UIMA: Unstructured Information Management Architecture for Data Mining Applications and developing an Annotator Component for Sentiment Analysis

Jan Hajič, jr.
Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics
Malostranské nám. 25, 118 00 Prague
Czech Republic

Kateřina Veselovská
Charles University in Prague
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics
Malostranské nám. 25, 118 00 Prague
Czech Republic

Abstract. *In this paper we present UIMA – the Unstructured Information Management Architecture, an architecture and software framework for creating, discovering, composing and deploying a broad range of multi-modal analysis capabilities and integrating them with search technologies. We describe the elementary components of the framework and how they are deployed into more complex data mining applications. The contribution is based on our experience in work on the sentiment analysis task for IBM Content Analytics project. Note that most of the information on UIMA in this paper can be obtained from UIMA documentation; our main goal is to give the reader an idea whether UIMA would be helpful for her or his task, and to do this in less time than reading the documentation would take.*

1 Introduction

UIMA is an acronym standing for Unstructured Information Management Architecture. "Unstructured information" means essentially any raw document: text, picture, video, etc. or a combination thereof. Unstructured information is mostly useless – a string of five digits will not tell us much unless we know it's a ZIP code and not a price tag. In order to make an unstructured document useful, we must first discover this type of meaning in the unstructured document and make this additional information available for further processing.

The process of making sense of unstructured information is in UIMA called annotating. Components that discover the hidden meanings and store them as annotations are, predictably enough, called annotators. The point of UIMA is to provide a common framework so that multiple components can be linked to create arbitrarily rich annotations of the unstructured (semantically "flat") input. This is the most important contribution of UIMA: a very flexible way of passing analysis results from one component to another, and so gradually discovering and making use of more and more information contained within the unstructured document.

The interoperability of various components is achieved by defining the Common Analysis Structure, CAS, and its interfaces to various programming languages (most notably JCAS for Java). The CAS is the tumbleweed that cascades through the various annotators, each of which adds its annotation results to the CAS. The CAS can also pass more information than annotations of document regions: the representation scheme for analysis results is very general and supports such information as "the annotation span 43-47 and 55-59 are references to the same company".

The component that houses an annotating pipeline is called an Analysis Engine. An Analysis Engine at its simplest contains just one annotator and is called a *primitive* AE. AEs that house more annotators are called *aggregate* engines. Also, an Analysis Engine can be composed of multiple other AEs. The Analysis Engine is the component that is actually *run* by the framework. The simplest AE is just this runnable wrapper of an annotator.

Individual annotators are usually used for small-scale, granular tasks: language identification, tokenization, sentence splitting, syntactical parsing, named entity recognition, etc. Analysis Engines are typically used to encapsulate "semantic" tasks on raw documents (or on somehow meaningful levels of document annotation, such as a document after linguistic analysis or a picture after segmentation), such as document-level sentiment analysis. Both annotators and Analysis Engines are easily re-usable in various UIMA pipelines.

UIMA is currently an Apache project, meaning it can be freely downloaded at http://uima.apache.org. (Previously, the architecture was proprietary to IBM. IBM still uses UIMA extensively in its applications like Content Analytics and Enterprise Search.)

The architecture also provides facilities to wrap components as network services and scale up to very large volumes by running annotation pipelines in parallel over a networked cluster. This is done through a server that provides analysis results as a REST service.

A number of annotators and other components is available as a part of the UIMA Sandbox from the Apache project. Others are lying around the internet.

UIMA is not oriented towards data mining research, although it is universal enough to be used as such. There are no built-in facilities for evaluating data mining performance.

For our sentiment analysis project, we have not worked with other media in UIMA than text, so we will limit ourselves to text analysis in this paper. However, UIMA is capable of multi-modal analysis as well.

Apache UIMA is very well-documented, from our experience significantly better than the IBM applications using it.

1.1 UIMA and GATE

The GATE (General Architecture for Text Engineering) is a project that has significant overlaps with UIMA and can embed UIMA Analysis Engines. However, as opposed to UIMA, its primary audience are not software developers, but researchers and businesses. As opposed to UIMA, GATE offers a front-end for „pointing and clicking“ in the GATE developer, a cloud and an information extraction component, ANNIE. However, GATE doesn't have multimodal capabilities.

We have no experience with GATE, but it seems the system is intended for a different audience: while UIMA focuses on the annotating pipeline and making the software developer's and natural language engineer's job easier, GATE is much more comprehensive and covers a complete application.

