IYAS

Interactive sYstem for Answer Selection

Qatar Computing Research Institute

CONTENTS

CHAPTER 1

INTRODUCTION TO IYAS

1.1 What is IYAS?

IYAS, which stands for Interactive sYstem for Answer Selection, is a framework for:

- deep natural language analysis

- extraction of features from textual content

- construction of structural representations of text which are enriched with syntactic and semantic information provided by the analysis components

- generation of training and test data for classification and reranking tasks

- instantiation of intelligent applications such as the Question Answering system built on top of the framework

1.2 The IYAS Framework

Iyas leverages several libraries and frameworks in order to reduce development effort and to reach a standard in the definition of the analysis components and the data they manipulate. The following sections describe the projects Iyas builds upon.

1.2.1 Apache UIMA

UIMA is a framework for building applications which manage unstructured information. It integrates single analysis components and their logic into bigger processing units, with immediate benefits in terms of reuse, scalability and extensibility. At the framework core there is the idea that every computation on some data can be encapsulated in a component. What this component requires in input in addition to the data, and what it outputs as a result of its work, can be defined using a standard XML format. Once developed, components can be integrated declaratively. An XML descriptor orchestrates the components with different degrees of complexity, enabling sequence flows or conditional flows.

The main UIMA elements are the Analysis Engines, the Annotators, the Type System and the Common Analysis Structure.

An **Analysis engine** (AE) is a program that analyses artefacts and infers information from them. The building blocks of AEs are annotators. The **Annotator** is the basic UIMA primitive for building useful applications and can be considered an analysis engine which performs a single analysis task. Indeed the annotator takes a piece of data and metadata associated with it, performs its analysis and produces results in form of annotations. An annotation has a type, which can be generic or specialized, and is associated with indexes specifying the range of characters covered by the annotation. The annotator should have clear inputs and outputs, and its job should be simple: complex tasks can be decomposed into smaller tasks to favour simplicity and reuse. Nevertheless, nothing stops an annotator from performing multiple and heterogeneous analysis tasks.

The output of an annotator is typed and in the form of **Feature Structures**. These are simple data structures with a type and a set of attribute and value pairs. UIMA has a comprehensive set of built-in data types, spacing from simple types to list of objects. Additional types can be defined and thus, even complex structures can be described.

At the core of this framework feature there is the **UIMA Type System** which is a declarative definition of an object model. It serves the purposes of specifying which metadata can be stored in the Common Analysis Structure (CAS), described later, and what is the behaviour of a component in terms of the types expected in the input CAS and the types produced in output.

An annotator is composed by:

- an XML definition file;

- a class containing the analysis logic;

- optionally, an XML file describing the types used by the annotator.

This section can indeed be included in the XML definition file, or specified in an external file. The definition file contains all the necessary information for integrating the annotator in the framework. An annotator can be primitive or aggregate. A primitive annotator performs operations on the data and cannot use others annotator. A class implementing these analysis operations must be supplied to the annotator, which will instantiate the class at run-time. This class implements the interface of UIMA annotators and provides the behaviours for its initialization, for the processing of data and for its destruction. The types used by the annotator are declared in the definition file, or they can be factored out in a specific XML file.

The **UIMA Type System** is hierarchical and usually for defining new types it is sufficient to set the uima.tcas.Annotation type as supertype. The concepts of type and annotation overlap. The name of the new type already carries the semantics of the annotation, and the UIMA Annotation supertype has slots for storing the range of the text covered by the annotation. For example, a type named Person for annotating people mentioned in the text should be declared as a subtype of $uima.tcas.Annotation$. Whenever the annotator finds a person in the text it must only add in the CAS a new Person annotation, after setting the initial and final character of the occurrence of that person in the text. If a user defined type needs more attributes, type descriptions of arbitrary complexity can nevertheless easily produced by writing them in the XML or preferably using the visual tools provided by the framework. **UIMA Capabilities** describe the types required in input by an annotator and the types produced in output. This information enforces the constraints on the execution order of annotators. If an annotator requires the presence in the CAS of metadata produced by another annotator, the first annotator must be called earlier than the second in the component chain. As a side note, the absence of standards to describe uniformly a given annotation type and the flexibility of the UIMA Type System induce the user to define a personal type system. Thus, every set of components live in a type ecosystem different from the others, with its own namespace, annotation names and semantics. This leads to an unavoidable integration effort for letting dialogue different components in the same analysis engine. For example, different NERs for entities of type Person will likely define different annotation types. Annotators using the output of one NER require a conversion of the output of the other, in order to exploit both NERs results. Alternatively, the user can convert the different types into a general one, and use the latter in input for other components. An aggregate annotator cannot performs operations on the data but can call other annotators. Thus, an aggregate annotator is the mean for creating a pipeline for NLP analysis. The annotators to call are specified in the component descriptor. The flow between components, always specified in the descriptor file, can have several connotations. The Fixed Flow executes the components sequentially in the specified order. The Capability Language Flow allows to execute analysis engines only if they can produce in output the capabilities specified for the language of the current document, and only if these capabilities have not been produced by another engine in the flow. The inputs and outputs of annotators are referred to as capabilities in UIMA. The User-defined Flow enables the inclusion of user created flow controller. These controller can inspect the CAS content at run-time and consequently modify the flow in arbitrary complex ways.

