PI3: UIMA ANALYSIS ENGINES

Name: Zhuyun Dai
AndrewId: zhuyund

---

1. OVERALL FEATURES

The task of the PI3 is to implement a logical architecture of a Question-Answering System.

In my implementation, the Collection Processing Engine (CPE) contains 3 parts: a Collection Reader that reads the documents from the input directory; an Aggregated Analysis Engine that annotates the document and scores each answer; and a CAS Consumer that evaluates the system and writes results to the output directory.

The Collection Reader reads document into the CAS. It also stores configuration metadata (N-gram configuration, output directory, input file name) into the CAS.

The Aggregated Analysis Engine is made up of 4 primitive AEs. All AEs extend JCasAnnotator_ImplBase. The logic of the AE is mainly implemented in the function `process()`.

- `TestElementAnnotator`. Annotates question and answer spans in the document.

- `TokenAnnotator`. Tokenizes question and answers with `stanford.nlp.process.Tokenizer`.
- `NgramAnnotator`. Annotates N-grams in the question and answers according the configuration parameter n. For the convenience of answer scoring, the text of the N-gram is also store in the annotation.
- `AnswerScoreAnnotator`. Scores each answer by the its N-gram overlapping with the question. For example, if the question is "Is it a blue ball" and the answer is "A blue ball", there will be 2 bigrams shared by the question and the answer, thus the answer scores 2 when using bigrams. Notice that the score should be in [0, 1], we normalize the score by the answer's number of tokens.

The CAS Consumer is the final part of the pipeline. It reads `answerScore` annotations from the CAS index, and evaluate the precesion@N. Evaluation results, as well as the ranked answer list, are written into the output directory specified in the command parameters.

Every component also write its id and confidence score to annotations that they creates. Here, the id is the class name of the component, e.g. ”NgramAnnotator”. The confidence score is 1, since every step is deterministic.

In addition to the pipeline components, 2 java classes are implemented to help the AEs. They are:

- `MemAnswerScore`. This class has the same fields as `AnswerScore (score, id, label)`. It implements the Comparable interface, so that `AnswerScore` objects can be easily sorted. This helps to get the ranked answer list.
- `FSListCreator`. This is a static class. It has one function: createFSList(). This function takes a java Collection object as the parameter, and returns a FSList. This helps to set features for `InputDocument` and `Ngram`, which have FSList features.

2. TYPE SYSTEM

Based on the default type system, I modified the Ngram type and added AnswerScore into the type system.

Type Name	Sub-Types	Description

Ngram	`int n`
	`FSList tokens`
	`String text`
		String text is the text of the N-gram. It is the text of all the tokens, joind by ' '.

AnswerScore	`float score`
	`Boolean label`
	`String id`
		This type records a answer candidate span in the input file. That is the span of:
		A<# question>
		`String text` records the text of the answer, e.g. "John loves Mary."
		`Boolean label` records whether the answer is correct (`True`) or (`False`).
		`Integer id` records the id of the question, e.g. id = 1. It is used to identify the answer so that we can match answers to their scores.

3. COLLECTION READER

The Collection Reader in the pipeline is implemented `in QaCollectionReader.java`. Its descriptor is in the CollectionReader.xml.

The `QaCollectionReader` extends `CollectionReader_ImplBase` class. It has two important functions:

Function Name	Parameters	Description

`void initialize`	`int n`
	`String inputDir`
	`String outputDir`
		This function accepts the main function parameters. It is called in the main function.
		This function also lists all files in the input directory.

void getNext	`CAS aCas`	This function read the next document from the collection into the CAS. It calls `jcas.setDocumentText` to set the document text. It also stores the configuration parameters (`n, inputDir, outputDir`) and document meta data (file name) into CAS. These will be used by downstream AEs and the CAS Consumer.

4. TEST ELEMENT ANNOTATOR

`TestElementAnnotator` is the first AE in the aggregated AE. It annotates the question and answer spans into `Question` and `Answer` annotations.

In the function `process()`,it create a `Question` for the first line of the document, and an `Answer` for other lines. It stores the sentence of the question and answers into the annotations. It also stores the label and id of answers into Answer annotations.

5. TOKEN ANNOTATOR

`TokenAnnotator` is the second component in the aggregated AE. It tokenizes question and answers with `stanford.nlp.process.Tokenizer`.

The work flow of the function `process()`is as follows:

1. Read `Question` and `Answer` annotations generated by the previous AE.
2. Call the `stanford.nlp.process.Tokenizer` to tokenize the sentences in the question and answers.
3. For each token, find its index in the sentence, and calculate its begin and end offsets in the document text. Write the token's `begin`, `end` and `text` into the `Token` annotation.

6. N-GRAM ANNOTATOR

`NgramAnnotator` is the third component in the aggregated AE. It reads all `Token` annotations from the CAS and find N-grams in the document.

The logic of the function `process()`is as follows:

1. Get the configuration parameter N from the CAS index. N is stored in the `Param` annotation.
2. Read `Token` annotations generated by the previous AE into memory. All tokens were stored into an array.
3. Since tokens are added to the CAS in the same order as they appear in the document, now tokens in the array are also in the same order as in the document text.
4. Every N consecutive tokens in the same sentence form into an N-gram. Store this N-gram into the CAS with a `Ngram` annotation, including its begin, end and text. The text is all token texts joined by " ", e.g. "a blue" and "blue ball".

7. ANSWER SCORING

In this step, the `AnswerScoreAnnotator` will assign a score to each answer candidates.

`AnswerScore` type records the score. To simplify the evaluation step, `AnswerScore` stores the answer's id and label as well.

The logic of the function `process()`is as follows:

1. Read `Ngram` annotations generated by the previous AE into memory. All N-grams were stored into an array.
2. N-grams are in the same order as in the document text. Therefore, we can quickly find out which question/answer an N-gram belongs to.

3. Use a `HashMap<String, Int>` to store the question's N-grams. The key is the text of the N-gram, the value is the N-gram's frequency in the question.
4. Go over all answer spans. For each answer span, go over it's N-grams. For each N-gram, look it up in the hashmap built in (3). If it exists in the hashmap, add 1 to the answer's score.
5. The score the normalized by the length of the answer.
6. Write `AnswerScore` annotations into CAS.

---

8. Evaluation: CAS CONSUMER

Finally, the CAS Consumer `QaCasConsumer` reads all `AnswerScore` annotations. In the function `process()`:

1. `AnswerScore` annotations are sorted by the score, in descending order.
2. Count the number of right answers.
3. Calculate the precision@N.
4. Read the `Param` from CAS to get the output file name and directory.
5. Write precision@N and ranked answer list to the output file.