GATE provides an interface to incorporate UIMA Analysis Engines.

2 UIMA Components Overview

UIMA is a software architecture which specifies component interfaces, data representations, design patterns and development roles for creating multi-modal analysis capabilities.

The *UIMA framework* provides a run-time environment in which developers can plug in their component implementations and with which they can build and deploy unstructured information management applications. The framework is not specific to any IDE or platform; Apache hosts a Java and a C++ implementation of the UIMA Framework.

The *UIMA Software Development Kit (SDK)* includes the UIMA framework, plus tools and utilities for using UIMA. Some of the tooling supports an Eclipse-based (http://www.eclipse.org/) development environment. These tools (specifically the Component Descriptor Editor and JcasGen, see below) proved to be extremely useful for orienting ourselves in the complex interfaces of annotation components.

There are two parts to a *component*: the code and the *component descriptor*, which is an XML document describing the capabilities and requirements of the component. The descriptor holds information such as the input and output types, required parameters and their default values, reference to the class which implements the component, author name, etc. The Component Descriptor Editor for Eclipse IDE is a tool for creating component descriptors without having to know the XML. The descriptor of a component serves as its declared interface to the rest of UIMA.

2.1 Types, CAS and SOFAs

A subject of annotation is called a SOFA. SOFAs can be texts, audio, video, etc. Annotations are anchored to the SOFAs. An annotator may work with any number of SOFAs it gets, even create new ones. Typically, SOFAs going through the pipeline together will be of more modalities, like a news story and an associated picture, or any other group of unstructured documents that we wish to process together to discover relevant information.

Discovered structured information about the SOFAs are kept in *types*. A type is anything: Company, Name, ParseNode, etc. Types are domain-, application- and (unless some coordination/sharing is involved) developer-specific. Each type has an associated *feature structure.* The feature structure holds additional information about the annotated span; for instance, the Company type may have a feature structure that holds whether the company is publicly traded, who its CEO is, where is it based, etc. The feature structure also may be empty. A type can also be a subtype: the type class can inherit from another (multiple inheritance is not allowed).

Types that are associated with a specific region of the SOFA are called *annotation types*. These types have a *span*, a start and end feature which delimit the annotated region.

A non-annotation type could be a Company, a type describing all there is to know about a company mentioned in the various SOFAs. Annotation types could be then various CompanyNameAnnotation, CompanyCEOAnnotation, etc. Our analysis goal could be to recover whatever there is to know about companies mentioned in a collection of SOFAs; we would gradually annotate the SOFAs by the annotation types and finally put all the pieces together into the Company type.

For working with type systems, the UIMA SDK provides a Type System Descriptor Editor within the Component Descriptor Editor tool. Defining types then becomes more or less a point-and-click operation. Additionally, once the type system descriptor is done, the JCasGen utility (also a part of the UIMA SDK for Eclipse) automatically generates the appropriate classes for the annotation types themselves, so that the user never needs to do anything with the types but define them in the Descriptor Editor.

The whole bundle of SOFAs, annotations and whatever else goes through the pipeline is housed inside a *Common Analysis Structure* (CAS). This class is passed from one annotator to another, from one AE into another and is available at the end for some consumer to use the discovered information however it wishes. The CAS class provides iterators to each annotation type and allows annotators to do whatever they wish to them (an annotator may even clear results of previous annotators, so as to keep the CAS uncluttered with intermediate steps that never get used later). An example of an UIMA multi-modal processing pipeline is in Fig. 1:

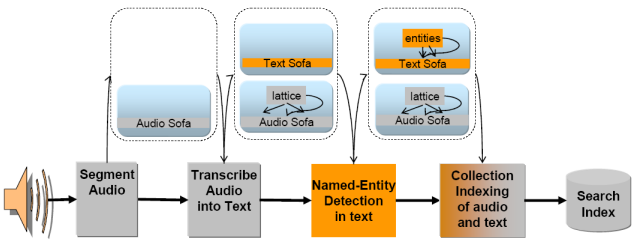

Fig. 1: multi-modal processing pipeline

2.2 Annotators

Annotators are the innermost components in an UIMA processing pipeline. They are the components that actually discover the information from the unstructured document: all the "interesting" things happen in annotators.

UIMA provides the base class for an annotator. At its simplest, all one needs to do to implement an UIMA annotator is to override the *process(...)* method of the base class.

