11-693 Design and Engineering of Intelligent Information System

Building a Pipeline for Biomedical Question Answering

Team 02: Xi LIU, Lei XIAO, Xiaoxu LU, Yifu WANG, Yan HE

Github: https://github.com/11693-2/project-team02

Language Technologies Institute, Carnegie Mellon University

Abstract

This report describes the participation of team 02 from course 11-693 in the task on Building a Pipeline for Biomedical Question Answering, in which we finish the design and engineering of intelligent information system based on the baseline system provided by our TAs. After studying carefully about the goal of our task and researching the related advanced algorithms, we successfully build up the pipeline system and then make several performance improvements based on our work. In particular, we propose three features, including the relevance feature, the sentiment-word score feature and the position weight feature, for classifying the answers to general questions. Then we integrate these three features to obtain a novel feature function that classifies the answers into two categories: "Yes", "No".

Keywords: Question Answering, Name Entity Recognition, Information Retrieval, GoPubMed

I. Introduction

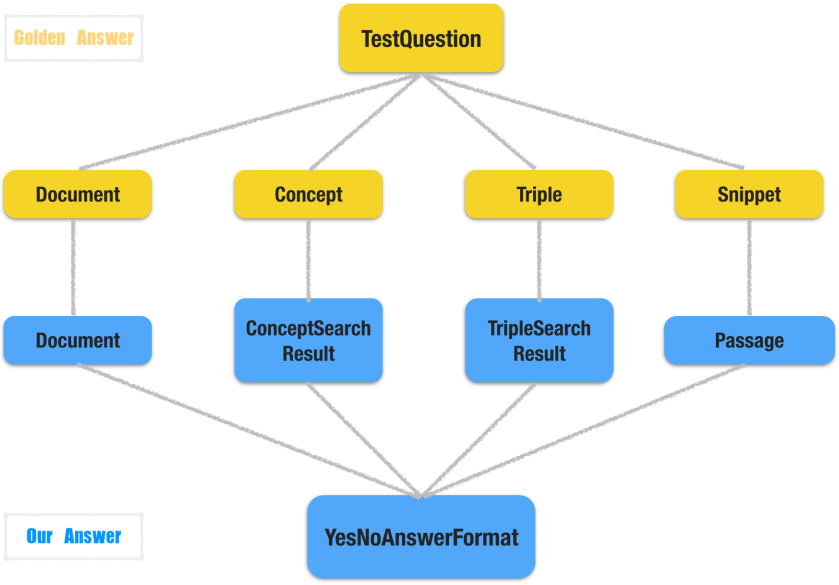
Question Answering (QA) is a computer science discipline within the fields of information retrieval and natural language processing (NLP) which is concerned with building systems that automatically answer questions posed by humans in a natural language.

This system mainly focus on answering Yes/No biomedical question (e.g. "Are there any DNMT3 proteins present in plants?"). As you can see, different from a general question answering, a biomedical QA system requires domain specific knowledge bases, NLP tools, literature collections, etc., in order to produce meaningful answers.

II. System Architecture

A. Type System

In our type system, we use Document, Concept, Triple and Snippet to cooperate with the TestQuestion in order to get the golden answer. On the other hand, we also use Document, ConceptSearchResult, TripleSearchResult, Passage to cooperate with the YesNoAnswerFormat to generate and output the answer by our own system. In particular, the main function of YesNoAnswerFormat is to store the output into JSON file. Then we compare these two kinds of answers and evaluate the result and performance of the system.



B. System Analysis

1) Overview

Unlike information retrieval systems and Web search engines, which typically return lists of relevant documents, QA systems for document collections aim to return exact answers (e.g., names, dates) or snippets (e.g., sentences) that contain the answers sought, typically by applying further processing to the user's question and the relevant documents that an information retrieval engine has returned.

2) Question Analysis

The user provides the question that need to be answered. At first the system will analysis the question itself and determine that which type it is. Then the system need to figure out

that what content is the question asking for. Moreover, the system need to generate a retrieval query with several retrieval component for the later processing.

In details, the questions need to be translated into collections of terms and phrases, and each term or phrase can be seen as a concept which should be stored in the cas. The main idea of generate the concept is to provide them for the document retrieval engine and retrieval the related documents.

3) Document Retrieval

The main function of document retrieval is to retrieve a list of documents that related to the question. As the question analysis part already generate a series of concepts for the user's question, the document retrieval engine will make use of these concepts and return a list of relevant documents for the question. For different kinds of document retrieval engine, we could get either an unranked set of documents that are likely to contain an answer, or a ranked list of documents by their possibilities of containing the target answer.

