Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP

Foreword

The development and incremental modification of large and complex HLT systems has long been an art rather than a workflow guided by software engineering practices and principles. Therefore, in the past, interoperability of system components was hard to achieve, exchange of different modules a pain-staking task due to the low level of abstraction of specifications which described interfaces to connect with each other, as well as data and control flow inter-dependencies between various modules.

UIMA, the Unstructured Information Management Architecture, is an open-platform middle-ware for dealing with unstructured information (text, speech, audio, video data), originally launched by IBM. In the meantime, the Apache Software Foundation has established an incubator project for developing UIMA-based software (http://incubator.apache.org/uima/). In addition, the Organization for the Advancement of Structured Information Standards (OASIS) has installed a Technical Committee to standardize the UIMA specification. Accordingly, an increasing number of NLP research institutes as well as HLT companies all over the world are basing their software development efforts on UIMA specifications to adhere to emerging standards.

As far as NLP proper is concerned, Carnegie Mellon University's Language Technology Institute is hosting an UIMA Component Repository web site (http://uima.lti.cs.cmu.edu), where developers can post information about their analytics components and anyone can find out more about free and commercially available UIMA-compliant analytics. Additionally, free analytic tools that can work with UIMA include those from the General Architecture for Text Engineering (GATE - http://gate.ac.uk/) and OpenNLP (http://opennlp.sourceforge.net/) communities, as well as Jena University's Language & Information Engineering (JULIE) Lab (http://www.julielab.de). Commercial analytics are available from IBM, as well as from other software vendors such as Attensity, ClearForest, Temis and Nstein.

We considered LREC to be a particularly apt conference platform for the growing UIMA-inspired NLP community in order to meet, to exchange ideas and experience, as well as to think about future plans related to the UIMA framework. Much of these expectations are reflected in the proceedings of the first international workshop fully dedicated to UIMA topics – *"Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP"* held in conjunction with LREC 2008 in Marrakech, Morocco, May 31, 2008**.** From the Call for Papers, we received twelve submissions out of which five were selected as long papers and another five as short papers. The Organisation Committee of the workshop wants to thank the members of the Program Committee who reviewed the papers (each of which received three reviews, non-blind). Especially warm thanks go to Katrin Tomanek and Ekaterina Buyko from the JULIE Lab in Jena for their invaluable contributions to setting up and keeping the workshop's web page up to date, managing much of the communication with authors and PC members and, finally, assembling the final proceedings. Great job!

The members of the Organisation Committee,

Udo Hahn, Thilo Götz, Eric W. Brown, Hamish Cunningham, Eric Nyberg April 2008

Workshop Programme

Session	Time	Title
Opening	09:00 – 09:15	*Welcome and Opening Session* U. Hahn (JULIE Lab)
Invited	09:15 – 10:15	*TBA*
	10:15 – 10:45	**Coffee Break and Poster Session**
Component Repositories	10:45 – 11:10	*ClearTK: A UIMA Toolkit for Statistical Natural Language Processing* P. V. Ogren, P. G. Wetzler and S. J. Bethard
	11:10 – 11:35	*An Overview of JCoRe, the JULIE Lab UIMA Component Repository* U. Hahn, E. Buyko, R. Landefeld, M. Mühlhausen, M. Poprat, K. Tomanek and J. Wermter
Systems I	11:35 – 11:55	*Integrating a Natural Language Message Pre-Processor with UIMA* E. Nyberg, E. Riebling, R. C. Wang and R. Frederking
	11:55 – 12:15	*UIMA for NLP based Researchers' Workplaces in Medical Domains* M. Kunze and D. Rösner
	12:15 – 12:35	*UIMA-based Clinical Information Extraction System* G. K. Savova, K. Kipper-Schuler, J. D. Buntrock and C. G. Chute
	12:35 – 14:00	**Lunch**

Diverse	14:00 – 14:25	*CFE – a system for testing, evaluation and machine learning of UIMA based applications* I. Sominsky, A. Coden and M. Tanenblatt
	14:25 – 14:50	*Tools for UIMA Teaching and Development* M. Kunze and D. Rösner
	14:50 – 15:15	*Shallow, Deep and Hybrid Processing with UIMA and Heart of Gold* U. Schäfer
	15:15 – 15:45	**Coffee Break and Poster Session**
Systems II	15:45 – 16:05	*Experiences with UIMA for online information extraction at Thomson Corporation* T. Heinze, M. Light and F. Schilder
	16:05 – 16:25	*Flexible UIMA Components for Information Retrieval Research* C. Müller, T. Zesch, M.-C. Müller, D. Bernhard, K. Ignatova, I. Gurevych and M. Mühlhäuser
Closing	16:25 – 17:00	**Discussion and Closing Session**

Workshop Organisers

Udo Hahn , Jena University, Germany
Thilo Götz, IBM Germany, Germany
Eric W. Brown, IBM T.J. Watson Research Center, USA
Hamish Cunningham, University of Sheffield, UK
Eric Nyberg, Carnegie-Mellon University, USA

Workshop Programme Committee

Eric W. Brown, IBM T.J. Watson Research Center, USA
Ekaterina Buyko, Jena University, Germany
Hamish Cunningham, University of Sheffield, UK
Dave Ferrucci, IBM T.J. Watson Research Center, USA
Stefan Geissler, TEMIS Deutschland, Germany
Thilo Götz, IBM Germany, Germany
Iryna Gurevych, TU Darmstadt, Germany
Udo Hahn, Jena University, Germany
Nancy Ide, Vassar College, USA
Eric Nyberg, Carnegie-Mellon University, USA
Sameer Pradhan, BBN, USA
Dietmar Roesner, University of Magdeburg, Germany
Graham Wilcock, University of Helsinki, Finland

Table of Contents

Author Index

An Overview of JCoRe, the JULIE Lab UIMA Component Repository

U. Hahn, E. Buyko, R. Landefeld, M. Mühlhausen, M. Poprat, K. Tomanek, J. Wermter

Jena University Language & Information Engineering (JULIE) Lab
Friedrich-Schiller-Universität Jena
Fürstengraben 30, D-07743 Jena, Germany
{hahn|buyko|landefeld|muehlhausen|poprat|tomanek|wermter}@coling-uni-jena.de

Abstract

We introduce JCoRe, a full-fledged UIMA-compliant component repository for complex text analytics developed at the Jena University Language & Information Engineering (JULIE) Lab. JCoRe is based on a comprehensive type system and a variety of document readers, analysis engines, and CAS consumers. We survey these components and then turn to a discussion of lessons we learnt, with particular emphasis on managing the underlying type system. We briefly sketch two complex NLP applications which can easily be built from the components contained in JCoRe.

1. Introduction

During the past years, we have witnessed an unmatched growth of language processing modules such as tokenizers, stemmers, chunkers, parsers, etc. This software was usually created in a stand-alone manner, locally at the implementator's lab, and sometimes made publicly available on the programmer's personal or institutional web pages. In the last couple of years, several repositories have been set up, including, e.g., those of the Linguistic Data Consortium,[1] the Open Language Archives Community,[2] the European Language Resources Association,[3] and the Natural Language Software Registry[4]. As a common feature, these repositories just posted software modules but offered no additional service besides making available the plain resources (i.e., code, with – often fairly limited or even no – documentation). Hence, reusability was hampered by various different data exchange formats, let aside dependencies of different programming languages and operating systems. Any attempt to reuse this software or even create composite NLP systems from modules selected from these repositories created a heavy burden for system developers to achieve at least a decent level of interoperability. Under these conditions, although substantial collections of code were available, the compilation of NLP pipelines based on such components was quite inefficient and time-consuming.

Those Human Language Technologists already involved in complex system building activities, at that time, rendered rather monolithic and hard-shell pipelines that often resisted flexible exchange of single, externally developed components and their easy adaptation. Modification of these systems' architecture and basic functionality often required a major re-design, and, hence, re-programming directly at the code level.

With the advent of NLP framework architectures this impediment started to be resolved at the design level. GATE (Cunningham, 2002) and ATLAS (Laprun et al., 2002) were among the first of those systems that abstracted away from nitty-gritty programming details and moved system architectures to the level of data abstraction. UIMA, the *Unstructured Information Management Architecture*, provided additional abstraction layers, most notably by explicitly requiring a type system which described the underlying data structures to be specified (Ferrucci and Lally, 2004; Götz and Suhre, 2004).

Recently, second generation NLP repositories have been set up such as the one located at Carnegie Mellon University[5] or the JULIE Component Repository[6] (JCoRe), which we will describe in more depth in the remainder of this paper. JCoRe offers a large variety of NLP components for diverse NLP tasks which may range from sentence splitting, tokenization, via chunking and parsing, to named entity recognition and relation extraction. At the heart of JCoRe lies a comprehensive common type system for text analytics. Thus, the components in this repository can easily and flexibly be assembled into a variety of NLP applications without the need of any format conversion and re-programming.

After a brief introduction to UIMA in Section 2., in Section 3., we will describe JCoRe, the JULIE Component Repository, including the type system and the different kinds of readers, analysis engines, and consumers we currently supply. After that, in Section 4., we will discuss our experience with management issues related to the type system, in particular, dealing with type incompatibility and type system modifications.

2. UIMA In Brief

UIMA is a software framework and a platform for unstructured information management solutions. While originally developed by IBM, UIMA is now an Apache-licensed open source project. In the following we will shortly describe the basic concepts of UIMA. For more detailed and technical information we refer the reader to the Apache UIMA documentation.[7]

[1] http://www.ldc.upenn.edu
[2] http://linguistlist.org/olac
[3] http://www.elra.info
[4] http://registry.dfki.de
[5] http://uima.lti.cs.cmu.edu
[6] http://www.julielab.de
[7] http://incubator.apache.org/uima/documentation.html

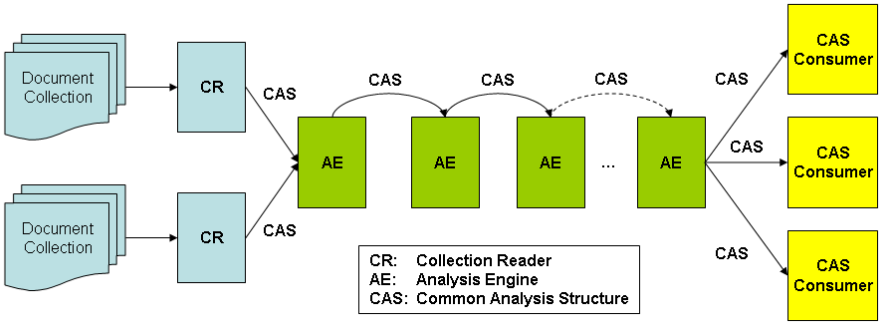


Figure 1: A schematic representation of an NLP pipeline to process unstructured data with UIMA components (Collection Readers (CR), Analysis Engines (AE), and Common Analysis Structure (CAS) Consumers). Information that is handed over and enriched or modified from AE to AE is managed within CAS objects that are based on a type system as their data model.

UIMA is a data-driven architecture which means that single components communicate with each other by exchanging (annotated) data. The integration of components in the UIMA framework thus requires clear interface definitions with respect to the input and the output data. The *Common Analysis Structure* (CAS) is UIMA's underlying object-oriented data structure (Götz and Suhre, 2004). The CAS represents one single item of unstructured data (e.g. a single document) and consists of one or more *Sofas* (subject of analysis, a view of the data item) with the meta data added by the single UIMA components. UIMA meta data objects are so called feature structures which are instances of UIMA *types*. These types can be arranged in an inheritance hierarchy and thus constitute a *type system* very similar to a class model in the object-oriented programming paradigm. The type system concept is UIMA's key feature to add structure to an unstructured chunk of data. For text processing purposes, UIMA comes with a predefined basic type, the annotation type. This annotation type and all its sub-types determine the particular annotation schema.

In UIMA, three different types of components are distinguished in the processing cycle (see also Figure 1): *Collection Readers* (CR) have different kinds of (typically unstructured, i.e., textual, audio or video) data as their input, textual documents in our case. CRs read this data from the selected source (files, database, etc.) and make it accessible for further processing steps. The linguistic processing proper of the documents is carried out by *Analysis Engines* (AE), each of which adds annotations according to different levels of analysis (e.g., POS tags, named entities, etc.). Finally, these annotation-enriched documents can be handed over to *CAS Consumers* (CC), which realize different functional requirements such as data conversion to specific output formats, search engine index construction, database feed, Web viewers, etc.

Several components, CRs, AEs, and CCs, can then be assembled into *pipelines* to build specific UIMA-based applications. Although UIMA is designed to be used for any kind of data, we here consider only textual data as our subject of analysis.

3. JCoRe — Julie Component Repository

JCoRe, the Julie Component Repository, provides all components to configure NLP analysis pipelines based on the UIMA framework presented in the previous section. A comprehensive annotation type definition is provided as the backbone of the whole system which allows flexible data exchange between all components involved. Several collection readers enable users to access markup and annotations from other projects within the UIMA framework. A continuously growing collection of text analytics components incorporates low-level NLP tasks such as sentence segmentation as well as high-end functionality in terms of relation extraction. Finally, several consumers are supplied to deploy or export the annotations.

All components of the JCoRe are written in Java. The components are available for download from `http://www.julielab.de/` as PEAR packages[8] and contain compiled classes, the source code, and an example model for components based on machine learning techniques.

Table 1 gives an overview of the components currently contained in JCoRe. In the remainder of this section we briefly describe the single components. For a more detailed explanation of any component, we refer the reader to the documentation contained in the PEAR packages and the cited publications.

3.1. Annotation Language: JCoRe's Type System

The data structure backbone of our component repository is a comprehensive annotation type system (Hahn et al., 2007; Buyko and Hahn, 2008), which covers major steps of NLP processing. It consists of several specification layers which provide (mostly) genre-, language-, and domain-independent definitions of the respective annotation types. When applied in specific scenarios, these generic annotation types might be extended by application-specific ones.

The *Document Meta* layer comprises annotation types for bibliographical and content information about a document – such as author, title, or year of publication. There is an extension to this layer for the biomedical domain allowing

[8] PEAR (Processing Engine ARchive) is a UIMA standard for packaging and automatically deploying components.

Component	Type	Comment	Source/Reference
JULIE Type System	TS	–	see Hahn et al. (2007), Buyko and Hahn (2008)
MEDLINE Reader	CR	–	–
ACE Reader	CR	–	–
MUC7 Reader	CR	–	–
JULIE Sentence Segmenter	AE	ML-based, self-developed	see Tomanek et al. (2007)
JULIE Token Segmenter	AE	ML-based, self-developed	see Tomanek et al. (2007)
Simple Sentence Segmenter	AE	rule-based, wrapper for JTokenizer	`http://www.andy-roberts.net/software/`
Simple Tokenizer	AE	rule-based, wrapper for JTokenizer	`http://www.andy-roberts.net/software/`
OPENNLP Sentence Segmenter	AE	ML-based, wrapper	`http://opennlp.sourceforge.net/`
OPENNLP Token Segmenter	AE	ML-based, wrapper	`http://opennlp.sourceforge.net/`
Stemmer	AE	rule-based, wrapper for Porter stemmer	`http://snowball.tartarus.org/`
OPENNLP POS Tagger	AE	ML-based, wrapper	`http://opennlp.sourceforge.net/`
OPENNLP Chunker	AE	ML-based, wrapper	`http://opennlp.sourceforge.net/`
OPENNLP Constituency Parser	AE	ML-based, wrapper	`http://opennlp.sourceforge.net/`
MST Dependency Parser	AE	ML-based, wrapped/modified	see McDonald et al. (2005)
Acronym Resolution	AE	rule-based, reimplementation	see Schwartz and Hearst (2003)
JULIE Named Entity Tagger	AE	ML-based, self-developed	–
Gazetteer	AE	dictionary, wrapper for Lingpipe's list look-up tool	`http://alias-i.com/lingpipe/`
JULIE Coordination Resolution	AE	rule-/ML-based, self-developed	see Buyko et al. (2007)
Relation Extractor	AE	ML-based, self-developed	–
Lucene Indexer	CC	–	–
CAS2DB Consumer	CC	–	–
CAS2IOB Consumer	CC	–	–

Table 1: Overview of Components in JCORE, the JULIE Component Repository

to store the meta information exclusively provided for documents when retrieved from PUBMED,[9] such as MESH[10] terms, chemicals, and genes referred to in a document. To incorporate information about the document structure, such as formal zones typically used in scientific texts (e.g., sections and paragraphs), the *Document Structure & Style* layer offers dedicated types. The types from the *Morpho-Syntax & Syntax* layer refer to linguistic annotations ranging from sentence up to parse annotations. Finally, the *Semantics* layer offers types for semantic annotations, including entities, relations, and events.

3.2. Preprocessing: Collection Readers

Currently, we provide three different exemplars of collection readers (see Table 1). Two of them import semantically annotated newswire corpora, *viz.* the ACE 2005 (Doddington et al., 2004) and the MUC-7 (Hirschman and Chinchor, 1998) corpora, and convert the given annotations to the CAS representation. From the annotated ACE corpus, the ACE *Reader* extracts named entities (persons, organizations, values, etc.), coreferences, relations, and events. From the MUC-7 data set, the MUC*7 Reader* extracts named entities and coreferences (event annotations are intentionally ignored). For both corpora, our type system has been extended with the respective types (Buyko and Hahn, 2008). Once, external annotations are read into the UIMA framework, this allows for further processing of the documents making immediate use of this annotated data. Moreover, such annotations might serve as input material for training and testing NLP components, such as named entities recognizers, coreference resolvers, and relation extractors.

The MEDLINE *Reader* parses MEDLINE records that come in an XML encoded format. They not only contain the plain text but also various meta data such as information about the authors and their affiliations, the publication date, information about the journal the article appeared in, manually assigned descriptors (mainly MESH terms), etc. In summary, all our readers extract the originally encoded (meta) data and map this information to the types and features of JCORE's type system.

3.3. Text Processing: Analysis Engines

JCORE contains text analytics components for different processing levels including linguistic preprocessing and semantic processing up to relation extraction at the time of this writing.

Depending on the task to be served, NLP component developers may either choose rule-based approaches or make use of machine learning (ML) methods (Hahn and Wermter, 2006). While, e.g., for the recognition of city names a simple gazetteer look-up might be sufficient, entity recognition in the biomedical domain is usually better performed by ML-based approaches due to complex and inconsistent naming conventions, ambiguities, etc. (Park and Kim, 2006).

JCORE contains both rule-based and ML-based components for language processing. For the ML-based compo-

[9] PUBMED (`http://www.ncbi.nlm.nih.gov/`) is a bibliographical database which includes over 17 million citations from MEDLINE and other life science journals for biomedical articles.

[10] Medical Subject Headings (MESH, `http://www.nlm.nih.gov/mesh`) is a high-coverage controlled biomedical terminology.

nents, we also provide some pre-trained models for download in case training material was freely available. Of course, these components can be retrained for usage in different domains or with different semantic types given the respective training material.

While some of our text processing components are entirely *self-developed*, others are based on already existing third party libraries or tools for which we wrote *wrappers* so that they could be used as a component inside the UIMA framework. Mostly, wrapping only meant to call the respective methods from within the analysis engine class and to convert and write the tool's output to the CAS. In some cases, however, wrapping required some *modifications* of the original tool to let it fit into the UIMA framework.

Linguistic Processing JCORE contains three components for both sentence and token segmentation. First, there are rule-based components which provide an UIMA wrapper for the JTokenizer,[11] a third party package mainly based on regular expression segmentation. The segmentation rules for these components can be flexibly defined by the user. Second, there are UIMA wrappers of the segmenter tools from the OPENNLP tool suite.[12] These are based on Maximum Entropy (ME) models (Berger et al., 1996). To address special intricacies of scientific subdomains such as biomedicine we have developed our own segmentation tools (Tomanek et al., 2007) based on Conditional Random Fields (CRF) (Lafferty et al., 2001) and a rich set of features.

To handle morphological variation of words (deletion of inflection suffixes, in particular) we have created a wrapper for the Java version of the SNOWBALL stemmers,[13] including the original Porter stemmer for English and additional versions for many other languages.

For syntactic analysis, we provide UIMA wrappers for the POS tagger, the phrase chunker, and the constituency-based parser from the OPENNLP tool suite. These are also based on ME models and have proven to work well on scientific documents when retrained on appropriate training data (Buyko et al., 2006). Further, we have integrated the MSTPARSER (McDonald et al., 2005), a parser for non-projective dependency structures, also based on ML methods. Writing an UIMA wrapper here also meant to slightly modify the MSTPARSER's source code so that the model needs to be loaded only once during the initialization phase.

Semantic Processing For acronym resolution, we reimplemented a simple, but well-performing algorithm originally presented by Schwartz and Hearst (2003): For each locally introduced acronym (some upper-case letters in brackets, such as "WHO"), the full form is searched to the left of this acronym until each letter from the acronym is found in the proper order of appearance. All occurrences of an acronym in a document are annotated with the identified full form.

Our repository comprises two tools for named entity recognition. One is based on Lingpipe's[14] list look-up tool. Provided with a list of names, these are searched for in the document. Both exact and approximate matching (based on weighted edit distance) are possible. Second, we have developed an entity tagger based on CRFs, which is similar in spirit to the one proposed by Settles (2004). Given appropriate training material, our ML-based entity tagger can be used for arbitrary domains and entity classes. It comprises a rich set of features which can be configured according to the respective scenario. Further, it allows for acronyms being expanded to their full forms during tagging (given they were marked before as such) to avoid erroneous tagging especially of ambiguous acronyms.

The repository also contains a component to resolve elliptical entity mentions in coordinations, such as normalizing *"Mr. and Mrs. Miller"* to *"Mr. Miller"* and *"Mrs. Miller"* (Buyko et al., 2007). Our coordination resolver can be configured either for the use of a set of rules considering POS information only, or for the use of an ML model with a variety of lexical, morpho-syntactic and even semantic features.

Finally, there is a component for relation extraction based on supervised ML. Here, an ME classifier is applied to determine for any ordered pair of two entities in a document whether these are in a specific relation. Relation extraction is currently the top level analysis component as it is based on the analysis results of many other components including, e.g., POS tagging, parsing, and entity recognition.

3.4. Postprocessing: CAS Consumers

The processing results of our text analysis components can be deployed by CAS Consumers. These components constitute an interface to arbitrary applications which use the UIMA annotations. A consumer that is a default part of UIMA allows to store the UIMA analysis results in the XMI (XML Metadata Interchange) format which is an OMG[15] standard for exchanging meta data based on XML (Extensible Markup Language). In many scenarios, however, consumers tailored to the specific needs of an application will be required. We here present three consumers which were created in the context of different NLP applications and information extraction research projects.

CAS2IOB Consumer The IOB format (inside, outside, begin) is a common exchange format for segmentation-based, non-nesting annotations (e.g., chunking (Ramshaw and Marcus, 1995)). Many publicly available corpora are annotated in this format (e.g., for the CONLL 2003 (Tjong Kim Sang and De Meulder, 2003) or the CONLL 2004 (Carreras and Màrquez, 2004) shared tasks). Furthermore, training, testing, and evaluation software that is based on this format has been developed for several competitions (e.g., CoNLL) and is widely accepted. The *CAS2IOB Consumer* extracts specified annotations from the CAS to the IOB format, following simple heuristics to resolve nested and overlapping annotations (e.g., preference for the longest annotation).

Lucene Indexer Apache Lucene[16] is an open source text retrieval software which can efficiently manage millions of

[11] http://www.andy-roberts.net/software/
[12] http://opennlp.sourceforge.net/
[13] http://snowball.tartarus.org/
[14] http://alias-i.com/lingpipe/
[15] http://www.omg.org/
[16] http://lucene.apache.org

documents. Our Lucene Indexer automatically creates a Lucene search engine index by mapping the CAS annotations to particular Lucene index fields. This allows us to retrieve documents not only by the information contained directly in the (unstructured) text itself but also by their meta data that is given by the documents' provider (e.g., author names, publication data, etc.) as well as the annotations added by the UIMA components. It can flexibly be used for any kinds of annotations. Mapping rules define the assignment of annotation types or their attribute to the particular fields in the Lucene index.

CAS2DB Consumer Whereas the Lucene Indexer makes feasible efficient search within the annotated documents, for further processing (such as displaying retrieved documents with all their annotations) the annotated documents must be stored in a way which allows fast access. This can be accomplished by our *CAS2DB Consumer*, which feeds the annotations derived from and assigned to the documents into a relational database. The CAS2DB Consumer is based on an abstract database schema which is independent of any particular annotation type system and thus allows flexible reuse. Once stored in the database, the data can be optimized and re-arranged according to the particular application's needs. Currently the database schema and the CAS2DB Consumer are implemented for the PostgreSQL[17] database management system, but it can easily be adapted to any other relational database system.

4. Lessons Learnt: Type System Management

Most difficulties we faced when extending our component repository and using the UIMA framework in our daily work are related to the management of the type system. In the following we discuss two of these problems together with a possible solution.

4.1. Type System Incompatibility

Although UIMA, in theory, allows for easy interoperability between UIMA components, this idea is only realized directly if all components are based on the same type system. Since, however, the NLP community has not yet agreed on a common NLP type system for UIMA, there are several home-grown, possibly very specific type systems in use for different components resulting in impaired interoperability. Assume the following example: Given a component A from the JCORE repository, e.g., a sentence splitter which writes its analysis output to the annotation type *de.julielab.types.sentence*, and a third party component B, e.g., a tokenizer which assumes as input sentence annotations stored in the annotation type *org.mylab.types.sentence*. Without further synchronization these two components cannot be linked in the same pipeline.

Now, we could of course modify component B to work on the sentence annotations of component A (or vice versa), given the source code were available. Yet, to avoid such source code modification, the following workaround seems helpful:[18] Write a small preprocessing AE which copies the relevant annotations from component A to the types component B expects. In a postprocessing AE, annotations created by component B will be copied and transferred to annotation types expected by the other components of a pipeline.

4.2. Type System Modification

Another problematic issue in the UIMA framework pops up when a type system used by several components of a repository is changed. Monotonic *extensions* of a type system, i.e., adding new types or extending the attributes of already existing types, is not really problematic. However, *modifications* of whole types or attributes (even if only names of types or attributes are changed), might lead to severe conflicts.

Why do type systems change? Although we have attempted to design a comprehensive and linguistically motivated type system before we started to implement the UIMA components, we continuously face the following two reasons for ubiquitous change:

- At the "borders" of a type system new subtypes are constantly required due to specific application scenarios. For example, for the semantics layer this could mean that we need special types inheriting from the general entity mention annotation type.
- But also the "core" of our type system is not immune to changes. This is mostly due to new findings regarding the design of specific annotation types.

The first issue can easily be solved by designing the type system as a domain- and application-independent core where application-specific types should not be integrated. Rather, specific requirements would be integrated into application-dependent extensions.

We have further split the "core" type system into several logical partitions, i.e., the layers addressed in Section 3.1. Each such layer is realized by a separate UIMA type system descriptor, possibly including other layers in case of dependencies between UIMA types. The availability of single logical units also adds to the clarity because our type system altogether contains several dozens of annotation types, partly arranged in a multi-level type hierarchy. Application-specific extensions can then be realized for the respective unit.

The second issue, i.e., modifications in the core type system, is more serious since components based on different versions of the type system might probably not be integrable into one pipeline due to conflicting annotation types. As we have organized all of our UIMA components as separate Java projects, managed by the build management tool Maven,[19] the modification of the type system currently implies lot of manual work because we need to go through all of these projects, exchange the type system (or at least the

[17] `http://www.postgresql.org`

[18] Thanks to Olivier Terrier from Temis for fruitful discussions and the idea for this solution.

[19] `http://maven.apache.org/`

respective layer if it has changed), see whether the component's source code needs to be updated with respect to the type modifications, and finally deploy the updated versions. This problem could be solved semi-automatically with the following labor-saving workflow. Once we have changed the type system in a way that it could impair the functionality of one of our UIMA components, and provided that for each component there is a (reasonable) functionality test (such as a JUnit[20] test), then, in a first step, all type system descriptors in every component will be replaced by the modified version and the Java classes of type system will be updated automatically. In a next step, the functionality test of each component will be executed. If the test runs successfully, the component will be marked as successfully type-system-updated. Otherwise, the developer in charge has to modify this particular component with respect to the new type system. Finally, only if the functionality tests of all components are passed without functional deviations, all components can be updated in a version control system (e.g., SVN[21]), be deployed as a Java project, and PEARs can be build in order to exchange the components easily. In particular by the last constraint which accepts only updated and fully functional components as part of the component repository, we can compile pipelines in a convenient way without running into errors caused by inconsistent type system versions.

The workflow proposed here has not been tested yet, but we are about to implement it as a repository management procedure in our lab.

5. Conclusion

We gave an overview of the JULIE Component Repository (JCORE). This work is motivated by our goal to develop complex NLP software in a disciplined and flexible way. We have implemented, up until now, two particular application systems both of which are fully set up by components from our UIMA component repository.

The STEMNET project[22] aims at building a semantic search engine for the biomedical subdomain of immunology. For this scenario, we set up an NLP pipeline which reads MEDLINE abstracts using the MEDLINE Reader and then processes these documents by means of linguistic preprocessing (sentence and token segmentation, POS tagging) and recognition of various entity types. The data sink here is the Lucene Indexer which stores the annotations in a search engine index which can then be queried by users searching for relevant scientific documents.

Our second application accounts for the automatic synthesis of a biomedical fact database as done in the BOOTSTREP project.[23] To identify interactions between proteins described in scientific documents, we set up a pipeline passing through all levels of linguistic and semantic processing, including especially relation extraction. Finally, the CAS2DB Consumer is used to store the identified relations as facts in a database.

Besides these lab-internal uses, our download statistics indicate that JCORE resources are of high interest to and used by many visitors of our web site. The Open Source policy we support allows external users to integrate our work in a flexible way in their application frameworks.

This is certainly an advantage over Web services often considered as an alternative. Relying on Web services, users have to turn to the developers of the code and negotiate with them changes they are after. With Open Source material they can do it on their own. Also Web services have certainly performance deficits when large amounts of data have to be shuffled across the WWW. Web services might be useful for testing but they might not really be competitive for large-scale production systems.

Finally, we hope that JCORE resources, the type system in particular, might stimulate discussions about emerging standards for NLP. It offers an appropriate level of abstraction to talk about the essential parameters of our research and development work.

Acknowledgements

This research was funded by the EC within the BOOTSTREP project (FP6-028099), and by the German Ministry of Education and Research within the STEMNET project (01DS001A-C). The first author is recipient of the *UIMA Innovation Award 2007* and holds an award grant from IBM.

6. References

Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Ekaterina Buyko and Udo Hahn. 2008. Fully embedded type systems for the semantic annotation layer. In *ICGL 2008 – Proceedings of the 1st International Conference on Global Interoperability for Language Resources*, pages 26–33. Hong Kong, SAR, January 9-11, 2008. City University of Hong Kong.

Ekaterina Buyko, Joachim Wermter, Michael Poprat, and Udo Hahn. 2006. Automatically adapting an NLP core engine to the biology domain. In Hagit Shatkay, Lynette Hirschman, Alfonso Valencia, and Christian Blaschke, editors, *Proceedings of the Joint BioLINK-Bio-Ontologies Meeting. A Joint Meeting of the ISMB Special Interest Group on Bio-Ontologies and the BioLINK Special Interest Group on Text Data Mining in Association with ISMB*, pages 65–68. Fortaleza, Brazil, August 5, 2006.

Ekaterina Buyko, Katrin Tomanek, and Udo Hahn. 2007. Resolution of coordination ellipses in biological named entities using conditional random fields. In *PACLING 2007 - Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics*, pages 163–171. Melbourne, Australia, September 19-21, 2007. Melbourne: Pacific Association for Computational Linguistics.

Xavier Carreras and Luís Màrquez. 2004. Introduction to the CONLL-2004 shared task: Semantic role labeling. In Hwee Tou Ng and Ellen Riloff, editors, *CoNLL-2004*

[20] http://junit.org/

[21] http://subversion.tigris.org/

[22] http://www.stemnet.de

[23] http://www.bootstrep.eu

– *Proceedings of the 8th Conference on Computational Natural Language Learning at HLT-NAACL 2004*, pages 89–97. Boston, MA, USA, 2004. Association for Computational Linguistics.

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

George Doddington, Alexis Mitchell, Mark Przybocki, Lance Ramshaw, Stephanie Strassel, and Ralph Weischedel. 2004. The Automatic Content Extraction (ACE) Program: Tasks, data, & evaluation. In *LREC 2004 – Proceedings of the 4th International Conference on Language Resources and Evaluation. In Memory of Antonio Zampolli. Vol. 3*, pages 837–840. Lisbon, Portugal, 26-28 May 2004. Paris: European Language Resources Association (ELRA).

