11-791 Jung In Lee
Andrew ID: junginl
HW2

The Analysis Engine I have designed (QnAScoringAnnotator.java) consists of the following steps:
1. Import the input data.
2. Convert it into arrays of strings, split by lines.
3. Preprocess the data (for Stanford Core NLP) to be arrays of strings with only the text (without "Q", "A", 1s and 0s).
4. Process the arrays from 3 as the input data for Stanford Core NLP.
5. Extract tokens and POS tags.
6. Identify verbs ("VBD" or "VBZ"), subjects (the first "NNP"), and objects (the second or last "NNP") for each of the question and answer sentences.
7. Match the three arguments of each answer candidates with those of the question sentence, taking into account the synonyms, passive voice, and negation.
8. Keep track of the scores during the iteration.
9. After the iteration, compute scoresF (final scores).
10. Compute precision.

Comparison with Primitive Engine and Aggregate Engine Type:

The Analysis Engine I have designed has one annotator, which means that it is a primitive engine type, whereas an analysis engine consisting of several annotators would be an aggregate engine type. For a complex system, breaking down into small pieces usually leads to easier debugging and easier adjustments incase the codes need to be modified to accommodate different purposes. Yet, for a simpler task like this, a primitive engine also serves. Having all the relevant and necessary parts all in one place, a primitive engine makes it easier to understand the codes, without having to spend time figuring out how multiple annotators are related to one another. Both could have served the purpose of this assignment, but I thought this one is simple enough to be designed as a primitive analysis engine.

Accounting for synonyms:

To properly deal with the issues of synonyms, my original intention was to incorporate already existing tools like Wordnet, which has a large data of vocabulary. When inserting two words as inputs, it would determine whether they are synonyms or not, and if so, it would return a value, ranging from 0 to 1 (here 1 indicating the exact synonyms). For example, (kill, assassinate) are very close synonyms, so it would return a number very very close to 1, whereas (kill, save) have opposite meaning, so it would return a number very very close to 0 or just 0. This value, then, would be incorporated to the score of the answer candidate.

Passive Voice:

This part is to account for the equivalence in meaning using passive voice. In

other words, "Booth shot Lincoln" means the same as "Lincoln was shot by Booth." In English, all passive sentences are constructed with a finite form of the verb "to be" and a past participle. In Stanford POS tagging, all finite verbs are tagged as either "VBZ" or "VBD" (present, past tense respectively), and the past participles are denoted as "VBN." The procedures are as follows:
1. Construct a set consisting of the finite forms of the verb "to be," i.e. {"am", "is", "are", "was", "were"}.
2. If the finite verb of the answer candidate does not match the finite verb of the sentence, check whether the verb is an element of this set.
3. If so, check whether "VBN" equals the past participle form of the verb in the question. If yes, at least the answer candidate have gotten the verb part, so add one to the score.
   1) If they are not equal, check again without the last character of each words, so that "loves" gets matched to "loved."
4. Now, check whether the subject of the answer candidate equals the object of the question sentence. If so, add one to the score.
5. Similarly, check whether the object of the answer candidate equals the subject of the question sentence. If so, add one to the score.

Negation:

To negate a finite verb in English, we put "don't", "doesn't," or "didn't" in front of the infinite form of the verb. For example, in the sentence "John doesn't love Mary.", the Stanford NLP pos tagger would output "John/NNP does/VBZ n't/RB love/VB Mary/NNP ./.". The procedures are as the following:
1. Check whether VBZ or VBD of the sentence is in the set {"does", "do", "did"}.
2. If so, if the next token is equal to "n't".
3. If so, the answer candidate clearly did not get the verb part. But check if the subjects match and if the objects match. If so, add one to the scores respectively.
*4. We know that "John does love Mary" is equivalent to "John loves Mary." Thus, even if the VBZ/VBD of the sentence is in the set {"does", "do", "did"}, if the next pos is VB and the corresponding token equals the infinite form of the sentence verb, we give one point for the verb.
5. Again, check whether the subjects and objects match in this case as well, and increase the score accordingly.

Computing scores:

In the given input data, the verbs of the question sentences all have two arguments. This notion of verb and its arguments (head and its dependencies) is definitely generalizable to every possible sentence, but in order to incorporate this into the system, we need more advanced parsing tools and more deeply analyzed lexicon tools. For this assignment, I designed the system to accommodate verbs with two arguments, namely subjects and objects.

The basic idea underlying my scoring function is that the system gives one point for each arguments the system predicts correctly. In other words, it assigns one

point if the system gets the verb correct, one point if it gets the subject correct, and another one point if it gets the object correct. For each of these three arguments, the system keeps track of how many points each answer candidate obtained, and then at the end divide them by the total number of the arguments (3 in this case).

The output for the first data:

scores = [3, 1, 3, 1, 3, 1, 3, 1]

scoresF = [1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0]

The output for the second data:

scores = [3, 3, 0, 2, 3]

scoresF = [1.0, 1.0, 0.0, 0.0, 1.0]

As can be seen from the raw scores, the system output more varying scores than just 3 or 0, but when dividing them by 3, java somehow outputs all either 1.0 or 0.0. I tried both "double" and "float," but couldn't figure out how to correct this issue. Googling was of no avail. (Sorry, I don't' have much experience in both java and programming in general.) More interesting (or more precise, rather) scores can be computed, when using synonym dictionary, when having two words as inputs, the tool outputs a score between 0 and 1 in accordance with how close the two words are related.

Precision:

The precision was computed by out of the number of answer candidates the system purports the be correct, how many among them were actually correct.

with value 1 given in the input among those with the scores = 1.0

/ # with the scores = 1.0

The precision for both input data were 1.0.

Possible improvements:

Wordnet:

As mentioned above, it would have been much better if the system used WordNet or some other lexicon tools for the synonyms part. The general reasoning behind would be the system outputting a score between 0 and 1, when inputting two words in accordance with how they are related. I didn't know how to use the tool, so I just made a very simplified version for this task.

Passive voice: Morphology

The system would improve the performance a lot, if we could incorporate the concept of morphology. For checking passive voice, we first need to see whether the verb of the question sentence is equivalent to the "am/is/are/was/were + past participle" form. In English, it is often not that trivial to check whether the answer candidate has the right past participle form of the verb in the question sentence. In the first input text given, the finite verb in question was "shot" (past tense of "shoot"), which in this case is exactly the same as it s past participle "shot." For the

synonyms as well, "assassinated" (past tense of "assassinate") has the same form as its p.p. "assassinated." Yet, in the second data, I had to account for the different morphemes for present finite verbs and their past participles. This worked out relatively nicely, because "love" is a regular verb. However, to generalize the code to make it work with any data, we need to incorporate the concept of morphology so that is knows "blown" is the pp of the verb "blow" which has "blew" for the past tense.

Parsing trees:

The system I designed assumes the last NNP in the sentence to be the object. In the realm of this task, it works well. However, with more complex sentences, this would fail if the object is not a proper noun but just a general noun, or if there is an adjunct inserted between the verb and the object, which contains another NNP. In order to fix this problem, the system needs a more advanced NLP tool that can parse the sentence more analytically.

Co-reference Problem:

Again, for this task, the subjects and objects of the given data were all proper nouns (NNP), but we could easily come up with a more complex sentence with pronouns or other nouns that are co-referencing either the subject or object, and still means the same thing. In this case, we need to deal with the co-reference problem. We would need a tool that can analyze the sentence more deeply in terms of semantics.