Development and Analysis of NLP Pipelines in Argo

Rafal Rak, Andrew Rowley, Jacob Carter, and Sophia Ananiadou
National Centre for Text Mining
School of Computer Science, University of Manchester
Manchester Institute of Biotechnology
131 Princess St, M1 7DN, Manchester, UK
{rafal.rak,andrew.rowley,jacob.carter,sophia.ananiadou}@manchester.ac.uk

Abstract

Developing sophisticated NLP pipelines composed of multiple processing tools and components available through different providers may pose a challenge in terms of their interoperability. The Unstructured Information Management Architecture (UIMA) is an industry standard whose aim is to ensure such interoperability by defining common data structures and interfaces. The architecture has been gaining attention from industry and academia alike, resulting in a large volume of UIMA-compliant processing components. In this paper, we demonstrate Argo, a Web-based workbench for the development and processing of NLP pipelines/workflows. The workbench is based upon UIMA, and thus has the potential of using many of the existing UIMA resources. We present features, and show examples, of facilitating the distributed development of components and the analysis of processing results. The latter includes annotation visualisers and editors, as well as serialisation to RDF format, which enables flexible querying in addition to data manipulation thanks to the semantic query language SPARQL. The distributed development feature allows users to seamlessly connect their tools to workflows running in Argo, and thus take advantage of both the available library of components (without the need of installing them locally) and the analytical tools.

1 Introduction

Building NLP applications usually involves a series of individual tasks. For instance, the extraction of relationships between named entities in text is preceded by text segmentation, part-of-speech recognition, the recognition of named entities, and dependency parsing. Currently, the availability of such atomic processing components is no longer an issue; the problem lies in ensuring their compatibility, as combining components coming from multiple repositories, written in different programming languages, requiring different installation procedures, and having incompatible input/output formats can be a source of frustration and poses a real challenge for developers.

Unstructured Information Management Architecture (UIMA) (Ferrucci and Lally, 2004) is a framework that tackles the problem of interoperability of processing components. Originally developed by IBM, it is currently an Apache Software Foundation open-source project[1] that is also registered at the Organization for the Advancement of Structured Information Standards (OASIS)[2]. UIMA has been gaining much interest from industry and academia alike for the past decade. Notable repositories of UIMA-compliant tools include U-Compare component library[3], DKPro (Gurevych et al., 2007), cTAKES (Savova et al., 2010), BioNLP-UIMA Component Repository (Baumgartner et al., 2008), and JULIE Lab's UIMA Component Repository (JCoRe) (Hahn et al., 2008).

In this work we demonstrate Argo[4], a Web-based (remotely-accessed) workbench for collaborative development of text-processing workflows. We focus primarily on the process of development and analysis of both individual processing components and workflows composed of such components.

The next section demonstrates general features of Argo and lays out several technical details about

[1] http://uima.apache.org
[2] http://www.oasis-open.org/committees/uima
[3] http://nactem.ac.uk/ucompare/
[4] http://argo.nactem.ac.uk

UIMA that will ease the understanding of the remaining sections. Sections 3–5 discuss selected features that are useful in the development and analysis of components and workflows. Section 6 mentions related efforts, and Section 7 concludes the paper.

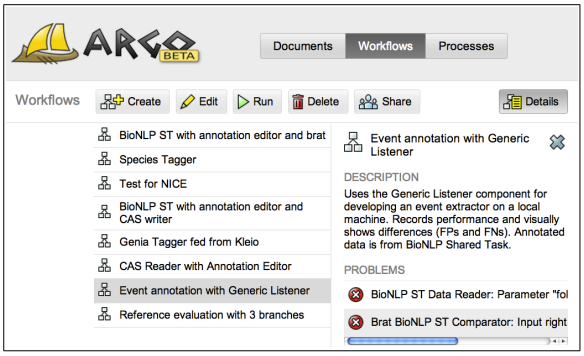
2 Overview of Argo

Argo comes equipped with an ever-growing library of atomic processing components that can be put together by users to form meaningful pipelines or workflows. The processing components range from simple data serialisers to complex text analytics and include text segmentation, part-of-speech tagging, parsing, named entity recognition, and discourse analysis.

