AMICUS Documentation

Gregory Finley

March 31, 2017

Acknowledgments

AMICUS was developed during my time as a postdoctoral researcher in the NLP/IE group of the Institute for Health Informatics at the University of Minnesota. All work was supported by the National Institute of General Medical Sciences (GM102282).

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.

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 only 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 not need Maven.

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 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.

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

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(s)>
```

If not building from source, use the examples JAR:

```
java -jar amicus-examples.jar <your-config-file(s)>
```

The JAR is also double-clickable. When launching AMICUS this way, you will be given a file browser window to choose configuration files on your system. Note, however, that any relative file paths given in the configurations will be interpreted relative to the JAR's location, whereas with the command line option, paths will be interpreted relative to your current directory. (To avoid this confusion, it is recommended to use absolute paths in your own configuration files.)

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, as well as templates showing which parameters can be configured for each type of module.

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 refer to this chapter while working through examples in the cookbook.)

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 3.)

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. Some 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; the first and third field names, after splitting on semicolons, will be empty and thus 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). If no values are present in any of the specified fields, `Puller` will not return any value; it is as if there were no annotation present. If you want to bypass this behavior, use `AllowNullPuller` (see below).

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. Specified fields are ignored.
- `edu.umn.amicus.pullers.PassthroughPuller` will yank the entire annotation object itself (ignoring any specified fields) and could be useful if later processing needs access to the whole annotation or if you want to copy it wholesale.
- `edu.umn.amicus.pullers.AllowNullPuller` behaves like the default `Puller`, but it will not quietly discard annotations that have null values for the specified fields.

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.
- `edu.umn.amicus.aligners.RequirePerfectOverlapAligner` and `RequirePartialOverlapAligner` will simultaneously perform alignments and filter: they do not return any alignments in which any of the inputs are missing. Any annotations that do not have corresponding annotations in all other inputs will be ignored.

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.

Other included distillers:

- `edu.umn.amicus.distillers.CountingDistiller` will count the number of aligned annotations (up to the number of inputs) and save this count in a new annotation rather than the contents of any input annotations.

- `edu.umn.amicus.distillers.ListDistiller` will create a list of all annotations from all inputs.
- `edu.umn.amicus.distillers.StringConcatDistiller` will concatenate the values of all annotations in string form with a pipe separator ('|'). This can be used for multiple outputs (or multiple fields on one output).
- `edu.umn.amicus.distillers.VotingDistiller` will vote across inputs and return the plurality value. In the event of a tie, priority is given to inputs listed first.

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 '|'.

Also include is a `PassthroughPusher`, which will copy the annotation as-is to the output View (should be used in conjunction with `PassthroughPuller`).

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.

The basic filtering approach allows annotations to be filtered based on the content of one of their own fields. To filter annotations based on the presence of another annotation, a useful idiom may be to do so not through a Filter but rather through `RequirePerfectOverlapAligner` or `RequirePartialOverlapAligner` in a Merger module.

To filter based on the *content* of another annotation, the specific idiom would be: filter the other annotation based on its content and create a new dummy annotation, then perform the merging step as described above. See the cookbook filtering recipe #3 for an example of this idiom.

To filter an annotation based on the contents of a different field of the same annotation, you can avoid merging and use another idiom: specify all fields for the Puller to pull (including the field whose content you want to filter on) using a semicolon-delimited list of field names (see Inputs subsection above), design a regular expression to operate on the pipe-delimited string, and write to unnamed fields for all fields other than your desired output, again using field names separated by semicolons. See filtering recipe #2 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 create any new annotations on the CAS, but rather writes them to files. Two export strategies are possible: writing one

