al., 2009).

\(^{13}\)http://iso-graf.sourceforge.net/
\(^{14}\)https://eclipse.org/
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 the Lucene search engine\textsuperscript{15} (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.

2.5.3 Querying annotations

The notion of wanting to query a corpus has been explored thoroughly in the literature, with a number of publications focusing on querying treebanks specifically. Ghodke and Bird (2010) provide a good summary of this literature. Implementing ad hoc treebank querying solutions quickly becomes problematic due to the diverse nature of both the structure of, and the information stored within, treebanks. For example, some treebanks use Penn Treebank style bracketing, others store dependency structures on the nodes (Čmejrek et al., 2004), and others store linguistically-rich categorial grammar tags (Hockenmaier and Steedman, 2007). Some treebanks even contain overlapping tree structures (Cassidy and Harrington, 2001; Heid et al., 2004; Volk et al., 2007). Well-known command-line tools such as tgrep\textsuperscript{2} (Rohde, 2005) handle some of these cases, but not all. Some corpora provide a suite of tools to perform such operations (Kloosterman, 2009).

There are many difficult technical and user interface aspects to treebank querying. From the user interface perspective, what kind of query language is required or desired? This depends on what the user wants to do — do they need to search by specific tree fragments, by graphical structure, or by attributes on the nodes? The requirements for such a query language have been explored in depth (Lai and Bird, 2004; Mirovský, 2008). On the technical side, how should the queries be executed and how are the

\textsuperscript{15}https://lucene.apache.org/
2.6. Interchange formats and interoperability

treebanks indexed? A number of proposals have been made to map the query language to existing performance-tuned query systems, such as to SQL (Bird et al., 2006; Nakov et al., 2005), to XQuery (Cassidy, 2002; Mayo et al., 2006), and to finite state automata (Maryns and Kepser, 2009).

Similar query-related problems and solutions have been proposed for general annotation graphs (Bird et al., 2000a; Cassidy and Bird, 2000).

Tools for interacting with structured binary data also exist, such as protostuff\(^\text{16}\) and ProtoBufEditor\(^\text{17}\) for interacting with Protocol Buffers, and msgpack-cli for interacting with MessagePack.\(^\text{18}\) In all of the these cases, these presented utilities aim reduce development effort, with an orientation towards data management and evaluation of arbitrary functions from the command-line.

2.6 Interchange formats and interoperability

Exchanging linguistic information between different NLP components is problematic with many NLP systems using their own custom data formats, and the underlying schemas for these formats not being isomorphic across systems. A number of attempts to solve this problem through a pivot format have been proposed, such as LAF and GRAF. Here we briefly review other formats used for interchange and representation.

Resource Description Framework\(^\text{19}\) (RDF) is a commonly-used metadata model based around subject-predicate-object triples. Collections of RDF records used within the NLP community include DBpedia,\(^\text{20}\) YAGO,\(^\text{21}\) and Freebase,\(^\text{22}\) are often used as part of the backends of knowledge base systems. There is also an inline variant of RDF called RDFa\(^\text{23}\) which has some traction in the semantic web space. Rich query

\(^{16}\)https://code.google.com/p/protostuff/
\(^{17}\)http://sourceforge.net/projects/protobufeditor/
\(^{18}\)http://cli.msgpack.org/
\(^{19}\)http://www.w3.org/RDF/
\(^{20}\)http://dbpedia.org/
\(^{21}\)http://www.mpi-inf.mpg.de/yago-naga/yago/
\(^{22}\)http://www.freebase.com/
\(^{23}\)https://rdfa.info/
languages such as SPARQL\textsuperscript{24} can be used to extract information from these knowledge bases via attribute and relational queries. RDF is often paired with the Web Ontology Language\textsuperscript{25} (OWL).

The NLP Interchange Format (NIF; Hellmann et al., 2013) is an RDF/OWL-based format which aims to achieve interoperability between NLP tools. NIF aims to fulfill the same needs as previously-defined pivot formats such as LAF and GRAF. As a result of being directly based on RDF, linked data, and ontologies, NIF supports useful features such as annotation type inheritance and alternative annotations. NIF has been implemented in a number of existing NLP and DRF systems, including UIMA, ANNIE within GATE, and DBpedia Spotlight. Similar to Cassidy (2008), NIF-aware applications produce output adhering to the NIF core ontology as REST services.

It should be noted here that both LAF/GRAF is isomorphic to RDF, which was an intentional design decision. At the time of development of GRAF, it was believed that RDF was not mature enough to be utilised, and as a result, XML was used instead. A result of this isomorphism is that a transformation from GRAF to RDF is trivial.

EARMARK (Peroni and Vitali, 2009) is another linguistic annotation interchange format similar to NIF. It uses a stand-off annotation format to annotate text with XML markup in RDF. Open Annotation (Sanderson et al., 2013), POWLA (Chiarcos, 2012), and BioC (Comeau et al., 2013) are other linguistic annotation interchange formats which aim to achieve the same goals as NIF and EARMARK.

More recently, JSON-LD\textsuperscript{26} has become a popular format for representing linked linguistic information. A JSON-LD document is both a JSON\textsuperscript{27} document and an RDF document. The lightweight and dynamic nature of JSON has been an attractive aspect of JSON-LD. Many languages and libraries already provide support for serialising and deserialising JSON data, minimising the technical cost of supporting JSON-LD.

\textsuperscript{24}http://www.w3.org/TR/rdf-sparql-query/
\textsuperscript{25}http://www.w3.org/TR/owl-features/
\textsuperscript{26}http://json-ld.org/
\textsuperscript{27}http://json.org/
These interchange format efforts, including LAF, GRAF, NIF, and UIMA’s CAS, all facilitate the syntactic interoperability of NLP systems but do not solve the semantic interoperability problem (Ide and Pustejovsky, 2010). There are some efforts to create annotation type repositories for such interoperability, such as the LAF DCR (Ide and Romary, 2004), ISOcat (Kemps-Snijders et al., 2009), and the NIF core ontology. These repositories aim to provide the same form of canonical type reference that more general schema repositories like Dublin Core\(^{28}\) and schema.org\(^{29}\) provide, except focused linguistic data and annotations. PubAnnotation (Kim and Wang, 2012) is another example of such a repository, from the BioNLP domain.

2.7 NLP corpora and components as a service

There are a number of different web-based services these days which provide access to NLP tools and corpora, as well facilitating their composition to create custom NLP pipelines. We briefly outline some more notable recent examples, highlighting their use of NLP interchange formats, DRFs, and existing linguistic annotation standards.

The Language Application (LAPPS) Grid project (Ide et al., 2014a) is one such service. Built upon previous work such as SILT (Ide et al., 2009) and The Language Grid (Ishida, 2006), the LAPPS Grid aims to provide access to basic NLP processing tools, resources, and corpora, while also facilitating the composition of NLP tools that the user does not have local access to. LAPPS Grid distinguishes itself by orchestrating access to language resources and tools located on research servers around the globe rather than hosting everything itself in a centralised location. This project defined an annotation type vocabulary which is used in conjunction with JSON-LD to facilitate the interchange between of a diverse range of NLP corpora and tools with custom pipelines (Ide et al., 2014b).

\(^{28}\)http://dublincore.org/
\(^{29}\)http://schema.org/
Another system which provides NLP corpora and components as a service is the Alveo Virtual Laboratory (Cassidy et al., 2014). Alveo has similar goals to the LAPPs Grid project but is more centralised in design. Internally, Alveo stores stand-off annotations using the DADA (Cassidy, 2010) RDF model, which was inspired by GRAF. By storing the annotations in an RDF data store, Alveo is able to harness the well-developed querying infrastructure for RDF by providing users the ability to perform SPARQL queries over the annotations. Alveo uses JSON-LD to return the documents and annotations to the user, with annotation types mapped to entries in Dublin Core, OLAC, and some custom namespaces. Alveo also has support for working with UIMA (Estival et al., 2014) and the popular NLTK Python library (Bird et al., 2009).

GATE has also been used in some service-oriented NLP systems, such as the commercial alternative AnnoMarket\(^{30}\), providing an affordable, open marketplace for pay-as-you-go, cloud-based extraction resources and services, in multiple languages.

A non-web-based approach to NLP pipelining is provided by the CURATOR framework. CURATOR (Clarke et al., 2012) provides a cross-language NLP pipeline using Thrift (Slee et al., 2007) to provide cross-language communication and RPC. CURATOR requires a server to coordinate the components within the pipeline.

Other popular NLP systems such as the Stanford CoreNLP pipeline\(^{31}\) (Manning et al., 2014) provide an all-in-one solution, where the pipeline exists within the framework itself. A downside to this approach is that the pipeline exists with a CoreNLP instance and must be run on a single machine. This is quite limiting when wanting to process large volumes of data.

### 2.8 Summary

In this chapter, we have described the state of linguistic annotation representation within NLP and the attempts to unify representations. The annotation standards

\(^{30}\)https://annomarket.eu/

\(^{31}\)http://nlp.stanford.edu/software/corenlp.shtml
introduced in Section 2.2 are suitable to act as a linguistic pivot format, but unfortunately exhibit usability issues which limit their use as the primary distribution format for corpora. Section 2.3 introduced the concept of a document representation framework (DRF) and the three notable DRF implementations: ATLAS, GATE, and UIMA. For each DRF, we outlined how they represent linguistic annotations and any usability issues from a developers perspective. We saw that existing DRFs were very heavy-weight, monolithic frameworks, requiring significant investment from a new user to use such a framework. Additionally, GATE and UIMA are very Java-oriented, restricting their use to applications which happen to be developed in Java. This excludes many high-performance NLP systems.

We conclude that the field is lacking a lightweight, efficient, and modern DRF that is programming language agnostic and easy to learn and use.

This thesis aims to solve this problem. In Chapter 3 we outline the requirements for this new DRF, making explicit the needs the use cases that the current DRFs fail to satisfy. Chapter 4 goes on to describe the implementation of this new DRF. Having an implementation, Chapter 5 and Chapter 6 go on to evaluate this DRF from two different perspectives — its ability to sufficiently model diverse linguistic annotations while doing so efficiently, and the ability for a new user to pick up and use the DRF.

With this DRF implemented and evaluated, showing that it fulfills our design requirements, the next two chapters go onto use this DRF for two different NLP applications. Chapter 7 uses the document structure information provided by our DRF to present a new tokenization framework unlike any other we are aware of, providing structured document offset information throughout whole NLP pipelines. Chapter 8 also uses the document structure provided by our DRF to achieve state of the art NER performance across multiple datasets.
3  \textbf{DOCREP: Design}

Through our experiences as computational linguists, we have worked extensively with
text corpora from a variety of sources, used many different internal and external NLP
tools and frameworks, and composed countless pipelines. We found that a significant
amount of time was dedicated to two things: 1) the handling of the plethora of input and
output formats used by corpora and tools alike; and 2) developing \textit{ad hoc} solutions for
representing whole documents and their annotations across heterogeneous components
coupled together in a single pipeline.

For pipelines containing Java-only components, using an existing DRF such as
GATE and UIMA is an option, but many of our components are not implemented in
Java. Additionally, the strictness and rigidity of the type systems in both GATE and
UIMA impede rapid prototyping patterns often employed in research environments
to test new ideas. These existing DRFs failed to fulfill our requirements and we were
constantly reinventing the wheel in terms of whole-document management across
pipelines. We were after a simple, expressive, programming language agnostic, efficient
DRF which allows for rapid prototyping and scale-out parallelisation.

Talking to other researchers and consumers of language technology, we found that
this is not an uncommon situation. Developers and researchers do not want to have
to spend the time to learn a large monolithic framework when the gain over an \textit{ad hoc}
solution is not apparent. Additionally, arguably the most commonly used DRF, UIMA,
is difficult to use outside of its suggested IDE integration, forcing developers to use a
particular IDE instead of their own desired development environment. We came to
the conclusion that the field lacks a DRF which meets all of these criteria. From this conclusion, our idea for DOCREP was born.

**DOCREP** (/dokrep/), a portmanteau of document representation, is a lightweight, efficient, and modern DRF for NLP systems that is designed to be simple and intuitive to use. We use the term lightweight to contrast DOCREP to the existing DRFs used within the CL and LT community. Our aim is that the overhead of using DOCREP to store and manipulate linguistic annotations is minimal, both in terms of the developers time and required system resources. Little developer effort is required to start using a markup-based flat file format. We want DOCREP to be so convenient and easy to use that a developer would not consider using a markup-based format again.

In this chapter, we outline our design goals and desired use cases we would like to satisfy with DOCREP, as well as why existing DRFs fail to meet these criteria. Chapter 4 goes on to outline specific implementation details and how they map back to our design requirements. With DOCREP implementations in place, the later four chapters go on to evaluate DOCREP against our design goals and against existing DRFs, as well as demonstrating use cases for the document structure information DOCREP provides to NLP pipelines.

Our design goals are broken down into three broad categories: usability requirements, how we would like annotations and the type system to work, and how annotations and documents should be serialised. We motivate our requirements by contrasting them against equivalent concepts in existing DRFs. When discussing design considerations relating to serialisation, we use the term “on the wire” to refer the serialised form of a DOCREP document.

### 3.1 Usability

There are many aspects that make a software framework usable. Identifying usability deficits in existing DRF implementations, we outline here a set of criteria we would
like DOCREP to satisfy. All of these usability goals aim towards the ease of uptake and minimising any impact using DOCREP has on a developers existing workflow or codebase.

3.1.1 Programming language and paradigm agnostic

ATLAS, arguably the first DRF, defined a programming language agnostic API and provided implementations in multiple languages (Section 2.3.1). GATE and UIMA unfortunately did not follow this approach and are heavily Java oriented. When we discuss Java, this also includes other languages which run on the Java Virtual Machine (JVM). While UIMA has a C++ API, it was not developed at the same as the Java API. The C++ API has not been updated since 2012 and is lacking many of the core features of its Java counterpart, which has been constantly under development. As such, we believe that the UIMA C++ API is seen as a second-class citizen in the UIMA world, a view which is reaffirmed by its lack of documentation.

NLP tools should be developed in the language that best suits the task at hand. There are a number of popular NLP tools implemented in C, C++, and Python which cannot easily be integrated with GATE and UIMA due to their choice of programming language; for example, the C&C parser (Clark and Curran, 2007), NLTK (Bird et al., 2009), and TurboParser (Martins et al., 2010). We do not want DOCREP to suffer from this same inflexibility. At a minimum, we want to provide DOCREP APIs in the major programming languages used in NLP: C++, Python, and Java. These three languages vary greatly in how they operate, so a language and paradigm agnostic design principle is important from conception.

3.1.2 Programming language constraints and idioms

Programming language and paradigm agnostic design impacts a number of decisions, such as the serialisation and deserialisation methodologies and formats available, the use of object-oriented inheritance capabilities, what primitive data types exist,
etc. This also affects runtime decisions, especially how available data types and their characteristics in each language affect the inter-language portability.

For example, in C++, strings are simple sequences of bytes so most Unicode-aware applications will use UTF-8 as it is a widely adopted 8-bit encoding with ASCII backwards compatibility. In Java and the JVM, strings are UTF-16 encodings of Unicode code points. In Python, the internal string representation varies between operating system and major version number, and could be UCS-4 strings, UTF-16 strings, UTF-8 strings, or even UCS-2 strings. We want DOCREP documents to be readable and writable anywhere, so an appropriate string representation needs to be used for serialisation so the conversion between strings on the wire and native strings is efficient across languages.

Part of what makes a library easy for a developer to use is how much it adheres to language idioms and conventions. The built-in Python unittest module\(^1\) is an example of an API that does not feel idiomatic to work with within the language in question (Python). The API for the module was originally copied from JUnit,\(^2\) a Java unit testing framework. The API was copied without respect to the naming conventions used throughout the rest of the Python standard library, making working with this library feel out of place for a Python developer. C++, Python, and Java all have different conventions and idioms, and the way developers interact with libraries is different in each. Ideally, we would like the DOCREP API to look and feel consistent across languages, but also feel idiomatic to each language. We want DOCREP to be easy and enjoyable to use as a software engineer, and act as a useful black box as a computational linguist.

\(^1\)https://docs.python.org/3/library/unittest.html
\(^2\)http://junit.org/
3.1.3 Low cost of entry

Our aim is that the overhead of learning to use DOCREP to store and manipulate linguistic annotations instead of a markup-based flat file format is minimal. We want DOCREP to be something a developer would use even for a small side project. This implies a very low effort to use the framework relative to the gain it provides. All aspects of using a framework are included in this effort cost: installation, vocabulary size, infrastructure requirements, development workflow requirements, etc.

One dimension of the low cost of entry requirement is how easy is the framework to pick up and use. GATE and UIMA have developed over time into large monolithic frameworks, resulting in a large cost of entry requirement for a new user. For example, the PDF user manuals for GATE and UIMA are roughly 700 and 400 pages respectively. If a new user was surveying the space of DRFs and was confronted with a document that large and the long-term gain of using the framework was not obvious, the user is likely to not pursue further investigation of the framework.

Ideally, the DOCREP installation process should be as simple as adding the DOCREP package as a dependency in the user’s package manager. From there, the user should be able to jump straight in and use DOCREP irrespective of any existing development workflow. This implies no non-package managed external dependencies, and no configuration requirements from the users perspective. Similarly, adding DOCREP to an existing codebase should not alter the way in which the codebase needs to be built or executed.

3.1.4 Lightweight, fast, and resource efficient

Following a UNIX design philosophy, we believe that a DRF should do one thing and do it well. The purpose of a DRF is to facilitate the management of multiple annotation layers over a document, allowing the user to interact with the annotations and providing ways to serialise and deserialise the documents it manages. Adding a DRF to an existing
pipeline should not change the resource requirements for the pipeline. A DRF should be space and time efficient, with the increase in memory usage and execution time being negligible when using a particular DRF compared to not using it. This entails an efficient implementation of the serialisation and deserialisation process so that I/O and object instantiation costs are negligible.

Additionally, a DRF should not hinder the use of parallel execution. Given a corpus of documents, NLP pipelines are often *embarrassingly parallel* at document-level granularity — a corpus of documents going into a pipeline often has the same process executed on each document independently of the other documents. A DRF should support exploiting parallelism for local or non-local execution. In fact, if possible, a DRF should make it as easy as possible for the user to distribute their work across a cluster of compute nodes and collate the results back at the end. These distribution and collation processes should not become bottlenecks or be resource intensive, otherwise the runtime gains produced by parallel execution might be lost.

### 3.1.5 No IDE required

NLP researchers who work with text corpora are often savvy users of UNIX tool, in part due to the common operations that need to be performed when working with text. For example, combining UNIX tools such as `grep` and `wc` together via a UNIX pipe to perform a filter and a count operation for statistics gathering or error analysis is very common (Church, 1994; Brew and Moens, 2002). If these researchers are already familiar with these kinds of command-line tools which harness the “chainable specific tools” philosophy, why not make use of this common background knowledge?

We would like DOCREP to have a rich set of command-line utilities for inspecting, manipulating, and transforming DOCREP streams. We would like these commands to feel as natural and idiomatic as possible to an experienced UNIX user. Ideally, the user would be able to (almost) use these tools out of the box without reading
the documentation by following standard UNIX tool idioms and conventions, again emphasising our goals for ease of use and low cost of entry.

One pragmatic difference between these proposed tools and UNIX tools is the level at which they operate. DOCREP, being a DRF, works with documents as its coarsest level of granularity. UNIX tools typically operate over one line of text at a time. This difference will potentially affect what kind of operations can be adequately expressed on the command-line as opposed to writing code.

## 3.2 Annotations, documents, and the type system

Figure 3.1 presents an example schema we wish to model in a DRF. Throughout this section we will refer back to this example schema, describing how existing DRFs model and handle certain situations. Additionally, we will describe what we would like DOCREP do to in those situations (instead). DOCREP introduces some new modelling concepts which do not exist in GATE or UIMA. We will also discuss where annotation instances reside.

Here we will give a brief description of the example model shown in Figure 3.1. The `Document` type contains a `(doc_id)` attribute, representing an identifier string for that document. The `Token` annotation type contains the raw underlying string of the token, as well as the start and end offset of this token into the original document. This is needed for implementing stand-off annotations, and is used by all existing DRFs (Section 2.3). The `ParseNode` annotation type has a label string and references to its child nodes. We use the term reference here in a programming language agnostic sense; this attribute refers to zero or more other parse node instances. The `ParseNode` also contains a reference to the corresponding `Token` object if the parse node is a leaf node. The `Sentence` annotation type is an annotation over sequential `Token` objects. Additionally, it has a reference to the root node of a gold and automatic parse for the sentence.
Figure 3.1: An example schema to be modelled in a DRF.

DOCREP does not have any predefined type system for annotation types. All definitions of the type system are up to the user and are defined per document. Ideally, if DOCREP is designed correctly, a higher level tool could be built on top of DOCREP to provide canonicalisation of annotation types into an existing ontology or type system repository. This approach is quite different to the main existing DRFs, such as GATE and UIMA.

3.2.1 Annotation types as classes

In existing DRFs, the description of annotation types is separated from their use. External configuration files (often XML) are used to describe annotation types and their attributes. There are two main approaches to the way these annotation types are accessed at runtime. The first is that annotation types and their instantiations are accessed via a generic typeless interface, where the user works with strings and generic mapping structures instead of working with first-class types. For example, Figure 3.2 shows a code snippet from the GATE manual.\(^3\) This code snippet shows how the user is meant to obtain all of the Person annotation objects in the order they appear in the document.

\(^3\)https://gate.ac.uk/sale/tao/splitch7.html#x11-1720007.4.3
3.2. Annotations, documents, and the type system

Figure 3.2: A code snippet from the GATE documentation showing how annotations of a given type are retrieved and iterated through.

The second approach is to generate programming language specific source code from the annotation type descriptions, which the user then imports into their codebase. This approach allows for annotation types to be first-class types in the programming language and is the approach taken by UIMA. The major disadvantage of this approach is that the user has no control of these machine-generated classes. If the annotation type definition changes, the source code needs to be regenerated to provide a complete definition of the class. This means that the user is not free to easily add additional member variables or methods to the class, limiting the usability of objects of these classes. These classes become, in essence, a typed interface between the developer and the UIMA serialisation and deserialisation process.

Neither of these approaches satisfy our expectations for usability. Working with typeless object interfaces is undesirable from an elegance, type safety, efficiency, and debugging perspective. Additionally, not allowing the user to work with DRF-based classes in the same way as normal classes is very cumbersome.

We want DOCREP annotation types to be mapped to first-class types in each programming language, but without the need for external files. The DOCREP API in each language should provide a way to “decorate” class and member variable definitions with DOCREP attributes so that the user is free to work with this class as a regular class. These decorations would allow DOCREP to know which classes are annotation types.
and which members of the class it should be aware of for serialisation purposes. We
would like DOCREP to be supported in languages without object-oriented capabilities.
In theory, annotation types and their attributes can be mapped to any form of record
structure that exists in a programming language, such as structs in C. Our description
of annotation types and their mappings to classes can equally be applied to vanilla
record structures.

3.2.2 Distinct, non-hierarchical types

The type system used to define annotation types in existing DRFs often supports in-
heritance between annotation types, a notion which has the same semantics as the
object-oriented programming concept with the same name. For example, in UIMA,
all annotation types must have a single parent type and must be a direct or indirect
descendant of the uima.cas.TOP type. In UIMA, obtaining an iterator to iterate over
annotation instances of more than one type can only be achieved through class inheri-
tance — annotation instances which match an instanceof comparison are yielded. As
discussed earlier, this forced single inheritance hierarchy is limiting when combined
with the inability to change the type system (Section 2.3.3).

There are two alternatives to a single inheritance hierarchy: no inheritance or
multiple inheritance. Other than the canonical textbook issues associated with multiple
inheritance (Cargill, 1991; Waldo, 1991), a limiting factor with this approach is that
not all programming languages with object-oriented capabilities support multiple
inheritance, e.g. Java. This does not fit with our goal for DOCREP annotation types to
map directly to runtime classes (Section 3.2.1).

No type inheritance is the simplest to implement and allows DOCREP to be used in
languages without object-oriented capabilities. When combined with our design goals
of annotation types mapping to classes (Section 3.2.1) and runtime-configurable type
mapping (Section 3.2.3), a no-inheritance approach would not forbid the user from
defining DOCREP-decorated classes which exist within an inheritance hierarchy. As
such, we would like DOCREP to not support annotation type inheritance as part of its type system, but to also not forbid runtime classes from existing within an inheritance hierarchy.

### 3.2.3 Under-specified type system

In many NLP applications which take linguistically annotated documents as input, not all of the annotation layers are used for the particular task at hand. For example, a NER system may only be interested in token and sentence annotation types present on a document with multiple annotation layers. Particular applications may also not be interested in some of the fields of annotation layers. For example, a named entity linking (NEL) application may only care about the NE category; the probability that the NER system provided as its confidence in the category may not be of interest.

Sticking to our desire for DOCREP to be as lightweight and efficient as possible, we want to exploit the fact that DOCREP consumers will frequently only use a particular subset of the annotations on the document. Ideally, we would like DOCREP APIs to be able to ignore the annotations types and the attributes on annotations that the application is not interested in, doing so in the most efficient way possible. Upon re-serialisation, these ignored values could be written back out verbatim since they were not changed during the execution of the current application. Additionally, the annotation types and annotation attributes defined in the application which are not present on a read-in document should be appended during re-serialisation. This results in the output document containing the union of the original schema and the application-defined schema. This functionality is something that neither GATE nor UIMA provide. In DOCREP terminology, this process of ignoring annotations and their attributes which the application does not care about is known as *laziness*, or specifically, or lazy serialisation.
3.2.4 Representation of annotation spans as first-class citizens

There are two different notions of an annotation span in Figure 3.1, indicated by begin and end attributes. The first is on the Token type, where token objects store their begin and offset into the original document. The exact semantics of this offset is defined by the DRF used, e.g. these values are character offsets in UIMA. Offset spans are needed to be able to project the stand-off annotation back onto the original document.

The second kind of annotation span is on the Sentence type, where a sentence object locally spans over a sequential block of token objects. While this is logically how the sentence to token relationship is modelled, it is not how the relationship needs to be represented in GATE or UIMA. In these DRFs, this is not modelled as a reference to the beginning and end token objects on the sentence object, as is shown in Figure 3.1. Instead, for a sentence, the user must store the begin attribute of the first token the sentence spans as its document offset begin value, and the end attribute of the last token the sentence spans as its document offset end value. This is partially an implementation detail that has been propagated to the data model.

An advantage of this approach is that querying annotation objects which exist between two offsets on the original document can be performed directly from the serialised representation. The main disadvantage is the lack of the concept of an annotation span in the data model. When an annotation logically spans over a sequence of annotations, such as a sentence spanning over a sequence of tokens, the majority use case of this relationship is to iterate over the spanned annotations. Requiring the user to perform an indirection step to obtain the spanned annotations (e.g. the index repository in UIMA) is highly undesirable computationally and for ease of use.

DOCREP should support annotations spans as first-class citizens in its type system. Both spans over the original document and spans over sequential annotation objects should be able to be represented with the same concept. There are two main benefits to this approach. First is the aforementioned data modelling advantage. The begin
and end pair are now modelled specifically as an annotation span where there is no room for ambiguity in the interpretation of its semantics. Second is the potential runtime API advantage. If this notion of an annotation span is promoted to a first-class type in the runtime type system, language constructs such as “foreach loops” could harness the knowledge that the underlying data represents a beginning and end pair of values over some sequence, and facilitate the iteration between them. This results in an elegant API which clearly illustrates the annotation span semantics of the attributes, as demonstrated below:

```java
Sentence sentence = ...
for (Token token : sentence.span) {
    ...
}
```

The querying scenario described earlier can still be achieved via the propagation of token offset information through the object references at runtime to create an index structure for querying. This process could be performed during deserialisation when all of the annotation instances need to be inspected anyway.

### 3.2.5 Shared heaps versus type-specific heaps

In existing DRFs, all annotation instances are stored at rest (on disk) and in memory in one location, often referred to as the heap. To extract the instances of a particular annotation type from the heap, the type is provided to the heap manager and the appropriate instances are returned. In UIMA, the index repository provides this interface between the user and the heap. From a usability perspective, this process of querying a centralised location of annotation instances feels unidiomatic as the annotation instances are objects of first-class types residing in local memory. Why not provide type-specific heaps as standard user-accessible data structures?

We would like DOCREP to use type-specific heaps rather than a single shared heap for a number of reasons. The main reason is the power this provides when used in conjunction with lazy serialisation. Lazy serialisation (Section 3.2.3) is the concept of ignoring existing annotations and annotation attributes that the application is not
aware of. If each annotation type has its own heap and entire heaps are serialised together on the wire, all annotation instances of a given type can be lazily ignored if the application does not know about that type. This provides a powerful mechanism for efficient lazy deserialisation.

We want DOCREP to go one step further than this. Why restrict heaps to be one per type? Often this is what the user will need and want; to be able to iterate through all annotations of a given annotation type. However, harnessing this new-found power provided by lazy serialisation, we can separate logical groups of annotation instances of the same annotation type into separate heaps. For example, imagine the user was evaluating a number of different parsers, all of which produced DOCREP parse trees of the same format — the ParseNode type shown in Figure 3.1. This should be modelled in DOCREP as multiple type-specific heaps; one per group of parse nodes for each parser. That is, there would be a gold_parse_nodes heap, an auto_parse_nodes heap, etc. This way, if a particular application only uses one of the groups of parses, all of the parse nodes generated from the other parses are skipped during deserialisation. When combined with our desire for runtime-configurable annotation and attribute mappings (see below), the user is provided with a very powerful, expressive, dynamic, and efficient type system.

3.2.6 Runtime-configurable annotation and attribute mappings

We stated earlier that one of our design goals was to have native class definitions "decorated" in some language-specific manner to mark the class as being a DOCREP annotation type (Section 3.2.1). A potential downside of this approach is that class and member variable names then directly correspond to names of types and attributes on the wire. Here are two situations where this is a problem:

1) The user has a DOCREP consumer implemented in Java and needs to consume in documents produced by a DOCREP producer implemented in Python. Both programs use the schema shown in Figure 3.1. If native class and member variable
names map directly to names on the wire, the Java program will be forced to use unidiomatic naming conventions for the two parse node references on the Sentence class. Idiomatic member variable naming Java is to use camelCase (goldParse), but the Python producer will be using idiomatic Python naming conventions (gold_parse). Forcing users to use a particular consistent fixed naming convention for their schemas between languages goes against our easy to use and idiomatic design goals (Section 3.1.2), so that is not an acceptable solution;

2) The user has reimplemented a standard parser bracketing evaluation script evalb\(^4\) to be DOCREP aware. This implementation defines a schema where the name of the annotation heap for the ParseNode instances is called parse_nodes. Consider the case of the user described in the previous section (Section 3.2.5). The user has a number of different annotation heaps containing parse trees produced by different parsers. They would like to be able to use this DOCREP-aware evalb implementation to evaluate the parse trees stored in one of their annotation heaps, but the heap is not named parse_nodes. The user should not be forced to rename their annotation stores in order to use this evaluation script.

Both of these situations need solutions for DOCREP to be as user friendly as possible. As such, we want DOCREP to support runtime-configurable annotation name and attribute name mappings. That is, the ability to change the mappings from annotation type and attribute names on the wire to their corresponding class and member variable names at runtime. These mappings should be configurable per document and allow different mappings to be provided for serialisation and deserialisation.

More formally, for a schema \(S\) defined by an application, if there exists a subset of another schema \(S'\) which is isomorphic to \(S\), then DOCREP should support using \(S'\) in the context of \(S\) via the combination of lazy serialisation and runtime name mappings. The combination of these design requirements provides a richer and significantly

\(^4\)http://nlp.cs.nyu.edu/evalb/
more expressive type system than what is provided by UIMA or GATE, even without supporting type inheritance.

3.2.7 Original document retention

All existing DRFs retain the original document as part of their serialisation. They provide read-only access to this original document at runtime, allowing users to easily project their stand-off annotations back onto the original document for visualisation and inspection purposes.

There are a number of advantages to not carrying around the original document within a DRF document serialisation. The first is the storage required — disk space is required for both a copy of the original document and its annotations. If the user wanted to store a large corpus such as ClueWeb12 (Gabrilovich et al., 2013) in an existing DRF, the storage costs are expensive. The raw ClueWeb12 data is 1.95 TB in size. Assuming the user does not want to delete the original version, even importing this data without any annotations into an existing DRF requires another 1.95 TB of disk space. This is an expensive investment for no immediate gain. This additional space required per serialised DRF document also affects the bandwidth required for inter-process communications, such as when an NLP pipeline is operating in a distributed environment, e.g. the CURATOR framework (Clarke et al., 2012).

Another disadvantage to retaining the original document along with its corresponding annotations is that it prevents the annotations from being distributed without licencing issues for the underlying document. If a user wishes to publish their new annotations over a corpus, it would be convenient to place the annotations online for others to easily obtain. An example of this situation is shared task data, such as the CoNLL 2003 annotations\(^5\) (Tjong Kim Sang and De Meulder, 2003). For this shared task, the organisers distributed the gold-standard annotations in a makeshift stand-off

\(^5\)http://www.cnts.ua.ac.be/conll2003/ner/
format with a script to combine it with the original data. The participants were required to have a copy of the corpus for the script to operate.

We do not want DOCREP to retain the original document as part of its serialisation. One disadvantage of this approach is that the user needs to know where the original documents are located in relation to the serialised DOCREP if they want to project the annotations back onto the original document. This is not a frequent operation relative to the number of times annotations are used independently of the underlying original document. In summary, not retaining the original document leads to smaller serialisations due to the original document not needing to be serialised, as well as to faster serialisations; and this allows annotations over a corpus to be distributed freely without worrying about the licencing of the original documents. If a user wants to project a received set of annotations back onto the original documents, they need to have access to the original documents.

In our example Token annotation type in Figure 3.1, the raw attribute retains a copy of the token string from the original document. Due to the structured nature of DOCREP, the content of the original document can often be mostly reconstructed from the combination of the token offset information and raw attribute. Distributing a DOCREP file containing populated raw attributes would violate any licencing agreements regarding the original documents. As such, DOCREP should support clearing these raw attributes so that the annotations can still be distributed without licencing issues. With the raw attributes cleared, anyone with access to the original document can reconstruct the raw value from the offset span on the token. The DOCREP command-line tools should facilitate this process (Section 3.1.5).

3.3 Serialisation

For DOCREP to be as widely accessible as possible, the serialisation format and serialisation process need to allow DOCREP to be used anywhere; from high throughput
production environments to client-side JavaScript in a web browser. This requirement imposes some constraints on certain aspects of the serialisation format and process. Additionally, DOCREP should facilitate the distribution of DOCREP documents, from the API all the way down to the serialisation format. Combining different serialised DOCREP document collections together should be a simple, efficient process.

### 3.3.1 Efficient and environment agnostic serialisation

For DOCREP to be as widely accessible as possible, the serialisation format and process needs to be environment agnostic, allowing DOCREP to be serialised and deserialised anywhere. Within systems implemented in Java, there has been a history of using native Java object serialisation, such as the machine learning trained models in CoreNLP. This form of serialisation is undesirable for two reasons. First, it ties the serialised data to languages which use the Java Virtual Machine (JVM), and second, the serialised data is not robust to modifications of the class structure of serialised objects.\(^6\) We want the DOCREP type system to be as flexible as possible, easily supporting schema changes. As such, this form of serialisation is not an option for DOCREP.

An operating system and programming language independent serialisation format is required so DOCREP is not restricted from running anywhere. Many existing standardised serialisation formats exist, such as XML, which both GATE and UIMA use for the serialisation of their annotations. XML has the advantage that there exists an entire ecosystem of libraries, tools, and technologies for working with it, but these do not out-weight the many efficiency disadvantages. In order to serialise an existing structure to XML, most XML serialisation libraries require the construction of the corresponding XML tree structure in memory first before being able to serialise this tree structure into its textual representation. This means that for a given annotation graph, the process of serialisation needs to double the number of objects in memory (at least) — one for the original annotation object and one for the corresponding entry in the XML tree.

---

\(^6\)http://docs.oracle.com/javase/8/docs/platform/serialization/spec/version.html
serialised form of the XML tree is also verbose, requiring significantly more space than other more compact object serialisation formats such as JSON, Protocol Buffers, and MessagePack.