The **Common Analysis Structure** is a data structure containing a unit of unstructured data and all the data inferred by the analysis carried on that data. The CAS instantiation

is based on an Analysis Engine. A CAS can contain only the types used in the analysis engine. Using other types raises errors. In UIMA applications, the CAS is initialized with the unstructured data and it is passed to an Analysis Engine. After the analysis execution, the CAS will contain all the produced metadata.

1.2.2 DKPro

DKPro is the Darmstadt Knowledge Processing Software Repository which contain reusable software for carrying out applications for content analysis. It includes, among other things, DKPRO Core, which provides a set of ready to use software components for natural language processing, based on the Apache UIMA framework; DKPro Similarity, an open source software package for developing text similarity algorithms.

1.2.3 uimaFIT

Configuring UIMA components is generally achieved by creating XML descriptor files which tell the framework at runtime how components should be instantiated and deployed. These XML descriptor files are very tightly coupled with the Java implementation of the components they describe. It is very difficult to keep the two consistent with each other especially when code refactoring is very frequent. uimaFIT provides Java annotations for describing UIMA components which can be used to directly describe the UIMA components in the code. This simplifies refactoring a component definition. It also makes it possible to generate XML descriptor files as part of the build cycle rather than being performed manually in parallel with code creation. uimaFIT also makes it easy to instantiate UIMA components without using XML descriptor files at all by providing a number of convenience factory methods which allow programmatic/dynamic instantiation of UIMA components. This makes uimaFIT an ideal library for testing UIMA components because the component can be easily instantiated and invoked without requiring a descriptor file to be created first. uimaFIT is also helpful in research environments in which programmatic/dynamic instantiation of a pipeline can simplify experimentation.

1.2.4 QCRI ALT components

The QCRI ALT group has developed several NLP components for Arabic. IYAS has preliminary support for the QCRI Arabic normalizer, and the Arabic Part of Speech Tagger Library, which provides a Tokenizer, a POS Tagger, a Named Entities Tagger and a Gender and Number Tagger.

1.2.5 iKernels components

IYAS integrates several components built by the iKernels group at the University of Trento. These components are mostly focused on question analysis and tree kernels: there are a Question Category classifier, a Question Focus classifier, a module for computing Partial Tree Kernel similiarity between trees, an LDA (Latent Dirichlet allocation) module for characterizing the topics contained in the text.

1.2.6 Limosine project

Limosine is an European project that integrates the research activities of leading researchers across diverse topics with a view to enabling new kinds of language-based search technology. The University of Trento is responsible of the information extraction through deep linguistic analysis module of the software platform developed in the project context. The main objective of the module is to provide semantic representations based on deep linguistic processing. The semantic model is based on disambiguated entities, relations between them, subjective expressions, opinion holders and relations among these pieces of semantic information. IYAS supports the NLP components developed within Limosine.

CHAPTER 2

IYAS INSTALLATION

2.1 UIMA

Go to *https://uima.apache.org/* and download the Apache UIMA Java framework & SDK. Decompress the archive and set the `&UIMA_HOME` environmental variable to the path of the UIMA distribution.

2.2 Setting up Eclipse

It is recommended to use Eclipse for developing with IYAS. Download and install the Eclipse IDE at *https://www.eclipse.org/*.

Then, you will need to add Maven support to Eclipse. Download and install the m2eclipse Maven plugin at *http://eclipse.org/m2e/download/*. Some versions of Eclipse come with integrated Maven support so this step could be unnecessary.

You will need some additional UIMA plugins. The instructions for installing Eclipse EMF can be found here: *http://tinyurl.com/UIMA4ECLIPSE*. The useful UIMA visual tools (UIMA tooling) can be found at this software update address:
http://www.apache.org/dist/uima/eclipse-update-site/.