David Ferrucci and Adam Lally. 2004. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

Thilo Götz and Oliver Suhre. 2004. Design and implementation of the UIMA Common Analysis System. *IBM Systems Journal*, 43(3):476–489.

Udo Hahn and Joachim Wermter. 2006. Levels of natural language processing for text mining. In Sophia Ananiadou and John McNaught, editors, *Text Mining for Biology and Biomedicine*, pages 13–41. Norwood, MA: Artech House.

Udo Hahn, Ekaterina Buyko, Katrin Tomanek, Scott Piao, John McNaught, Yoshimasa Tsuruoka, and Sophia Ananiadou. 2007. An annotation type system for a data-driven NLP pipeline. In *The LAW at ACL 2007 – Proceedings of the Linguistic Annotation Workshop*, pages 33–40. Prague, Czech Republic, June 28-29, 2007. Stroudsburg, PA: Association for Computational Linguistics.

Lynette Hirschman and Nancy Chinchor. 1998. Muc-7 coreference task definition (version 3.0). In *Proceedings of the MUC-7, Message Understanding Conference*, `http://www.itl.nist.gov/iad/894.02/related_projects/muc/proceedings/co_task.html`.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML-2001 – Proceedings of the 18th International Conference on Machine Learning*, pages 282–289. Williams College, MA, USA, June 28 - July 1, 2001. San Francisco, CA: Morgan Kaufmann.

Christophe Laprun, Jonathan G. Fiscus, John Garofolo, and Sylvain Pajot. 2002. A practical introduction to ATLAS. In M.G. Rodriguez and C. Paz Suarez Araujo, editors, *LREC 2002 – Proceedings of the 3rd International Conference on Language Resources and Evaluation*, pages 1928–1932. Las Palmas de Gran Canaria, Spain, 29-31 May, 2002. Paris: European Language Resources Association (ELRA).

Ryan McDonald, Fernando Pereira, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *HLT-EMNLP'05 – Proceedings of the 5th Human Language Technology Conference and 2005 Conference on Empirical Methods in Natural Language Processing*, pages 523–530. Vancouver, B.C., Canada, October 6-8, 2005. East Stroudsburg, PA: Association for Computational Linguistics.

Jong C. Park and Jung-Jae Kim. 2006. Named entity recognition. In Sophia Ananiadou and John McNaught, editors, *Text Mining for Biology and Biomedicine*, pages 121–142. Norwood, MA: Artech House.

Lance Ramshaw and Mitchell P. Marcus. 1995. Text chunking using transformation-based learning. In *Proceedings of the 3rd ACL Workshop on Very Large Corpora*, pages 82–94. Cambridge, MA, USA, June 30, 1995. Association for Computational Linguistics.

Ariel S. Schwartz and Marti A. Hearst. 2003. A simple algorithm for identifying abbreviation definitions in biomedical text. In Russ B. Altman, A. Keith Dunker, Lawrence Hunter, Tiffany A. Jung, and Teri E. Klein, editors, *PSB 2003 – Proceedings of the Pacific Symposium on Biocomputing 2003*, pages 451–462. Kauai, Hawaii, USA, January 3-7, 2003. Singapore: World Scientific Publishing.

Burr Settles. 2004. Biomedical named entity recognition using conditional random fields and rich feature sets. In Nigel Collier, Patrick Ruch, and Adeline Nazarenko, editors, *JNLPBA – Proceedings of the COLING 2004 International Joint Workshop on Natural Language Processing in Biomedicine and its Applications*, pages 107–110. Geneva, Switzerland, August 28-29, 2004.

Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CONLL-2003 shared task: Language-independent named entity recognition. In Walter Daelemans and Miles Osborne, editors, *CoNLL-2003 – Proceedings of the 7th Conference on Computational Natural Language Learning*, pages 142–147. Edmonton, Canada, 2003. Association for Computational Linguistics.

Katrin Tomanek, Joachim Wermter, and Udo Hahn. 2007. Sentence and token splitting based on conditional random fields. In *PACLING 2007 - Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics*, pages 49–57. Melbourne, Australia, September 19-21, 2007. Melbourne: Pacific Association for Computational Linguistics.

Experiences with UIMA for online information extraction at Thomson Corporation

Terry Heinze, Marc Light, Frank Schilder

Thomson Corp.
R&D
610 Opperman Drive, Eagan, MN 55123, USA
FirstName.LastName@Thomson.com

Abstract

We have built a pair of information extraction systems using UIMA (Unstructured Information Management Architecture). These systems have very low latency and run on financial news. We outline the implementation of these systems and report on our web service injection process, our type system, and an ANTLR (ANother Tool for Language Recognition) wrapper we implemented. We conclude with a list of UIMA strengths from our perspective and a wish list for future releases.

1. Overview

This paper reports on experiments at Thomson Corporation[1] R&D using UIMA[2] (Ferrucci and Lally, 2004) for the design and implementation of an information extraction system for processing financial news in an online (vs. batch) mode.

We have built a number of human language processing solutions most of which are part of one or more production systems. We often reuse code but not as systematically as we would like. We have experimented with the GATE framework (Cunningham et al., 2002) and have worked on a homegrown text processing framework. However, we decided to work with UIMA because we believed that it would support high throughput and low latency and because the APIs were intuitive for us.

In this paper we discuss a UIMA pipeline for identifying and linking entities (persons, locations, companies, money amounts, and percentages) and for extracting relations and events between these entities. The entities are linked to relevant authority files. The pipeline is being applied to financial newswire.

There are three highlights that we wish to present: (i) We developed a web service that wraps around the named entity recognizer implemented as an UIMA Analysis Engine. (ii) We added a new type system to the CAS type system representing the actual entities (not the text span in the text), relations and events. (iii) We interfaced UIMA with ANTLR in order to run parsers tagging money, time expressions etc.

2. Short System description

This section provides a high-level overview of our systems.

[1]The Thomson Corporation provides information-based solutions for lawyers, business people, nurses, doctors, scientists, and other professionals. Many of these solutions involve textual sources in combination with more structured sources such as databases of numeric and nominal information. Part of the "intelligent information" that Thomson products use is the identification of entities, relations, and events in textual sources and links from these mentions to database records that contain further information.

[2]`http://incubator.apache.org/uima/`

2.1. The named entity tagger and resolver

We use a pipeline of UIMA components that is constructed to process a single document at a time. The input documents may be XML, HTML, or plain text. The pipeline consists of multiple readers, our named entity tagger, our authority resolver, and finally multiple, optional consumers. A controller module receives the input document and a related metadata property file that contains processing options for the specific document. The controller instantiates the UIMA components described by a XML configuration file (similar to a Collection Processing Engine (CPE) descriptor, but allows for dynamic reconfiguration of the component descriptors) and creates the Common Analysis System (CAS) pool.

The original document is inserted into a SOFA (subject of annotation) by the controller. It also inserts any xpath instructions from the metadata into a secondary SOFA. Based on the document type specified in the metadata, a reader component is executed. The reader will parse the document (using the metadata SOFA instructions), create a plain text SOFA, insert the extracted plain text into that SOFA, and additionally store the information (in the form of feature structures) needed to re-construct the orginal document with inlined annotations in the source document SOFA.

The named entity tagger and resolver analysis engines are executed next. These engines create annotations on the plain text SOFA.

Finally, the controller calls in succession the requested consumers (enumerated in the metadata). We have written consumers for inlined annotated XML, HTML, and plain text. We also have a consumer that creates just a report of the standalone annotations. There is also a consumer that produces XMI. In each case, the consumer creates a separate SOFA and inserts the resulting document. Upon completion of the pipeline execution, the controller extracts each consumer document and returns them in an array.

2.2. Fact and event extraction

We are also using UIMA for the extraction of relations and events. Building on the extracted named entities, we identify relations and events that encompass multiple entities or

money or time expressions. Our current focus is on financial events, but we are also exploring legal events or events described in the scientific literature (e.g., protein-protein interaction).
The standard pipeline for relation and event extraction contains three Analysis Engines (AEs):

Named entity recognizer extracts organizations, person names, locations as well as money, percent and time expressions. We reused the same recognizer, as described in the previous section.

Sentence classifier decides whether a sentence potentially contains the desired relation or event patterns to be extracted.

Slot filler determines the entities that fill the slots of a predetermined template.

The type system for the financial domain covers events such as Merger & Acquisition and Earning announcements. The type system is a sub system of our Master type system described in section 3.2.
The input data are news wire text represented in XML and HTML. A first AE extracts the text and possibly useful metadata from the XML file and write this information into the CAS. Then the pipeline of the three AEs mentioned above is run resulting in an index of relations and events found in the document. Different CASConsumers produce outputs in tabular ASCII text format or in RDF output.

3. Highlights

3.1. Web service injection process

We have a web service that wraps our pipeline controller. The web service module instantiates our pipleine controller object and implements a single method. The method accepts two string parameters: the document itself, and a string with the metadata procssing instructions. The service method returns an array of strings, each corresponding to the output of the requested consumer(s).
The service has been implemented using open source AXIS2[3] (implemented using both simple Java beans and xml binding) within Tomcat.[4] The instantiation of the controller object occurs during service startup. This is typically an expensive operation, requiring several minutes to create in memory our requisite data structures for named entity tagging.
Once started, the service is capable of processing typically sized documents in the range of 50ms. We have an overhead of appoximately 25ms for network transmission and document decomposition (StAX parsing[5]) and reconstruction. An input document of 30k takes about 25ms to annotate and resolve. The time required by the named entity tagger is linear. Documents longer than our average 30k size take correspondingly longer to annotate.

[3] `http://ws.apache.org/axis2/`
[4] `http://tomcat.apache.org/`
[5] `http://stax.codehaus.org/`

3.2. Type System

While developing complex NLP systems that annotate, resolve and relate named entities, we saw the need for an internal representation of the described entity or relation. Representing a unique entity one can refer to as Fred Center, Mr. Center or he in the text has several advantages: (a) information about this person can be collected in one object, (b) relations can be represented between the actual entity and not between the annotated text string, and (c) an output routine can simply print a list of entities mentioned by the article. A natural place for storing such an abstract data structure that represents the entity and can refer back to all the mentions of the entity in the text would be the CAS.
The APACHE UIMA documentation 2.3.2. discussing the CAS system alludes to the possibility of resolving several mentions in a text (e.g., *Fred Center, Mr. Center, he*) to the same entity. An annotator could link the different mentions of the same person to an internal representation of the entity. However, no more details are provided and we are not aware of any AE that implements this concept of having an entity type that refers to the occurrence of an entity and links the annotations that refer to this entity together.
In order to utilize such a representation in the CAS, we developed a type system (cf. figure 1) that introduces a new type in parallel to the annotation type that represents entities, relations and events. We call this type `Element`. This type is a sub-type of `TOP` and hence does not inherit the begin and attribute from the `Annotation` type. The `Annotation` type has three sub-types for named entities such as organization, person and location names (i.e., `ENAMEX`), descriptions such as job titles, product names (i.e., `Descr`), and value expressions such as money, percent and time expressions (i.e., `Val`)

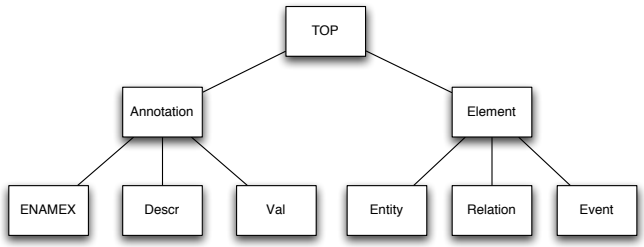


Figure 1: The top layers of the type system

On the other side, the `Element` type has three sub-types, too. `Entity` represents the entities that are annotated by an `ENAMEX` annotation and `Relation` represents the relations between descriptions (e.g., job title) and an entity. The type `Event` covers the most complex type that has entities, values and possibly other relations as attributes. `ENAMEX` and `Descr`, on the one hand, have attributes that reference the entities and the relations, and `Entity` and `Relation` reference the occurrences of the the entities and relations in the text (e.g., the mention of *Fred Center* and *Mr. Center*).
Consider the following example sentences that contains two persons and two job title *relations* even though the job title (i.e., CEO) is only mentioned once.

(1) In 2000 Gates transferred the title of CEO to Ballmer.

The elements generated for this sentence are listed in figure 2. Each element accumulates information from other parts of the document (e.g., first name) and refers back to all mentions of this element in the text via the attribute `occurences`.
The top layer of the type system can easily be extended with specialized sub types (e.g., financial events or events from the biomedical domain). The proposed type system could be plugged into the type system proposed by Hahn et al. (2007). They developed a multi-layered type system that covers different types covering document meta information, syntax, document structure as well as semantics. Our type system can be seen as an extension of the semantics layer.

3.3. ANTLR wrapper

ANTLR, ANother Tool for Language Recognition, developed by Terence Parr is a parser generator mainly used for computer languages.[6] However, it can also be used for recognizing smaller fragments of natural language such as money, percent or time expressions. We are using ANTLR for tagging such sub-languages because (a) a rule-based approach is for those expressions more appropriate than a statistical approach due to the regularity of these expressions, (b) the EBNF formalism is easier to maintain than (Java) regular extractions, and (c) ANTLR grammars are easily translatable into Java code.
ANTLR version 3 uses a flexible parsing strategy LL(*) which is more powerful than a standard LL(k) parser, because k does not have to be determined before running the parser. Still, the runtime complexity of a parser generated by ANTLR is linear.
A typical ANTLR grammar is divided into a lexer and a parser. The lexer takes a character stream and generates a token stream according to a lexer grammar. Such a lexer grammar could define simple token or complex expressions such as temporal expressions.
The token stream is then fed to the parser which parses the tokens according to the rules defined in the grammar. We utilized the ANTLR parser generator for parsing money, percent and time expressions as well as the parsing of educational background sentences in SEC filings.
Different interfaces we explored:

- UIMA-ANTLR *Lexer* takes a character stream (e.g., a sentence) and passes it to the lexer. The lexer gives back the offset information of tokens such as money or percent expressions. ANTLR can be run in a scanning mode that simply skips expressions that are not defined by the lexer.

  This interface can be easily integrated into a UIMA AE, because ANTLR generates a lexer java class that reads in a string and returns the offset information for the tokens. Additional information about the tokens can be collected, too. The ANTLR grammar formalism allows for inserting Java code within the rules. Hence, the normalization of money amounts, for example, can be carried out on the fly.

- UIMA-ANTLR *Parser* uses the ANTLR token stream manipulating the stream by introducing UIMA annotations. This interface comes in two flavors: (a) creating you own ANTLR token stream from UIMA annotations or (b) only change the token classes for tokens that have annotations in UIMA.

  Integrating an ANTLR parser requires an additional step, because an ANTLR token stream has to be generated. Iterating over entity annotations derived from other UIMA AEs, an ANTLR token source stream can be created that then can be fed into the parser for more complex analysis.

Figure 3 shows two example rules for temporal expressions. Note that non-terminals of the rules can be identified via a variable (e.g., `d`) which then has access to the actual value of the terminals covering this non terminal. The non-terminal `DAY` could, for example, cover the string '29th' or 'twenty-ninth' which will be normalized to the integer 29. This number will then be used to create an ISO time expressions such as `2008-02-29`.

```
TIMEX
:  ((d=DAY WS m=MONTH',' (WS? y=YEAR)?)|
          (s=SEASON WS 'of' WS y=YEAR)|
[...]


YEAR :       (('1'..'2')('0'|'9'|'8'))
NUMBER NUMBER;

NUMBER : ('0'..'9');
```

Figure 3: Example ANTLR grammar rules

The lexer will provide the offset information and the normalized values for the time expressions. As a next step the annotations for the temporal expressions and verb and noun chunks of the sentence could be merged into a token source stream that can be read by another ANTLR grammar that contains rules about attaching the temporal expressions to a noun or verb chunk (e.g., *he left at 3pm*, *the report for 2006*).
We used ANTLR for different sub-langages and found the generated Java code fast and reliable. Our previous experience with interfacing GATE with UIMA showed that GATE required a lot of memory and was generally slower than grammars written in ANTLR.

4. UIMA experience and a wish list

4.1. UIMA strengths

Our experience with using UIMA was generally positive. Here are three aspects we would like to emphasize.

- Speed: latency is a concern for us and UIMA performed well with respect to latency. We compared our UIMA pipeline to a pipeline where instead of using a CAS to pass annotations, we passed simple Java

[6] `www.antlr.org`

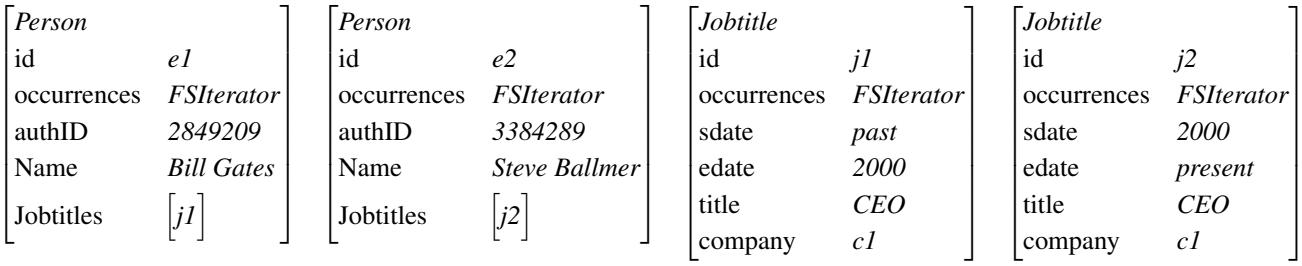


Figure 2: Elements generated from example sentence (1)

objects such as String[]. We found that the old IBM release of UIMA performed nearly as well. In addition, we believe that the Apache release is substantially faster.

- Ease of wrapping: we found that fitting our modules into the Analysis Engine API straightforward. In fact, we felt that doing so improved the structure of our code. In addition, we made extensive use of the Resource Files mechanism and again found fitting our modules into this mold straightforward.

- Feature structures & types: a number of members of our group have experience with HPSG and thus the use of UIMA feature structures and types was natural.

4.2. UIMA wish list

There are a number of UIMA components that if included in the Apache distribution, would make the move to UIMA easier:

- Event handling "collection" reader: Many applications run as a web service or daemon waiting for a document to push through the pipeline. Currently, the UIMA examples and documentation focuses on batch mode examples.

- Readers and consumers that handle XML and HTML: These are standard modules that most pipelines will need.

- Cas Editor: we eagerly await Jörn Kottmann's Cas Editor's release within Apache UIMA. We are currently working with Callisto (callisto.mitre.org) for manual correction of output for production of gold data. However, a tighter integration with UIMA and eclipse would simplify our work flow.

- "Standard" type hierarchy: a combined type hierarchy of the hierarchy proposed by Hahn et al.(2007) and our proposal could cover different document types (e.g., news, Medline abstracts) and standard entities and relations to be extracted. Given such a standard, Cas-Consumers could be shared for the presentation of the results (e.g., RDF viewer, tabular output).

- Access to objects outside of the scope of the CAS: We have encountered two cases where we have used a UIMA enabled application within existing workflows where we needed objects or artifacts that did not strictly fit the SOFA metaphor. In one case, we needed to retain ancillary information about the structure of the document from an HTML parser. We wanted to leverage existing code structures rather than re-implementing them as feature structures for the sole purpose of storing them in the CAS for later reference. Using global objects such as singletons did not support a long term design goal of distributing UIMA components across a network. Our solution was to serialize such objects to a byte array and then store them in a value of a feature structure in the CAS. Subsequent UIMA components can then reconstruct the object through de-serialization. This process seems to be against the intented use of feature structures.
  In another case, we needed to obtain metrics data from our UIMA components that is orthogonal and temporary to the pipeline. Here, we did use a singleton to collect data. It would be useful if UIMA implemented a form of aspects for cases such as this. Another standard example of this usage would be logging.

5. Conclusions

In this paper, we have described our information extraction pipeline which we implemented using UIMA. We then presented three highlights of the system: (i) a web service wrapping and document injection method, our method for interfacing our UIMA pipeline with other non-UIMA systems, (ii) a type system connecting annotations and extracted elements, and (iii) a wrapper for ANTLR.

6. References

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*.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4):327–348.

Udo Hahn, Ekaterina Buyko, Katrin Tomanek, Scott Piao, Yoshimasa Tsuruoka, John McNaught, and Sophia Ananiadou. 2007. An uima annotation type system for a generic text mining architecture. In *UIMA-Workshop, GLDV Conference*, April.

Tools for UIMA Teaching and Development

Manuela Kunze, Dietmar Rösner

Otto-von-Guericke-Universität Magdeburg
Institut für Wissens- und Sprachverarbeitung
P.O. Box 4120, D–39016 Magdeburg, Germany
makunze@iws.cs.uni-magdeburg.de, roesner@iws.cs.uni-magdeburg.de

Abstract

To support teaching UIMA a number of tools have been developed that are useful for UIMA developers as well. The UIMA comparator supports the comparison between the results of different UIMA analysis engines on the same documents. The UIMA class generator helps to minimize typing in UIMA development by automating the definition of classes and stubs for their methods based on an XML declaration. The usage of these tools in teaching is described and options for additional support are discussed in this paper.

1. Introduction

Our course on Information Extraction (IE) teaches concepts and algorithms for IE, using UIMA[1] and GATE (Cunningham et al., 2002) as reference models and as platforms for the assignment projects of the participating students.
The aim of the IE course is to make students acquainted with the problems of, approaches to and recent developments in IE in general and specifically with the architectures of frameworks for IE like UIMA and GATE. The students have to use various tools available in UIMA and GATE in order to solve given assignments. Within the UIMA framework students develop and extend analysis engines and lexical resources (e.g. lists of various types of named entities) and then apply these AEs to corpora of documents (mostly in German). In the last year, this included corpora with news, announcements of theatre plays, sport/games reports etc.[2]
The evaluation of students assignments related to UIMA is very time consuming. To reduce this effort, we developed an UIMA comparator that systematically compares the results of different UIMA analysis engines.
Students (or in general UIMA beginners) criticized that no tools are available that minimize typing in the development of UIMA. The effort for creating a new Analysis Engine (AE) puts beginners off to continue their work or experiments with UIMA. We created a UIMA Class generator that generates the Java classes and stubs for their methods based on an XML description.

2. Computer-Aided Assessment with eduComponents

As with all our lectures, the IE course makes use of our eduComponents software (Rösner et al., 2007), a collection of tools for enriching classes with e-learning and computer-aided assessment (CAA).
The UIMA comparator will be employed for supporting UIMA teaching within the framework for Computer-Aided Assessment (CAA) within the eduComponents system (Amelung et al., 2008). This section serves to give the necessary background information on this system, its design, implementation and employment.

2.1. Requirements

When we introduced computer-assisted assessment (CAA), we have been targeting for a system which supports:
- automatic testing of programming assignments in different programming languages from different programming paradigms as well as automatic testing of assignments in other formal systems, (e.g., regular expressions, XSLT transformations, UIMA analysis engines);
- evaluation and grading of assignments that demand for short natural language texts as answers;
- ease of integration of additional assignment types, programming languages or test methods.

2.2. A Generic Architecture for CAA

Based on the motivation and the requirements the main design idea is to separate all concerns of managing students, assignments, and submissions from the actual testing.
The former includes, for instance, storage of assignments and solutions, proper treatment of submission periods and re-submissions, communication of results to students, grading of the results, statistics for individual students and whole cohorts – and is typically provided and supported by a LMS (Learning Management System). In the following, we will use the term *frontend* for the LMS employed.
The actual testing is highly dependent on the kind of test method, programming language or other formal notation. For example, when testing programming assignments, the output of a student solution can be compared to that of a model solution for a set of test data, or the assignment can be tested for properties which must be fulfilled by correct programs. It also includes security precautions since running unknown code in an insecure environment could be dangerous. Thus, all aspects regarding the exact testing should be encapsulated and implemented in self-contained modules, we will call them *backends*.
Each backend defines a schema which describes all input fields necessary to fully specify the tests, e.g., a model solution and test data for test data based evaluation. It also

[1] http://incubator.apache.org/uima/index.html

[2] For detailed information (in German) about the IE courses given cf. for Summer 2007:
http://wdok.cs.uni-magdeburg.de/studium-und-lehre/lehrveranstaltungen/sommer2007/ie/

defines at least one *test method option* which allows the selection from different compilers or interpreters for a programming language and from different comparison functions (e.g., exact match vs. tolerance interval while comparing floating point numbers).
To integrate frontends and backends we introduce a new component which, similar to a printer spooler, manages a submission queue and several backends. The *spooler* provides the following functions: add new submissions for testing; get results from tests performed by a backend; show status information (e.g. available backends, number of submissions in queue); add or remove backends; get required input fields for testing with a certain backend; get available test method options.
Implementing spooler and backends in a service-oriented way would also increase the flexibility. *Web services*[3] represent an important approach to realize a service-oriented architecture. They enable an integration of applications more rapidly, easily, and with less costs. From a consumer perspective a Web service can be seen as black box that publishes its interface and functionality.
Fig. 1 shows an UML component diagram of our approach (Amelung et al., 2008). It differs from most other systems in its architecture, which clearly separates frontends, spooler, and backends, offering a high degree of flexibility and enabling a variety of frontends and backends to be used. It also allows running the components on disparate operating systems and in different environments over a network.

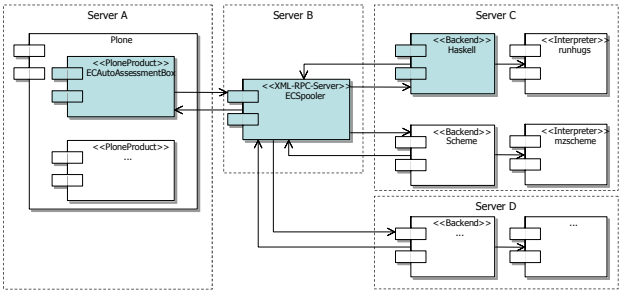


Figure 1: UML component diagram of the general system architecture

2.3. Implementation

To demonstrate that the generic approach described in 2.2. is actually working we used our already existing e-learning environment as frontend and implemented the spooler and various backend instances.
For this purpose we used *Python's XML-RPC* server and client API. XML-RPC[4] is a remote procedure call method using HTTP as the transport and XML as the encoding, and allowing complex data structures to be transmitted, processed and returned. With it, a client can call methods on a remote server.

[3]We realize a Web service as a piece of software designed to support interoperable machine-to-machine interaction over a network through standardized XML messages.

[4]`http://www.xmlrpc.com/`

2.3.1. Spooler

The spooler—we call it *ECSpooler*—is implemented as a Python XML-RPC server. When a student submits for an assignment via a Web interface, the submission as well as all necessary test data are first sent to ECSpooler which in turn passes it on to the backend specified by the teacher for this assignment. If the backend is busy, the test job (submission) remains in the queue until it can be handled. The results of the tests performed by the backend are temporarily stored and fetched by the frontend.
To avoid misuse or denial-of-service attacks job submissions and administrative tasks (e.g., start/stop spooler oder add/remove backends) require authentication. ECSpooler uses a simple user/password authentication scheme in which a user normally corresponds to a LMS, not to an individual person.

2.3.2. Backends

All available backends (e.g., for the programming languages Haskell, Scheme, Erlang, Prolog, Python, and Java) are implemented as Python XML-RPC servers and are derived from general backend classes. The latter provide a lot of functionality necessary to start/stop a backend or add it to a spooler. Thus, implementing new backends can be done simply by defining a new input schema, test options and at least one remote procedure call function.

3. Tools for UIMA

One of our teaching objectives is that our students learn to actively make use of UIMA. Therefore students get hands-on experiences and assignments where they first modify given UIMA analysis engines and later develop AEs (Analysis Engines) from scratch.
The tools that we discuss in the following have been designed and implemented based on experiences with the previous student cohorts and they will be employed in future teaching (e.g. in summer 2008).

3.1. UIMA class generator

The UIMA class generator is a plug-in for Eclipse[5] with the following functionality: From an XML description of an UIMA processing resource – i.e. AE, CAS Consumer, Collection Reader – the resp. Java classes together with stubs for their methods are automatically derived. The intention is to minimize typing in the development of UIMA processing resources. Please note that our students have argued for the need of such a tool.
During their work with UIMA, the students complained the lack of tools that could make the work with UIMA easier. An example: In all but the most trivial UIMA application, it is necessary to create more than one AE or consumer. UIMA does not yet support the automatic creation of stubs for the respective Java classes (i.e. empty AE classes or consumer classes), but repeatedly copying and editing existing Java files was experienced by the students as an avoidable waste of time. The deficiency is overcome by the UIMA class generator, who takes the name of the class (and/or information about capabilities and parameters) and automatically generates the Java files with plain (standard)

[5]`http://www.eclipse.org/`

methods, e.g. initialize(), process(),..., that can then be refined.

3.1.1. The Eclipse Plugin

The UIMA class generator can be started in the Eclipse environment after editing the XML descriptor of an AE, CAS Consumer or Collection Reader via mouseclick on the icon (see Fig. 3) or via selection of the menu entry in the menu bar. The UIMA class generator takes as input the name of an active project in the Eclipse environment and the XML file opened in the Eclipse editor. The generated class file is saved in the source directory of the active project. If the generated Java class exists, the user can choose between several file operations to solve the conflicts (e.g. renaming and overwriting of files).
If the file is not a valid XML descriptor, the process of class creation is aborted.
In the following example, we describe which information is used for the generation. In this case, we exploit a XML descriptor for an analysis engine. For each type of descriptor (Collection Reader, CAS Consumer, and Analysis Engine), we developed different class generators. The UIMA class generator reads the XML descriptor, analyses the tags within the descriptor, and starts the specific class generator.

3.1.2. An Example for Generation

The relevant excerpts from a descriptor file of an analysis engine are given (see Ex. 1). The content of the node 'annotatorImplementationName' is used to define the package statement and the name of the Java class. For each 'ConfigurationParameter', the declaration statement is inserted and the *initialize* method contains the statements for reading the values of parameters. The information of the node 'capabilities' is used to add additional *import* statements in the Java file. Fig. 3 presents the generated Java file for the given example.

(1)
```
...
  <annotatorImplementationName>tools.DemoAE
   </annotatorImplementationName>
...
    <configurationParameters>
      <configurationParameter>
        <name>demoParameter1</name>
        <type>Integer</type>
        <multiValued>true</multiValued>
        <mandatory>true</mandatory>
      </configurationParameter>
      <configurationParameter>
        <name>demoParameter2</name>
        <type>String</type>
        <multiValued>false</multiValued>
        <mandatory>false</mandatory>
      </configurationParameter>
    </configurationParameters>
...
    <capabilities>
      <capability>
        <inputs/>
        <outputs>
          <type allAnnotatorFeatures="true">
           types.wozStatements</type>
          <type allAnnotatorFeatures="true">
           types.woz</type>
          <type allAnnotatorFeatures="true">
           types.sub</type>
          <type allAnnotatorFeatures="true">
           types.Emotion</type>
        </outputs>
        <languagesSupported/>
      </capability>
    </capabilities>
 ...
```

3.1.3. UIMA Class Generator in Practice

The developed Class Generator is provided for students of our current exercise courses that are related to UIMA and student assistants of our UIMA projects. Our student assistants confirm that the UIMA class generator is an useful tool that saves time during development of UIMA based processing resources. We hope that we get a similar feedback from the students of our exercise courses at the end of the course.
The next step in development is to generate a UIMA Class Generator that has the same functionalities like the Eclipse plugin but that can be started in a console, so that developers that not use the Eclipse environment can also use the UIMA Class Generator. As parameters, the XML Descriptor and the source directory must then be given.

3.2. UIMA comparator

For automatic testing of student assignments in UIMA, a comparator has been developed and implemented and will now be evaluated and then employed in future courses. The comparator validates students analysis results against results of a master solution. The manual evaluation of students solutions is time consuming. To avoid this effort, we defined an interface (see Fig. 2) for the definition of assignments and for the comparison of results. This generalizes our experience with automatic testing of programming assignments (Rösner et al., 2007).
The definition of assignments contains the following information: a Type System Description and additionally a definition of the name and type (form) of external resources. A list of implementations of AEs and a corpus of documents for the evaluation has to be provided to the comparator. Each implementation is then applied by the comparator on the evaluation corpus. Different views on a document resulting from different AEs are compared with a master solution. The output of the comparator is a summary containing information which student solutions have the same results and what differences and varieties are detected. The comparator compares the position and content of annotations (including features of annotations).