Another aspect of serialisation efficiency is how many passes of an input stream are required to construct the corresponding DOCREP annotation graph. The inverse is also true: how many traversals of the annotation graph are needed to serialise it. In order to ensure that the serialisation pipeline is not the bottleneck in any DOCREP implementation, we would like the serialised form to be designed and structured in such a way that the number of passes required for both serialisation and deserialisation is as close to one as possible.

### 3.3.2 Streaming model

In order to maximise the ease of use of DOCREP serialisations, and the ability for DOCREP serialisations to facilitate parallel and distributed computation, we would like the DOCREP to use a streaming serialisation model. This streaming model is a model that many NLP researchers are already familiar with from writing UNIX pipelines. In traditional UNIX pipelines, text-processing applications are chained together via pipes, where streams of text flow between each of the components. UNIX pipelines also facilitate parallelism by design. We would like this same idea to work with DOCREP streams, except instead of lines of text, the streamable unit would be a document and its annotations.

For a collection of DOCREP serialised documents to be streamable, a number of restrictions are imposed on the structure of the serialisation format. One such restriction is that each document in a DOCREP stream needs to appear directly after one another, with no additional stream-level metadata. This restriction implies that simply concatenating two valid DOCREP streams together should form a valid DOCREP stream. However, this restriction imposes a modelling constraint. Since the top-level unit in
the DOCREP data model is the document and not the corpus (as is the case in UIMA, for example), documents cannot refer to one another.

There are many advantages to a streaming model. One such advantage is that streaming models make distributed processing very easy when using a typical work queue architecture. Individual documents can be easily read off the stream one at a time and sent off to the appropriate worker, with resulting documents being simply appended to the end of the output stream when they have been processed. A disadvantage to this approach is we lose the ability to natively model between-document relationships, such as in parallel corpora. A non-native modelling can still be achieved by storing an identifier on each document for its parallel counterpart. There is no constraint that offset spans (Section 3.2.4) must refer to a single document only. In the case of parallel corpora, an annotation on a particular document could have an offset slice into a parallel document, representing the parallel span. While this may not be an ideal representation for some applications, we believe that the advantages gained from using a streaming representation for DOCREP outweigh the disadvantages of not being able to explicitly model a reference to another document.

### 3.3.3 Normalised data storage

We would like DOCREP streams to contain normalised data only. By this, we mean that additional metadata and data structures (e.g. indices) should not be included in the data storage format. Only the annotations, and their direct metadata, should be stored. This is in contrast to the CAS structure used in UIMA (Section 2.3.3), which contains both annotations and indices. Because all annotation instances are stored within the one heap, an indexing structure into the heap is needed to obtain the annotation objects of a particular type. The index repository within the CAS can also be used to house additional indexing structures over a given annotation type.

In DOCREP, we would like to have type-specific heaps (Section 3.2.5) instead of a single heap for all annotation instances. This means that an equivalent indexing
structure to UIMA’s index repository is not required; the only reason DOCREP would still need one is for storing user defined indexes over annotations. This is an aspect that should be left to the application layer because different applications may want to index over the annotations in different ways. Storing arbitrary indexes on the wire seems unnecessary, especially since most documents do not have large numbers of annotations placed over them. Instead of storing indexing structures on the wire, the DOCREP APIs should facilitate the creation of per-document index structures during deserialisation. For some applications, corpus-level index structures are also desirable. The method used to facilitate per-document index creation upon deserialisation should also facilitate in the population of corpus-level index structures.

3.3.4 Per-document metadata

As mentioned earlier (Section 3.1.5), when working with text corpora, two frequent operations are filtering for annotation instances which matching some condition, followed by optionally counting the matches. We would like DOCREP to not get in the way of these existing workflows. Ideally, a serialised DOCREP document should not be fully deserialised to extract information about its type system (Section 3.2.3) or the size of annotation heaps (Section 3.2.5). This would allow for efficient metadata collation over a stream of DOCREP documents as only the per-document header needs to be deserialised — the serialised annotation instances can be skipped entirely.

3.3.5 Self-describing types

Our design goal to not need external files to define annotation types and their attributes (Section 3.2.1) has one primary implication: the annotation type schema needs to be part of the document serialisation. Without an external file to describe how the serialised data should be interpreted, the data needs to be self-describing. This leads to the design requirement for DOCREP that a serialised document needs to be fully self-describing. The schema describing annotation types, their attributes, and their
relationships needs to be included so that DOCREP consumers are able to deserialise the document into the appropriate runtime objects.

As DOCREP documents are self-describing, it would be convenient to be able to view the schema of an existing DOCREP document. Given a DOCREP document, the DOCREP command-line tools (Section 3.1.5) should facilitate both the visualisation of the schema and also the generation of programming language specific DOCREP annotation type definitions. The source code generation is useful mainly as a convenience for the developer, saving them writing the code manually.

3.4 Summary

This chapter began by highlighting our identified need for a new lightweight, programming language agnostic DRF for the CL and LT community to use. We then went on to outline each of our design requirements and design goals for proposed new DRF, DOCREP. Our design requirements satisfy the criteria outlined in Bird and Liberman (2001) and Ide et al. (2003) in Sections 2.2.2 and 2.2.3:

**Generality, specificity, simplicity, and expressive adequacy**  Our outlined requirements do not place any limits on linguistic formalism. Our many usability requirements (Section 3.1) strive towards simplicity and flexibility.

**Searchability and browsability**  Our serialisation requirements (Section 3.3) demand an efficient, streaming serialisation protocol capable of scaling to web-scale corpora and supporting scale-out parallelisation.

**Maintainability, durability, and incrementality**  Our type system requirements (Section 3.2) mandate and facilitate flexibility through lazy serialisation and runtime-configurable schema mappings. Our rich set of tools (Section 3.1.5) for working with DOCREP streams aid in corpus management.
Separability  The combination of type-specific heaps (Section 3.3.3), lazy serialisation, and rich set of tools facilitates the extraction of annotation layers.

In summary, our design requirements for DOCREP are as follows:

- The core ideas should be programming language and paradigm agnostic, but the realised APIs should be idiomatic in each language and consistent across languages.
- It should have a low cost of entry for a new user, while being lightweight, fast, resource efficient, elegant, and expressive.
- It should be easy to use regardless of development environment or workflow.
- Annotation types should be realised as first-class types in each programming language without the need for an external type definition file.
- Isomorphic and underspecified schemas should be supported without any loss in fidelity of the underspecified types and their annotations.
- The serialisation format should be environment agnostic, while being efficient to work with and utilise a streaming paradigm.
- The schema should be serialised as header information for each document in a completely self-describing manner such that the interpretation of the serialised annotations is unambiguous.

Chapter 4 goes on to realise these design goals by implementing DOCREP APIs in the three most widely used languages within the NLP community, motivating various aspects of the implementations with reference back to these design goals. Each of the implemented APIs are idiomatic within their language yet consistent between languages. Following the realisation of DOCREP APIs, we go on to evaluate them against these design criteria, and state how the realised APIs satisfy our design requirements.
In the previous two chapters, we outlined the background and motivation which lead to the development of DOCREP. Highlighting usability deficits in existing DRFs, we want DOCREP to be a lightweight, programming language agnostic, easy to use DRF which is capable of scaling to web-scale corpora.

With our design goals outlined, we move on to discussing the implementation details of the framework, motivating specific implementation decisions with reference to their corresponding design goals. In this chapter we first go through the data model and serialisation protocol chosen for DOCREP at an abstract level. The design of the data model as well as many aspects of the serialisation protocol stem from, and ultimately are directly or indirectly influenced by, the streaming nature of DOCREP. We then go on to discuss the various technical details about our implementation of DOCREP, demonstrating the consistency of the API across programming languages while also being idiomatic within each language. The last section of this chapter discusses the DOCREP runtime, including how lazy serialisation (Section 3.2.3) and runtime-configurable annotation and attribute mappings (Section 3.2.6) are implemented.

This chapter is designed to provide enough insight into the implementation details of our provided APIs to allow for reimplemention or the implementation of a DOCREP API in another language.
4.1 Data model

The data model for DOCREP is similar to the data model for existing DRFs such as GATE and UIMA. By design, the data model is a bit less general, targeting the majority use cases rather than aiming for complete generalisability. The decision here was deliberate — DOCREP is meant to be a lightweight framework with a specific focus on dealing with text documents.

DOCREP does not have an equivalent concept of the Subject of Analysis (SOFA) in the UIMA framework (Section 2.3.3). The SOFA facilitates different “views” of the same document. For example, an audio recording of a speech as well its corresponding transcript can be group together as being part of the same conceptual document, with each subject of analysis (view) being able to be treated independently for annotation purposes. The streaming nature of DOCREP does not facilitate this concept — each DOCREP document is treated as independent.

There are three main data modelling concepts in DOCREP: documents, stores, and annotations. Annotations have attributes of varying data types. Documents have attributes, and contain the annotation objects in stores. The rest of this section discusses these three concepts in detail.

4.1.1 Annotations

As with most DRFs, arguably the most fundamental data modelling concept is the notation of the annotation. Annotations capture any information placed over a document such as token boundaries, named entities, and coreference mentions. Additionally, annotations can model entities associated with the document that are not directly represented in the document text. For example, for a speech transcript, each sentence in the underlying document might contain metadata stating who spoke the sentence. Modelling this information in DOCREP can be achieved through a speaker annotation type to represent the real-world speakers, and an instance of this type can be instantiated...
for each speaker present in the document. A sentence annotation can then refer to specific speaker object, facilitating a normalised data model for speakers.

In DOCREP, an annotation instance consists of an annotation type and attribute key-value mappings. The attribute values can be primitive data types, references to one or many other annotation instances, a span over the bytes of the original document, or a span over some other sequential block of annotation objects. Annotation types correspond to an (object-oriented) class or record in each implementation language, and provide a vocabulary of available attributes and associated semantics.

Intuitively, linguistic annotations are applied to a span of characters, not bytes. DOCREP provides access to both layers, but uses the byte offsets as its underlying representation. Defining annotations in terms of bytes, instead of characters, allows a direct mapping back to the original data source. The original byte offsets into the original encoding are important for many applications with binary formats or non-UTF-8 encodings. For example, annotations over binary files such as PDFs can be made directly into the original document without having to first perform some conversion to a text-based format. Performing this conversion means that annotations can not be mapped back to the original document.

For example, a token annotation spans a sequence of bytes in the original document corresponding to its underlying representation. A named entity annotation can be viewed as spanning a usually sequential block of token annotations. However, annotations do not have span over something. When modelling coreference annotations, there might be a generic “entity” annotation object for each canonical mention in the document, and each coreferent mention points back to the canonical entity object.

In terms of implementation, we want the definition of a DOCREP annotation to be tied to a class (or record) definition in each programming language (Sections 3.2.1 and 3.2.2). Ideally, the fact that a class is a DOCREP annotation class should not affect what the user is able to do with the class, nor with objects of that class. Adding additional methods, inheriting from other classes, and adding additional member
variables that DOCREP is not aware of should all be possible. For example, adding methods such as “am I a leaf node?” or “am I a root node?” on a parse node annotation type can drastically improve the usability and usefulness of the instantiated objects at runtime. If any of the produced DOCREP APIs do not meet any of these conditions, they fail the low cost and easy to use design criteria.

4.1.2 Documents

In DOCREP, the document is treated as a special kind of annotation. There can only be one (and exactly one) DOCREP document object per conceptual document.

Like an annotation, a document object can have named attributes. One restriction on this is that an annotation cannot refer to another document object. This comes down to the streaming nature of DOCREP (Section 3.3.2), where documents are treated as completely disjoint from one another.

In addition to having named attributes, documents are where the annotation objects reside conceptually. This again comes from the streaming nature of DOCREP. Each document is independent, but all of the annotations for that document are somehow associated with the document itself. The annotation objects are contained in what we call stores. Store instances are available as named attribute values on document objects.

4.1.3 Stores

When an annotation object is created for a DOCREP document, the object needs to live somewhere on the document so that DOCREP knows the document owns the object and to serialise it as part of the document. In DOCREP, these storage locations are called stores. Annotation stores are an ordered collection of annotation objects of a given type.

There can be more than one annotation store for a given annotation class. For example, users might want to store the gold standard version of a particular annotation layer separately to an auto version, or multiple gold annotations from different annotators.
4.1. Data model

There are advantages to doing this which relate back to the design goal of supporting lazy serialisation — if another application only cares about the auto annotation, it can ignore the gold store entirely rather than having to iterate through each of the objects in a combined store and work out which ones are gold and which ones are not.

The order that the objects appear within a store may or may not meaningful outside of the context of slice semantics. For example, parse node annotation objects could be stored using a tree ordering where siblings are always stored adjacent so that they can be referred to as a slice. In this case, adjacency within the store does not always imply two nodes are siblings.

Stores are implemented as an ordered data structure in each DOCREP API, wrapping the standard ordered container in each language, usually an array or list type.

4.1.4 Primitive-typed fields

For DOCREP to be flexible for representing linguistic annotations of various corpora, the API needs to support attributes and values of a range of primitive and non-primitive data types. Primitive types are data types which should be considered “built-in” to DOCREP itself.

The primitive types DOCREP supports are signed and unsigned integers, floating point numbers, Boolean values, Unicode strings, and arbitrary binary strings. These types are primitive types in most programming languages, making consistent the implementation of the DOCREP APIs, and should cover most types of data associated with linguistic annotations. Primitive-typed attribute values could be used to represent annotation attributes as diverse as a probability (floating point number), a POS tag (Unicode string or integer enum), or whether or not the current node in a parse tree is a trace node (Boolean).

One difficult data type was the timestamp. While it might be quite useful to have a timestamp data type defined as a numeric-based primitive, most programming languages do not have a standardised, nor consistent, way to represent a timestamp
Chapter 4. **DOCREP: Implementation**

value. Additionally, a typical use case was unclear, meaning that users may want timestamp values outside of some range we defined our timestamp type to support. Since we are aiming for maximal cross-application and cross-language portability, we ended up deciding against including timestamp as a primitive data type. Thus far this has not caused many issues, with applications implementing the ISO8601 encoding and decoding logic at the application layer when timestamps have been needed (ISO8601, 1988). ISO8601 encodes timestamp values in a string representation.

4.1.5 **Pointer and self-pointer fields**

In addition to primitive-typed values, attributes on annotations need to reference other annotation objects. These can be used to construct graphical structures such as parent or child relationships in a parse tree, or to reference the canonical mention from a coreferent mention.

Some situations require this concept of a reference to another annotation to have an arity that is not 1. For example, instead of representing a tree structure via a parent pointer on the child node, the user may need to store multiple child pointers on the parent. This sequence of child pointers could be zero or more in size. As such, **DOCREP** needs to support both a single pointer and a many pointers annotation value type. In **DOCREP** nomenclature, these types are referred to as **pointer** and **pointers** respectively.

One issue that arises with pointers is that their target annotation store is defined by the schema. That is, for a given annotation type $T$ with pointer attribute $P$, the store $S$ that $P$ is stored in is defined as part of the definition of $T$. Since this attribute-to-store mapping is defined as part of the type definition, it cannot change be changed per object. This restriction can make working with self-referential types problematic.

Consider the case where a document has two different annotation stores $S$ and $S'$ for a parse tree node type $T$. One store contains gold parse instances and the other contains auto parse instances. The parent and/or child pointers $P$ on this parse node type $T$ need to state which annotation store they point into. If a parse node object is
located in store \( S \), then the definition of \( P \) needs to declare it points into \( S \). However, if a parse node object is located in store \( S' \), then the definition of \( P \) needs to declare it points into \( S' \). As we stated before, the declaration of \( P \) is part of the type definition and cannot be changed per object, so this situation is problematic.

Supporting both the gold and auto annotation stores would not be possible without duplicating the definition of the parse node type \( T \); one with \( P \) pointing into \( S \) and one with \( P \) pointing into \( S' \). Instead, DOCREP introduces a variant on the pointer and pointers types to handle this situation. These types are referred to as a self-pointer and a self-pointers respectively. These types should behave as a normal pointer or pointers object at runtime, but they have the added semantics that the target object at the other end of the reference is contained in the same store as the current object.

The pointer types are implemented as nullable references in the DOCREP APIs. This means as a standard object reference in Python and Java, and as a pointer in C++.

### 4.1.6 Slice fields

In the discussion about annotations, we mentioned that an annotation often spans over either some consecutive sequence of bytes in the original document, or over some other annotation objects. We call the annotation value type which supports this notion a slice (Section 3.2.4). A slice, by definition, is simply a start index and end index pair. Two different kinds of slices need to be supported — a slice over sequential bytes of the original document and a slice over sequential annotation objects on the current document. These are referred to as a byte slice and an annotation slice respectively.

The primary use case for byte slices is to indicate, within the original document, which sequential block of bytes the token covers. That is, a slice encodes that the token “Thor” starts at byte 20 and ends at byte 24 in your ASCII-encoded original document (where end is one past the last byte), or that the token “Þunor” starts at byte 30 and ends at byte 36 in your UTF-8-encoded original document, or that the token “document”
maps to the byte string "<em class="defn">document</em>" in the original document.

As highlighted in these examples, slices are byte offsets, not Unicode code point offsets. As such, byte slices can also be used to encode annotations over non-textual corpora. For example, phonemes in an audio file can be indicated using byte slices, assuming there exists a sequential span of bytes in the underlying audio format which represents the duration of the audio track during which the phoneme was produced. If this is not the case, the phonemes could still be represented as a slice of millisecond offsets relative to the start of the audio track.

An annotation slice is a start and end index pair over a sequential block of other annotation objects. Annotations are stored in an ordered data structure (Section 4.1.3) so the annotation slice (as indices into that data structure) has the same semantics as the byte slice. There are many use cases for annotation slices. One example is for named entity annotations — a named entity spans over some sequence of tokens in the document. DOCREP allows this to be modelled natively via a slice attribute over the token store, which is an attribute value on the named entity type.

Annotation slices can be thought of as simply a pair of pointers but with an implied “spanning” semantics. That is, this slice spans from the first pointer to the second pointer. The user could implement this as two pointers, but the implied spanning semantics are not present. Spans are such a common occurrence in linguistic annotations that we made them their own data type.

A self-slice has not been defined in DOCREP at the current point in time, mainly because we have not found a use case for it. If such a use case is found, the DOCREP wire protocol (Section 4.2.1) is already able to handle it.

### 4.1.7 Overview

In order to ground the use of these data modelling components, we present an example of their use in Figure 4.1. This model is intended to show the power of DOCREP
Figure 4.1: An overview of how each of the DOCREP data modelling components fit together in an example schema. This example is an implementation of the example schema from Figure 3.1.

for multi-layer annotation. We use a UML-style notation as we do not assume any particular DOCREP API. This example is an implementation of the abstract definition schema presented in Figure 3.1. The similarity between these two figures should be noted — the implementation is almost identical to the abstract definition. This was by design in an effort to maximise usability and mimics the notion of Python’s “executable pseudocode” — the definition should be as close to pseudocode as possible.

The Document type contains both attributes (doc_id) and stores. More than one store exists for the ParseNode type, facilitating gold and auto parses. The Token annotation type has a byte slice and a Unicode string representing its raw token as attributes. The ParseNode annotation type has a primitive label string as well as self pointers to its child nodes. This type also has a pointer to a Token object for leaf nodes. A sentence in this example is modelled as a span over tokens, and as such, the Sentence annotation type has an annotation slice over Token objects. Additionally, it has a pointer to the root node of the gold and auto parse for the sentence.
4.2 Serialisation

DOCREP documents created in memory need to be able to be serialised and deserialised for persistence and transfer between processes. In this section, we describe the way DOCREP documents are serialised. We first design a wire protocol capable of representing the object models that DOCREP supports. We then describe our implementation of this wire protocol, including experiments with various candidate serialisation formats, concluding that MessagePack best supports the needs of DOCREP. Finally, we conclude with a discussion of the consequences of choosing MessagePack for the wire protocol.

4.2.1 Wire protocol design

For the sake of generalisation, the discussion here is done in abstract terms, independent of serialisation implementation which is discussed in Section 4.2.4.

The grammar rule style notation here uses a BNF-style syntax, with the following syntactic attributes:

- \([ \text{xxx} ]\) indicates a list;
- \((\text{xxx, yyy, zzz})\) indicates a fixed-length list;
- \(\{\text{xxx : yyy}\}\) indicates a map;
- \(\text{xxx}*\) indicates that \(\text{xxx}\) is repeated zero or more times;
- \(#\text{xxx}\) indicates a comment.

We note again that list and map here are abstract data structure concepts independent of any particular implementation.

In a number of places in the grammar rule definitions, there are non-terminals which are either repeated or appear one after another, which are not wrapped in a list structure. This might seem like an odd design decision for a streaming protocol, but it was done intentionally. Since we want to support very fast deserialisation as
well as lazy attributes and stores, we want to delay the deserialisation of objects from the wire as long as possible. If these parts of the wire protocol were wrapped in a list and the deserialisation API did not support low-level per-attribute deserialisation (that is, it only provides “deserialise the whole list”), we would lose support for lazy deserialisation.

Below, we describe the wire protocol in detail, explaining the purpose of each of the components and outlining why they are needed. In reality, the wire protocol was developed iteratively in conjunction with the exploration of serialisation formats, but we present it here as disjoint. Certain aspects of the wire protocol design might seem unusual without the knowledge that MessagePack is used as the underlying format.

**Header information**

```
<wire> ::= <doc>*  # Zero or more <doc>s.
<doc> ::= <wire_version> <types> <stores> <doc_instance> <instances_groups>
```

Valid DOCREP streams start off with the `<wire>` non-terminal. A stream consists of zero or more `<doc>` instances. A `<doc>` consists of five non-terminals appearing directly after one another on the wire: the DOCREP serialisation protocol version number, the annotation type definitions, the annotation store definitions, the document object itself, and then each of the annotation objects on the document.

```
<wire_version> ::= unsigned_int  # Version number of the wire protocol.
```

As DOCREP has been iteratively developed, the serialisation protocol has changed to support additional functionality and improve its ability to model certain linguistic phenomena. The `<wire_version>` unsigned integer indicates which version of the wire protocol this document uses. The DOCREP website maintains the most up to date wire protocol definition.
Chapter 4. DOCREP: Implementation

Annotation type definitions

```
<types> ::= [ <type> ]  # Zero or more element list.
<type> ::= ( <type_name>, <fields> )  # 2-element list.
<type_name> ::= utf8_string  # The name of the annotation type.
               # The name of the Document type is "__meta__".
<fields> ::= [ <field> ]  # Zero or more element list.
[field] ::= { <field_attr> : <field_val> }  # One or more element map.
            # NAME key is required.
```

Each DOCREP annotation type is defined by a `<type>` definition in the `<types>` list. The document type is also defined in this list — it has the special value `__meta__` as its `<type_name>`. While the wire protocol itself does not place any restrictions on what UTF-8 strings are valid annotation type names, different programming languages will have different restrictions on what are valid class names. For example, the Chinese word for sentence (句子) is a valid class name in C++11, Python 3, and Java, but not in Python 2. As such, one should avoid using certain Unicode code points in annotation type names so that DOCREP consumers are not forced to provide runtime mappings between type names on the stream and runtime class names (Section 4.4.2). Additionally, if users wish to implement namespacing of annotation type names (Section 3.2.6), they may wish to follow the same guidelines with their namespace separation so that consumers of their DOCREP streams are not forced to provide runtime mappings between stream type names and runtime class names.

```
```

Each DOCREP annotation type definition has zero or more fields (Section 4.1.1), each one defined by a `<field>` definition inside the `<fields>` list for the current type. This `<field>` map has one or more elements, with the required key/value pair being the name of the field.
The allowable keys in the `<field>` map are defined by the `<field_attr>` enum. The `NAME` key is used to define the name of the field. Its corresponding value is a UTF-8 string with the same Unicode recommendation as annotation type names. The `POINTER_TO` key is used to indicate that this field is a pointer. Its corresponding value is the ID of the store that this pointer points into (see the next set of grammar rules). The `IS_SLICE` key is present if the field is a byte slice or an annotation slice. Its corresponding value is `nil` as this key acts as a flag rather than a key/value pair. The `IS_SELF_POINTER` key is present if the field is a self pointer rather than just a vanilla pointer. Having both `POINTER_TO` and `IS_SELF_POINTER` present in the `<field>` definition is invalid. Its corresponding value is `nil` as this key acts as a flag — the corresponding store that the pointer points into is derived at runtime as the store containing the annotation object that the pointer is an attribute of. If the `IS_COLLECTION` key is present, it acts as a modifier of either `POINTER_TO` or `IS_SELF_POINTER` to indicate that the field stores multiple pointers instead of just a single pointer. Having this key present with neither `POINTER_TO` nor `IS_SELF_POINTER` set is invalid. Like the previous two enums, its corresponding value is `nil` as this key acts as a flag.
In order to clarify this abstract definition, the above code snippet shows the contents produced `<type>` non-terminal for the `ParseNode` annotation type definition from Figure 4.1. The `??` indicates that this value is not currently known since this snippet does not contain a `<stores>` definition.

**Store definitions**

```plaintext
<stores> ::= [ <store> ]  # Zero or more element list.
<store> ::= ( <store_name>, <type_id>, <store_nelem> )  # 3-element list.
<store_name> ::= utf8_string  # The serialised name for this store.
<type_id> ::= unsigned_int  # The i'th type definition in <types>.
<store_nelem> ::= unsigned_int  # The number of objects in this store.
```

After the annotation and document types are defined, the annotation stores which appear in the document are defined (Section 4.1.3). Each store is defined by the `<store>` non-terminal, which consists of a 3-tuple of the name of the store, the ID of the annotation type, and the number of annotation instances contained within this store. The same advice about Unicode values in annotation type and field names also applies to store names. The number of instances is stored here as an optimisation to make it possible to calculate metadata statistics over documents without having to fully deserialise them (Section 3.3.4).
4.2. Serialisation

Document object data

After the definition of the annotation types and stores, the annotation objects appear on the wire. The first annotation object to appear is the document itself (Section 4.1.2), represented by the `<doc_instance>` non-terminal, followed by one `<instances_group>` per store defined in `<stores>`. The ith `<instances_group>` definition corresponds to the ith store definition in `<stores>`. The document `<instance>` and each of the stores `<instances>` non-terminals are prefixed with the number of bytes on the wire that the object occupies. This is here to allow for the lazy deserialisation of the store if it is not needed in the current application — that is, the bytes for the store can be skipped entirely without having to deserialise them (Section 3.2.3 and Section 4.1.3).

Annotation object data

Each instance is then serialised as a zero or more element map on the wire. The keys of this map are the ID of the field being serialised, and its corresponding value is dependent on the value being serialised. A map structure is used here to allow missing fields or fields with default values to be skipped during serialisation.

One value worth mentioning explicitly is how pointers are represented in the wire protocol. Since annotation instances are contained within stores, pointers to other
annotation instances serialise as the (integer) index of the pointed to object within its containing store. This approach is commonly referred to as pointer swizzling.

Overview

In order to clarify this abstract wire protocol, the snippet below shows example serialisation of an empty DOCREP document which uses the schema presented in Figure 4.1. This example assumes the wire protocol is version 3. The ??? indicates that this value is unknown at the current point in time, as how many bytes are needed to represent an empty list or map is format dependant.

```
3  # <wire_version>
[ ( "__meta__" , [ ( 0 : "doc_id" ) ] ),
  ( "Token" , [ ( 0 : "span" , 2 : nil ) ,
    ( 0 : "raw" ) ] ),
  ( "Sentence" , [ ( 0 : "span" , 1 : 2 , 2 : nil ) ,
    ( 0 : "gold_parse" , 1 : 1 ) ,
    ( 0 : "auto_parse" , 1 : 0 ) ] )
  ( "ParseNode" , [ ( 0 : "label" ) ,
    ( 0 : "token" , 1 : 3 ) ,
    ( 0 : "children" , 3 : nil , 4 : nil ) ] )
] # <types>
[ ( "auto_parse_nodes" , 3 , 0 ) ,
  ( "gold_parse_nodes" , 3 , 0 ) ,
  ( "sentences" , 2 , 0 ) ,
  ( "tokens" , 1 , 0 ) ,
] # <stores>
??? { } # <doc_instance>
??? [ ] # <instances_group> for "auto_parse_nodes"
??? [ ] # <instances_group> for "gold_parse_nodes"
??? [ ] # <instances_group> for "sentences"
??? [ ] # <instances_group> for "tokens"
```

4.2.2 Serialisation format

With a wire protocol designed, the next step is to decide how to serialise it. This decision comes down to two other, possibly connected, questions: 1) should we use


4.2. **Serialisation**

a binary or plain text format?, and 2) should we use an existing object serialisation format or should we create one specific to our needs?

Many generic object serialisation formats already exist, and a number of them have been used for over 15 years. Some well-known plain text examples include JavaScript Object Notation (JSON)\(^1\) and Simple Object Access Protocol (SOAP)\(^2\) and some well-known binary examples include Common Object Request Broker Architecture (CORBA)\(^3\) and Component Object Model (COM)\(^4\). There are also many other more recent object serialisation formats which are perhaps less well known, such as Protocol Buffers and MessagePack.

The binary versus plain text decision was not a hard decision to make due to three main factors. First, one of our design goals was to make DOCREP as lightweight and fast as possible. Reducing the amount of data that needs to be read and written is an obvious way to reduce the overall I/O cost of an application; less I/O means less time spent performing I/O. This argues for a binary format. Second, representing arbitrarily nested annotation structures is problematic for text-based serialisation formats. GRAF is able to achieve this in a plain text format by having objects encoded as XML nodes with identifiers, and having annotations reference other annotations via XPath queries serialised in the XML itself. While GRAF showed that it is not impossible to use a plain text format to achieve this, it is not elegant nor easy to work with. Third, the main motivation for using a plain text serialisation format is readability. In Chapter 2, we saw many examples of stand-off annotations serialised as XML where the readability criteria was hard to justify (e.g. Figure 2.3). On the other hand, Chapter 2 also showed that plain text formats less structured that XML are not able to represent arbitrarily nested linguistic annotations. These three considerations directed the decision very strongly towards a binary serialisation format.

\(^1\)http://json.org/
\(^2\)http://www.w3.org/TR/soap/
\(^3\)http://www.corba.org/
\(^4\)https://www.microsoft.com/com/
The second question, should we use an existing object serialisation format or should we create one specific to our needs, was not as easy to answer. We surveyed the space of modern binary serialisation formats to find the best-performing, open-source, feature-rich format that would best suit our needs for serialising the proposed wire protocol. This survey narrowed our choices down to just four candidates:

**BSON** Short for Binary JSON, BSON\(^5\) is binary serialisation format that looks very similar to JSON. BSON was originally developed by MongoDB\(^6\) as its primary data representation. Like JSON, BSON is schema-less — no structure definition is required because the values themselves are all typed.

**MessagePack** MessagePack\(^7\) is a binary serialisation format that feels similar to JSON, but was designed to be as compact as possible. Like BSON and JSON, MessagePack is schema-less in that no structure definition is required. An official extension to MessagePack exists which defines how it can be used for RPC.

**Protocol Buffers** Developed at Google for internal cross-application data transfer and RPC, Protocol Buffers\(^8\) has since been open sourced and has become widely used by the public. Serialisation records and attributes are defined in an external file which allows for record versioning as well as attribute modifiers such as being optional or repeated.

**Thrift** Developed at Facebook and then released as an Apache project, Thrift\(^9\) is a binary serialisation and RPC specification. It provides the same set of functionality as Protocol Buffers, but its implementation and wire protocol differ in many aspects. Serialisation records and attributes are defined in an external file which allows for record versioning as well as attribute modifiers such as being optional.

---

\(^5\)http://bsonspec.org/
\(^6\)https://www.mongodb.org/
\(^7\)http://msgpack.org/
\(^8\)http://code.google.com/p/protobuf/
\(^9\)http://thrift.apache.org/
4.2. Serialisation

<table>
<thead>
<tr>
<th></th>
<th>Self-describing</th>
<th>Supports RPC</th>
<th>Open Specification</th>
<th>Existing Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSON</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MessagePack</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Thrift</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.1: A summary of the high-level differences between our considered binary serialisation formats.

Tables 4.1, 4.2, and 4.3 summarise feature differences between these four binary serialisation formats that are relevant to DOCREP. In Table 4.1, we see that 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, while also being self-describing on the wire (Section 3.3.5). Likewise, Protocol Buffers and Thrift are similar to each other in their design. They are not self-describing — instead, they require an external schema definition file which defines how to interpret the messages on the stream. In this external file, users of the serialisation library define the structure of the messages they wish to serialise and deserialise. The library provides a tool to convert this external file into source code for the programming language of choice, which the user then calls from with their application.

One of our design considerations when creating DOCREP was for the wire protocol to be self-describing (Section 3.3.5). With a self-describing wire 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 format because including the definition of the type system with each document comes at a cost. This is different to GATE and UIMA, both of which require their XML type definition files for the serialised data in order to be able to perform deserialisation. Therefore, BSON and MessagePack look more suitable for DOCREP than Protocol Buffers or Thrift.
Remote Procedure Calls (RPC) are an inter-process communication technique that allows the programmer to invoke a subroutine in a different process without explicitly specifying the details of the remote interaction. MessagePack, Protocol Buffers, and Thrift all provide RPC functionality. However, DOCREP does not need RPC support, as the purpose of a DRF is only to provide a serialisation API. That being said, RPC support would be beneficial in NLP pipelining frameworks. The CURATOR NLP pipelining framework (Clarke et al., 2012) uses Thrift to provide both serialisation and RPC functionality between cross-language disjoint components in the pipeline. UIMA gets around the need for RPC through the use of an ActiveMQ\textsuperscript{10} message broker to pass messages between Analysis Engine instances running in different processes.

All four of these considered binary serialisation formats have open source specifications as well as mature open source library implementations. Therefore, based on the high-level features considered in Table 4.1, MessagePack is the most suitable.

We would like to have made decisions which allow for a highly efficient C++ DOCREP API and for APIs in at least Java and Python to be possible. As such, it was important that the specification for the binary serialisation format we chose to use be open source so that we could implement our own serialisation if required.

Each of the considered binary serialisation formats supports a different set of primitive and non-primitive data types. Support for primitive data types is summarised in Table 4.2. The main difference between the formats is their support for ranges of differently sized numeric types, explicitly unsigned integer values, and fixed-sized encodings. Fixed-sized encoding are a space optimisation to use fewer bytes to represent small values. An example of this in MessagePack is illustrated in Figure 4.2. UTF-8 strings shorter than 32 bytes do not need a separate length prefix value specifying how many bytes are in the UTF-8 sequence. Instead, the length is encoded into the header byte that specifies that the value following is a UTF-8 string. Many strings in NLP fall

\textsuperscript{10}http://activemq.apache.org/
4.2. Serialisation

The $U$ column specifies whether unsigned integers are supported in addition to signed integers. The $F$ column specifies whether fixed-sized encodings exist in addition to the length-prefixed encodings. The “bin” column specifies whether arbitrary binary strings are supported distinct from Unicode strings.

within this 32 byte window, such as tokens and POS and NER labels, making fixed size string encodings an attractive feature.

In addition to primitive data type support, these binary serialisation formats also support some non-primitive and collection data types. A comparison of these is shown in Table 4.3. At a glance, it may seem odd that Protocol Buffers does not support a native representation for maps. Strictly speaking, maps are not needed in a serialisation format as the data can still be represented as a list of pairs of values. The same is also true of sets. Attributes such as key uniqueness and potential sorted order traversal are aspects of a runtime implementation of a map, not the underlying data the map contains. In Protocol Buffers, if you want to have a map of type $X$ to type $Y$, you would define a list of records which have an attribute of each type. At runtime, if the user wants the data in a map, they would iterate through these records to populate the map. Like the primitive data types, MessagePack supports fixed-sized variants for list and map objects of cardinality less than 16. Our proposed wire protocol specification (Section 4.2.1) has a number of list and map structures that will likely fall within this cardinality threshold, making MessagePack an attractive choice.