2.3 Obtaining IYAS

2.3.1 Getting IYAS from Github

IYAS can be downloaded from the private Github repository of the Qatar Computing Research Institute. You must request permission to access this repository. You can clone the repository with the command:

```
git clone git@github.com:Qatar-Computing-Research-Institute/Iyas.git
```

2.3.2 Getting the IYAS distribution

Access to a packaged distribution of IYAS (a compressed snapshot of the Github repository) is given at this link:

```
https://dl.dropboxusercontent.com/u/12803333/Iyas-master.zip
```

2.4 Importing IYAS in Eclipse

IYAS is a Maven project. You should import it into Eclipse using the Eclipse import wizard. Select `File -> Import -> Existing Maven Projects`. Make sure Eclipse is using the 1.7 version of the JDK.

2.5 Setting up UIMA types

Type classes are not included in the distribution. They are automatically generated by the UIMA JCasGen tool from the typesystem description. Under the `resources/` folder of the IYAS distribution there is the `generate-types.sh` script. Run this script to automatically generate the type classes for most of the types. Make sure your `$UIMA_HOME` environmental variable is set.

The QuestionFocus type requires manual intervention. Open the Component Descriptor Editor right-clicking the `desc/Iyas/QuestionFocus.xml` definition file. Go to the `TypeSystem` tab and click the `JCasGen` button.

After all the types are generated you can run `setup-types.sh`. This script copies the types to the paths used by Eclipse to load classes for launching the executable programs and unit tests.

2.6 Downloading and installing additional components

The IYAS distribution contains several scripts for downloading and installing external software.

2.6.1 QCRI and Limosine software

Go to the `resources/` folder.

`download-resources.sh` will download the archives containing the QCRI and Limosine software. QCRI software can be found at *alt.qcri.org*. Limosine is hosted on a cloud service.

`install-resources.sh` will decompress the archive, move files and carry out some installation procedures and compiling.

2.6.2 SVMLightTK

IYAS makes use of SVMLightTK for learning models and classifying/reranking instances. SVMLightTK is an extension of SVMLight supporting tree kernels. IYAS contains two SVMLightTK distributions: the binary tools and the Java Native Interface library to be used from Java code.

Go to `tools/SVM-Light-1.5-rer/` and type:

```
make clean && make.
```

This will compile the binary tools.

Go to `tools/SVM-Light-TK-1.5.Lib/` and type:

```
make clean && make.
```

This will compile the JNI library. If the make process says the `jni.h` file cannot be found, fix the path of the JDK distribution in the Makefile according to your system. You can use the command locate `jni.h` to find the path of the right JDK distribution. Another approach to solve the issue is to make sure your `$JAVA_HOME` directory is defined.

2.7 Installation troubleshooting

2.7.1 Maven problems

These are some exceptions you can encounter using Maven.

ArtifactTransferException

Solution: remove `*.lastUpdated` files under `/.m2` Maven directory

Go to `/.m2` directory and run:

```
find .  -name "*.lastUpdated" -type f -delete
```

2.7.2 CRFPP

These are common problems when compiling CRFPP.

Unable to locate jni.h

Solution: type `locate jni` and put the jni path in the `CRF++-0.58/java/Makefile`

Compiling CRFPP: cannot find crfpp.h

Solution: open `CRF++-0.58/java/CRFPP_wrap.xx` and fix the include to `../crfpp.h`

Compiling CRFPP: ld does not find lcrfpp

Solution: copy `CRF++-0.58/.libs/libcrfpp.so` to `/usr/lib`

CHAPTER 3

USING IYAS

3.1 Test examples

The IYAS codebase contains test cases for most of the components. They are useful for testing the behaviour of code and serve as documentation.

They can be found under the `scr/test/` directory.

3.2 Tutorial code

The code used in two tutorials given on the IYAS framework is contained in the `qa.qcri.qf.tutorial` package under `scr/test/`.

The `AnalyzerAndTreeFrameworkTutorial` shows how to run analysis pipelines, serialize results, retrieve and use annotations, produce tree structures from text and annotation and manipulate trees.

The `LowerCaseExample` and `LowerCaseAnnotator` show how to build a simple UIMA annotator using uimaFIT.

3.3 TREC Reranking Pipeline

The TREC Reranking Pipeline is a pipeline for learning and evaluating reranking models exploiting shallow structural representations of text and feature vectors.

The pipeline analyzes questions and candidate passages, builds training and test data for the reranking model and contains tools for validating the models.

