DEFT/A Guide to Incorporating a New or Updated Algorithm in A2KD

From BU E Wiki

Contents

General Overview

A2KD system currently has support for the following algorithms (for the languages that are currently supported):

Algorithm Type	English	Chinese
Entity Coreference	UIUC Coref	Stanford Coref
Entity Linking	RPI EDL	RPI EDL
Nil Clustering	UIUC	None
Relation Extraction	Stanford	None
Event Extraction	BBN Serif	None

This guide primarily describes the mechanism to integrate a new (or currently unsupported) algorithm type in A2KD pipeline. To update the algorithm for a supported algorithm type (e.g. update the version of UIUC Coref algorithm for English) or to change the algorithm for a supported type (e.g. using Stanford Coref for English instead of UIUC Coref), you can jump to this section.

For any algorithm that A2KD system runs, it does the following:

- Run the algorithm on input documents and get the resulting HltContentContainers
- Include the HltContentContainers from the algorithm in chunk-alignment. Chunk-alignment is the process of aligning any mentions, relation/event arguments etc (textual chunks) generated by the algorithm to the mentions and chunks extracted by the coreference algorithm. This is done so that the entities extracted from various different algorithms could be mapped to one another.

- Include the artifacts produced by the algorithm (entities, relations, events, etc) in a "master" HltContentContainer. The master HltContentContainer is a single HltContentContainer that contains only those artifacts from an algorithm's output that are relevant to that algorithm's type. For example, a master HltContentContainer will contain only relations from the relation-extraction algorithm even if it produced entities or events in its output.
- Merge algorithm-produced artifacts from multiple HltContentContainers (i.e. multiple master HltContentContainers)
    - A2KD does a batch-level (where a batch is a Spark partition containing one or more documents) merging or aggregation of artifacts and holds the merged artifacts in objects that subsequently take part in the upload process.
    - This merging is done in order to ensure that there are no duplicates of a single real-world artifact. For example, if two entities extracted from two different documents are linked to the same external KBID, they refer to a single real-world artifact, and are therefore merged together.
    - The definition of merged artifact depends on the type of the artifact itself. For example, in case of entities, two or more entities with the same external KBID are merged by taking a union of their provenances (entity-mentions), averaging their confidences, and taking the most-frequent canonical-mention as the canonical-mention of the merged entity. For more details on how the merging is done for various kinds of artifacts, refer to this wiki.
- Upload the artifacts merged at batch level to the KB
- De-duplicate artifacts in the KB
    - This is required so that a global de-duplication of the batch-level unique artifacts can happen. The logic for de-duplication is similar to batch-level deduplication or merging.

A developer integrating a new algorithm in A2KD will have to implement the above stages for the new algorithm in the A2KD pipeline by following the steps below.

Implementing Spark wrapper for the algorithm

The main algorithm processing class (or module class) is usually a class that extends `adept.module.AbstractModule` class, and expects an HltContentContainer as the object to process. However, since A2KD runs in Spark environment, we have to ensure that the module class can run over an RDD of HltContentContainers instead of a single HltContentContainer. To this end, the user needs to implement a Spark wrapper for the algorithm. A ready-to-use generic Spark wrapper class is `adept.e2e.algorithms.GenericAlgorithmSparkWrapper`. In case the algorithm does not need any additional pre-processing or post-processing besides whatever is implemented by the module class, the user can use `GenericAlgorithmSparkWrapper` as the algorithm's Spark wrapper. If, instead, the algorithm does require any additional pre-processing or post-processing, the user will need to implement their own Spark wrapper class by extending `adept.e2e.algorithms.SparkAlgorithmComponent`. In either case, the Spark wrapper class will have to be configured in a2kd config file. We will deal with the configuration part at the end.