4) Document Processing

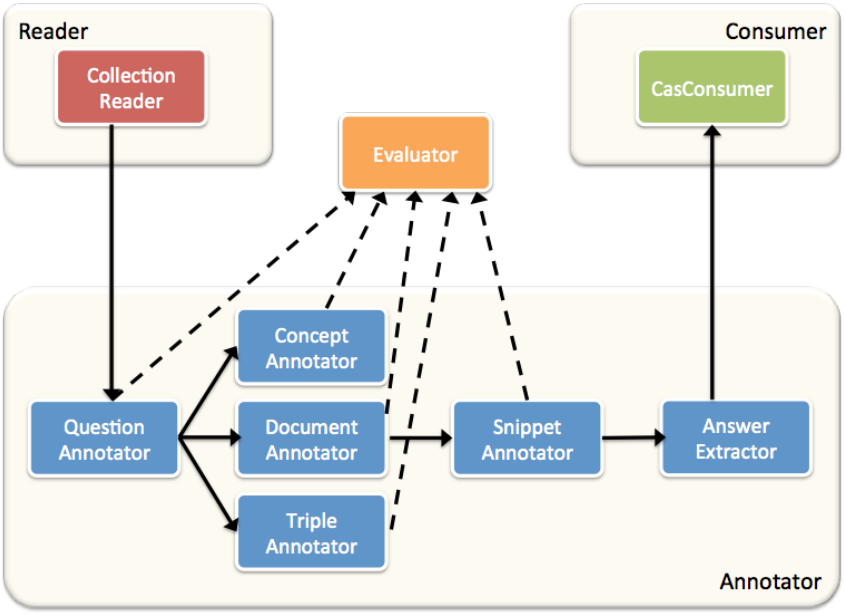
The main function of the document processing component is to search the list of documents that provided by the document retrieval component and the concept query that provided by question analysis component. Then The a list of candidate answers will be provided to the answer selection component.

5) Answer Analysis

The main function of answer analysis is to select the final answer from the list of candidate answers that provided by the document processing component. And then provide this answer to the user.

C. System Structure

Our system is based on Apache UIMA structure, employing GoPubMed as web service for query, employing LingPipe as Name Entity Recognizer to identify the terms. The Question Annotator generates better query for given question. The Document Annotator retrieve the related document, the Concept Annotator retrieve the related concept and the Triple Annotator retrieve the related triple. The Snippet Annotator retrieve the snippet based on what the document, concept and triple has generated. The AnswerExtractor extract the answer from the snippet for the query.



III. Implemented Strategies

A. Query Building

For this part, we have implemented several steps for query building, which are Stanford NLP Lemmatizer, OPEN NLP Tokenizer, Stop Word Removal, LingPipe Name Entity Recognizer and Query Operator.

```
// stemming
String stemmedQue = StanfordLemmatizer.stemText(queText);

// tokenize question with OpenNLP
ArrayList<String> wordList = new ArrayList<String>();
OpenNLPTokenization OpenNLPTokenizer = OpenNLPTokenization
        .getInstance();
wordList = OpenNLPTokenizer.tokenize(stemmedQue);

// stop word removal
StopWordRemover stopWordRemover = StopWordRemover.getInstance();
wordList = stopWordRemover.removeStopWords(wordList);

// remove repeated words
HashMap<String, Integer> tokenMap = new HashMap<String, Integer>();
for (String token : wordList) {
    if (tokenMap.containsKey(token)) {
        tokenMap.put(token, tokenMap.get(token) + 1);
    } else {
        tokenMap.put(token, 1);
    }
}
```

B. Named Entity Recognizer

For this part, we use a lingpipe NER (Named Entity Recognizer) to recognize the gene and use the confidence chunker to return a confidence in which we only choose the gene with confidence larger than 0.5. After this, since GoPubMed provide the web service to search result using tag like [mesh] and [go] (gene ontology), we also use this service to improve the correctness of the result.

```
// lingpipe as NER
ConfidenceChunker chunker = null;
String gene;
try {
    chunker = (ConfidenceChunker) AbstractExternalizable
            .readResourceObject(pathOfGene);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
char[] queryChar = t.toCharArray();
Iterator<Chunk> it = chunker.nBestChunks(queryChar, 0,
        queryChar.length, 1);
for (int n = 0; it.hasNext(); n++) {
    Chunk c = it.next();
    double conf = Math.pow(2.0, c.score());
    if (conf > 0.5) {// only leave those whose confidence make sense
        gene = (t.substring(c.start(), c.end()));
        int start = c.start();
        int end = c.end();
        t = t.substring(0, start) + gene + "[mesh]"
                + t.substring(gene.length() + start, t.length());

    }

}
//Operator Experiments
t = t.replaceAll("\\s", " AND ");

//[mesh] and [go] tagger
t = t.replace("(", "");
t = t.replace(")", "[go]");
```