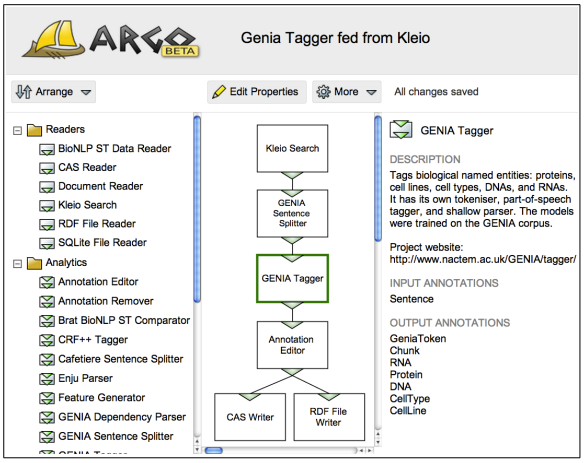
Users interact with the workbench through a graphical user interface (GUI) that is accessible entirely through a Web browser. Figure 1 shows two views of the interface: the main, resource management window (Figure 1(a)) and the workflow diagramming window (Figure 1(b)). The main window provides access to emphdocuments, *workflows*, and *processes* separated in easily accessible panels.

The *Documents* panel lists primarily user-owned files that are uploaded (through the GUI) by users into their respective personal spaces on the remote host. Documents may also be generated as a result of executing workflows (e.g., XML files containing annotations), in which case they are available for users to download.

The *Workflows* panel lists users' workflows, i.e., the user-defined arrangements of processing components together with their settings. Users compose workflows through a flexible, graphical diagramming editor by connecting the components (represented as blocks) with lines signifying the flow of data between components (see Figure 1(b)). The most common arrangement is to form a pipeline, i.e., each participating component has at most one incoming and at most one outgoing connection; however, the system also supports multiple branching and merging points in the workflow. An example is shown in Figure 2 discussed farther in text. For ease of use, components are categorized into readers, analytics, and consumers, indicating what role they are set to play in a workflow. Readers are responsible for delivering data for processing and have only an outgoing port (represented as a green triangle). The role of an-



(a) Workflow management view



(b) Worflow diagram editor view

Figure 1: Screenshots of Argo Web browser content.

alytics is to modify incoming data structures and pass them onto following components in a workflow, and thus they have both incoming and outgoing ports. Finally, the consumers are responsible for serialising or visualising (selected or all) annotations in the data structures without modification, and so they have only an incoming port.

The *Processes* panel lists resources that are created automatically when workflows are submitted for execution by users. Users may follow the progress of the executing workflows (processes) as well as manage the execution from this panel. The processing of workflows is carried out on remote servers, and thus frees users from using their own processing resources.

2.1 Argo and UIMA

Argo supports and is based upon UIMA and thus can run any UIMA-compliant processing component. Each such component defines or imports *type systems* and modifies *common annotation structures* (CAS). A type system is the represen-

tation of a data model that is shared between components, whereas a CAS is the container of data whose structure complies with the type system. A CAS stores *feature structures*, e.g., a token with its text boundaries and a part-of-speech tag. Feature structures may, and often do, refer to a *subject of annotation* (Sofa), a structure that (in text-processing applications) stores the text. UIMA comes with built-in data types including primitive types (boolean, integer, string, etc.), arrays, lists, as well as several complex types, e.g., Annotation that holds a reference to a Sofa the annotation is asserted about, and two features, begin and end, for marking boundaries of a span of text. A developer is free to extend any of the complex types.

2.2 Architecture

Although the Apache UIMA project provides an implementation of the UIMA framework, Argo incorporates home-grown solutions, especially in terms of the management of workflow processing. This includes features such as workflow branching and merging points, user-interactive components (see Section 4), as well as distributed processing.