<table>
<thead>
<tr>
<th></th>
<th>Integers</th>
<th></th>
<th>Floats</th>
<th></th>
<th>Strings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 16 32 64</td>
<td>$U$</td>
<td>32 64</td>
<td>UTF-8</td>
<td>bin</td>
<td>F</td>
</tr>
<tr>
<td>BSON</td>
<td>× × ✓ ✓ ✓ x</td>
<td>x ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>MessagePack</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>x x ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Thrift</td>
<td>✓ ✓ ✓ ✓ ✓ x</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

Table 4.2: Primitive data type support in our considered binary serialisation formats.
Figure 4.2: The four different encodings for UTF-8 strings in MessagePack (from the MessagePack documentation), depending on the length of the encoded byte sequence. A fixed-size encoding is supported for strings of less than 32 bytes in length.

<table>
<thead>
<tr>
<th></th>
<th>Non-primitives</th>
<th>Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boolean</td>
<td>Enums</td>
</tr>
<tr>
<td>BSON</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>MessagePack</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Thrift</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.3: Non-primitive data type and collection support in our considered binary serialisation formats. The F column specifies whether fixed-sized encodings exist in addition to the length-prefixed encodings.
4.2.3 Evaluation

While MessagePack appears to be the best suited serialisation format for our task, in order to have some resource utilisation or efficiency measurements on which to compare these binary serialisation formats, we implemented a version of the proposed DOCREP wire protocol using each format. As a simple stand-off annotation corpus for this experiment, we chose to use the CoNLL 2003 NER shared task data training data, randomly sampling around 50 MB of sentences from the English dataset. The serialisation of this data contains the documents, sentences, and tokens, along with the POS and NER tags for the tokens. For each format, we compared the size of the serialised data, the speed at which the data was serialised, as well as how well each of the serialised payloads compressed using common compression algorithms. The appropriate external schema files were written for Protocol Buffers and Thrift, and the appropriate type definitions were encoded as header information in the BSON and MessagePack serialisations since they need to be self-describing. For the compression tests, we chose three different algorithms which optimise for different properties: the commonly used DEFLATE algorithm (gzip), the fast Snappy algorithm,\(^{11}\) and the compact LZMA algorithm (xz).\(^{12}\)

Table 4.4 tabulates the results of this experiment. The reported size of the original data is smaller than the sample size as we used a more concise text representation than the data was originally distributed in. The compressed size of the original data is significantly smaller than any of the binary serialisation formats; over 40 times smaller compared to BSON when compressing with LZMA. This is due to the inherit repetition present in the CoNLL 2003 data, where each token contains its surface form, POS tag, syntactic chunk tag, and NER tag. BSON performs noticeably worse than the other formats in terms of original size and speed, as well as its compressed size. While serialisation was performed slightly faster, the size of the serialised data produced by

---

\(^{11}\text{https://code.google.com/p/snappy/}\)

\(^{12}\text{http://tukaani.org/xz/}\)
Chapter 4. DOCREP: Implementation

<table>
<thead>
<tr>
<th></th>
<th>Uncompressed</th>
<th>DEFLATE</th>
<th>Snappy</th>
<th>LZMA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>Original data</td>
<td>—</td>
<td>31.30</td>
<td>1.0</td>
<td>5.95</td>
</tr>
<tr>
<td>BSON</td>
<td>2.5</td>
<td>188.42</td>
<td>5.3</td>
<td>30.32</td>
</tr>
<tr>
<td>MessagePack</td>
<td>1.6</td>
<td>52.15</td>
<td>3.2</td>
<td>16.61</td>
</tr>
<tr>
<td>Protocol Buffers</td>
<td>1.4</td>
<td>51.51</td>
<td>3.5</td>
<td>18.52</td>
</tr>
<tr>
<td>Thrift</td>
<td>1.0</td>
<td>126.12</td>
<td>3.5</td>
<td>20.64</td>
</tr>
</tbody>
</table>

Table 4.4: A comparison of binary serialisation formats being used to serialise DOCREP documents. 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.

Thrift is more than 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.

Being self-describing, having support for fixed-sized encodings of primitives and collections, as well performing well in our serialisation experiment, we concluded that MessagePack was the most suitable binary serialisation format for DOCREP to use. MessagePack also has the advantage of having library implementations in over 40 programming languages. The combination of these two factors significantly eases the development process for a DOCREP API in a programming language we do not provide an official implementation for.

We decided not to pursue the idea of implementing a custom serialisation format for a couple of reasons. First, MessagePack provides all of the functionality we need to be able to implement the wire protocol, including a wide range of fixed and non-fixed sized primitive types and data structures. Second, by using an existing serialisation format, we facilitate the serialisation and deserialisation of DOCREP streams in languages which have a MessagePack library but not a DOCREP library; a valid DOCREP stream is a valid MessagePack stream.
4.2. Serialisation

4.2.4 Serialising with MessagePack

Having decided to use MessagePack as our binary serialisation format, the next step is to outline the impact of choosing MessagePack on DOCREP.

Since MessagePack supports list and map structures natively, very little has to be changed from the original proposal. The \texttt{<obj_val>} non-terminal defining how to serialise an annotation attribute value now becomes any arbitrary MessagePack value.

One other small change we will make is to specify how to serialise slice values to take advantage of the fixed-size integer encodings available in MessagePack. If the field is a \texttt{(start\_value, end\_value)} slice, the \texttt{<obj\_val>} value to be serialised by MessagePack is a 2-element list of \texttt{(start\_value, end\_value - start\_value)} instead of the original object. The delta encoding of the second element in the list allows the variable length encoding of integer values in the binary serialisation format to use fewer bytes than the original \texttt{end\_value} value. For example, consider a token which spans from byte offset 70940 to 70944 in the original document. Since both of these values exceed $2^{16}$, using the \texttt{(start\_value, end\_value)} format to encode this slice requires 11 bytes in MessagePack: 1 for the list header, and 5 bytes for each 32-bit integer. Using the delta encoded representation of the slice, we can reduce this to just 7 bytes: 1 for the list header, 5 for the first integer, and 1 for the delta (70944 – 70940 = 4). Since slices are frequent annotation attributes (at a minimum, tokens will most likely have a byte slice over the original document), this small encoding optimisation is worthwhile.

The full concrete DOCREP wire protocol specification can be seen in Figure 4.3. For the latest version of the protocol, readers are advised to consult the DOCREP website.\footnote{https://github.com/schwa-lab/libschwa}

Existing DRFs could be altered to use an alternative serialisation protocol, such as MessagePack. Some DRFs (UIMA) already provide multiple serialisation protocols to trade off factors including efficiency, storage size, and human readability. Converting existing DRFs to use MessagePack is outside the scope of this thesis.
Figure 4.3: The DOCREP wire protocol specification.
4.3 Implementing the DOCREP APIs

One of the motivations for constructing DOCREP was the lack of a good DRF in programming languages other than Java. DOCREP itself is the definition of a concrete wire protocol and a data model for interacting documents and annotations at runtime. Creating a DOCREP API for a particular programming language requires implementing the wire protocol serialisation and deserialisation, as well as the appropriate language-specific scaffolding to define type annotations as (object-oriented) classes or records.

As we mentioned earlier, we wanted to provide DOCREP APIs in at least the programming languages most commonly used within the CL community: Python, Java, and C++. As a starting point for the DOCREP project, we have implemented and deployed APIs in these three languages. All three APIs are open source and publicly available, released under the MIT licence. We used an MIT licence to make the use of DOCREP as permissive as possible.

Each API implements the DOCREP data modelling concepts (Section 4.1) in an idiomatic manner for the language. Each API also implements serialisation and deserialisation logic (Section 4.2), converting between runtime objects and MessagePack data streams.

4.3.1 The Python API

Our provided Python API supports Python 2.7 and ≥ 3.3, and is available on PyPI, making it installable via the standard Python package manager pip. At the current release (0.4.0), the Python API utilises the official MessagePack Python library as it provides sufficient APIs to implement lazy serialisation.

Figure 4.4 shows an example of each of the DOCREP data modelling concepts (Section 4.1) being used in the Python API. It is important to note here that these are

14https://github.com/schwa-lab
15https://pypi.python.org/pypi/libschwa-python
from schwa import dr

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

class ParseNode(dr.Ann):
    label = dr.Field()
    token = dr.Pointer(Token)
    children = dr.SelfPointers()

class Sentence(dr.Ann):
    span = dr.Slice(Token)
    gold_parse = dr.Pointer(ParseNode, store='gold_parse_nodes')
    auto_parse = dr.Pointer(ParseNode, store='auto_parse_nodes')

class Doc(dr.Doc):
    doc_id = dr.Field()
    tokens = dr.Store(Token)
    sentences = dr.Store(Sentence)
    gold_parse_nodes = dr.Store(ParseNode)
    auto_parse_nodes = dr.Store(ParseNode)

Figure 4.4: An example DOCREP schema definition using the Python API. This example uses the same schema as the abstract example given earlier in Figure 4.1.

fully-fledged Python classes — they are not external schema definitions as would be the case when using Protocol Buffers or Thrift, or if UIMA provided a Python API.

The interaction between Python classes and the DOCREP annotation type declarations was heavily inspired by the ORM used in the Django project,\footnote{https://www.djangoproject.com/} a popular Python web framework. Python metaclasses and class attributes are used to provide all of the underlying machinery to convert a Python class definition into a DOCREP annotation type. A Python class becomes a DOCREP annotation type by subclassing from `dr.Ann`. Class attributes of the form `dr.X` dictate what member variables on the object the DOCREP serialisation and deserialisation process should know about. Developers are free to add additional member variables and methods, as well as subclass from additional classes and act as parent classes.

dr.Pointer, dr.Pointers, and dr.Store all have one required argument — a reference to the DOCREP annotation type that the attribute deals with. dr.Slice takes this same argument if it is an annotation slice, not a byte slice. If the target store for a pointer is ambiguous, as is the case for the ParseNode pointers on the Sentence
4.3. Implementing the DOCREP APIs

```python
with open(..., 'rb') as f:
    reader = dr.Reader(f, Doc)
    for doc in reader:
        process_doc(doc)

with open(..., 'wb') as f:
    writer = dr.Writer(f, Doc)
    for doc in generate_docs():
        writer.write(doc)
```

Figure 4.5: An example of how document reading and writing is performed using the DOCREP Python API.

```python
def process_doc(doc):
    for sentence in doc.sentences:
        unique = set()
        for token in doc.tokens[sentence.span]:  # Slice being used here.
            unique.add(token.raw)
        print('Found', len(unique), 'unique token(s).')
```

Figure 4.6: Line 4 shows DOCREP slices being used with slice semantics.

object in Figure 4.4, the name of the store must be provided. If the target store is not ambiguous, it is deduced by the DOCREP runtime. An example unambiguous store deduction is the token attribute of the ParseNode annotation type, and an example ambiguous deduction is the gold_parse attribute of the Sentence annotation type.

Reading and writing DOCREP documents is quite straightforward, as shown in Figure 4.5. The dr.Reader and dr.Writer classes both take the file object on which to operate, as well as the document type that it expects incoming or outgoing document objects will be instances of. The dr.Reader class implements the Python iterator protocol, meaning it can be idiomatically placed on the right hand side of the foreach loop, as shown in this aforementioned example. The uncommon situation of processing streams containing heterogeneous document schemas can be dealt with by constructing multiple dr.Reader or dr.Writer objects.

Another idiomatic aspect of the Python API is how the DOCREP concept of a slice is implemented. Python has a primitive data type also called a slice, which consists of three integers: a start value, an end value, and a step size. DOCREP slices are instantiated as Python slice objects (with a step size of 1), meaning they can be used with standard Pythonic “slice semantics”. Line 4 of Figure 4.6 shows an example of
Figure 4.7: An example of how to use the automagic reader in the DOCREP Python API. Class definitions are inserted into the Python runtime on the fly based on the schema definitions in the read-in DOCREP.

The list of all Token objects on the document is indexed (“sliced”) by a slice object, yielding all objects located at an index between the start and end values of the slice.

The Python API supports some additional functionality that the other two provided APIs do not, taking full advantage of the dynamic nature of Python. While working with the early releases of DOCREP within our research lab, we often found that we wanted to be able to quickly load the documents contained within a DOCREP stream into a Python interpreter session to quickly perform some analysis. We wanted to do this interactively, without having to invest the effort to define the schema for the annotation layers we cared about. This lead us to create the automagic reader (Nothman et al., 2014). Being an interpreted, dynamic programming language, Python allows classes to be defined on the fly at runtime. Taking full advantage of this, the automagic reader creates class definitions based on the schemas of the documents on the input stream. This allows the user to read in documents of potentially heterogeneous types, on the fly, without having to explicitly define the schema. Figure 4.7 shows an example use of the automagic reader. While this is very attractive from a scripting and ease of use perspective, this functionality comes with the additional runtime cost of having to fully deserialize all annotation layers. Without a schema, the DOCREP reader does not know which annotation layers the user does and does not care about, so all layers are fully deserialized. This full deserialization on top of runtime class creation means that the automagic reader is slower than the normal reader, but still a very useful tool to have available when working with DOCREP streams.

\[17\] “dogfooding”
4.3. Implementing the DOCREP APIs

4.3.2 The Java API

Our provided Java API supports Java ≥ 6, and is available from the Maven repository,\textsuperscript{18} making it installable via the standard Java package manager \texttt{mvn}. At the current release (0.2.0), the Java API utilises the official MessagePack Java library as it provides sufficient APIs to implement lazy serialisation.

Figure 4.8 shows an example of each of the DOCREP data modelling concepts (Section 4.1) being used in the Java API. Interfaces as well as field and class annotations are used to provide all of the underlying machinery to transform Java class definitions into DOCREP annotation types. The lack of support for multiple inheritance in Java means that we could not require the user to subclass from one of our classes, as they might already have their own existing inheritance hierarchy. As such, a Java class becomes a DOCREP annotation type implementing the \texttt{Ann} interface and being annotated with \texttt{dr.\textbackslash Ann} annotation. We provide a base implementation of the \texttt{Ann} interface, \texttt{AbstractAnn}, for users who do not have a pre-existing inheritance hierarchy that needs to be preserved. Member variables annotated with a \texttt{dr.X} annotation dictate which members on the object the DOCREP serialisation and deserialisation process should know about. Objects of these classes are free to add additional member variables and methods. This scaffolding is similar to other Java serialisation mixin libraries, such as Hibernate.\textsuperscript{19}

Note that the member variable names used in Figure 4.8 differ from those used in the corresponding Python example (Figure 4.4). Each language uses idiomatic naming conventions. DOCREP still facilitates cross-language and cross-application portability, bypassing any potential issues of these names being different, through the use of runtime-configurable name mappings (Sections 3.2.6 and 4.4.2).

It is important to note here that the way the user interacts with Java classes which are DOCREP annotation types is vastly different from the way they interact with Java

\textsuperscript{18}http://mvnrepository.com/artifact/org.schwa/libschwa-java/
\textsuperscript{19}http://hibernate.org/orm/
import java.util.List;
import org.schwa.dr.*;

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

@dr.Ann
public class ParseNode extends AbstractAnn {
    @dr.Field
    public String label;
    @dr.Pointer
    public Token token;
    @dr.SelfPointer
    public List<ParseNode> children;
}

@dr.Ann
public class Sentence extends AbstractAnn {
    @dr.Field
    public Slice<Token> span;
    @dr.Pointer(store="goldParseNodes")
    public ParseNode goldParse;
    @dr.Pointer(store="autoParseNodes")
    public ParseNode autoParse;
}

@dr.Doc
public class Doc extends AbstractDoc {
    @dr.Field
    public String docId;
    @dr.Store
    public Store<Token> tokens;
    @dr.Store
    public Store<Sentence> sentences;
    @dr.Store
    public Store<ParseNode> goldParseNodes;
    @dr.Store
    public Store<ParseNode> autoParseNodes;
}

Figure 4.8: An example DOCREP schema definition using the Java API. This example uses the same schema as the abstract example given earlier in Figure 4.1. Constructor declarations have been removed for brevity.

import java.io.InputStream;
InputStream in = ...;
Reader<Doc> reader;
reader = Reader.create(in, Doc.class);
for (Doc doc : reader)
    processDoc(doc);

import java.io.OutputStream;
OutputStream out = ...;
Writer<Doc> writer;
writer = Writer.create(out, Doc.class);
for (Doc doc : ...)
    writer.write(doc);

Figure 4.9: An example of how document reading and writing is performed using the DOCREP Java API.
4.3. Implementing the DOCREP APIs

classes which are UIMA annotation types. In UIMA, the user defines their annotation types in an XML file and then runs the external jcasgen program to convert this XML file into source code for the corresponding Java class. The generated source can then be copied into the user’s codebase. This machine-generated Java class cannot be altered by the user in any way — they cannot change its inheritance hierarchy to be their own, nor can they easily add methods or member variables to the class. Even adding simple methods to annotation classes, such as isLeaf or isRoot on a ParseNode class, can improve usability substantially.

The Slice and Store DOCREP classes require a generic type which implements the Ann interface. As was the case in the Python API, if the target store for a pointer is ambiguous, the user must specify which store the pointer refers to. The dr.Pointer and dr.Slice annotations can take an optional argument to specify the store name, as is shown in dr.Pointer-annotated members on the Sentence class in Figure 4.8.

Like the Python API, reading and writing DOCREP documents is quite straightforward, as shown in Figure 4.9. The Reader and Writer classes both take a file object on which to operate, as well as the document type that it expects incoming or outgoing document objects will be instances of. The Reader<T> class implements the java.lang.Iterable<T> interface, meaning it can be idiomatically placed on the right hand side of the foreach loop.

4.3.3 The C++ API

Our provided C++ API is implemented in C++11 and does not require any other libraries to be installed (e.g. Boost\textsuperscript{20} or ICU\textsuperscript{21}). The C++ API is configured and built using the standard GNU Autotools build system. As a result, the installation process is the standard ./configure, make, and make install cycle familiar to most UNIX developers. Source code releases are available,\textsuperscript{22} and are also available via Homebrew

\textsuperscript{20}http://www.boost.org/
\textsuperscript{21}http://site.icu-project.org/
\textsuperscript{22}https://github.com/schwa-lab/libschwa/releases
on Mac OS X. The source code releases also provide scripts to generate Debian and
RedHat compatible package bundles.

In accordance with C++ practice, efficiency and flexibility are of great importance
in the C++ DOCREP API. We implemented our own C++ MessagePack library as the
official library at the time did not provide sufficient APIs for us to implement lazy
serialisation (Section 3.2.3). MessagePack is a relatively simple binary serialisation
format, so implementing serialisation was not a technical burden.

Figure 4.10 shows an example of each of the DOCREP data modelling concepts
(Section 4.1) being used in the C++ API. Unlike Python and Java, C++ offers no form
of runtime class introspection, so a lot of the automatic schema generation mappings
that happen behind the scenes in the Python and Java APIs are not possible to compute
programatically. As a result, the schema mappings always need to be explicitly defined
in C++. The schema mappings dictate how the DOCREP serialiser and deserialiser map
between fields on the wire and member variables on objects. Any C++ class becomes
a DOCREP annotation type by inheriting from dr::Ann. A subclass of dr::Ann must
also define an inner class Schema, which has member variables defining the schema
mappings. The DOCREP macros DR_FIELD, DR_POINTER, DR_SELF, and DR_STORE
allow users to easily establish these schema mappings. As a technical side note, the
Schema inner-classes are not declared inline due to a forward-reference limitation
within the C++ language.

Like in the Java API, the dr::Slice and dr::Store DOCREP classes require a
template type which inherits from the dr::Ann class. Since the schemas have to be
explicitly defined in C++, the store ambiguity problem present in the Python and Java
APIs is not present — all DR_POINTER schema mapping declarations must provide the
target store anyway.

The location in memory of deserialised DOCREP annotations cannot be controlled
in the Java or Python APIs as these languages do not allow fine-grained memory
allocation strategies. We want the C++ DOCREP API to be fast while also being familiar
### 4.3. Implementing the DOCREP APIs

```cpp
#include <schwa/dr.h>

namespace dr = ::schwa::dr;

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

class ParseNode : public dr::Ann {
public:
  std::string label;
  dr::Pointer<Token> token;
  dr::Pointers<ParseNode> children;
  class Schema;
};

class Sentence : public dr::Ann {
public:
  dr::Slice<Token **> span;
  dr::Pointer<ParseNode> gold_parse;
  dr::Pointer<ParseNode> auto_parse;
  class Schema;
};

class Doc : public dr::Doc {
public:
  std::string doc_id;
  dr::Store<Token> tokens;
  dr::Store<Sentence> sentences;
  dr::Store<ParseNode> gold_parse_nodes;
  dr::Store<ParseNode> auto_parse_nodes;
  class Schema;
};

class Token::Schema : public dr::Ann::Schema<Token> {
public:
  DR_FIELD(&Token::span) span;
  DR_FIELD(&Token::raw) raw;
};

class ParseNode::Schema : public dr::Ann::Schema<ParseNode> {
public:
  DR_FIELD(&ParseNode::label) label;
  DR_POINTER(&ParseNode::token, &Doc::tokens) token;
  DR_SELF(&ParseNode::children) children;
};

class Sentence::Schema : public dr::Ann::Schema<Sentence> {
public:
  DR_POINTER(&Sent::span, &Doc::tokens) tokens;
  DR_POINTER(&Sent::gold_parse, &Doc::gold_parse_nodes) gold_parse;
  DR_POINTER(&Sent::auto_parse, &Doc::auto_parse_nodes) auto_parse;
};

class Doc::Schema : public dr::Doc::Schema<Doc> {
public:
  DR_FIELD(&Doc::doc_id) doc_id;
  DR_STORE(&Doc::tokens) tokens;
  DR_STORE(&Doc::sentences) sentences;
  DR_STORE(&Doc::gold_parse_nodes) gold_parse_nodes;
  DR_STORE(&Doc::auto_parse_nodes) auto_parse_nodes;
};
```

Figure 4.10: An example DOCREP schema definition using the C++ API. This example uses the same schema as the abstract example given earlier in Figure 4.1. Constructor declarations have been removed for brevity.
and easy to use as a C++ developer. As such, one must consider how to arrange objects in memory while satisfying these constraints. The four main options are:

**Contiguous objects**  Annotation objects are laid out one after another in a contiguous block of memory (e.g. `std::vector<dr::Ann>`). This option, illustrated in Figure 4.11a, is the most efficient in terms of space and access (single dereference for iteration, and the next and previous can be calculated in a single step), but is much more costly for insert/delete because of the reallocation and copy/move of larger objects. C++11 alleviates this problem slightly with move semantics, but ultimately there still more memory to move. A disadvantage of this option is that any resizing of the underlying array potentially invalidates all pointers into the array. This could be combated by pointer swizzling and unswizzling before and after resize, but requires knowledge of all pointers into the array.

**Contiguous pointers to objects**  Annotation objects are allocated randomly but pointed to by a contiguous block of pointers (e.g. `std::vector<dr::Ann *>` or alternatively `boost::stable_vector<dr::Ann>`). This option, illustrated in Fig-
4.3. Implementing the DOCREP APIs

Implementing 4.11b, is more expensive in terms of space and execution, requiring an extra 8 bytes per annotation on a 64-bit architecture and a second dereference. This option does not allow access the previous or next annotation objects directly from an annotation object (via pointer arithmetic). One way to combat this is to define and set previous and next pointers on the annotation objects. Another strategy is to store a pointer back to the array on the annotation objects. From this, an annotation could perform a $O(1)$ lookup to locate its neighbours. However, both of these strategies are unsafe across resize.

**List of objects** Annotation objects are allocated randomly but are connected through a singly or doubly linked list (e.g. `std::list<dr::Ann>`), or through intrusive lists (e.g. `boost::intrusive`). This option, illustrated in Figure 4.11c, is more expensive in space and access than the previous two options. This is true even for the intrusive form which is the most efficient: annotations require two extra pointers (for previous and next), and random access is no longer available. However, this is not necessarily a problem since NLP applications often use context that is close to the current annotation. Insert and delete operations are cheap in this representation as only one or two pointers need to be updated to adjust the linked list structure.

**List of contiguous objects** This strategy, illustrated in Figure 4.11d, is a hybrid of the first and third strategy designed to minimise the number of memory copies required when constructing annotations. If a DOCREP producers knows that it is about to create $n$ objects in a store, this strategy allows for an array of $n$ objects to be allocated and linked to the existing blocks of objects in the store via a pointer. As such, insertions do not require a copy of the existing objects in the store as reallocation is not performed. Within the group of contiguous objects, previous and next annotations are accessible via pointer arithmetic.
Chapter 4. DOCREP: Implementation

```cpp
#include <iostream>

std::istream &in = ...;
std::ostream &out = ...;
Doc::Schema &schema = ...;

dr::Reader reader(in, schema);
dr::Writer writer(out, schema);
while (true) {
    Doc doc;
    if (!(reader >> doc))
        break;
    mutate_doc(doc);
    writer << doc;
}
```

Figure 4.12: An example of how document reading and writing is performed using the DOCREP C++ API.

If the annotations are particularly large or dynamically generated, then contiguous objects is problematic. For instance, you would not use annotations directly to represent all of the named entities considered during Viterbi decoding of a named entity tagger. DOCREP is more designed for linguistic structures after they have been determined by the applications, rather than as an application-internal representation. However, the contiguous objects approach is a sensible strategy if the DOCREP annotations are being used in a read-only manner since reallocations are not an issue.

For the DOCREP C++ API, we have decided to implement two strategies. The C++ API supports both the contiguous objects strategy (dr::Store), primarily targeted at DOCREP consumers, as well as the list of blocks of contiguous objects strategy (dr::BlockStore), targeted at DOCREP producers. DOCREP producers are still able to use the contiguous object strategy, but developers need to be aware that pointers into the underlying vector are not stable until reallocations have ceased. One common strategy here is to store swizzled pointers to annotations while the vector is unstable, and deswizzle these pointers once annotation object allocation has finished.

Like both the Python and Java APIs, reading and writing DOCREP documents is quite straightforward, and performed in an idiomatic C++ manner. An example of this is shown in Figure 4.12. The dr::Reader and dr::Writer classes both take a reference
4.3. Implementing the DOCREP APIs

#include <iostream>
#include <string>
#include <unordered_set>

void process_doc(const Doc &doc) {
    for (auto &sentence : doc.sentences) {
        std::unordered_set<std::string> unique;
        for (auto &token : sentence.span) // Slice being used here.
            unique.insert(token.raw);
        std::cout << "Found " << unique.size() << " unique token(s)." << std::endl;
    }
}

Figure 4.13: Line 9 shows DOCREP annotation slices being used as an iterator.

to the stream on which to operate, as well as the document schema instance to use for field mapping. While it would be possible to add support for the C++11 foreach loop syntax to the dr::Reader class, doing so would impose how memory will be managed with regards to the read-in document objects. Instead, we support the standard C++ iostream >> and << operators for reading and writing, and leave memory management to the caller.

Annotation slices are implemented as a very simple start and end pointer pair which also supports the C++11 enhanced for loop protocol (“foreach loop”). This is very handy as one often wants to iterate through all annotation objects contained within a span. An example of this is shown on line 9 of Figure 4.13. Another nice artefact of slices being implemented as a start and end pointer pair is that the length of the span can be computed by simple pointer arithmetic, assuming the contiguous objects store is being used rather than the list of contiguous objects store.

4.3.4 Consistency and idiomaticity

During the implementation of the DOCREP APIs, we aimed to make the interface as similar as possible between the three languages, while still feeling idiomatic within that language. Figures 4.4, 4.8, and 4.10 show an example set of identical schema definitions in Python, Java, and C++. To save referring back to these figures, we have copied the definition of the ParseNode annotation type in each language into Figure 4.14.
Figure 4.14: Comparing the definition of the example ParseNode annotation type in all three DOCREP APIs.

Comparing these three implementations, we can see that the way in which a class is “adorned” with DOCREP metadata is very similar. In both Python and Java, where classes are able to be introspected at runtime, we have utilised features provided by the language to automatically induce default schema mappings. In the case of C++ where schema mappings need to be explicitly defined, the manner in which they are defined is consistent with the manner in which they are defined in Python and Java. This is even more obvious when the user needs to intervene with the automatic schema induction, such as when ambiguous stores exist. The manual specification of the target stores for pointers is identical across all three APIs.

4.4 The DOCREP runtime

Having implemented and discussed the language-specific aspects of the DOCREP API in each of our three target programming languages, we analyse the aspects of the DOCREP API that are common across all languages. In particular, this section discusses
aspects of the extended functionality the DOCREP runtime and how the APIs support these operations behind the scenes.

### 4.4.1 Lazy serialisation

Lazy serialisation (Section 3.2.3) allows an application to work with a subset of the annotations on an existing DOCREP document efficiently. For instance, given a DOCREP version of the OntoNotes corpus, a user might want to run a named entity recogniser over the documents and store the produced tags. For this task, the application only need to know, or care about, the token and sentence annotation layers. Anything written by the user should be appended — that is, the original annotations layers should be retained despite the application not knowing about them.

There are two different forms of lazy serialisation (Section 3.2.3) implemented by the DOCREP APIs. The first is store-level laziness. If a store exists on a DOCREP document that is being read but there is no corresponding definition of that store in the application’s coded schema, the DOCREP API does not deserialise the store. Instead, it simply retains this store in its serialised form so that it can be written out verbatim when the document is later re-serialised. The second kind of lazy serialisation is field-level laziness. If a field exists on a read-in annotation that does not exist in the application’s coded schema, the DOCREP API again does not perform the deserialisation of this field. Like store-level laziness, the serialised version of the field is retained and written out verbatim when that annotation object is later re-serialised. The combination of these two forms of lazy serialisation allow DOCREP applications to define only the annotation layers and annotation fields they care about, and allow any other annotation layers and fields which exist on read-in documents to be dealt with transparently.

A convenient side effect of having infrastructure keep track of the original serialised representations of incoming stores and fields is that it gives us a simple way to optimise read-only DOCREP consumers. If the user defines that a store or a field should be read-only, we know that this store or field will not change at runtime from what was
**Figure 4.15:** The relevant code snippets from the DOCREP C++ and Java APIs to support store and field-level lazy serialisation.

originally read in. As such, the DOCREP API also keeps track of the original serialised forms of any fields marked as read only and uses these during serialisation.

The base classes in the APIs described earlier, (Abstract)Ann and (Abstract)Doc, store these serialised payloads. In addition to this, the base document class may also store any necessary schema information about application-unknown annotation types which appear in an input document. If an annotation type exists on an incoming document that the application is not aware of, the wire definition for this annotation type needs to be retained so that when the document is re-serialised, the header information for this annotation type can be output (Section 4.2.1).

Figure 4.15 shows the relevant declaration code snippets from the DOCREP C++ and Java APIs which facilitate lazy serialisation. Note this is not user-defined custom code — these snippets are taken from the implemented DOCREP APIs which the user inherits from. Similar code exists in the Python API but is not included for brevity. We will discuss implementation details in terms of the C++ API.

Document and annotation objects have three hidden member variables: one to store the serialised payload (_lazy), one to store how many bytes are in the serialised payload (_lazy_nbytes), and one to store how many MessagePack objects are in the
4.4. The DOCREP runtime serialised payload (\_lazy_nelem). On a 64-bit architecture, this equates to 16 bytes of overhead per annotation object, which is a small trade-off given the usability lazy serialisation provides. The schema information about application-unknown annotation types and fields is stored in the runtime manager, housed on the Doc object (\_rt). The Reader and Writer classes interact with this runtime manager, facilitating lazy serialisation across a read-mutate-write document life cycle.

To demonstrate how these attributes are used to implement lazy serialisation, we will briefly explain how field-level laziness is implemented. Store-level lazy serialisation is implemented in a similar fashion. Recall from the wire protocol implementation details (Section 4.2.1) how a single annotation instance is serialised:

\[
\text{<instance>} ::= \{ <field_id> : <obj_val> \} \quad \# \text{Zero or more element map.}
\]

This is a MessagePack map structure with one key-value pair per annotation attribute. In MessagePack, a map of cardinality \( n \) is serialised as \( 2n + 1 \) consecutive MessagePack objects: the cardinality (\( n \)) followed by key\(_0\), value\(_0\), key\(_1\), value\(_1\), \ldots, key\(_n\), value\(_n\). When serialising an annotation, the serialisation process needs to know the cardinality of this map; that is, how many attributes are on the annotation. The total number of attributes on an annotation is the number of attributes the application knows about plus the number of lazily stored attributes; \_lazy_nelem. After serialising the cardinality, the serialisation process can then write out verbatim the \( 2 \times \_lazy_nelem \) already-serialised MessagePack objects stored in \_lazy. The \_lazy_nbytes value is needed as the \_lazy binary string may contain intermediate \text{NUL} values, so it cannot be \text{NUL}-terminated. Following the lazy attributes, the attributes that the application knows about can then be serialised.

In addition to facilitating lazy serialisation, the DOCREP APIs in each language also provide runtime access to the lazily stored annotations and attributes. This access is not optimised for efficiency, but allows for dynamic applications to be developed. For example, this runtime access powers the dynamic components of a number of
Figure 4.16: Different variable naming conventions potentially cause issues with automatic schema generation. The runtime schema mappings provided by DOCREP alleviate this issue, allowing `gold_parse` and `goldParse` to be mapped to one another.

the DOCREP command-line tools (Section 3.1.5), such as `dr count`, `dr grep`, and `dr srcgen`. These tools are covered in depth in Section 6.2.

### 4.4.2 Configurable wire to schema mappings

In our three earlier example schema implementations (Figures 4.4, 4.8, and 4.10), idiomatic naming conventions for member variables were used in each language. The Sentence class from the Python and Java examples have been reproduced in Figure 4.16, removing aspects that are not important for this discussion.

In the Python case, the DOCREP API will be looking for a field named `gold_parse` on the wire, and in the Java case, the API will be looking for a field named `goldParse`. For both cross-language portability and cross-application portability, it is necessary to provide a way to map between names on the wire and class and member variable names defined in code. The DOCREP schema mappings provide exactly this — a configurable mapping between names on the wire and names in code (Section 3.2.6). Specifically, for a schema $S$ defined by an application, if there exists a subset of another schema $S'$ which is isomorphic to $S$, then the combination of runtime renaming and lazy serialisation (Section 4.4.1) provided by DOCREP allows $S'$ to be used in the context of $S$ with no runtime performance degradation.

Users are able to programatically change these mappings to adapt their DOCREP schemas to other isomorphic schemas. In each of our three APIs, we have provided
integrations with a standard command-line argument parsing library so that these
wire-to-code mappings can be specified at runtime as simply as possible. For example,
if we had an existing Java application which produced annotations containing the
Sentence type defined in Figure 4.16 and a Python application with a definition of a
Sentence annotation type as per Figure 4.16. We would like the Python application to
be able to consume the DOCREP produced by the Java process. Since the two schemas
are isomorphic, we can provide the appropriate name mapping on the command-line:

```
$ ./my-application.py --dr--Sentence--gold_parse goldParse < corpus.dr
```

In this example, our renaming is simply a camelCase to underscore_case renaming,
but DOCREP name mappings support arbitrary renaming.

Another very useful case for renaming is the second scenario described in Sec-
tion 3.2.6. This scenario was as follows: imagine that a user has reimplemented the
parser bracketing evaluation script evalb to be DOCREP-aware, and uses the store
named parses to store the ParseNode instances which are used to form parse trees.
Imagine also that a user was evaluating a number of different parsers and has a DOCREP
document with a number of different ParseNode stores; one for each set of parse trees.
The user would like to be able to use the evalb implementation though their store
names to not align with the store name used in evalb. This scenario is handled by
runtime renaming, illustrated here via command-line argument renaming integration:

```
$ ./evalb --dr--Doc--parses gold_pareses_nodes < corpus.dr > gold.evalb
$ ./evalb --dr--Doc--parses auto_pareses_nodes < corpus.dr > auto.evalb
```

This dynamic mapping between DOCREP annotation type and field names provides
great flexibility for interoperability between cross-language and cross-application
components. Allowing users to use idiomatic variable names independent of the wire
enhances the usability of DOCREP as developers are allowed to write code using their
normal style and naming conventions without the framework getting in their way.
Additionally, this facilitation of renaming should allows further substitution of tools
and services in and out of various NLP applications.
The power provided by the combination of runtime renaming and lazy serialisation is considerable — it allows for generic schemas to be implemented in applications and used in the context of any isomorphic schema. This helps to solve one of the great problems in NLP of passing a document through a number of disjoint NLP applications while retaining all of the existing annotations at each point. Currently, developers often have to “slice and dice” various data formats to provide tools only the annotations they require in specific data formats, and then splice the produced output back into a single collated representation. DOCREP lazy serialisation and runtime renaming almost entirely solves this tedious, error prone problem.