Implementing Algorithm Spark Wrapper by Extending SparkAlgorithmComponent

Although it's not usual, but sometimes it may be required to do additional processing or activate additional modules than what the algorithm module class does by itself. A good example of this is the UIUC Coref algorithm. The UIUC Coref algorithm needs pre-processing steps of sentence-segmentation, POS-tagging and chunking before processing the input HltContentContainer for coreference extraction. This requires IllinoisSentenceSegmenter, IllinoisPosTagger and IllinoisChunker to be activated along with IllinoisCoreferenceResolver module, which is the main algorithm module. Take a look at the overridden `activateModule` method in the class `adept.e2e.algorithms.IllinoisCorefSpak`.

```
@Override
public void activateModule() throws Exception {
    super.activateModule();
    synchronized (adjunctAlgorithmsActivationStatus) {
      if (!adjunctAlgorithmsActivationStatus.isAlgorithmActivated()) {
        adjunctAlgorithmsActivationStatus.setAlgorithmAsActivated();
        iss = new IllinoisSentenceSegmenter();
        posTagger = new IllinoisPosTagger();
        chunker = new IllinoisChunker();
        ner = new IllinoisEntityRecognizer();
        config = new UiucModuleConfig(this.algorithmSpecifications.configFilePath());
        tokenizerType =
            TokenizerType.valueOf(config.getString(IllinoisConstants.TOKENIZER_TYPE));
        chunker.activate(this.algorithmSpecifications.configFilePath());
        posTagger.activate(this.algorithmSpecifications.configFilePath());
        iss.activate(this.algorithmSpecifications.configFilePath());
        ner.activate(new UiucModuleConfig("edu/uiuc/ner/conllConfig.xml"));
      }
    }
  }
```

In the first line, we call `super.activateModule()` to activate the main IllinoisCoreferenceResolver module (the module class is configured through a2kd config). Subesquently, we activate other modules (posTagger, chunker, etc) in a synchronized block. The reason for that is that in A2KD we need to ensure that module activation happens only once per module. To achieve that, the `SparkAlgorithmComponent` class makes use of an `AlgorithmActivationStatus` object. One could call the `isAlgorithmActivated()` method on this object to check if the algorithm module is already activated, or the `setAlgorithmAsActivated()` method to mark the algorithm as activated in the first place. Note that the synchronized block in the above example uses local static `AlgorithmActivationStatus` object `adjunctAlgorithmStatus` to mark and check one-time activation of the adjunct modules (posTagger, chunker, etc).

The `preProcessHltCC` method is overridden as follows to ensure running of all adjunct modules on an input HltContentContainer before it is processed by the main algorithm module.

```
@Override
  public HltContentContainer preProcessHltCC(HltContentContainer hltContentContainer)
      throws Exception {
    Document document = hltContentContainer.getDocument();
    hltContentContainer = iss.tokenizeDocument(document);

    if (hltContentContainer == null || !checkRequiredViews(document, hltContentContainer)) {
      hltContentContainer = new HltContentContainer();

      TokenStream tokenStream =
          new TokenStream(tokenizerType, null, "English", null, ContentType.TEXT, document);
//                              tokenStream.setDocument( document );
      document.addTokenStream(tokenStream);
      List<Sentence> sentences =
          iss.getSentences(document.getValue(), document.getTokenStream(tokenizerType));
      hltContentContainer.setSentences(sentences);
      posTagger.process(document, hltContentContainer);
      chunker.process(document, hltContentContainer);
    }
    return hltContentContainer;
  }
```

Finally, a post-processing step is implemented by overriding the `postProcessHltCC` method. The post-processing step for UIUC Coref algorithm does the following things:

- Set the entity mentions extracted by UIUC Coref algorithm to the entityMentions list of the output HltContentContainer (by calling the setEntityMentions API)
    - IllinoisCoreferenceResolver does not populate entityMentions list. A2KD needs this list to access the entity-mentions extracted by an algorithm. Since other algorithms (e.g. Stanford and BBN Serif) always have this list set, setting this list in the post-processing step ensures that A2KD can treat UIUC Coref's output the same way as that of other algorithms.

```
A2KD also requires the Coreference list to be populated for any coreference or entity-linking algorithm. See the postProcessHl
```

- Use entity-types from IllinoisNER instead of those from IllinoisCoreferenceResolver
  - During the course of integration, BBN found that the entity-types produced by IllinoisNER were better than the ones produced by the coreference algorithm module itself. Therefore, in the post-processing step, we overwrite the entity-types in the output of IllinoisCoreferenceResolver with entity-types produced by IllinoisNER.