The primary processing is carried out on a multi-core server. Additionally, in order to increase computing throughput, we have incorporated cloud computing capabilities into Argo, which is designed to work with various cloud computing providers. As a proof of concept, the current implementation uses HTCondor, an open-source, high-throughput computing software framework. Currently, Argo is capable of switching the processing of workflows to a local cluster of over 3,000 processor cores. Further extensions to use the Microsoft Azure[5] and Amazon EC2[6] cloud platforms are also planned.

The Argo platform is available entirely using RESTful Web services (Fielding and Taylor, 2002), and therefore it is possible to gain access to all or selected features of Argo by implementing a compliant client. In fact, the "native" Web interface shown in Figure 1 is an example of such a client.

3 Distributed Development

Argo includes a Generic Listener component that permits execution of a UIMA component that is running externally of the Argo system. It is primarily intended to be used during the development of processing components, as it allows a developer to rapidly make any necessary changes, whilst continuing to make use of the existing components available within Argo, which may otherwise be unavailable if developing on the developer's local system. Any component that a user wishes to deploy on the Argo system has to undergo a verification process, which could lead to a slower development lifecycle without the availability of this component.

Generic Listener operates in a reverse manner to a traditional Web service; rather than Argo connecting to the developer's component, the component connects to Argo. This behaviour was deliberately chosen to avoid network-related issues, such as firewall port blocking, which could become a source of frustration to developers.

When a workflow, containing a Generic Listener, is executed within Argo, it will continue as normal until the point at which the Generic Listener receives its first CAS object. Argo will prompt the user with a unique URL, which must be supplied to the client component run by the user, allowing it to connect to the Argo workflow and continue its execution.

A skeleton Java project has been provided to assist in the production of such components. It contains a Maven structure, Eclipse IDE project files, and required libraries, in addition to a number of shell scripts to simplify the running of the component. The project provides both a command-line interface (CLI) and GUI runner applications that take, as arguments, the name of the class of the locally developed component and the URL provided by Argo, upon each run of a workflow containing the remote component.

An example of a workflow with a Generic Listener is shown in Figure 2. The workflow is designed for the analysis and evaluation of a solution (in this case, the automatic extraction of biological events) that is being developed locally by the user. The reader (BioNLP ST Data Reader) provides text documents together with gold (i.e., manually created) event annotations prepared for the BioNLP Shared Task[7]. The annotations are selectively removed with the Annotation Remover and the remaining data is sent onto the Generic Listener component, and consequently, onto the developer's machine. The developer's task is to

[5] http://www.windowsazure.com

[6] http://aws.amazon.com/ec2

[7] http://2013.bionlp-st.org/

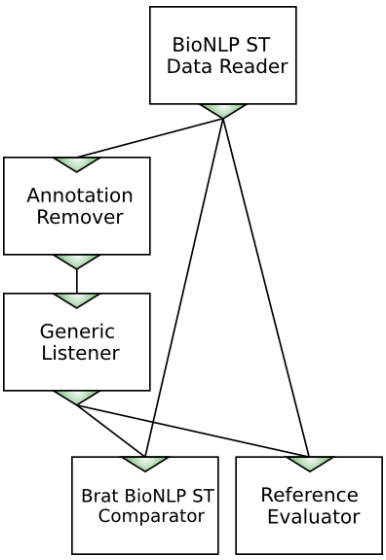


Figure 2: Example of a workflow for development, analysis, and evaluation of a user-developed solution for the BioNLP Shared Task.

connect to Argo, retrieve CASes from the running workflow, and for each CAS recreate the removed annotations as faithfully as possible. The developer can then track the performance of their solution by observing standard information extraction measures (precision, recall, etc.) computed by the Reference Evaluator component that compares the original, gold annotations (coming from the reader) against the developer's annotations (coming from the Generic Listener), and saves these measures for each document/CAS into a tabular-format file. Moreover, the differences can be tracked visually though the interactive Brat BioNLP ST Comparator component, discussed in the next section.

