AMICUS Documentation

Gregory Finley

March 8, 2017

Chapter 1

Running AMICUS

AMICUS (A Metasystem for Interoperation and Combination of UIMA Systems) is a toolkit that facilitates a number of operations on the outputs of natural language processing (NLP) systems that store their annotations in the Unstructured Information Management Architecture (UIMA).

As a "metasystem," AMICUS maintains its own pipeline of components that are capable of translating UIMA annotations from one type to another, extracting and exporting information from annotations, and even merging several annotations from different type systems. A full pipeline can have any number of uses, from preparing outputs of one system for consumption by another system, to generating a new virtual system output by drawing from several systems, to simultaneously evaluating multiple systems against a manually annotated standard. The AMICUS distribution includes a "cookbook" of pipeline ideas for a variety of applications.

The guiding philosophy behind AMICUS is to achieve interoperability and synthesis between different NLP systems not by enforcing a common type system between them, but rather by providing an easily configurable process that will translate between type systems.

Note that AMICUS was originally designed for use with biomedical NLP systems, although it is not limited to text in that domain. As such, some included components act on annotation types commonly used in biomedical and clinical NLP.

The rest of this chapter describes the process of setting up and running AMICUS. The user will certainly want to modify or create new pipelines to perform their own tasks, so the next chapter covers in detail how to use the various pipeline components. More hands-on users may prefer to learn by modifying the example recipes given in the cookbook, which are discussed in 3.

1.1 Installation

Most users will need to build AMICUS from the Java source code because the build is dependent on the UIMA type systems used.[1] You will need JDK 7 or higher and Maven 3 installed.

(If you just want to run recipes from the cookbook and/or only use the same NLP source systems used in the cookbook, you do not need to rebuild from source—skip Steps 2 and 3 below—and you will only need a Java 7 Runtime Environment or JDK.)

1. Clone using Git (alternatively, download the source from GitHub):
   `git clone https://github.com/gpfinley/amicus.git`

2. Copy all XML type system descriptions you will want to use into `src/main/resources/typesystems/`. Typically, every NLP system will provide its own single type system description. Note that this directory already includes type system descriptions for several popular biomedical NLP systems, which are required for recipes in the cookbook. These files can be removed if you do not need them, with the exception of `AmicusTypeSystem.xml`.

3. Run `mvn clean install` from the `amicus` directory. This will build a JAR file in the `target` directory which will work with any inputs covered by the type system descriptions used in Step 2.

1.2 Usage

The AMICUS pipeline reads CAS objects from XMI or XML files as produced by other UIMA systems, joins them into one CAS per document, performs some processing, and saves summary CASes. For each source NLP system being used with AMICUS, all documents should be saved into a separate directory. All systems' directories should contain files with identical base names—e.g., `ctakes/doc1.xmi`, `ctakes/doc2.xmi`, `metamap/doc1.xmi`, `metamap/doc2.xmi`, etc. (It is fine if the file extensions are different—one system may serialize CASes with XMI and another with XML.)

For each document, AMICUS will store annotations from each system in a different View of the AMICUS CAS. Pipeline components specify which View to take annotations from and save new annotations to. Whenever new annotations are created, it is always recommended that these be saved into new Views to keep the source systems' views intact and to make debugging easier.

The pipeline's inputs, outputs, and components are defined in a single YAML configuration file. (YAML is a human- and machine-readable format that can be

[1] More technically, AMICUS uses Java classes and methods for setting specific annotations on JCas objects, which are built using UIMA JCasGen.

modified in any text editor.) After importing the CASes from the specified systems, pipeline components are executed in the order given in the configuration file.

Several example pipelines are included in the cookbook. The quickest way to start would be to choose one of these and modify it according to your needs.

1.2.1 Running a pipeline

The simplest way to run a pipeline is from the command line, with a configuration file as the argument. From the AMICUS base directory, type:

```
java -jar target/amicus.jar <your-config-file>
```

TODO: include instructions for running JAR from Windows, or include a .bat file

1.2.2 Editing configurations

The configuration can be edited in detail; see the next chapter for a discussion of how the various engines and swappable modules work.