3.2.1. A Multipurpose Tool

The UIMA comparator serves to apply two AEs to a file and then to systematically compare the resulting annotations. It may be employed both in UIMA AE development as well as in UIMA teaching for automatic checking of student solutions in assignments with AE development:

- For UIMA AE development differences between analysis results of a preceding and a (hopefully) improved version of an AE are of interest.
- In UIMA teaching it is a very valuable asset to be able to compare a bunch of student versions of an AE with the master solution of the teacher.

3.2.2. A Closer Look

The UIMA comparator works as follow:

- Given pointers to two AEs (one seen as the master AE, the other as the student AE) and a document, it runs the AEs on (separate versions of) the document.
- The resulting files – called master result and student result - are then compared on the basis of their annotations and their features:
  - Can all annotations from the master result be found in the student result (see Fig. 6)?
  - If not, which ones are missing (i.e. 'false negatives')?
  - Are there annotations in the student result that are not in the master result (i.e. 'false positives')?
- For each pair of corresponding annotations the analysis can be continued for the set of features and their values.
  - A flag allows to declare, if leading or trailing white-space shall be significant when comparing the span of annotations

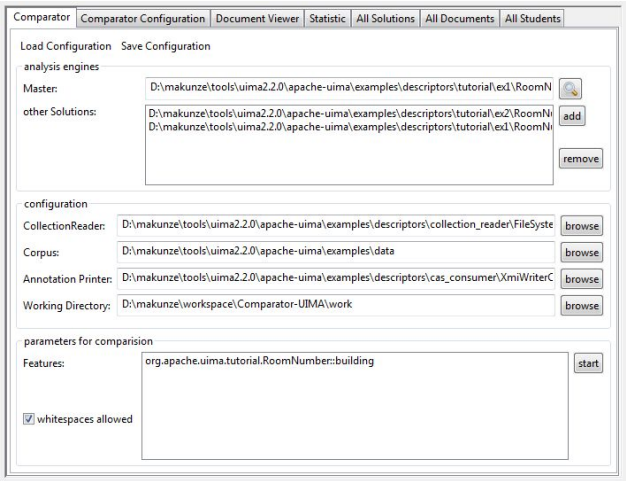


Figure 2: Configurator for UIMA Comparator.

3.2.3. Process Pipeline

The process pipeline contains the following steps: A selected collection reader reads the documents, then the several analysis engines (master and selected other annotators) are applied. After each annotator an analysis engine called *sofaCreator* is applied. For this annotator, we used an UIMA based concept of a specific presentation of annotations. This concept is defined as Sofa (Subject of Analysis) and it allows to group different annotations in a kind of a 'view'. It is possible to define different views on a document. We used this concept to group the annotations of an annotator in a Sofa. This step simplifies the comparison process, because handling and accessing of the annotations of the several annotators will be easier.
The annotator *sofaCreator* creates a Sofa with the name of the previous annotator, which contains all annotations of this annotator. That means, the annotations of an analysis engine are moved from the (standard) '_InitialView' to a new Sofa.

After running all annotators, the comparator is applied. The comparator insert two new annotations *CompResults* and *SolResult* into the '_InitialView'.

3.2.4. Annotations of the Comparator

These annotations contain the results of comparison for each annotation of the master solution and a summary for each (student) annotator and for each document in the corpus.
For each annotation of a solution (except the master annotation), an annotation of type *CompResults* is created. This annotation contains the following data:

- name of solution,
- information about annotation of the analysed solution,
- information about corresponding master annotation,
- flag about correct position and
- flag about correct feature values.

These annotations will be analysed and summarized by the comparator to generate the annotation *SolResult*. This annotation contains the following information in detail:

- name of solution,
- number of correct annotations,
- number of annotations that are annotated by the master solution and
- number of annotations with wrong feature values.

These comparator annotations are exploited to generate the different overviews about results.

3.2.5. Presentation of Results

The presentation of results from the UIMA comparator are available in several variations (We will give only a short description about the different views here. For more details cf. the appendix with several screenshots of result presentations):

Document View. There is a graphical interface that displays master result and student result on two parallel panes with color coded annotations (see Fig. 4).

Detailed Statistical Overview. For a given document and a given solution, a table view is presented that contains name of annotation, results about correctness of position and features (e.g. highlighted by a red background), and a summary about all features (and feature values) of the annotation (see Fig. 5).

Overview about all Solutions for a Document. This presentation is a tabular view that contains for each solution the results for a document (see Fig. 6).

Comparison of all Solutions. All documents and results of all solutions are presented. This overview provides a fast overview about all solutions for a specific document (see Fig. 7).

Student's Perspective. In this table view (see Fig. 8), the results are listed for all documents and a summary about all documents for a solution.

4. Future Work

Our future work is focussed on two projects: 1. The integration of the comparator into eduComponents and 2. Developing of an UIMA DocWriter.

Integration of the Comparator into eduComponents
In this project, we started with the conceptualization of the interfaces for the comparator. Following reimplementations are necessesary for the integration:

- interfaces for input of parameters (by teacher)
- interfaces for students (upload for their solution)
- several presentations for results (teacher view and student view)

These reimplementations are related to user interfaces. The core of the UIMA Comparator, i.e. the comparison and evaluation, will be the same like described in this paper. The relevant Java libraries are integrated into in the framework of our CAA system.

UIMA DocWriter
To manage large UIMA projects or several UIMA applications, it is necessary to quickly get an overview of the used annotators, annotation types, and so on. A PEAR package supports only the installation of a new UIMA application, but it delivers no documentation about the internal workflow of the application. Another useful tool for documentation, Javadoc, manages only information from Java files but gives not a holistic picture of the complete UIMA workflow.
Currently, a complete documentation of an UIMA project must be written by hand and this is very time consuming.
This bottleneck can be overcome by a 'UIMA DocWriter'. An UIMA DocWriter will take an AE descriptor or a CPE (Corpus Processing Engine) descriptor as input and will automatically generate a documentation comprising relevant information:

- a description of annotators,
- used annotation types and their features,
- capabilities,
- workflow of CAS processes,
- applied external resources,
- ...

This documentation will be supplemented by the generated Javadoc documentation of the containing Java files.
For a quick overview, the generated documentation offers a kind of concise data sheet, which contains only the most relevant information but with links to further details. The data sheet can as well be used for presentation and retrieval of a specific UIMA application in the 'UIMA' download portal.
Following descriptors should be supported:

- Analysis Engine (primitive and aggregate)
- Type System
- CAS Consumer
- Collection Reader
- Collection Processing Engine

The Document Writer will return the documentation as HTML, as PDF or as ASCII text file. Links between descriptors (e.g. Analysis Engines refers to a Type System) will also be presented as e.g. hyperlinks. The user can choose in an interface which information should be printed and described in the documentation.
We started with the documentation of Analysis Engines as HTML based description.

5. Summary

We reported about several tools supporting the development of UIMA based applications as well as the evaluation of UIMA based annotators.
The UIMA Class Generator reduces the time effort for creation of new annotators, collections readers, or CAS Consumers. For a beginner, it is easier to work with such a tool. Besides minimizing the typing, the generated Java class avoids the beginners' search for failures caused by typing errors or missing statements in the class file.
The second tool presented in this paper is the UIMA comparator. The comparator compares annotations of several Analysis Engines with a given master solution. The comparison is based on a match of position and features of the annotations. We developed this tool to reduce the effort while comparing students' solutions with a master solution. The comparator provides different presentations of results (give overview about all solutions vs. show a document with all solutions). Our aim is to integrate this comparator into CAA facility of the eduComponents framework.
The comparator is as well a helpful tool in UIMA development: It allows to quickly detect how changes in UIMA AEs lead to changes in annotation results.

Acknowledgement

We have to thank our student researchers, Y. Vershynin, F. Fricke, and A. Shaker for their efforts in developing and implementing these tools.

Note on Availability

The tools described in this paper are available from the authors upon request. eduComponents are open source and are available from
`http://wdok.cs.uni-magdeburg.de/software/`.

6. References

M. Amelung, P. Forbrig, and D. Rösner. 2008. Towards generic and flexible web services for e-assessment. In *ITiCSE '08: Proceedings of the 13th annual SIGCSE conference on Innovation and technology in computer science education*, New York, NY, USA. ACM Press. to appear.

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. Rösner, M. Piotrowski, and M. Amelung. 2007. A sustainable learning environment based on an open source content management system. In Wilhelm Bühler, editor, *Proceedings of the German e-Science Conference (GES 2007)*. Max-Planck-Gesellschaft.

A Appendix: Screenshots

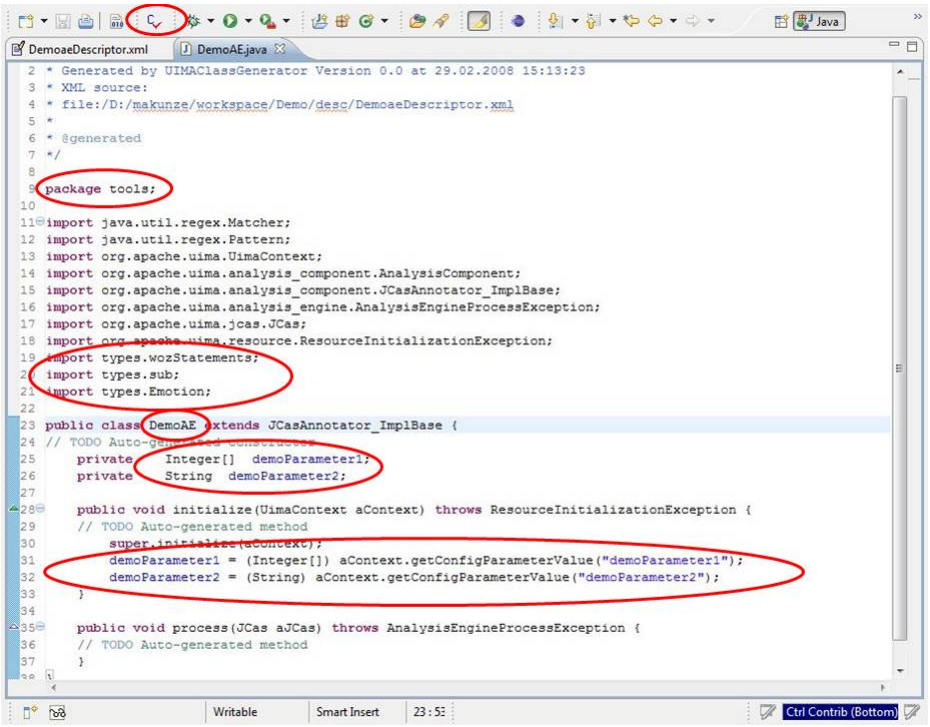


Figure 3: UIMA Class Generator: Generated Java Class.

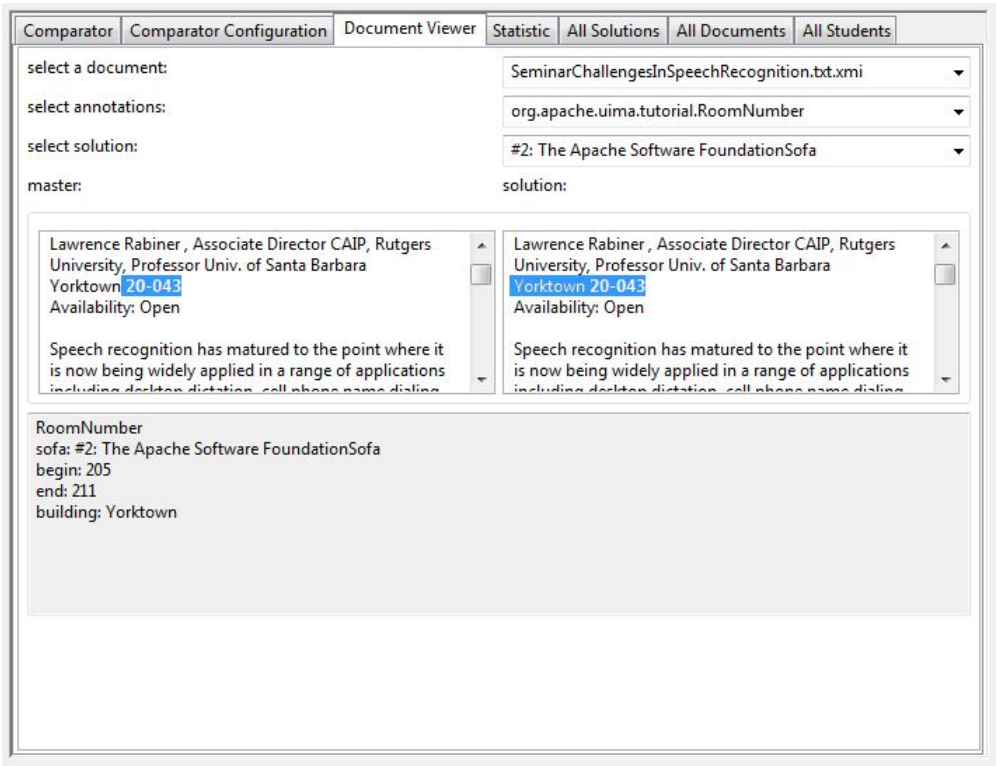


Figure 4: UIMA Comparator: Document viewer for results from different analysis engines (master and student).

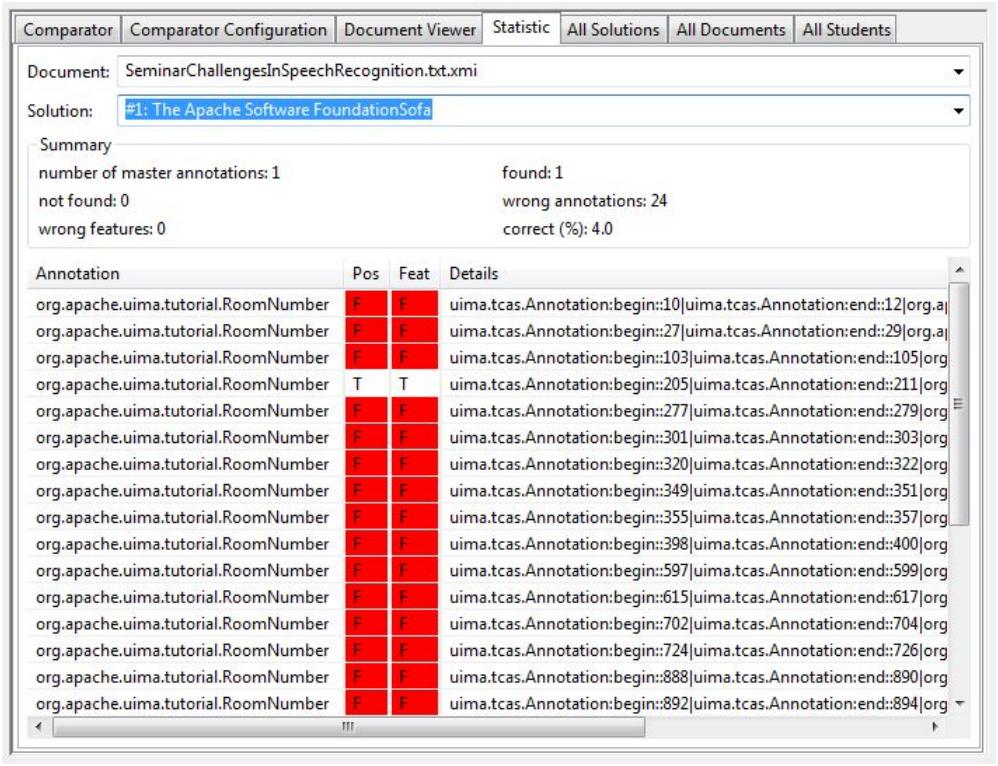


Figure 5: UIMA Comparator: Detailed statistical overview.

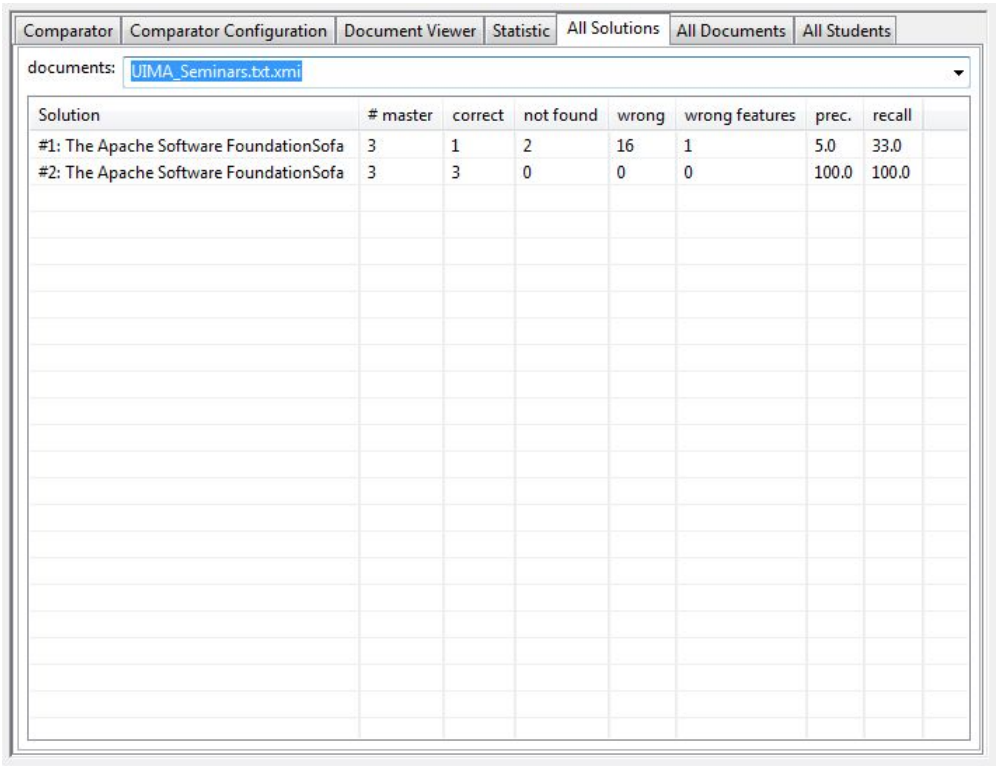
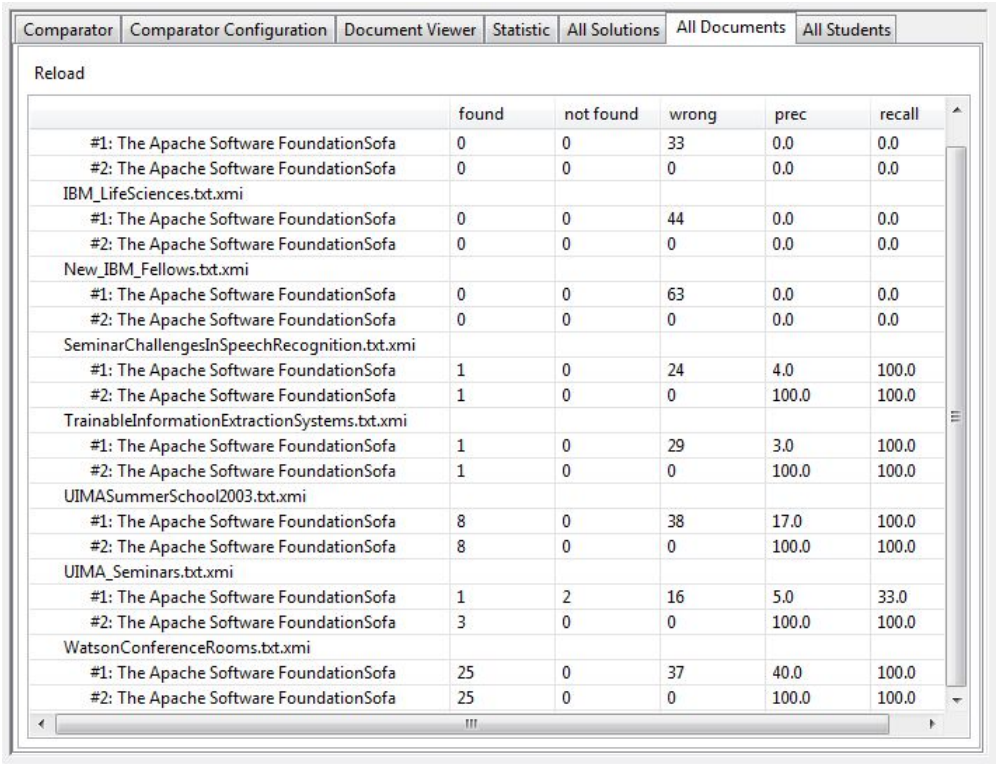
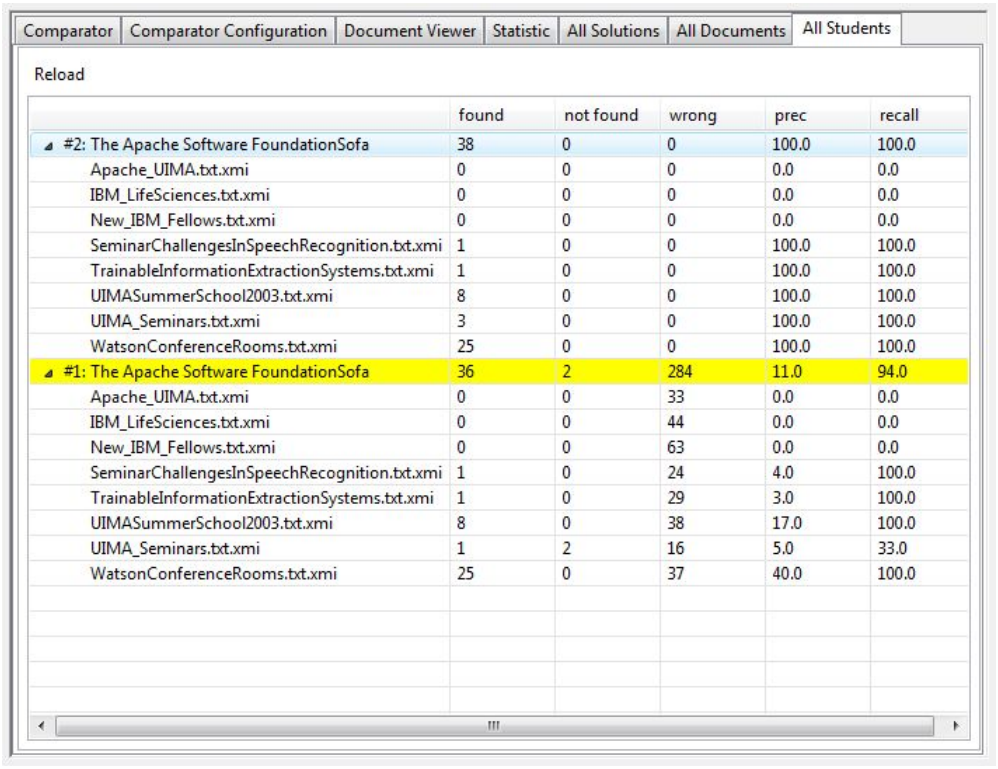


Figure 6: UIMA Comparator: Overview about all analysis engines and a specific document in the collection.



	found	not found	wrong	prec	recall
#1: The Apache Software FoundationSofa	0	0	33	0.0	0.0
#2: The Apache Software FoundationSofa	0	0	0	0.0	0.0
IBM_LifeSciences.txt.xmi					
#1: The Apache Software FoundationSofa	0	0	44	0.0	0.0
#2: The Apache Software FoundationSofa	0	0	0	0.0	0.0
New_IBM_Fellows.txt.xmi					
#1: The Apache Software FoundationSofa	0	0	63	0.0	0.0
#2: The Apache Software FoundationSofa	0	0	0	0.0	0.0
SeminarChallengesInSpeechRecognition.txt.xmi					
#1: The Apache Software FoundationSofa	1	0	24	4.0	100.0
#2: The Apache Software FoundationSofa	1	0	0	100.0	100.0
TrainableInformationExtractionSystems.txt.xmi					
#1: The Apache Software FoundationSofa	1	0	29	3.0	100.0
#2: The Apache Software FoundationSofa	1	0	0	100.0	100.0
UIMASummerSchool2003.txt.xmi					
#1: The Apache Software FoundationSofa	8	0	38	17.0	100.0
#2: The Apache Software FoundationSofa	8	0	0	100.0	100.0
UIMA_Seminars.txt.xmi					
#1: The Apache Software FoundationSofa	1	2	16	5.0	33.0
#2: The Apache Software FoundationSofa	3	0	0	100.0	100.0
WatsonConferenceRooms.txt.xmi					
#1: The Apache Software FoundationSofa	25	0	37	40.0	100.0
#2: The Apache Software FoundationSofa	25	0	0	100.0	100.0

Figure 7: UIMA Comparator: Overview about all documents and all solutions.



	found	not found	wrong	prec	recall
#2: The Apache Software FoundationSofa	38	0	0	100.0	100.0
Apache_UIMA.txt.xmi	0	0	0	0.0	0.0
IBM_LifeSciences.txt.xmi	0	0	0	0.0	0.0
New_IBM_Fellows.txt.xmi	0	0	0	0.0	0.0
SeminarChallengesInSpeechRecognition.txt.xmi	1	0	0	100.0	100.0
TrainableInformationExtractionSystems.txt.xmi	1	0	0	100.0	100.0
UIMASummerSchool2003.txt.xmi	8	0	0	100.0	100.0
UIMA_Seminars.txt.xmi	3	0	0	100.0	100.0
WatsonConferenceRooms.txt.xmi	25	0	0	100.0	100.0
#1: The Apache Software FoundationSofa	36	2	284	11.0	94.0
Apache_UIMA.txt.xmi	0	0	33	0.0	0.0
IBM_LifeSciences.txt.xmi	0	0	44	0.0	0.0
New_IBM_Fellows.txt.xmi	0	0	63	0.0	0.0
SeminarChallengesInSpeechRecognition.txt.xmi	1	0	24	4.0	100.0
TrainableInformationExtractionSystems.txt.xmi	1	0	29	3.0	100.0
UIMASummerSchool2003.txt.xmi	8	0	38	17.0	100.0
UIMA_Seminars.txt.xmi	1	2	16	5.0	33.0
WatsonConferenceRooms.txt.xmi	25	0	37	40.0	100.0

Figure 8: UIMA Comparator: Results of different AEs for all documents for a solution.

UIMA for NLP based Researchers' Workplaces in Medical Domains

Manuela Kunze, Dietmar Rösner

Otto-von-Guericke-Universität Magdeburg
Institut für Wissens- und Sprachverarbeitung
P.O. Box 4120, D–39016 Magdeburg, Germany
makunze@iws.cs.uni-magdeburg.de, roesner@iws.cs.uni-magdeburg.de

Abstract

We present our experiences from building domain specific NLP applications on the basis of the UIMA framework. The framework is used in processing of documents in German from the medical domain. In this paper, we report about two such NLP applications. The first one is processing of diagnostic summaries in the domain of psychotherapy and the second one is processing of autopsy protocols. The results of both applications are user interfaces that support the domain experts in their specific research.

1. Introduction

In this paper, we present our experiences from building domain specific NLP on the basis of the UIMA framework.[1] Our experiences with UIMA relate to using UIMA in university teaching as well as in different NLP applications. Two of our projects are presented in this paper. These projects analyse and annotate documents from the medical domain. For the purpose of processing the resp. corpora, we combined different tools and resources. The UIMA framework provides the basis for this work. The results are used from researchers in medical domains. We have built interfaces as prototypical researchers' workplaces that present results to researchers and that support the researchers to define their own search queries against the annotated corpus. A complete overview about our UIMA activities (including screenshots of several applications) is given at our UIMA project site.[2]

Why UIMA?

UIMA promises to support the re-use of tools (e.g. analysis engines) from other applications. The tools are configured by XML based descriptions that are linked to the implementations of these tool. Parameters, resources and types of annotations are defined in these descriptors. The handling of such descriptions is relatively easy, and parameters for an analysis engine can be changed also by a non UIMA expert. For the Java implementation of these tools there exist a variety of interfaces which can be extended.
For each processing step (reading a document, analysing it, and printing results), Java interfaces are available. UIMA provides an engine (collection processing engine) to manage and to launch these processing steps for a whole corpus. The modular structure of the framework makes it easy to use tools (e.g. analysis engines) in different applications, while the concept of annotations defined in UIMA makes it possible to exchange results between different applications.
In the next sections, we give a description of our applications in more detail. We describe two use cases for UIMA based medical applications. The first one processes diagnostic summaries and the second application analyses autopsy protocols. For each use case, we give a short description of the corpus and an overview about the processing pipeline. Finally, a comparative summary is given.

[1] http://incubator.apache.org/uima/

[2] http://wdok.cs.uni-magdeburg.de/forschung/projekte/uima-workbench/projects/

2. NLP Architecture

In general, the architecture of our applications can be divided into four modules:

- preprocessing of documents,
- analysis of documents,
- preparing results for postprocessing, and
- user-interface for domain experts (a kind of researchers' workplace).

Each module integrates functionalities from the UIMA framework. The different applications are described in the following use cases.

2.1. Use Case #1: Processing Epicrises

This application will be used to support psychotherapists to analyze a corpus of so-called epicrises (i.e. summary reports of diagnoses and treatment for specific patients). Their central research question is whether this corpus allows to detect significant changes in the distribution of diagnoses that can be related to the fundamental changes of the socio-political system after 1989 in former East Germany. The general issue here is the development of evaluation strategies for corpora in which temporal information is relevant and has to be taken into account.
The work started with a feasability study based on a corpus containing diagnostic summaries only (which are parts of epicrises which will be under investigation in the near future) from patient records from a psychotherapeutic clinic located in former East Germany. The dates of the corpus documents span over a time period from before the collapse of the communist regime (the so-called 'Wende' or 'Change' of 1989) till today. The diagnostic summaries are written in a very compact form and those from after 1990 are in the most of cases even in a verbless form. A representative summary contains the following parts (a predefined order of these parts is not given):

- symptoms,
- characteristics or personality of the patient,
- diagnosis,
- and (optional) causal incidents.

2.1.1. Preprocessing Module

The task of the preprocessing module is to prepare the documents for the analyses. This requires also some analysis engines. The documents in our corpus contain the collected diagnostic summaries made in a year. For the purpose of subsequent analyses, these documents are splitted into distinct files. Each of these files then contains only one diagnostic summary. Besides different collection readers, the module contains a diagnostic detector and a diagnostic printer. The diagnostic detector annotates the different diagnostic summaries and extract from the summary the ID for this summary. Then the diagnostic printer creates for each diagnostic summary a file. The name of the file consists of the ID of the diagnostic summary.

2.1.2. Analysis Module

In this module, different analysis engines were combined. We used domain specific and non domain specific tools. In the following, we will give a short description of these tools.

Taggers. The application exploits a *structure tagger* and a *POS tagger*. The *structure tagger* annotates numbers, urls, ip numbers, punctuations, and abbreviations. The *POS tagger* uses a Java based reimplementation of the POS Tagger of the XDOC system (Rösner and Kunze, 2002) and of the morphological component MORPHIX for German (Finkler and Neumann, 1988). The POS annotation contains information about the word category, stem form, as well as information about case, gender, and number. This information is later used by other analysis engines.

Concept Detection. The annotation of concepts within documents is based on three different approaches. The first annotator uses a kind of *gazetteer lists*. Each list contains phrases or tokens for a specific category. The lists are generated by hand or extracted from Wikipedia.[3] There are lists about syndroms, symptoms, diseases etc.
The second concept annotator is a tagger (for details see (Kunze and Rösner, 2004)) that uses the resouces of *GermaNet* (Kunze, 2001). For each token, the annotator looks for a synset in GermaNet which contains the token. If a synset was found, then the synset that is assigned in the hyperonymy relation is used as concept description for the token.
The *UMLS annotator* is the third annotator for detection of concepts. This annotator uses the resources of UMLS (Unified Medical Language System).[4] We developed a tagger that used the information about concepts from the Metathesaurus and information about types in the Semantic Network of UMLS.[5] An example: The term 'Angst' (in English: fear) will by this annotator be assigned with the following information:

- type: T184,
- type description: Sign or Symptom,
- concept: C0003467, and
- concepts description: MSH|Persistent feeling of dread, apprehension, and impending disaster.[6]

The GermaNet and UMLS analysis engines are also applied in our other projects. By an extension or exchange of gazetteer lists, the gazetteer annotator could easily be used for documents from other domains.

Synonym Recognition. The synonym recognition tool annotates domain specific synonymous terms and phrases for diagnoses. In the feasability study ten types of diagnoses (e.g. 'depression', 'psychosomatic disorders', etc.) are distinguished. The synonyms are recognised by regular expressions and are annotated with their type (e.g. depression). This type will be evaluated by other analysis engines (e.g. by the subfragment classifier).

