Logical Architecture and UIMA Analysis Engines Design & Implementation

Name: Xiang Li Andrew ID: xiangl2 E-mail: lixiangconan@gmail.com

1. Task Requirement

In this task, we need to build analysis engines based on a given type system to implement a simple information processing task. As in HW1, the system reads input text from a file, and generates answers after processing it. The input file contains a question and several candidate answers to the question. The system needs to choose correct answers from the given candidate answers.

Specifically, the input question starts with an uppercase letter "Q" followed by a sentence, which is the content of the question. For example, the input sentence could be:

Q John loves Mary?

The input file also contains several candidate answers. Each answer has three parts: the first part is an uppercase letter "A" that indicates it is an answer; the second part is a number (1 or 0) showing whether the answer is correct; the third part is a sentence, which is the answer's content. These three parts are separated by spaces. For instance, one input answer could be:

A 0 John doesn't love Mary.

Since the second part is "0", we can know that the answer is actually not correct.

After processing the input file, the system will choose N answers from the candidate ones as the correct answers. Then a precision that measures the performance of the system will be calculated according to how many answers given by the system is actually correct.

2. Processing Pipeline

To achieve the final goal, the system will use a multi-step pipeline to process the input file. The pipeline is shown in Figure 1:

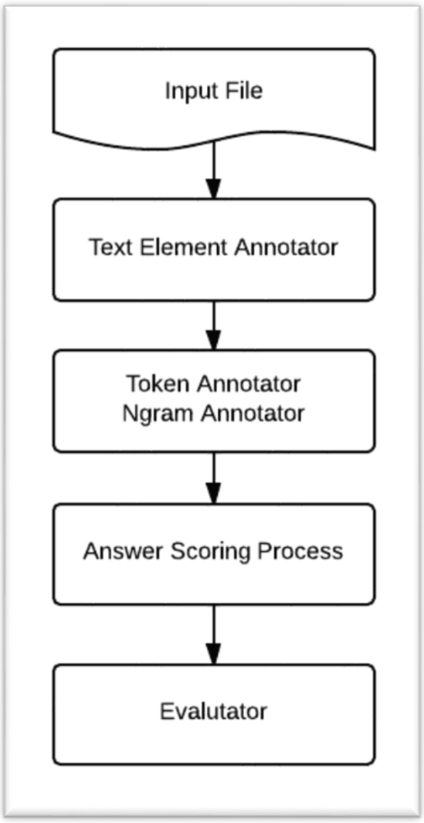

Figure 1 System Pipeline

According to the pipeline, the input file is firstly processed by the Text Element Annotator, which marks the question and answers in the file. Then, the input questions and answers are divided into tokens using the Token Annotator, and consequent tokens are connected into several 1-Gram, 2-Gram, 3-Gram sequences by the N-gram Annotator. After that, an Answer Scoring Process will compare the N-gram sequences in the question and answers, and assign a score to each answer. Finally, the Evaluator will rank answers according to their scores, choose the top N answers as correct answers, and then calculate precision at N to evaluate the performance.

3. Analysis Engines

To implement the pipeline, we need to design several analysis engines, and finally aggregate them together. Here are the details for the analysis engines.

3.1Text Element Annotator

When analysis the input file, the first analysis engine we need to use is the Text Element Annotator. Specifically, the test element annotator should divide the input file input several questions and answers, and record whether an answer is a correct answer. To do this, we notice that the input file is composed of several lines, while each line is a question or an answer. Therefore, the Text Element Annotator could read the input file line by line, and check the first character in each line. If the character is ‘Q’, the line should contain a question. Otherwise, if the character is ‘A’, the line should contain an answer. After determining whether a line is a question or an answer, the Text Element Annotator will generate a Question Annotation or an Answer Annotation for it to store necessary information.

3.2Token Annotator

After getting the result of the Text Element Annotator, the system will use the Token Annotator to divide the sentences into tokens. To tokenize the sentence, we uses the Stanford CoreNLP library. We also notice that not all characters in the input file should be in tokenization. So we need to ignore these parts, including ‘Q’, ‘A’ characters at the beginning of each line and answer scores before doing tokenization. Finally, the analysis engine will generate a Token Annotation for each validate token in the input file.

3.3N-Gram Annotator

With the output of the Token Annotator, the N-Gram Annotator could then connect tokens into N-Gram sequences. For this task, we need to generate 1-Gram, 2-Gram and 3-Gram Annotations. Since N-Gram Annotations are composed of continues Token Annotations, we can easily generate them using the index of Token Annotations. In addition, when connecting tokens into N-Gram sequences, we should only consider tokens in the same sentences. The generated N-Gram Annotations will then be used in the scoring process.

3.4Answer Scoring Process

After getting the N-Gram Annotations, we can then use them to check the similarity between question and answers. With this result, a score will be assigned to each answer to show how likely it could be a right answer. Specifically, we use a scoring method based on N-Gram hit rate. That is to say, the system would count the number of N-Gram Annotations in the answer sentences that could be matched with N-Gram Annotations in the question sentence, and then calculates the ratio of matched N-Gram Annotation number to total N-Gram Annotation number. The ratio, which is between zero and one, is then used as the score of the answer directly. Generally, a high score indicates that the answer has a high probability to be a correct answer. For example, if a question is “Q John loves Mary?”, and an answer is “A 0 John doesn’t love Mary.”, then there are totally 12 N-Gram Annotations in the answer, while 2 of them, which are “John” and “Mary”, can be matched with N-Gram Annotations in the question. So the score for this answer is 1/6.

3.5Evaluator

Finally, we can use the score for each answer to infer which answers are correct answers to the question. To do this, we firstly order the answers according to their score. After that, N answers with the highest scores are chosen as the output answers. In addition, we also counts how many chosen answers are actually correct, and then calculates a precision to measure the system performance.

4. Detail Analysis

In this task, the most important thing is to design a scoring method to find how likely an answer could be a correct answer to the question. Two basic ways are token overlap and N-Gram overlap, while the second way is the way we use. These two ways are quite simple, but their performance are not good enough. In general, N-Gram overlap could have a better performance than token overlap, because it uses the relations between tokens better. However, we can still use other method to improve the performance of the system. For example, we do not consider synonym in the match procedure. As a result, words with the same meaning, such as “love” and “like” could not be matched. Besides, we cannot match different tenses of the same word. To solve this problem, we need to do some preprocess before the match procedure, and use a dictionary to detect synonym. In addition, some unimportant part in the sentences, such as the parenthesis may also interfere the match problem. Since this problem is related to the sentence structure, it is hard to solve it using the N-Gram sequences and tokens. As a result, we may need to use a parse tree represent the sentence and distinguish principle parts from unimportant parts of the sentence. In addition to this, other methods such as named entity reorganization and sentimental analysis could also be used to improve the system performance.