The pipeline is defined by headings under `pipelineComponents`. See the cookbook for examples of defining various engines. Aside from the pipeline itself, only a few parameters need to be set:

- `allSystemsUsed`: names for all systems (optional), the input directories for all systems, the Views that those systems saved annotations into (often '`_InitialView`'), and the Views to copy annotations into when loading the inputs into the AMICUS CAS.
- `xmiOutPath`: the directory to write the AMICUS CASes for all documents.

Some configuration parameters need Java package notation: classes corresponding to AMICUS modules (described in detail in the next chapter); and UIMA types, which are manipulated through compiled Java classes. If you are unsure about a Type's full path notation, search for it in the XML type system description file.

1.3 UIMA basics

Familiarity with some elementary UIMA concepts is helpful when configuring the pipeline, so this section provides the barest minimum of a crash course in UIMA.

UIMA stores all text and annotations in an object called the CAS. A CAS can be stored ("serialized") to XMI or XML format, either of which can be read by AMICUS. All types of annotations made on a CAS must also be defined in a type system description, which is stored in XML format (and typically named `TypeSystem.xml` or something similar).

CASes can have multiple Views, which function like additional virtual CASes and are key to configuring the AMICUS pipeline. When reading CASes from XMI/XML, AMICUS will copy the content from one system's imported CAS into a View as defined in the configuration. (It is suggested that this View be named something derived from the NLP system used to create the CAS—'CtakesView', for example.)

Pipeline modules' inputs and outputs are configured by specifying the Views they act upon rather than the names of the source systems. All Views creating during system import and during processing are stored in a single CAS (per source document), which is written to disk at the end of the pipeline.

Chapter 2

Pipeline Components

This chapter explains the various types of modules available for the AMICUS pipeline as well as some of the interchangeable and configurable "analysis pieces" that these modules use. This chapter is intended as a reference; the reader may wish to skip directly to the descriptions of cookbook recipes in the next chapter for a more hands-on introduction.

There are only a handful of different UIMA engines used by the AMICUS pipeline, although they can be chained together any number of times in any order. Each module can also be configured further and rely on smaller components, here called analysis pieces, which exist in different flavors and all of which can be extended using the Java API if you need more specific functionality. This chapter summarizes the modules as well as the swappable pieces that are used by one or more engines. (For guidelines to extending pieces using the API, see the developer guidelines in 4.)

AMICUS modules are designed to be simple and general; individually, they may not be capable of more complex operations. Complex functionality can usually be achieved through certain combinations of engines, which we refer to as an "idioms," and which may not be immediately intuitive. Many recipes in the cookbook demonstrate the use of idioms, and we refer the reader to such recipes where appropriate in this chapter.

2.1 Merger

The Merger is the workhorse of AMICUS: it draws annotations from multiple systems, finds their overlap across systems, and distills them into a summary annotation that can then be added to the CAS. Each Merger takes one or more annotation types as input and writes one or more as output.

2.1.1 Inputs

Each input to the Merger should specify the Annotation Type, the field(s) of that type to draw from, and the View holding the annotations. To draw from more than one field, include all fields separated by semicolons, e.g.: `name;age;occupation`. Multiple fields will be concatenated into a pipe-delimited string (e.g.: "James|42|physician") when used for components that process strings. When written to outputs, the same number of fields will need to be specified, although null fields are possible: for example, to write just the `age` field on the output, specify `;age;` in the configuration, which will be split along the semicolons, causing the empty first and third fields to be ignored.

Pullers

Each input has its own Puller, which is an object that has a specific behavior for getting useful information out of an Annotation. The default puller, `edu.umn.amicus.pullers.Puller`, simply calls the input Type's getter(s) for the specified field(s).

Note that custom Puller implementations are free to ignore the Type and field(s) of the input, as they have direct access to the UIMA Annotations themselves.