C. Answer Extraction

1) Cosine Similarity Scoring

For this part, we use the cosine similarity to calculate a score for the extracted answer.

```
private double computeCosineSimilarity(Map<String, Double> queryVector, Map<String, Double> docVector) {
    double cosine_similarity = 0.0;

    double vec = 0.0;

    Map<String, Double> temp = new HashMap<String, Double>();

    temp.putAll(queryVector);
    temp.putAll(docVector);
    /**
     * combine the two map together to form a new map which contains the key
     * in query or doc
     **/
    for (Map.Entry<String, Double> entry : temp.entrySet()) {
        String a = entry.getKey();

        /** if the key is both in query and document, we will calculate **/
        if (queryVector.containsKey(a) && docVector.containsKey(a)) {
            vec += queryVector.get(a) * docVector.get(a);
        }

    }
    double doc1 = 0.0;
    double doc2 = 0.0;

    /** using the cosine similarity formula to calculate the score **/

    for (Map.Entry<String, Double> entry : queryVector.entrySet()) {
        doc1 += Math.pow(entry.getValue(), 2);
    }
    for (Map.Entry<String, Double> entry : docVector.entrySet()) {
        doc2 += Math.pow(entry.getValue(), 2);
    }

    double sq1 = Math.sqrt(doc1);
    double sq2 = Math.sqrt(doc2);

    cosine_similarity = (double) (vec) / (double) (sq1 * sq2);

    return cosine_similarity;
}
```

2) Position Scoring

For this part, we use the position score of the given sentence to calculate a score for the extracted answer.

```
int position = 3;
if (pos == 1 || pos == sentencelist.size())
    position = 3;
else if (pos == sentencelist.size() / 2)
    position = 1;
else
    position = 2;

passage.setPosition(position);
passage.setWordlist(word);
passage.addToIndexes();
rank++;
```

3) Sentiment Scoring

For this part, we use the sentiment in the sentence which the positive value represent a supportive sentiment and vice versa. In particular, we use the TF-IDF (Term Frequency –

Inverse Document Frequency) score and dictionary AFINN (an English sentiment words

```
for (i = 0; i < alist.size(); i++) {

    temp = alist.get(i);

    int co = calculate(newtext, temp);
    // System.out.println(temp + " " + count);
    int co2=calculate(t,temp);

    /************* tf*idf for each word****/
    double tf=(double)co/(double)tokenize0(newtext).size();
    double idf=Math.log((double)snippetList.size()/(double)co2+1);

    docVector.put(temp, (double)tf*idf);

    System.err.println("tfidf** tf:"+ (double)tf*idf + temp);

    /**********find sentiment score in the dictionary****************/

    //Pattern p = Pattern.compile(temp);
    //Matcher matcher = p.matcher(stopwords);
    if(doc.containsKey(temp))
        {
            senVector.put(temp, (double)doc.get(temp));
            System.err.println("^^^^^^^:"+ (double)doc.get(temp) + temp);

        }
    else senVector.put(temp, 0.0);

}
```

dictionary) to calculate a score for the extracted answer.

4) Voting Scoring

For this part, we multiply the three scores that calculated by the previous function and check whether it is positive or negative. If the result is positive, then the vote score will

```
        subscore=rel*sen*(double)position;

        System.err.println("subscore:"+ subscore + a.getText());

        /***final score***/
        //score+=subscore;
        if(subscore>0) positive++;
        else if(subscore<0) negtive++;

        /*********************************************************************/
        if(Math.abs(subscore)>=max_score) x=text;

        System.err.println("&&&&&:"+ subscore + a.getText());
    }
 //System.out.println("Yes");

 System.out.println(positive+"***********************************"+negtive);

    if(positive>=negtive){
        Answer answer = TypeFactory.createAnswer(aJCas,"Yes," + x,new ArrayList<>(),0);

        answer.addToIndexes();
         System.out.println("Yes."+x);
    }
    else {
        Answer answer = TypeFactory.createAnswer(aJCas,"No," + x,new ArrayList<>(),0);
        answer.addToIndexes();
         System.out.println("No."+x);
    }
```

be increased by 1. And if the result is negative, the vote score will be decreased by 1. At last, if the voting score is positive, then the answer will be "Yes" and if the voting score is negative, the answer will be "No".

IV. Experiments and Reviews

After designing type systems, pipeline, and strategies, we experimented with different combinations to maximize the final performance. The experiment includes the following stages, and at each stage we perform manual case-by-case error analysis to select the best combination.

A. Improve Retrieval Precision by Building Complex Queries

At the very beginning, we experimented query with the original questions and get low performance with a small number of document, concept, and triple retrievals. Then we applied the following approaches to solve the following problems by query pre-processing.

1) Low number of Retrieval

Stemming: we increased the number of retrievals by stemming.

Punctuation and Stop-word removal: by removing punctuations and frequent words, we increased the number of retrievals by stemming.

Repeat Word Removal: we query by terms instead of sentences to increase the query scope.