4 Annotation Analysis and Manipulation

Traditionally, NLP pipelines (including existing UIMA-supporting platforms), once set up, are executed without human involvement. One of the novelties in Argo is an introduction of *user-interactive components*, a special type of analytic that, if present in a workflow, cause the execution of the workflow to pause. Argo resumes the execution only after receiving input from a user. This feature allows for manual intervention in the otherwise automatic processing by, e.g., manipulating automatically created annotations. Examples of user-interactive components include Annotation Editor and Brat BioNLP ST Comparator.

The Brat BioNLP ST Comparator component

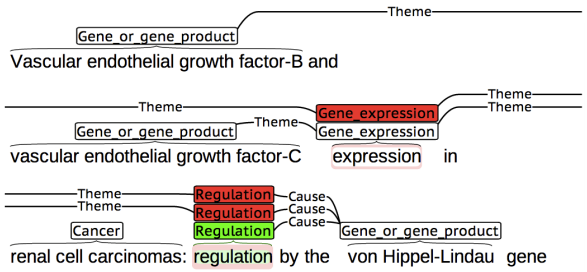


Figure 3: Example of an annotated fragment of a document visualised with the Brat BioNLP ST Comparator component. The component highlights (in red and green) differences between two sources of annotations.

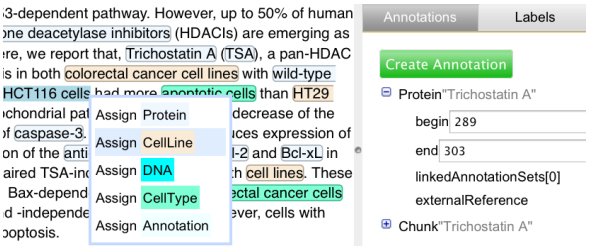


Figure 4: Example of manual annotation with the user-interactive Annotation Editor component.

expects two incoming connections from components processing the same subject of annotation. As a result, using brat visualisation (Stenetorp et al., 2012), it will show annotation structures by laying them out above text and mark differences between the two inputs by colour-coding missing or additional annotations in each input. A sample of visualisation coming from the workflow in Figure 2 is shown in Figure 3. Since in this particular workflow the Brat BioNLP ST Comparator receives gold annotations (from the BioNLP ST Data Reader) as one of its inputs, the highlighted differences are, in fact, false positives and false negatives.

Annotation Editor is another example of a user-interactive component that allows the user to add, delete or modify annotations. Figure 4 shows the editor in action. The user has an option to create a span-of-text annotation by selecting a text fragment and assigning an annotation type. More complex annotation types, such as tokens with part-of-speech tags or annotations that do not refer to the text (meta-annotations) can be created or modified using an expandable tree-like structure (shown on the right-hand side of the figure), which makes it possible to create any annotation

```
select ?neAText ?neACat ?neBText ?neBCat (count(*) as ?count)
{
  # retrieve sentence data
  { select ?sent ?sentBegin ?sentEnd ?sofa {
    ?sent a syn:Sentence ; b:sofa ?sofa ;
          t:begin ?sentBegin ; t:end ?sentEnd . } }

  # retrieve the 1st entity data
  { select ?ne1 ?ne1Cat ?ne1Begin ?ne1End ?sofa {
    ?ne1 rdf:type/rdfs:subClassOf* sem:NamedEntity ;
         a ?ne1Cat ; b:sofa ?sofa ;
         t:begin ?ne1Begin ; t:end ?ne1End . } }

  # retrieve the 2nd entity data
  { select ?ne2 ?ne2Cat ?ne2Begin ?ne2End ?sofa {
    ?ne2 rdf:type/rdfs:subClassOf* sem:NamedEntity ;
         a ?ne2Cat ; b:sofa ?sofa ;
         t:begin ?ne2Begin ; t:end ?ne2End . } }

  # the 2 entities must be enclosed in the sentence
  filter (?ne1Begin>=?sentBegin && ?ne1End<=?sentEnd
          && ?ne2Begin>=?sentBegin && ?ne2End<=?sentEnd)

  # extract covered text from the Sofa
  ?sofa s:sofaString ?text .
  bind (substr(?text, ?sentBegin+1, ?sentEnd-?sentBegin)
        as ?sentText) .
  bind (substr(?text, ?ne1Begin+1, ?ne1End-?ne1Begin)
        as ?ne1Text) .
  bind (substr(?text, ?ne2Begin+1, ?ne2End-?ne2Begin)
        as ?ne2Text) .

  # prevent symmetric duplicates
  filter(str(?ne1)<str(?ne2))
  bind(if(?ne1Text<?ne2Text, ?ne1Text, ?ne2Text) as ?neAText)
  bind(if(?ne1Text<?ne2Text, ?ne1Cat, ?ne2Cat) as ?neACat)
  bind(if(?ne1Text>=?ne2Text, ?ne1Text, ?ne2Text) as ?neBText)
  bind(if(?ne1Text>=?ne2Text, ?ne1Cat, ?ne2Cat) as ?neBCat)
}
group by ?neAText ?neACat ?neBText ?neBCat
order by desc(?count)
```