Detection and Classification of Subfragments. For a better evaluation of results, it was necessary to split the diagnoses into different parts (e.g. personal characteristics, symptoms, etc.). For this task, we first use a discourse marker annotator and after this a splitter has been applied. For this process, we created a list of discourse markers. These discourse markers are classified into several groups that introduce specific subfragments. For example, the following phrases introduce the fragment about causal incidents: *infolge*, *in Folge*, *als Folge*, *nach*, *reaktiviert durch*, *als Reaktion auf*, *im Zusammenhang mit* (some examples in English for this category are: as result of, in consequence of, after, because of, in conjunction with, etc.).
This information is used by a subfragment classifier. The task of the subfragment classifier is to assign parts of a diagnostic summary to a fragment category. These fragment categories are diagnosis, symptoms, personal characteristics and causal incident.
We developed two approaches. The first one is rule-based and the second employs an implementation of an OpenNLP-Maxent-based classifier.[7]
The rule-based classifier analyses information about the introducing discourse marker and the discourse marker following of a subfragment and checks for specific keywords within the subfragment. After separation, the subfragments of personal characteristics and causal incidents were classified into specific categories (e.g. familial conflicts vs. job-related conflicts). This was done by analysing the keywords.
The maximum entropy classifier uses the following features:

[3] http://www.wikipedia.de/

[4] http://www.nlm.nih.gov/research/umls/

[5] The Metathesaurus is a large multi-lingual database that contains several descriptions about biomedical and health related concepts from different thesauri, classifications etc. Each concept description contains information about its source vocabulary. The concepts are assigned to at least one semantic type from the Semantic Network in UMLS.

[6] MSH is the source for this concept description and stand for MeSH: Medical Subject Headings (see http://www.nlm.nih.gov/mesh/).

[7] http://maxent.sourceforge.net/

- category of introducing discourse marker,
- category of following discourse marker,
- occurences of specific UMLS and GermaNet categories within the subfragment, and
- occurences of categories of synonyms in the subfragment.

For the purpose of building the model of the classifier, we used a training corpus of about 50 diagnostic summaries. The subfragments were extracted by a specific collection processing engine and classified by hand. This data is used as input for training the classifier.

2.1.3. Results Processing and User Interface

In this module, the results are prepared for the evaluation by psychotherapists. The annotations of the documents were transformed into different formats. The psychotherapist can use the documentation viewer of UIMA or a domain specific viewer. This viewer presents the different subfragments (personal characteristics, causal incident, symptoms, and diagnosis) separately together with the whole document content. When a classification was possible for the subfragments 'personal characteristics' and 'causal incident', the results are also presented in the viewer (otherwise the original text of the subfragment is presented).

Another presentation form is a statistical overview about detected concepts. This view will be extended with master file data to generate more specific statistics (e.g. number of depressions in a concrete year). Furthermore, there exists several consumer implementations for indexing the annotated documents. We integrated a search engine based on Lucene[8] and UIMA's semantic search engine.

To operate with the system, an user interface is available for psychotherapists. The interface provides functions to launch the different processing steps (separation, analyses), to view results and to search in the corpus via UIMA's semantic search engine or Lucene.

2.2. Use Case #2: Processing Autopsy Protocols

This application analyses a corpus of autopsy protocols (Wittig et al., 2006). Results of processing will be used by medical doctors in their research, e.g. for the purpose of detection of injury patterns and creation of resp. statistics. The corpus contains forensic autopsy protocols in German with more than 1 million running word forms. The autopsy protocols have a strictly defined content and layout. They are separated into different document parts, e.g. findings, background, discussion, death causes, etc. Each document part has its own characteristics (sub-language).

2.2.1. Preprocessing Module

In this case, we need a preprocessing step for anonymisation. For this task, an analysis engine is used, which annotates sensible data like names of person, locations and dates. A CAS consumer replaces this data by placeholders and a human reader rechecks the results of this preprocessing step before subsequent processing steps will be launched.

[8] http://lucene.apache.org/java/

2.2.2. Analysis Module

For analysing, we used different analysis engines too. The non domain specific tools for POS tagging, concept detection etc. described in the previous sections are again used. In this section, we will only describe in more detail the domain specific analysis engines.

Context Based Analysis. This tool combines several domain specific annotators for the detection and analysis of medical concepts. These annotators detect relations between injuries and their resp. locations (heart, kidney, etc.).

Personal Data Annotator. This annotator extracts for each autopsy protocol information about age and weight of the person. This data is relevant for statistical evaluation (e.g. number and kinds of death causes for specific age groups).

Traumata Annotator. The traumata annotator searches for specific mentions of injuries in autopsy protocols. It applies a list of regular expressions. We differentiate between several trauma categories: hematoma, fractures, stab wound, gunshot wound, and so on. For example, the category fractures covers different kinds of fractures: comminuted fracture, splintered fracture, cranial fracture, etc.

Weapons Annotator. For evaluations about numbers of acts of violence (e.g. in a specific year), it is necessary to search for occurrences of weapons. The weapons annotator annotates these occurrences (e.g. in the background part of an autopsy protocol). The annotator uses several regular expressions and each expression belongs to a specific weapon category. For example: poniard and knife are assigned to the category *thrusting*, while axe and hatchet are assigned to the category *baton*.

Summary Annotator. The summary annotator uses regular expressions to extract information about death cause and manner of death. This information is used for the purpose of a first classification of autopsy protocols.

Criminal Offense Signs Annotator. This annotator creates annotations for terms and phrases that describe signs of criminal offense. These terms are categorized into several types. For example: 'Durchtrennung' (in English: transection by a knife) and 'Stichkanal' (in Englisch: stab canal) are assigned to the category 'stab injury'.

2.2.3. Results Processing and User Interface

The results of the different annotators are prepared for textual summaries and for UIMA's semantic search engine. The annotations are used for building up a search index. To support the user for creating specific user queries, we extend UIMA's query interface. The user can select a specific annotation and its features from a list of possible annotations and feature values (e.g. the annotation 'Alter' (in Englisch: age) has a feature labeled with 'Bereich' (in English: range). This feature can have the values: 'bis10', '11-19', '20-29', and so on.) The selected entry is inserted as XML fragment into the query for the UIMA search engine. This query interface can also be used for other applications (e.g. see use case # 1). As input, the interface expects the following data:

- directory of indexed files,
- directory of CAS files,

- type system descriptor file,
- the XML description for the indexer, and
- a XML based description of possible values for features (a predefined list of values or a link to a file that contains the values).

The XML descriptions are used to create the list of possible entries for the user interface.

2.3. Comparision of Researchers' Workplaces

There are some relevant differences between the two types of researcher's workplaces:
Researchers in forensic medicine use information extracted from autopsy protocols to answer a broad spectrum of questions, mostly of a statistical nature. These may be repetitive questions that ask for developments over time periods, e.g. questions related to criminal statistics. Other questions may be related to possible changes of data in relation to specific events. A typical example for the latter type is the question if, and when how and under what circumstances, a new security device in automobiles influences injury patterns and death rates in car accidents. For reliable results this type of researchs needs to be based on a large and continually updated corpus of – ideally – protocols from throughout Germany (and additionally from Austria and Switzerland). The ease of formulation of ad hoc queries is a major requirement for this type of researchers' workplace.
In contrast, the epicrisis project has a single central research question: Have psychotherapeutic diagnoses and diagnostic argumentation and terminology changed in relation to the 'Wende' in East Germany and if so, in what respect? The corpus of about 1000 discharge summaries ranging from 1979 till 1999 is fixed (ie. it is a 'historical' corpus) and covers all resp. patients from this time period from the clinic under investigation. The research question demands for much more creativity and experimentation here in contrast to the more repetitive nature of some of the statistical questions in the evolving corpus of autopsy protocols.

3. Developing of Resources

The UIMA framework is also useful for extension of language resources used in NLP applications. We used different CAS consumers for generating of lists about missing information in our language resources. For example:

- terms not covered by POS Tagger,
- abbreviations not covered by structure tagger, and
- subfragments that could not be assigned to a fragment category.

The CAS consumers return lists with e.g. terms that are not covered by the POS Tagger. The developer can use this list as input for approaches in automatic extension of lexical resources or for the evaluation of tools.
Furthermore, we used another CAS consumer to create our training data for the classifier. The CAS consumer extracts all relevant information for the classifier. These data were annotated (by hand) with the correct classification.

4. Summary

In this paper, we presented two complex example systems for processing documents from the medical domain:

- evaluation of (diagnostic summaries of) epicrises and
- analyses of autopsy protocols.

Both architectures are build up on the UIMA framework.
We used different implementation interfaces (e.g. collection reader, analysis engine, and consumer) to create a complex application for supporting the domain experts in their analyses. The whole system is build up in a modular manner, and the integrated tools could also be used for processing other documents. The strict separation of resources and process methods simplifies the extension of domain specific resources.
The unique principle of using XML based descriptors supported fast implementation and configuration of tools for specific domains. The framework delivers different APIs for accessing and managing the information in the descriptors.
Besides UIMA, the application contains tools and integrates resources, like Lucene or GermaNet.
During the developing of the described applications, we also used UIMA for the creation of lexical resources (e.g. print-out of terms not covered by POS Tagger) and training data for the subfragment classifier.
When you have some experiences with UIMA, the process of creation of complex NLP applications is simple. For beginners, it is a little bit hard to understand the internal structure and mechanisms within UIMA. The tutorials and documentation are a good start to working with UIMA, but the description of internal classes, methods etc. are insufficient. Some processes are essential while developing UIMA based applications (e.g. merging of different type system descriptions, reading of descriptors and so on), a tutorial for developer that describes such aspects of the processes could be helpful.
All tools described in this paper are available from the authors upon request.

5. References

W. Finkler and G. Neumann. 1988. MORPHIX: a fast Realization of a classification-based Approach to Morphology. In H. Trost, editor, *Proceedings der 4. Österreichischen Artificial-Intelligence Tagung, Wiener Workshop Wissensbasierte Sprachverarbeitung*, pages 11–19, Berlin, August. Springer Verlag.

M. Kunze and D. Rösner. 2004. Issues in Exploiting GermaNet as a Resource in Real Applications. *LDV Forum*, 19(1):19–30. ISSN 0175-1336.

C. Kunze, 2001. *Lexikalisch-semantische Wortnetze*, pages 386–393. Spektrum, Akademischer Verlag, Heidelberg; Berlin.

D. Rösner and M. Kunze. 2002. An XML based Document Suite. In *Proceedings of Coling 2002*, pages 1278–1282, Taipei, Taiwan, August.

H. Wittig, W. Kuchheuser, M. Kunze, D. Krause, and D. Rösner. 2006. Erfahrungen bei der Nutzung des Computerprogramms UIMA als Werkzeug für die zielorientierte Suche in rechtsmedizinischen Dokumentensammlungen. In *Jahrestagung der Dt. Ges. f. Rechtsmedizin*, Innsbruck. Tagungsband.

Flexible UIMA Components for Information Retrieval Research

**Christof Müller*, Torsten Zesch*,
Mark-Christoph Müller*, Delphine Bernhard*, Kateryna Ignatova*,
Iryna Gurevych* and Max Mühlhäuser†**

* Ubiquitous Knowledge Processing Lab
† Telecooperation Division
Technische Universität Darmstadt, Germany
{mueller|zesch|chmark|delphine|ignatova|gurevych|max}@tk.informatik.tu-darmstadt.de

Abstract

In this paper, we present a suite of flexible UIMA-based components for information retrieval research which have been successfully used (and re-used) in several projects in different application domains. Implementing the whole system as UIMA components is beneficial for configuration management, component reuse, implementation costs, analysis and visualization.

1. Introduction

Existing information retrieval (IR) tools and frameworks like Apache Lucene[1] focus primarily on application building, where fast indexing and retrieval capabilities for large data collections are the driving factor. In IR *research* however, indexing and retrieval speed are not the (only) important factors. For rapidly performing successful IR experiments, it is crucial to

- support an easy integration, combination and configuration of new IR algorithms,
- manage vast numbers of runs of IR experiments resulting from different system configurations,
- provide evaluation methods for retrieval performance, and
- visualize the data, the retrieval process and the results.

Successful research in the field of IR and the development of new IR models involve constant changes to both the algorithm implementations and the preprocessing components, as well as the handling and visualization of (potentially huge amounts of) textual data for analysis purposes. A recent shift in IR towards semantics and NLP methods, as indicated by emerging search engines like Powerset, Hakia, Lexxe, and CognitionSearch,[2] shows the need for integrating more sophisticated preprocessing capabilities into IR frameworks.

In this paper, we present a suite of flexible UIMA-based components for IR research which have been successfully used (and re-used) in several projects in different application domains. The components are part of the DKPro (Darmstadt Knowledge Processing) repository[3], a collection of UIMA-based components for NLP tasks. The focus of this paper is on a description of the IR components in the DKPro repository. Section 2. briefly describes some requirements for research-oriented IR systems. Section 3. outlines a generic IR workflow and how it is realized by our DKPro components. Section 4. describes some of the projects in which they have been successfully applied.

[1] http://lucene.apache.org

[2] http://www.powerset.com, http://www.hakia.com, http://lexxe.com, http://cognitionsearch.com

[3] http://www.ukp.tu-darmstadt.de/software/repository

2. UIMA for Research-Oriented IR

From the above characterization of IR research, some clear requirements for the implementation of IR systems can be deduced, including the ability to process (potentially huge amounts of) unstructured natural language text, and to quickly configure different setups using varying combinations of (pre-)processing and retrieval components.

The modular nature of our components (as brought about by the UIMA architecture) simplifies within-project configuration management (i.e. different system configurations for different experiment runs), and minimizes the effort for cross-project employment (i.e. re-use) of components. The implementation of IR algorithms as UIMA components also offers the possibility to use the results of sophisticated NLP methods in the retrieval process without having to build custom indexing formats. Moreover it enables a thorough analysis of data and results as the visualization component can create combined views of the preprocessing and retrieval process.

3. IR Components in DKPro

The DKPro software repository is a collection of UIMA components for various NLP tasks. Among components for tasks in areas as diverse as topic segmentation, opinion mining, and community mining, it also contains flexible and efficient IR components.[4] The components cover all steps in what can be regarded as a generic IR workflow. Figure 1 provides an overview.

3.1. Collection Reading

This initial step relates to the basic task of importing the test collections (i.e. the documents and the related topics[5]) into the IR system. In UIMA, it is to be performed by instances of *reader* components. In different application domains, document collections come in vastly different formats, and it is in the reader (and only here) that the peculiarities of the respective formats are dealt with. The DKPro repository contains several readers for various formats. A

[4] Currently based on Lucene. Work for supporting further IR toolkits like e.g. Terrier is ongoing.

[5] The *topic* is a natural language statement of a user's information need which is used to create a *query* in an IR system.

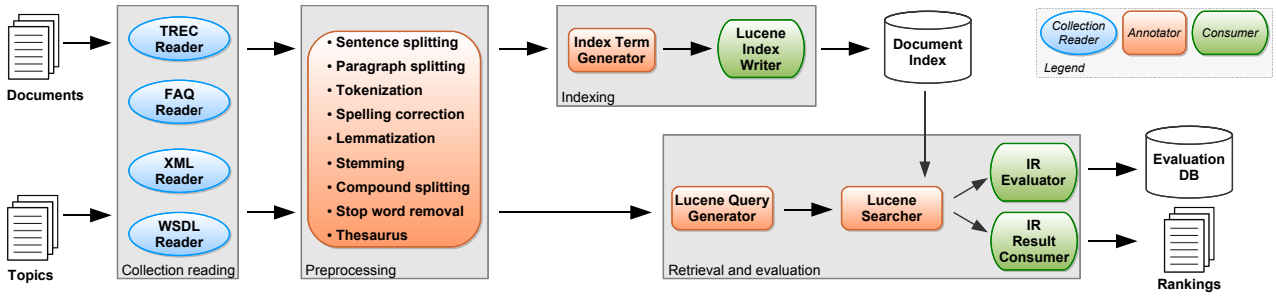


Figure 1: DKPro Components in a Generic IR Workflow

core functionality performed by all readers is the annotation of each processed collection item (i.e. document and topic) with a *DocumentMetaData* annotation. Apart from providing a unique ID for each item, this annotation also contains information like the title of a document or the ID of the collection it belongs to. This information is used in several downstream processing steps, including retrieval and visualization (cf. below). Some of the readers conserve collection-specific formatting information by adding annotations to the document. The `XMLReader` e.g. can be parameterized to create annotations for arbitrary XML elements found in a document. Other readers (like e.g. the `WSDLReader`) use more elaborate analysis to create more specific annotations.

3.2. Preprocessing

IR document collections normally consist of natural language text (but cf. Section 4.3.). Some preprocessing is commonly performed in order to (1) make explicit hidden structure within the texts (e.g. sentence or paragraph splitting or tokenization), (2) normalize their content (e.g. lemmatization, stemming, compound splitting, or spelling correction), or (3) add linguistic meta information (e.g. POS tagging, parsing, or stop word identification).
In UIMA, this is modelled as a task for *annotator* components, which add the new information and the normalized content in the form of annotations. More substantial modifications (like e.g. spelling correction in error-prone user-generated discourse, cf. Section 4.2.1.) can be implemented by having the annotator component actually *modify* the underlying content.[6] This method is used by `SpellingCorrector`. For numerous preprocessing tasks, powerful stand-alone tools are already available in the NLP research community. Where possible, the components in DKPro utilize these. Our `POSTagger` and `Lemmatizer` e.g. are wrappers for the TreeTagger (Schmid, 1994). In a broader sense, preprocessing can also be understood to comprise less generic and more application-specific tasks. For IR, one of these tasks is query expansion, in which related terms are added to the query text. The DKPro repository contains a component which adds related terms (e.g. based on various types of semantic relatedness (Gurevych, 2005)) in the form of annotations. Keeping the original query text and the expansion terms apart by adding the latter in the form of annotations is particularly useful because it allows explicit control over the use of the query expansion feature by downstream components, e.g. for assigning a different weight to expansion terms in the query generation and retrieval process.

3.3. Indexing

The generation of a document and query index is a prerequisite for efficient retrieval. The scope and nature of the index can vary for different collections and different applications. In some settings, all document and query tokens (presumably excluding stop words) have to be indexed, while in other settings only certain parts might be relevant. In DKPro, the `IndexTermGenerator` annotator is responsible for identifying terms to be indexed. Provided that the respective preprocessing has been performed earlier, it can create index terms of entire tokens, lemmata, stems, and/or other arbitrary annotation elements. If the `POSTagger` annotator was applied to the documents and queries to be indexed, index term generation can also be constrained by POS information. The resulting index terms are then written by a *consumer* component to an index file in the format required by the IR engine to be used. Up to now, the DKPro repository contains a `LuceneIndexWriter` and some project specific components, which are described in Section 4.

3.4. Retrieval, Evaluation, and Visualization

In the retrieval step, the previously generated document and query indices and a set of parameter settings (e.g. threshold values to be used) are employed to create actual IR *runs*. A run consists of the application of all queries to a document collection and yields a quantitative evaluation of the overall effectiveness of the applied (pre-)processing pipeline, parameter settings, and retrieval engine for a particular document collection. The retrieval step is broken down into query generation, search, evaluation, and (optionally) visualization. For each of the first three steps, there is a dedicated component in DKPro. The first two (`LuceneQueryGenerator` and `LuceneSearcher`) are particular to the retrieval engine to be used. The third one (`IREvaluator`) is a general-purpose IR evaluation component which computes common IR evaluation measures by wrapping the `trec_eval`[7] tool, but which also offers other evaluation measures like Spearman's rank cor-

[6] Technically, this is implemented by having the annotator create a new *view* containing the altered content.

[7] http://trec.nist.gov/trec_eval

relation coefficient. The `IREvaluator` can optionally store the evaluation results in a relational database. The stored results include not only the overall retrieval results, but also detailed information about individual topics and documents.

In contrast to visualization of IR results in an end-user oriented setting[8], IR *research* is best supported by allowing researchers to trace individual topics and documents through the entire retrieval run, e.g. for error or general performance analysis. For the DKPro IR components, this is supported by a component which allows result visualization and browsing. As browsing is inherently interactive, it is not naturally implemented as a (pipeline-oriented) UIMA component. Therefore, result browsing is implemented as a servlet-based web application which reads evaluation information from the database (created by the `IREvaluator`) and displays it in a web browser. The analysis process which is necessary for understanding and improving the IR model requires data browsing on different information levels:

- *run level:* configuration parameters and overall results;
- *query level:* evaluation results of each query (for selected runs);
- *document level:* relevance scores and relevance assessments of each document (for a certain query and selected runs);
- *process level:* visualization of the retrieval process of a document (for a certain query and selected runs).

The component uses the original documents and topics, the output of the retrieval process and the relevance assessments. In order to provide detailed information on the process level, the component offers the possibility to rerun the processing pipeline for a selected document and query, adding a special consumer to the pipeline which creates an HTML document with preprocessing and retrieval information. In this step, topic and document are passed simultaneously through the pipeline (in the same *CAS* object, but in two separated *views*) and the retrieval components can add additional information that helps to understand the details of the retrieval process.

Especially for research purposes, the tight coupling of preprocessing and retrieval can be beneficial when developing new IR algorithms. Instead of investing time in (re-)adjusting or implementing new indexing formats, the retrieval components can (temporarily) work directly on the annotations created by the preprocessing components.

3.5. Configuration Management

As mentioned above, IR research aims at finding new and improved algorithms and optimized settings for IR parameters. Also, different configurations for preprocessing steps yield multiple indices. In practice, therefore, the processing workflow described above has to be executed very often. The DKPro IR components are complemented with a number of helper components for batch execution of experimental runs. The helper classes provide functionality for programmatically configuring and executing collection processing engines. The configurations can be stored in a relational database which enables the visualization and comparison of IR results in the visualization component.

4. DKPro IR Components in Use

In this section, we give a detailed account of how some of the components in the DKPro repository are employed in several projects in different application domains. Where available, experimental results are also reported.

4.1. Electronic Career Guidance

The task of electronic career guidance is to support school leavers in their search for a profession or a vocational training to take up. In (Gurevych et al., 2007), we describe work in which electronic career guidance is modelled as an IR task. Vocational trainings are represented by documents which were automatically extracted from BERUFEnet, a database created by the German Federal Labour Office. Topics are short essays collected from students in which they describe in their own words what they would like their future job to be like. One special challenge of this task is the large *vocabulary gap* between the language of the (expert-authored) documents from the database and the language of the students. The term *vocabulary gap* relates to the fact that people with different backgrounds or different levels of expertise use (sometimes strikingly) different vocabularies when describing similar things. String-based IR approaches (as represented e.g. by Lucene) are not able to adequately handle this phenomenon. The best results reported in (Gurevych et al., 2007) were therefore produced by a *semantic* information retrieval component, which scores the similarity of documents and queries on the basis of their semantic relatedness. The components come as the annotators `RelatednessScorer` and `SemanticSearcher` and the consumer `SemanticIndexWriter`, and fit seamlessly into the pipeline of the other DKPro components.

4.2. Question Answering

Question Answering (QA) systems aim at giving precise answers to natural language questions. The architecture of traditional QA systems is therefore more complex than IR systems, since they have to include a component which extracts answers from documents. The answer extraction problem can be avoided by leveraging the wealth of information available on the Web in the form of Frequently Asked Questions (FAQ) pages and question-answer services such as Yahoo!Answers[9] or WikiAnswers[10]. When answers are retrieved from question-answer repositories, the QA task can be redefined as an IR task where topics are natural language questions and documents are the question-answer pairs. There are actually two ways to address this task: by identifying paraphrases of the input question in a question-answer repository (Section 4.2.1.), or by retrieving the most similar question-answer pair from an FAQ (Section 4.2.2.).

[8] http://people.lis.uiuc.edu/~twidale/irinterfaces/2classics.html, http://people.ischool.berkeley.edu/~hearst/tb-overview.html

[9] http://answers.yahoo.com

[10] http://wiki.answers.com

4.2.1. Question Paraphrase Identification

The objective of this task is to retrieve those questions in the question-answer repository which are most similar to the input question. A first difficulty lies in the fact that most online question-answer services record real user questions, which may be ill-formulated or may contain spelling errors. Prior to indexing, therefore, we apply the `SpellingCorrector` annotator. In order to perform the matching of an input question to the most similar question in a question-answer pair, we have implemented several text similarity measures based on the work by Tomuro & Lytinen (2004) and Zhao et al. (2007), among others. These measures include matching coefficient, word overlap coefficient, edit distance and term vector cosine similarity. Two UIMA annotators are in charge of computing the similarity values and ranking the results for each input question. These annotators replace the `LuceneQueryGenerator` and `LuceneSearcher` components in the generic retrieval step described above. Since the similarity measure to be used in a given experiment is a component's parameter, the available measures can be easily tested and new text similarity measures can be conveniently added.

4.2.2. FAQ Mining

Based on the work by Jijkoun & de Rijke (2005), we aim at answering users' questions by retrieving relevant question-answer pairs found in FAQ pages. Within this task, a document is considered as a collection of several fields: question and answer of a question-answer pair, title of the corresponding FAQ page, and the full text of the FAQ page. In order to keep this document-specific information, annotations are added to the document in the collection reading step by means of a parameterized `XMLReader` annotator. Further, the preprocessing stage allows to normalize the content by performing lemmatization and stemming, which are required for later building both stemmed and lemmatized indices. Additional information, such as the document's language and contained stopwords, is also added at this point. The `IndexTermGenerator` allows to index different fields, e.g. a non-stemmed question keeping stopwords, a stemmed answer without stopwords, etc. Easy combination of annotation components and flexibility during indexing make it possible to easily evaluate different system configurations as described by Jijkoun & de Rijke (2005). Our current baseline system reimplements several of their models with comparable results. E.g., the performance of the baseline model for the retrieval of the so called 'adequate' and 'material' answers is 45% in the top 10 results.

4.3. Web Service Retrieval

Web service retrieval is the task of retrieving from a repository of web services those services that provide a particular functionality. When cast as an IR task, topics are descriptions of required functionalities, while the services to be retrieved are represented as semi-structured documents. These documents have been created by crawling known web service repositories and processing the collected WSDL files. Within each WSDL file, the `WSDLReader` identifies and analyzes operation names and operation signatures (i.e. names and types of operation parameters) and creates a textual representation to be processed using the standard IR workflow.

5. Conclusion

In this paper, we presented a suite of flexible UIMA-based components for research-oriented information retrieval which have been successfully used (and re-used) in several projects in different application domains. The usage of UIMA as framework not only shows benefits for the preprocessing components, but also for the actual retrieval components. Apart from well-known features of UIMA like configuration management, component reuse, and replicating processing pipelines, the tight coupling inside UIMA of the preprocessing and the actual retrieval process offers possibilities for fast prototyping of new IR algorithms by directly using UIMA annotations instead of developing custom indexing formats. It also extends analysis and visualization capabilities by offering combined views of preprocessing and retrieval on different levels of granularity.

The described IR and preprocessing components are part of the DKPro repository and (with some exceptions) will be made available to interested researchers.

Acknowledgements Parts of this work were carried out in two projects funded by the German Research Foundation (DFG): "Semantic Information Retrieval from Texts in the Example Domain Electronic Career Guidance" (grant GU 798/1-2), and "Mining Lexical-Semantic Knowledge from Dynamic and Linguistic Sources and Integration into Question Answering for Discourse-Based Knowledge Acquisition in eLearning" (grant GU 798/3-1).

References

Gurevych, Iryna (2005). Using the structure of a conceptual network in computing semantic relatedness. In *Proceedings of the 2nd International Joint Conference on Natural Language Processing (IJCNLP'2005)*. Jeju Island, Republic of Korea.

Gurevych, Iryna, Christof Müller & Torsten Zesch (2007). What to be? - Electronic career guidance based on semantic relatedness. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics*, pp. 1032–1039. Prague, Czech Republic.

Jijkoun, Valentin & Maarten de Rijke (2005). Retrieving answers from frequently asked questions pages on the web. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pp. 76–83. New York, NY, USA: ACM.

Schmid, Helmut (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the International Conference on New Methods in Language Processing (NeMLaP). Manchester, U.K., 14–16 September 1994*.

Tomuro, Noriko & Steven Lytinen (2004). Retrieval Models and Q&A Learning with FAQ Files. In Mark T. Maybury (Ed.), *New Directions in Question Answering*, pp. 183–194. AAAI Press.

Zhao, Shiqi, Ming Zhou & Ting Liu (2007). Learning Question Paraphrases for QA from Encarta Logs. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pp. 1795–1801. Hyderabad, India.

Integrating a Natural Language Message Pre-Processor with UIMA

Eric Nyberg, Eric Riebling, Richard C. Wang and Robert Frederking
Language Technologies Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
E-mail: {ehn,er1k,rcwang,ref}@cs.cmu.edu

Abstract

This paper describes the use of the Unstructured Information Management Architecture (UIMA) to integrate a set of natural language processing (NLP) tools in the RADAR system. The challenge was to define a common data model and a set of component interfaces for these tools, so that they could be integrated into a single system. The integrated system is used to pre-process each email arriving in the RADAR user's IMAP store. We present a UIMA collection processing engine for RADAR, including a common type system for text analysis results and annotators for each of the NLP tools. The paper also includes an analysis of system performance and a discussion of the lessons learned through use the of UIMA for this integration task.

1. Introduction

This paper describes the use of the Unstructured Information Management Architecture (UIMA) to integrate a set of natural language processing (NLP) components in the RADAR system. The RADAR (Reflective Agent with Distributed Adaptive Reasoning) system is comprised of a set of intelligent agents that assist the user with routine tasks such as email and scheduling [1]. Its initial test domain is conference planning.

RADAR agents include a Calendar Agent, which notices requests for appointments and helps the user to fit them into his or her calendar, and a Briefing Assistant, which extracts important parts of documents such as meeting minutes to provide automatic briefings (Kumar et al. 2007). These two RADAR agents assume that email messages have been pre-processed with text analysis software to recognize important ranges of text (for example, a request for a meeting. or an action item). This pre-processing is accomplished by a large set of both pre-existing, and project-developed, NLP tools.

The architectural challenge was to define a common data model and a set of component interfaces for these tools, so that they could be integrated into a single system. The integrated system is used to pre-process each email arriving in the RADAR user's IMAP store; the output of the NLP tools is stored in the form of *standoff annotations* - data structures derived from text analysis which are stored separately from the text itself. The UIMA framework is used to define a common type system for text analysis results. An annotator wrapper was written for each NLP component in the pre-processor. Each annotator wrapper is responsible for providing input to an NLP component in its native format, and converting the output of the component back into standoff annotations. The annotator wrappers were integrated into a single collection processing engine (CPE). An object referred to as the common analysis structure (CAS) is created for each input message; this structure includes storage for the original text, as well as storage and an index for each annotation type. Components produce instances of annotation types, which are stored in the CAS as it is passed from component to component. The RADAR CPE also includes Collection Reader and CAS Consumer components, which are responsible for reading and writing email messages and their annotations to and from the persistent database storage. The full CPE is depicted in Figure 1.

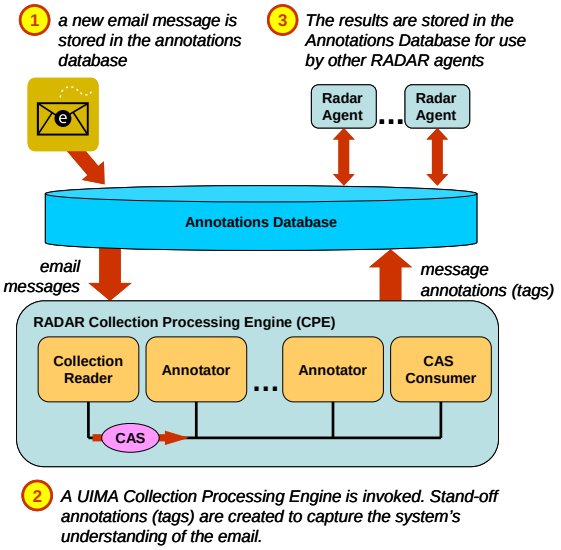


Figure 1: The RADAR Collection Processing Engine

The following sections provide more detail regarding the design and implementation of the RADAR CPE. Section 2 describes the NLP components that were integrated. Section 3 describes the process that was followed to integrate the different modules into the CPE. Section 4

[1] http://radar.cs.cmu.edu/

provides an analysis of system performance and discusses the lessons learned when using UIMA to integrate our suite of NLP components. Section 5 concludes with some suggestions for future work.