Descriptions of the structural representation and of the experimental setting can be found in **Building structures from classifiers for passage reranking**, *Aliaksei Severyn, Massimo Nicosia, Alessandro Moschitti,* CIKM 2013.

Reranking of passages containing answers to a question is a fundamental part of Question Answering system.

3.3.1 Task

The reranking tasks consists in, given a question, retrieving from a corpus indexed by a search engine candidate passages which may contain or not the answer to that question, and eventually producing a permutation of the retrieved passages, hopefully promoting at the top of the list, the passages which actually contain the answer.

3.3.2 Data

The data used is collected from the TREC Question Answering tracks. We used questions from 2002 and 2003 years (TREC 11-12), which totals to 824 questions. The AQUAINT newswire corpus is used for searching the supporting answers. The lists of candidate passages related to the questions are obtained using the TERRIER search engine.

The experimental data can be found under the `data/trec-en/` folder.

`questions.txt` contains the 824 questions.

`terrier.BM25b0.75_0` contains the candidate passages retrieved by the TERRIER search engine for each of the TREC questions.

3.3.3 Running the pipeline

The pipeline can be run from the Eclipse IDE setting the program and Java Virtual Machine arguments from the Run Configuration panels, or using the `run.sh` script in the main folder of the IYAS distribution. However, for the first time, it is recommended to run it from Eclipse in order to download all the required JARs with Maven.

Launch the `TrecPipelineRunner.java` program with these arguments:

```
argumentsFilePath arguments/trec-en-pipeline-arguments.txt
```

This file defines all the required arguments for the program, such as the options and the files with the data.

The Java Virtual Machine requires these arguments:

```
-Djava.util.logging.config.file="resources/logging.properties"
-Xss128m
```

The logging option specifies the record file for the Mallet library. The -Xss128m option increases the JVM memory in order to use the JNI SVMLightTK library. Every program making use of the JNI classifier **must** use additional memory, otherwise it will fail silently.

The output of the pipeline consists in the training and test data for learning and evaluating the reranking model. You will find the generated data in the output train and test folder.

3.3.4 Learning and evaluating the model

You can learn the models using the SVMLightTK programs for learning and classification.

IYAS provides Python scripts for carrying out cross-validation, learning and evaluating the models.

For generating 5 folds from the output data use the following script on the output folder:

```
python scripts/folds.py data/trec-en/ 5
```

For launching the learning, classification and evaluation script on the generated folds use:
`python scripts/svm_run_cv.py --ncpus 2 data/trec-en/folds/`

The evaluation report will be displayed on screen.

CHAPTER 4

UIMA AS

UIMA Asynchronous Scaleout is an extension of UIMA which enables support for scaling analysis pipelines. The idea behind UIMA AS is exposing analysis engines as services which can be accessed through network interfaces. In this way, an analysis engine can be wrapped and executed as a server waiting for input. The service takes a serialized CAS and adds its annotations to it. Such services can be deployed on several machines so the workload can be distributed across a network. Services use a message broker (ApacheMQ) to communicate. Moreover, on a single machine, multiple instances of the same annotator can be created to be executed on multiple cores. Check `http://uima.apache.org/doc-uimaas-what.html` for additional information about UIMA AS.

4.1 Setting up UIMA AS

Download UIMA AS from `http://uima.apache.org/downloads.cgi` and decompress the archive on your machine. The UIMA AS folder will become your new `UIMA_HOME`. Open a terminal and change your current directory to the UIMA AS folder. Then, set `UIMA_HOME` to the current directory:

```
> export UIMA_HOME=`pwd`
```

4.1.1 Running the Message Broker

In order to be able to execute a distributed analysis process you must launch a message broker on a machine in the network. The message broker of choice for UIMA AS is Apache ActiveMQ, which is included in the UIMA AS distribution. To set the broker up type you have to define ACTIVEMQ_HOME:

```
> export ACTIVEMQ_HOME=${UIMA_HOME}/apache-activemq/
```

Now you can run the broker with:

```
>./bin/startBroker.sh
```

4.1.2 Using annotators in UIMA AS

UIMA AS requires XML descriptors of analysis engines. In the `qa.qcri.qf.uimaas` package there is the `PipelineDescriptorWriter` class. You can modify this class to output XML descriptors for the annotators which compose your complex pipeline. For maximum flexibility you will need an XML descriptor for each of the primitive annotators in your pipeline. Running the class as-is will generate several XML descriptors in the textttdesc/uima-as folder.

Now we are ready to aggregate the primitive annotators and craft the XML file used by UIMA AS to instantiate and expose them as services.