4.4.3 Decorators

The DOCREP API provides decorators to make working with structured linguistic annotation as easy as possible. Decorators remove the need for users to write the same repetitive linguistic boilerplate code, abstracting out common operations to modular, reusable, templated units. Decorators also facilitate the storage of normalised annotations in the DOCREP serialisation format, by providing a convenient mechanism to place derived attributes onto objects at runtime from the normalised representation. This is in contrast to methodologies observed in many UIMA projects where non-normalised pointers utilised for graph traversal are often attributes of an annotation type definition due to the rigidity of the UIMA type system and lack of control the user has on the corresponding runtime class (Section 2.3.3). The decorator concept was developed in collaboration with Joel Nothman, as noted earlier in Section 1.4.

A DOCREP decorator performs common operations over all annotation objects in an annotation store. Depending on the user’s data model, the use cases for built-in decorators will vary. The decorator concept is implemented in the Python and C++ APIs. Decorators are not implemented in the Java API as obtaining pointers to member variables is not something that Java supports elegantly, but the decorator definition makes judicious use of. For example, the simple \&MyClass::my_member expression
4.4. The **DOCREP** runtime

To achieve this in C++ becomes `MyClass.class.getDeclaredField("myMember")` in Java in addition to having to handle two checked exceptions on `getDeclaredField`. Instead of implementing a poor version of the decorators in Java, we have left them until a more elegant and idiomatic solution in Java is devised.

We will demonstrate decorator usage using the `Token` and `Sentence` annotation type definitions previously listed in Figure 4.10. The stored relationship between a sentence instance and a token instance is unidirectional; it is defined by the slice attribute on the sentence object which points to two token objects. Given a token object, finding its corresponding sentence object requires a $O(\log n)$ binary search over the store of sentence objects (assuming an ordering comparison between two sentence objects can be performed in $O(1)$). It is not uncommon for the token to want to know what sentence object it is spanned by. For example, when performing feature extraction in a machine learning context, a token might want to know whether it is the first or last token in the sentence. The data model should not need to store this relationship twice, once in each direction, in order to facilitate this bidirectional access. All of the information necessary to support this “slice inversion” is available at runtime.

The “reverse slices” decorator iterates through every object in an annotation store, and for the given slice field, inverts the references for each object contained within the slice. That is, for a slice field $S$ on annotation type $A$ which slices over annotations of type $B$, this decorator sets a pointer member variable on each $B$ object contained with $S$ to point to the $A$ object the slice is from. To illustrate how this is used, Figure 4.17 shows C++ snippet where this decorator is implemented as the `DR_REVERSE_SLICES` macro. There are three interesting segments in this example. Line 5 adds a `Sentence` pointer member variable to the `Token` class which will later be populated to point back to the sentence object that the token is spanned by. This `sentence` member variable is not added to `Token::Schema` since it does not play a part in serialisation. Lines 11 to 13 create an instance of the reverse slices decorator, providing to it the annotation store to iterate through, the annotation store that the target slice points into, the slice member
class Token : public dr::Ann {
public:
    dr::Slice<uint64_t> span;
    std::string raw;
    Sentence *sentence; // Added this field so Token can point back to Sentence.
};

// Define the decorator instance.
const auto REVERSE_SENTENCE_SLICE = DR_REVERSE_SLICES(&Doc::sentences, &Doc::tokens,
                                                      &Sentence::span, &Token::sentence);

...
4.4. The DOCREP runtime

```cpp
class Token : public dr::Ann {
public:
  dr::Slice<uint64_t> spn;
  std::string raw;
  NamedEntity *ne; // Pointer back to the NamedEntity instance.
  std::string ne_label; // Sequence tag encoded named entity label.
  class Schema;
};

class NamedEntity : public dr::Ann {
public:
  dr::Slice<Token *> span;
  std::string category; // The category of the named entity (e.g. "LOC").
  class Schema;
};
...
// Define the decorator instance.
const auto SEQUENCE_TAG_NES =
  DR_SEQUENCE_TAGGER(&Doc::named_entities, &Doc::sentences, &Doc::tokens,
                   &NamedEntity::span, &Sentence::span, &Token::ne,
                   &NamedEntity::category, &Token::ne_label);
...
reader.read(...);
while (true) {
  Doc doc;
  if (!reader >> doc)
    break;
  // Decorate the document.
  REVERSE_SENTENCE_SLICE(doc);
  SEQUENCE_TAG_NES(doc, SequenceTagEncoding::IOB2);
  process_doc(doc);
}
```

Figure 4.18: An example use of the “sequence tagger” decorator.
sequence tags, given some sequence tag encoding (IOB1, IOB2, BMEWO, etc.) For example, given a named entity annotation type which spans over tokens, the sequence tagger decorator is able to project onto the tokens, the appropriate IOB tag for that token, taking into account the category of the named entity and the sequence tag encoding in question.

An example use of this decorator is shown in Figure 4.18 where the named entity example just described is portrayed in code. Lines 6 to 7 add a `NamedEntity` pointer which will be later populated to point back to the named entity object that the token is spanned by, and a string for the sequence tag encoded named entity category. Lines 20 to 23 create an instance of the sequence tagger decorator. This decorator requires lots of information regarding annotation relationships and their attributes mainly due to having to support IOB1 encoding, which requires knowledge about immediately adjacent spanning objects (see Section 8.1.1 for more information). Lastly, line 34 uses the defined sequence tagger instance.

Decorators provide the user with flexible runtime views of annotation graph structures while only storing normalised data at rest. Most derived attributes users wish to construct are handled by just a handful of decorators. The DOCREP APIs provide a number of built-in decorators to handle scenarios we identified while using DOCREP.

### 4.5 Summary

In this chapter, we made concrete all of our design goals and principals outlined in the previous chapter. Section 4.1 described, in an abstract sense, how DOCREP models documents and annotations and how each of these modelling units compose together to form a complete DRF. Section 4.2 then described our developed wire protocol for serialising and deserialising DOCREP documents, ensuring it supported the modelling concepts presented earlier as well as facilitated all of our design needs. Consideration
of the properties of candidate serialisation formats as well as efficiency experiments led us to choose MessagePack for serialisation.

With these two fundamental implementation decisions of DOCREP made, Section 4.3 then described the implementation the DOCREP APIs in our three target programming languages and compared language specific aspects of their implementation. In making the DOCREP APIs familiar across languages while idiomatic in each, we had to tackle their different conventions and needs, such as naming conventions, memory management, how references to objects are handled, how annotation slices should be implemented, and runtime efficiency considerations. Section 4.4 concluded this chapter with a discussion about the problems that the DOCREP runtime solves in NLP application development:

- we solved the problem that applications often only need a fraction of the annotation layers associated with a document through lazy serialisation (Section 4.4.1);
- we solved the problem of isomorphic schemas with different names through runtime-configurable wire to schema mappings (Section 4.4.2);
- we solved the problem of storing normalised representations for efficiency but requiring derived values at runtime through the use of decorators: pluggable templated annotation graph traversal functions (Section 4.4.3).

With the implementations of DOCREP in place, we need to evaluate how well they perform from multiple perspectives. The DOCREP APIs need to support the creation and manipulation of corpora containing multiple varied annotation layers. Next, in Chapter 5, we evaluate the ability for DOCREP to represent the OntoNotes 5 corpus, and demonstrate how it meets its design goals while also being significantly more efficient that the existing DRF solutions. In addition to the ability to model and process complex corpora, the APIs need to be evaluated in terms of their ease of use as a developer, and whether they are powerful enough to perform real-world CL research and develop real-world NLP and LT applications. This evaluation is presented in Chapter 6.
5 Evaluating DOCREP on OntoNotes

Computational linguistics is increasingly a data-driven research discipline with researchers using diverse collections of large-scale corpora (Parker et al., 2011; Gabrilovich et al., 2013). 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, parse trees, and coreference relations. 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.

The OntoNotes corpus is a large manually-annotated corpus containing cross-language and cross-genre documents, with each document having multiple annotation layers. To evaluate the effectiveness of DRFs, in particular DOCREP, we construct data models in both UIMA and DOCREP for representing the OntoNotes corpus. We discuss the adequateness of the data models with respect to their linguistic representation and argue that while these two DRFs are similar in their data modelling components, DOCREP provides more convenient components for linguistic modelling.

In addition to linguistic effectiveness, we compare the usability and efficiency of using a DRF to store and process complex corpora compared to more traditional corpus distribution formats. To evaluate the effectiveness of DRFs in particular DOCREP we construct data models in both UIMA and DOCREP for representing the OntoNotes corpus. The data model used in here is taken directly from the OntoNotes corpus and documentation. We show that DOCREP outperforms all other distribution formats.
Chapter 5. Evaluating DOCREP on OntoNotes

while facilitating reproducibility of experiments and the quality assurance of data. We also evaluate DOCREP against the design requirements outlined in Chapters 2 and 3.

5.1 The OntoNotes 5 corpus

OntoNotes (Hovy et al., 2006; Weischedel et al., 2011) is a large corpus of linguistically-annotated documents from multiple genres in three different languages. At the time of writing, OntoNotes has had four additional releases after its initial release; each new release growing the size of the corpus and correcting identified annotation errors. The latest release (OntoNotes 5) 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, 2,002 are in Chinese, and 599 are in Arabic.

Each of the documents in the OntoNotes 5 corpus (Weischedel et al., 2013) contain multiple layers of syntactic and semantic annotations. The annotation layers include syntax, predicate-argument structure, named entity annotations, coreference, and word sense disambiguation layers. Figure 5.1 shows an example of the annotation layers applied to two sentences from the OntoNotes 5 corpus. The documents in the corpus have different subsets of these annotation layers due to the way OntoNotes was created: merging existing corpora together which annotated the same datasets. For the English documents, the original source corpora include the Penn Treebank (Marcus et al., 1993), PropBank (Palmer et al., 2005), and the work on named entities and coreference by Weischedel and Brunstein (2005) at BBN. For Chinese and Arabic, the source corpora include the Chinese Treebank (Xue et al., 2005) and the Arabic Treebank (Maamouri and Bies, 2004). The size of each of the annotation layers in the OntoNotes 5 corpus, broken down by language, is shown in Table 5.1.
Figure 5.1: An example of the multiple annotation layers provided by the OntoNotes corpus being overlaid onto a two sentence subset of a document.

<table>
<thead>
<tr>
<th>Annotation layer</th>
<th>Attribute</th>
<th>English</th>
<th>Chinese</th>
<th>Arabic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>Documents</td>
<td>13 109</td>
<td>2002</td>
<td>599</td>
</tr>
<tr>
<td></td>
<td>Tokens</td>
<td>2 919 605</td>
<td>1 133 460</td>
<td>438 422</td>
</tr>
<tr>
<td>Proposition</td>
<td>Documents</td>
<td>6124</td>
<td>1 861</td>
<td>599</td>
</tr>
<tr>
<td></td>
<td>Verb Prop.</td>
<td>301 656</td>
<td>148 396</td>
<td>29 643</td>
</tr>
<tr>
<td></td>
<td>Noun Prop.</td>
<td>18 596</td>
<td>6570</td>
<td>—</td>
</tr>
<tr>
<td>Named Entity</td>
<td>Documents</td>
<td>3637</td>
<td>1 911</td>
<td>446</td>
</tr>
<tr>
<td></td>
<td>Tokens</td>
<td>2 198 509</td>
<td>1 072 735</td>
<td>325 692</td>
</tr>
<tr>
<td>Coreference</td>
<td>Documents</td>
<td>2359</td>
<td>1 727</td>
<td>447</td>
</tr>
<tr>
<td></td>
<td>Tokens</td>
<td>1 750 313</td>
<td>1 039 969</td>
<td>326 466</td>
</tr>
<tr>
<td>Speakers</td>
<td>Documents</td>
<td>670</td>
<td>206</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Tokens</td>
<td>1 834</td>
<td>1 009</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 5.1: Size of each annotation layer in OntoNotes 5 by language.
Chapter 5. Evaluating DOCREP on OntoNotes

The annotations in OntoNotes 5 are distributed in two different formats. The first, more canonical format, is a series of flat files. The second format is an export of a MySQL database, where each of the annotation layers are stored in a completely normalised representation. Both of these distribution formats are undesirable for reasons we will now discuss.

5.1.1 Flat files

The flat file distribution of OntoNotes is as a series of 53,308 flat files. Each annotation layer for each document is distributed in its own file. Additionally, each annotation layer has its own file format. The result is that if the user wishes to use more than one annotation layer, they will need to write separate code to parse each format. As any software developer that deals with text corpora knows, writing corpus parsing code is error prone and tedious. Forcing the user to write parsing code for multiple file formats is undesirable and an unnecessary complication for the user. To make this situation worse, the user will then need to align the annotations in each layer. This would not be too much of a burden if the flat files all used the same segmentation, but unfortunately, they do not.

Figure 5.2 shows an example snippet from four different annotation layers for a document in the OntoNotes 5 corpus. All four of these annotation layers are in a different file format. The coreference and named entity formats might look to be similar at first glance, but upon closer inspection, there are segmentation differences. The parse and coreference annotation layers include trace nodes in their tokenization whereas the other annotation layers, named entity included, do not. This makes the alignment process that the user has to perform even more error prone and tedious.

One additional complication with parsing these file formats is that, like many linguistic corpora, there are documents which do not conform to the specified syntax or semantics of the file format. For example, the corpus states that the named entity
5.1. The OntoNotes 5 corpus

The parishioners of St. Michael and All Angels stop to chat at the church door, as members here always have.

Another women wrote from Sheffield to say that in her 60 years of ringing, I have never known a lady to faint in the belfry.

Figure 5.2: Snippets from various annotation layer flat files for the OntoNotes 5 document nw/wsj/00/wsj_0089@0089@wsj@nw@en@on.
Chapter 5. Evaluating DOCREP on OntoNotes

annotations are non-nested, yet there are five documents which contain nested entity definitions.

5.1.2 Database

The MySQL database version of the OntoNotes 5 corpus contains the annotation layers stored in a fully normalised database schema, spread across 55 tables. Figure 5.3 shows an entity-relationship diagram for the database schema taken from the “db-tool” documentation (Weischedel et al., 2013), though this diagram does not entirely match the distributed schema. Working with the database requires knowledge of how the tables are related to one another, as well as knowledge of SQL or access to an API for querying the database. Since the annotations are completely normalised, utilising more than one form of linguistic information, including simple combinations such as sentence bounds and tokens, requires joining data across tables.

This schema used in this database was not designed with efficiency in mind, with no indices being created and unusual data types chosen for certain columns. Additionally, the primary key in each table is a denormalised dynamic-length value, fully encoding the context of the database record, somewhat defeating the purpose of the normalised database representation. Figure 5.4 shows the how the tokens for one of our example sentences can be obtained from the database, as well as what the returned records contain. Note also the time taken to execute the query. This database is hosted on an unloaded, powerful machine, but the design of the schema and primary keys results in very poor performance.

The OntoNotes 5 distribution contains a “db-tool”, which is a Python program, API, and data models for interacting with the data location in the database. This tool might be useful for infrequent and explorative access of the database, but it is far too slow for any real-world use. The slowness stems from the way the tool accesses the database, performing a separate single-record query for each normalised value. Obtaining all
Figure 5.3: Entity-relationship diagram for the database version of the OntoNotes corpus. This image comes from the “db-tool” documentation which is distributed with OntoNotes 5 (Weischedel et al., 2013).
information attached to a single sentence with this tool often initiates thousands or sometimes even tens of thousands of queries to the database.

While using a relational database for representing linguistic annotations has its advantages, such as a normalised data model and no specific I/O formats to account for, there are many disadvantages. This relational database approach does not perform well when evaluated against the design criteria for linguistic representation outlined by Bird and Liberman (2001) and Ide et al. (2003) (Section 2.2). The scalability and searchability criteria can in theory be achieved with a relational database, but the design of the OntoNotes 5 schema does not facilitate this, as shown earlier. The browsability criteria states that annotations should be easily searchable. This is true for simple searches, but the combination of SQL and a normalised data model makes performing queries such as “which sentences match this regular expression?” or “which parse trees contain this tree pattern?” difficult and potentially inefficient. How disjoint sets of annotations stored in a relational database are merged together is not apparent, a requirement of the incrementality criteria.

While their intentions were good in providing an alternative distribution format to flat files, the creators of the OntoNotes database have either not considered these design
criteria for linguistic representation or they have chosen to ignore them. The database provides an alternative method for developers to ingest the linguistic annotations, but requires knowledge of SQL and the database schema. Overall, the experience as a developer interacting with this version is poor due to many factors, including ill-designed schemas and documentation not matching the distributed data.

5.2 Modelling OntoNotes in DOCREP and UIMA

We have seen that both of the data formats that the linguistically-rich OntoNotes 5 corpus is distributed in have usability issues. The flat files require significant software engineering overhead to parse and align the data across annotation layers, and the database schema is ill-designed and requires either knowledge of SQL or a sufficient API to access the data. Here we demonstrate how to model the OntoNotes corpus in a DRF, and outline why a DRF representation is superior to both flat files and database representations.

Our conversion process pulls data from the database version of the OntoNotes corpus as the alignment of data across annotation layers is less error prone than in the flat file version. Another advantage of this is that the tables in the database already specify a number of the fields the DRF models will need.

5.2.1 Overview

For this evaluation, we chose to model OntoNotes in two different DRF's so that their modelling decisions and the ability to model linguistic phenomena can be compared. We construct data models in both UIMA (Section 2.3.3) and DOCREP (Chapters 3 and 4).

The choices that were made on how to model the different annotation layers in OntoNotes were mostly identical in UIMA and DOCREP. The main difference occurs when you have an annotation over a sequential span of other annotations. An exam-
### Figure 5.5: Defining a named entity annotation type in UIMA.

```xml
<typeDescription>
  <name>ontoNotes5.to_uima.types.NamedEntity</name>
  <description/>
  <supertypeName>uima.tcas.Annotation</supertypeName>
  <features>
    <featureDescription>
      <name>label</name>
      <description>The NE label.</description>
      <rangeTypeName>uima.cas.String</rangeTypeName>
    </featureDescription>
  </features>
</typeDescription>
```

### Figure 5.6: Defining a named entity annotation type in DOCREP.

```java
@dr.Ann
public class NamedEntity extends AbstractAnn {
  @dr.Pointer public Slice<Token> span;
  @dr.Field public String label;
}
```

A common example of this situation is named entity annotations. In OntoNotes, named entities are represented as annotation spans over a sequence of successive token annotations.

UIMA does not provide a way to model this linguistic phenomena directly. The most common method that UIMA users choose to model this situation is as a normal `uima.cas.Annotation` subtype with its begin offset set to the begin offset of the first spanned `Token` annotation and its end offset set to the end offset of the last spanned `Token` annotation. Figure 5.5 shows how this is defined in UIMA. The main disadvantage of this approach is that there is no direct representation of the named entity annotation as a sequence of token annotations. It is up to the user and their code to infer this from the data model.

In DOCREP, the named entity annotation can be modelled directly as a sequence of token annotations using an `annotation slice` (Section 4.1.6). The DOCREP definition for the named entity type is shown in Figure 5.6. Modelling named entities as a span over tokens (DOCREP) is more intuitive and representative than modelling named
5.2. Modelling OntoNotes in DOCREP and UIMA

entities as a span over character offsets into a document (UIMA). Additionally, in
the DOCREP approach, named entity annotation instances have direct access to their
spanned tokens. To retrieve the spanned tokens in UIMA, the developer needs to query
the index repository in the CAS, providing the begin and end offsets in conjunction
with the token annotation type.

Another difference in modelling capability between DOCREP and UIMA is that
DOCREP is not capable of natively modelling cross-document information due to its
streaming nature (Section 4.1.2). DOCREP documents are treated as completely inde-
pendent from one another, meaning that cross-document pointers are not supported.
The need for this modelling capability stems from the parallel document and parallel
tree annotations provided in OntoNotes 5. In the case of the DOCREP data models, we
chose to represent the parallel document information as metadata on the documents.
Cross-document information is supported by UIMA though its concept of the SOFA
(Section 2.3.3).

5.2.2 Modelling decisions

Here we present a brief description of some of the modelling decisions we made.
Figure 5.7 shows the DOCREP models used for representing the syntax annotation

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

@dr.Ann
class ParseNode extends AbstractAnn {
    @dr.Field public String tag;
    @dr.Field public String pos;
    @dr.Pointer public Token token;
    @dr.Field public String phraseType;
    @dr.Field public String functionTags;
    @dr.Field public int corefSection;
    @dr.SelfPointer public List<ParseNode> children;
}
```

Figure 5.7: DOCREP models for representing the syntax annotation layer in OntoNotes 5.
Chapter 5. Evaluating DOCREP on OntoNotes

```java
@dr.Ann
public class Speaker extends AbstractAnn {
  @dr.Field public String name;
  @dr.Field public String gender;
  @dr.Field public String competence;
}
```

Figure 5.8: DOCREP models for representing a speaker entity in OntoNotes 5.

layer of OntoNotes. One modelling decision here which deviates from the OntoNotes database schema is that we decided to place POS tags on leaf parse nodes instead of on the tokens. If we run another parser and POS tagger over the data, we would like to store both the non-gold parse nodes and POS tags. By placing the POS tags on the parse nodes, the non-gold data is grouped together, allowing whole annotation stores to be ignored or discarded (Sections 3.2.3 and 4.4.1). Additionally, if we wanted to add output from a second parser and POS tagger, with our data model presented here, this is achieved simply through the addition of another ParseNode annotation store on the document. If the POS tags were instead stored directly on the Token, there would need to be a field for each POS tag source; e.g. gold_pos, candc_pos, mxpost_pos, etc. If it was convenient within an application to have access to the POS tags on the token annotations, they can be easily projected down from the parse nodes through a decorator (Section 4.4.3).

Figure 5.8 shows an example of an annotation that does not correspond to anything directly on the document itself, but instead represents a real-world object. Note that this data model has neither an annotation slice nor byte slice as fields. The Speaker class models the real-world entity that is the person who spoke a segment of text. Both DOCREP and UIMA are well equipped to handle this kind of annotation. In UIMA, the supertype for the annotation becomes uima.cas.AnnotationBase instead of uima.cas.Annotation, meaning that the Speaker type does not inherit begin and end attributes. In DOCREP, nothing special needs to be done, as shown in the example.

Figure 5.9 shows how the coreference annotation layer is modelled. There is nothing particularly of note in this example apart from demonstrating how we chose to model
coreference as canonical entity collecting all mentions of that entity. At runtime, the
user will most likely want to apply a “reverse pointer” decorator (Section 4.4.3) to
CorefChain objects so that the mentions know which chain they belong to. Like
the Speaker annotation type, the CorefChain type represents an entity rather than
something directly on the document.

The last data model we will explicitly discuss here is the Sentence, shown in
Figure 5.10. For the subcorpora in OntoNotes 5 which come from transcriptions of
spoken text, speaker information as well as start and end time offsets are provided.
This is the first example shown here of mixed-media annotations. In this case, the start
and end times refer to the offset, in seconds, into a media track which is not provided as
part of the OntoNotes corpus. An example of the raw speaker information provided by
OntoNotes in its flat file format is given in Figure 5.11. The file format is one sentence
per line, with start and time offset values, followed by the name of the speaker and their attributes, in a fully denormalised manner. The DRF representation is much cleaner than the flat file representation, with speakers and their attributes modelled as entities and their relationship to sentences modelled in a normalised fashion. Additionally, there is no arbitrary line-aligned file parsing and cross-referencing required to project the speaker information back to other annotation layers.

Other than the speaker information, our Sentence model also provides the more obvious linguistic information such as the tokens over which it spans, as well as a pointer to the root of the parse tree. The propositions which exist in this sentence are also pointed to from here, though they are not discussed here.

5.3 Evaluation via corpus representation

Thus far, we have provide design arguments for why corpora should be represented in a DRF such as DOCREP. Some of these reasons include richer linguistic modelling, metadata access, and the normalisation of entity information. In this section, we provide some further evidence for supporting the claims that DRFs are a better choice for corpus representation and that DOCREP is a superior DRF to UIMA for use in computational linguistics.

With our data models for the OntoNotes corpus created, we now convert the corpus from its database representation into both a DOCREP and UIMA representation. We
then validate that our conversion process was lossless by converting both DRF versions of the corpus into their flat file counterparts and compare the flat files against the original OntoNotes 5 flat files.

5.3.1 Corpus conversion

We have claimed that DOCREP is easy to use, and provides fast idiomatic APIs in multiple programming languages. In these experiments, we use the UIMA and DOCREP APIs to convert the annotation layers in OntoNotes 5 corpus into annotations in a DRF. We compare the runtime resources required to perform this conversion, reporting the amount of space needed to store the corpus in its original and DRF formats, as well as the time taken to perform this conversion. We will later discuss how each of the DRF APIs were to use from the perspective of a software developer working with a linguistic corpus.

These experiments use the database version of the OntoNotes corpus. To perform these experiments, we first load all of the data from the database into local memory for the document we are currently processing. This allows us to exclude database latency from our reported times. This loaded data is stored in an in-memory object structure which does not know about document representation frameworks, nor the DOCREP or UIMA data models we defined earlier. The in-memory object representation is then converted into the appropriate UIMA and DOCREP annotation objects, recording how long this conversion process took to execute. The UIMA and DOCREP versions of the documents are then serialised to disk, recording how long the serialisation took and the 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. Runtimes were averaged across multiple runs of the conversion process.
Table 5.2: 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.

<table>
<thead>
<tr>
<th>DRF</th>
<th>API</th>
<th>Output Format</th>
<th>Conversion Time</th>
<th>Serialisation Time</th>
<th>Serialisation Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Java</td>
<td>XMI</td>
<td>25</td>
<td>131</td>
<td>1894</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>XCAS</td>
<td>25</td>
<td>122</td>
<td>3252</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>binary</td>
<td>25</td>
<td>2103</td>
<td>1257</td>
</tr>
<tr>
<td>UIMA</td>
<td>Java</td>
<td>cbinary</td>
<td>25</td>
<td>76</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>XMI</td>
<td>77</td>
<td>630</td>
<td>2141</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>XCAS</td>
<td>77</td>
<td>611</td>
<td>3252</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>binary</td>
<td>77</td>
<td>695</td>
<td>2135</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>—</td>
<td>12</td>
<td>61</td>
<td>371</td>
</tr>
<tr>
<td>DOCREP</td>
<td>C++</td>
<td>—</td>
<td>12</td>
<td>23</td>
<td>371</td>
</tr>
<tr>
<td></td>
<td>Python</td>
<td>—</td>
<td>27</td>
<td>32</td>
<td>371</td>
</tr>
</tbody>
</table>

5.3.2 Empirical evaluation

In order to provide a fair comparison between UIMA and DOCREP, we perform the conversion using both the Java and C++ UIMA APIs,¹ as well as using all three DOCREP APIs (Java, C++, and Python). The kinds of evaluations performed here are commonplace in areas of computer science, including databases, where benchmarks are used to compare systems and evaluate efficiency. The code to load the data from the database and construct the in-memory object structure was common between both the UIMA and DOCREP conversion processes. For UIMA, we serialise in all available output formats: both the XMI and XCAS XML formats, the binary format, and the compressed binary (cbinary) format. The UIMA C++ API does not support the compressed binary output.

The results of the corpus conversion process are shown in Table 5.2. The fourth column shows the accumulated time spent to convert all of the documents from their in-memory representation into the appropriate UIMA and DOCREP annotation objects.

¹We used the latest releases at the time of writing: 2.6.0 for Java and 2.4.0 for C++.
5.3. Evaluation via corpus representation

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++. It is unusual that the C++ UIMA API takes over three times as long as the Java API to perform this conversion. This is further evidence that the C++ API is a second-class citizen in the UIMA world, not undergoing the same level of performance tuning as the Java API.

The fifth column in this table shows the accumulated time taken to serialise 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++. We are unsure why the UIMA binary output for the Java API is an order of magnitude slower than the other output formats for the Java API. It is unclear why the UIMA compressed binary format serialises faster than the binary format given that it is (presumably) performing compression during serialisation. Given the difference in serialisation size between the compressed and uncompressed formats, the time difference might all be in I/O.

The last column in Table 5.2 shows the serialisation size on disk. Apart from the UIMA compressed binary output format, the DOCREP serialisation requires up to 9 times less space than any of the UIMA output formats. It is unsurprising that the UIMA compressed binary format serialises smaller than the corresponding DOCREP serialisation as it is compressed. It is unclear why the serialisation sizes for the UIMA XMI and binary formats do not align between the Java and C++ APIs. In both cases, the serialisation produced by the C++ API is significantly larger that its Java counterpart; up to 70% larger in the case of the binary output format.

Another interesting empirical comparison is how well each of these serialisation formats compress. Table 5.3 shows how well each of the serialisations produced by the OntoNotes 5 corpus conversion compress using two standard compression utilities: gzip (DEFLATE) and xz (LZMA). Each compression utility was run with their default settings. For the UIMA serialisations, we used the serialisations produced by the Java API rather than the C++ API. The serialisation files produced by UIMA as well as the original OntoNotes “flat file” files were first placed into an archive (a tar archive) so
Chapter 5. Evaluating DOCREP on OntoNotes

Table 5.3: A comparison of the how well each of the annotation serialisation formats compress using standard compression libraries. All sizes are reported in MB.

<table>
<thead>
<tr>
<th>Format</th>
<th>Original</th>
<th>gzip</th>
<th>xz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat files</td>
<td>375</td>
<td>52</td>
<td>30</td>
</tr>
<tr>
<td>SQL</td>
<td>4560</td>
<td>646</td>
<td>262</td>
</tr>
<tr>
<td>MySQL − indices</td>
<td>4303</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MySQL + indices</td>
<td>5812</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>UIMA XMI</td>
<td>1894</td>
<td>268</td>
<td>144</td>
</tr>
<tr>
<td>UIMA XCAS</td>
<td>3252</td>
<td>330</td>
<td>185</td>
</tr>
<tr>
<td>UIMA binary</td>
<td>1257</td>
<td>375</td>
<td>150</td>
</tr>
<tr>
<td>UIMA cbinary</td>
<td>99</td>
<td>66</td>
<td>65</td>
</tr>
<tr>
<td>DOCREP</td>
<td>371</td>
<td>115</td>
<td>69</td>
</tr>
</tbody>
</table>

that the compression algorithms could be run over the whole corpus in one go instead of running on each document in isolation, allowing for better overall compression.

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 made read-only after the initial import, and was not altered in any way.

Unsurprisingly, the DOCREP binary serialisation format does not compress as well as text serialisation formats with lots of repetition, such as XML or the original stand-off 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.

5.3.3 Quality assurance

In order to ensure that our conversion of the corpus into our DOCREP and UIMA representations was not lossy, we additionally performed a verification step. Our verification procedure was to try and reproduce the flat file version of the OntoNotes
5.3. Evaluation via corpus representation


(a) The coreference annotations in the flat file.

(b) The coreference annotations in the database.

Figure 5.12: An example of annotation discrepancies between the OntoNotes 5 flat file and database distributions. The coreference annotations for sentence 12 of nw-wsj/13-wsj_1312 do not match.

corpus from our DRF versions. Since the database version of the corpus and the flat file version of the corpus supposedly contain the same data, we should be able to fully recreate the flat files.

This verification process yielded many surprising results, all of which only further strengthen our argument that DRFs are a better corpus distribution format than both flat files and databases. While trying to recreate the flat files, we found that we could not do so for many of the documents. After much debugging of our DRF conversion process, we found that the error was not in fact with our conversion process, but with many discrepancies between what is modelled in the OntoNotes 5 database and what is represented in the corresponding flat files. An example discrepancy is the coreference annotations for sentence 12 of document nw-wsj/13-wsj_1312, shown in Figure 5.12. None of the numerous coreference annotations which exist in the flat file for this sentence exist in the database version.

Another surprising find was both broken annotations and broken file formats. Figure 5.13 shows a snippet from the named entity file for document tc/ch/00/ch_0002.

The first issue here is that the named entity annotations make no sense, marking seem-
**Chapter 5. Evaluating DOCREP on OntoNotes**

Figure 5.13: An example of both broken annotations and a broken file format in the gold standard data. This snippet comes from the named entity annotations for document tc/ch/00/ch_0002@0002@ch@tc@en@on.

ibly random tokens as entities, and also crossing sentence boundaries. Additionally, this file violates the assertion that the named entity annotations are non-nested, with the last line in this example containing a nested annotation.

It is unclear how these issues came about. Since the 5th release of OntoNotes is the first to contain the database version, we presume that database was constructed from the flat files rather than the other way around. There are many potential reasons for how and why the data discrepancies exist between the flat files and the database. One can hypothesise that this was possibly the result of a corpus parsing bug or a bug in the conversion process. If the annotations were stored in a DRF in the first place, these kinds of bugs cannot happen as easily as there is no arbitrary file formats to parse, and because the relationship between annotations and entities is already natively modelled.

Once it was clear that there existed discrepancies between the database and the flat file representations, we constructed a blacklist of documents\(^2\) to ignore during our conversion verification procedure. Ignoring the documents we manually identified as having discrepancies, our DRF representations were able to fully reconstruct the flat file versions of the OntoNotes 5 corpus. This successful conversion process from database to DRF to flat files indicates that DRFs are capable of sufficiently modelling the needs of this complex corpus while providing normalised and native annotation and entity relationships.

\(^2\)There are on the order of 20 documents in this blacklist.
5.4 Evaluation against our design requirements

In this chapter, we have shown that DOCREP is capable of representing the OntoNotes 5 corpus: a large multilingual corpus containing multiple annotation layers per document. By doing so, we have shown that DOCREP has satisfied our design requirements outlined in Chapter 3:

**Simplicity and expressive adequacy** We have shown that DOCREP able to model the many kinds of annotations which exist within the OntoNotes 5 corpus. In doing so, we have demonstrated its expressive adequacy. This chapter also provided our DOCREP data models and compared them to their equivalent models in UIMA (Section 5.2), showing DOCREP’s simplicity.

**Lightweight, fast, and resource efficient** We have shown that DOCREP is lightweight, fast, and resource efficient through a comparison of converting the OntoNotes 5 corpus to UIMA and DOCREP (Section 5.3). DOCREP serialises up to 34 times faster and serialises up to 9 times smaller than UIMA for this corpus. We also compared these serialisations to the OntoNotes database, showing DOCREP requires less than \( \frac{1}{15} \)th of the storage requirements.

5.5 Summary

In this chapter, we have shown that DRFs such as DOCREP and UIMA a more suitable and useful format for linguistic corpora than flat files or relational databases by providing native support for normalised annotation and entity relationship modelling as well as being easier to use as a developer. Annotation alignment and file format parsing are error prone and time consuming tasks. Bypassing the need for these operations while additionally providing runtime models for accessing and manipulating the annotations makes DRF representations the clear winner.
In the previous chapter we outlined how DOCREP was implemented and what functionality it provides as a new streaming DRF. This chapter has evaluated the ability for DOCREP to act as a DRF for linguistic annotation by successfully representing all of the annotation layers in the linguistically-rich OntoNotes 5 corpus, while also outperforming the existing standard DRF, UIMA. The next chapter goes on to evaluate DOCREP as a developer, assessing it from a usability perspective and how well it adapts itself to various CL research scenarios and LT applications. The next chapter also discusses how users interact with DOCREP streams and how a rich set of UNIX-inspired tools provide a superior experience to working with flat file versions of linguistic corpora.
6 Evaluating DOCREP as a user

In Chapter 4 we outlined how DOCREP was implemented and what functionality it provides as a new streaming DRF. In Chapter 5, we then went on to evaluate DOCREP on its ability to act as a DRF for linguistic annotation by converting the linguistically-rich OntoNotes 5 corpus into DOCREP. We showed that using a DRF for corpus representation was superior to flat file and relational database representations for a multitude of reasons. We also showed that DOCREP outperformed the most commonly used DRF, UIMA, at representing this corpus. This chapter continues the evaluation process of DOCREP, evaluating it from the perspective of a user.

When presented with a new library or framework, three questions often pop into the minds of developers: “How do I install it?”, “How do I work with it?”, and “What do others say about it?” As a developer, all of these questions are important as they will be the one interacting with the library. If it is hard to install or hard to jump in and use straight away, that presents an immediate barrier to uptake. If what the library produces is hard to work with, or its APIs are insufficient to perform the desired tasks, the library is not useful. If other developers are not advocating the use of the library, why not? In this chapter we aim to address these three questions.

In this chapter, we first describe the “getting started” experience when using DOCREP versus using GATE and UIMA. We then answer the question about working with DOCREP by describing our rich set of command-line tools for interacting with and manipulating with DOCREP streams. Last, we provide testimonials from DOCREP users to answer the “what do others say about it?” question.
Like the DOCREP decorators presented in Section 4.4.3, work presented in this chapter was done in collaboration with Joel Nothman. The user workflows and idioms developed in this collaboration appear throughout the DOCREP command-line tools. Joel was a driving force behind the initial prototyping and brainstorming of these patterns, however he was not involved as deeply in the final development, other than to provide feedback. Our back and forth discussions have helped make the DOCREP command-line tools what they are today.

6.1 Starting out using a DRF

This section aims to answer the question “How to I start using a DRF?”. That being said, the question a user is more likely to be asking is “How do I start using <insert DRF name here>?”. We approach this question from the perspective of a Java developer, a C++ developer, and a Python developer, outlining the steps required to get started with GATE, UIMA, and DOCREP. During this, we draw upon our own experiences from the OntoNotes 5 conversion (Chapter 5) in each of these languages. We acknowledge that we cannot entirely compare the installation process as a new user given our intimate knowledge of DOCREP.

As a Java developer

GATE is a widely-used Java-only DRF and general library for text engineering. As a user starting out with GATE, the download and installation process is quite straightforward. The user navigates to the GATE website\(^1\), locates the download link, and downloads the latest version. At the time of writing, the latest version is the 8.0 release, which comes as a large 450 MB download. This release requires Java 1.7 or greater.

Buried quite far down in the GATE manual are the instructions for using GATE as a DRF only, ignoring all of the GUI components and pre-trained models. The GATE core

\(^{1}\)https://gate.ac.uk/
library is hosted on Maven central, meaning it is accessible via standard Java package management tools.

According to the UIMA website, the recommended installation path for a new UIMA user is to download and install the Eclipse IDE and then install the UIMA Eclipse integrations. To start a new UIMA project, users are then advised to use the UIMA Eclipse new project creator wizard, followed by the Eclipse XML type creator wizard to create and define the appropriate XML definition for a UIMA annotation type. Eclipse then runs the jcasgen program to convert this XML type definition into the appropriate Java source files, and copies them into the Eclipse workspace. While this workflow will work for some developers, others will want to use their own development environments or work from the command-line rather than from an IDE such as Eclipse. Working with the UIMA library outside of Eclipse is not easy, with most of the documentation explaining how to perform required operations, such as defining a new type, in the context of using Eclipse only.

Apart from the Eclipse-based installation, other installation options presented on the UIMA website include source and binary JAR files (20 MB), as well as details on how to reference the library hosted on Maven central for use in standard Java package management tools. The latest release requires Java 1.7 or greater.

The DOCREP Java library is also on Maven central, making it easily installable via any of the standard Java package management tools. The library is also available as a JAR (49 kB) from both Maven central and GitHub, and requires Java 1.6 or greater. The DOCREP library is simply a set of APIs — there is no GUI or IDE integration. As such, once the library is installed, from a developers perspective, they can import the appropriate components (org.schwa.dr.*) and start using DOCREP straight away, irrespective of chosen IDE or development workflow.

\(^2\text{http://uima.apache.org/}\)
As a C++ developer

There is no C or C++ implementation of GATE, nor is there any easy way to use the Java implementation from within C++. Java Native Interface (JNI) is one option, which provides an interface for native applications to interact with code running in the JVM, but this option is brittle at best. GATE is not really an option as a DRF in C++.

While the UIMA Java API has a new major releases every 6 months or so, the UIMA C++ API has not been updated since 2012. It is very lacking in both documentation and examples compared to the Java API, requiring lots of source code reading in order to work out how to use the APIs, as well as to discover what APIs actually exist. In addition to its stagnated development, the UIMA C++ API depends on four very large existing C/C++ frameworks which the user needs to download, configure, build, and install even before they can start building the C++ API. The four dependencies are the Apache Portable Runtime (APR), International Components for Unicode (ICU) for Unicode support, Apache Xerces for XML interpretation and manipulation, and Java Native Interface (JNI) for the integration between UIMA analysis engines (AEs) implemented in C++ and a UIMA Java pipeline.

Without an update in three years, these dependencies have progressed and the latest version of many of them do not compile with the UIMA C++ API, requiring the user to perform the tedious task of going back one release of the dependant library, repeat the configure-make-install cycle, and test whether the UIMA C++ API the compiles, repeating if it does not. Getting the UIMA C++ API to compile took us many hours by itself, and we are experienced C++ developers. This was an extremely negative user experience and if we were not doing this for the sake of experimentation, we would have rejected using the UIMA C++ API long before we got it compiled.

As mentioned earlier (Section 4.3.3), the DOCREP C++ library is configured and built using the standard GNU Autotools library, meaning installation is the configure-make-install cycle UNIX users are familiar with:
6.1. Starting out using a DRF

If the user is running Mac OS X, they can install DOCREP via Homebrew:

```
$ brew install libschwa
```

The source code releases also provide scripts to generate Debian and RedHat compatible package bundles, facilitating the installation across multiple machines in a network. Both installation paths install the DOCREP API and the command-line tools. The library has no dependant packages, meaning the user does not have to install anything else ahead of time apart from the standard build tools.

As a Python developer

There is no Python wrapper for GATE, nor is there SWIG\(^3\) bindings since there is no C or C++ implementation of GATE. The only real option is to use Jython\(^4\) and interact with the GATE Java library this way. This, of course, restricts the user to Jython, disallowing any Python packages which depend on the more traditional CPython runtime. As a Python developer, this is not viewed a very attractive option and they would probably look for an alternate solution.

For UIMA, there are SWIG bindings for Python distributed with the UIMA C++ implementation. However, this requires building the UIMA C++ API and reading the source code for documentation. It is possible to use this, but it is not a very **Pythonic** library to work with, and the developer would probably seek an alternate solution. One such solution would be to treat UIMA as a service and interact with it via message passing rather than interacting with it at an API level. Newer releases of UIMA support interaction via the ActiveMQ\(^5\) message passing service. This solution has the additional infrastructure cost of maintaining a message passing service server as well as incurring runtime latency associated with non-local execution.

\(^3\)http://www.swig.org/
\(^4\)A Python implementation that runs on the JVM: http://www.jython.org/
\(^5\)http://activemq.apache.org/
The DOCREP Python library is hosted on PyPI, making it easy to install via the standard Python package management tool pip:

```bash
$ pip install libschwa-python
```

Once installed, the user can start using DOCREP straight away in whatever development environment they desire. The DOCREP Python API supports Python 2.7, the final release of the 2.x series, as well as versions 3.3 and up, making it usable in most actively-maintained Python codebases.

### 6.2 Working with DOCREP streams

Annotated text processing frameworks such as GATE and UIMA 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. In this section, we discuss our rich suite of command-line utilities, summarised in Table 6.1, designed for the same purpose for DOCREP.

This section primarily aims to answer the question “How do I work with DOCREP streams?” DOCREP represents annotation layers in a binary, streaming format, such that process pipelining can be achieved through UNIX pipes. Our command-line utilities have been developed organically as needed over the past three years, and are akin to UNIX tools (e.g. grep, head, wc, etc.) which instead operate over plain text formats. Some of our utilities are comparable to utilities provided by UIMA and GATE, while others are novel. A number of our utilities exploit the evaluation of user-supplied Python code over each document, providing great expressiveness while avoiding engineering overhead when exploring data or prototyping.
6.2. Working with DOCREP streams

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.

6.2.1 Already-familiar utilities

One of our original design goals with DOCREP was to make it as easy to use as possible (Section 3.1.3). This philosophy should be consistent throughout all aspects of the framework: easy to install, easy to learn, and easy to work with. In an ideal situation, the user would already know how to use our utilities for working with DOCREP streams, but of course this is not possible. We can however, get close.

For users familiar with UNIX tools to process text streams, such as linguistic corpora distributed in flat file formats, implementing similar tools for working with structured DOCREP files seems natural. For each common operation that needs to be performed on a DOCREP stream, we implemented a utility which looks, feels, acts, and is named similarly to its UNIX counterpart. This way, even if the user has never used or seen DOCREP before, they should be able to make an educated guess at how to perform the desired operation based on how the same task would be achieved with plain text streams.

Table 6.1 summarises our rich set of supplied command-line utilities for working with DOCREP streams. We describe a number of the tools according to their application in the next section. Here we first outline common features of their design.
Table 6.1: The rich set of command-line utilities we provide for working with DOCREP streams and their comparable UNIX tools, including required input and output types.

**Invocation and API**

Command-line invocation of utilities is managed by a dispatcher program, `dr`. The behaviour of this dispatcher mirrors that of the `git` versioning tool dispatcher. An invocation of `dr cmd` delegates to an executable named `dr-cmd` located 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. Note that DOCREP processing is not limited in these languages: any language with a DOCREP library or even just a MessagePack library is capable of processing DOCREP streams. This dispatching approach allows the user to have their own rich set of command-line accessible DOCREP processors specific to their needs. For example, for a user working in parsing, a DOCREP equivalent of `tgrep2` (Rohde, 2005) for `grep`’ing tree structures might be useful.

**Streaming I/O**

As shown in Table 6.1, most of our DOCREP utility commands take a single stream of documents as input (defaulting to `stdin`), and will generally output either plain text
6.2. Working with **DOCREP streams**

or another DOCREP stream (defaulting to stdout). This parallels the UNIX philosophy 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.

As we saw earlier (Section 5.1), the OntoNotes corpus is distributed in multiple annotation layers per document as a series of flat files, one file per annotation layer per document. The primary reason for this method of distribution is that there is no simple way to combine multiple, potentially overlapping, annotation layers into a single text-based representation. This combining annotation layers problem can be solved through the use of a DRF, and as such, when using a DRF, this separate-files requirement no longer exists.

**Self-description for inspection**

Generic command-line tools require access to the schema as well as the data of an annotated corpus. DOCREP documents are self-describing (Section 3.3.5). On the wire, documents include a description of the data schema along with the data itself. The self-describing nature of DOCREP documents makes such tools possible with minimal user input as additional parameters or configuration options do not need to be specified to instruct the tool how to interpret the data stream. Thus by reading from a stream, fields and annotation layers can be referenced by name, and pointers across annotation layers can be dereferenced.

Extensions to this basic schema may also be useful. For a number of our tools, a Python class (on disk) can be referenced that provides *decorations* over the document (Section 4.4.3): in-memory fields that are not transmitted on the stream, but are derived at runtime from the modelled data. For example, a document with pointers from dependent to governor may be decorated with a list of dependents on each governor. For many purposes, the self-description is sufficient, but it is convenient to have this easily-accessible additional power at hand when required.
In contrast, UIMA’s type system annotations are not self-describing and instead require the XML type definition files in order to be able to interpret the annotation data. The uimaFIT library (Ogren and Bethard, 2009) goes some way to provide this kind of runtime flexibility by hiding a lot of the details of working with the raw UIMA components, and by interacting with the XML files that the user would otherwise be having to interact with.

**Custom expression evaluation**

Our core utilities are implemented in C++ for efficiency, while others are able to exploit the expressiveness of an interpreted programming language; specifically Python. Such tools include `dr format`, `dr split`, and `dr sort`. The ability to execute arbitrary supplied Python code on each document and act upon the result provides the underlying tool with great flexibility and power. The input to the supplied Python code is an object representing the current document, and index of that document in the stream (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.

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 `index` (offset index) are injected as well as built-in names like `len`. 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.

### 6.2.2 Working with DOCREP streams on the command-line

There are many tasks that users of annotated corpora need to perform on a corpus, especially if the corpus is being created or mutated in some way. For example, during manual corpus annotation, the supervisors need to easily see what has changed in
the corpus, mutate the annotations appropriately, or even merge in annotations from
different sources. In the context of NLP pipelining, developers need to be able to inspect
documents which have raised errors, in addition to normal inspection of documents
throughout the various stages of the pipeline for debugging and quality assurance
purposes. In a machine learning or stochastic process, it is often useful to trace through
why machine-generated annotation decisions were made by tracking probabilities and
other metadata throughout a pipeline.

All of these use cases require interaction with the corpus. In a binary format
like DOCREP, it is not obvious how a user does this. Having described their shared
structure in the previous section, this section presents examples of the utilities available
for working with streams of DOCREP documents. We consider three broad application
areas: corpus management, quality assurance, and more specialised operations.

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. It might be used
to extract particular documents or to remove problematic documents. The user provides
an expression that evaluates to a Boolean value — when true, the 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, that like its namesake, it performs a linear search, while a non-
streaming data structure could provide fast indexed access. During its execution over
a DOCREP stream, each traversed document is at least partially deserialised, adding
to the computational overhead. dr grep is often piped into dr count to print the
number of documents (or sub-document annotations) in a subcorpus.
\texttt{dr split} moves beyond such binary filtering. Like UNIX’s \texttt{split}, it partitions a file into multiple files, using a templated filename. In \texttt{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 \texttt{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, \texttt{dr split k 10} will round-robin assigning documents to files named \texttt{fold000.dr} through \texttt{fold009.dr}, which is useful to derive a $k$-fold cross-validation strategy for machine learning. In order to stratify a particular field across the partitions for cross-validation, it is sufficient to first sort the corpus by that field, passing the result to \texttt{dr split k}. This is one motivation for \texttt{dr sort}, which may similarly accept a Python expression as the sort key; e.g. \texttt{dr sort py doc.doc_id}. As a special case, \texttt{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 \texttt{dr split} is to concatenate multiple streams. Given DOCREP’s streaming design, UNIX’s \texttt{cat} suffices. For other corpus representations, a specialised tool may be necessary to merge corpora. For example, such simple combination is not possible when using an SGML-based corpus representation.

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 \texttt{cut} and \texttt{paste}), or renaming fields would thus be useful operations, but are not currently available as a DOCREP command-line utility.\footnote{Deleting and renaming are needed less than in other frameworks due to DOCREP’s lazy serialisation and runtime-configurable schema renaming.} Related tasks may be more diagnostic, such as identifying annotation layers that consume undue space on disk; \texttt{dr count --bytes} shows the number of bytes consumed by each store (cf. UNIX’s \texttt{du}), rather than its cardinality.
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 debugging. This is especially true when working with a binary format such as DOCREP, we need tools to assist when things go wrong.

Basic quality assurance may require viewing the raw in a DOCREP stream: the schema, the annotations, or both. This could ensure, for instance, that a user has received the correct version of a corpus or that a particular field was used as expected. Since DOCREP uses a binary wire format, `dr dump` (cf. UNIX’s `hexdump`) provides a decoded text 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. `dr less` (cf. UNIX’s `less`) provides a more user-friendly but still raw view of a DOCREP stream, facilitating the quick inspection of annotations while also providing low-level information to potentially aid with debugging. Figure 6.1 shows some screenshots of `dr less` in action, inspecting the OntoNotes 5 corpus we converted to DOCREP (Section 5.3.1).

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. The number of annotations in each store (total number of tokens, sentences, named entities, etc.) can be printed `-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 when used in conjunction with UNIX’s `tee`, as followings:

```
$ ./my-program < /path/to/input | tee /path/to/output | dr count -t -a -c -e 1000
```

This process outputs cumulative totals (`-c`) over all stores (`-a`) every 1000 documents (`-e 1000`), prefixing each row of output with a timestamp (`-t`).
Figure 6.1: Three example screenshots of dr less being used to inspect documents from the converted OntoNotes 5 corpus.
Problems in pipeline or application deployments can often be identified from only a small sample of documents. `dr head` and `dr tail` (cf. UNIX’s `head` and `tail` respectively) extract a specified number of documents from the beginning or end of a stream, defaulting to 1. Providing a stochastic alternative, `dr sample` employs *reservoir sampling* (Vitter, 1985) to efficiently yield 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 where standard UNIX tools cannot operate. Note that while this is particularly valuable for a binary format such as ours, even simple tasks such as splitting a file on document boundaries may not be a trivial operation with standard UNIX tools even for simple text representations, such as CoNLL shared task formats.

**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:

```bash
$ dr format py \n> "{}\n*{}*::format(doc.id, *{}*::join(t.raw for t 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. POS tags), or fields derived from annotations over tokens (e.g. IOB-encoded named entity recognition tags) can be added as 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 the populate Django project, 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 here is small but very useful in practice since it makes interactive corpus exploration as cheap as possible. This, in turn, lowers the cost of developing complex processors by using interactive exploration to validate a particular technique. The following shows an interactive session in which the user prints the 100 lemmas with highest document frequency on a corpus:

```
$ dr shell /path/to/input.dr
>>> from collections import Counter
>>> df = Counter()
>>> for doc in docs:
...    lemmas = [t.lemma for t in doc.tokens]
...    df.update(lemmas)
...>>> for lemma, count in df.most_common(100):
...    print('{:5d}  {}'.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 in the vein of `sed` or `awk`.

### 6.2.3 Evaluating UNIX tools against the DOCREP command-line

Thus far, we have discussed and attempted to motivate why each of our provided tools exist and what common need they each serve. Here, we give some concrete use cases to solidify these arguments, and demonstrate the power that the UNIX philosophy provides for corpus linguistics when backed by a DRF.

We will compare how a computational linguist performs common operations or queries on the OntoNotes 5 corpus. We will compare working with the OntoNotes...
6.2. Working with DOCREP streams

flat files in conjunction with the traditional UNIX tools to working with the DOCREP stream and the DOCREP command-line tools. Throughout these examples, we assume we are in the top-level directory of the English annotations folder in the OntoNotes 5 LDC distribution.

Each annotation layer in OntoNotes is stored in a separate file, one set of files per document. Each annotation layer has its own file with an extension representing the layer. For example, the files for the first WSJ document are:

```bash
$ ls nw/wsj/00/wsj_0001*
 nw/wsj/00/wsj_0001.name  nw/wsj/00/wsj_0001.onf  nw/wsj/00/wsj_0001.parse
 nw/wsj/00/wsj_0001.prop  nw/wsj/00/wsj_0001.sense
```

We will assume that the DOCREP-converted OntoNotes 5 corpus is in the current directory in a file called ontonotes5.dr.

In all of the following examples, as far as we are aware, there is no real equivalent concept for GATE or UIMA for this rich command line-oriented corpus querying. To answer most of the following questions, GATE and UIMA users would need to write code specific to the question asked rather than being able to composing several task-agnostic tools together as per the UNIX tool philosophy.

**How do we inspect the multiple annotation layers?**

With the plain text corpus, annotations can be viewed using `less`. Each annotation layer must be viewed separately as they are located in different files. For example:

```bash
$ less nw/wsj/00/wsj_0001.name
$ less nw/wsj/00/wsj_0001.parse
...
```

In DOCREP, since all of the layers are stored together, a single invocation of the equivalent command `dr less` allows the user to see all of the annotation layers:

```bash
$ dr less ontonotes5.dr
```

If instead of DOCREP, the user was using GATE as their DRF, the GUI provided by GATE provides a rich set of visualisation components to facilitate the graphical visualisation of multiple annotation layers. On a similar note, an advantage of the plain text versions here is that they allow the user to easily see the annotation projected
onto the document text, since the annotations are inline. The stand-off nature of DRF annotations does not facilitate this without some richer visualisation to project the stand-off annotations back onto the document text.

### How many instances of each annotation layer are there in document X?

With the plain text corpus, since each annotation layer is in its own file and uses its own file format, a separate command will need to be developed and executed for each annotation layer. For example, for the first WSJ document, we can get the number of tokens and named entity spans by:

```bash
$ egrep -o '([^()]+ |[^()]+)' nw/wsj/00/wsj_0001.parse | wc -l
31
$ egrep -o '<[A-Z]+ TYPE="[^"]*">[^<]+</[A-Z]+>' nw/wsj/00/wsj_0001.name | wc -l
6
...```

This is an error prone and fiddly task, especially when dealing with SGML or XML files which, by strict definition, cannot in general be processed using regular expressions in this manner.

**DOCREP** stores all of the annotation layers together along with metadata about their cardinality, making this question easy to answer:

```bash
$ dr grep 'doc.doc_id ~ /00001@wsj0nv@en@/' ontonotes5.dr | dr count -a
ndocs ... named_entities parse_nodes ... sentences speakers tokens
1 ... 6 53 ... 2 0 31
```

Like its plain text counterpart, the DOCREP pipeline involves a filter operation followed by a count, except the filter in this case is to extract the requested document from the corpus rather than extracting specific instances of the annotations.

### Combine the named entity annotations for all documents together to provide to a third-party system for training a model.

Imagine the user is training a third-party NER system which takes as input a single file containing all of the named entity annotations to train on. Working with the plain text version of the corpus, combining all of the flat files together is a non-trivial operation due to the fact they are in SGML/XML markup. SGML requires there be only one root
node, so simply concatenating the separate files together into one will not result in a valid SGML document. The user would need to write a specific script to perform this merging operating.

In the DOCREP case, the question is a bit difficult to answer. If this external tool knows how to read DOCREP, the answer is of course trivial — simply feed the appropriate documents into the system:

```
$ dr grep 'doc.lang == "en"' ontonotes5.dr | the-external-system
```

If the external system did not know how to read DOCREP, the user would need to write some code to convert the input documents into a format that the system accepted. Alternatively, if the NER system knew how to read a common data format such as the CoNLL, it might be possible to use `dr conll`. Similarly, for a different common data format, the user could implement their own DOCREP tool, `dr otherformat`, to facilitate command-line format conversion.

**How many documents in the English broadcast news section of OntoNotes 5 contain more than 50 named entities?**

With the plain text corpus, this question is trickier to answer than the previous questions due to the need for aggregation. We want to know how many X’s per document, and then order by this count. Answering this with the standard UNIX tools is still quite achievable, but it does require some more forethought. Many more steps in the pipeline are needed, totalling 7 different UNIX tools:

```
$ find bn -name '*.name' -exec grep -H -o '<ENAMEX' {} ; \
> | cut -d : -f 1 | sort | uniq -c | awk '$1 > 50' | wc -l
```

```
76
```

With the DOCREP corpus, the `dr grep` utility provides some simple metadata-related functions in its query language. One such function retrieves the cardinality of a store, allowing this question to be answered quite simply:

```
$ dr grep 'doc.doc_id ~/@bn@en@/ && len(doc.named_entities) > 50' ontonotes5.dr \ 
> | dr count
```

```
76
```
What document contains the most named entities?

Working with the plain text corpus, this is very similar to the previous question. The cardinality constraint is removed from the previous pipeline and instead, the number of annotation instances is sorted and the most frequent is extracted:

```
$ find . -name '*.name' -exec grep -H -o '<ENAMEX' \; 
| cut -d : -f 1 | sort | uniq -c | sort -rn | head -n 1
```

The DOCREP case is more interesting as this is the first case where it is useful to combine both DOCREP tools and traditional UNIX tools to achieve the end goal:

```
$ dr grep 'doc.lang == "en"' ontonotes5.dr \ 
| dr count -s named_entities -e -d doc.doc_id -H -F -N \ 
| sort -rnk2 | head -n 1
```

The additional -H -F -N flags to `dr count` disable aspects of the default output formatting so that `sort` can operate correctly.

Split the corpus into 10 folds for cross-validation.

When working with the plain text version of the corpus, how splitting is achieved is very format dependant. The most likely scenario is that a custom script is needed to perform the partitioning, especially if multiple annotation layers are needed in each of the folds.

In the case of the DOCREP corpus, there are multiple options available on the command-line, depending on how the user wants the corpus partitioned. The simplest option is to partition via round-robin:

```
$ dr split k 10 ontonotes5.dr
```

With the addition of `dr sort`, the user could randomly partition the documents instead, round-robining over the randomly ordered documents:

```
$ dr sort random ontonotes5.dr | dr split k 10
```
6.2.4 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.

A substantial area for future work within DOCREP is to provide more graphical tools, such as those GATE provides out of the box, 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.

6.3 Testimonials

This section aims to answer the question “What do others say about it?”. As with any new library or framework, getting the word out about why people should use it is difficult. Our research lab has used DOCREP extensively for the past three years. Additionally, all external and industry projects our lab and members from our lab have been involved with have used DOCREP as the document annotation data store. After the initial DOCREP publication in Dawborn and Curran (2014), additional users outside of our lab starting using it and were as excited about it as we were.

Here, we quote feedback from NLP researchers and LT application developers from inside and outside of our lab who have been using DOCREP over the past three years...
for a variety of NLP tasks. All testimonies collected are included below. All users sampled are commercial NLP application developers or NLP research students and academics. We provide these real-world examples of DOCREP’s use to demonstrate that DOCREP is a valuable tool for researchers, and how it assists in rapid research prototyping.

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.


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 forethought than most DOCREP applications so far.

Related publications: Nothman et al. (2012); Nothman (2014).

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); Radford (2015).
6.3. Testimonials

Named Entity Linking. Our NEL system takes advantage of document datasets in a variety of formats. DOCREP greatly simplifies our pipeline by providing a single internal representation for this data. In addition to this, we take advantage of DOCREP’s streaming API to efficiently deserialise and build feature models over different aspects of each document.


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 publications: O’Keefe et al. (2013); O’Keefe (2014).

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.

Related publications: Pink et al. (2014).

Named Entity Linking on Personal Email. We used DOCREP at Composure, an early-stage startup harnessing the benefits of NLP techniques such as NEL to locate relevant emails based on the content of the email being written. With its declarative data modelling syntax, DOCREP allowed us to prototype quickly without
focusing on the specifics of serialisation, a massive boon in a lean startup, where the requirements can change quickly. We appreciated the mature Python API, which allowed us to integrate the document model seamlessly and powerfully with existing web service and machine learning code written in Python.

Dr. Daniel Tse, Composure.

6.4 Evaluation against our design requirements

In this chapter, we aimed to evaluate DOCREP from the perspective of a user, showing that DOCREP as a DRF is easier to get started with and use than GATE and UIMA. We demonstrated how users interact with DOCREP streams, running through a number of common corpus-linguistic use cases and outlining the advantages DOCREP has over the equivalent operations required when working with plain text or GATE/UIMA versions of a corpus. By doing so, we have shown that DOCREP has satisfied our design requirements outlined in Chapter 3:

Low cost of entry, lightweight, fast, and resource efficient Section 6.1 demonstrates that DOCREP has a low cost of entry as a library, especially compared to GATE and UIMA. Chapter 5 demonstrated that DOCREP was lightweight, fast, and resource efficient through a comparative analysis of corpus representations. Additionally, we have provided testimonials from DOCREP users within the CL and LT community demonstrating that they enjoy the simplicity and ease of use of DOCREP.

Development workflow and environment agnostic We have shown that DOCREP is development workflow and environment agnostic (Section 6.1). Users are free to use DOCREP within their IDE and build-workflow of choice.

Searchability, browsability, and separability We have shown that the combination of type-specific heaps (Section 3.3.3), lazy serialisation, and rich set of command-
line tools facilitates the searchability, browsability, and separability of different annotation layers (Section 6.2).

6.5 Summary

In this chapter, we aimed to evaluate DOCREP from the perspective of a user, answering three commonly asked questions by developers being proposed a new library: “How do I install it?”, “How do I work with it?”, and “What do others say about it?”. We have shown that DOCREP as a DRF is easier to get started with than GATE and UIMA, both from an ease of installation perspective and as a lighter-weight library. We went on to demonstrate how users interact with DOCREP streams, running through a number of common corpus-linguistic use cases and outline how these use cases are handled when using plain text corpora as well as when using GATE or UIMA. We then finished this usability evaluation of DOCREP with testimonials from internal and external DOCREP users, providing further evidence, albeit subjective, that DOCREP performs very well on the tasks we aimed to solve.

The next two chapters change direction. Now that we have described and evaluated DOCREP, we now start using it and the document structure information it provides to improve NLP applications and pipelines. We demonstrate this ability through two separate tasks. In Chapter 7, we outline our new document structure-aware tokenization framework and how it makes use of the document structure and offset-maintaining capabilities DOCREP provides to significantly ease the development process of full-stack NLP pipelines. In Chapter 8, we develop a document structure-aware named entity recognition system and show that the addition of document structure-based features achieves state-of-the-art performance on multiple NER datasets.
NLP researchers and application developers increasingly need to map linguistic units, commonly tokens or sentences, back to their location in the original document. For example, the Text Analysis Conference (TAC) Knowledge Base Population (KBP) shared tasks (McNamee et al., 2009) require systems to include offsets back into the original document to justify their answers. If the components in an NLP pipeline are not offset-aware, maintaining these offsets throughout the whole pipeline is difficult without significant engineering overhead. The vast majority of NLP tools operate over vanilla plain text, and so tracking this metadata is non-trivial.

This chapter describes our novel tokenization framework in which the tokens are aware of their byte and Unicode code point offsets in the original document. The tokenization and document structure interpretation are produced natively as DOCREP annotations. Users of this framework’s output are able to map linguistic units back to their location in the original document while simultaneously having access to the document’s internal structure. The combination of DOCREP’s underspecified type system, lazy serialisation, and runtime-configurable schema renaming (Chapters 3 and 4) allows this framework to produce rich document structure annotations without downstream applications suffering from runtime performance degradation nor requiring them to be aware of these annotation layers.

The core contribution of this chapter is the approach to retaining offset information and storing document structure throughout tokenization and sentence boundary detection (SBD) — precursor tasks to almost all NLP pipelines. An implementation of an
efficient, high quality tokenizer is provided as part of this approach. This tokenizer and sentence boundary detector builds upon the power of the DOCREP framework.

If we want pipelines to utilise document structure information, it needs to be available from the start of the pipeline. We perform an initial evaluation on the quality of the tokens and sentence boundaries produced by our framework, but as Read et al. (2012) conclude, evaluating these between systems is surprisingly difficult. The quality of our produced tokens and sentences is not the primary contribution of this chapter; the quality can be improved through further refinement of the rules and heuristics used. The primary contribution is our approach to retaining offset information throughout the transcoding, document structure interpretation, and tokenization and SBD process, allowing downstream tasks access to original document offset information and document structure. This is only possible with a way efficient represent both linguistic information and document structure, which DOCREP provides.

This information is useful for both NLP research and user-facing NLP applications. NLP researchers are able to harness document structure information in their models. For example, encoding document structure features into a NER system such as whether or not the token appears stylised in the original document, or using a different language model for sentences that appear in list items versus paragraphs. In addition to NLP researchers, application developers can make use of offset information. Being able to highlight all tokens which are covered by named entity annotations in a HTML document requires inserting HTML markup around the token span. Discovering the location of the tokens in the original document post hoc can be a difficult and inexact process due to issues such as encoding differences, tokenization normalisation, and interleaving non-token document structure or markup.

Dridan and Oepen (2012) provided a brief summary of the state of tokenization within the NLP community. They show that NLP corpora and systems have slowly been shifting away from the PTB tokenization rules due to recognised limitations and weaknesses in those rules. Read et al. (2012) provide a similar analysis and argument
for sentence boundary detection (SBD). Both papers go on to describe the difficulties in evaluation for both tokenization and SBD due to differences in tokenization guidelines and implementations of those guidelines, as well as differences in the way these systems operate. Some tokenization and SBD systems are rule based, some use supervised machine learning models, and others use unsupervised machine learning (e.g. Kiss and Strunk, 2006).

7.1 Motivation

In order to demonstrate the interaction between the layers of our tokenization framework, we will use a common example throughout this chapter. Figure 7.1 shows the original document we will be working with. This document is a fragment of a larger HyperText Markup Language (HTML) document and is encoded in the Windows-1250 encoding.¹ Both the Euro sign (€; U+20AC) and the ü with umlaut (ü; U+00FC) characters are encoded in 1 byte in Windows-1250 — 0x80 and 0xFC respectively.

Before going on to explain how our tokenization framework operates, we will first provide an example to demonstrate why maintaining document structure and encoding-aware offset information is currently difficult across a whole NLP pipeline.

We would like to perform tokenization and SBD on this document (Figure 7.1) so that it can be passed to further downstream NLP applications. The first step in this process is to ensure the character encoding of the document is something that each NLP component can work with. These days, most applications default to expecting the UCS Transformation Format 8-bit encoding (UTF-8) as input due to its popularity, wide-coverage support across platforms and languages, and ASCII backwards compatibility. Transcoding the original document from Windows-1250 could be performed using an external tool such as iconv² or using the built-in string encoding and decoding facilities in the user’s programming language of choice.

¹https://msdn.microsoft.com/en-US/goglobal/cc305143
²https://www.gnu.org/software/libiconv/
The next step to be able to use this document in a NLP pipeline is to remove the document structure, which in this case is HTML markup. There are a number of different ways this can be achieved. A lightweight solution is to use a package such as Beautiful Soup, as is suggested by the NLTK book (Bird et al., 2009). A more heavyweight, but potentially more accurate, solution is to load the HTML document into a headless web browser such as PhantomJS or HtmlUnit and use the browser’s Document Object Model (DOM) API to extract the document text from the browser’s internal document model.

Once these two steps have been performed, a UTF-8-encoded plain text version of the original document exists which we can feed to downstream applications. Imagine now that a downstream application has identified that the token sequence Down € 6 M is important and should be highlighted in the original document for the user to see. To perform this highlighting, we need to insert HTML tags around this token sequence. The problem now faced is how do we locate the positions of these tokens in the original document? A simple search through the original HTML for these tokens is not sufficient for three reasons. First, in the original document, the € is encoded differently than in UTF-8. Second, there is interleaving HTML document structure between Down and €6M which does not appear in the textual content. Third, if there are multiple matches

---

3 http://www.crummy.com/software/BeautifulSoup/
4 http://www.nltk.org/book/ch03.html#dealing-with-html
5 http://phantomjs.org/
6 http://htmlunit.sourceforge.net/
found in the original document for the searched token sequence, how do you know which match the downstream application has asked you to highlight?

Our novel tokenization framework solves this problem through joint transcoding, document structure interpretation, tokenization, and SBD.

7.2 The tokenization framework

Our tokenization framework is the first to maintain byte and Unicode code point offset information relative to the original (structured) document while also natively producing its tokenization, sentence bounds, and document structure segmentations in a DRF. The tokenization system presented in Dridan and Oepen (2012) maintains offset information, but these offsets are not exportable to a DRF, nor does the system account for document structure. Our tokenization framework is resource and runtime efficient as a result of being implemented in C++ and designed for efficiency. Like the DOCREP APIs, this tokenization framework is open sourced under the MIT licence.

Our framework consists of three layers which feed into one another: input transcoding, document structure and formatting interpretation, and tokenization. Depending on the type of documents being processed, the first two layers may be run in either order, zero or more times. After all of the layers have finished executing, the framework has constructed a model of the document with offset information preserved throughout. This document model is stored in DOCREP.

Our tokenization framework performs Unicode-aware tokenization and SBD simultaneously. The tokenization layer of this framework is a rule-based system, with grammar rules defined in a regular language used for the construction of finite state machines. The transitions on the produced finite state machine execute heuristics to determine whether a sentence boundary has been encountered. Additionally, these SBD heuristics can be informed by the document structure interpretation layer of our framework. This interaction between SBD and document structure interpretation has
Figure 7.2: The original byte input stream from Figure 7.1 is transcoded into UTF-8. The second channel on the UTF-8 stream maintains byte consumption counts relative to the original input stream.

been touched upon in the literature (Liu, 2005; Liu and Curran, 2006) and aligns with the observations made by Read et al. (2012) that document structure information looks to improve SBD performance.

7.2.1 Input transcoding

Having covered the pipeline-level problems that our tokenization framework is aiming to address, we will now cover each of the three layers of the framework. The first layer is input transcoding.

The original document (Figure 7.1), encoded in Windows-1250, is first transcoded into UTF-8, keeping track of the byte offsets in the process. This process is illustrated in Figure 7.2. Each row of parallel data produced by this process are referred to as channels. At a particular index $i$ in the produced data stream, the information present across each channel $c_j$ at that position ($c^i_j$) is connected.