2. Annotators & Related Components

This section provides a list of the annotators in the RADAR CPE, including a brief description of each annotator's subtask, the component that it wraps, its input annotation types (if any) and its output annotation types. The annotators are described in the order that they are run for each input email.

2.1 Email Opening Annotator

The Email Opening Annotator identifies the span of text in the message that contains the opening or greeting, e.g. "Dear Blake,". This annotator is implemented by a hand-coded set of surface patterns written in the Mixup language provided by the MinorThird toolkit (Cohen, 2004). It is the first component to process the email text, and it requires no prior input annotations. It outputs instances of a single, simple annotation type with no attributes: EMAIL_HEADING.

2.2 Typo Annotator

The Typo Annotator identifies spans of text in the message that are likely to be misspelled words, and lists alternatives. This annotator is implemented via the open source Jazzy spell checker[2], and it requires no prior input annotations. It outputs instances of a single annotation type called TypoAnnotation, which has two attributes: Typo, the token string containing the spelling error, and Corrections, an array of strings containing proposed corrections, sorted in order of increasing string edit distance from the original token string.

2.3 Connexor Annotator

The Connexor Annotator uses the Connexor parser[3] to mark spans of text denoting sentences, tokens, parts of speech and lemmas for tokens, and functional dependency grammar parses for sentences. This annotator requires no prior input annotations, and outputs instances of three annotation types: ConnexorSentence, ConexorToken (including attributes POS and Lemma), and ConnexorParse (which includes an attribute, Value, which contains a string representation of the Connexor parse analysis).

2.4 Temporal Expression Annotator

The Temporal Expression Annotator identifies spans of text containing temporal expressions such as "next Thursday". This annotator is implemented using MinorThird compiled annotation rules. It outputs a single annotation type, TimeExpression, which includes an attribute, AnchoredValue, containing a canonical time expression for the precise date and time indicated by the surface text, in a format based on ISO8601 (Han et. Al, 2006). Note that a relative time expression like "next Thursday" can only be resolved to an AnchoredValue by calculating its calendar position relative to the time that the email was sent.

2.5 Functional Structure Annotator

The Functional Structure (FS) Annotator processes the information provided by the prior annotations to produce a grammatical functional structure or *f-structure* for each sentence. Each f-structure contains information about the grammatical functions (or roles) expressed in the sentence, such as subject, object, indirect object, etc. The FS Annotator is implemented… (need something from Eric R. here). This annotator requires input annotations EMAIL_HEADING, CXR_PARSE, and TEMPORAL_EXPRESSION, and produces a single output annotation, F_STRUCTURE.

2.6 General Frame (GFrame) Annotator

The General Frame (GFrame) Annotator processes the F_STRUCTURE annotations to produce a general (that is, not domain-specific) semantic frame representation. The GFrame Annotator uses the Mapper component from the KANTOO machine translation system (a general transformation engine for feature structures) (Nyberg et al., 2002), and a set of general (non-domain-specific) interpretation rules. The GFrame annotator outputs a single annotation type, GFRAME.

2.7 Domain Frame (DFrame) Annotator

The Domain Frame (DFrame) Annotator processes the F_STRUCTURE and Gframe annotations for each sentence to produce a domain-specific semantic frame representation, for the conference scheduling domain. The DFrame annotator is implemented using the KANTOO Mapper component and a set of domain-specific interpretation rules. The DFrame annotator outputs a single annotation type, DFRAME.

2.8 Person Name Annotator

The Person Name Annotator identifies possible person names using a Hidden Markov Model trained with the MinorThird toolkit. Outputs a single annotation type, PERSON_NAME_VPHMM.

2.9 RADAR Person Annotator

The RADAR Person Annotator identifies names of known individuals, e.g. "Blake Randal", using a Hidden Markov Model trained with the MinorThird toolkit[4]. Outputs a single annotation type, RADAR_PERSON.

2.10 SCONE Implicit Feature Annotator

The SCONE Implicit Feature Annotator looks up information from the SCONE Knowledge Base for terms in email. Outputs a single annotation type,

[2] http://jazzy.sourceforge.net/
[3] http://www.connexor.eu/
[4] http://minorthird.sourceforge.net/

IMPLICIT_FEATURE.

2.11 SCONE Semantic Annotator

The SCONE Semantic Annotator connects and communicates with a network-based SCONE and SconeGrammar server, retrieving semantic and/or element data relations that SCONE has for a given text. It outputs three annotations: DISCOURSE_STRUCTURE (attributes: SEMANTIC_VALUE, STRUCTURE_SPEC_TYPE), BRIEFING_CONCEPT (attribute: VALUE), and BRIEFING_HEURISTIC (attribute: VALUE).

2.12 Vendor XML Annotator

The Vendor XML Annotator produces a custom XML representation for conference vendor order confirmation and vendor quote e-mails (as for, e.g., food providers). This annotator outputs a single annotation type, VENDOR_XML (with attribute VALUE).

2.13 Space Request Annotator

The Space Request Annotator extracts information from e-mails requesting physical space (office/room/lab space, etc.) and produces a custom XML representation. This annotator outputs a single annotation type, SPACE_XML (with attribute SPACE_REQUEST_VALUE).

2.14 Task Annotator

The Task Annotator identifies overall tasks for the RADAR agents, where tasks are pre-defined in the conference scheduling domain. This annotator outputs a single annotation type, TASK (with attributes TASK_TEMPLATE and TASK_CATEGORY).

2.15 Briefing Annotator

The Briefing Annotator uses 6 sets of trained Minorthird models to guess the likelihood of each email being one of six types of briefing request. (These are used to generate a briefing email to the conference organizer's supervisor.) Returns a single annotation whose value is a string containing the subset of the 6 types deemed possible, as comma separated values: "attendance", "av", "food", "general", "reschedule", "room". This annotator produces a single annotation type, BRIEFING (with attribute BRIEFING_CATEGORIES).

3. RADAR CPE Integration

The components listed in Section 2 were integrated into a single Collection Processing Engine (CPE) for RADAR. In addition to the annotator listed above, two additional UIMA components were required: a) a Collection Reader to read incoming emails from the Annotations Database and convert them into run-time CAS objects, and b) a CAS Consumer to store the annotated CAS objects back into the Annotations Database (see Figure 1).

The annotators in the RADAR CPE include components which are written in Java, and which integrate directly into the UIMA run-time (which is also written in Java). The exceptions are legacy components (Connexor parser, KANTOO Mapper, and SCONE) which are deployed as network services; for these components, the annotator implementation consists of a UIMA wrapper which maintains a network connection to the appropriate network server and takes care of translation to/from the CAS representation when the remote service is used to process the email text.

4. Evaluation

The use of UIMA in deploying the RADAR NLP component architecture was evaluated along three dimensions: overall cost of adoption, measured in programmer effort; run-time performance of the completed system, measured in seconds; and robustness of the resulting implementation, which is discussed in terms of general observations about the system after several months of use.

4.1 Overall Cost of Adoption

The RADAR CPE was integrated by programmer who had already completed a UIMA tutorial and one prior UIMA deployment. The programmer was able to wrap and integrate the 15 annotators listed in Section 2 in about 6 weeks of full-time work. This work was greatly facilitated by the UIMA framework, which allowed the initial deployment to take place very quickly. Remaining concerns about use of UIMA in the longer term are related to robustness of the communication with remote services; see Section 4.4 for further discussion.

4.2 Run-Time Performance

The run-time speed of the annotators (measured over 250 sample messages) is shown in Table 1.

%	Time(ms)	s/doc	Annotator
65.27	5310311	21.24	DFrame
24.60	2001145	8.00	GFrame
2.99	243653	0.97	RADAR Person
2.65	215952	0.86	SCONE Sem.
1.50	122228	0.49	Temporal Expr.
1.03	83563	0.33	Person Name
0.71	57742	0.23	SCONE Impl.
0.54	44187	0.18	F-Structure
0.18	14889	0.06	Email Opening
0.17	13513	0.05	SpaceRequest
0.17	13445	0.05	Conexor
0.07	5835	0.02	Typo
0.06	4746	0.02	CAS Consumer
0.03	2725	0.01	Collection Reader
0.03	2415	0.01	Task
100.00	8136349	32.55	Entire Pipeline

Table 1: Annotator Processing Time, 250 messages

Most of the annotators required less than a second per document, on average. The most time-consuming

annotators are the DFrame and GFrame annotators, which evaluate two different sets of semantic interpretation rules at run time to transform the original functional structure into a final frame output.

4.3 Accuracy

We also evaluated the accuracy of some of the annotators in the RADAR CPE through human evaluation of the output. We randomly selected 50 messages and evaluated whether or not each annotation was correct. The precision (the percentage of annotations that were correct) are shown in Table 2. For comparison, the number of structures which were correct but is also shown.

Annotator	% Correct	% Partly Correct
Vendor Order Annotator	100%	--
Task Annotator	73%	77%
Person Name Annotator	76%	85%
Space Request Annotator	64%	79%

Table 2: Annotator Precision

4.4 Transparency and Robustness

Although UIMA provided excellent support for quickly integrating different NLP components, the most straightforward implementation of legacy components as networked services was not completely robust. The first implementation used direct TCP socket connections to remote server machines, and parsed the low-level string protocols provided by each service. There was no support for process logging or server restart in our implementation, so it became time-consuming to debug system failures when a networked component was involved. For example, if a new, buggy set of KANTOO Mapper rules was deployed for the KANTOO DFrame server, the DFrame Annotator might experience an error state when calling out to the server; the only means of debugging such a failure at present is to manually inspect the log messages on the KANTOO server, and to restart the service manually as required.

5. Conclusion and Future Work

Our adoption of UIMA for integrating the RADAR CPE was an overall success. In only six weeks a single programmer was able to integrate 15 different NLP components into a single pre-processor for incoming email messages. The system includes native Java components as well as remote services integrated via Java wrappers, and reads and writes annotated email messages from a persistent relational store.

In future work, we intend to address the robustness issues with better design for remote NLP services. It would be preferable to integrate such services via a common standard that is already well-supported in Java (for example, WSDL). This would simplify the integration of remote services in RADAR while placing responsibility for standards compliance on the service remote side, rather than the client side. In retrospect, it would have been a cleaner approach to write web service wrappers for each of the remote NLP components before integrating them into UIMA, but this would have taken additional programmer time before a working prototype would have been achieved.

Another possible approach is to deploy third-party annotators as brokered services. This would promote the migration of code for handling native service protocol messages from the UIMA client pipeline out to the remote service. Such a design would provide cleaner separation of responsibility between the service and client, since it does not require the UIMA client pipeline to incorporate low-level details of the third-party service protocol.

In order to improve the transparency and robustness of the system, better logging is required, especially with respect to the operations of remote services that are integrated into the pipeline. We are beginning to investigate the new UIMA-EE framework as a means to achieve better logging in the overall pipeline. Eventually, we hope to build a predictive model of remote service performance that will allow us to dynamically allocate back-end processing nodes for optimal pipeline throughput.

6. Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. We also thank the anonymous reviewers for their helpful comments on an earlier draft of this paper.

7. References

Cohen, William W. (2004). Minorthird: Methods for Identifying Names and Ontological Relations in Text using Heuristics for Inducing Regularities from Data, http://minorthird.sourceforge.net.

Han, Benjamin, Donna Gates and Lori Levin (2006).Understanding temporal expressions in emails. *Proceedings of the Human Language Technology Conference*, Association for Computational Linguistics.

Kumar, M. et al. (2007). Summarizing Non-textual Events with a 'Briefing' Focus. *Proceedings of RIAO*, Centre De Hautes Etudes Internationales D'Informatique Documentaire.

Nyberg, E., T. Mitamura, K. Baker, D. Svoboda, B. Peterson and J. Williams (2002). "Deriving Semantic Knowledge from Descriptive Texts using an MT System", *Proceedings of AMTA 2002*.

Yang, Y. et al. (2005). Robustness of Adaptive Filtering Methods in a Cross-Benchmark Evaluation. *Proceedings of ACM SIGIR*, 98–105. ACM Press.

ClearTK: A UIMA Toolkit for Statistical Natural Language Processing

Philip V. Ogren, Philipp G. Wetzler, Steven J. Bethard
Center for Computational Language and Education Research
University of Colorado at Boulder
Boulder, CO USA
E-mail: philip@ogren.info, Philipp.Wetzler@colorado.edu, Steven.Bethard@colorado.edu

Abstract

This paper describes a toolkit, ClearTK, which was developed at the Center for Computational Language and Education Research at the University of Colorado at Boulder. ClearTK is a framework that supports statistical natural language processing and implements a number of tasks such as part-of-speech tagging, named entity identification, and semantic role labelling. ClearTK is written in the Java programming language on top of Apache UIMA. The core of this framework contains a flexible and extensible feature extraction library and a set of interfaces to several popular machine learning libraries including OpenNLP's MaxEnt, Mallet's Conditional Random Fields, LibSVM, SVM^{light}, and Weka. We demonstrate that ClearTK can be used to achieve state-of-the-art performance on biomedical part-of-speech tagging.

1. Introduction

The Center for Computational Language and Education Research (CLEAR)[1] has had a strong presence in the Natural Language Processing (NLP) community for over a decade publishing dozens of high quality research papers. Despite this research success the center has not produced software that has been widely used outside of it other than the speech recognition software Sonic[2]. Software written for research purposes is not necessarily easy to install, compile, use, debug, and extend. As such, we undertook to create a framework for statistical NLP that would be a foundation for future software development efforts that would encourage wide distribution by being well documented, easily compiled, designed for reusability and extensibility, and extensively tested. The software we created, ClearTK[3], is a framework that supports statistical NLP by providing a rich feature extraction library, interfaces to popular machine learning libraries, and a set of components for tackling NLP tasks such as tokenization, part-of-speech tagging, syntactic parsing, named entity identification, and semantic role labeling. ClearTK is available with source code under a research-use only license[4]. ClearTK currently consists of 175 classes and over 20 unit test suites containing nearly 1300 assertions.

ClearTK is built on top of the increasingly popular Unstructured Information Management Architecture (UIMA) described in (Ferrucci and Lally 2004). Briefly, UIMA provides a set of interfaces for defining components for analyzing unstructured information and provides infrastructure for creating, configuring, running, debugging, and visualizing these components. In the context of ClearTK, we are focused on UIMA's ability to process textual data. All components are organized around a *type system* which defines the structure of the annotations that can be associated with each document. This information is instantiated in a data structure called the Common Analysis Structure (CAS). There is one CAS per document that all components that act on a document can access and update. Every annotation that is created is posted to the CAS which is then made available for other UIMA components to use and modify. Here is a short list of the most important kinds of components:

- Collection Reader – a component that reads in documents and initializes the CAS with any available annotation information.
- Analysis Engine – a component that performs analysis on the document and adds annotations to the CAS or modifies existing ones.
- CAS Consumer – a component that processes the resulting CAS data (e.g. write annotations to a database or a file)
- Collection Processing Engine (CPE) – an aggregate component that defines a pipeline that typically consists of one collection reader, a sequence of analysis engines, and one or more CAS consumers.

We chose UIMA for a wide variety of reasons including but not limited to its open source license, wide spread community adoption, strong developer community, elegant APIs that encourage reusability and interoperability, helpful development tools, and extensive documentation. While UIMA provides a solid foundation for processing text, it does not directly support statistical NLP. ClearTK provides a framework for creating UIMA components that use statistical learning as the foundation for decision making and annotation creation.

In the following sections we describe how statistical NLP is performed with ClearTK (section 2), give an overview of some of the NLP tasks ClearTK supports (section 3), and give results on a part-of-speech tagger written using ClearTK (section 4).

[1] http://clear.colorado.edu, formerly named the Center for Spoken Language Research
[2] http://clear.colorado.edu/Sonic/
[3] http://clear.colorado.edu/ClearTK/
[4] https://www.cusys.edu/techtransfer/downloads/Bulletin-ResearchLicenses.pdf

2. Statistical NLP in ClearTK

ClearTK was designed and implemented with special attention given to creating reusable and flexible code for performing statistical NLP. As such, the library provides classes that facilitate extracting features, generating training data, building classifiers, and classifying annotations. ClearTK introduces *classifier annotators* which are analysis engines that perform feature extraction, classify the extracted features using a machine learning model, and interpret the results of the classification by e.g. labeling annotations or creating new annotations. A classifier annotator can also be run in training mode in which it performs feature extraction and then writes out training data which is then used for building a model.

2.1. Feature Extraction

The ClearTK feature extraction library is highly configurable and easily extensible. Each *feature extractor* produces a feature or set of features for a given annotation (or pair or collection of annotations as the feature extractor requires) for the purpose of characterizing the annotation in a machine learning context. A feature in ClearTK is a simple object that contains a value (i.e. a string, boolean, integer, or float value), a name, and a context that describes how the feature value was extracted. Most features are created by querying the CAS for information about existing annotations. Because features are typically many in number, short lived, and dynamic in nature (i.e. features often derive from previous classifications), they are not represented in the CAS but rather as simple Java objects.

The *spanned text extractor* is a very simple example of a feature extractor that takes an annotation and returns a feature corresponding to the covered text of that annotation. The *type path extractor* is a slightly more complicated feature extractor that extracts features based on a path that describes a location of a value defined by the type system with respect to the annotation type being examined. For example, Figure 1 shows a simple hypothetical type system. A type path extractor initialized with the path `headword/partOfSpeech` can extract features corresponding to the part-of-speech of the head word of examined constituents.

A much more sophisticated feature extractor is the *window feature extractor*. It operates in conjunction with

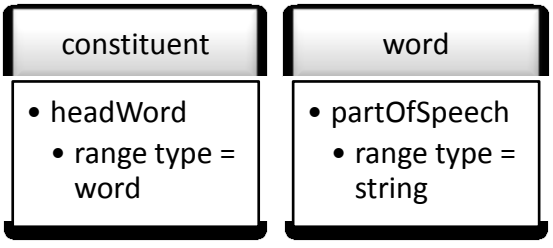


Figure 1: A hypothetical type system that contains the path `headWord/partOfSpeech`.

a simple feature extractor (such as the *spanned text extractor* or *type path extractor*) and extracts features over some numerically bounded and oriented range of annotations (e.g. five token to the left) relative to a focus annotation (e.g. a named entity annotation or syntactic constituent) that are within some window annotation (e.g. a sentence or paragraph annotation.) The "featured" annotations, the focus annotation and the window annotation are all configurable with respect to the type system. This allows the window feature extractor to be used in a wide array of contexts. The window feature extractor also handles boundary conditions such that e.g. words appearing outside the sentence that the focus annotation appears in would be considered as "out-of-bounds." This feature extractor allows one to extract features such as:

- The three part-of-speech tags to the left a word.
- The part-of-speech tag of the head word of constituents to the right of an annotation.
- The identifiers of recognized concepts to the left an annotation.
- The penultimate word of a named entity mention annotation.
- The last three letters of the first two words of a named entity mention annotation.
- The lengths of the previous 10 sentences.

A feature extractor is any class that generates feature objects. For example, the window extractor has a method that takes a focus annotation (e.g. a word) and a window annotation (e.g. a sentence) and produces features relative to these two annotations according to how the feature extractor was initialized. Many feature extractors implement an interface that designate them as *simple feature extractors* which allows them to be used by more complicated feature extractors such as the window extractor. It is the responsibility of the classifier annotator to know how to initialize feature extractors and how to call them. Table 1 lists some of the feature extractors provided by ClearTK.

Similar to feature extractors, *feature proliferators* create features suitable for characterizing an annotation in a machine learning context by taking as input features created by a feature extractor and creating new features. An example of a feature proliferator is *lower case proliferator* which takes features created by e.g. the spanned text extractor and creates a feature that contains the lower cased value of the input feature. Another example of a feature proliferator is the *numeric type proliferator* which examines a feature value and determines if it is a string that contains some digits, contains only digits, looks like a year, looks like a Roman numeral, etc. Table 2 lists some of the feature proliferators provided by ClearTK.

2.2. Classification

After a classifier annotator extracts features it then

Extractor	features extracted derived from…
spanned text	spanned text of an annotation
distance	the "distance" between two annotations
type path	value defined by a path through the type system
syntactic path	the syntactic path from one syntactic constituent to another
white space	existence of whitespace before or after an annotation
gazetteer	entries from a gazetteer found in the text
head word	headword of syntactic constituents
relative position	the relative position of two annotations (e.g. before, overlap left, etc.)
window	some window of annotations in or around the focus annotation
n-gram	generates n-gram style features relative to some window of annotations in or around the focus annotation
bag	generates bag-of-words style features

Table 1: Feature extractors provided by ClearTK

classifies the features and interprets the results. Classification is performed by a *classifier* which is a wrapper class that handles the details of providing a set of features to a machine learning model so that it can classify them and return a result. Currently there are classifier implementations for LibSVM[5] described in (Chang and Lin 2001), Mallet Conditional Random Fields (CRF)[6] described in (McCallum 2002), OpenNLP MaxEnt[7], SVM^{light}[8] described in (Joachims 1999) and Weka[9] described in (Witten and Frank 2005)[10]. Because the classifier wrappers handle the details of passing features to and results from a machine learning library, the developer of a classifier annotator does not have to worry about the low level details of working with each library and can focus on the NLP task itself. An additional benefit of the classifier abstraction is that it allows one to swap out one machine learning library for another and compare and contrast the fitness of a machine learning library for a particular task (see the results section below).

Each of these machine learning libraries are implemented in Java except for SVM^{light} which is implemented in the C programming language. As is common for machine learning libraries, the code for training is much more complicated than the code that performs classification. As such, we re-implemented the classifier in Java such that it can directly classify using the models generated by SVM^{light}. We do not support all of the many variations of SVM^{light} (e.g. SVM^{struct}, SVM^{cfg}, and SVM^{hmm}) or all of the various kernels that are available in SVM^{light}.

[5] http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[6] http://mallet.cs.umass.edu/

[7] http://maxent.sourceforge.net/

[8] http://svmlight.joachims.org/

[9] http://www.cs.waikato.ac.nz/ml/weka/

[10] There are licensing incompatibilities between ClearTK and Weka and SVM^{light} depending on how ClearTK is to be used. Weka and SVM^{light} are not distributed with ClearTK.

Proliferator	Description
capital type	all uppercase, all lowercase, initial uppercase, mixed case
numeric type	all digits, 4 digit year, some digits, roman numeral
character n-grams	character prefixes and suffixes
hyphen	contains hyphen
lower case	Lower case version of string feature

Table 2: Feature proliferators provided by ClearTK

The classifier interface has two classification methods. The first is a method that takes a set of features corresponding to an instance to be classified and returns a single result. The second is a method that takes a list of feature sets that correspond to a sequence of instances to be classified together (or in sequence) and returns a list of results. The latter method is needed for sequential learners such as Conditional Random Fields or Hidden Markov Models because these learners must be able to view the full sequence of instances at once.

After a result or list of results is returned from the classifier the classifier annotator is responsible for interpreting those results by updating existing annotations or creating new ones. For example, part-of-speech tagging is typically accomplished by classifying features extracted for each word annotation. The classification returned will correspond to a part-of-speech tag which can be used to set the part-of-speech of the word annotation in the CAS. This example assumes a type system similar to the one shown in Figure 1. For other tasks, such as named entity identification, the resulting classifications will result in new annotations. Named entity identification is often performed by classifying each word as beginning, inside, or outside a named entity (or some variant of this basic approach). In this example, the classifier annotator would be responsible for taking these word-level classifications and creating named entity annotations for the words that are classified as beginning or inside a named entity. Other NLP tasks such as semantic role labeling are significantly more complicated and require classification of pairs of annotations and as such, feature extraction is performed, in part, on pairs of annotations using feature extractors such as the distance extractor and the syntactic path extractor (see Table 1). In this context, the classification results are interpreted as establishing e.g. a predicate-argument relationship.

2.3. Training Data Consumers

When a classifier annotator is in training mode it creates training data from extracted features rather than classifying them. Each machine learning library specifies its own particular format that it expects as input when learning a model. Figures 2, 3, and 4 show snippets of training data for MaxEnt, Weka, and LIBSVM, respectively. In ClearTK, training data are created by *training data consumers* which are classes that know how to take a set of features and an outcome and write training

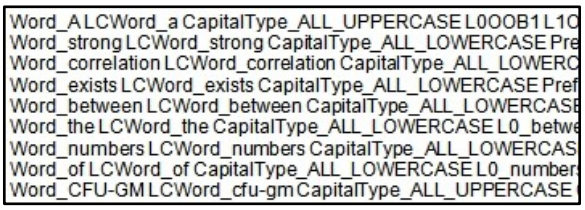


Figure 2: Example MaxEnt training data

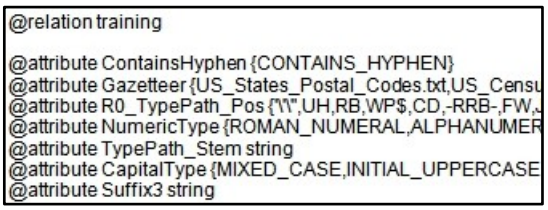


Figure 3: Example Weka training data

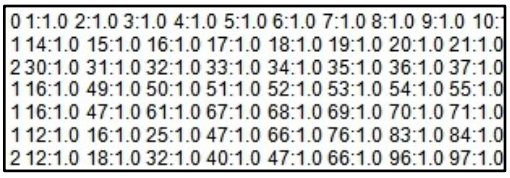


Figure 4: Example LIBSVM training data

data suitable for a given machine learning library. There is one (or more) training data consumer for each machine learning library that ClearTK supports.

2.4. Workflow

A typical workflow for a statistical NLP task in ClearTK involves creating training data, building a model, and then classifying annotations on unseen or test data. Creating training data involves taking an annotated corpus provided by some third party (e.g. Penn Treebank or GENIA) and transforming the data along with a host of features into a data format consumable by a machine learner. Creating training data in ClearTK involves the following steps for each document in the corpus:

a) A collection reader reads in a document from an annotated corpus from its distribution format and adds whatever useful annotation information is provided in the format (e.g. part-of-speech labels, named entities, syntax parse, etc.) to the CAS.
b) A series of analysis engines process the document by creating the annotations needed for the classifier annotator's feature extractors.
c) A classifier annotator in training mode iterates through annotations that are to be classified. For each focus annotation(s) the following two steps are performed:
    - extract features for the annotation(s)
    - pass the features to a training data consumer

The result of this process is a training data file that can be read in by a machine learning library. Model building is performed directly by the machine learning library and is typically invoked from the command line or via a simple script. ClearTK provides basic scripts and examples to invoke the various learners. After a model has been built it is packaged up into a jar[11] file along with some additional meta-data so that a classifier wrapper class can be instantiated from the jar file.

When a classifier annotator is run in classification mode the following steps are performed on unseen or separate test data for each document:

a) A collection reader reads in a document
b) A series of analysis engines process the document by creating the annotations needed for the classifier annotator's feature extractors.
c) The classifier annotator iterates through annotations that are to be classified. For each focus annotation(s) the following three steps are performed:
    - extract features for the annotation(s)
    - pass the extracted features to the classifier wrapper for classification
    - interpret results of the classification
d) A CAS consumer writes out results in a format appropriate for an evaluation script.

3. NLP Components in ClearTK

ClearTK provides a growing library of UIMA components that support a variety of NLP tasks. The library consists of three main types of components: collection readers, analysis engines, and classifier annotators which are summarized in Table 3. The collection readers of particular interest provided by ClearTK are those that read in widely used annotated corpora such as Penn Treebank[12] or PropBank[13]. The Penn Treebank reader reads in constituent parse trees into the CAS such that the full syntactic parse of each sentence is represented in the CAS such that constituents and their relations can be retrieved. The PropBank reader extends this reader by layering on the predicate/argument structure provided by the PropBank corpus. There are also collection readers for reading in the ACE 2005 corpus[14] and the CoNLL 2003[15] shared task data.

The analysis engines provided by ClearTK include a pattern-based tokenizer, a gazetteer annotator, and various wrappers around other NLP libraries. The tokenizer is based on Penn Treebank tokenization rules[16]. The gazetteer annotator finds entries from a gazetteer in text using simple string matching. Other analysis engines include wrappers around the OpenNLP part-of-speech tagger, sentence detector, and syntax parser and a wrapper around the Snowball stemmer[17,18].

[11] http://java.sun.com/javase/6/docs/technotes/guides/jar/index.html
[12] http://www.cis.upenn.edu/~treebank/
[13] http://verbs.colorado.edu/~mpalmer/projects/ace.html
[14] http://www.nist.gov/speech/tests/ace/2005/
[15] http://www.cnts.ua.ac.be/conll2003/ner/
[16] http://www.cis.upenn.edu/~treebank/tokenization.html
[17] http://snowball.tartarus.org/
[18] We use a modified version of the Snowball stemmer

component	type	description
Penn Treebank reader	CR	Reads the Penn Treebank corpus
PropBank	CR	Reads the PropBank corpus
ACE2005 reader	CR	Reads in named entity mentions from the ACE 2004 and 2005 tasks
CoNLL2003 reader	CR	Reads in named entity mentions from the CoNLL 2003 task
GENIA reader	CR	Reads in the GENIA corpus
tokenizer	AE	Penn Treebank style tokenizer
sentence detector	AE	Wrapper around OpenNLP sentence detector
syntax parser	AE	Wrapper around Open NLP syntax parser
stemmer	AE	Wrapper around the Snowball stemmer
gazetteer annotator	AE	Finds mentions of entries in a gazetteer using simple string matching
POS tagger	CA	performs part-of-speech tagging
BIO chunker	CA	performs BIO-style chunking
predicate annotator	CA	Identifies predicates
argument annotator	CA	Identifies and classifies semantic arguments of predicates

Table 3: Components provided by ClearTK. CR = collection reader, AE = analysis engine, and CA = classifier annotator

ClearTK currently provides a small handful of classifier annotators: a part-of-speech tagger (described in section 4), a BIO-style chunker, and a pair of classifier annotators that support semantic role labelling. The BIO chunker performs text chunking using the popular **B**egin, **I**nside, **O**utside labelling scheme for classifying annotations as members of some kind of "chunk." For example, in named entity recognition labels such as "B-person" or "I-location" are used for words that begin a person mention or are inside a location mention, respectively. The BIO chunker is used for named entity recognition, shallow parsing, and tokenization. Semantic role labelling is achieved by the predicate and argument annotators. The predicate annotator decides whether constituents of a syntactic parse are predicates or not. The argument annotator runs subsequently and finds the arguments of a predicate.

4. Results

4.1. Biomedical part-of-speech tagging

To demonstrate the utility of a flexible feature extraction library coupled with interfaces to several popular machine learning libraries we report results for part-of-speech tagging on biomedical scientific literature. The part-of-speech tagger that was created using ClearTK consists of less than 60 lines of code. For training and testing we used the GENIA corpus which consists of 2000 MEDLINE abstracts and about 500,000 part-of-speech tagged words (Tateisi and Tsujii 2004). An inter-annotator agreement study was conducted on fifty of the abstracts and was found to be 98.62% (as simple agreement). The GENIA tagger performs at 98.49% accuracy (Tsuruoka, Tateishi et al. 2005) when trained on the first 90% of the data and tested on the remaining 10%. This represents state-of-the-art performance for this corpus.

distributed by Lucene: http://lucene.apache.org/

	description	Features
F1	current word	$word_i$
F2	word and word bigram features	F1, $word_{i+1}$, $word_{i+2}$, $word_{i-1}$, $word_{i-2}$, $word_i$+$word_{i-1}$, $word_i$+$word_{i+1}$
F3	previous tags	F2, pos_{i-1}, pos_{i-2}, pos_{i-1}+pos_{i-2}
F4	character prefixes and suffixes	F3, prefixes sizes 1, 2, and 3, suffixes sizes 1, 2, 3, 4, 5, and 6
F5	lexical characteristics	F4, capital type (e.g. all caps, initial caps, etc.), numeric type (e.g. all digits, contains digit, roman numeral, etc.), contains hyphen
F6	lower case	F5, lower_case($word_i$)
F7	Stem	F6, stem($word_i$)

Table 4: Feature sets used for training part-of-speech tagging models.

4.2. Feature sets

Table 4 provides a listing of the feature sets that were used for training a part-of-speech tagging model. These feature sets are loosely based on the features used in (Tsuruoka, Tateishi et al. 2005). Given a word in a sentence, $word_i$, each numbered feature set in Table 4 describes a set of features extracted for that word for part-of-speech classification. The feature sets are cumulative such that F2 contains all of the features in F1, F3 contains all of the features in F2, and so on. Feature set F3 is problematic for Mallet because it is a sequential learner tagging an entire sequence at once. This means that previous part-of-speech labels of words earlier in the sentence are not available as features for words later in the sentence. For this reason there are no performance results given for Mallet for feature set F3. For the Mallet experiments using features sets F4 through F7 the part-of-speech features introduced in F3 are excluded.