4.1.3 Aggregating the annotators

The following XML descriptor (`desc/uima-as/pipeline-aggregate.xml`) composes primitive annotators in a single aggregate pipeline. In such way annotators running together can be specified. It is important to note the `key` attribute of the delegate analysis engines which identifies the analysis engine in the aggregate pipeline. Moreover, this descriptor references the XML descriptors associated with the primitive components.

```
<?xml version="1.0" encoding="UTF-8" ?>
<analysisEngineDescription xmlns="http://...">
  <frameworkImplementation>org.apache.uima.java</frameworkImplementation>
  <primitive>false</primitive>

  <delegateAnalysisEngineSpecifiers>
    <delegateAnalysisEngine key="segmenter">
      <import location="segmenter.xml" />
    </delegateAnalysisEngine>

    <delegateAnalysisEngine key="postagger">
      <import location="postagger.xml" />
    </delegateAnalysisEngine>
  </delegateAnalysisEngineSpecifiers>

  <analysisEngineMetaData>
    <name>Pipeline Aggregate</name>
```

```
    <description>Pipeline Aggregate</description>

    <flowConstraints>
      <fixedFlow>
        <node>segmenter</node>
        <node>postagger</node>
      </fixedFlow>
    </flowConstraints>
    <operationalProperties>
      <modifiesCas>true</modifiesCas>
      <multipleDeploymentAllowed>true</multipleDeploymentAllowed>
      <outputsNewCASes>false</outputsNewCASes>
    </operationalProperties>
  </analysisEngineMetaData>
</analysisEngineDescription>
```

4.1.4 The Deployment XML descriptor

The following deployment XML descriptor (`desc/uima-as/deploy-pipeline4uima-as.xml`) is used by UIMA AS to instantiate the analysis pipeline on the current machine.

The `inputQueue` element specifies an endpoint which identifies the communication channel used later by an application to send to and receive data from an analysis pipeline. The `brokerURL` attribute specifies where is the location of the ApacheMQ broker.

Then, the aggregated pipeline is imported and some annotator specific properties are specified using the `key` attributes.

```
<?xml version="1.0" encoding="UTF-8"?>

<analysisEngineDeploymentDescription
  xmlns="http://...">

  <name>Iyas Sample UIMA-AS pipeline</name>
  <description>Analyze content running several NLP annotators</description>

  <deployment protocol="jms" provider="activemq">
      <casPool numberOfCASes="5"/>
    <service>
      <inputQueue endpoint="iyasSamplePipeline" brokerURL="${defaultBrokerURL}"/>
      <topDescriptor>
       <import location="pipeline-aggregate.xml"/>
      </topDescriptor>
      <analysisEngine>
        <delegates>
          <analysisEngine key="segmenter">
            <scaleout numberOfInstances="2"/>
          </analysisEngine>
          <analysisEngine key="postagger">
```

```
          <scaleout numberOfInstances="2"/>
        </analysisEngine>
      </delegates>
    </analysisEngine>
  </service>
 </deployment>

</analysisEngineDeploymentDescription>
```

4.1.5 Running the services and a sample client

During deployment, UIMA AS need to know the location of the JARs containing the classes and the models used by the analysis annotators. Iyas dependencies are managed by Maven. To put Iyas dependencies in a folder you can go in the Iyas main directory and type:

```
> mvn clean dependency:copy-dependencies package
```

This will copy all the dependencies into the `target/dependency` folder. To tell UIMA AS about this folder set the `UIMA_CLASSPATH` variable:

```
export UIMA_CLASSPATH=Iyas/target/dependency
```

To run the pipeline service use the script in the UIMA AS distribution:

```
> ./bin/deployAsyncService.sh Iyas/desc/uima-as/deploy-pipeline4uima-as.xml
```

4.1.6 Running a client application

The `qa.qcri.qf.uimaas.UimaASPipelineClient` is a Java client application for reading textual content and send it to the UIMA AS pipeline for analysis. The application requires data about the broker URI (ServerURI) and the analysis engine endpoint in order to connect to it and establish a communication channel. The text processing is done asynchronously via a callback class. Data is sent to the endpoint and when the analysis completes the callback class perform operations on the analyzed content.

The following dependencies were added to the POM for developing applications using the UIMA AS primitives: `uimaj-as-core`, `uimaj-as-jms`, `uimaj-as-activemq`.

4.1.7 Distributed pipeline

The prototype pipeline presented in this chapter describes all the building blocks needed for designing and deploying more complex distributed pipelines. For more information and options please refer to the official UIMA AS documentation:
`https://uima.apache.org/documentation.html`.