Transcoding the encoded byte sequence for each input character could yield a different number of bytes in UTF-8 compared to the original input encoding. These differences are accounted for by the second channel (black) in this figure, which contains the byte offset information relative to the original input stream. For example, the Euro sign, highlighted in bold, is encoded in a single byte in the original document, but requires three bytes after transcoding to UTF-8. The zeros in the second channel for
the latter two bytes indicate no byte should be consumed in the original document when consuming this byte. Again, highlighted in bold, the ü also requires more bytes in UTF-8 than in the encoding of the original document.

Our framework chooses to normalise the input document into UTF-8 for a number of reasons. The first reason was stated earlier: most NLP applications expect UTF-8 input by default. The second, more pragmatic, reason is that our tokenization rules are defined in terms of UTF-8 byte sequences, not Unicode code points. This decision is discussed later in Section 7.2.3. The C++ DOCREP API can operate over UTF-8 or Unicode strings, so UTF-8 is not a limiting decision in terms of I/O.

No existing transcoding framework (e.g. iconv or ICU) supports the tracking of byte offsets back into an original document. Our tokenization framework supports conversion from most commonly used input encodings into UTF-8, maintaining offset information in the process. At the time of writing, 32 input encodings are supported; these encodings are listed in the documentation.

7.2.2 Document structure interpretation

Our tokenization framework is document format aware, meaning it knows how to tokenize with respect to the format and internal structure of the document. There are many advantages of this approach, including the document structure interpreter being able to inform the tokenizer about whitespace, paragraph, and sentence boundaries encoded within the document structure, as well as the overall framework being able to maintain offset information relative to the original document. If a separate pre-processing step was invoked to interpret the document structure, as in the case outlined earlier, the pre-processor needs to be offset-aware. Unfortunately most NLP tools still operate over vanilla plain text. If the document structure is removed during pre-processing, back-mapping the tokens to their location in the original document is a difficult, error-prone, and potentially inexact task.

\footnote{http://site.icu-project.org/}
Figure 7.3: The process of interpreting the document structure and producing a stream of text that can be passed to the tokenizer. The second channel maintains byte consumption counts relative to the original input stream, now accounting for document structure interpretation. The third channel contains byte skip consumption counts to skip over non-text document structure. The fourth channel contains control instructions to the tokenizer generated by the document structure interpretation.

A disadvantage of this tighter coupling of the tokenization and SBD layer and the document structure interpretation layer is that an offset-aware document structure interpreter needs to be implemented for each document format the users wishes to tokenize. Our framework currently supports the main document formats NLP corpora are distributed in: plain text, SGML, (well-formed) HTML, and WARC (ISO28500, 2009).

Figure 7.3 illustrates the transformation process that occurs to the data stream during the document structure interpretation layer of our tokenization framework. As with the input transcoding layer, the second channel on the produced output stream contains the byte offset information relative to the original input stream. However, the second channel of the input stream in Figure 7.3 is slightly different to the second channel of the output stream. In the output stream, document structure which encodes text content has been decoded, and the text appropriately inserted into the output stream. An example of this is the \&quot;:, highlighted in bold in the figure. This sequence of characters is a special escape sequence in HTML indicating a double straight quotation mark ("; U+0022). During document structure interpretation, this
7.2. The tokenization framework

escape sequence has been decoded and the appropriate text characters and byte offset consumption counts inserted.

Two additional channels have been added to the output stream during the document interpretation layer. The third channel (purple) contain byte skip counts indicating the consumption of a given number of bytes of non-text document structure. An example of this is the six bytes of HTML markup at the start of the document (\texttt{<p><b>}) before the first textual data (Sales). Highlighted in bold in Figure 7.3, these six bytes of markup are not reproduced on the output text stream. Instead, the document structure skip stream contains a value of 6 at the appropriate position. The document structure skip stream is 1 element longer than the other streams as document structure skip counts need to be accounted for before and after the text produced on the output stream.

The fourth channel (red) is a mostly empty control stream where the document structure interpreter can provide instructions to the tokenization and SBD. This control stream is used to inform the tokenization and SBD layer about token and sentence breaks encoded in the structure of the document which are not apparent in the produced text. Highlighted in bold in Figure 7.3, we can see that the HTML document structure interpreter has placed a newline indicator on the control stream in place of the interpreted \texttt{<br>} HTML tag because this tag indicates a line break.

The processes of input transcoding and document structure interpretation are intertwined. Depending on the document format being processed, these two layers might need to be run in either order, and potentially more than once. For example, the ClueWeb 2009 and 2012 datasets (Gabrilovich et al., 2013) consist of multiple Hypertext Transfer Protocol (HTTP) requests to retrieve HTML documents, and their associated HTTP responses. These HTTP requests and associated responses are stored in the WARC file format (ISO28500, 2009). A WARC file consists of multiple WARC documents. A WARC document contains key-value headers and a body HTTP response. A HTTP response contains key-value headers and a body HTML document. The encoding of the HTML content is not known until the document structure of the HTTP response is
Chapter 7. **DOCREP for Tokenization and SBD**

Figure 7.4: The token and sentence segmentations generated by the tokenization and SBD layer. Each token object has three attributes: its byte offset span over the original input stream, the raw UTF-8 text of the token (rendered here for clarity), and an optional normalised form of the token. In this example, the tokenizer has provided directional normalised versions of the quotation mark tokens.

processed to extract the HTTP headers, which in turn is not known until the structure of the WARC document is processed. If the encoding of the HTML document is not stated in the HTTP response headers, then the beginning of the HTML document needs to be processed in a shallow manner to extract the contents of any `<meta>` tags. If the input encoding still cannot be determined, common practice is to attempt to decode the encoded stream with a number of commonly-used encodings.

As this example shows, processing a WARC file requires multiple document structure interpretation steps as well as potentially multiple encoding transcoders. Processing ClueWeb requires WARC document format interpretation followed by HTTP response format interpretation, followed by transcoding, followed by HTML format interpretation. Maintaining offset information across these stages would be difficult without an integrated solution.

### 7.2.3 Tokenization and SBD

Once the document structure interpretation layer has produced an output text stream, the stream can be provided to the Unicode-aware tokenization and SBD layer. This
layer produces token and sentence segmentations directly as DOCREP annotations. Figure 7.4 shows the tokens and sentences produced by this process. Note that a sentence break has been produced after the `Sales` token as indicated by the control stream. As well as maintaining byte and Unicode code point offsets (not shown in the diagrams for brevity), our tokenizer will provide an optional normalised version of some token forms in addition to the raw form. These normalised variants include converting straight quotation marks into directional ones and down-mapping some Unicode punctuation to a canonical form. Highlighted in bold in Figure 7.4, we can see that the straight quotation mark tokens have directional normalised forms.

Another novel contribution of this tokenization framework is the separation of token and sentence-level segmentations from higher-level document structure, such as paragraphs and headings, while still giving the tokenizer access to this document structure. Our tokenization and SBD layer can only produce token and sentence annotations. It is the responsibility of the document structure interpretation layer to form larger segmental units, such as paragraphs. There is a clear motivation behind this — the definition of a higher level unit is defined by the document structure and not by the underlying text. The way this works in practice is that a document structure interpreter often invokes the tokenization layer multiple times per document; once per top level unit it wishes to form. For example, in a HTML or SGML interpreter, for each identified `<p>` node, the interpreter could interpret the node-internal markup and tokenize the resulting text. A paragraph annotation can then be created which spans over the sentences produced by the tokenization and SBD layer. This rich integration between the document structure interpretation layer and the tokenization and SBD layer allows accurate document structure modelling over the produced token and sentence annotations; something which is not easily achievable when document structure interpretation and tokenization and SBD are performed as separate processes.

The tokenizer we have implemented is a Unicode-aware rule-based tokenizer. We chose a rule-based system over a statistical machine learning system for runtime ef-
Figure 7.5: A small snippet from our Ragel tokenization rules. This snippet deals with various forms of numeric values. The machine-instantiated rules are defined in terms of UTF-8 byte sequences while the human-readable version is defined in terms of Unicode code points.

The human-readable form of these rules are defined in terms of Unicode code points, but the generated FSM operates over UTF-8. As an implementation detail, we populate the Ragel namespace with variables of the form `unicode_*` which have been automatically extracted from the Unicode database.\(^{10}\) For example, the `unicode_digit` identifier maps to the 550 Unicode code points in the Nd category, each UTF-8 encoded.

The decision to operate over UTF-8 rather than Unicode streams was for implementation reasons. If Unicode streams were used, the lookup tables in the table-driven FSM

---

\(^{8}\)http://www.cis.upenn.edu/~treebank/tokenization.html
\(^{9}\)http://www.colm.net/open-source/ragel/
\(^{10}\)http://www.unicode.org/Public/UCD/latest/
generated by Ragel would be impractically large for efficient use. Tokenization and SBD are precursor tasks to many NLP pipelines, and as such, should be performed as efficiently as possible so they are not a bottleneck.

As mentioned earlier, the transitions on the produced FSM execute heuristics to determine whether a sentence boundary has been encountered. These heuristics take into account different groups of observed punctuation as well as the control stream populated by the document structure layer. In practice, we find that these heuristics in combination with the document structure informed control stream produce accurate segmentations in a runtime and resource efficient manner.

7.3 Evaluation and comparison

In order to compare the speed and approximate recall of our tokenizer to existing tokenizers, we use the Wall Street Journal (WSJ) section of the Tipster corpus as test data.\footnote{\url{https://catalog.ldc.upenn.edu/LDC93T3A}} This corpus contains 173,252 WSJ documents between 1987 and 1992, stored in SGML markup. Figure 7.6 shows the first document from the 1987 section of the corpus. As shown, there is both explicit and implicit document structure in these documents. The explicit structure comes in the form of the SGML tags. Documents are identified by <DOC> nodes. Each document has a number of child nodes of potential interest: <DOCNO> for the document identifier, <H1> for the headline of the document, and <TEXT> containing the body text of the document. Implicit structure comes in the form of paragraph boundaries. By manual inspection, it appears that paragraph boundaries are encoded as a blank line within the body content of the <TEXT> node. Additionally, this document contains an SGML escape sequence: \&amp; representing a literal ampersand character.

For this experiment, we compare a number of different tokenizers: the PTB tokenization sed script, the NLTK Treebank tokenizer (Bird et al., 2009), the Stanford
Figure 7.6: The first WSJ document in the 1987 section of the Tipster corpus.

CoreNLP tokenizer (Manning et al., 2014), the default LingPipe\textsuperscript{12} tokenizer module, the downloadable OpenNLPl\textsuperscript{13} English tokenizer model, and our DOCREP tokenization framework. Apart from our system, none of the tested systems interpret document structure. As such, we run a pre-processing script to extract the raw text from the SGML. An additional SBD pre-processing step was run for the PTB \texttt{sed} and NLTK tokenizers as they assume the input is already in sentences. This pre-processing was performed using the NLTK SBD module. Our framework is capable of performing the document structure interpretation so to provide a fair comparison, we run it twice — once with the same plain text input that the other tokenizers see and once with the raw SGML input.

The results of this experiment can be seen in Table 7.1. The total execution time column represents the wall-clock time taken by each tokenizer, which includes I/O and other application logic. Our tokenization framework can perform a full SGML

\textsuperscript{12}\url{http://alias-i.com/lingpipe}
\textsuperscript{13}\url{https://opennlp.apache.org/}
7.3. Evaluation and comparison

<table>
<thead>
<tr>
<th>Tokenizer</th>
<th>Execution Time (s)</th>
<th>Output (#)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SGML</td>
<td>SBD</td>
<td>Tok’n</td>
<td>Total</td>
</tr>
<tr>
<td>PTB sed</td>
<td>162</td>
<td>220</td>
<td>196</td>
<td>578</td>
</tr>
<tr>
<td>NLTK</td>
<td>162</td>
<td>220</td>
<td>552</td>
<td>934</td>
</tr>
<tr>
<td>CoreNLP</td>
<td>162</td>
<td>26</td>
<td>29</td>
<td>228</td>
</tr>
<tr>
<td>LingPipe</td>
<td>162</td>
<td>6</td>
<td>9</td>
<td>187</td>
</tr>
<tr>
<td>OpenNLP</td>
<td>162</td>
<td>17</td>
<td>894</td>
<td>1080</td>
</tr>
<tr>
<td>DOCREP&lt;sub&gt;text-only&lt;/sub&gt;</td>
<td>162</td>
<td>—</td>
<td>53</td>
<td>234</td>
</tr>
<tr>
<td>DOCREP&lt;sub&gt;doc-aware&lt;/sub&gt;</td>
<td>36</td>
<td>—</td>
<td>53</td>
<td>160</td>
</tr>
</tbody>
</table>

Table 7.1: Tokenization statistics for the WSJ section of the Tipster corpus. The execution time columns report respectively, in seconds, SGML extraction, sentence boundary detection (SBD), tokenization (Tok’n), and total time. The number of produced sentences and tokens are reported in the final two columns.

parse, Unicode-aware tokenization, and output a structured DRF serialisation of the document over 3 times faster than the PTB sed script which is performing only simple textual replacements. Additionally, our document-aware tokenizer runs faster than all of the other compared tokenizers, including almost 7 times faster than OpenNLP. We do not report SBD times for our system as the tokenization and SBD are jointly performed.

The number of produced tokens and sentences differs wildly between tokenizers. LingPipe runs quickly but produces significantly more tokens than the other systems. These two artefacts are related — this tokenizer performs only simple string splitting rules with little additional logic. These simple splits are fast to execute but over-split many tokens, producing ~25 million more tokens than the other tokenizers. This over-splitting is apparent in Figure 7.7, where we show the tokens and sentence boundary decisions made by our system and LingPipe for the document shown in Figure 7.6.
John Blair & Co. is close to an agreement to sell its TV station advertising representation operation and program production unit to an investor group led by James H. Rosenfield, a former CBS Inc. executive, industry sources said. Industry sources put the value of the proposed acquisition at more than $100 million.

John Blair was acquired last year by Reliance Capital Group Inc., which has been divesting itself of John Blair’s major assets.

John Blair represents about 130 local television stations in the placement of national and other advertising.

Mr. Rosenfield stepped down as a senior executive vice president of CBS Broadcasting in December 1985 under a CBS early retirement program.

Neither Mr. Rosenfield nor officials of John Blair could be reached for comment.

Figure 7.7: A comparison of the tokens and sentence boundaries produced by our system and LingPipe.
The system closest to ours in terms of the number of tokens and sentences produced is CoreNLP. By manual inspection, we observe that most of the differences fall into a couple of categories:

- Following the PTB guidelines, CoreNLP inserts an end of sentence period as an additional token if the sentence ends in a period-ending acronym. We do not perform this insertion. For example, we produce:

```
ATARI CORP., $ 75 million of convertible debentures due 2012, via Paine Webber Inc.
```

whereas CoreNLP produces:

```
ATARI CORP., $ 75 million of convertible debentures due 2012, via Paine Webber Inc.
```

- CoreNLP does not yield multiple tokens for split fractions. Instead, it creates a single token with a non-breaking space between the fragments. For example, we produce:

```
Separately, Universal said its 15\text{3/4}\% debentures due Dec. 15, 1996, will be redeemed April 19.
```

whereas CoreNLP produces:

```
Separately, Universal said its 15\text{3/4}\% debentures due Dec. 15, 1996, will be redeemed April 19.
```

The difference is in the whitespace of the highlighted region. Our system produces 15 and 3/4 as two separate tokens, whereas CoreNLP produces 15\text{3/4} as a single token with a non-breaking space (U+00A0) between the fragments.

- Most other differences come from ambiguous sentence boundaries involving period-ending acronyms mid-sentence. Sometimes our system erroneously splits and other times CoreNLP erroneously splits. For example, we produce:

```
The award was made by a unit of Algoma Steel Corp. Ltd., Sault Ste. Marie, Ontario.
```

whereas CoreNLP produces:

```
The award was made by a unit of Algoma Steel Corp. Ltd., Sault Ste. Marie, Ontario.
```
We conclude that the tokenization and SBD segmentations made by our framework are sensible. They roughly equate in quality to the segmentations produced by CoreNLP and are better quality than the segmentations produced by LingPipe.

7.4 DOCREP models

Figure 7.8 shows the relevant code snippets from the DOCREP models used our tokenization and SBD framework. These models are very straightforward. Both sets of maintained token offset counts are represented with a slice, and the raw and optional normalised token forms are stored with a regular C++ string. The sentence model is a simple span over a block of Token objects.

7.4.1 Discontiguous spans

Discontiguous spans are layers of annotation that are broken by other intervening document or language structure. The work in this chapter directly addresses the document structure case. For example, an NLP pipeline represents the document structure explicitly between and within tokens. It can break a token into pieces, such as in this HTML snippet: `<b>doc</b>ument <b>rep</b>resentation`. Here, the tokens `document` and `representation` have infix document markup. Our tokenization framework handles this inherently and produces the desired tokens. The begin and end byte offsets for each token will include the document markup.

In other cases, it is the language, and not the document (or other) structure, that produces discontiguous spans. One example of this is infix morphology in languages like Finnish. In these situations, the desired tokenization may not be contiguous. DOCREP models this using an additional annotation layer over the token parts themselves. Most tokens will only have a single token part, but some will have multiple token parts that can be reordered. A similar mechanism can be used for reordering in machine translation applications.
7.5. Summary

Maintaining token offset information relative to the original document is increasingly important for NLP tasks (e.g. TAC KBP), and user-facing NLP applications. This chapter presents a novel tokenization framework which jointly performs transcoding, document structure interpretation, tokenization, and sentence boundary detection so that downstream NLP applications have access to offsets into the original document. We have shown that the process of maintaining offsets can be performed efficiently using DOCREP as the structured output format.

The next chapter goes on to use DOCREP for NER, implementing document structure features into a NER system to achieve state of the art results.

```cpp
class Token : public dr::Ann {

public:
    dr::Slice<uint64_t> byte_span;
    dr::Slice<uint64_t> char_span;
    std::string raw;
    std::string norm;
};

class Sentence : public dr::Ann {

public:
    dr::Slice<Token *> span;

};
```

Figure 7.8: The relevant code snippets from the Token and Sentence DOCREP models used in our tokenization and SBD framework.
8 DOCREP for NER

In Chapter 7, we utilised DOCREP to represent tokenization, sentence boundary, and document structure annotations. In this chapter, we move to using DOCREP primarily as a consumer of annotations. This chapter describes our newly developed state-of-the-art named entity recognition (NER) system which utilises document structure information provided by DOCREP. This system also takes advantage of DOCREP’s native span representation for working with named entity annotations.

We begin this chapter with a review of named entity recognition, describing datasets, learning methods, features, and current state-of-the-art performance. We then go on to show how having an underlying document representation can help improve NER performance by implementing a new NER system. This new NER system utilises best practice methods and exploits our document structure to produce state-of-the-art performance on multiple datasets.

8.1 Named Entity Recognition

Information extraction (IE) is the task of extracting structured information from unstructured documents. As a subtask of IE, named entity recognition (NER) is the task of identifying spans of tokens naming an entity, and classifying them as belonging to a one of a pre-defined set of categories, such as person, location, organisation, geo-political entity, etc. This task was initially defined in the DARPA-funded Message Understanding Conferences (MUC) which ran during the 1990’s. Since then, NER has become a
crucial precursor task of many NLP pipelines, including named entity linking (NEL) and relation extraction.

In this section we first outline the way NER systems typically define the task as a machine learning (ML) problem. We outline the shared tasks on NER that have occurred over the years and the datasets they yielded, as well as other NER datasets that have been created. The issues involved with evaluating NER systems are then discussed, and how they relate to the shared tasks and the evaluation of real-world systems. After this, the discussion moves covers the learning methods and external resources commonly used.

8.1.1 Sequence tagging

Sequence labeling (or sequence tagging) is a machine learning task involving the classification of each item in a given sequence with one of the categories learnt during training. The act of classification normally involves consulting a probabilistic model and returning the most likely category for the current item.

Many NLP problems are posed as a sequence tagging problem. The most commonly known examples are POS tagging (Brill, 1993; Ratnaparkhi, 1996), syntactic chunking (Ramshaw and Marcus, 1995), and NER. In NER and syntactic chunking, each token is assigned a label which encodes its position with a NE or chunk span, as well as the category of the span. A special label is normally added to the label set to encode “not part of an entity”. How one chooses to encode “is part of an entity” onto the tokens can greatly affect sequence tagging performance (Tjong Kim Sang and Veenstra, 1999).

Sequence tag encoding

Tjong Kim Sang and Veenstra (1999) performed the first analysis of how the encoding of span category information into the sequence tag label can affect accuracy. The authors present a number of different sequence tag encodings, shown in the top section of
Figure 8.1, which they use in NP chunking experiments. The definition of the different sequence tag encoding schemes is as follows:

IO [ Tokens which begin a span are labelled I. All other tokens are labelled O.

IOB1 Tokens inside a span are labelled I. All other tokens are labelled O. This technique was introduced and used in Ramshaw and Marcus (1995). The IOB1 encoding extends the IO encoding to support the identification adjacent spans. Without B, two adjacent spans would be seen as one contiguous sequence of I tags, and the boundary information is lost.

IOB2 Tokens inside spans are labelled B if they are the first token or I otherwise. All tokens not inside a span are labelled O. This technique was introduced and used in Ratnaparkhi (1998). The IOB2 encoding states that the identification of the start of a span is easy.

IOE1 This is the same as IOB1 except that instead of the beginning token of a span being labelled, the end-of-span token on a neighbouring span boundary is labelled E.

IOE2 This is the same as IOB2 except that instead of the beginning token of a span being labelled, the end-of-span token is labelled E. IOE2 states that the identification of the end of a span is easy.

BMEWO While not explicitly stated with these letters, Borthwick (1999) introduces this encoding in some of his features. Single-token spans are labelled W (word). Beginning, middle, and end of span tokens are then labelled B, M, and E respectively. All other tokens are labelled O. Ratinov and Roth (2009) use this encoding but
under a different set of letters (BILUO). An example of this encoding is shown in the bottom row of Figure 8.1. The BMEWO encoding lexicalises more information about the position of the token within a span, and additionally the cardinality of the span.

Tjong Kim Sang and Veenstra (1999) show that by changing the sequence tag encoding only, they could achieve state-of-the-art performance on their NP chunking task. This observation was further explored for syntactic chunking, with techniques including lexicalising the encodings (Molina and Pla, 2002), voting between different sequence tag encoding schemes (Shen and Sarkar, 2005), and automatically searching for the best encoding scheme for the dataset (Loper, 2007).

When used in an NER setting, these sequence tags are usually combined with the category of the spanning named entity. For example, if the named entity Westpac Banking Corporation (ORG) was being sequence tag encoded with the BMEWO encoding, the position-in-span labels assigned to each token would be B-ORG, M-ORG, and E-ORG respectively. An example of how the different sequence tag encodings are utilised in NER is shown in Figure 8.2. The position-in-span label is used as a prefix to the category of the named entity. If the named entity dataset has \( C \) categories and the sequence tag encoding scheme used has \( P \) prefixes, the number of resulting labels that
8.1. Named Entity Recognition

<table>
<thead>
<tr>
<th></th>
<th>The</th>
<th>Swiss</th>
<th>Grand</th>
<th>Prix</th>
<th>1994</th>
<th>World</th>
<th>Cup</th>
<th>race</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOBI</td>
<td>0</td>
<td>I-MISC</td>
<td>B-MISC</td>
<td>I-MISC</td>
<td>B-MISC</td>
<td>I-MISC</td>
<td>I-MISC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IOB2</td>
<td>0</td>
<td>B-MISC</td>
<td>B-MISC</td>
<td>I-MISC</td>
<td>B-MISC</td>
<td>I-MISC</td>
<td>I-MISC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BMEWO</td>
<td>0</td>
<td>W-MISC</td>
<td>B-MISC</td>
<td>E-MISC</td>
<td>B-MISC</td>
<td>M-MISC</td>
<td>E-MISC</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 8.2: An example sentence with IOBI, IOB2, and BMEWO named entity tags.

the model sees is $CP + 1$: each category appears labelled with each prefix, plus the special 0 label to indicate not an entity.

Despite experiments in chunking, the use of different sequence tag encodings in NER had been mostly ignored until Ratinov and Roth (2009). Here, the authors switch to using a BMEWO encoding rather than the IOBI encoding that the CoNLL 2003 gold-standard annotations were distributed in. They report their final system performance on two test sets using IOBI and BMEWO encoding, and show that using the richer BMEWO encoding significantly increases the performance of their system, jumping by 1.42 $F_1$.

It should be noted that these sequence tag encoding schemes have an implicit assumption in them — the spans which are being encoded as projections onto the tokens are not overlapping or nested. If the spans are overlapping or nested, the projection at the token level needs to encode the membership of the potentially multiple spans the token is a member of. Alex et al. (2007) outlines some techniques for posing the task of nested named entity recognition as a sequence labelling problem. This is achieved by projecting different subsets of information from multiple levels of the nested entity tree structure down into the encoded label on each token.

**Native span representation**

Since most corpora and linguistic annotations are not distributed via a DRF, named entity annotations need to be stored on disk in a sequence tag encoded format so that the boundary point between directly adjacent NEs of the same category can be identified. This is suboptimal for multiple reasons. First, changing sequence tag encoding requires
Figure 8.3: NE spans can be natively represented in a DRF such as DOCREP. decoding the tags back to a spans model (which a DRF natively provides), and then projecting the new encoding back down onto the tokens. This is an error prone task and requires both encoding and decoding code to be written. Second, performing metadata analysis on the documents, such as counting how many NEs there are, can be a non-trivial operation depending on the encoding used.\footnote{This count can be calculated with a simple `awk` `\'$2 ~ /^[BW]-[A-Z]+$/` | `wc -l` for IOB2/BMEWO-encoded CoNLL format data, but not for IOBI.}

By using a DRF such as DOCREP to store span annotations, the span and the NE category are stored separately as first class data attributes, and it is the job of the application layer to project the NE category down to the token level with the appropriate sequence tag encoding. The DOCREP “sequence tagger” decorator make this projection operation simple to perform (Section 4.4.3). Figure 8.3 shows how NEs can be represented in DOCREP. The fact that NER is often modelled as a sequence tagging task is not a good reason for storing the linguistic annotations in a sequence tag encoded manner. Additionally, since the NE is modelled natively as a span over tokens, the DRF supports operations for a token to know which NE objects it is spanned by.

Nested named entity recognition has issues in span representation. The CoNLL format cannot easily be used due to its flat one-token-per-line format. Most nested NE corpora choose to use XML with nested annotation tags, but this limits the tags to not be overlapping — overlapping tags are not valid XML. Both of these problems are trivially solved by using a DRF for span representation as the nested case is modelled no differently to the non-nested case. It is up to the application layer to establish how it wants to utilise the nested tags at runtime.
8.1. Named Entity Recognition

The Pentagon has denied a request that top U.S. commanders in Hawaii in 1941 be absolved of blame for failing to be on alert for the Japanese attack on Pearl Harbor, but the military agreed that top Washington officials also must share the blame.

A Pentagon study re-affirmed the conclusion of previous government investigations that both Rear Admiral Husband E. Kimmel and his Army counterpart, Maj. Gen. Walter C. Short, "committed errors of judgment" leading up to the Dec. 7, 1941, debacle.

Figure 8.4: A snippet from the MUC-7 NER training data.

8.1.2 Shared tasks and data

MUC-6 and MUC-7 shared tasks

Between 1987 and 1997, the Defense Advanced Research Projects Agency (DARPA) run the Message Understanding Conference (MUC). The purpose of these conferences was to better understand the IE techniques of the time. MUC-6 (Sundheim, 1995) and MUC-7 (Chinchor, 1998) were amongst the earliest formal shared tasks in NLP. These shared tasks required participants to identify named entities in seven categories, broken down into three groups: entity expressions, temporal expressions, and number expressions. The entity expression group consisted of three categories: LOCATION, ORGANISATION, and PERSON; temporal expressions consisted of two categories: DATE, and TIME; and number expressions consisted of two categories: MONEY, and PERCENT.

The documents in both MUC-6 and MUC-7 consisted of American and British English newswire text. The data for the MUC NER shared tasks was distributed as untokenized SGML files, with inline named entity annotation tags. An example of this can be seen in Figure 8.4. The named entity annotations are SGML tags enclosing the text which they span, with a SGML tag attribute indicating the category of the entity.
**MET-1 and MET-2 shared tasks**

Following the success of the MUC-6 shared task, the Multilingual Entity Tasks (MET) were established to assess the performance of NER on languages other than English. The MET tasks (Merchant et al., 1996) provided participants with documents of Spanish, Chinese, and Japanese newswire text. To simplify the technical cost of participation, MET used the same data format and NE category definitions as the MUC shared tasks.

**CoNLL 2002 and 2003 shared tasks**

Many of the techniques used for the MET and MUC shared tasks relied on language-specific resources. After these shared tasks, a number of papers began to apply more statistically-driven techniques to NER in languages other than English. Palmer and Day (1997) applied statistical methods to locate NEs in multilingual newswire text, finding that the difficulty of each language varied but that a large percentage of the task could be done with simple methods. Following this, Cucerzan and Yarowsky (1999) implemented a fully language-independent NER pipeline, yielding $F_1$-scores between 40 and 70 depending on the language.

The Conference on Natural Language Learning (CoNLL) ran two shared tasks on language-independent named entity recognition. CoNLL 2002 (Tjong Kim Sang, 2002) concentrated on non-English, asking participants to identify NEs in Spanish and Dutch newswire text, while CoNLL 2003 (Tjong Kim Sang and De Meulder, 2003) asked participants to identify NEs in English and German newswire text. All four sets of data were partitioned into training, development, and test splits. Both shared tasks required participants to identify named entities in four categories: location (LOC), organisation (ORG), person (PER), and miscellaneous (MISC).

The data for these shared tasks was distributed as a plain-text one-token-per-line format, with a blank line indicating a sentence boundary. This file format later became known as the “CoNLL format” and was expanded for other tasks including semantic role labelling, dependency parsing, and coreference resolution. The special token
8.1. Named Entity Recognition

-DOCSTART- -X- O O

EU NNP I-NP I-ORG
rejects VBZ I-VP O
German JJ I-NP I-MISC
call NN I-NP O
to TO I-VP O
boycott VB I-VP O
British JJ I-NP I-MISC
lamb NN I-NP O
.. O O

Peter NNP I-NP I-PER
Blackburn NNP I-NP I-PER
BRUSSELS NNP I-NP I-LOC
1996-08-22 CD I-NP O

The DT I-NP O
European NNP I-NP I-ORG
Commission NNP I-NP I-ORG
said VBD I-VP O
on IN I-PP O
Thursday NNP I-NP O

it PRP B-NP O
disagreed VBD I-VP O
with IN I-PP O
German JJ I-NP I-MISC
advice NN I-NP O
to TO I-PP O
consumers NNS I-NP O
to TO I-VP O
shun VB I-VP O
British JJ I-NP I-MISC
lamb NN I-NP O
until IN I-SBAR O
scientists NNS I-NP O
determine VBP I-VP O
whether IN I-SBAR O
mad JJ I-NP O
cow NN I-NP O
disease NN I-NP O
can MD I-VP O
be VB I-VP O
transmitted VBN I-VP O
to TO I-PP O
sheep NN I-NP O
.. O O

Figure 8.5: A snippet from the CoNLL 2003 English NER training data.

-DOCSTART- was used to indicate a document boundary so that multiple documents could be represented within a single flat file. A snippet from the English training data can be seen in Figure 8.5.

For the CoNLL shared tasks, the linguistic data provided with each document is not as consistent as in the MUC shared tasks. The Spanish CoNLL 2002 data contains tokens and IOB2-encoded NE labels, and does not provide any document boundary information. The Dutch CoNLL 2002 data contains tokens, POS tags, and IOB2-encoded NE labels, but does provide document boundary information. Both the English and German CoNLL 2003 datasets contain token, POS tag, syntactic chunk tag, and IOB1-encoded NE labels, as well as document boundary information. The German data additionally contains lemma information for each token. The breakdown of this data by token, sentence, document, and named entity count is shown in Table 8.1.

The CoNLL 2003 English dataset has become the de facto canonical evaluation set for English NER. Unfortunately, this dataset has many well known issues, and some lesser known issues. The well known issues include the fact that the test set is a lot harder than the development set, with the documents frequently discussing
Table 8.1: Size breakdown of CoNLL shared task data splits. The Spanish data from CoNLL 2002 does not contain document boundary information.

names of sports teams whose names are locations. Some of the lesser known issues include tokenization mistakes and many sentence boundary errors arising from the tokenization and SBD not being gold standard (see Section A.2 for more details). The breakdown of this data by entity category is shown in Table A.1.

ACE

The Automatic Content Extraction (ACE) 2008 shared task required participants to perform NER on a variety of English and Arabic documents of varying source domains and formats. The training data for ACE is available from the LDC. The ACE became the foundation for the Knowledge Base Population (KBP) track of the Text Analysis Conference (TAC) in 2009.

\(^2\)https://catalog.ldc.upenn.edu/LDC2014T18
8.1. Named Entity Recognition

OntoNotes

The OntoNotes 5 corpus (Hovy et al., 2006; Weischedel et al., 2011) is discussed in detail in Section 5.1. Of the 15,710 documents in the corpus, 13,109 are in English, 2002 are in Chinese, and 599 are in Arabic. Of the 13,109 English documents, there are 3,637 that have named entity annotations.

OntoNotes does not provide any official or suggested training splits due to the fact that different documents have different sets of annotation layers, and so the notion of a “good” set of splits becomes task dependant. The CoNLL 2011 and 2012 shared tasks on coreference resolution created multilingual stratified splits for OntoNotes. The algorithm they used to create the splits is outlined in Pradhan et al. (2011). These splits are available online\(^3\) and have been used as the training, development, and test splits for the OntoNotes 5 corpus. Pradhan et al. (2013) used a variation on the CoNLL 2012 splits such that the documents in the test sets for each language always contained all annotation layers, including coreference (the smallest annotation layer in OntoNotes).

Not all of the documents in the OntoNotes 5 corpus contain NE annotations. For the purposes of English NER discussion, we exclude the documents in the CoNLL 2012 splits which do not contain NE annotations, as well as documents which are not in the English portion of the corpus. The breakdown of these splits can be seen in the first column of Table 8.2.

These NE annotations are across 18 categories (Weischedel and Brunstein, 2005), significantly more than the MUC or CoNLL shared tasks. The categories are CARDINAL, DATE, EVENT, FAC, GPE, LANGUAGE, LAW, LOC, MONEY, NORP, ORG, PERCENT, PERSON, PRODUCT, QUANTITY, TIME, and WORK_OF_ART. The distribution of these categories across the English CoNLL 2012 splits can be seen in Table B.1. The OntoNotes 5 category distribution is dominated by a small selection — just four out of the 18 categories (DATE, GPE, ORG, and PERSON) account for 67.5% of the entities, while seven of the 18 categories each contribute less than or equal to 1% of the entity counts.

\(^3\)http://conll.cemantix.org/2012/download/ids/
<table>
<thead>
<tr>
<th></th>
<th>CoNLL 2012</th>
<th>Passos et al. (2014)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>train</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tokens</td>
<td>1 644 219</td>
<td>1 329 197</td>
</tr>
<tr>
<td>sentences</td>
<td>82 122</td>
<td>52 783</td>
</tr>
<tr>
<td>documents</td>
<td>2 946</td>
<td>3 141</td>
</tr>
<tr>
<td>NEs</td>
<td>128 794</td>
<td>138 256</td>
</tr>
<tr>
<td><strong>dev</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tokens</td>
<td>251 043</td>
<td>102 201</td>
</tr>
<tr>
<td>sentences</td>
<td>12 678</td>
<td>4 313</td>
</tr>
<tr>
<td>documents</td>
<td>430</td>
<td>231</td>
</tr>
<tr>
<td>NEs</td>
<td>20 366</td>
<td>10 620</td>
</tr>
<tr>
<td><strong>test</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tokens</td>
<td>172 077</td>
<td>125 220</td>
</tr>
<tr>
<td>sentences</td>
<td>8 968</td>
<td>5 093</td>
</tr>
<tr>
<td>documents</td>
<td>2 61</td>
<td>2 61</td>
</tr>
<tr>
<td>NEs</td>
<td>12 594</td>
<td>12 586</td>
</tr>
</tbody>
</table>

Table 8.2: Size breakdown of the two known OntoNotes 5 splits. The “official” splits come from the CoNLL 2012 shared task. Passos et al. (2014) report performance on their own splits, which deviate from the CoNLL 2012 splits in a number of ways.