Two alternative Pullers are included:

- `edu.umn.amicus.pullers.CtakesCuiPuller` will get concept unique identifiers (CUIs) from cTAKES `IdentifiedAnnotation` types, as cTAKES stores CUIs in FSArrays rather than directly in a text field.
- `edu.umn.amicus.pullers.PassthroughPuller` will yank the entire Annotation object itself and could be useful if later processing needs access to the whole Annotation or if you want to copy it wholesale.

2.1.2 The Aligner

Each Merger runs an Aligner, which restructures Annotations so they can be more easily compared or combined across systems. For each input, The Merger will find all Annotations of the specified Type in the specified View. All Annotations are then passed to the Aligner, which re-organizes them into sets that align in the text, one from each input. Generally, each Annotation will be represented exactly once when re-organized into these sets. There are three basic Aligner implementations included:

- `edu.umn.amicus.aligners.PerfectOverlapAligner` will only align Annotations that perfectly line up. Any non-aligning Annotations will still be present in the output, but they will be paired with `null` values for the other inputs.
- `edu.umn.amicus.aligners.PartialOverlapAligner` will create alignments for any Annotations that all overlap at at least one character. Alignments

will be optimized to line up whole words as much as possible. Note that this optimization process can be slow if there are many Annotations that form long "chains" of overlaps across systems (this will not the case for things like concept identifications but could be for things like syntactic constituents from a parser).

- `edu.umn.amicus.aligners.EachSoloAligner` is a non-aligner: it will output each Annotation exactly once by itself. This can be useful for generating summary statistics over all Annotations, or any other application where comparing them directly is not necessary.

2.1.3 Outputs

The Merger will also output one or more new Types onto a specified View. As with the inputs, the Type and its relevant field(s) need to be specified, as well as the View to write to. It is suggested that all outputs are written to an entirely new View ('MergedView', e.g.), which AMICUS will automatically create, rather than one used by one of the inputs. Each output also has its own Distiller, which is described below.

Distillers

Distillers are modules that act on a list of aligned values (one from each input) and produce a summary value that will then be written by the Pusher. For each Annotation tuple that the Aligner iterates over, the Merger will call the relevant Puller for each input and pass the results as a list to the Distiller.

Several Distiller options are included. The simplest (and default) is `edu.umn.amicus.distillers.PriorityDistiller`, which simply takes the first present (i.e., non-null) object from the list. This could be useful if, for example, you wanted to augment coverage of one system with that of another, but did not want to replace any annotations from the first system in cases of overlap.

TODO: ... talk about other distillers

Pushers

Analogous to the Puller used for each input, the output needs a Pusher to write to a specific Annotation. The default Pusher is `edu.umn.amicus.pushers.Pusher`. An output type and field(s) need to be specified. See the discussion of Pullers for getting multiple fields from an Annotation. If only a single value is to be written to multiple fields, AMICUS will try to expand it to the requisite number of fields by splitting on the pipe character '|'.

2.2 Translator

The Translator functions similarly to a Merger but only takes a single input and allows for further operations: filtering and mapping. Multiple outputs are still

possible and might be useful if reading multiple fields from the input.

2.2.1 Filter

A Translator can be used with an optional Filter, which will selectively write new annotations based on their contents. The basic Filter is `edu.umn.amicus.filters.RegexFilter`, which will only allow annotations that match the regular expression specified in the Translator configuration. If no pattern or alternative Filter implementation is specified, Annotations will not be filtered.

Another included Filter option is `edu.umn.amicus.filters.NumberFilter`, which will use the `filterPattern` configuration parameter as a numeric threshold; specifying relationships other than $\geq$ can be done by prepending the number with `<`, `=`, `<=`, or `>`. For example, using `2` for the pattern will allow to pass only those annotations for which the value of the specified field is at least 2, while `<=10` will allow to pass values less than or equal to 10. (In a specific idiom, this Filter can be used for enforcing a minimum number of annotations; see recipe ???.)