Annotating in UIMA means attaching UIMA annotation *types* to certain spans of text (more precisely, creating the annotation type with attributes that denote which region in the associated document is "responsible" for this particular piece of information). The definition of input and output types – the annotator *type system* – is the critical decision in designing an annotator.

Annotators have two sources of information they can work with: the CAS that runs through the pipeline, from which an annotator gets the document and previously done annotations, and the UIMA Context class, which contains additional information about the environment the annotator is run in: various external resources, configuration parameters, etc. All the inputs an annotator needs to run are described in its component descriptor.

An example annotator could be a tokenizer, an annotator responsible for segmenting text into tokens. It may simply delimit tokens, or it may also add features such as lemma, part of speech, etc. The simpler annotator will require nothing and output the type, let's say, TokenAnnotation with only the span defined and no associated features. The more complex tokenizer will require no input types and maybe an outside resource with a trained statistical model for lemmatization and POS tagging; it will output the type ComplexTokenAnnotation with a feature structure containing the lemma and part of speech features. Maybe the annotators should require an input LanguageAnnotation type that is associated with the whole document and its feature structure has a *language* feature, containing some pre-defined language code, so that the tokenizer knows which statistical model to load. This LanguageAnnotation might be the output of a language recognition algorithm implemented by another annotator further up the pipeline. There can instead be a LanguageSpanAnnotation type can also be designed to allow for multi-lingual documents, by actually giving it a span. The Tokenizer will then iterate over those LanguageSpanAnnotations and will load a different model for each of them, etc.

This example only illustrates the flexibility and ease of use for UIMA: other than the type system, there is no restriction on what the components actually do inside. The input and output of annotators is standardized by the UIMA framework, so that if you think you have a better language identifier which uses a completely different algorithm, you can plug it in and as long as it keeps outputting the LanguageAnnotation type, nothing needs to be changed for the tokenizer. This is no magic – any good programmer knows how to keep things modular – but the point is, UIMA already does this for free, and with a great amount of generality.

2.3 Analysis Engines

The basic blocks in the UIMA architecture that "do something" are the analysis engines. At their simplest, an Analysis Engine simply wraps an annotator so that the UIMA framework can run the annotator inside. Analysis Engines with a single annotator are called *primitive* AEs. An *aggregate* Analysis Engine links more annotators together into a pipeline:

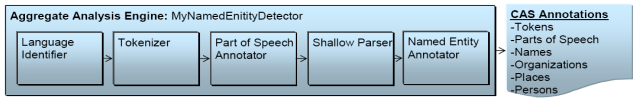

Fig.2: Aggregate Analysis Engine

Analysis Engines provide encapsulation: the only thing a developer needs to know is its input and output types (and some technical information), described in its component descriptor. This enables the developer to aggregate AEs into a more complex UIMA application, perhaps a hierarchical one where top-level AEs consist of multiple sub-engines. These sub-engines are called *delegate AEs*. Generally, Analysis Engines can be thought about as CAS-in, CAS-out processing components and any two where the outputs and inputs in the CAS match can be linked together.

Starting from UIMA 2.0, there is a *flow control* facility of UIMA which can even make decisions, based on what the processing pipeline has come up with so far, as to which analysis engine to use next. (We have no experience with this, though.) The UIMA framework, given an aggregate analysis engine descriptor, will run all delegate AEs, ensuring that each one gets access to the CAS in the sequence produced by the flow controller.

The UIMA framework is also equipped to handle different deployments: the delegate engines, for example, can be tightly-coupled (running in the same process) or loosely-coupled (running in separate processes or even on different machines). The framework supports a number of remote protocols for loose coupling deployments of aggregate analysis engines, including SOAP (which stands for Simple Object Access Protocol, a standard Web Services communications protocol).

3 Scenarios of UIMA Applications

How the components described in the previous section can be fit together to actually do something and what is the "division of labor" between the UIMA framework and the developer is best described in images.