We are only aware of two publications which report English NER performance on the OntoNotes 5 CoNLL 2012 splits: Pradhan et al. (2013) and Passos et al. (2014). The script used to create the splits in Pradhan et al. (2013) had bugs. This script has since been corrected and is available online, but the old version is not available, prohibiting comparison against the reported numbers. Table 8.3 shows the entity count discrepancies between what is reported in Pradhan et al. (2013), what is contained in both the flat-file and database versions of the official LDC data,\(^4\) and what is in Pradhan’s OntoNotes 5 CoNLL-formatted release.\(^5\)

The data sizes reported in Passos et al. (2014) also did not match the CoNLL 2012 splits mentioned above. We contacted the authors, and they stated that they were given the splits and did not know how they were created. Additionally, they did not have the document IDs for the documents in their splits — they only had sequenced tagged

---

\(^4\)https://catalog.ldc.upenn.edu/LDC2013T19
\(^5\)http://cemantix.org/data/ontonotes.html
Pradhan et al. (2013)
LDC data
Pradhan CoNLL format
Passos et al. (2014)

<table>
<thead>
<tr>
<th></th>
<th>BC</th>
<th>BN</th>
<th>MZ</th>
<th>NW</th>
<th>TC</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pradhan et al. (2013)</td>
<td>1671</td>
<td>2180</td>
<td>1161</td>
<td>4679</td>
<td>362</td>
<td>1133</td>
</tr>
<tr>
<td>LDC data</td>
<td>1697</td>
<td>2184</td>
<td>1163</td>
<td>4696</td>
<td>380</td>
<td>1137</td>
</tr>
<tr>
<td>Pradhan CoNLL format</td>
<td>1697</td>
<td>2184</td>
<td>1163</td>
<td>4696</td>
<td>386</td>
<td>1131</td>
</tr>
<tr>
<td>Passos et al. (2014)</td>
<td>1697</td>
<td>2184</td>
<td>1163</td>
<td>4696</td>
<td>386</td>
<td>1131</td>
</tr>
</tbody>
</table>

Table 8.3: Discrepancies in the reported number of entities in the CoNLL 2012 test set using the OntoNotes 5 corpus. The columns indicate sections of the OntoNotes corpus.

data in CoNLL format. The authors were able to provide us with a copy of their split data. Since we would like to report comparable numbers to the numbers reported in this paper, we reverse engineered how these splits were created. A discussion about the reverse engineered procedure and the algorithm for generating the Passos et al. (2014) splits is presented in Section B.3. If the creators of these splits were using a DRF such as DOCREP, it is possible that these splits might not have been different to the original CoNLL 2012 splits.

Since we would like to compare these numbers, we will need to train and test on these splits. The data breakdown for the Passos et al. (2014) splits can be seen in the second column of Table 8.2. One immediate difference in the data distribution between these splits and the CoNLL 2012 splits (Table 8.2) is the relative number of NEs in each of the split files. The distribution is roughly 80%/12%/8% for the CoNLL 2012 splits, but for the Pradhan and Passos splits the distribution is roughly 86%/6%/8%. We hypothesise that the combination of having more entities to train on and there being no sentences without any named entities makes the data in these splits easier than the CoNLL 2012 splits. The category breakdown for these splits can be seen in Table B.2.

Another partitioning of the English OntoNotes NER data is presented in Finkel and Manning (2009), and then used again in Finkel and Manning (2010) and Tkachenko and Simanovsky (2012). This partitioning uses six of the broadcast news subcorpora of the OntoNotes 2 release, which Finkel and Manning (2009) used to train joint parsing
and NER models. However, due to data inconsistencies in this release of OntoNotes, they manually corrected a number of the NE annotations. Additionally, they collapsed the 18 NER categories down to just four: the three most dominant categories stayed and the rest were collapsed into a new MISC category. Each subcorpus was split into a 75% training set and 25% testing set, and a joint parser-NER model was trained and tested per subcorpus. Their joint model performed well, with improvements up to 1.35% $F_1$ for parsing and up to 9.0% $F_1$ for NER, but we are unsure how these numbers would compare with the current release of OntoNotes without their manual corrections, nor how they would compare when training a large joint model using all of the data OntoNotes provides.

Che et al. (2013) use the named entity annotation layers from the OntoNotes 4 corpus in conjunction with the fact that OntoNotes has some parallel English/Chinese texts to perform NER with bilingual constraints. They use all parallel English/Chinese documents which have NE annotations as their training and test data, and all other English and Chinese documents which have NE annotations as their monolingual training data. Additionally, they only use the four most frequent NE categories — all other NE instances were discarded. Performance of their monolingual English and Chinese NER models are reported in addition to their joint model, but this evaluation is only performed over the parallel English/Chinese texts.

**Domain adaptation and domain-specific corpora**

Poibeau and Kosseim (2001) showed that newswire-trained NER systems perform poorly when applied to email text. Recently, using NLP systems on social media data has become a popular topic, especially the use of Twitter data. The language used in the average tweet differs substantially from the language used in newswire, and as a result, state-of-the-art NER systems perform very poorly — overall $F_1$ of 41% (Ritter et al., 2011). A number of approaches to NER domain adaptation have been attempted, including using search engines to help with ambiguous decisions (Rüd et al., 2011),
augmenting features to utilise data from multiple domains (Daumé III, 2007), and utilising word cluster and embeddings trained on large collections of text (Turian et al., 2010).

Domain-specific NER corpora have been created in order to improve in-domain NER performance and extract NE categories of importance to those domains. Domain-specific NER normally also includes NE categories specific to that domain. Unsurprisingly, a NER system trained on English newswire does not perform well on biomedical text, nor on tweets from Twitter. Scientific domains contain a lot of jargon, as well as sentence structure and mathematical formula that are not seen in the newswire domain. A number of biomedical NER corpora have been created, including the nested named entity GENIA corpus (Kim et al., 2003), the NLPBA corpus (Kim et al., 2004), the EPPI corpus (Alex et al., 2007), and the BioInfer corpus (Pyysalo et al., 2007). There has also been some NER work in the astronomy domain (Murphy et al., 2006).

Corpora containing document structure

To the best of our knowledge, no NER training corpus with rich document structure exists. If such a corpus did exist, it would allow us to properly explore the use of document features in NER systems. Given the current state of NER corpora, we have chosen to use the OntoNotes corpus for our exploration. We look forward to further investigation when a document structure rich corpus is created.

8.1.3 Evaluation

There are many aspects of NER which make it difficult to define a standard evaluation metric for all situations. The two main types of error are boundary errors and categorisation errors. Figure 8.6 shows an example sentence with an entity mention referring to the United States. The first three examples in this figure show the possible valid boundaries for the United States as token when treated as a location. The fourth example shows a type mismatch, where the entity was classified as an organisation
Figure 8.6: Possible valid NE annotations for the entity string the U.S. appearing at
the end of a sentence. The first three cases show possible correct entity boundaries.
The fourth example makes an entity category distinction between the country as a
location and the country as an organisation (referring to the government, for example).

(The government), instead of as the location (country). NER evaluation metrics vary
primarily on how strict they are on these two issues.

The evaluation procedure used for MUC (Chinchor, 1998) awarded correct category
and correct bounds equally. The correct category match was awarded where the
predicted category of an entity was correct with at least one boundary correct. The
correct bounds match was awarded when both bounds for the entity were correct,
regardless of the predicted category. One criticism of this evaluation procedure is that
awarding both of these equally is unrealistic in most situations, as some boundary
errors are more significant compared to others.

The CoNLL 2002 and 2003 shared tasks both used the same evaluation script,
conlleval, which was originally created and used by the CoNLL 2000 shared task
on chunking (Tjong Kim Sang and Buchholz, 2000). conlleval awards exact phrase
matching, stating that both bounds as well as the entity category have to be correct.
This can be seen as the harshest form of sequence tag evaluation, and provides a
lower-bound across NER evaluation metrics.

Both the MUC evaluation and conlleval report performance on true positives (tp;
correctly predicted), false positives (fp; incorrectly predicted), and false negatives (fn;
8.1. Named Entity Recognition

not predicted as being an entity). From these counts, they calculate precision ($P$) and recall ($R$) per category as well as overall (micro-averaged).

$$P = \frac{tp}{tp + fp} \quad \quad R = \frac{tp}{tp + fn}$$

Precision and recall are then combined into an $F_\beta$-score value:

$$F_\beta = (\beta^2 + 1) \cdot \frac{PR}{(\beta^2 P + R)}$$

$\beta$ is set to 1 during evaluation, weighting both precision and recall equally. This reduces the formula down to the harmonic mean of the two values:

$$F_{\beta=1} = \frac{2PR}{P + R} \quad (8.1)$$

$F_1$-score is also known as $F$ measure.

conlleval takes as input NE sequence tagged sentences in CoNLL format. Since this evaluation metric cares about entity bounds, it has to perform the appropriate decoding of the various sequence tag encodings back into entity spans. If NE systems produced their annotations in a DRF such as DOCREP, this tricky and messy decoding process would not be the job of the evaluation script. This leaves the authors of the evaluation script to focus purely on the evaluation logic, rather than the input format.

The evaluation procedure used by ACE is more complicated than MUC or CoNLL. ACE avoids the traditional $F_1$-score by using a customisable parameterised evaluation metric where different kinds of errors can have different weights. Nadeau and Sekine (2007) provide a good summary of the evaluation procedure. While being the most complex and arguably most powerful NER evaluation metric, the scores produced by this metric are only comparable if the parameters are fixed. Additionally, a more complex scoring function means that error analysis can be more difficult to perform.

Manning (2006) argues that any evaluation metric which uses a combination of precision and recall (such as $F_1$-score) is biased towards systems which do not tag entities with ambiguous bounds. This is due to the fact that assigning an entity the
wrong label, getting an ambiguous bound wrong, or getting both wrong is penalised doubly by $F_1$-score, as both $fp$ and $fn$ are increased. Instead of optimising using a metric which combines precision and recall, Manning proposes counting labeling errors, boundary errors, and label-boundary errors in addition to exact match $tp$, $fp$, and $fn$. To provide evidence for this proposal, Manning analysed his own output from CoNLL 2003 and found that over two-thirds of the errors produced by his system belonged to one of these three additional categories; the categories that are multiply penalised by $F_1$-score.

Despite its criticisms, conlleval has become the canonical evaluation procedure for English NER systems due to both the prevalence of the CoNLL 2003 corpus as training data and its simplicity of implementation and execution (a single Perl script). We use this evaluation procedure for our experiments in order to report numbers comparable to existing systems and publications.

### 8.1.4 Learning method

For a NER system to be useful, it needs to perform well at identifying and classifying previously unseen entities. While early NER systems were heavily rule-based, the use of machine learning techniques and supervised learning quickly produced state-of-the-art results. Techniques for supervised learning include Hidden Markov Models (HMM) (Leek, 1997; Freitag and McCallum, 1999), Maximum Entropy (MaxEnt) models (Jaynes, 1957; Bergert et al., 1996), Perceptrons (Rosenblatt, 1957; Freund and Schapire, 1999), and Conditional Random Fields (CRF) (Lafferty et al., 2001; McCallum and Li, 2003). More recently, CRF and perceptron-based learning methods have become popular for NER systems, with the two most well-known English NER systems using these: the Stanford Named Entity Recogniser (Finkel et al., 2005) and the Named Entity Tagger (Ratinov and Roth, 2009).

CRFs have increasing use in the field of NER, with many top-performing systems utilising them (McCallum and Li, 2003; Finkel et al., 2005; Passos et al., 2014). The most
The news agency Tanjug reported that the airport Tanjug said... Figure 8.7: Ideally, the two instances of Tanjug should get the same label.

common configuration of the CRF in NER is the linear chain CRF (Lafferty et al., 2001), which has an efficient exact inference algorithm analogous to the forwards-backward algorithm (Baum and Petrie, 1966) and Viterbi algorithm (Viterbi, 1967) in the case of HMMs. Exact inference is only tractable when the graphical structure of the CRF is a tree or a linear chain. When modeling non-local information, being restricted to a non-cyclic graphical structure is limiting. Consider the document fragment shown in Figure 8.7, where there are two instances of the entity span Tanjug. Ideally, these instances should receive the same NE category, but encoding this constraint requires non-local information.

Some variations to the linear chain graphical structure have been proposed in order to utilise non-local information. Skip-chain CRFs were introduced in Sutton and McCallum (2004) which maintain the underlying CRF sequence model while adding additional skip edges between non-adjacent nodes which need to influence one another. This architecture yields a CRF graph structure exactly as shown in Figure 8.7. Some problems with this approach include determining which nodes to join together with these skip edges, and that loopy belief propagation (Pearl, 1982) is needed for approximate learning and inference since the CRF structure is no longer a tree. Finkel et al. (2005) propose a way of incorporating non-local information into factored probabilistic sequence models with approximate inference via Gibbs sampling (Geman and Geman, 1984) and performing decoding via simulated annealing (Kirkpatrick et al., 1983; Černý, 1985). Another approach to modelling non-local information with CRFs was proposed in Krishnan and Manning (2006) where the output of a first-stage linear chain CRF trained on local-only information is available to a second-stage linear chain CRF. This
allows the second-stage CRF to use the labels produced by the first-stage CRF in features to help it learn label consistency. This approach has become known as “stacking” CRFs.

Apart from supervised learning, a number of well-performing NER systems have used semi-supervised learning. The term semi-supervised (or weakly-supervised) refers to using annotated training data in conjunction with unlabelled data to boost performance. Ando and Zhang (2005b) and Ando and Zhang (2005a) propose their structural learning framework which attempts to learn how to learn using unsupervised data. This is achieved by learning from thousands of automatically generated auxiliary classification problems on unlabelled data, and seeing what common predictive structures exist in the well-performing classifiers. Another notable use of semi-supervised learning in NER is the work presented in Suzuki et al. (2007) and Suzuki and Isozaki (2008). In this work, the authors provide an extension to the CRF model to directly incorporate semi-supervised data into the learning process with very compelling results, beating the prior state-of-the-art performance on the CoNLL 2003 English test set by 0.61 $F_1$.

8.1.5 External resources

Many different kinds of external resources have been used in NER systems to help provide some robustness against the unseen entity problem. One way external resources are used is by direct lookup (gazetteers). Another way is as input to an unsupervised learning process, to produce clusters or embeddings over words or phrases.

Gazetteers

Gazetteers are important for helping improve both the precision and recall of NER systems (Florian et al., 2003; Cohen and Sarawagi, 2004). However, gazetteers come at a cost: building and maintaining high-quality gazetteers is very time consuming. Many techniques have been proposed over the years to solve this problem by automatically extracting gazetteers from large amounts of text (Riloff and Jones, 1999; Etzioni et al., 2005). More recently, Wikipedia has become a target for automatic gazetteer extraction,
and a number of relatively successful techniques have been established (Toral and Muñoz, 2006; Kazama and Torisawa, 2007).

**Cluster-based representations**

A number of different clustering algorithms have been used in NER systems. The Clark clustering algorithm (Clark, 2000, 2003) clusters over the context distribution of the words immediately to the left and right of the current word. The similarity of words is measured by the similarity of their context distributions using KL divergence (Kullback and Leibler, 1951). This algorithm runs in an iterative manner and produces hard clusters, assigning each word to exactly one (non-hierarchical) cluster.

The Brown clustering algorithm (Brown et al., 1992; Liang, 2005) performs bottom-up agglomerative word clustering to produce a hierarchical clustering of words. This algorithm greedily merges clusters to maximise the mutual information of bigrams, making it a class-based bigram language model. While a hierarchical clustering algorithm has many advantages over a hard clustering algorithm, the main disadvantage of the Brown clustering algorithm is its runtime. A naïve implementation runs in $O(k^3)$, and an optimised implementation runs in $O(kw^2 + T)$, where $k$ is the number of unique words in the training data, $w$ is the number of desired clusters, and $T$ is the number of words in the training data. Even the optimised version of this algorithm is very slow for large training corpora. For example, Turian et al. (2010) cites over three days of computing time were required to induce 1000 Brown clusters over a heavily cleaned version of the Reuters 1 corpus.

Lin and Wu (2009) present a distributed clustering algorithm based on $k$-means clustering (MacQueen, 1967), producing phrase clusters as opposed to word clusters. Highly polysemous words are not handled well by word clustering algorithms which need to assign each word into a single cluster, as all senses of the word are conflated into a single node in the cluster space. These phrase clusters work in a similar fashion to multi-word gazetteers: given a sentence $t_1, \ldots, t_n$, if a token $n$-gram $t_p, \ldots, t_q$ appears
as a phrase cluster \( c \), then a feature is fired on tokens \( t_{p-1}, \ldots, t_{q+1} \) indicating that cluster \( c \) was observed. These phrase-based clusters yielded state-of-the-art performance on the English CoNLL 2003 NER dataset.

### Distributed representations

*Neural language models* are a class of distributed word representations which produce *word embeddings*. Neural models generally work by mapping each word type to a dense real-valued vector in a low-dimensional vector space and assigning probabilities to \( n \)-grams by processing their embeddings in a neural network. A large number of neural language models have been proposed (Bengio et al., 2003; Schwenk and Gauvain, 2002; Mnih and Hinton, 2007; Collobert and Weston, 2008). Turian et al. (2010) investigates the effectiveness of two neural language models for NER, finding that they perform well but not as well as cluster-based word representations.

There are also algorithms for computing word embeddings that do not rely on a language model. A popular example is the Canonical Correction Analysis (CCA) family of word embeddings (Dhillon et al., 2011, 2012). Neelakantan and Collins (2014) extend CCA word embeddings to produce phrase embeddings to improve the performance of biomedical NER.

Mikolov et al. (2013a) and Mikolov et al. (2013b) introduce a number of fast-to-train log-linear language models, the most successful of which was the *skip-gram model*. Passos et al. (2014) extend upon the skip-gram model to infuse lexicon information into the training process. Using these lexicon infused word embeddings, they achieve state-of-the-art performance on the CoNLL 2003 English NER dataset and on their OntoNotes 5 split (Section 8.1.2).
8.2 Features used in NER systems

Here we will briefly outline the most widely used features from NER systems over the years before using them later in our new NER system. In the descriptions below, we use the sequence $w$ to represent the tokens and the sequence $y$ to represent the NER tags assigned to the tokens. These features are from a large survey of NER systems, including Borthwick (1999), Curran and Clark (2003), Zhang and Johnson (2003), Miller et al. (2004), Finkel et al. (2005), Lin and Wu (2009), Ratinov and Roth (2009), and Turian et al. (2010).

8.2.1 Morphosyntactic features

These features utilise the morphological and syntactic nature of the current token.

**Token window around** $w_i$ This string feature is simply a particular token in the sentence. It is common to use the current token and its surrounding tokens within some window ($w_{i+\delta}$). A common window size is $\pm 2$.

**Affixes of** $w_i$ This string feature is the prefix and/or suffix of some length of the current token. Common length values are between 1 and 8.

**Shape of** $w_i$ This is a collapsed or uncollapsed word shape of the current token. For example, the token *Dream* might have the collapsed word shape $\AA\AA$, and similarly the token *£1234.95* might map to the word shape *£9.95*.

**Capitalisation pattern** This string feature is the concatenation of whether or not the first character of the tokens within some window around the current token are capitalised ($w_{i+\delta}^0$). A common window size is $\pm 2$.

8.2.2 Other current-token features

These features use information local to the current token only.
**POS tag** Earlier systems, especially those prior to Ratinov and Roth (2009), used POS tags within some window of the current token. Later systems have found that short-length Brown cluster paths (see next) provide equivalent or richer information than POS tags, and have not utilised POS tags at all.

**Brown cluster paths** These string features were initially introduced as NER features by Miller et al. (2004). As mentioned earlier, Brown clusters are hierarchical in nature, forming a binary tree of clusters. A Brown cluster path is the binary path taken from the root of the tree down to a node. Each feature value is a fixed-length prefix of the Brown cluster path for the current token. Ratinov and Roth (2009) observed that Brown cluster path prefixes of length 4 roughly equate to POS tags, even though there is only $2^4 = 16$ possible values in paths of length 4, much less than the 36 English PTB POS tag set.

**Clark cluster number** This string feature is the number of the Clark cluster that the current token appears in. Clark clusters are a hard clustering, meaning each token only appear in one cluster. This feature is not fired for tokens which do not appear in any cluster.

**Word embeddings** Turian et al. (2010) test the effectiveness of C&W (Collobert and Weston, 2008) and HLBL (Mnih and Hinton, 2009) embeddings in an NER system by using each dimension of the embedding as a separate feature whose weight is the value of the embedding in that dimension. Before use, they scale the values of all embeddings so that they have a standard deviation of 0.1. Passos et al. (2014) use skip-gram embeddings as well as their lexicon infused skip-gram embeddings in the same manner.

### 8.2.3 Contextual features

These features utilise information outside of just the current token.
The two previous predictions $y_{i-1}$ and $y_{i-2}$ These string features are to help model contextual decisions, and partially help with ambiguity resolution when classifying the current token. This feature is only possible to implement if feature extraction is tightly coupled with the learning process — some feature extraction cannot be done ahead of time.

Concatenation of $w_i$ and the previous prediction $y_{i-1}$ This string feature exists to help with contextual ambiguity and has the same implementation complexities as the previous feature.

Multi-word gazetteer match Gazetteers have been used in the vast majority of NER systems as they allow the classifier to easily adapt to unseen-but-known entities. For example, the set of demonyms is a mostly closed set, but it is unlikely that all members of the set will be observed during training. Ratinov and Roth (2009) describe an algorithm for extending gazetteer matches to be multi-word matches, allowing richer multi-token dictionaries to be used.

Extended prediction history This feature was first introduced in Curran and Clark (2003) where they use per-token “memory” feature which yields the tag most recently assigned to the current token within the current document. Ratinov and Roth (2009) extended this idea to include multi-token history and corpus-level history. History features are supported by Barrena et al. (2014) when they investigated a “one entity per discourse” (Gale et al., 1992) and “one entity per collocation” (Yarowsky, 1993) hypothesis in terms of NER and NEL, finding that making these assumptions normally helps improve the accuracy of NER systems.

Ratinov and Roth’s extended prediction history works as follows: each time a NE label is assigned to a token, keep track of how many times it was assigned that label within the context of the current document. When this token is seen later in the document, add as a feature each label it has previously been assigned within
the current document. These features are then weighted with the relative number of times this label was assigned compared to all of the labels it was assigned.

For example, if the token Australia appeared earlier in the document twice with the tag \textit{W-ORG} and three times with the tag \textit{W-LOC}, when this token is next seen, two features will be added: \textit{W-ORG} with feature weight $\frac{2}{5}$ and \textit{W-LOC} with feature weight $\frac{3}{5}$.

\section{State-of-the-art English NER}

Here we briefly describe the two state-of-the-art publicly available NER systems as well as state-of-the-art performance on the commonly used English NER datasets. The first system is the Stanford Named Entity Recogniser (Finkel et al., 2005) which is distributed as part of the CoreNLP suite of NLP tools. This tagger is implemented in Java, and uses a CRF with \textit{L-BFGS} (Nocedal and Wright, 1999) for numerical optimisation and the Viterbi algorithm for decoding. The second system is the Illinois Named Entity Tagger (Ratinov and Roth, 2009), which uses a regularised averaged perceptron (Freund and Schapire, 1999) and beam search for decoding. It is implemented in Java and utilises the \textit{LBJ} modelling language (Rizzolo and Roth, 2010) to implement its features. We will compare the performance of our system against both of these systems.

\subsection{CoNLL 2003}

As previously mentioned, the CoNLL 2003 English dataset has become the de facto standard for comparing the performance of English NER systems. Table 8.4 shows the progression of state-of-the-art performance on this dataset since its initial release in 2003 with the CoNLL shared task.

Florian et al. (2003) was the top-performing system from the shared task with a system combining the output of four different classifiers, each of which utilised morphosyntactic and gazetteer features. Ando and Zhang (2005a) and Suzuki and
Table 8.4: Progression of and current state-of-the-art reported $F_1$-score performance on the CoNLL 2003 English dataset.

<table>
<thead>
<tr>
<th>System</th>
<th>dev</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Florian et al. (2003)</td>
<td>93.87</td>
<td>88.78</td>
</tr>
<tr>
<td>Ando and Zhang (2005a)</td>
<td>—</td>
<td>89.31</td>
</tr>
<tr>
<td>Suzuki and Isozaki (2008)</td>
<td>94.48</td>
<td>89.92</td>
</tr>
<tr>
<td>Ratinov and Roth (2009)</td>
<td>93.50</td>
<td>90.57</td>
</tr>
<tr>
<td>Lin and Wu (2009)</td>
<td>—</td>
<td>90.90</td>
</tr>
<tr>
<td>Turian et al. (2010)</td>
<td>93.95</td>
<td>90.36</td>
</tr>
<tr>
<td>Passos et al. (2014)</td>
<td>94.46</td>
<td>90.90</td>
</tr>
</tbody>
</table>

Isozaki (2008) pushed the state-of-the-art upwards through two quite different semi-supervised learning techniques: learning how to learn via many unsupervised models, and the incorporation of unlabelled data directly into a CRF model. Ratinov and Roth (2009) again pushed the test set number upwards through the amalgamation of a number of existing ideas, as well as some new techniques for modelling non-local information. In the same year, Lin and Wu (2009) presented a new state-of-the-art using semi-supervised learning with phrase clusters induced over a giant corpus of 700 billion web tokens. Unfortunately this system is hard to replicate with the web corpus being proprietary. Recently, more cluster and embedding based features have helped increase the state-of-the-art. Turian et al. (2010) utilises both word cluster and embeddings to boost performance, whereas Passos et al. (2014) introduces phase embeddings to achieve their state-of-the-art performance.

8.3.2 OntoNotes

As discussed in Section 8.1.2, there has been very little consistency between publications on how the OntoNotes NER data has been used for training and evaluation. To add to the complexities of comparing performance on OntoNotes data, the NE annotations change between each OntoNotes release as annotations are corrected, adjusted, or
Table 8.5: Reported performance on the OntoNotes English NER Finkel and Manning (2009) splits and category down-mapping. The “Ver.” column indicates which OntoNotes version was used to produce the numbers. Numbers across versions are not directly comparable due to annotation adjustments made in each release.

<table>
<thead>
<tr>
<th>System</th>
<th>Ver.</th>
<th>ABC</th>
<th>CNN</th>
<th>MNB</th>
<th>NBC</th>
<th>PRI</th>
<th>VOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finkel and Manning (2009) vanilla</td>
<td>2</td>
<td>74.51</td>
<td>75.78</td>
<td>62.21</td>
<td>63.90</td>
<td>83.44</td>
<td>79.23</td>
</tr>
<tr>
<td>Finkel and Manning (2009) joint</td>
<td>2</td>
<td>74.91</td>
<td>78.70</td>
<td>66.49</td>
<td>67.96</td>
<td>86.34</td>
<td>88.18</td>
</tr>
<tr>
<td>Finkel and Manning (2010) vanilla</td>
<td>3</td>
<td>76.00</td>
<td></td>
<td>57.50</td>
<td>62.40</td>
<td>79.50</td>
<td>82.70</td>
</tr>
<tr>
<td>Finkel and Manning (2010) joint</td>
<td>3</td>
<td>77.80</td>
<td></td>
<td>67.00</td>
<td>65.70</td>
<td>86.20</td>
<td>87.10</td>
</tr>
<tr>
<td>Tkachenko and Simanovsky (2012)</td>
<td>4</td>
<td>76.76</td>
<td>81.40</td>
<td>71.52</td>
<td>67.41</td>
<td>83.72</td>
<td>87.12</td>
</tr>
</tbody>
</table>

We are aware of three publications which have used the broadcast news subcorpus splits presented by Finkel and Manning (2009), with the original 18 categories down-mapped to just 4. Unfortunately, all three publications report performance numbers using a different version of OntoNotes, so the numbers are not directly comparable. Nonetheless, these numbers are shown in Table 8.5. Deviating from the splits they proposed in the previous year, Finkel and Manning (2010) decided to use the CNN subcorpus as additional training data for other parts of the system. As such, they did not train a vanilla NER model on the CNN subcorpus.

As far as we are aware, Passos et al. (2014) is the only publication to use their OntoNotes splits, which they report were produced using the OntoNotes 5 release. They report a $F_1$ score of 80.81 on dev and 82.24, micro-averaged across all 18 categories. No further breakdown of results were reported, such as overall precision or recall, nor per category or per subcorpus values.
8.4 Building a NER system with document structure

To illustrate how document representation can help improve the performance of a NER system, we implemented a NER system with features which utilise the document structure provided by a DRF such as DOCREP. In this section, we outline the baseline features used, the document structure features used, and perform a comparative analysis of our system against the Illinois and Stanford taggers on the standard English NER evaluation datasets.

8.4.1 Implementation

Our NER system is implemented on top of the DOCREP framework, and is released as open source under the MIT licence.\(^6\) For learning, our system uses a linear chain CRF, backed by CRFsuite (Okazaki, 2007). We used L-BFGS for numerical optimisation with L2 regularisation, 10 histories for approximating the Hessian, and the strong Wolfe conditions for line search backtracking (Nocedal and Wright, 1999). We experimented with a stacked linear chain setup as used by Krishnan and Manning (2006) and Ratinov and Roth (2009), implementing the token, entity, and super-entity majority features, but did not find any performance improvement over a single CRF.

Pre-processing

Our NER system has a small number of pre-processing steps that run over each document before that document is used for training or tagging. The first pre-processing step attempts to perform truecasing (Lita et al., 2003) on sentences which are seen as “all-caps”, using a simple set of heuristics which take into account in-document capitalisation frequencies, as well as frequencies from our large unlabelled corpus which is used by our unlabelled data features (see later).

\(^6\)https://github.com/schwa-lab/libschwa
// Construct capitalisation distribution counts for each token in the document.
for (auto &sentence : doc.sentences) {
    // Don’t include sentences which are all-caps.
    if (sentence.is_all_caps())
        continue;
    for (auto &token : sentence.span) {
        // Don’t trust the capitalisation of the first token in the sentence, unless
        // it is all-caps (e.g. an acronym).
        const UnicodeString u = UnicodeString::from_utf8(token.raw);
        if (!(token.starts_sentence() && !u.is_upper()))
            capitalisation_counts[u.to_lower()][token.raw] += 1;
    }
}

Figure 8.8: Constructing per-document token frequency counts is trivial when the
system uses a document model, such as what DOCREP provides.

Since our NER system was built from the ground up using DOCREP, all parts of the
NER pipeline can fully exploit the available document structure. Obtaining and utilising
in-document token frequency counts was trivial because of the underlying document
model. Figure 8.8 shows a code snippet for this process. Collating over the tokens
which only appear within the span of sentences which are not all-caps is trivial when
the data model natively represents all of this information. When the NER system has
finished using the current document, this capitalisation_counts mapping structure
can easily be erased and rebuilt when the next document is processed.

The other pre-processing steps involve normalising all digits and ordinals to 9 and
9th respectively. This is to help reduce the sparsity of numerical quantities in the under-
lying model, and helps provide some sense of equivalence between all numerical values
relative to a given context. This normalisation process is used in a number of other
NER systems, including the Illinois tagger (Ratinov and Roth, 2009) and Turian et al.
(2010). The Stanford tagger (Finkel et al., 2005) can also use this normalisation process
when using word clusters which have been generated with the same normalisation.

Figure 8.9 shows snippets from the Token DOCREP model used in our NER system.
This normalised value is stored as a member variable on the class and is not declared on
the DOCREP schema. If our NER system was using UIMA, storing this non-serialised
attribute on the model would not be straightforward. This figure shows other non-
8.4. Building a NER system with document structure

```cpp
class Token : public dr::Ann {
public:
    std::string raw; // From a read-in docrep model.
    std::string norm; // From a read-in docrep model.
    std::string pos; // From a read-in docrep model.
...
    Sentence *sentence; // Projected here via a "reverse slices" decorator.
    NamedEntity *ne; // Projected here via a "sequence tagger" decorator.
    std::string ne_label; // Projected here via a "sequence tagger" decorator.
    std::string ne_normalised; // The NER system's normalised form of the token.
...
    bool starts_sentence(void) const { return this == sentence->span.start; }
    bool ends_sentence(void) const { return this == sentence->span.stop - 1; }
};
```

Figure 8.9: A snippet from the DOCREP Token class used in our NER system. The first three members are loaded from a read-in DOCREP document. The next three members are derived at runtime after deserialisation via decorators. The last four members are application-defined and not on the DOCREP schema.

Serialised attributes on our Token model, each of which are projected onto Token instances after deserialisation via DOCREP decorators (Section 4.4.3). These projected attributes in conjunction with the design of the DOCREP C++ API make many common operations simple to perform, such as checking whether a token instance starts or ends a sentence. With a pointer to the Sentence object projected onto the tokens it spans, this check can reduces to an efficient and simple pointer comparison.

**Morphosyntactic features**

Here we describe the morphosyntactic features of our NER system. In these descriptions, the notation $w_i$ indicates the surface form of the current token after the pre-processing steps listed above. All morphosyntactic features have a weight of 1.

$w_{i+\delta}, \forall \delta \in [-2, 2]$ This string feature is the current token and its surrounding tokens in a window of ±2. We use a special sentinel value for tokens which fall off either end of the sentence when performing the windowing. The check for this is achieved via simple pointer arithmetic using our DOCREP Token model.
prefix($w_i$, l), $\forall l \in [2, 5]$ This string feature is the prefixes of length 2 to 5 (inclusive) of the current token.

suffix($w_i$, l), $\forall l \in [2, 5]$ This string feature is the suffixes of length 2 to 5 (inclusive) of the current token.

word_shape($w_i$) This is a collapsed word shape of the current token. For the shape of each Unicode code point, we use its Unicode category name. If two adjacent code points have the same Unicode category name, the category name is only added to shape representation once, as the “collapsed” part of the name implies. For example, the token Dream has the collapsed word shape LuLl, and the token £1234.95 has ScNdPoNd.\(^7\)

has_digit($w_i$) This Boolean feature indicates whether or not the current token contains a Unicode digit.

has_hyphen($w_i$) This Boolean feature indicates whether or not the current token contains a Unicode dash or hyphen.

has_upper($w_i$) This Boolean feature indicates whether or not the current token contains a Unicode uppercase code point.

is_acronym($w_i$) This Boolean feature indicates whether the current token looks like an acronym.

is_roman_numeral($w_i$) This Boolean feature indicates whether the current token looks like a roman numeral.

$\bigcup_{\delta=-1}^{\delta=1} \text{unicode_category} \left( w_{i+\delta}^0 \right)$ This string feature is the concatenation of the Unicode category of the first code point of the tokens in a window of $\pm 1$ around the current token. This is often referred to as a “capitalisation pattern”.

\(^7\)Lu is an uppercase letter, Ll is a lowercase letter, Sc is a currency symbol, Nd is a decimal digit, and Po is other punctuation. See http://www.unicode.org/reports/tr44/.
\[ \bigcup_{\delta=-2}^{2} \text{unicode\textunderscore category} \left( w^0_{i+\delta} \right) \] This is the same as the previous feature, except in a window of \( \pm 2 \).

**Other current-token features**

Here we describe the features in our NER system which are per-token but are not directly morphosyntactic in nature. All of these features utilise external information. Unless otherwise stated, these per-token features have a weight of 1.

**brown\textunderscore cluster\textunderscore path** \((w_i, l), \forall l \in \{4, 6, 10, 20\} \) This string feature is the Brown cluster path of length \( l \) for the current token. We use the `english-wikitext.c1000` clusters distributed with the Illinois tagger, with a minimum frequency threshold of 5. The path lengths originally came from Ratinov and Roth (2009) and have been used in other NER systems (Turian et al., 2010; Passos et al., 2014).

**clark\textunderscore cluster** \((w_i) \) This string feature is the Clark cluster number for the current token. We could not find any commonly used set of Clark clusters so we generated 200 Clark clusters from the Reuters 1 corpus using Clark’s original code.\(^8\) All digits and ordinals received the same pre-processing steps as used in our NER system.