The basic filtering approach allows annotations to be filtered based on the content of one of their own fields. To filter annotations based on another annotation, it is necessary to use a Merger to line them up first. The cookbook recipe "???" illustrates an idiom that can be used here.

To filter an annotation based on the contents of a different field of the same annotation, another specific idiom may be helpful: specify all fields for the Puller to pull and design the regular expression to operate on the pipe-delimited string. See the recipe "???" for an illustration.

2.2.2 Mapper

A Translator can optionally include one or more Mappers to map annotation values from one domain to another. This may be useful if one system uses different values for functionally equivalent annotations (e.g., 'False' vs. 'No'), to replace annotation values with potentially more useful values (e.g., converting UMLS CUIs to their preferred string forms), or to prepare values for easier comparison (e.g., convert to lower case).

Whereas other modules are specified by providing the class name, most Mappers require further configuration, which is provided in a separate file. Only simple Mappers, like `edu.umn.amicus.mappers.ToLowerCaseMapper`, can be specified by using the class name like other pieces. For others, provide the path to a Mapper configuration file. Different classes of Mapper are configured differently; see the included examples in the `mapperConfigurations` directory. For a generic hash-based Mapper (implemented in the `edu.umn.amicus.mappers.Mapper` class), the configuration file can simply be YAML-formatted map.

Other Mappers in the distribution:

- `edu.umn.amicus.mappers.RegexMapper` is configured with a find pattern and a replace pattern and will perform regular expression substitutions.

- `edu.umn.amicus.mappers.EquivalentAnswerMapper` will convert any strings that are part of a cluster to the first string listed in that cluster. This can be useful for evaluations where multiple possible strings could be considered correct (e.g., "CT scan" and "computed tomography scan"). This mapping is case insensitive.

2.3 Exporter

The Exporter is used to export annotation values into other formats than UIMA, such as CSV or plain text. Like the Merger, the Exporter configuration can take multiple inputs; however, it does not output any new Annotations, but rather saves files to a directory. (One file per document will be generated; if you are looking to generate a single summary of all documents, see 2.4.)

2.3.1 Aligner implementation

The Exporter uses a different default Aligner from the Merger: `edu.umn.amicus.aligners.EachSoloAligner`, which does not align so much as it iterates over every Annotation of every input type in the CAS. This implementation is the default because it is assumed that the Exporter will typically be used to extract every instance of an Annotation independently. It is always possible to specify other Aligners if the goal is to output aligned Annotations.

2.3.2 The ExportWriter

A writer can be specified for specific output formats. TODO: describe export writers

- AlignedCsvExportWriter
- AlignedTsvExportWriter
- EachSoloCsvExportWriter
- EachSoloTsvExportWriter

2.4 Summarizer

Like the Exporter, the Summarizer is used to extract information from the CAS and report it in another format. The Summarizer works differently from all other engines, however, in that it operates over all documents in the collection rather than each one in parallel.

Typical uses would be to generate descriptive statistics or to summarize results of an evaluation over all documents. As a UIMA engine, the Summarizer collects contents of Annotations during the `process()` phrase and summarizes

them in the `collectionProcessComplete()` method, which is called after all threads have finished.

The Summarizer takes a single input configuration and a SummaryWriter implementation.

2.4.1 The SummaryWriter

The SummaryWriter is a component that acts on a list of objects and produces a text summary of them. This summary can be written to a file or to standard out (the default if no output file is specified in the configuration). The default implementation is `edu.umn.amicus.summary.CounterSummaryWriter`, which outputs the token counts of every type of value for the given input.

2.5 Evaluation modules

AMICUS can be used for evaluation of individual systems and of their processed translations or combinations. Evaluation is not its own engine, however, but rather an idiom that chains two engines: summary objects measuring matches between systems are created using a Merger, and summary statistics are reported using a Summarizer. These engines can be configured like any other components, although for evaluation they will use a specifically designed Aligner, Distiller, Pusher, Puller, and SummaryWriter implementations. In the source code, these implementations are organized under the `eval` package rather than their respective `aligner`, `distiller`, etc. packages because they work slightly differently from non-evaluation modules.