Adding New Algorithm Type

The new algorithm type should be added as a String constant in `adept.e2e.driver.E2eConstants` class. Conventionally, the name of this constant should start with the prefix `ALGORITHM_TYPE_`. The actual value of the string constant should be lower cased where each token is separated by underscores, and should make clear the purpose of the algorithm. A few examples of currently supported algorithm types are as follows:

```
    public static final String ALGORITHM_TYPE_ENTITY_LINKING = "entity_linking_algorithm";
    public static final String ALGORITHM_TYPE_RELATION_EXTRACTION =
        "relation_extraction_algorithm";
    public static final String ALGORITHM_TYPE_EVENT_EXTRACTION = "event_extraction_algorithm";
```

Following table lists the constant identifiers for various types of algorithms currently supported by a2kd:

Algorithm Type	Identifier
Entity Coreference	coref_algorithm
Entity Linking	entity_linking_algorithm
Nil Clustering	nil_clustering_algorithm
Relation Extraction	relation_extraction_algorithm
Event Extraction	event_extraction_algorithm

Including algorithm in Chunk-Alignment

You will need to include your algorithm in chunk-alignment phase. This can be done by modifying the chunk-aligner class, `adept.e2e.driver.ChunkAlignerSpark`. Take a look at the existing code in the `call()` method. For any algorithm type, the following two things need to be done:

- Create and configure a `adept.e2e.chunkalignment.DefaultChunkAligner.DocumentAlignmentOptions` object
  - Take a look at the `adept.e2e.chunkalignment.DefaultChunkAligner.DocumentAlignmentOptions` class for all the alignment options available.
  - Following is an example:

```
  DefaultChunkAligner.DocumentAlignmentOptions entityLinkingAlignmentOptions =
              chunkAligner.new DocumentAlignmentOptions();
          entityLinkingAlignmentOptions = entityLinkingAlignmentOptions.alignAllEntityMentions().
              allowableChunkClasses(allowableChunkClasses).allowCrossType();
```

- Put the `DocumentAlignmentOptions` object created above in the `alignmentOptionsMap`. For example:

```
        alignmentOptionsMap
            .put(E2eConstants.ALGORITHM_TYPE_ENTITY_LINKING, entityLinkingAlignmentOptions);
```

Including the Algorithm in Master HltContentContainer

Next we have to ensure that the artifacts from the algorithm make it to the master HltContentContainer (or simply, master container). All code related to the creation of master container lives in `adept.e2e.mastercontainer` package. To include the artifacts from various algorithms in the master container, the first thing that needs to happen is creation of an `AlgorithmChunkMap` object. This object has an aggregate of all the information from various algorithm containers required to facilitate creation of the master container. The `AlgorithmChunkMap` class makes use of HltContentContainer output of various algorithms and the chunk-alignment object (created above) to create the following mappings:

- A mapping of entity-mentions from non-coref algorithms to the entities from coref algorithm
- A mapping of non-coref entity-mentions to best non-coref entity for every non-coref algorithm involved
    - It is possible for a single entity-mention to map to more than one entities with different confidences. This mapping makes it possible to determine which non-coref entity is the best representative for a non-coref entity-mention.
    - This mapping is used in conjunction with the non-coref entity-mention to coref entities mapping (listed above) to map non-coref entities to coref entities (see `MasterCorefEntities.java`).
    - The non-coref to coref entities map is subsequently used to replace non-coref entities with mapped coref entities wherever the non-coref entities are part of an artifact of interest. For example, for a relation extracted by the relation_extraction_algorithm (a non-coref algorithm), the arguments that are of type `adept.common.Entity` (non-coref entities) are replaced by mapped coref `Entity` objects (see `MasterDocumentRelation.java`).
- A mapping of entities from coref algorithm to external KBIDs from entity-linking and nil-clustering algorithm
    - This mapping ensures that all coref entities are either linked to a known real world entity (through a non-nil ID pointing to an external KB--the external KBID), or to a cluster of similar entities that seemingly belong to a real world entity not existing in any KB (nil KBID produced by nil-clustering algorithm)
    - The nil or non-nil KBID for an entity is set to it before inserting the entity in the master container

Note that you may or may not need to make any modifications in `AlgorithmChunkMap` class. In the `createAlgorithmChunkMap` method of that class, you can see how special handling is done for chunks from coref_algorithm, entity_linking_algorithm and nil_clustering_algorithm. However, for all other algorithms (i.e. relation_extraction and event_extraction), the processing is generic which creates the entity-maps described above.

The `MasterAlgorithmContainer` uses the `AlgorithmChunkMap` object to extract specific artifacts (entities, relations, events, etc) from algorithm containers and creates specific "master" objects (e.g. master coref entities, master document relations).