(a) Select query

neAText	neACat	neBText	neBCat	count
Ki-67	Protein	p53	Protein	85
DC	CellType	p53	Protein	61
DC	CellType	KCOT	Protein	47

(b) Results (fragment)

```
insert {
  _:relationship a sem:Relationship ;
                 sem:Relationship:arg1 ?ne1 ;
                 sem:Relationship:arg2 ?ne2 . }
{
  # retrieve sentence data
  { select ?sent ?sentBegin ?sentEnd ?sofa {
    ?sent a syn:Sentence ; b:sofa ?sofa ;
          t:begin ?sentBegin ; t:end ?sentEnd . } }

  # retrieve the 1st entity data
  { select ?ne1 ?ne1Begin ?ne1End ?sofa {
    ?ne1 rdf:type/rdfs:subClassOf* sem:NamedEntity ;
         b:sofa ?sofa ;
         t:begin ?ne1Begin ; t:end ?ne1End . } }

  # retrieve the 2nd entity data
  { select ?ne2 ?ne2Begin ?ne2End ?sofa {
    ?ne2 rdf:type/rdfs:subClassOf* sem:NamedEntity ;
         b:sofa ?sofa ;
         t:begin ?ne2Begin ; t:end ?ne2End . } }

  # the 2 entities must be enclosed in the sentence
  filter (?ne1Begin>=?sentBegin && ?ne1End<=?sentEnd
          && ?ne2Begin>=?sentBegin && ?ne2End<=?sentEnd)

  # prevent symmetric duplicates
  filter(str(?ne1)<str(?ne2))
}
```

(c) Insert query

Figure 5: Example of (a) a SPARQL query that returns biological interactions; (b) a fragment of retrieved results; and (c) a SPARQL query that creates new UIMA feature structures. Namespaces and data types are omitted for brevity.

structure permissible by a given type system.

5 Querying Serialised Data

Argo comes with several (de)serialisation components for reading and storing collections of data, such as a generic reader of text (Document Reader) or readers and writers of CASes in XMI format (CAS Reader and CAS Writer). One of the more useful in terms of annotation analysis is, however, the RDF Writer component as well as its counterpart, RDF Reader. RDF Writer serialises data into RDF files and supports several RDF formats such as RDF/XML, Turtle, and N-Triple. A resulting RDF graph consists of both the data model (type system) and the data itself (CAS) and thus constitutes a self-contained knowledge base. RDF Writer has an option to create a graph for each CAS or a single graph for an entire collection. Such a knowledge base can be queried with languages such as SPARQL[8], an official W3C Recommendation.