4.3. Part-of-speech tagging accuracy

Table 5 shows the results of the ClearTK part-of-speech tagger using the seven feature sets against four machine learning libraries. LibSVM out-performed the other learners for every feature set with a top performance of 98.63%. The general trend, as expected, is that adding

	LibSVM	MaxEnt	Mallet	SVM^{perf}
F1	**94.91**	93.40	91.80	90.21
F2	**96.85**	94.44	91.51	92.38
F3	**96.75**	93.76		91.94
F4	**98.49**	98.14	95.88	96.99
F5	**98.58**	98.22	96.35	97.31
F6	**98.58**	98.28	96.37	97.42
F7	**98.55**	98.16	96.34	97.43

Table 5: Part-of-speech tagging accuracy results

	LibSVM	MaxEnt	Mallet	SVM^{perf}
F1	343	0.5	755	102
F2	959	13	669	68
F3	469	16		81
F4	324	20	598	77
F5	258	21	646	83
F6	297	21	597	77
F7	280	21	435	83

Table 6: Training time for building models for part-of-speech tagger in minutes.

	LibSVM	MaxEnt	Mallet	SVM^{perf}
F1	40	0.3	0.5	0.6
F2	186	0.3	1.0	2.5
F3	200	0.4		2.6
F4	137	0.7	1.3	4.7
F5	123	0.7	1.5	4.4
F6	133	0.7	1.5	2.4
F7	127	0.7	1.5	3.1

Table 7: Tagging time for test data in minutes.

more features improves performance. Interestingly, feature set F3, which introduces part-of-speech tag features, performs worse than using feature set F2. Similarly, for feature set F7, which introduces stemmed word features, the performance generally degrades slightly.

Each data point in Table 5 represents five-fold cross validation. For the columns labelled LibSVM, MaxEnt, and SVM^{perf} (a variant of SVM^{light}) each fold is trained on 80% of the data and tested on the remaining 20%. The Mallet learner was prohibitively slow when training on 80% of the data and so only 20% of the data was used for training in each fold. We trained Mallet on 80% of the data using feature set F6 and it took five days to train, 2.5 minutes to tag, and performed at 97.84% accuracy. This was a frustrating result because Mallet has consistently outperformed the other learners on other sequential tagging tasks such as tokenization and named entity recognition.

4.4. Part-of-speech model learning time

Table 6 provides a general approximation of how long each learner requires to train a model for a given feature set. Because the model training took place on a wide variety of CPUs ranging from a single processor running at 1.7 GHz with 1GB of RAM to a doubly hyper-threaded CPU running at 3.4GHz with 4GB of RAM, these results are not strictly comparable. In general, however, the models that took longer to build were run on more powerful machines and so it is possible to make rough conclusions about the relative performance. The minimum training time across the five folds is given rather than the average. The clear trend is that the MaxEnt learner is much faster than the other three.

Despite the smaller amount of training data, Mallet is still consistently the slowest learner (as shown in Table 6.) We have trained Mallet models on similarly sized data sets with a much smaller training cost. However, the number of possible classification outcomes in these models was much less. This finding is consistent with the time complexity of training CRFs which includes a term that is the square of the number of outcomes.

4.5. Part-of-speech tagging time

Table 7 provides a general approximation of the time it took to assign part-of-speech tags to the test set. Again, for the same reasons described above, this data can only be used for rough comparison. Still, there are two trends which seem quite clear. LibSVM is by far the slowest classifier of the three and MaxEnt is the fastest. Another interesting trend is that there is little or no connection between how long it takes to train a model and how long it takes to classify words. For example, the slowest learner, Mallet, provides the second fastest tagger while the second slowest learner, LibSVM, provides the slowest tagger.

4.6. Discussion

Clearly, this experiment is nowhere near an exhaustive search through the space of possible feature sets and machine learning libraries. Furthermore, each learner has its own set of configuration parameters that can and should be tuned for a particular task. SVM^{light}, for example, comes in many flavors including SVM^{light}, SVM^{struct}, SVM^{hmm}[19], and SVM^{perf}. The latter being the one used in this experiment with essentially default configuration parameters, i.e. a linear kernel was used with the regularization parameter C set to 20.0. SVM^{perf} generates binary classifiers which were normalized using Platt's probabilistic outputs for SVMs as described by (Lin, Lin et al. 2007). LibSVM was trained using the linear kernel and default values for all other parameters. Similarly, Mallet was trained without changing any of the default parameter settings. MaxEnt was run with 150 iterations with a feature frequency cut-off of four. We experimented with using beam search with MaxEnt but found that it made very little difference in the outcomes. Weka was excluded from this experiment because it

[19] SVM^{hmm} is not currently supported by ClearTK but seems the most appropriate choice for part-of-speech tagging.

contains a wide variety of machine learning algorithms, many of which do not handle string features, and because we are not familiar enough with this library to make a confident selection among the many choices to represent this library well.

Despite the limitations of the experiment (e.g. using 20% of the data for training Mallet, no parameter tuning, and limited feature space exploration) it is interesting to note that several of the configurations performed at state-of-the-art for this task. The results also point to possibilities for future experimentation such as combining the fastest taggers (MaxEnt and Mallet), removing the features introduced in feature set F3, and replacing SVM^{perf} with SVM^{struct} or SVM^{hmm}.

More important than the scientific contributions of this experiment are the observations on how ClearTK makes this kind of experimentation possible and easy. By having a feature extraction library that is highly configurable with respect to a user-defined type system it is possible for developers to reuse feature extractors for a wide variety of NLP tasks. In fact, the feature extractors used for the part-of-speech taggers were written in the context of supporting named entity recognition and there were no new feature extractors created for the part-of-speech tagger (though, admittedly, these are very similar tasks.) Additionally, because feature extraction is performed independently of any particular machine learning library, it is possible to swap out one learner for another and directly compare them with respect to performance (both accuracy and throughput) with all other factors being equal. This allows a best-of-breed approach to classification in which e.g. Mallet CRF could be used for certain sequential labelling tasks while LibSVM or SVM^{light} can be used for binary classifications required for semantic role labelling while implementations for all of these tasks share common code infrastructure.

This experiment was carried out using an experimental configuration file for defining a set of feature extractors to be used for training and classification. This allows decisions about which feature extractors to use at runtime without having to change the code that calls the feature extractors for each feature set of interest. Additionally, UIMA is highly configurable at runtime via the use of configuration files called *descriptors* which allows, for example, one to define a CPE using XML to specify a particular execution order of components. These two configuration mechanisms allowed each run of the experiment to be executed without any code changes or recompilation. Each run was executed by calling a single script that took in a set of properties associated with the learner being used, a feature extraction configuration file, a set of descriptor files, and parameters that determined the testing and training sets.

5. Conclusion

We have described a new framework for statistical NLP called ClearTK. ClearTK provides a rich feature extraction library, interfaces to several popular machine learning libraries, and a library of components that perform a variety of NLP tasks. We have demonstrated that ClearTK can perform at state-of-the-art level for the task of biomedical part-of-speech tagging and discussed and compared the performance of different learners on this task.

6. Acknowledgements

The authors gratefully acknowledge the following members of CLEAR for their input during the creation of ClearTK: Ying Chen, Ken Griest, James Martin, Martha Palmer, and Wayne Ward. We also thank the respective authors of the previously mentioned machine learning libraries and the members of the UIMA community that have answered our many questions via the UIMA users list.[20] We would also like to thank the University of Colorado Technology Transfer Office[21] who helped fund this project.

7. References

Chang, C. C. and C. J. Lin (2001). "LIBSVM: a library for support vector machines." Software available at http://www. csie. ntu. edu. tw/cjlin/libsvm **80**: 604–611.

Ferrucci, D. 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.

Joachims, T. (1999). "Making large-Scale SVM Learning Practical. Advances in Kernel Methods-Support Vector Learning." B. Scoelkopf, C. Burges, A. Smola.

Lin, H. T., C. J. Lin, et al. (2007). "A note on Platt's probabilistic outputs for support vector machines." Machine Learning **68**(3): 267-276.

McCallum, A. K. (2002). "Mallet: A machine learning for language toolkit." Unpublished. http://mallet. cs. umass. edu.

Tateisi, Y. and J. Tsujii (2004). "Part-of-speech annotation of biology research abstracts." Proceedings of LREC04.

Tsuruoka, Y., Y. Tateishi, et al. (2005). "Developing a Robust Part-of-Speech Tagger for Biomedical Text." Advances in Informatics: 10th Panhellenic Conference on Informatics, PCI 2005, Volos, Greece, November 11-13, 2005: Proceedings.

Witten, I. H. and E. Frank (2005). Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann.

[20] http://incubator.apache.org/uima/mail-lists.html
[21] https://www.cusys.edu/techtransfer/

UIMA-based Clinical Information Extraction System

Guergana K. Savova, Karin Kipper-Schuler, James D. Buntrock, and Christopher G. Chute
Division of Biomedical Informatics, Mayo Clinic College of Medicine, Rochester, Minnesota, USA
E-mail: savova.guergana@mayo.edu, schuler.karin@mayo.edu, buntrock@mayo.edu, chute@mayo.edu

Abstract

This paper describes the Mayo Clinic information extraction system for the clinical domain which was developed using IBM's Unstructured Information Management Architecture. The system is being used to process and extract information from free-text clinical notes (>25M documents). Annotators are strung together to build a pipeline for the discovery of clinical named entities such as diseases, signs/symptoms, anatomical sites and procedures. Attributes related to the named entities – context, status and relatedness to patient – are also extracted from the text. The pipeline consists of a context free tokenizer, context sensitive spell corrector annotator, abbreviation disambiguation annotator, lexical normalizer annotator, sentence detector, context dependent tokenizer, part of speech tagger, shallow parser, dictionary look-up annotator, a machine learning component, negation annotator and WSD component. We describe the architecture, annotators and evaluation of select annotators. Some extensions and applications of the system are presented and future challenges outlined. The system has been in production since 2005 and is to be released in the public domain in 2008.

1. Introduction and background

The vast amount of free text clinical information is a rich source for clinical research and knowledge discovery. Mayo Clinic's record indexing heritage started with Dr. Plummer who began organizing patient dossiers in 1907. Since 1935, our institution has maintained a comprehensive, machine-readable set of coded diagnostic and procedural data that provides indexing to patient records for retrieval and data analyses to serve the broader goal of patient care, research and education.

Recent advances in Natural Language Processing (NLP) technology resulted in an ever increasing interest of the biomedical community towards using NLP tools to process large amounts of textual data. Until recently such systems for processing biomedical and clinical texts have been narrowly focused on a specific type of clinical document. The field of NLP has matured to the point that commodity functions could be reused in larger systems.

Several research projects have been developed that have focused specifically on the development of NLP components to process and analyze free-text. However, little research deals directly with textual clinical data mainly due to the limited number of NLP researchers with full access to patient clinical data. A notable example of a production NLP system for processing clinical reports is Medical Language Extraction and Encoding System (MedLEE) (Friedman, 1997) initially designed for radiology reports and later extended to other medical domains such as mammography reports and discharge summaries. MetaMap developed at the National Library of Medicine (Aronson et al., 2000) is an example of an NLP system designed to process scholarly biomedical literature. It provides mapping of unrestricted text to the Unified Medical Language System [1] (UMLS) Metathesaurus concepts. MetaMap involves parsing the text into noun phrases using the SPECIALIST™ minimal commitment parser, the generation of lexical, syntactic and orthographic variants, followed by a sophisticated mapping and ranking of the phrases. University of Pittsburgh's Cancer Text Information Extraction System [2] (caTIES) -- a part of the Cancer Biomedical Informatics Grid [3] (caBIG) project at the National Cancer Institute -- encodes information extracted from free text surgical pathology reports to populate caBIG-compliant data structures.

There are two main efforts to build flexible and modular frameworks that can accommodate the need for large volume processing of unstructured data. Information Extraction (IE) systems which transform unstructured data into structured databases can be built using either of these technologies. GATE [4] (Generalized Architecture for Text Engineering) is an open-source environment that provides an architecture, software framework, and graphical development environment for the development of text annotators and resources. ANNIE is a highly modularized general purpose IE system developed on the GATE platform. GATE and ANNIE have been successfully used in the biomedical domain (Frank et al., 2004; Goldin and Chapman, 2003). Like GATE, IBM's Unstructured Information Management Architecture (UIMA) [5] is a framework which allows the development of text analysis systems. Besides text, UIMA has the capabilities of handling other unstructured data. Annotations can be extracted and written into a relational database. The four main UIMA services are acquisition, unstructured information analysis, structured information access, and component discovery (Ferrucci and Lally, 2004). A UMIA annotator performs a specific task. Once annotators are written to conformance, both UIMA and GATE provide pipeline development and permit the developer to quickly customize processing to a specific task.

IBM's Biological Text Knowledge Services (BioTeKS) (Mack et al. 2004) initiative is the first major application of UIMA. It is designed for text analysis and search methods for the Life Sciences domain. Clinical IE systems in general rely on a number of NLP components each implementing a different technology. An extensive overview of recent advances in the clinical domain is (Meyster et al., 2008).

In this paper we present the design, architecture and

[1] http://www.nlm.nih.gov/research/umls/

[2] https://cabig.nci.nih.gov/tools/caties

[3] https://cabig.nci.nih.gov/

[4] http://www.gate.ac.uk/

[5] http://incubator.apache.org/uima/

select component evaluation of a large-scale, modular, real-time clinical IE system built within the UIMA framework. The system is being used at the Mayo Clinic to discover important clinical facts from relatively loose clinical text which are then stored in a structured database to afford many applications and use cases. The system is to be released in the public domain in 2008[6].

2. Clinical text and its characteristics

Clinical notes mostly follow the XML-based Health Level 7 Clinical Document Architecture (CDA) structure which "provides an exchange model for clinical documents such as discharge summaries and progress notes, and brings the healthcare industry closer to the realization of an electronic medical record"[7]. The CDA document specifies a <Section> element which has a narrative element <text> within which one can find the relevant clinical content. Beyond the CDA structure, the text within each section is unstructured. (Pakhomov et al., 2006) present the characteristics of clinical language along with examples from Mayo Clinic clinical notes. In summary, clinical notes are textual descriptions of physician-patient encounters. The narrative is typical of quasi-spontaneous speech with the following characteristics: incomplete sentences, inverted constructions, conversational grammar, misspellings and spelling variations, abbreviations and acronyms. All these characteristics of clinical texts set it apart from newswire texts and even scholarly biomedical literature texts and present NLP challenges of varying degrees. A clinical domain IE system needs to address these challenges.

3. System design

Mayo Clinic and IBM collaborated on a Text Analysis project with the goal to build a clinical IE system that would enable the retrieval of clinical documents at Mayo Clinic. A number of requirements were established from the genesis of the project from an indexing and retrieval perspective: scalability, flexibility, stability, real-time processing, distributed processing, and annotation versioning. The ability to re-index the document corpus was considered critical in the architecture. Best practices included software modularity allowing best-of-breed annotator components, whether the components were developed in-house, adopted from open source, or purchased commercially. The architecture was designed to accommodate bulk and real-time processing and leveraged existing interface feeds for clinical documents by routing them to a warehouse. A work manager was written using messaging queues to distribute the work for text analysis. Additional text analysis engines can be configured and added with appropriate hardware to increase document throughput of the system. The text analysis engines are deployed as Enterprise Java Beans on IBM WebSphere Application Server™ (WAS) cluster consisting of twenty dual-processor blade servers. Each of the WAS servers have local message queues populated by a home-grown load balancing process with metadata about the request. For production deployment we used an asynchronous architecture. The text analysis engines were wrapped into Message Driven Beans. Each JMS message consumed by the Message Driven Bean was fed directly to the text analysis engine for annotation. The Message Driven Beans were deployed using WAS and the JMS Provider was IBM WebSphere MQ™.

4. Annotators and UIMA pipeline

To implement the annotators we used the AlphaWorks version of UIMA[8] and are in the process of converting them to the Apache version as part of our open-source release package.

Our annotators are strung together to build a pipeline for clinical named entities (NEs) discovery. NEs are textual mentions that belong to the same class, e.g. diseases, signs/symptoms, anatomical sites, procedures. Each NE, in turn, has additional attributes:

- Context with values of *current*, *historyOf*, and *familyHistoryOf*.
- Status with values of *confirmed*, *possible*, and *negated*.
- Related_to_patient with values of *true* and *false* according to whether the information is about the patient.

For example, in "There are no complaints worrisome for recurrent metastatic oropharynx cancer", "metastatic oropharynx cancer" has a context of *historyOf*, status of *negated* and related_to_patient is *true.*

The *context sensitive spell corrector annotator* is used for automatic spell correction on word tokens. This annotator uses a combination of isolated-word and context-sensitive statistical approaches to rank the possible suggestions (Thompson-McInness et al, 2004). The suggestion with the highest ranking is stored as a feature of a token.

The *context free tokenizer* is a finite state based annotator that parses the document text into the smallest meaningful spans of text. A token is a set of characters that can be classified into one of these categories: word, punctuation, number, contraction, possessive, or symbol without taking into account any additional context. The *context dependent tokenizer* uses context to detect complex tokens such as dates, times, and ordered or numbered problem lists.

The *lexical normalizer annotator* is applied only to words, possessives, and contractions. It generates a canonical form by using the National Library of Medicine UMLS Lexical Variant Generator (LVG)[9] tool. It also generates a list of lemma entries with Penn Treebank tags as input for the part-of-speech tagger.

The *sentence detector annotator* parses the document text into sentences. The sentence detector is based on a Maximum Entropy classifier[10] trained to recognize sentence boundaries from hand-annotated data.

The *part-of-speech (POS) pre-tagger annotator* executes prior to the POS tagger annotator. The pre-tagger loads a list of words with unambiguous POS with predetermined Penn Treebank tags which the POS tagger ignores. The *POS tagger annotator* assigns a part of speech tag to all tokens. The current production version is a proprietary IBM tagger which uses Hidden Markov Models trained on a combination of the Penn Treebank

[6] Supported by an IBM UIMA Innovation Award

[7] http://xml.coverpages.org/ni2004-08-20-a.html

[8] http://www.alphaworks.ibm.com/tech/uima

[9] http://SPECIALIST.nlm.nih.gov

[10] http://maxent.sourceforge.net/

corpus[11] of general English and a corpus of manually tagged clinical data developed at the Mayo Clinic (Coden et al., 2005; Pakhomov et al., 2006).

The *shallow parser annotator* makes higher level constructs at the phrase level. The shallow parser currently being used in our system is the proprietary IBM shallow parser. It uses a set of rules operating on tokens and their part-of-speech category to identify linguistic phrases in the text such as noun phrases, verb phrases, and adjectival phrases. For our public release of the pipeline, we will substitute the proprietary IBM components with open source ones from the openNLP project[12].

The *dictionary NE annotator* uses a set of enriched dictionaries (SNOMED-CT[13], MeSH[14], RxNorm[15], and Mayo Synonym Clusters (MSC)) to look up NEs of type drugs, disorders/diseases, signs, and symptoms in the document text. The MSC database contains a set of clusters each consisting of diagnostic statements that are considered to be synonymous. Synonymy here is defined as two or more terms that have been manually classified to the same category in the Mayo problem list repository, which contains over 20 million manually coded diagnostic statements. These diagnostic statements are used as entry terms for dictionary look up.

The *ML (Machine Learning) NE annotator* is based on a Naïve Bayes classifier trained on a combination of the UMLS entry terms and the MSC. Each diagnostic statement is represented as a bag-of-words and used as a training sample for generating a Naive Bayes classifier which assigns MSC identifiers to noun phrases identified in the text of clinical notes. In operation, the annotator scans through the input text attempting to match any portion of it to the MSC database using the dictionary *NE annotator.* If a match is detected, then a new NE is introduced and assigned an MSC id; otherwise, the output of the *shallow parser annotator* is used to identify noun phrases whose heads match single word entities in MSC and are classified with the Naïve Bayes classifier.

The negation annotator assigns a certainty attribute to each named entity with the exception of drugs. This annotator is based on a generalized version of Chapman's NegEx algorithm (Chapman et al., 2001.)

The Word Sense Disambiguation (WSD) component aims to uniquely associate a term with a single ontology entry. We built a classifier for 50 of the most frequent and relevant ambiguities occurring in the Mayo clinic clinical notes (Savova et al., 2008b.). The classification method is based on the empirical risk minimization principle, which aims to minimize prediction errors measured by a loss function (Zhang, 2004). The *abbreviation disambiguation annotator* attempts to detect and expand abbreviations and acronyms based on Maximum Entropy classifiers trained on automatically generated data (Pakhomov, 2002). A set of Mayo compiled dictionaries are also used that detect abbreviations and hyphenated terms.

We additionally developed the freely available Mayo Weka/UIMA Integration [16] (MAWUI) which provides a link between UIMA and the machine learning package Weka[17]. MAWUI allows feature selection and extraction within UIMA after which models can be trained within WEKA and subsequent classifiers integrated back within the UIMA pipeline.

5. Evaluation of select annotators

Although a difficult and time-consuming task due to the need to develop evaluation testbeds, we have formally evaluated some of the components and are currently in the process of evaluating others. (Pakhomov et al., 2006) describe the development of a POS-tagged corpus from Mayo Clinic's clinical notes. In addition, they trained and evaluated the performance of the TnT open source tagger (Brants, 2000) on the combination of Penn Treebank data and the Mayo Clinic corpus. Results were reported as

$$Accuracy = 100 \times (hits/total) \quad (1)$$

The best accuracy of 94.7% is achieved when the model is built from data that included clinical texts. Without such combined corpus, the accuracy drops to 89.8%. The performance of the IBM proprietary POS-tagger is described in (Coden et al., 2005.) The best accuracy is close to 93% achieved when the model was trained using the combined Penn Treebank and Mayo Clinic data.

Additionally, we have developed a corpus of manually annotated disease named entities (Ogren et al., 2008.) The dataset consists of 160 notes and 1,556 named entity annotations which were mapped to 658 unique SNOMED codes. Inter-annotator agreement is for span (90.9%), concept code (81.7%), context (84.8%), and status (86.0%) agreement. Complete agreement for span, concept code, context, and status was 74.6%. In (Kipper-Schuler et al., 2008) we describe the evaluation of the dictionary look-up algorithm against these gold standard annotations in terms of

$$Recall = correctSystemHits/allInGoldStandard \quad (2)$$

$$Precision = correctSystemHits/allSystemHits \quad (3)$$

$$F\text{-}score = (2*P*R)/(P+R) \quad (4)$$

F-score range is 0.56-0.81 depending on the strictness of match criteria. The most strict one (F-score of 0.56) is for exact span, context, and status match. The least constrained matching criterion is a partial match of the textual span. The F-score for the negation detection is 0.96 pointing to the robustness and stability of that component (Kipper-Schuler et al., 2008.)

Evaluation of our WSD component is presented in (Savova et al., 2008b.) Experimentation with 28 different feature sets identified the most productive combinations for each of the 50 clinical ambiguities with a reported F-score of 0.82. A disadvantage of the current WSD component is that no universal feature set can be applied to every ambiguous word to achieve maximum performance. Some customization for each ambiguous term is needed. The performance of the abbreviation disambiguation component was measured at approximately 89% accuracy (Pakhomov, 2002). As previously pointed out, the presence of acronyms and abbreviations pose a considerable challenge and are found to be a major source of system errors.

Currently, we are in the process of training and evaluating the openNLP part-of-speech tagger and shallow parser which will be released with the public release of the system.

[11] www.TB-2.upenn.edu

[12] http://opennlp.sourceforge.net/api/index.html

[13] http://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html

[14] http://www.nlm.nih.gov/mesh/

[15] http://www.nlm.nih.gov/research/umls/rxnorm/index.html

[16] http://informatics.mayo.edu/text/index.php?page=weka

[17] http://www.cs.waikato.ac.nz/ml/weka/

6. Discussion

The Mayo Clinic IE system has been applied to a number of use cases. Clinical research usually involves the identification of patient cohorts to be included in a study which have to meet a number of criteria per disease case definition. A recent application of our system is the successful identification of a cohort of patients for a congestive heart failure study.

Our system has proven flexible and adaptable. We were able to quickly extend it to new variables, e.g. the discovery of patient smoking status (Savova et al., 2008a) as defined in one of five categories: SMOKER, CURRENT SMOKER, PAST SMOKER, NON-SMOKER and UNKNOWN based on patients' respective medical records. We customized the dictionary to perform focused NE extraction. Our classifier was built off WEKA using the MAWUI component to link UIMA features and WEKA. Feature extraction and selection was performed off UIMA annotations. The WEKA classifier was added as a new component in our existing pipeline. This smoking status classifier will be used as part of a more general identification of risk factors for peripheral vascular diseases.

Currently, we are also developing a classifier to automatically categorize neuroradiology reports according to tumor progression. The annotations generated by our UIMA pipeline are to be used as classifier features, e.g. NEs, parsed phrases and sentences.

We are also in the process of building additional annotators that will extend the described pipeline. One such more complex annotator is the Drug Profile. A knowledge model which identifies the components necessary for drug identification based on several use cases is under construction. The model centers on the relation of the drug to the particular patient to distinguish it from the generic drug characteristics provided by pharmaceutical companies. This more complex drug identification annotator will handle the discovery of dosage, frequency, side-effects, status (e.g. discontinued, started, past, as needed), among other drug attributes related to a given patient at a given time point.

We demonstrated here that we have successfully built an extendible IE system for the clinical domain. There are a number of challenging NLP tasks that we are planning to tackle in the future – in particular, coreference resolution, relation discovery between NEs, temporal resolution and text-based reasoning.

7. Conclusions

In this paper we presented a high throughput information extraction system for the clinical domain developed within IBM's Unstructured Information Management Architecture. It consists of a number of annotators each performing a specific natural language processing task. We have evaluated most of them and reported results. The system has been in production at the Mayo Clinic since 2005 and is planned to be released in the public domain in 2008. Its flexibility and modularity allow for quick extensions and applications to a variety of use cases.

References

Aronson AR, Bodenreider O, Chang HF, Humphrey, SM, Mork, JG, Nelson SJ, Rindflesch TC and Wilbur WJ. (2000). The NLM Indexing Initiative. Proc AMIA Symp. 2000:17-21.

Brants, T. (2000). TnT – a statistical part-of-speech tagger. Proc. NAACL/ANLP-2000 Symposium.

Chapman WW, Bridewell W, Hanbury P, Cooper G, Buchanan B. (2001). Evaluation of Negation Phrases in Narrative Clinical Reports. Proc AMIA, 2001.

Coden, AR., Pakhomov SV, Ando R and Chute CG. (2005). Domain-specific language models and lexicons for tagging. J Biomed Inform. 38: 422-30.

Ferrucci D, Lally A. (2004). UIMA: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering. 2004;10 (3:4):327-348.

Frank E, Hall M, Trigg L, Holmes G, and Witten IH. (2004). Bioinformatics Advanced Access. Bioinformatics online. April 8, 2004.

Friedman C. (1997). Towards a Comprehensive Medical Language Processing System: Methods and Issues. Proc AMIA, 1997.

Goldin IM, Chapman WW. (2003). Learning to Detect Negation with 'Not' in Medical Texts. Paper presented at: Workshop on Text Analysis and Search for Bioinformatics at the 26th Annual International ACM SIGIR Conference, 2003.

Kipper-Schuler KC, Kaggal V, Masanz J and Savova GK. (2008). System evaluation on a named entity corpus from clinical notes. Proc LREC 2008.

Mack R., Mukherjea S., Soffer A., Uramoto N., Brown E., Coden A., Cooper J., Inokuchi A., Iyer B., Mass Y., Matsuzawa H. and Subramaniam LV. (2004). Text Analytics for life sciences using the unstructured information management architecture. IBM Sys J, vol. 43, N 3, 2004.
http://www.research.ibm.com/journal/sj/433/mack.html.

Meystre SM; Savova GK; Kipper-Schuler KC and Hurdle JE. (2008). Extracting information from textual documents in the electronic health record: a review of recent research. IMIA Yearbook of Medical Informatics 2008. Methods Inf Med 2008; 47 Suppl 1:138-154.

Ogren PV, Savova GK and Chute CG. (2008). Constructing evaluation corpora for automated clinical named entity recognition. Proc LREC 2008.

Pakhomov S, Coden A. and Chute CG. (2006). Developing a corpus of clinical notes manually annotated for part-of-speech. Int J Med Inf (2006) 75, 418-429.

Pakhomov S. (2002). Semi-Supervised Maximum Entropy Based Approach to Acronym and Abbreviation Normalization in Medical Texts. 40th Meet ACL, Philadelphia, PA.

Savova GK, Ogren PV, Duffy P, Buntrock JD and Chute CG. (2008a). Patient smoking status identification within Mayo Clinic Life Sciences System. J Am Med Inform Assoc. 2008; 15(1).

Savova GK; Coden A; Sominsky I; Johnson R; Ogren P; de Groen P and Chute CG. (2008b). Word sense disambiguation across two domains: biomedical literature and clinical notes. J Biomed Inform. doi:10.1016/j.jbi.2008.02.003

Thompson-McInness B, Pakhomov S, Pedersen T. (2004). Automating Spelling Correction Tools Using Bigram Statistics. Medinfo, San Francisco, CA, USA.

Zhang T. (2004). Solving large scale linear prediction problems using stochastic gradient descent algorithms. In ICML 2004, pp. 919–926.

Shallow, Deep and Hybrid Processing with UIMA and Heart of Gold

Ulrich Schäfer

German Research Center for Artificial Intelligence (DFKI), Language Technology Lab
Campus D 3 1, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
email: ulrich.schaefer@dfki.de

Abstract

The Unstructured Information Management Architecture (UIMA) is a generic platform for processing text and other unstructured, human-generated data. For text, it has been proposed and is being used mainly for shallow natural language processing (NLP) tasks such as part-of-speech tagging, chunking, named entity recognition and shallow parsing. However, it is commonly accepted that getting interesting structure and semantics from documents requires deeper methods. Therefore, one of the future goals for UIMA will be inclusion of openly available, deep linguistic parsing technology for the generation of semantics representations from documents.
Heart of Gold is a lightweight, XML-based middleware architecture that has been developed for this purpose. It supports hybrid, i.e. combined shallow and deep processing workflows of multiple NLP components to increase robustness and exploit synergy, and linguistic resources for multiple languages. The notion of explicit transformation between component input and output enables flexible interaction of existing NLP components. Heart of Gold foresees both tightly (same process) and loosely coupled (via networked services) processing modes. Assuming familarity with UIMA, we introduce Heart of Gold and propose and discuss hybrid integration scenarios in the context of UIMA. Possible applications include precision-oriented question answering, deep information extraction and opinion mining, textual entailment checking and machine translation.

1. Introduction

At last with the incubation of UIMA as an Apache project, language technology and natural language processing tools are becoming standard techniques usable in mainstream application software. More and more pre-existing tools for text processing got news clothes and found their way into the UIMA component repository[1]. So, job done – what's next?
If one looks closer at the different types of integrated tools, then only the same few types of components appear – at least those openly available: shallow tools such as part-of-speech taggers, chunkers, named entity recognizers and entity detectors, the latter ones for specific tasks or domains. But this is only half the range of natural language processing (besides the language dimension that is currently mostly English).
To get structure and semantics from unstructured text, much more is needed than identifying types of named entities or part-of-speech tags. Ultimately, one needs text understanding, getting the relations between the various entities mentioned in the text, or at least a predicate-argument structure per sentence. This cannot be provided only by shallow tools, but requires deep parsing.
Moreover, even rather shallow tasks such as template-based information extraction work better in rather fixed word-order languages such as English, but perform worse on free word-order languages. Again, deep syntactic parsing could help to improve results. While efficiency is no longer a problem for deep parsing, robustness can be overcome using a hybrid approach we will discuss below.
The distinction between shallow and deep processing is a continuum rather than a strict dichotomy. Deep means knowledge-intensive, comprehensive, generic. By shallow, we mean partial, less informed analysis, often domain-dependent. It has to be pointed out that the distinction between statistical and rule-based NLP is orthogonal to that, as deep and shallow analyses may involve both. For more in-depth discussions, cf. (Uszkoreit, 2002; Schäfer, 2007).
There is one further distinction that plays a role when characterizing the kind of analysis results and its relation to NLP software architecture. (Cunningham et al., 1997) present a classification of software infrastructures for NLP by distinguishing three models they call

- *referential* (analyses are stored as separate representations with pointer references into the original text),
- *additive* (e.g. cumulative SGML/XML annotation markup), and
- *abstraction-based* (as in typed feature structures of deep analysis where the analysis result consists of a closed, integrated information structure for larger text entities, typically a whole sentence).