file per document, or one file for the entire collection. These types of output are enabled by, respectively, the DocumentSummarizer and CollectionSummarizer.

Like the Merger, the Exporter takes an Aligner implementation, enabling it to export multiple inputs at once. Note that it does not use Distillers; if you want the use of a Distiller, it is best to run a Merger beforehand and then export its outputs.

2.3.1 The Summarizer

All Summarizers included with the AMICUS distribution can function as both DocumentSummarizers and CollectionSummarizers; new implementations contributed via the API do not have to function as both. The included implementations are listed below.

- `edu.umn.amicus.summary.AlignedCsvSummarizer` will save annotations from all inputs into a CSV file, with each alignment between systems written to its own line. Alternatively, the user can select tab-delimited or "pipe"-delimited files by modifying the associated configuration file in `classConfigurations`.
- `edu.umn.amicus.summary.EachSoloCsvSummarizer` saves one annotation per line. The Aligner used here does not matter, except in that some Aligners do not represent every annotation exactly once; it is suggested to use `edu.umn.amicus.aligners.EachSoloAligner` here for full representation and fastest performance.
- `edu.umn.amicus.summary.CounterSummarizer` will count the types of data from each input source and output a summary of these counts to a text file.
- `edu.umn.amicus.summary.EvalPrfSummarizer` will print precision, recall, F-scores, and accuracy statistics for any number of systems. The *first* input provided to the Exporter will be assumed to be the reference standard, with all other inputs being hypothesis systems. Only two Aligners are recommended when performing an evaluation: the default `PerfectOverlapAligner`, for a strict evaluation, or `edu.umn.amicus.aligners.EvalPartialOverlapAligner`, for less strict alignments. `PartialOverlapAligner` is *not* recommended, as it does not prioritize the first system and may lead to extra false negatives.

Note that DocumentSummarizers create outputs in parallel during the processing phase, whereas CollectionSummarizers collect annotations during this phase but do not write outputs until after all documents have finished processing.

2.4 "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 briefly describes the behavior of the hidden engines.

2.4.1 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.

2.4.2 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.

2.4.3 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 description will also be written to the same directory, with the name `TypeSystem.xml`.

Chapter 3

Using the API

AMICUS provides a Java API for defining your own custom analysis pieces (Filters, Mappers, Aligners, Distillers, Pushers, Pullers, CollectionSummarizers, and DocumentSummarizers). 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. Note that an empty constructor for any of these pieces can be used if no configurability is required.

All custom classes should be added to the classpath when running AMICUS.

3.1 Filter

Interface: edu.umn.amicus.filters.Filter
Constructor arguments: A String from the `filterPattern` configuration field.
Key method: passes
Returns: `true` if the object passes the filter

3.2 Mapper

Class: edu.umn.amicus.mappers.Mapper
Constructor arguments: A Map between Objects
Key method: map
Returns: any Object (probably a String)

3.3 Aligner

Interface: edu.umn.amicus.aligner.Aligner
Constructor arguments: None

Key method: alignAndIterate
Returns: an iterator over all aligned tuples of AMICUS-native annotations

3.4 Distiller

Interface: edu.umn.amicus.distillers.Distiller
Constructor arguments: None
Key method: distill
Returns: A single AMICUS-native annotation (ANA)—an Object value, plus integer begin/end values

3.5 Puller

Class: edu.umn.amicus.pullers.Puller
Constructor arguments: A string containing the name(s) of the field(s) to pull from (multiple fields should be separated by semicolons)
Key method: pull
Returns: Any kind of object (probably `java.util.List` if multiple fields)

3.6 Pusher

Class: edu.umn.amicus.pullers.Puller
Constructor arguments: The name of the output type and the field(s) to write to
Key method: push
Returns: Nothing, but should create a new UIMA Annotation on the JCas.

3.7 DocumentSummarizer

Interface: edu.umn.amicus.summary.DocumentSummarizer
Constructor arguments: The names of all input views, type names, and input fields (all three are String[])
Key method: summarizeDocument
Returns: A string containing the summary to write to file.

3.8 CollectionSummarizer

Interface: edu.umn.amicus.summary.DocumentSummarizer
Constructor arguments: The names of all input views, type names, and input fields (all three are String[])
Key method: summarizeCollection
Returns: A string containing the summary to write to file.

3.9 Instantiating modules with uimaFIT

All modules used in the AMICUS pipeline can also be instantiated through UIMA or uimaFIT. Note that these modules take many configuration parameters (see the module code), some of which are arrays that should have the same length. When executing a pipeline, AMICUS converts its own configuration format into the format required by these engines.