Figure 5 shows an example of a SPARQL query that is performed on the output of an RDF Writer in the workflow shown in Figure 1(b). This workflow results in several types of annotations including the boundaries of sentences, tokens with part-of-speech tags and lemmas, chunks, as well as biological entities, such as DNA, RNA, cell line and cell type. The SPARQL query is meant to retrieve pairs of seemingly interacting biological entities ranked according to their occurrence in the entire collection. The interaction here is (naïvely) defined as co-occurrence of two entities in the same sentence. The query includes patterns for retrieving the boundaries of sentences (`syn:Sentence`) and two biological entities (`sem:NamedEntity`) and then filters out the crossproduct of those by ensuring that the two en-

[8] http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

tities are enclosed in a sentence. As a result, the query returns a list of biological entity pairs accompanied by their categories and the number of appearances, as shown in Figure 5(b). Note that the query itself does not list the four biological categories; instead, it requests their common semantic ancestor `sem:NamedEntity`. This is one of the advantages of using semantically-enabled languages, such as SPARQL.

SPARQL also supports graph manipulation. Suppose a user is interested in placing the retrieved biological entity interactions from our running example into the UIMA structure Relationship that simply defines a pair of references to other structures of any type. This can be accomplished, without resorting to programming, by issuing a SPARQL insert query shown in Figure 5(c). The query will create triple statements compliant with the definition of Relationship. The resulting modified RDF graph can then be read back to Argo by the RDF Reader component that will convert the new RDF graph back into a CAS.

6 Related Work

Other notable examples of NLP platforms that provide graphical interfaces for managing workflows include GATE (Cunningham et al., 2002) and U-Compare (Kano et al., 2010). GATE is a standalone suite of text processing and annotation tools and comes with its own programming interface. In contrast, U-Compare—similarly to Argo—uses UIMA as its base interoperability framework. The key features of Argo that distinguish it from U-Compare are the Web availability of the platform, primarily remote processing of workflows, a multi-user, collaborative architecture, and the availability of user-interactive components.

7 Conclusions

Argo emerges as a one-stop solution for developing and processing NLP tasks. Moreover, the presented annotation viewer and editor, performance evaluator, and lastly RDF (de)serialisers are indispensable for the analysis of processing tasks at hand. Together with the distributed development support for developers wishing to create their own components or run their own tools with the help of resources available in Argo, the workbench becomes a powerful development and analytical NLP tool.

Acknowledgments

This work was partially funded by the MRC Text Mining and Screening grant (MR/J005037/1).

References

W A Baumgartner, K B Cohen, and L Hunter. 2008. An open-source framework for large-scale, flexible evaluation of biomedical text mining systems. *Journal of biomedical discovery and collaboration*, 3:1+.

H Cunningham, D Maynard, K Bontcheva, and V Tablan. 2002. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*.

D Ferrucci and A Lally. 2004. UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Natural Language Engineering*, 10(3-4):327–348.

R T Fielding and R N Taylor. 2002. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May.

I Gurevych, M Mühlhäuser, C Müller, J Steimle, M Weimer, and T Zesch. 2007. Darmstadt knowledge processing repository based on uima. In *Proceedings of the First Workshop on Unstructured Information Management Architecture*, Tübingen, Germany.

U Hahn, E Buyko, R Landefeld, M Mühlhausen, M Poprat, K Tomanek, and J Wermter. 2008. An Overview of JCORE, the JULIE Lab UIMA Component Repository. In *Language Resources and Evaluation Workshop, Towards Enhanc. Interoperability Large HLT Syst.: UIMA NLP*, pages 1–8.

Y Kano, R Dorado, L McCrochon, S Ananiadou, and J Tsujii. 2010. U-Compare: An integrated language resource evaluation platform including a comprehensive UIMA resource library. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation*, pages 428–434.

G K Savova, J J Masanz, P V Ogren, J Zheng, S Sohn, K C Kipper-Schuler, and C G Chute. 2010. Mayo clinical Text Analysis and Knowledge Extraction System (cTAKES): architecture, component evaluation and applications. *Journal of the American Medical Informatics Association*, 17(5):507–513.

P Stenetorp, S Pyysalo, G Topić, T Ohta, S Ananiadou, and J Tsujii. 2012. brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107, Avignon, France.