**word\textunderscore embedding** \((w_i) \) The word embedding features are string features with a non-1 weighting. The string value is the dimension of the embedding, and the feature weight is the embedding value scaled to have a standard deviation (\( \sigma \)) of 0.1 as per Turian et al. (2010). We use the 50-dim-unscaled HLBL embeddings (Mnih and Hinton, 2009) produced and used by Turian et al., which are available online.\(^9\) We also tried the C&W embeddings (Collobert and Weston, 2008) as per Turian et al., but found that the HLBL embeddings performed better.

\(^8\)http://www.cs.rhul.ac.uk/home/alexc/
\(^9\)http://metaoptimize.com/projects/wordreps/
Figure 8.10: A snippet of top-level block structure as represented in our NER system.

**Contextual features**

Here we describe the features which operate over more context than just the current token. All contextual features have a weight of 1.

**Multi-word gazetteer match** We use the multi-word gazetteer matching algorithm and feature generation procedure used in the Illinois tagger (Ratinov and Roth, 2009). We use the 79 gazetteers distributed with the Illinois tagger, as of the 2.8.2 release. Like the normalised version of the token, we cache the multi-word gazetteer matches directly on Token objects. This cache was not included in Figure 8.9 for brevity.

**Extended prediction history** We implement the extended prediction history feature as described in Ratinov and Roth (2009), except we restrict its memory to the current document instead of the previous 1000 tokens. Per-document history is easy to implement with a DOCREP model and document-oriented system — when the next document is received, clear the existing history memory and start again. The history stored per token is the label the classifier assigned to the token, sequence tag encoding included (that is, \( w-LOC \) instead of \( LOC \)).

**Document-level features**

Here we describe the features which attempt to utilise document-level information, such as document structure. These features are novel and unique to our NER system. All document-level features have a weight of 1.

```cpp
class Block : public dr::Ann {
public:
    dr::Pointer<Heading> heading;
    dr::Pointer<Paragraph> paragraph;
...};
```
8.4. Building a NER system with document structure

```cpp
// Does the document have block structure information?
if (doc.blocks.empty()) {
    // If not, iterate through each sentence in turn.
    for (auto &sentence : doc.sentences)
        extract_sentence(sentence);
} else {
    // Iterate through the paragraphs first.
    for (auto &block : doc.blocks)
        if (block.paragraph != nullptr)
            for (auto &sentence : block.paragraph->span)
                extract_sentence(sentence);
    // Iterate through the non paragraphs last.
    for (auto &block : doc.blocks)
        if (block.heading != nullptr)
            extract_sentence(*block.heading->sentence);
}
```

Figure 8.11: A snippet illustrating how richer document structure is utilised in our NER system if it is available.

<table>
<thead>
<tr>
<th>Soccer NN I-NP O</th>
<th>Rotor NNP I-NP I-ORG</th>
</tr>
</thead>
<tbody>
<tr>
<td>RotoN NN I-NP I-ORG</td>
<td>Volgograd NNP I-NP I-ORG</td>
</tr>
<tr>
<td>FANS VBZ I-VP O</td>
<td>must MD I-VP O</td>
</tr>
<tr>
<td>Locked NNP I-NP O</td>
<td>play VB I-VP O</td>
</tr>
<tr>
<td>Out NNP I-NP O</td>
<td>...</td>
</tr>
<tr>
<td>After NNP I-NP O</td>
<td>that WDT B-NP O</td>
</tr>
<tr>
<td>Volgograd NNP I-NP I-LOC</td>
<td>ended VBD I-VP O</td>
</tr>
<tr>
<td>Violence NNP I-NP O</td>
<td>Rotor NNP I-NP I-ORG</td>
</tr>
<tr>
<td>...</td>
<td>'s POS B-NP O</td>
</tr>
<tr>
<td>Moscow RB I-ADVP I-LOC</td>
<td>brief JJ I-NP O</td>
</tr>
<tr>
<td>1996-08-30 CD I-NP O</td>
<td>spell NN I-NP O</td>
</tr>
<tr>
<td>...</td>
<td>as IN I-SBAR O</td>
</tr>
<tr>
<td>...</td>
<td>league NN I-NP O</td>
</tr>
<tr>
<td>...</td>
<td>leaders NNS I-NP O</td>
</tr>
</tbody>
</table>

Figure 8.12: The document heading and a snippet from the first sentence from a document in the CoNLL 2003 English NER development set. The first sentence in the body helps disambiguate the entity bounds in the heading.
Block-ordered versus Sentence-ordered iteration The document structure information provided by DOCREP is utilised in our NER system in the form of blocks. A code snippet from our DOCREP Block model is shown Figure 8.10. A document consists of multiple consecutive Blocks, and a block is either a paragraph, or a heading, or a list, etc.

If an incoming document has this block structure, instead of iterating through each sentence in order as they appear in the document, our NER system will iterate through the sentences in paragraphs before non-paragraph sentences. This logic is outlined in the code snippet in Figure 8.11 and comes from the intuition that entities in headings, especially newspaper headlines, are often hard to classify without first reading the document. Figure 8.12 shows an example of this situation, where by the heading becomes less ambiguous by reading just the first sentence of the document.

This alternate iteration order affects the behaviour of the extended prediction history feature.

The CoNLL 2003 NER data contains limited document structure information if you align the datasets back to their original documents in the Reuters 1 corpus. The OntoNotes 5 data does not provide any form of document structure. Unfortunately, there is currently no suitable NER corpus for us to explore richer document-level features in depth. If such a corpus exists in the future, there are a number of other document-level features we would like to explore. These include:

Stylistic information This involves utilising any available stylistic information about the tokens. For example, in some domains, tokens which have been emphasised (e.g. italicising or change in font weight) are often named entities, and the bounds of the stylised text may indicate entity boundaries.

Utilising hyperlinks Many rich-media documents contain hyperlinks. There are a number of ways information provided by hyperlinks could be incorporated
8.4. Building a NER system with document structure

into NER features. For example, the endpoint of the hyperlink could provide entity type disambiguation information via the genre of the targeted web page or domain.

**Utilising target names** Many rich-media documents contain images, especially news articles. Journalists publishing these articles often name associated images in a way that relates to the prominent named entities within the document. This kind of information could be useful for entity bounds disambiguation. Additionally, the end of hyperlink paths might provide additional disambiguation. For example, if a hyperlink was pointing to a Wikipedia article and the article happened to contain a disambiguation phrase (e.g. *Mercury (element)* vs. *Mercury (planet)* vs. *Mercury (mythology)*), this disambiguation phase could aid NER entity category disambiguation.

8.4.2 Results and evaluation

We train and test our system and document-level features on the standard CoNLL 2003 English dataset, as well as on two different OntoNotes 5 splits. For the CoNLL 2003 dataset, we compare against the numbers reported in the literature. For the OntoNotes splits, models were also trained using the Stanford and Illinois taggers as there were no existing directly comparable numbers reported in the literature.

Unless otherwise stated, all experiments used BMEWO sequence tag encoding. For each dataset, we tuned the L2 regularisation value on the development set.

**CoNLL 2003**

In order to evaluate the performance of our new NER system, we first trained a model without the document structure features enabled so that we can assess the performance of those features in isolation. Tuning the L2 regularisation parameter on the development set yielded an optimal value of 0.6. The $F_1$ score on the development and test sets
Table 8.6: Document structure features allow us to achieve state-of-the-art performance on the CoNLL 2003 English NER evaluation.

<table>
<thead>
<tr>
<th>System</th>
<th>dev</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Florian et al. (2003)</td>
<td>93.87</td>
<td>88.78</td>
</tr>
<tr>
<td>Ando and Zhang (2005a)</td>
<td>—</td>
<td>89.31</td>
</tr>
<tr>
<td>Suzuki and Isozaki (2008)</td>
<td>94.48</td>
<td>89.92</td>
</tr>
<tr>
<td>Ratinov and Roth (2009)</td>
<td>93.50</td>
<td>90.57</td>
</tr>
<tr>
<td>Lin and Wu (2009)</td>
<td>—</td>
<td>90.90</td>
</tr>
<tr>
<td>Turian et al. (2010)</td>
<td>93.95</td>
<td>90.36</td>
</tr>
<tr>
<td>Passos et al. (2014)</td>
<td>94.46</td>
<td>90.90</td>
</tr>
<tr>
<td>DOCREPregular</td>
<td>94.16</td>
<td>90.45</td>
</tr>
<tr>
<td>DOCREPdoc-aware</td>
<td><strong>94.51</strong></td>
<td><strong>91.08</strong></td>
</tr>
</tbody>
</table>

can be seen in the second-last row of Table 8.6. The performance of our system without document structure features is close to state-of-the-art. This is not overly surprising considering we brought together techniques and features from a number of different high-performing NER systems.

In order to assess how well our document structure features work, we need a corpus which has document structure. Unfortunately the CoNLL 2003 corpus does not contain any document structure outside of sentence and document boundaries. However, the data for the CoNLL 2003 shared task is a subset of the Reuters 1 corpus, which does contain heading and dateline information for each document. We aligned all of the English CoNLL 2003 data to its source document in the Reuters 1 corpus so that we know, for each sentence, whether it is in a heading, a dateline, or in the body text of an article. The document structure present here is minimal, but it is enough to demonstrate that these document structure features are worth pursuing.

Once this document structure information was projected onto each sentence in the CoNLL 2003 English dataset, we trained and evaluated a model in the same way as before. Tuning the L2 regularisation parameter yielded an optimal value of 0.4. The result of our NER system with document structure features enabled is shown
8.4. Building a NER system with document structure

<table>
<thead>
<tr>
<th></th>
<th>regular</th>
<th>doc-aware</th>
<th>ΔF₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>96.86</td>
<td>96.40</td>
<td>+0.02</td>
</tr>
<tr>
<td>MISC</td>
<td>91.09</td>
<td>91.66</td>
<td>+0.55</td>
</tr>
<tr>
<td>ORG</td>
<td>89.74</td>
<td>89.83</td>
<td>+1.09</td>
</tr>
<tr>
<td>PER</td>
<td>97.27</td>
<td>97.85</td>
<td>+0.07</td>
</tr>
<tr>
<td>Overall</td>
<td>94.48</td>
<td>94.59</td>
<td>+0.35</td>
</tr>
</tbody>
</table>

Table 8.7: Per-category precision, recall, and F₁-score breakdown for our system on the CoNLL 2003 English NER evaluation development dataset. These percentages are taken from the conlleval evaluation script.

<table>
<thead>
<tr>
<th></th>
<th>regular</th>
<th>doc-aware</th>
<th>ΔF₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>92.24</td>
<td>92.80</td>
<td>+0.64</td>
</tr>
<tr>
<td>MISC</td>
<td>81.62</td>
<td>83.24</td>
<td>+1.38</td>
</tr>
<tr>
<td>ORG</td>
<td>86.47</td>
<td>86.90</td>
<td>+0.48</td>
</tr>
<tr>
<td>PER</td>
<td>96.81</td>
<td>97.07</td>
<td>+0.41</td>
</tr>
<tr>
<td>Overall</td>
<td>90.49</td>
<td>91.06</td>
<td>+0.63</td>
</tr>
</tbody>
</table>

Table 8.8: Per-category precision, recall, and F₁-score breakdown for our system on the CoNLL 2003 English NER evaluation test dataset. These percentages are taken from the conlleval evaluation script.

in the last row of Table 8.6. Enabling the document structure features boosts F₁ by 0.35 on the development set and 0.63 on the test set, producing a new state-of-the-art number for this dataset. The per-class breakdowns of our results with and without document structure features are shown in Tables 8.7 and 8.8. These tables show that the per-category F₁ increases for all categories on both datasets, but the per-category gain from enabling document structure features is not consistent across datasets.
OntoNotes

In order to compare our system against the Illinois tagger and the Stanford tagger, we needed to train new models for both. There were two reasons for this. First, of the two of these taggers, only the Illinois tagger provides an 18 category OntoNotes model. However, it is not stated what subset of OntoNotes this model is trained on. Secondly, due to the lack of consistency in data splits, we trained on all known splits to provide the maximum comparability between our system and other systems.

For training the Illinois tagger, we downloaded the latest version at the time (version 2.8.2). We based our configuration file on their provided OntoNotes configuration file, modified as necessary to suit the needs of the training split in question (e.g. using only four categories instead of 18). For training the Stanford tagger, we downloaded the latest release at the time (release 2015-01-30). The codebase does not come with a recommended properties file for a corpus like OntoNotes, nor do they distribute their distributional similarity clusters which are used for their “distsim” features. For training the Stanford tagger, we used their CoNLL 2003 properties file as the starting point. For the distributional similarity clusters, we generated 200 Clark clusters (Clark, 2003) from the Reuters 1 corpus using Clark’s original code.\textsuperscript{10}

Unfortunately, the OntoNotes data does not have document structure metadata outside of basic sentence and token information, so we are not able to fully utilise our document structure features. Nonetheless, we evaluate our system against the Illinois and Stanford Named Entity taggers on various OntoNotes splits.

The first set of splits we compare on is the Finkel and Manning (2009) splits, which use only four of the broadcast news subcorpora and map the 18 NE categories down to just four. The results are shown in Table 8.9. The results reported here for the Stanford tagger are not comparable to the reported results in Finkel and Manning (2009) nor in Finkel and Manning (2010) for a number of reasons. First, the version of OntoNotes is different, and second, the exact training configuration used for these experiments is

\textsuperscript{10}http://www.cs.rhul.ac.uk/home/alexc/
Table 8.9: Performance on the OntoNotes 5 English NER Finkel and Manning (2009) splits and category down-mapping.

<table>
<thead>
<tr>
<th>System</th>
<th>ABC</th>
<th>CNN</th>
<th>MNB</th>
<th>NBC</th>
<th>PRI</th>
<th>VOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illinois 2.8.2</td>
<td>72.95</td>
<td>77.67</td>
<td>73.28</td>
<td>64.63</td>
<td>83.96</td>
<td>83.73</td>
</tr>
<tr>
<td>Stanford 2015-01-30</td>
<td>69.57</td>
<td>72.90</td>
<td>63.10</td>
<td>62.72</td>
<td>81.97</td>
<td>84.27</td>
</tr>
<tr>
<td><strong>DOCREP</strong></td>
<td><strong>74.91</strong></td>
<td><strong>79.45</strong></td>
<td><strong>74.52</strong></td>
<td><strong>70.59</strong></td>
<td><strong>87.62</strong></td>
<td><strong>86.55</strong></td>
</tr>
</tbody>
</table>

Table 8.10: Performance on the OntoNotes 5 English NER Passos et al. (2014) splits.

<table>
<thead>
<tr>
<th>System</th>
<th>dev</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passos et al. (2014)</td>
<td>80.81</td>
<td>82.24</td>
</tr>
<tr>
<td>Illinois 2.8.2</td>
<td>82.32</td>
<td>84.00</td>
</tr>
<tr>
<td>Stanford 2015-01-30</td>
<td>81.93</td>
<td>84.51</td>
</tr>
<tr>
<td><strong>DOCREP</strong></td>
<td><strong>84.12</strong></td>
<td><strong>85.98</strong></td>
</tr>
</tbody>
</table>

unknown, including what distributional similarity clusters were used. The results in Table 8.9 show that our implementation of an amalgamation of state-of-the-art NER features yields top-performing results in all six broadcast news domains. The Stanford NER tagger performs noticeably worse on the first four broadcast news subcorpora than the other two systems. This might be due to the lack of external resources used by this system — this tagger might not be as robust to unseen tokens and contexts.

The second set of OntoNotes splits we compared the three NER systems on was the Passos et al. (2014) splits, the results of which can be seen in Table 8.10. Our system outperforms the reported state-of-the-art performance in Passos et al. (2014) by over 3.5% $F_1$, and also outperforms the Illinois by a significant margin.

8.5 Summary

In this chapter, we have shown demonstrated another very successful use case for a DRF such as DOCREP — native span representations and document structure features.
By combining work from the previous top-performing NER systems and then adding document structure features, we achieve state-of-the-art performance on two English NER datasets. On the de facto English NER dataset, CoNLL 2003, we achieved a $F_1$-score of 91.08% on the test set (testb). On the Passos et al. (2014) OntoNotes 5 split, using 18 NE categories, we beat the previous reported state of the art by 3.74% $F_1$. On the other evaluated datasets which do not have directly comparable numbers in publications, we trained state-of-the-art NER taggers and compared our performance against theirs. In all instances, we outperformed the existing taggers on $F_1$.

It is rather unfortunate for our purposes that, to the best of our knowledge, there does not exist a NER training corpus with rich document structure. This could be in the form of web pages, Markdown, Microsoft Word documents, etc. Our initial experiments with document structure features on documents with minimal document structure yielded positive and encouraging results. In future work, we would like to create a NER training corpus which contains rich document structure in order to further progress the research into document structure features.
9 Conclusion

Experience has shown that stand-off annotations are a superior linguistic approach to annotation representation as they can represent arbitrarily nested and overlapping annotations, and they preserve any structure contained within the original document. Document representation frameworks (DRFs) have been developed to facilitate the use of stand-off annotations, but unfortunately, they have had disappointing uptake within the NLP community. There are a number of reasons for this, including usability issues, resource requirements, specific development workflows, and the fact they are not programming language agnostic. This thesis aims to solve this problem with our novel DRF, DOCREP (document representation). DOCREP is designed to be efficient, elegant, expressive, programming language and environment agnostic, and most importantly, easy to install, learn, and use.

In Chapter 2 we presented existing approaches to document representation, describing existing annotation formats and DRFs. We covered existing sets of design criteria and proposals for the representation of linguistic annotations and discussed how they have been adhered to by existing linguistic annotation standards and formalisms. While covering existing DRF implementations, we pointed out usability issues in each and outlined why these factors have contributed to the disappointing uptake of DRFs by NLP researchers. We concluded that field was lacking a lightweight, efficient, elegant, and modern DRF that is programming language agnostic, easy to learn, and minimalist in design.
Chapter 3 went on to introduce DOCREP, our newly designed and implemented DRF. This chapter outlined the design requirements for DOCREP, relating these back to the existing criteria from the literature. Our design requirements were made explicit through the identification of use cases that existing DRFs fail to satisfy. We concluded that our design requirements were similar to those in the literature, except we included additional pragmatic requirements that should encourage greater use of DRFs.

In Chapter 4, we described the DOCREP runtime data model, the serialisation protocol, and aspects of the DOCREP runtime that are common across all APIs. We also described our implementations of DOCREP in Python, C++, and Java, the main programming languages used within the NLP community. This chapter went into enough technical detail to support the implementation of a DOCREP API in other programming languages. The data model, serialisation format, and each of our implementations were evaluated against our design requirements presented in Chapter 3.

In Chapter 5, we showed how a diverse multilayered corpus, in particular the OntoNotes 5 corpus, can be represented in a DRF. We outlined existing corpus distribution strategies and highlighted usability and quality assurance issues with them. This chapter also presented the DOCREP model definitions for various annotation layers in the OntoNotes 5 corpus, demonstrating how a wide range of linguistic phenomena can be modelled in DOCREP. We converted this corpus into both DOCREP and UIMA annotations, and showed that DOCREP performed this conversion up to 34 times faster than UIMA and required up to 9 times less space to serialise the same annotations. We also discussed the use of a DRF for corpus distribution, demonstrating that they aid in reproducibility of experiments as well as for the overall quality assurance of the annotations. This chapter showed that DOCREP satisfies our efficiency and modelling design requirements.

Chapter 6 presented an evaluation of DOCREP from the perspective of a user, showing that DOCREP meets our lightweight, programming language agnostic, and ease to use design requirements. This chapter went though a number of use cases encoun-
tered when working with text corpora, comparing how operations are performed with traditional UNIX tools and their DOCREP counterparts. This chapter also highlights the advantages that a streamable serialisation format provide to DOCREP. Chapter 6 concluded with testimonials from DOCREP users from within our research lab and from the LT application development community, demonstrating that DOCREP satisfies our ease of use design requirement.

Chapter 7 presented our novel document-aware tokenization framework. This framework maintains byte and Unicode code point offsets back into the original document during document structure interpretation, input transcoding, tokenization, and SBD. The token, sentence boundary, and document structure annotations produced by this framework are yielded as DOCREP annotations. This framework allows downstream applications to exploit document structure typically destroyed during the initial stage of most NLP pipelines. If all components in a pipeline use DOCREP, our lazy serialisation and underspecified type system allows these annotation layers to be propagated through the pipeline even if the intermediate consumers and producers are unaware of these annotations. In the process, we have also contributed a high-quality English tokenizer and sentence boundary detection tool producing DOCREP for text, SGML, and HTML documents.

Chapter 8 demonstrated that downstream NLP applications can benefit from document structure information. We implemented a new NER system with document-level features. These document-level features allowed us to achieve state-of-the-art results on multiple NER datasets, including the canonical CoNLL 2003 English NER dataset where our system achieves 94.51% $F_1$ on the development set and 91.08% $F_1$ on the test set. Our NER system also beat the previous reported state of the art by 3.74% $F_1$ on the OntoNotes 5 test set. As soon as document structured NER datasets are available, we expect substantial further performance gains by document-aware systems.
9.1 Future Work

There are four broad categories for future work: evangelising the use of DOCREP, improving the DOCREP ecosystem, improving document format support for the tokenization framework, and constructing a NER corpus on which document-level features can be more thoroughly exploited.

Improving the reproducibility of results, interoperability and substitutability of components, and overall quality assurance of NLP systems and corpora can only be achieved through action from the whole community. This thesis contributes the tools and methodology to support such a change. For change to happen, NLP systems and corpora need to adopt DOCREP or a similar technology. As such, future work includes the evangelising of DOCREP within the community, and facilitating its integration into existing tools, pipelines, and corpus distribution techniques. Providing off-the-shelf DOCREP producers and consumers for commonly used NLP tools is one way this process can be aided. For example, providing wrappers like the DKPro Core UIMA wrappers for components of the CoreNLP stack.\(^1\)

There a number of ways the DOCREP command-line tools and APIs can be improved. For example, providing tools to facilitate the deletion of annotation layers, the merging of annotations from different streams, or the renaming of fields would be useful tools. Another useful improvement is a convenient way for the runtime schema renaming to map between camelCase and underscore_case naming conventions for all annotation types and attributes. This would simplify the idiomatic interoperability between languages.

Another significant area for improvement in the DOCREP ecosystem is the visualisation and editing of annotations. It would be beneficial to provide a richer visualisation experience beyond \texttt{dr less} — visualising a each node independently in a parse tree is quite different to visualising the overall tree structure. A lightweight, interactive

\(^1\)https://code.google.com/p/dkpro-core-asl/wiki/StanfordCoreComponents
web-based visualisation tool for overlaying annotations and annotation layers onto the original document would satisfy this use case. Since DOCREP annotations are self-describing, little back-end boilerplate code is required to establish such a tool.

For the tokenization framework presented in Chapter 7, greater support for document formats will help with its future uptake. Some notable formats missing from our supported set include include MediaWiki and Markdown markup. Supporting MediaWiki would allow the ingestion of Wikipedia articles directly into DOCREP; a boon for computational linguists who work with Wikipedia.

We concluded in Chapter 8 that it was unfortunate for our purposes that, to the best of our knowledge, there does not exist a large NER corpus with rich document structure. To further explore these document-level features, such a corpus needs to be created. Existing corpora of HTML pages, Wikipedia articles, or \LaTeX scientific reports are potential candidates for such an annotation task.

9.2 Summary

We have shown through the design, implementation, and evaluation of a new document representation framework, DOCREP, that DRFs can be lightweight, efficient, programming language agnostic, elegant, and easy to use. We went on to show that by providing document structure from the beginning of the NLP pipeline, downstream applications can exploit this information. Our initial investigation into the use of encoding document structure into a NER system yielded state-of-the-art results. Adoption of DOCREP throughout the NLP community will assist in the reproducibility of results, substitututability of components, and overall quality assurance of NLP systems and corpora, all of which are problematic areas within NLP research and applications. Above all, it will make developing and combining NLP components into applications faster, more efficient, and more reliable.
Appendices
## A NER datasets: CoNLL 2003

### A.1 Category distribution

<table>
<thead>
<tr>
<th></th>
<th>train</th>
<th>dev</th>
<th>test</th>
<th>Σ</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>7140</td>
<td>1837</td>
<td>1668</td>
<td>10 645</td>
<td>30.3%</td>
</tr>
<tr>
<td>MISC</td>
<td>3438</td>
<td>922</td>
<td>702</td>
<td>5062</td>
<td>14.4%</td>
</tr>
<tr>
<td>ORG</td>
<td>6321</td>
<td>1341</td>
<td>1661</td>
<td>9323</td>
<td>26.6%</td>
</tr>
<tr>
<td>PER</td>
<td>6600</td>
<td>1842</td>
<td>1617</td>
<td>10 059</td>
<td>28.7%</td>
</tr>
<tr>
<td>Σ</td>
<td>23 499</td>
<td>5942</td>
<td>5648</td>
<td>35 089</td>
<td>100.0%</td>
</tr>
<tr>
<td>%</td>
<td>67.0%</td>
<td>16.9%</td>
<td>16.1%</td>
<td>35 089</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Table A.1: NE category distribution across the CoNLL 2003 English splits.
Appendix A. NER datasets: CoNLL 2003

A.2 Sentence boundary errors

It is well known that the CoNLL 2003 English NER test set is a lot harder than the development set, with the documents frequently discussing names of sports teams whose names are locations. Some of the lesser known issues include tokenization mistakes and many sentence boundary errors arising from the tokenization and SBD not being gold standard. In order to quantify this observation, we went and manually corrected the sentence boundaries for all three split files, without changing the tokenization or document boundaries. The change in the number of sentences and named entities due to this correction process can be seen in Table A.2. The number of NEs changes slightly as some NEs are erroneously split across sentences in the original data, so correcting the sentence boundary split merges two NEs back into one. We did not use this altered data for any reported experiments for consistency with existing results, but present these numbers just to highlight one of the many issues with this dataset being the de facto canonical dataset for English NER evaluation.

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Corrected</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>train</td>
<td>sentences</td>
<td>14,041</td>
<td>13,226</td>
</tr>
<tr>
<td></td>
<td>NEs</td>
<td>23,499</td>
<td>23,431</td>
</tr>
<tr>
<td>dev</td>
<td>sentences</td>
<td>3,250</td>
<td>3,145</td>
</tr>
<tr>
<td></td>
<td>NEs</td>
<td>5,942</td>
<td>5,929</td>
</tr>
<tr>
<td>test</td>
<td>sentences</td>
<td>3,453</td>
<td>3,328</td>
</tr>
<tr>
<td></td>
<td>NEs</td>
<td>5,648</td>
<td>5,623</td>
</tr>
</tbody>
</table>

Table A.2: The number sentences and named entities in the CoNLL 2003 English dataset splits before and after manual sentence boundary correction.
B  NER datasets: OntoNotes 5

B.1  Category distribution for the official splits

<table>
<thead>
<tr>
<th>Category</th>
<th>train</th>
<th>dev</th>
<th>test</th>
<th>Σ</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARDINAL</td>
<td>10908</td>
<td>1724</td>
<td>1006</td>
<td>13638</td>
<td>8.4%</td>
</tr>
<tr>
<td>DATE</td>
<td>18807</td>
<td>3211</td>
<td>1789</td>
<td>23807</td>
<td>14.7%</td>
</tr>
<tr>
<td>EVENT</td>
<td>1009</td>
<td>179</td>
<td>85</td>
<td>1273</td>
<td>0.8%</td>
</tr>
<tr>
<td>FAC</td>
<td>1158</td>
<td>133</td>
<td>149</td>
<td>1440</td>
<td>0.9%</td>
</tr>
<tr>
<td>GPE</td>
<td>21944</td>
<td>3651</td>
<td>2547</td>
<td>28142</td>
<td>17.4%</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>358</td>
<td>35</td>
<td>22</td>
<td>415</td>
<td>0.3%</td>
</tr>
<tr>
<td>LAW</td>
<td>459</td>
<td>65</td>
<td>44</td>
<td>568</td>
<td>0.4%</td>
</tr>
<tr>
<td>LOC</td>
<td>2161</td>
<td>316</td>
<td>215</td>
<td>2692</td>
<td>1.7%</td>
</tr>
<tr>
<td>MONEY</td>
<td>5220</td>
<td>854</td>
<td>355</td>
<td>6429</td>
<td>4.0%</td>
</tr>
<tr>
<td>NORP</td>
<td>9341</td>
<td>1278</td>
<td>991</td>
<td>11610</td>
<td>7.2%</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>2196</td>
<td>335</td>
<td>207</td>
<td>2738</td>
<td>1.7%</td>
</tr>
<tr>
<td>ORG</td>
<td>24163</td>
<td>3798</td>
<td>2002</td>
<td>29963</td>
<td>18.5%</td>
</tr>
<tr>
<td>PERCENT</td>
<td>3802</td>
<td>656</td>
<td>408</td>
<td>4866</td>
<td>3.0%</td>
</tr>
<tr>
<td>PERSON</td>
<td>22050</td>
<td>3164</td>
<td>2137</td>
<td>27351</td>
<td>16.9%</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>993</td>
<td>214</td>
<td>90</td>
<td>1297</td>
<td>0.8%</td>
</tr>
<tr>
<td>QUANTITY</td>
<td>1240</td>
<td>190</td>
<td>153</td>
<td>1583</td>
<td>1.0%</td>
</tr>
<tr>
<td>TIME</td>
<td>1704</td>
<td>361</td>
<td>225</td>
<td>2290</td>
<td>1.4%</td>
</tr>
<tr>
<td>WORK_OF_ART</td>
<td>1281</td>
<td>202</td>
<td>169</td>
<td>1652</td>
<td>1.0%</td>
</tr>
</tbody>
</table>

| Σ        | 128794 | 20366 | 12594 | 161754 | 100.0% |
| %        | 79.6%  | 12.6% | 7.8%  | 161754 | 100.0% |

Table B.1: NE category distribution across the OntoNotes 5 English NE-annotated documents, using the official splits.
### B.2 Category distribution for the Passos et al. (2014) splits

<table>
<thead>
<tr>
<th>Category</th>
<th>train</th>
<th>dev</th>
<th>test</th>
<th>Σ</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARDINAL</td>
<td>11733</td>
<td>862</td>
<td>1006</td>
<td>13601</td>
<td>8.4%</td>
</tr>
<tr>
<td>DATE</td>
<td>20567</td>
<td>1415</td>
<td>1789</td>
<td>23771</td>
<td>14.7%</td>
</tr>
<tr>
<td>EVENT</td>
<td>1043</td>
<td>144</td>
<td>85</td>
<td>1272</td>
<td>0.8%</td>
</tr>
<tr>
<td>FAC</td>
<td>1182</td>
<td>106</td>
<td>149</td>
<td>1437</td>
<td>0.9%</td>
</tr>
<tr>
<td>GPE</td>
<td>23427</td>
<td>2139</td>
<td>2547</td>
<td>28113</td>
<td>17.4%</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>356</td>
<td>32</td>
<td>22</td>
<td>410</td>
<td>0.3%</td>
</tr>
<tr>
<td>LAW</td>
<td>504</td>
<td>20</td>
<td>44</td>
<td>568</td>
<td>0.4%</td>
</tr>
<tr>
<td>LOC</td>
<td>2252</td>
<td>225</td>
<td>215</td>
<td>2692</td>
<td>1.7%</td>
</tr>
<tr>
<td>MONEY</td>
<td>5826</td>
<td>243</td>
<td>355</td>
<td>6424</td>
<td>4.0%</td>
</tr>
<tr>
<td>NORP</td>
<td>9715</td>
<td>892</td>
<td>991</td>
<td>11598</td>
<td>7.2%</td>
</tr>
<tr>
<td>ORDINAL</td>
<td>2286</td>
<td>240</td>
<td>207</td>
<td>2733</td>
<td>1.7%</td>
</tr>
<tr>
<td>ORG</td>
<td>26326</td>
<td>1622</td>
<td>2002</td>
<td>29950</td>
<td>18.5%</td>
</tr>
<tr>
<td>PERCENT</td>
<td>4281</td>
<td>176</td>
<td>408</td>
<td>4865</td>
<td>3.0%</td>
</tr>
<tr>
<td>PERSON</td>
<td>23138</td>
<td>1955</td>
<td>2129</td>
<td>27222</td>
<td>16.9%</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>1133</td>
<td>72</td>
<td>90</td>
<td>1295</td>
<td>0.8%</td>
</tr>
<tr>
<td>QUANTITY</td>
<td>1293</td>
<td>137</td>
<td>153</td>
<td>1583</td>
<td>1.0%</td>
</tr>
<tr>
<td>TIME</td>
<td>1845</td>
<td>208</td>
<td>225</td>
<td>2278</td>
<td>1.4%</td>
</tr>
<tr>
<td>WORK_OF_ART</td>
<td>1349</td>
<td>132</td>
<td>169</td>
<td>1650</td>
<td>1.0%</td>
</tr>
</tbody>
</table>

| Σ           | 138256 | 10620 | 12586 | 161462 | 100.0% |
| %           | 85.6%  | 6.6%  | 7.8%  | 161462 | 100.0% |

Table B.2: NE category distribution across the OntoNotes 5 English NE-annotated documents, using the Passos et al. (2014) splits.
B.3 Generating the Passos et al. (2014) splits

The reverse engineered procedure for generating the Passos et al. (2014) splits is presented in Algorithm 1. A number of questions are raised about the split creation process:

• Why are WSJ sections 00, 01, and 22 taken from the development set and placed in the training set instead? One guess might be that the original creators of these splits were wanting more WSJ data in their splits, possibly for improving a parser model.

• Why are all sentences which contain no NEs discarded? By doing this, the evaluation process is given less of a chance to make false positives.

• Why are sentences which consist of one single-token NE discarded? These seem like they would be easy instances to get correct, such as a location dateline in a newswire article.

• Why are four specific consecutive documents discarded? Inspecting these documents, there appears to be a misalignment with the gold standard named entity annotations — the gold standard files are marking nonsense as named entities (see Section 5.3.3 for examples). We are not aware of this artefact being documented anywhere.

We posed these questions to the authors, but did not get a response.
Algorithm 1 The process used by Passos et al. (2014) for creating their NER training, dev, and test splits over the OntoNotes 5 corpus.

procedure PROCESSSENTENCE(sentence, document, split)
  discard ← false
  \( n \) ← the number of NEs in the sentence
  if \( n = 0 \) then \( \triangleright \) All sentences without NEs are discarded
      discard ← true
  else if \( n = 1 \) then \( \triangleright \) All sentences which are a single one-token NE are discarded
    \( a \) ← does that NE span the whole sentence
    \( b \) ← is that NE one non-trace node in length \( \triangleright \) Parses contain trace nodes
    discard ← \( a \land b \) \( \triangleright \) Discard if both are true
  if \( \neg \text{discard} \) then
    KEEPSENTENCE(sentence, doc, split) \( \triangleright \) Add this sentence to the split

procedure PROCESSDOCUMENT(document, split)
  if the document does not have named entity annotations then
    return
  if the document ID matches ^tc/ch/00/ch_000[2-5].*$ then
    return \( \triangleright \) Arbitrarily discard these four documents
  if the document ID matches ^nw/wsj/(00|01|22)/.*$ then
    split ← train \( \triangleright \) Relocate these WSJ sections from dev to train
  for all sentence \( \in \) document do
    PROCESSSENTENCE(sentence, document, split)

\( D \) ← Download the “all” English CoNLL 2012 splits from the website
Sort \( D \) lexicographically by document ID \( \triangleright \) Needed to reproduce the provided files
for all document, split \( \in \) \( D \) do \( \triangleright \) split \( \in \) \{train, dev, test\}
  PROCESSDOCUMENT(document, split)


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 Eighth International Conference on Language Resources and Evaluation*.


David Ferrucci, Adam Lally, Karin Verspoor, and Eoric Nyberg. 2009. Unstructured information management architecture (UIMA) version 1.0, OASIS standard. Standard, OASIS.


Donna Harman and Mark Liberman. 1993. TIPSTER Complete. LDC catalog no.: LDC93T3A.


Frank Rosenblatt. 1957. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, Buffalo, NY, USA.


Charles Sutton and Andrew McCallum. 2004. Collective segmentation and labeling of distant entities in information extraction. In ICML Workshop on Statistical Relational Learning and Its Connections to Other Fields.