Thus, architectures for shallow *and* deep components should support at least referential and abstraction-based representations. The latter is not supported by architectures such as GATE (Bontcheva et al., 2004).
Although the designers of UIMA had deep processing in mind already when they started developing their framework (Ferrucci and Lally, 2004; Götz and Suhre, 2004), at least openly available deep processing is currently less developed in UIMA than in other approaches, and so is the novel hybrid (combined deep and shallow) integration paradigm.
In this paper, we will present another framework, Heart of Gold, and discuss its relation to UIMA. This framework has been developed independently of and in parallel to UIMA. It integrates mainly openly available shallow and deep processing components and linguistic resources for many languages.
Heart of Gold (Callmeier et al., 2004; Schäfer, 2007)[2] is a lightweight, XML-based middleware architecture that has

[1] http://uima.lti.cs.cmu.edu

[2] Download, documentation: http://heartofgold.dfki.de

been developed in the context of DELPH-IN[3], a collaboration of various research groups developing and sharing open source tools and linguistic resources for the Head-driven Phrase Structure Grammar (Pollard and Sag, 1994). Being open source, Heart of Gold is also contained in the OpenNLP collection[4].

The main motivation why Heart of Gold has been devised is flexible support for the combination of multiple shallow NLP analysers with a deep HPSG parser, and for generating robust deep semantic representations of the meaning of natural language sentences. It could be shown that through integration with PoS tagging and named entity recognition, deep parsing coverage on newspaper text can be doubled, even on broad-coverage grammars with relatively large lexica (Crysmann et al., 2002; Schäfer, 2007).

We will in the following discuss the Heart of Gold approach, how it differs from and can be brought together with UIMA. The idea is that if Heart of Gold would be migrated to UIMA (hypothetically), not only single components should be migrated, but also the efforts invested in elaborated hybrid integration workflows should be preserved, e.g. for English, German and Japanese.

2. Heart of Gold

2.1. Design principles

One of the design decisions that have been made in Heart of Gold is the choice of open XML standoff markup as the only representation format for input and output of the components. It contains aspects of both *referential* (through character offset positions encoded in attributes) and *additive* representation architectures mentioned in the introduction.

Standoff markup is easy to exchange, transformable using standard XML transformation languages such as XSLT (Clark, 1999), and interoperability benefits from Unicode being part of the XML standard. The XML approach is in principle compatible with UIMA which in addition supports isomorphic object structure in the supported programming languages. The elegance of the XML approach lies in the closeness to XML corpus annotation, i.e. persistently 'multidimensionally' stored analysis results form an automatically annotated corpus.

Fig. 1 gives a schematic overview of the Heart of Gold middleware architecture in between applications (top) and external NLP components (bottom). Communication with the middleware is supported via XML-RPC web service or programmatically via a Java API. When a new application session is started, it takes a configuration specifying the wrapped NLP components to start for this session. Each component is started according to its own configuration.

An application client can send texts to the middleware and the NLP components are then queried in a numerically defined processing order ('depth'). The shallowest components (e.g. tokenizer) are assigned a low number and are started first etc. The output of each component must be

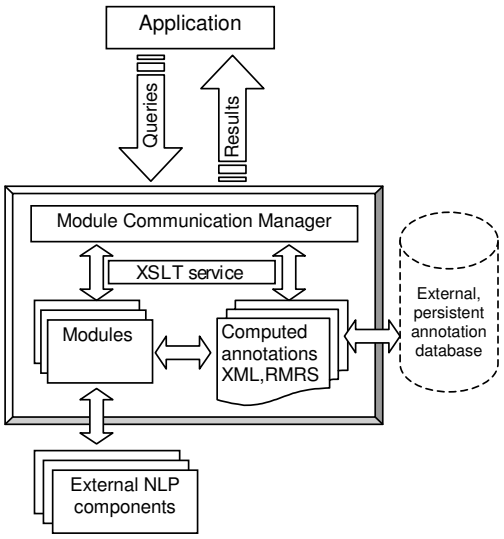


Figure 1: Middleware architecture

XML markup. Each component gets the output of the previous component as input by default, but can also request (via configuration) other annotations as input.

As there is no commonly accepted XML standard for linguistic annotation, the architecture itself makes no assumption about the XML format as long as it is well-formed XML. XML transformation is used to mediate between different I/O formats.

Components may produce multiple output annotations (e.g. in different formats). Thus, the component dependency structure in general forms a graph. In Section 2.8., we describe a further generalization of the default pipeline.

2.2. Session and annotation management

The resulting NLP annotations are stored in a per-session markup storage (Fig. 2) that groups all annotations for an input query (a sentence or text) in *annotation collections*. The markup can also be made persistent by saving it to XML files or storing it in an XML database. Annotations can be accessed uniquely via a URI of the form

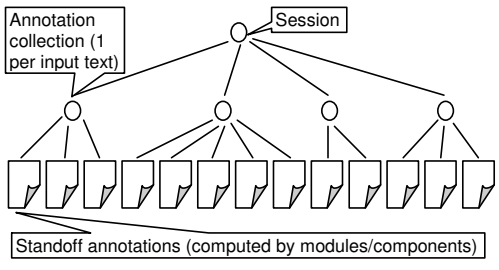


Figure 2: Session and multi-dimensional markup storage

`hog://sid/acid/aid` in XPath expressions where `sid` is a session ID, `acid` is an annotation collection ID and `aid` is an annotation identifier typically signifying the name of the producing component. Structured metadata like configuration and processing parameters (e.g. processing time and date, language ID etc.) are automatically stored within the annotation markup as first root daughter element.

[3]DEep Linguistic Processing with HPSG Initiative; http://www.delph-in.net

[4]http://opennlp.sf.net

Component	NLP Type	Languages	Implemented in
JTok	tokenizer	de, en, it,...	Java
ChaSen	Japanese segm./tagger.	ja	C
TnT	HMM tagger	de, en,...	C
Treetagger	statistical tagger	en, de, es, it,...	C
Chunkie	HMM chunker	de, en,...	C
ChunkieRMRS	chunk RMRSes	de, en	XSLT, SDL/Java
LingPipe	statistical NER	en, es,...	Java
FreeLing	morph./tagger/NER	ca, en, es, gc, it	C++
Sleepy	shallow parser	de	OCaml
SProUT	morph., shallow NLP/NER	de, el, en, ja,...	XTDL, Java
LoPar/wbtopo	PCFG parser	de	C, XSLT
Corcy	coref resolver	en	Python
RASP	shallow NLP	en	C, Lisp
PET	HPSG parser	de, el, en, ja,...	C, C++, Lisp
RMRSmerge	RMRS merger	de, en,...	XSLT, SDL/Java
SDL	generic sub-architectures		SDL/Java

Figure 3: Integrated components from shallow (top) to deep (bottom). Details and references on http://heartofgold.dfki.de.

2.3. Wrapped NLP components

NLP components are integrated through adapters called modules (either Java-based, subprocesses or via XML-RPC) that are also responsible for generating XML standoff output in case this is not supported natively by the underlying, pre-existing component. Various shallow and deep NLP components have already been integrated, cf. Fig. 3.

2.4. Integration through transformation

Heart of Gold heavily relies on the use of XSLT for combining and integrating XML markup produced by the NLP components. The general idea is to use XSLT to transform XML to other XML formats, or to combine and query annotations. In particular, XSLT stylesheets may resolve conflicts resulting from multi-dimensional markup, choose among alternative readings, follow standoff links, or decide which markup source to give higher preference.

(Carletta et al., 2003), e.g. propose the NXT Search query language (for corpus access) that extends XPath by adding query variables, regular expressions, quantification and special support for querying temporal and structural relations. Their main argument against standard XPath is that it is impossible to constrain both structural and temporal relations within a single XPath query. Our argument is that XSLT can complement XPath where XPath alone is not powerful enough, yet providing a standardized language.

Further advantages we see in the XSLT approach are portability and efficiency (in contrast to 'proprietary' and slow XPath extensions like NXT), while it has a quite simple syntax in its (currently employed) 1.0 version. XSLT can be conceived as a declarative specification language as long as an XML tree structure is preserved (not necessarily fully isomorphic to the input structure). However, XSLT is Turing-capable and therefore suited to solve in principle any markup integration or query problem.

Finally, extensions like the upcoming XSLT/XPath 2.0 version or efficiency gains through XSLTC (translet compilation) can be taken on-the-fly and for free without giving up compatibility. Technically, the built-in Heart of Gold XSLT processor could easily replaced or complemented by an XQuery processor. However, for the combination and transformation of NLP markup, we see no advantage of XQuery over XSLT.

Heart of Gold comes with a built-in XSL transformation service, and module adapters can easily implement transformation support by including a few lines of code. Stylesheets can also be generated automatically in Heart of Gold, provided a formal description of the transformation input format is available. An example is the mapping from named entity grammar output type definitions in the deep-shallow integration scenario we will describe briefly by example below.

2.5. Performance

There is a slight performance drawback Heart of Gold shares with other service-oriented architectures. It is imposed by the XML framework, yet partly countervailed by fast XSL transformation. While deep parsing alone is in the range of milliseconds per sentence thanks to the very efficient PET system, a hybrid parse may take up to 1-2 seconds including PoS tagging, named entity recognition, and some more seconds for very long sentences.

The majority of the time goes into Java-based XML processing, and there is room for optimization. However, we think this is an acceptable tradeoff for very flexible and quick experimental integration of (new) NLP components in exciting new, rapidly prototyped applications, including the benefits of Unicode given for free in multilingual integration scenarios.

2.6. Integrating shallow and deep processing

The main motivation for integrating deep and shallow processing is that deep parsing alone is not robust enough. Open class words such as names, locations, time expressions not in the deep lexicon prevent construction of full parse trees. A simple, yet very efficient way of making parsing more robust to gaps in the lexicon is using PoS tagging as pre-processing. From the PoS information for a word unknown to the deep lexicon, one or more generic

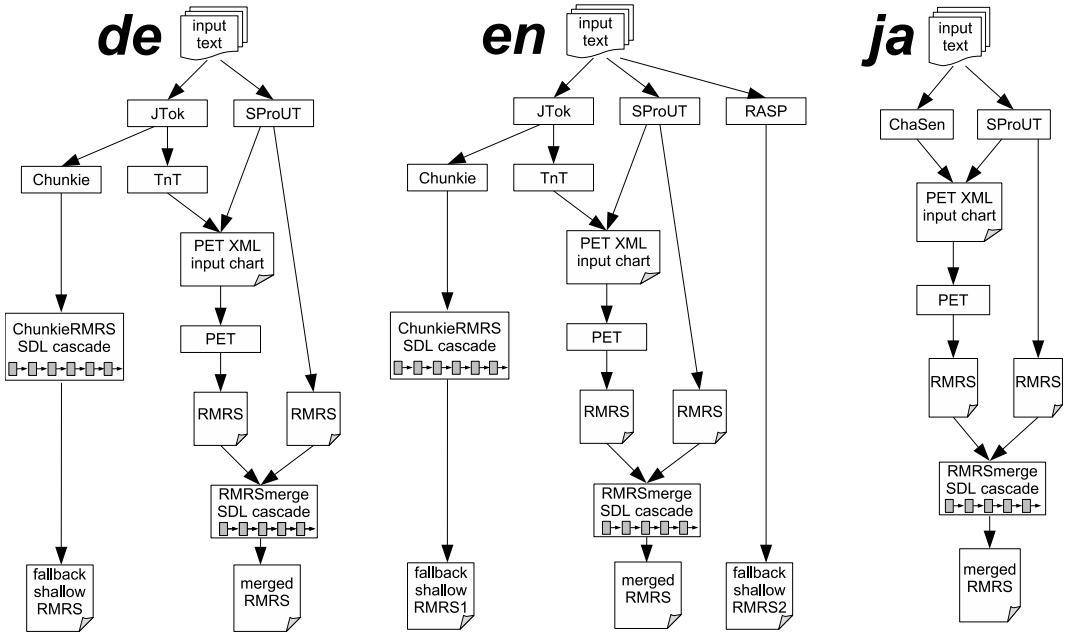


Figure 4: Hybrid workflows for German, English, Japanese

lexicon entry is put on the deep parser's chart containing at least the information about the word class and maybe other information such as morphological or basic semantics features.

In the same way, named entity recognizers and gazetteers may contribute e.g. domain-specific information missing in the deep grammars. This forms a division of labor: the (expensive) deep grammar is responsible for modelling correct general language use, syntax and generating a sentence-semantic representation, while the shallow components add domain-specific information that does not need to be maintained in the deep lexicon and can be easily changed for a different application domain.

We now give an example for such a hybrid workflow, depicted for English in the middle of Figure 4. The configuration for German is analogous except that there is no secondary shallow fallback component.

The raw input sentence text is sent to the JTok tokenizer and the named entity recognizer SProUT (Drożdżyński et al., 2004), because SProUT comes with its own tokenizer with a finer-grained token classification. Chunkie (HMM chunker) and TnT (HMM tagger) use the tokenized output from JTok as input, Chunkie output is used as secondary input for the ChunkieRMRS cascade (left branch in Figure 4 for German and English) we will be explain Section 2.8.

The output of this cascade (shallow RMRS) can be used as shallow fallback result in case the deep parser fails to parse the input sentence. Similarly, RASP (English only) produces another shallow RMRS as fallback annotation.

Back to the middle pipeline, the tagger output for the sentence 'George Washington was born in Virginia'

```
<w id="TNT0" cstart="0" cend="5">
  <surface>George</surface>
  <pos tag="NNP" prio="1.000000e+00"/>
</w>
<w id="TNT1" cstart="7" cend="16">
  <surface>Washington</surface>
  <pos tag="NNP" prio="1.000000e+00"/>
</w>
<w id="TNT2" cstart="18" cend="20">
  <surface>was</surface>
  <pos tag="VBD" prio="1.000000e+00"/>
</w>
<w id="TNT3" cstart="22" cend="25">
  <surface>born</surface>
  <pos tag="VBN" prio="1.000000e+00"/>
</w>
<w id="TNT4" cstart="27" cend="28">
  <surface>in</surface>
  <pos tag="IN" prio="1.000000e+00"/>
</w>
<w id="TNT5" cstart="30" cend="37">
  <surface>Virginia</surface>
  <pos tag="NNP" prio="1.000000e+00"/>
</w>
```

as well as the recognized named entities from SProUT

```
<w id="SPR1" cstart="0" cend="16" prio="0.5"
   constant="yes">
  <surface>George Washington</surface>
  <typeinfo id="TIN1" baseform="no">
    <stem>$genericname</stem>
  </typeinfo>
</w>
<w id="SPR2" cstart="30" cend="37" prio="0.5"
   constant="yes">
  <surface>Virginia</surface>
  <typeinfo id="TIN2" baseform="no">
    <stem>$genericname</stem>
  </typeinfo>
</w>
```

are transformed into the deep parser's (PET; (Callmeier, 2000)) input chart format using XSLT (shown above is already the transformed version). Another XSLT stylesheet

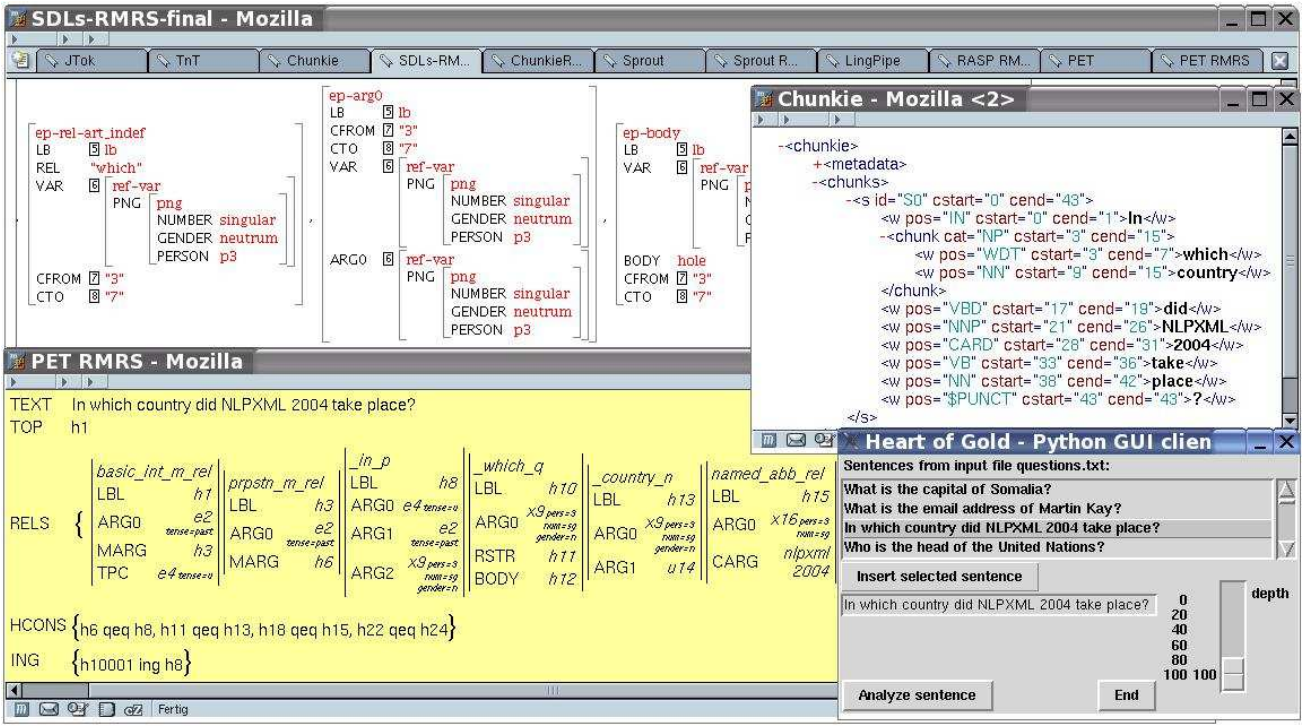


Figure 5: Heart of Gold analysis results in GUI with specialized XML visualizations

is used to combine these and possibly other annotations in a single PET input chart document [5].
From this XML input chart, the deep parser generates or looks up deep lexicon entries, then starts HPSG parsing.

2.7. Output: semantics representation

Instead of huge typed feature structures containing the monotonically assembled unification result of the HPSG parse tree per sentence, applications are rather interested in a distilled sentence semantics representation. This distillate largely omits linguistics details from morphology and syntax, but provides a graph structure of connected semantic entities for the whole sentence, including its predicate-argument structure.
One such representation generated by many modern HPSG grammars is MRS - minimal recursion semantics (Copestake et al., 2005) or its robust XML variant RMRS (Copestake, 2003). RMRS turns the semantics representation of a sentence into an XML standoff format as well (including references back into character positions of the input sentence) and thus is appropriate for being processed by the middleware and forwarded to applications.
An RMRS contains EPs (elementary predications) with argument connected via handle and individual variables. The idea is that shallow NLP components may deliver equivalent where possible, but maybe underspecified representations, e.g. the argument positions of a transitive verb may be empty when a shallow parser cannot find the appropriate object. The HCONS (handle constraints) attribute allows to concisely express scopus ambiguities via handles. The ING (in-group) attribute explicitly indicates conjunction of its contained pairs.
A sample RMRS as produced by the deep parser PET running the HPSG grammar ERG[6] in Heart of Gold is shown in Figure 6, depicted in the MRS matrix format instead of raw XML for better readability.
Figure 7 shows a structured result from the named entity recognizer SProUT transformed to the RMRS format. It contains information such as name variants or the indication that Virginia is of type province. This information was not passed to the deep grammar as it is irrelevant for parsing in this case, but it might be interesting for consuming applications.
Thus, RMRS is used as a uniform, though not mandatory output format of both deep and shallow components. The RMRSmerge module at the end of the shallow-deep pipeline can be used to merge RMRSes produced by multiple components into a single representation ('merged RMRS' in Figure 4).

2.8. Sub-architectures

Heart of Gold modules roughly correspond to TAEs (Text Analysis Engines) in UIMA. The equivalent to UIMAs composed TAEs are sub-architectures in Heart of Gold.
The SDL module enhances Heart of Gold with a compilable NLP module control flow for sub-architectures, i.e., enabling declarative specification of modules that are composed of other modules. SDL (System Description Language) has been developed independently of Heart of Gold by (Krieger, 2003).
SDL generates Java code for declaratively defined architectures of NLP systems obeying a class interface imposed by

[5] Stylesheets are also employed to visualize the linguistic markup, e.g. by transforming analysis results to HTML (Fig. 5) or LaTeX.

[6] English Resource Grammar; http://www.delph-in.net/erg/

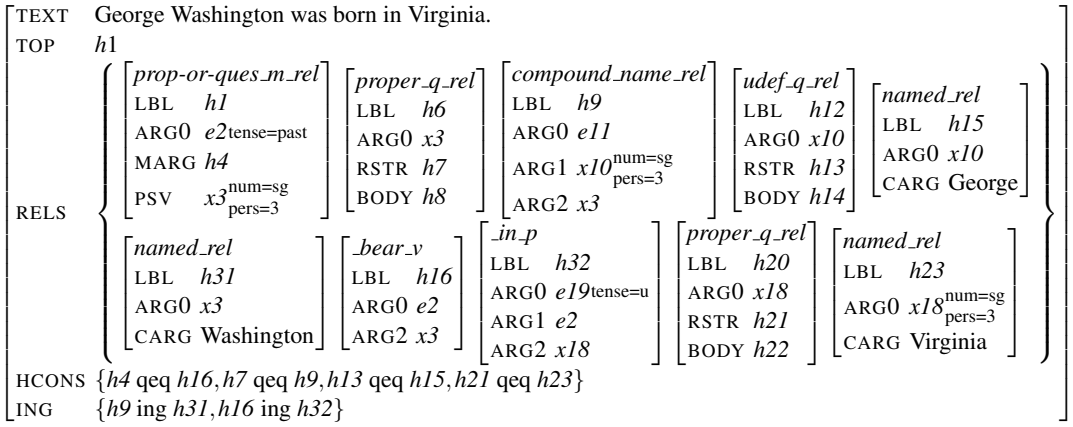


Figure 6: Deep semantics representation (RMRS) by ERG and PET for "George Washington was born in Virginia".

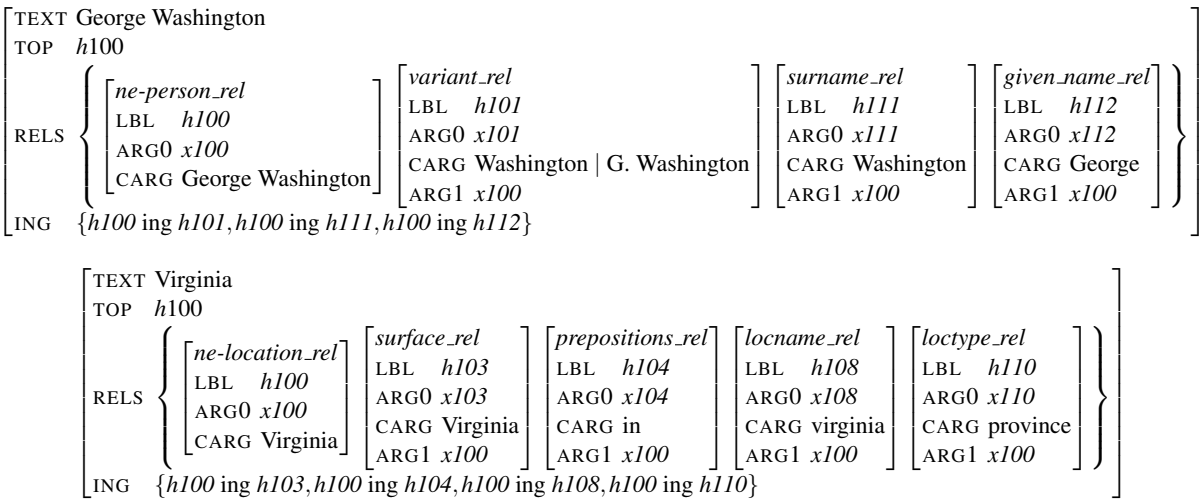


Figure 7: Shallow RMRS by SProUT for the named entities "George Washington" and "in Virginia".

the SDL framework. The initial intention was to be able to declaratively define cascaded SProUT instances, e.g. for shallow chunk parsing. An application are e.g. cascades of (shallow) NLP modules and XSL transformations.

Although the described mainly sequential control flow approach in Heart of Gold for NLP modules by defining a depth and canonical processing order based upon, augmented with potentially multiple input and multiple output annotations in each processing step, was flexible enough for deep-shallow integrations for many languages, it turned out that some envisaged, RMRS-related shallow processing applications required additional features such as loops and parallelism – which SDL supports.

The declarative specification of the architecture is a single expression consisting of symbolic module names connected via operators, plus assignment of these symbolic module names to Java class names, constructor arguments, and some processing options.

The SdlModule is a generic wrapper plugging SDL subarchitectures into the Heart of Gold. SdlModule acts like any other Heart of Gold module in that it takes a (configurable) XML annotation as input, and returns an output annotation.

The name of the embedded SDL Java class containing the compiled architecture description (previous section) is part of the SdlModule configuration. The generated Java code of the SDL description is compiled and executed at runtime in the SdlModule code using Java reflection.

ChunkieRMRS (Frank et al., 2004), left branch of the German and English workflows in Figure 4, shall now serve as an example of such a compound, SDL-based component. Externally, it acts like a single component, but consists of eight sub-modules in this case (Fig. 8).

A robust, partial semantics representation is generated from a shallow chunker's output and morphological analysis by means of a processing cascade consisting of four SProUT grammar instances with four interleaved XSLT transformations. SProUT is used here for intermediate, rule-based transformation of complex typed feature structures.

The scenario is equally a good example for XSLT-based annotation integration. Chunker analysis results are included in the RMRS to be built through an XSLT stylesheet using the XPath expression

```
document($uri)/chunkie/chunks/chunk[
   @cstart=$beginspan and @cend=$endspan]
```

where `$uri` is a variable containing an annotation identifier of the form `hog://sid/acid/aid` as explained in Section 2.2.

2.9. Applications

A recent application of the middleware for English is hybrid processing of scientific papers in the field of language

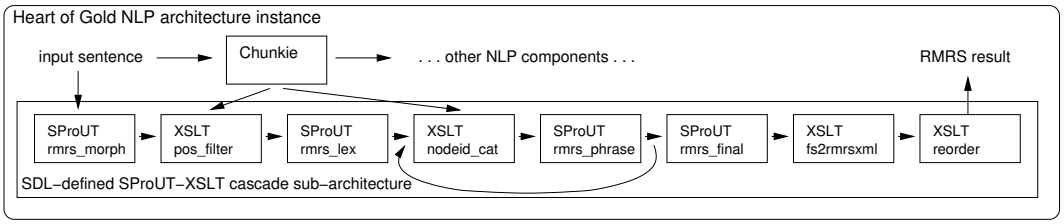


Figure 8: SDL sub-architecture for constructing RMRSes from chunks in Heart of Gold

technology (Schäfer et al., 2008). Currently abstracts, later full papers from the ACL Anthology (Bird et al., 2008) are extracted from PDF, parsed with Heart of Gold, and so-called quriples are extracted from the RMRS. Quriples are query-oriented subject-verb-object-rest tuples that are indexed and made searchable from a GUI application called the Scientist's Workbench. 62.5% full parse coverage could be reached with out-of-the-box components and lingware resource in a pipeline as described in Section 2.6.

Another application is question answering from structured knowledge sources such as ontologies or databases. In the QUETAL system (Frank et al., 2006), the hybridly computed RMRSes of natural language questions, both German and English, are directly translated to SPARQL ontology queries of which the results are returned as answers formulated by a template-based generator.

There are various further applications of purely shallow configuration instances of Heart of Gold, e.g. for information extraction on soccer game descriptions (Buitelaar et al., 2006) and opinion mining.

2.10. Related Work

There is few related work on hybrid NLP architecture. Most others such as (Grover and Lascarides, 2001) are systems that integrate specific instances of shallow and deep tools without having the right or claiming themselves to form generic architectures. GATE is shallow by design and (without modification) not suited for abstraction-based components such as deep parsers.

An interesting approach from a research area unrelated to language technology by (Löwe and Noga, 2002) bears some similarity with Heart of Gold. They describe a generic XML-based, network-enabled middleware architecture for re-usable components that explicitly makes use of XSLT as adapter language between components. It has been proposed as a generic middleware in the spirit of CORBA, DCOM or EJB. However, it can well be conceived as a supporting, independent argument that the XML and XSLT-based middleware approach is a useful design pattern for software architecture.

3. UIMA Integration Scenarios

In this section, we discuss a hypothetical migration of hybrid processing in Heart of Gold to UIMA. The cheap way of migrating to UIMA would be to wrap Heart of Gold configuration instances as a whole in a UIMA TAE (text analysis engine). But this would probably not add any value. There is no doubt that components currently integrated in Heart of Gold could be migrated to UIMA, each in a separate TAE, as well as the simple, 'direct' pipelines for hybrid processing, as composed TAEs.

Going this way would require more implementation work, but the result would be (hopefully) analogous configurability, then UIMA-enabled. To keep the same flexibility as in Heart of Gold, the configurable stylesheets for transformation between components could be put in separate TAEs or as adapters. At the end, UIMA would benefit from new (mostly open source) TAEs, and the new paradigm of hybrid analysis.

An interesting, but even more implementation-intensive approach would be separating linguistic resources such as grammars or lexica specific to components by putting them behind KSAs (knowledge source adapters). Currently, each component comes with its own resources and resource format. There is some synergetic gain forseeable through KSAs, but there is doubt that this will be worth the effort for every component.

Another interesting approach would be sharing the type hierarchy among deep and shallow components. Currently, this is possible for the deep parser PET and the generic NLP engine SProUT. Both use the same very efficient bit-vector encoding technique for their type system (Callmeier, 2000). As it is for HPSG, it necessarily supports multiple inheritance, while in the UIMA, only single-inheritance type systems seem to be supported which would cause a problem e.g. for the feature structure structure representation of parse results.

The biggest effort will probably have to be invested in the CAS (Common Analysis Structure). The lightweight Heart of Gold proposes and supports RMRS as optional common format, but is also open to any other standoff format. Agreements on the formats are only necessary between connected components.

In UIMA, the I/O of TAEs has to be specified more rigidly as part of the CAS. In the ideal case, this could result in systems where the workflow can be computed automatically (in the ideal case) from a global I/O specification, e.g. by an application. Currently, this is a manual task in Heart of Gold.

4. Summary and Outlook

We have presented Heart of Gold and discussed its relation to and possible connection with UIMA. UIMA is an emerging, industrial-strength platform for application-oriented processing of unstructured data such as natural language text. It has been designed very thoroughly and now constitutes a rather complex framework. Therefore, mainly shallow NLP tools have been migrated to UIMA so far.

Heart of Gold is meant mainly as a lightweight research instrument for flexible experimentation with hybrid, XML-

based NLP component integration and for rapid prototyping of applications using semantic analyses of text. Research on deep processing and improving it with respect to robustness through various approaches, also other than integrating it with shallow tools, e.g. through additional statistical models and extensions, is a hot research topic.

Now that hybrid processing has turned out promising and proven successful for a range of applications, UIMA may help to bring deep and hybrid processing faster to a broader community and market. And vice versa: UIMA and UIMA-based applications will benefit from increased analysis depth gained through hybrid processing.

5. Acknowledgments

This work has been supported by a grant from the German Federal Ministry of Education and Research (FKZ 01 IW F02). Thanks to the anonymous reviewers for their valuable, concise and encouraging comments.

6. References

Steven Bird, Robert Dale, Bonnie Dorr, Bryan Gibson, Mark Joseph, Min-Yen Kan, Dongwon Lee, Brett Powley, Dragomir Radev, and Yee Fan Tan. 2008. The ACL anthology reference corpus: a reference dataset for bibliographic research. In *Proceedings of LREC-2008*, Marrakech, Morocco.

Kalina Bontcheva, Valentin Tablan, Diana Maynard, and Hamish Cunningham. 2004. Evolving GATE to meet new challenges in language engineering. *Natural Language Engineering*, 10(3-4).

Paul Buitelaar, Thomas Eigner, Greg Gulrajani, Alexander Schutz, Melanie Siegel, Nicolas Weber, Philipp Cimiano, Günter Ladwig, Matthias Mantel, and Honggang Zhu. 2006. Generating and visualizing a soccer knowledge base. In Frank Keller and Gabor Proszeky, editors, *Proceedings of the EACL06 Demo Session*, Trento, Italy.