2) Large number of Irrelevant Results

Biological Keyword Recognition: overwhelmed by the irrelevant results, we added [mesh] and [go] after biological terms and significantly improved the relevance. We experimented with Stanford NER and several models with Lingpipe, we found n-best LingpipeNER.

Operators: we added AND, OR, and NOT to narrow the searching scope

3) Poor Term Selection

Multiple Tokenization approaches: we found out OpenNLP tokenizer achieved better tokenization than simply tokenizing by space, and get better precision.

B. Trade-off Retrieval Precision with Snippet Generation

At the first stage, we successfully retrieved smaller number of documents with higher precision. However, this lowered the number of snippets because the number of candidate was reduced. For some questions, no snippets were generated.

Through manual error analysis, we found some gold answer were picked from the documents with lower rank. Besides, in current pipeline better performance on concept and triple retrieval barely contributes to the snippet generation.

After several experiments, we found stemming, removing stop words and punctuations, and eliminating repeated words could generate snippets with acceptable precision.

After discussion, we think there is a trade-off between the precision of material retrieval and snippet generation. Stricter queries results fewer materials returned, furthermore makes the candidate pool for the generating snippets smaller. For better decision making for answer extraction, which is based on snippets, sufficient number of snippets is necessary. Thus we should guarantee a larger candidate pool by trading off the precision of retrieval.

C. Select Best Answer Selection Strategy

We experimented with sentence similarity, sentiment scoring to evaluate the candidates., and found combining sentence similarity and sentiment scoring achieved better accuracy. Then we introduced a voting strategy to select yes/no answer. After manual error analysis and literature review, we found sentence at the beginning or end position summarize and we used position scoring to assign larger weights to these voters.

V. Summary of Milestones

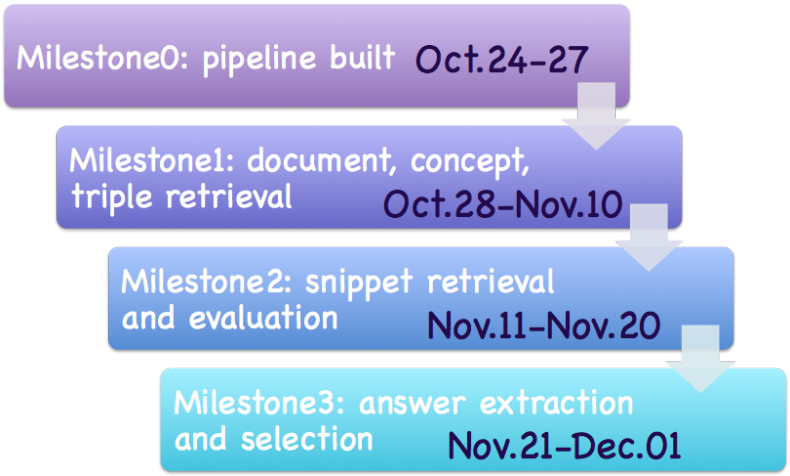
In order to keep on the right track and finish the designing and building our system in time, we have set several milestone and timeline for this final project.

In the milestone 0, we set the vision of our system which is to solve the Yes/No question with high precision. Then we figure out the requirements of our system, which is to read biological and biomedical questions, answer each question and evaluate the answer at last.

In the milestone 1, we implemented the pipeline system, built annotators for question, document, concept and triple. However, we met some problems with Github and UIMA environment settings.

In the milestone 2, we built the annotators for snippet, query preprocessing and hierarchical evaluation. Then we proposed some strategies for answer extraction and answer selection, as well as overall evaluation.

In the milestone 3, we finished the ComplexConceptQuery, answer extraction and evaluation for Yes/No questions, CasConsumer result integration, error analysis and Javadoc generation.



VI. Reference

[1] He, Jing, and Decheng Dai. "Summarization of Yes/No Questions Using a Feature Function Model." ACML. 2011.

[2] Tsatsaronis, George, et al. "BioASQ: A Challenge on Large-Scale Biomedical Semantic Indexing and Question Answering." AAAI Fall Symposium: Information Retrieval and Knowledge Discovery in Biomedical Text. 2012.

[3] Lewis, David D. "Naive (Bayes) at forty: The independence assumption in information retrieval." Machine learning: ECML-98. Springer Berlin Heidelberg, 1998. 4-15.

[4] Liu, Yuanjie, et al. "Understanding and summarizing answers in community-based question answering services." Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1. Association for Computational Linguistics, 2008.

[5] Svore, Krysta Marie, Lucy Vanderwende, and Christopher JC Burges. "Enhancing Single-Document Summarization by Combining RankNet and Third-Party Sources." EMNLP-CoNLL. 2007.

[6] Pradhan, Sameer S., et al. "Building a Foundation System for Producing Short Answers to Factual Questions." TREC. 2002.