- Take a look at `MasterCorefEntities` and `MasterDocRelations` classes for example
- These classes take the relevant mappings from `AlgorithmChunkMap` object as input and create adept artifacts (e.g. coreferences, entities, document relations) that can then be set to the master container
- For a new algorithm type, you will need to create a similar class that can create "master" adept artifacts from the output of your algorithm
    - You will also have to enhance `MasterAlgorithmContainer` to include the output from the above class in the master container

Implementing Batch-level Artifact-Merger for the Algorithm

As described in the General Overview section, A2KD does a batch-level merging of artifacts. To implement artifact-merger for the artifact produced by your algorithm, you'll have to determine what it means to merge the artifacts in the first place, and what data structure you'd want to store the merged artifacts in. For examples, you can see `ItemWithProvenances.java` or `MergedDocumentRelation.java` (also refer to DEFT/A Note On Merging Adept Artifacts in E2E). Currently, A2KD has merging implementation for entities, relations, events and non-entity arguments of relations and events. All the code relevant to artifact-merging lives in `adept.e2e.artifactextraction`.

Once you have created a placeholder for merged-artifacts from your algorithm, you will have to enhance the following classes:

- `ExtractedArtifacts`, which encapsulates all merged-artifacts
- `MergeUtils`, which implements the logic to merge various artifacts
- `ArtifactExtractor`, which is the top-level class called by the driver to iterate over master containers and call relevant methods in `MergeUtils` in order to create the resulting `ExtractedArtifacts` objects

Uploading merged-artifacts

Upload of merged artifacts is handled by artifact-specific uploader classes (see `EntityUploader.java`, `RelationUploader.java` for example). You will have to implement a similar Uploader class for the artifact that your algorithm produces. These uploader classes then make calls to specific upload methods implemented in `UploaderUtils` class. You can take a look at the existing methods in this class (e.g. `uploadEntities`,`uploadRelations`) and note the usage of relevant KBAPI methods to do the actual artifact-upload.

De-duplicating artifacts from the KB

Just like upload, deduplication of artifacts is handled by artifact-specific deduplicator classes (see `EntityDeduplicator.java`,`RelationDeduplicator.java` for example). You will have to implement a similar deduplicator class for the artifacts that you want to deduplicate in the KB. The actual deduplication logic will depend on the particular artifact being deduplicated, and will be similar to the logic you employed when doing batch-level merging or de-duplication.

Bringing it all together--modifying the drivers

After you have implemented the above stages, you will have to add relevant method calls in the A2KD pipeline. This can be done by modifying the `adept.e2e.driver.MainE2eDriver` class, which is the top-level controller of the pipeline, and the `adept.e2e.E2eDriver` class, which runs the pipeline for every language involved.

In the `E2eDriver`, the steps for running new algorithm, chunk-alignment, creation of master-container, batch-level merging of artifacts will happen by themselves. All that you will have to do is add a method that calls your artifact-specific uploader (see the implementation of methods `uploadEntities` or `uploadBatchLevelRelations` for example).

In the `MainE2eDriver`, you will have to include calls to your artifact-specific upload method that you added in `E2eDriver` above. You will also have to add a method that calls your artifact-specific deduplicator (see the implementation of methods `deduplicateEntities` or `deduplicateRelations` for example).

Configuring Your Algorithm in A2KD Config

Finally, in order for A2KD to be able to run your algorithm, you will need to configure it in the a2kd config. This can be done by configuring the `algorithm` element of the config XML.

```
<algorithm type="" name="" providerName="">
            <sparkWrapperClass> </sparkWrapperClass>
            <algorithmModule> </algorithmModule>
            <configFile> </configFile>
            <ontology> </ontology>
            <moduleUsesDeprecatedProcessCall> </moduleUsesDeprecatedProcessCall>
</algorithm>
```

Following is the description of the attributes and child elements of `algorithm`:

- type: The type of your algorithm
  - This should be exactly the same as the value of the String constant that you added for your algorithm above.
  - For currently supported algorithms, the type should one of the types listed in algorithm types table.
- name: Name of the TA1 algorithm
  - This could be anything that you want to identify your algorithm with
- providerName: Name of TA1 algorithm provider
  - Any string that identifies the provider of the algorithm
- algorithmModule: Fully qualified name of the class implementing the algorithm
  - This is the main algorithm class that extends `adept.module.AbstractModule`
- sparkWrapperClass: Fully qualified name of the Spark wrapper class that internally runs TA1 algorithm
  - This could either be `adept.e2e.algorithms.GenericAlgorithmSparkWrapper`, or the name of the custom Spark wrapper class that you have implemented (as described in this section).
- configFile: Path to config file for TA1 algorithm (relative to classpath)
  - All TA1 algorithms are required to use a config file. This property should point to the config file for your algorithm.
- ontology: the ontology that TA1 algorithm's output is compliant with
  - All types produced by TA1 algorithms are required to conform to an ontology. The recommended ontology is Adept ontology, which is defined by the specifications in `adept-kb/src/main/resources/ontology/adept-core.ttl` and `adept-kb/src/main/resources/ontology/adept-base.ttl`.
  - However, if TA1 algorithms deviate from this convention and use a different ontology, they are required to provide ontology mapping files which are XML files (compliant to java properties DTD) that map types from their ontology to adept ontology and vice-versa.
  - The allowed values for `ontology` property are `tac2012`, `rere`, `adept`, and `custom`.
    - `tac2012` and `rere` are A2KD provided ontology mappings from TAC 2012 and Rich ERE ontologies to Adept ontology respectively.
    - `adept` refers to the Adept ontology, and should be used if your algorithm is compliant to the Adept ontology.
    - `custom` should be used if your algorithm is compliant to none of the above ontologies, in which case you will need to configure the following two properties
- ontologyMappingFile: For `custom` ontology, you must use this config to specify path of a properties file (relative to classpath) that maps your algorithm's ontology to Adept ontology (see adept-kb/kbapi/stanford-to-adept.xml for example).
- reverseOntologyMappingFile: For `custom` ontology, you must use this config to specify path of a properties file (relative to classpath) that maps Adept ontology to your algorithm's ontology (see adept-kb/kbapi/adept-to-stanford.xml for example).
- moduleUsesDeprecatedProcessCall: If the main processing method of the algorithm extends the deprecated method `process(Document,HltContentContainer)` instead of the recommended method `process(HltContentContainer)`, set this config to true. Default value for this config is false. This config helps A2KD determine the right method to call on the algorithm class in order to run it.

This will conclude your algorithm's integration into A2KD. You will also have to update the unit-tests as required.

A Note On Updating Or Changing The Algorithm

In case you want to update your algorithm with a newer version, all that is needed is that you update the a2kd's pom.xml to reflect the version of updated algorithm. For example, if there's a newer version available of UIUC Coref algorithm, say version `2.8`, you will have to update the dependency on the respective module to version 2.8 as follows:

```
<dependency>
      <groupId>edu.uiuc</groupId>
      <artifactId>illinois-coreference-adept</artifactId>
      <version>2.8</version>
</dependency>
```

This, however, assumes that the main algorithm class (e.g. `edu.uiuc.corereference.IllinoisCoreferenceResolver`) and other configurations for the algorithm (like sparkWrapperClass, configFile, etc) are the same as before. If any of that has changed, you will have to update the a2kd config file to reflect those changes for your algorithm (see the above section for details of config properties for an algorithm).

If you want to change what artifacts are used from a particular algorithm, all that you need to do is change the `type` attribute of the `algorithm` config for that algorithm (assuming that other properties remain the same). For example, if you want to use coreferences from Stanford algorithm instead of UIUC, you can change the config for Stanford algorithm as follows:

```
<algorithm type="coref_algorithm" name="StanfordRE" providerName="Stanford">
            <sparkWrapperClass>adept.e2e.algorithms.StanfordSpark</sparkWrapperClass>
            <algorithmModule>edu.stanford.nlp.StanfordCoreNlpProcessor</algorithmModule>
            <configFile>edu/stanford/nlp/StanfordCoreNlpProcessorConfig.xml</configFile>
            <ontology>tac2012</ontology>
            <moduleUsesDeprecatedProcessCall>true</moduleUsesDeprecatedProcessCall>
</algorithm>
```

If you want to continue using Stanford as the relation-extraction algorithm, you can leave its configuration with type `relation_extraction_algorithm` as is:

```
<algorithm type="relation_extraction_algorithm" name="StanfordRE" providerName="Stanford">
            <sparkWrapperClass>adept.e2e.algorithms.StanfordSpark</sparkWrapperClass>
            <algorithmModule>edu.stanford.nlp.StanfordCoreNlpProcessor</algorithmModule>
            <configFile>edu/stanford/nlp/StanfordCoreNlpProcessorConfig.xml</configFile>
            <ontology>tac2012</ontology>
            <moduleUsesDeprecatedProcessCall>true</moduleUsesDeprecatedProcessCall>
</algorithm>
```

Note that you can use a single algorithm with multiple algorithm-types, but not the other way around. A2KD currently only supports one algorithm of a given type.