AMICUS is currently capable of evaluating annotations for precision, recall, and F-score. Other types of evaluation would need to be implemented (see 4), although they may be able to use some of the components already in place.

2.5.1 The Evaluation Aligner

Evaluation can either use `PerfectOverlapAligner` or, if a less strict evaluation that allows partial overlaps is desired, the evaluation-specific `edu.umn.amicus.eval.EvalPartialOverlapAligner`. The regular partial overlap aligner should not be used for evaluation because it does not prioritize the ground truth annotations in the first input slot.

This Aligner assumes that the first input contains ground truth Annotations, and all other inputs are hypotheses. It will represent every Annotation from every input exactly once. For every ground truth Annotation, all Annotations from other systems that line up are also presented in a single iteration; these are the true positives and false negatives (which will be represented as null in the list for the systems where they are not present). The Aligner will subsequently present all the false positives from each system, with every iteration containing a null Annotation in the first slot to represent the absence of these annotations from the ground truth.

2.5.2 The Evaluation Distiller and Pusher

Based on the structure of the Annotation tuple passed from the Aligner, the Distiller creates a list of objects that encode the status of Annotations in this tuple (true positive, false positive, or false negative). The Pusher stores this object in a special AMICUS Type (`edu.umn.amicus.type.EvalAnnotation`).

2.5.3 The Evaluation Puller and Summarizer

The evaluation-specific Puller and SummaryWriter (`edu.umn.amicus.eval.EvalPrfSummaryWriter`) are to be used in a Summarizer module and are designed to read the custom evaluation objects and report basic statistics such as accuracy, precision, recall, and F-measure across documents in a collection.

2.6 "Hidden" engines

Some engines are invariant components of the AMICUS pipeline and are hidden from the typical user. Advanced users may wish to instantiate AMICUS UIMA components independently through uimaFIT. This section is briefly describes the behavior of the hidden engines.

2.6.1 CommonFilenameCR

This is a UIMA collection reader that creates the initial CAS for each document. It reads common filenames from the directories of all source systems specified and creates `edu.umn.amicus.DocumentID` Annotations from the original filename. No other data is loaded into the CAS at this point.

2.6.2 CasAdderAE

This engine is a CAS multiplier that adds a new view to the shell CAS created by `CommonFilenameCR` and reads a serialized CAS from one of the source systems, saving it into a new view. The `AmicusPipeline` class's main method will add one `CasAdderAE` to the pipeline for each source system.

2.6.3 XmiWriterAE

Following the execution of all configured pipeline components, the full CAS and all its imported and created Views will be exported using the standard UIMA XMI CAS serializer. The directory to use is specified in the configuration file. An XML type system file will also be written to the same directory as `TypeSystem.xml`.

Chapter 3

Cookbook

This chapter contains descriptions of included configuration recipes.
TODO

3.1 Simple cases

...

3.2 Idioms

...

Chapter 4

Using the API

AMICUS provides a Java API for defining your own custom analysis pieces (Filters, Mappers, Aligners, Distillers, Pushers, Pullers, SummaryWriters, and ExportWriters). You may want to do this to deal with idiosyncratic Type implementations or to introduce specialized logic into your pipeline that is not possible with the basic versions of these engines. Base classes or interfaces exist for each of these, and the following sections describe considerations to be made when extending them.

4.1 Filter

4.2 Mapper

4.3 Aligner

4.4 Distiller

4.5 Puller

4.6 Pusher

4.7 ExportWriter

4.8 SummaryWriter

4.9 Instantiating modules with uimaFIT

All modules used in the AMICUS pipeline can also be instantiated through UIMA or uimaFIT.

- all pipeline components can be instantiated normally through UIMA and uimaFIT, but you need to pay special attention to the config parameters, which are organized differently than they are for the AMICUS pipeline