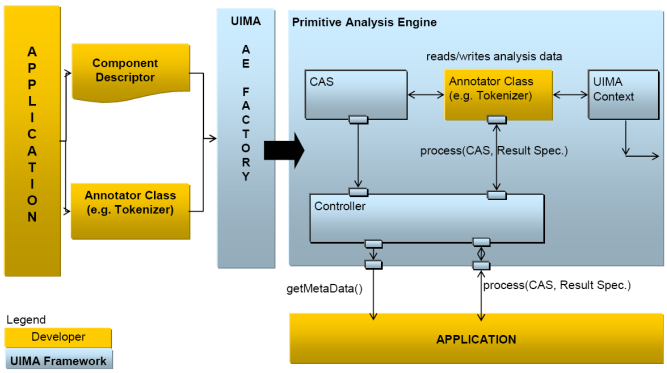

Fig. 3: An elementary UIMA application

This is the elementary deployment scenario. The developer determines what she or he wants to annotate, creates the appropriate Type System descriptor and incorporates it into the Component Descriptor, implements the annotator itself and passes this information to the UIMA Analysis Engine factory. This factory is a part of the UIMA framework. The factory then takes the information from the Component Descriptor and the class and instantiates the Analysis Engine.

The UIMA framework provides methods to support the application developer in creating and managing CASes and instantiating, running and managing AEs. However, since our work is only to implement the annotator class and provide the descriptor into an IBM application, we have no experience with actually using an Analysis Engine.

3.1 Collection Processing

Typically, the application will be processing multiple documents, a *collection*. This presents additional challenges in creating a *collection reader* and managing the iterative workflow (distributed processing, error recovery, etc.) Almost any UIMA application will have a source-to-sink workflow like in Fig. 3:

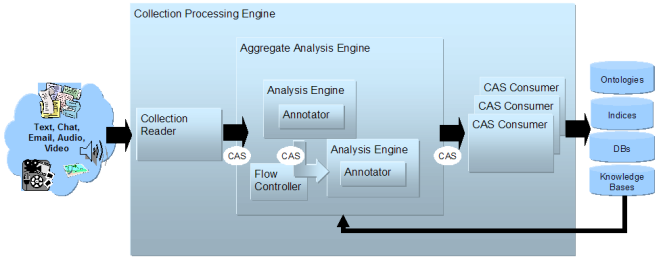

Fig. 4: UIMA collection processing workflow

UIMA supports this larger operation through its *Collection Processing Engines*. This support is through two additional components: the *Collection Reader* and the *CAS Consumer,* and a Collection Processing Manager. The Collection Reader is responsible for feeding documents into the Analysis Engine. The CAS consumers are, predictably enough, responsible for processing the output CASes – for instance, indexing them for a semantic search application, or in a sentiment analysis setting, track opinion trends. The CPEs, as any UIMA component, need an associated Component Descriptor.

UIMA provides some CAS consumers for semantic search and a simple consumer for storing the data into an Apache Derby database.

Creating a CPE is a process analogous to creating an Analysis Engine; of course, the sum of what needs to be configured is different. The following figure illustrates the analogy:

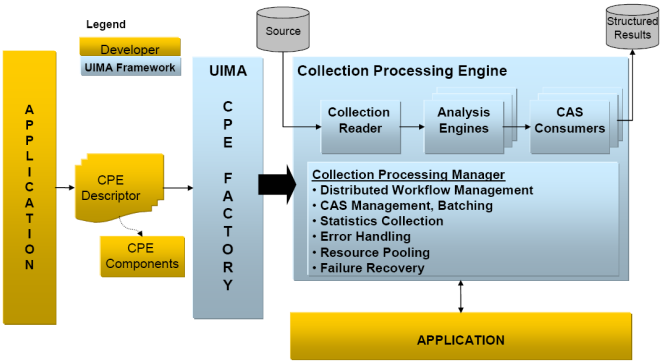

Fig. 5: An UIMA application with a Collection Processing Engine

The descriptor for a CPE specifies things like where to read from, how to do logging, what to do in case of error or what the control flow among various Analysis Engines inside the CPE should be.

Collection processing is the "final step": this is UIMA at its fullest.

4 Annotating Sentiment with UIMA

We are working on a Sentiment Analysis annotator for Czech for IBM Content Analytics (ICA). The task of Sentiment Analysis is currently two-fold. First, we determine whether a segment of the input document contains an evaluative statement: whether a sentence says something nice or bad about about something. Second, we determine the orientation of the segment – whether the statement is positive, or negative.

We work with lexical features using statistical methods, essentially estimating the precision of individual words in predicting a sentiment class (negative/neutral/positive). The precision of a word w for a sentiment class c is estimated as $P(c|w) = N(c,w)/N(w)$, where $N(w)$ is the token count of w in the data and $N(c,w)$ is the count of w in data tagged as class c. From this number, we subtract $P(c)$ to obtain an *additive-precision* aprec(w,c). Classification is done in two stages: determining whether a segment is polar or non-polar and if deemed polar, segment orientation is determined. For a segment $S = (w_1, w_2, \ldots w_{|S|})$, in each stage, we take the argmax of the sum of aprec(w,c) for all w in S to be the output label (first either polar or nonpolar, then, if classified as polar, positive or negative).