Ulrich Callmeier, Andreas Eisele, Ulrich Schäfer, and Melanie Siegel. 2004. The DeepThought core architecture framework. In *Proceedings of LREC-2004*, pages 1205–1208, Lisbon, Portugal.

Ulrich Callmeier. 2000. PET – A platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering*, 6(1):99–108.

Jean Carletta, Stefan Evert, Ulrich Heid, Jonathan Kilgour, Judy Robertson, and Holger Voormann. 2003. The NITE XML toolkit: flexible annotation for multimodal language data. *Behavior Research Methods, Instruments, and Computers, special issue on Measuring Behavior*, pages 353–363.

James Clark, 1999. *XSL Transformations (XSLT)*. World Wide Web Consortium, http://w3c.org/TR/xslt.

Ann Copestake, Dan Flickinger, Ivan A. Sag, and Carl Pollard. 2005. Minimal recursion semantics: an introduction. *Journal of Research on Language and Computation*, 3(2–3):281–332.

Ann Copestake. 2003. Report on the design of RMRS. Technical Report D1.1b, University of Cambridge, Cambridge, UK.

Berthold Crysmann, Anette Frank, Bernd Kiefer, Stefan Müller, Jakub Piskorski, Ulrich Schäfer, Melanie Siegel, Hans Uszkoreit, Feiyu Xu, Markus Becker, and Hans-Ulrich Krieger. 2002. An Integrated Architecture for Deep and Shallow Processing. In *Proceedings of ACL 2002*, pages 441–448, Philadelphia, PA.

Hamish Cunningham, Kevin Humphreys, Robert Gaizauskas, and Yorick Wilks. 1997. Software infrastructure for natural language processing. In *Proceedings of the 5th Conference on Applied Natural Language Processing*, pages 237–244, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Witold Drożdżyński, Hans-Ulrich Krieger, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu. 2004. Shallow processing with unification and typed feature structures – foundations and applications. *Künstliche Intelligenz*, 2004(1):17–23.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

Anette Frank, Kathrin Spreyer, Witold Drożdżyński, Hans-Ulrich Krieger, and Ulrich Schäfer. 2004. Constraint-based RMRS construction from shallow grammars. In *Proceedings of the HPSG-2004 Conference, Center for Computational Linguistics, Katholieke Universiteit Leuven*, pages 393–413. CSLI Publications, Stanford, CA.

Anette Frank, Hans-Ulrich Krieger, Feiyu Xu, Hans Uszkoreit, Berthold Crysmann, Brigitte Jörg, and Ulrich Schäfer. 2006. Question answering from structured knowledge sources. *Journal of Applied Logic*, pages 20–48. DOI: 10.1016/j.jal.2005.12.006.

Thilo Götz and Oliver Suhre. 2004. Design and implementation of the UIMA common analysis system. *IBM Systems Journal*, 43(3). DOI: 10.1147/sj.433.0476.

Claire Grover and Alexis Lascarides. 2001. XML-based data preparation for robust deep parsing. In *Proceedings of ACL/EACL 2001*, pages 252–259, Toulouse, France.

Hans-Ulrich Krieger. 2003. SDL – A description language for building NLP systems. In *Proc. of the HLT-NAACL Workshop on the Software Engineering and Architecture of Language Technology Systems*, pages 84–91.

Welf Löwe and Markus L. Noga. 2002. A lightweight XML-based middleware architecture. In *Proceedings of IASTED AI 2002*, Innsbruck. ACTA Press.

Carl Pollard and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. University of Chicago Press, Chicago.

Ulrich Schäfer, Hans Uszkoreit, Christian Federmann, Torsten Marek, and Yajing Zhang. 2008. Extracting and querying relations in scientific papers on language technology. In *Proc. of LREC-2008*, Marrakesh, Morocco.

Ulrich Schäfer. 2007. *Integrating Deep and Shallow Natural Language Processing Components – Representations and Hybrid Architectures*. Ph.D. thesis, Faculty of Mathematics and Computer Science, Saarland University, Saarbrücken, Germany.

Hans Uszkoreit. 2002. New Chances for Deep Linguistic Processing. In *Proceedings of COLING 2002*, pages xiv–xxvii, Taipei, Taiwan.

CFE – a system for testing, evaluation and machine learning of UIMA based applications

Igor Sominsky, Anni Coden, Michael Tanenblatt
IBM Watson Research Center
19 Skyline Dr., Hawthorne NY, 10532 USA
E-mail: sominsky@us.ibm.com, anni@us.ibm.com, mtan@us.ibm.com

Abstract

There is a vast quantity of information available in unstructured form, and the academic and scientific communities are increasingly looking into new techniques for extracting key elements - finding the structure in the unstructured. There are various ways to identify and extract this type of data; one leading system, which we will focus on, is the UIMA framework. Tasks that are often desirable to perform with such data after it has been identified are testing, correctness verification (evaluation) and model building for machine learning systems. In this paper, we describe a new Open Source tool, CFE, which has been designed to assist in both model building and evaluation projects. In our environment, we used CFE extensively for both building intricate machine learning models, running parameter-tuning experiments on UIMA components, and for evaluating a hand-annotated "gold standard" corpus against annotations automatically generated by a complex UIMA-based system. CFE provides a flexible, yet powerful language for working with the UIMA CAS - the results of UIMA processing - to enable the collection and classification of resultant data. We describe the syntax and semantics of the language, as well as some prototypical, real-world use cases for CFE.

1. Introduction

A wealth of information is captured in unstructured sources, ranging from text to streaming video. Analysis of these sources and extraction of knowledge from them is the goal of several frameworks currently in use within the research community. Two open source frameworks, the Gate system (http://www.gate.ac.uk) and the UIMA framework (http://incubator.apache.org/uima) have gained popularity. Although different in several aspects, both systems are modular, providing a mechanism for creating and executing a pipeline of components, known as "annotators". These annotators implement various algorithms, each of which performs a specific analysis task. In this paper, we will focus on textual unstructured data sources. Hence, examples of annotators are natural language processing (NLP) components, such as part-of-speech taggers and parsers, rule based annotators or named entity annotators based on a variety of machine-learning algorithms.

One of the challenges faced by all application developers is the testing and evaluation methodology. At a high level, the issues typically are regression testing and computation of accuracy metrics (e.g. precision/recall) against a "gold standard". There are many tools available (e.g., Knowtator (http://knowtator.sourceforge.net) and Callisto (http://callisto.mitre.org)) for manually annotating documents, both for building machine learning training data and for creating "gold standard" corpora to be used as a reference set in testing. Evaluation and testing involves comparing annotations from different executions. Within the UIMA framework, this can be accomplished by extracting and comparing values of properties of UIMA annotations. These annotations can be arbitrarily complex. Extraction of these properties, called *features*, is also one of critical sub-tasks in creating machine learning models, as the feature vectors for building the models can be generated from features values of UIMA annotation.

It should be noted that the term *features*, which is frequently used throughout this paper, is often used in different contexts. This term may refer to properties of UIMA annotation types or features that are used to build/evaluate models for machine learning algorithms. In this paper we will use the term *features* in relation to properties of UIMA annotations, while values of models for machine learning will be referred as *ML features*.

What we needed, but were not able to discover, was a tool that could be configured to extract specific portions of a UIMA CAS (Common Analysis Structure: the object-based data structure in which UIMA represents objects, properties and values), specifically a set of features from some set of annotations based on user specified conditions. Traditionally, application-specific "CAS Consumers" have been written to satisfy this requirement. While this approach is reasonable for a fixed (or nearly fixed) set of output requirements, it can be unwieldy when experimenting with different sets of features to be extracted, an underlying annotation model is in flux, or if two or more differing (yet equivalent) models need to be extracted and aligned. For these reasons, we created a system to perform these kinds of extraction tasks, and which provides a powerful declarative extraction specification language. The same functionality is also needed to generate *ML features* to build models that underlie machine learning algorithms. To accomplish the final steps of evaluation tasks, we combine the generalized feature extractor with a system within which accuracy metrics can be computed.

This paper is organized as follows. In section 2, we will describe the challenges of testing and evaluating UIMA

pipelines in detail and discuss why other testing and evaluation environments proved to be inadequate. The feature extraction specification language (FESL) – is introduced in section 3. Section 4 will describe a real-word use case of FESL performance evaluation of an NLP system and section 5 will demonstrate how FESL can be used for machine learning related processing. We conclude in section 6 with proposing some potential extensions.

2. Problem statement

Evaluation of an information extraction system consists of several steps: defining a baseline against which to compare, defining the comparison criteria, extracting relevant information from sources (e.g., the baseline and the system to be evaluated) and subsequent comparative analysis.

At a very general level, for a given textual document, a UIMA pipeline executes as shown in Figure 1.

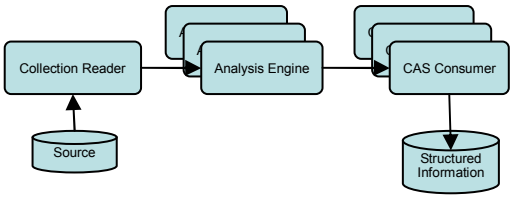


Figure 1: UIMA pipeline

First, the document is read into a Common Analysis System (CAS) structure. Next, a set of analysis engines (AEs) mark up this piece of text, producing *annotation* objects, each of which is usually associated with a span of text in that document. Finally, one or more CAS Consumers read these annotations, perform any necessary processing, and then output results.

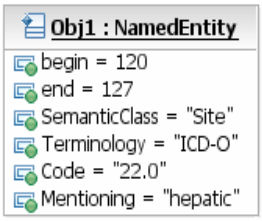


Figure 2: An abstract view of typical UIMA annotation

Each annotation (as shown for example in Figure 2) has properties associated with it. These properties contain specific information about the annotation, and as described in the introduction, are called *features*. Although the actual implementation of UIMA annotation objects is much more complex, this abstract view reflects information stored in these objects. The values of features are set by AEs and could either be modified or used without modification by subsequent annotation engines. In the example in Figure 2, the annotations are created with a dictionary lookup mechanism against a medical terminology, the attributes being the begin and end offsets of the relevant piece of text in the document that this annotation object is associated with, the semantic class of the named entity that is described by the annotation, the terminology name and code associated with it from that dictionary, and the actual text fragment.

The first step in the process of evaluation is the definition of equality between two types to be compared. This necessitates a specification of a set of features from both the test and reference sets that should be compared, and the criteria for the comparison. In the next step, the annotations of those types and their significant properties are extracted. We developed the language FESL to specify the details of this extraction. FESL contains sufficient semantics for expressing rules for generation of parameters for building machine learning models. The extraction can be implemented as part of a standard UIMA component (AE or CAS consumer) depending on particular application requirements. For the evaluation environment, we developed a tool that extracts required feature values using a CAS consumer. It performs the extraction from two CAS structures that are to be compared and loads the extracted information into a Microsoft Excel spreadsheet, where the final stages of the evaluations are executed, as described in section 4.

3. The Feature Extraction Specification Language (FESL)

To enable a high degree of flexibility and extensive functionality, we defined an XML-based specification language that expresses semantic rules for feature extraction. One of the key concerns in defining the language was to avoid any dependency upon any particular application of the extraction process. This allows reusing the same extraction semantics for different purposes, whether for comparative analysis, subsequent algorithm execution or machine learning related processing. The feature extraction process is independent of the representation of the feature in the final output. This enables different output formats for different use cases, such as machine learning or testing. As a simple example, extracted values for comparison could contain spaces in their representation, while the same values extracted for machine learning could replace spaces with underscore characters. The component also defines a destination for output. For instance, the analysis engine (AE) could store the extracted features values within a CAS structure and/or a subsequent CAS Consumer might output them to an external source such as a disk file or database.

The semantics of the specification language allow the definition of complex multi-parameter criteria that could identify a particular concept of interest. Such criteria allow locating the information expressed by any particular UIMA annotation and/or its features in a CAS structure, evaluating its value against one or more

conditions and recording the results in an internal depository for post processing. The criteria for such search can be specified by a combination of the following conditional expressions, written with FESL:

a. type of an annotation object that contains the feature (in the general case, the feature does not have to be a property of the object, but should be accessible (i.e *on the path*) from its properties, as will be shown further down in this section)
b. surrounding (enclosing) annotation type and relative location of the object within the enclosure, as indicated by the *enclosingAnnotation* attribute of the *targetAnnotations* XML tag, shown in Figure 3 (the significance of the enclosing annotation is explained below)
c. path to the feature from the annotation object, as indicated by the *featurePath* attribute of the *featureMatchers* XML tags, as shown in Figure 3
d. type and value of the feature itself; the feature value can be evaluated against different constraints expressed with FESL, as explained further down in this section
e. values of any public Java *get*-style methods (methods that accept no parameters and return a value) implemented by the underlying class of the feature
f. location of the object or the feature on a specific path (in cases when it is required to select/bypass annotations if they are features of certain annotation types)

One of the key capabilities of FESL mentioned in items (a), (c) and (f) is an ability to specify a "path" to a feature from an annotation object. This path is a sequence of feature/method names, separated by the colon character, that mimics the sequence of Java method calls required, starting at the annotation object, in order to extract the feature value. It should be noted that, as UIMA annotations support arrays as feature types, FESL also provides the ability to extract values of features that are arrays or properties of annotations that are contained in arrays. Figure 5 contains a sample of how arrays are specified in FESL. In addition, special array semantics allow accessing elements of arrays by index and sorting them by offset before extraction.

Some applications require performing an extraction of information relevant to a certain concept within sentence boundaries; other may extend the scope of the extraction to a paragraph. As mentioned in item (b) FESL has the ability to define such a scope by specifying an enclosing annotation as illustrated in Figure 3.

Typically, values of UIMA annotation features are required to be extracted, but FESL also enables an extraction of non-UIMA properties of an object by using Java reflection mechanism. As specified by item (e), a value returned by any public method that has no arguments can be extracted and treated in the same way UIMA features are processed. As shown in Figure 3, *getCoveredText* is not a property of a UIMA Annotation type, but rather a method that this type defines.

As previously mentioned in item (d) the feature values can be evaluated by conditional expressions stated in FESL. Particularly, the feature values can be evaluated whether they:

i. are of a certain type
ii. belong to a specific set of values (vocabulary), where the set of values, as shown on Figure 3, is defined by the *enumFeatureValues* XML tag
iii. belong to a range of numeric values (inclusively or non-inclusively) as defined by the *rangeFeatureValues* XML tag
iv. match certain bits of a bit mask (integer values only); the *bitmaskFeatureValues* XML tag will contain an integer bitmask along with a flag indicating whether the bitmask should exactly match to a feature value
v. match a Java regular expression pattern, where the *patternFeatureValues* XML tag will contain a regular expression against which a feature value will be evaluated

The evaluation of the search criteria can be specified in disjunctive normal form. Conjunctions are bounded by FESL *groupFeatureMatcher* XML tags and are referred to as groups. Disjunction is implicit between multiple groups. This gives a powerful and flexible way of defining fairly complex criteria for a search of a required annotation and/or its value.

It should be noted that the semantics of FESL, as shown in Figure 3, separate the concept and specification of target annotations (*TA*) from feature annotations (*FA*). Although they use identical semantic rules for specifying the search criteria, the ways the results of the search are processed are different. In particular, *TAs* are used to locate a concept, while *FAs* are the annotations upon which the extraction of features is performed. Target annotations are specified by the *targetAnnotationMatcher* XML tag, and feature annotations by the *featureAnnotationMatcher* XML tag. During the extraction process, a *TA* is located according to its search criteria. Once the TA is found, *FAs* that correspond to the *TA,* and match to their own search criteria, are located and feature values are extracted from them. Additionally, the semantics allow the extraction of features from multiple *FA*s, where each *FA* is located by its specific context relative to the *TA*. This is particularly useful in machine learning related processing where it is often required to select features from annotations that are located "near" another annotation with certain properties.

Let us consider a quite common example taken from the machine learning domain: extracting "a bag of words within a window of size 5 centered around the word 'tumor', excluding prepositions, conjunctions, articles and punctuation". This could be understood as: search for token-based annotations that corresponds to the word "tumor" (*TA*), and on every match consider the 5 nearest

token-based annotations (*FA*s) on both sides, and excluding tokens that have associated part-of-speech tags indicating they are of one of the following categories: preposition, conjunction, article or punctuation, then extract the token that corresponds to that *FA*. The FESL semantics allow the unambiguous specification of criteria for such a search that is shown in Figure 3.

```
<targetAnnotations className="BOW5Tumor"
    enclosingAnnotation="SentenceAnnotation">
  <targetAnnotationMatcher annotationTypeName="TokenAnnotation">
    <groupFeatureMatchers>
      <featureMatchers featurePath="getCoveredText" featureTypeName="String">
        <enumFeatureValues>
          <values>tumor</values>
        </enumFeatureValues>
      </featureMatchers>
    </groupFeatureMatchers>
  </targetAnnotationMatcher>
  <featureAnnotationMatchers annotationTypeName="TokenAnnotation"
      windowsizeLeft="5" windowsizeRight="5">
    <groupFeatureMatchers>
      <featureMatchers featurePath="getCoveredText" featureTypeName="String"/>
      <featureMatchers featurePath="pennTag" featureTypeName="String"
          exclude="true">
        <enumFeatureValues caseSensitive="true">
          <values>IN</values>
          <values>CC</values>
          <values>DT</values>
          <values>null</values>
        </enumFeatureValues>
      </featureMatchers>
    </groupFeatureMatchers>
  </featureAnnotationMatchers>
</targetAnnotations>
```

Figure 3: Bag of words extraction sample

In this figure, short versions of UIMA annotation type names are shown for better readability. In the example, all extracted feature values are assigned a label "BOW5Tumor" (the value of the *targetAnnotation*'s "className" attribute). The label could be used in subsequent processing for the grouping of related results of extraction. The search is limited to token annotations (*TokenAnnotation*) within the same sentence (*SentenceAnnotation*), as specified by *enclosingAnnotation* attribute. Also, annotations of type *TokenAnnotation* have a property called *pennTag* that contains their part-of-speech tags. As illustrated in this example, the *TokenAnnotation*'s *getCoveredText* attribute is evaluated if, and only if, that same *TokenAnnotation*'s *pennTag* contains a value in the set specified under *enumFeatureValues* XML tag.

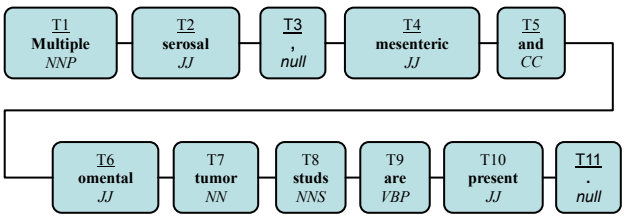


Figure 4: Tokenized sentence

To demonstrate how this FESL specification is applied consider the sentence from Figure 4. Each box on this figure corresponds to a single *TokenAnnotation*. These *TokenAnnotations* are all enclosed within a single *SentenceAnnotation*. Each *TokenAnnotation* contains a unique label, a text string covered by this annotation and a POS tag. According to the FESL specification in Figure 3, first a *TA* of a *TokenAnnotation* type with covered text *tumor* is searched for. Once it is found (T7), a search is performed for 5 *FAs* of a type *TokenAnnotation* to the left from T7. Only annotations whose POS tag is not *IN, CC, DT or null* are selected during the search. Thus the selected *FAs* will be T6, T4, T2 and T1. The same algorithm applied on the right context of T7 will produce a selection of *FAs* labeled T8, T9 and T10. As has been mentioned earlier, the search for *FAs* is limited by sentence boundaries. For this reason, even though a *windowSizeRight* and *windowSizeLeft* are specified with the value 5, fewer than five *TokenAnnotations* are actually selected.

As opposed to this previous example for machine learning, in the case of feature extraction for a comparative analysis (e.g. evaluation), the *TA* and *FA* usually are the same.

To demonstrate another set of capabilities of FESL, consider a case where it is necessary to process annotations that implement hierarchical models, (i.e., annotations containing other annotations, which may themselves contain annotations, etc.), with multiple levels of containment. The set of particular features that are required for extraction depend on where in the hierarchy the annotation is located and how it is related to the higher level annotation. As an example, we consider a case where it is required to distinguish between a dimension of a surgical margin and dimensions of a tumor. Figure 5 illustrates these capabilities, where the requirement is to extract values of features of *Dimension* annotations that are constituents of *Size* annotations which in turn are properties of two different containing UIMA annotations: *PrimaryTumor* annotations and *MetastaticTumor* annotations. An additional requirement is to extract feature values of all other *Dimension* annotations under a separate label. Figure 5 illustrates how these complicated requirements can be specified with FESL:

```
<targetAnnotations className="PrimaryTumorDimension"
    enclosingAnnotation="SentenceAnnotation">
  <targetAnnotationMatcher annotationTypeName="Size" fullPath="
      PrimaryTumor:Size"/>
  <featureAnnotationMatchers annotationTypeName="Size"
      windowsizeInside="1">
    <groupFeatureMatchers>
      <featureMatchers featurePath="Dimensions:toArray:Unit"
          featureTypeName="String"/>
      <featureMatchers featurePath="Dimensions:toArray:Extent"
          featureTypeName="String"/>
    </groupFeatureMatchers>
  </featureAnnotationMatchers>
</targetAnnotations>
<targetAnnotations className="MetastaticTumorDimension"
    enclosingAnnotation="SentenceAnnotation">
  <targetAnnotationMatcher annotationTypeName="Size"
      fullPath="PrimaryTumor:Size"/>
```

```
  <featureAnnotationMatchers annotationTypeName="Dimension"
      windowsizeInside="3">
    <groupFeatureMatchers>
      <featureMatchers featurePath="Unit" featureTypeName="String"/>
      <featureMatchers featurePath="Extent" featureTypeName="String"/>
    </groupFeatureMatchers>
  </featureAnnotationMatchers>
</targetAnnotations>
<targetAnnotations className="Processed"
    enclosingAnnotation="SentenceAnnotation">
  <targetAnnotationMatcher annotationTypeName="Dimension"
      fullPath="PrimaryTumor:Size:Dimensions:toArray"/>
</targetAnnotations>
<targetAnnotations className="Processed"
      enclosingAnnotation="SentenceAnnotation">
  <targetAnnotationMatcher annotationTypeName="Dimension"
      fullPath="MetastaticTumor:Size:Dimensions:toArray"/>
</targetAnnotations>
<targetAnnotations className="OtherDimension"
    enclosingAnnotation="SentenceAnnotation">
  <targetAnnotationMatcher annotationTypeName="Dimension"/>
  <featureAnnotationMatchers annotationTypeName="Dimension"
      windowsizeInside="1">
    <groupFeatureMatchers>
      <featureMatchers featurePath="Unit" featureTypeName="String"/>
      <featureMatchers featurePath="Extent" featureTypeName="String"/>
    </groupFeatureMatchers>
  </featureAnnotationMatchers>
</targetAnnotations>
```

Figure 5: Dimension extraction sample

In the example above, a path to features of interest that are properties of feature annotations (*FA*) is specified by a sequence of properties/methods that are required in order to locate the final feature. For example:

```
fullPath="PrimaryTumor:Size:Dimensions:toArray"
```

specifies that *PrimaryTumor* contains a property called "*Size*" of a type that has an array of dimensions, and elements of that array should be of type *Dimension* as enforced by the *annotationTypeName* attribute. The first target annotation (TA) with a class label *PrimaryTumorDimension* is specified to be of a type *Size* and located on a path *PrimaryTumor:Size*. This specification ensures that only Size annotations that are constituents of *PrimaryTumor* annotations are matched. Once the *TA* is located, a feature annotation (*FA*) of the same type *Size* is searched for within the offset boundaries of the *TA,* which is enforced by *windowsizeInside* attribute. In this example, the value of *windowsizeInside* attribute is set to 1, guaranteeing that the same *Size* annotation that was previously selected as the *TA* will also be selected as the *FA*. The same rules apply to the processing of target annotations referenced by the *MetastaticTumorDimension* class label. Also in this example, a specification of an arbitrary label "Processed" with no *FA* specification should be noted. This illustrates the functional feature of FESL of excluding annotations (*TAs*) that have been matched during the previous search from further processing. Thus, dimensions matched for tumor sizes will not by considered during the search specified by criteria with label *OtherDimension*.

4. Automated performance metrics evaluation

Comparison of results produced by a pipeline of UIMA annotators to a "gold standard" or results of two different NLP systems is a frequent task, and should be automated. Creating a uniform methodology that would not just simplify the comparison, but would also facilitate the identification of common sources of errors and measure performance improvements gained by correcting these errors, is crucial in the NLP research and development process.

Using FESL as an information extraction mechanism, we developed such a methodology that includes several steps:

- defining the comparison criteria
- extracting the relevant features
- extracting relevant information from two sources to be compared into a spreadsheet-compatible format
- comparative analysis of extracted information

Only the first step has to be done manually; all others can be completely automated.

4.1 Defining the comparison criteria

The definition of the comparison criteria between two CAS structures is a critical step in the evaluation process. Each CAS structure can have its own type system, and the information represented by an individual type from one type's system does not necessarily mirror the information stored in the corresponding type of a different type system. In fact, its constituent parts could be spread across multiple types. For complex types (types that include other types and are also a part of the comparison process), the relevant constituents to be used in the definition of equality must be defined. The result of this step is a set of FESL configuration files and set of custom comparison Excel spreadsheet templates (*CST*). The FESL configuration files specify the feature extraction, whereas the templates implement the comparison criteria.

It is within the CST's that the comparison between two CAS structures is executed. Information from both structures is loaded into a CST, and then the comparison is implemented with a set of macros that perform the following:

- compare two corpora based on the user defined equality criteria
- calculate performance metrics such as precision, recall and F-score.

In addition they could include macros to take into account errors in the "gold standard" or estimate the performance gain by fixing a specific algorithm or implementation errors in automated annotators.

4.2 Feature extraction

Feature extraction is performed using a custom UIMA CAS consumer that uses a FESL configuration file and a CAS structure as its input and outputs delimited files with feature values. This CAS consumer contains code which interprets and executes the FESL configuration. A sample

of such an implementation will be released into Open Source as part of the Apache UIMA incubator project. The fundamental semantic rules implemented by FESL were covered in section 3. Features are extracted from both sources that are being compared, resulting in two delimited files that are merged into a single file. This process uses the offsets of annotations within the document to guide the merger. The merged file can be easily imported into a custom Excel spreadsheet for further analysis, as discussed in section 4.1. In our environment, the creation of a spreadsheet from two delimited files is completely automated. Figure 6 shows typical content of a merged file with feature values extracted from two sources. We used a vertical bar ("|") character as the value separator, since our data can never contain one—for use with other data sets, this can be customized accordingly:

```
$ head  set1-SizeDim-report.txt
18|24|4.0|cm|18|24|4.0|cm|gold/doc0.fve|medtas/doc0.fve
40|47|12.0|cm|40|47|12.0|cm|gold/doc0.fve|medtas/doc0.fve
106|118|6.5|cm|106|118|6.5|cm|gold/doc0.fve|medtas/doc0.fve
112|118|2.0|cm|112|118|2.0|cm|gold/doc0.fve|medtas/doc0.fve
249|261|3.8|cm|249|261|3.8|cm|gold/doc0.fve|medtas/doc0.fve
255|261|2.5|cm|255|261|2.5|cm|gold/doc0.fve|medtas/doc0.fve
275|281|5.0|cm|275|281|5.0|cm|gold/doc0.fve|medtas/doc0.fve
182|187|30|cm|182|187|30|cm|gold/doc10.fve|medtas/doc10.fve
211|216|20|cm|211|216|20|cm|gold/doc10.fve|medtas/doc10.fve
72|85|0.05|cm|72|85|0.05|cm|gold/doc100.fve|medtas/doc100.fv
```

Figure 6: Merged results of feature extraction

4.3 Comparative Analysis

Comparative analysis usually includes several steps – calculation of performance metrics, error analysis, and evaluation of the most effective ways of improving accuracy (e.g. identification of types of errors and corrections that would maximize accuracy). As was mentioned earlier, the calculations are done automatically by macros, while error analysis and evaluation, for the most part, must be done manually. One way that the evaluation can be partially automated is that one of the implemented macros allows errors to be classified according to a code (e.g., errors in the gold standard vs. errors in the automatic annotations) and performance metrics recalculated based on these error codes.

5. Using feature extraction for machine learning

Machine learning algorithms build and apply models to extract pertinent information from sources such as a text documents or images (Mitchell, 1997). In addition to the machine learning algorithm, the process of defining of ML feature set itself is a critical factor in building accurate models of the information to be identified. In general, extensive experimentation with a variety of parameters is done to create models which perform with the desired accuracy for a particular task.

The complexity of feature extraction varies, but it is desirable to have a comprehensive mechanism to rapidly extract them. CFE is such a mechanism for textual data sources. In section 3 we described FESL and its semantics, which can be used to specify which features should be extracted. In this section we will describe some details as they pertain to feature extraction within the machine learning domain.

In particular, information from a surrounding context of a specific term has to be taken into account, and additionally, that context can be constrained by multiple conditions specific to the task. Design of FESL takes such considerations into account by allowing specification of fairly complex and precise criteria for locating and extracting particular pieces of information. For Word Sense Disambiguation (WSD), in addition to the FESL configuration, we developed a CAS consumer that generates machine learning models and AEs that evaluate the models within a classification task.

One of the steps in building models for machine learning for textual data is generation of parameter sets from a text corpus. The syntax and semantic of FESL, as previously described, is sufficient for this task. The generated parameter set contain individual machine learning features (*MLFs* - not to be confused with UIMA features) whose symbolic names are constructed from values extracted according to FESL specification. In cases where more then one UIMA feature value is extracted for a particular *MLF,* the extracted values are concatenated to produce a MLF symbolic name. For instance when extracting size information from a context of a term to be disambiguated we could produce an MLF that is presented as "L1_Size_53_58_25_cm" which is a combination of an annotation type that the information was extracted from (Size), numeric extents for three dimensions (53, 58, 25) and a measurement unit (cm). It should be noted that prefix "L1" indicates that FESL configuration specified to include a position of a MLF relative to the term into the MLF name. A position is characterized by direction and distance, thus "L1" should be read as "first size annotation to the left from the term to be disambiguated". In cases where neither the distance nor the direction is required to be a part of an MLF name it will be prefixed with "X0". Figure 7 shows a typical content of an MLF file for WSD:

```
$  cat ml_feature.txt
X0_Dimension_12_cm
X0_Size_45_50_12_cm
X0_Dimension_45_cm
X0_Size_4_6_5_cm
X0_Dimension_10_cm
X0_Size_20_35_10_cm
```

Figure 7: Sample of MLF file for WSD

6. Conclusion

In this paper, we proposed CFE (Common Feature Extraction), a methodology and system for testing and evaluating complex NLP applications executed within the UIMA framework. The core of the system is a declarative

language FESL, and a UIMA component that processes FESL specifications, using them to guide extraction of data from a UIMA CAS in a completely generalized way, and providing a method for subsequent processing to format the output as needed for any downstream use. In addition, CFE can be used to rapidly specify and extract features to build models for machine learning algorithms. The flexibility and ease-of-use of the system enables easy experimentation with different models in the machine learning space. CFE was used in quite different tasks: experimenting with large numbers of feature sets to build models for word sense disambiguation, evaluating a sizable set of parameters for dictionary lookup and evaluating the automatic filling of hierarchical knowledge models. The comparison spreadsheets proved to be invaluable in determining which algorithmic improvements would result in the most substantial improvements in precision and recall.

For a next step, a GUI for generating FESL configuration files is planned. Other possible extensions are automating the process of building refined models and automatically evaluating them. The CFE system, the FESL declarative language specification and the UIMA component to interpret it will be released into Open Source as part of the Apache UIMA incubator project.

7. Acknowledgements

We thank Rie K. Johnson for her support and Wei Guan for experimenting with CFE.

8. References

Coden A.R., Savova G.K., Buntrock J. D., Sominsky I.L., Ogren P.V. , Chute C.G., de Groen P.C. (2007) *Text* Analysis Integration into a Medical Information Retrieval System: Challenges Related to Word Sense Disambiguation: *Medinfo 2007*

Mitchell Tom. (1997). *Machine Learning*, McGraw Hill

Ogren, P.V. (2006). Knowtator: A Protégé plug-in for annotated corpus construction. Rochester, MN. *Abstract for HLT-NAACL 2006.*

Savova G.K., Coden A.R, Sominsky I.L., Johnson R.K., Ogren P.K., de Groen P.C. and Chute C.G. (2008). Word Sense Disambiguation across Two Domains: Biomedical Literature and Clinical Notes. To appear in *Journal of Biomedical Informatics*