How to implement sentiment analysis in UIMA? We are only creating an annotator, so we do not need to concern ourselves with application deployment. All that needs to be

done is defining the inputs and outputs. We have determined the input type system from an IBM Content Analytics developer tool that was provided to us through the IBM Academic Initiative. The output types are ours to define. There is one external resource our annotator will need to access: the statistical model of lexical units' precision.

4.1 Creating the Component Descriptor

Since we used lemmatization to increase feature recall, we need lemmatized input. Also, we need some sentence segmentation, in order to determine which segments our classifier should operate on. The ICA standard UIMA pipeline has a lexical analysis component that outputs types *uima.tt.TokenAnnotation* and *uima.tt.SentenceAnnotation* (this component is available in the ContentMaker component from the UIMA Sandbox). We will need the *lemma* feature of *TokenAnnotation*.

On output, we will provide both the token sentiment annotations and the sentence annotations. We named the corresponding types *UFALSentimentToken* and *UFALSentimentSentence*. The token's feature structure consists of the Lemma, the lexicon tag for negation and various precision statistics for the polarity classes. The sentence annotation type contains cumulative statistics over the tokens that span regions contained within the sentence span and a final classification statement.

Our external resource is a CSV file that we pack in one JAR together with the annotator. The default parameter value for the UIMA Context, through which the annotator can access this resource, is provided in the Component Descriptor as well.

Creating the component descriptor using the Component Descriptor Editor did not take more than several minutes. Aside from the type system and the external parameter specification, we only needed to provide the relative path to the annotator class.

4.2 Code

Creating the annotator code then consisted of running the JCasGen utility to generate from the type system the classes that go into the CAS as annotations and writing the algorithm itself. The only UIMA-specific code that had to be written raw was reading from the CAS and adding annotations to it (no more than some 10 lines of code). On the UIMA side, development was easy; on the IBM side, we are still encountering interoperability issues.

5 Conclusions

Our experience with UIMA is not very extensive: currently, our task is simply to implement a sentiment annotator for the IBM Content Analytics application. However, according to our up-to-date knowledge of the system, we are convinced that UIMA is a very thorough, flexible and robust framework. Moreover, the documentation of UIMA is *extremely* good.

The UIMA SDK does its utmost to relieve the developer of tedious, repetitive tasks through utilities like the JCasGen and the Component Descriptor Editor, as long as the Eclipse IDE is used. These utilities make it easy to start working with UIMA. The SDK does its best to help the developer focus on the meaningful parts of the task at hand only: on implementing the algorithms that discover information inside the unstructured data.

This is, we feel, the greatest contribution of UIMA: standardizing and automating the common parts of more or less any data mining application and providing an easy enough way of filling in the “interesting bits”, while at the same time being flexible enough to meet most application demands (multi-modality, complex control flow, etc.) At the same time, it also provides robust runtime capabilities.

Also, UIMA aggregate AEs enable and encourage granularity, so individual annotators can be designed so that they do not require more than one person to implement them reasonably fast. Therefore, once the component descriptors are agreed upon, teamwork should be easy.

A weak point may be flexibility on the programmer side: as soon as one strays from a development scenario where the UIMA SDK tools are useful, the amount of work necessary to get an UIMA application up and running increases dramatically. We also do not know how complicated it is to administer an UIMA application.

UIMA might not be the framework of choice in an academic, experimental setting, since it provides no facilities for evaluating the performance of the data mining algorithms inside and is probably unnecessarily complex for most experimental scenarios. Implementing such an evaluator, however, might not be too difficult, either as an external application operating on the database one of the available CAS consumers generates, or as an UIMA component, using some of the semantic search CAS consumers. Furthermore, given UIMA flexibility, modularity and re-usability, if such an evaluation component was present, an UIMA CPE could be a great way of testing data mining algorithms in various complex settings.

We are convinced that UIMA is worth knowing about.

Acknowledgment

The work on this project has been supported by IBM Academic Initiative program. Text and images from the UIMA documentation (http://uima.apache.org/documentation.html) have been used. The sentiment analysis research is also supported by the GAUK 3537/2011 grant and by SVV project number 267 314.