Question & Answer System (QAS)

Business Insider Articles 2013~2014

James Jia, sjs8610, Northwestern IE308

Executive Summary

In this assignment, the goal is to be able to conduct the full process of question-and-answer analysis, using the corpus from Business Insider articles in 2013 and 2014. In general, four categories of questions should be answered: names of the CEO of a certain company, the list of company names that went bankrupt in a certain month and year, the factors that likely affect GDP, and what is the specific percentage effect of their effect on GDP as a follow-up question.

Various text processing and analytics methodologies are adopted in order to complete the tasks. Python `nltk` package is used to conduct text processing, such as word tokenizing, sentence segmentation, stop word removal, and lemmatization. Python `spacy` package is also used to simplify the final output generation process, by identifying the NER tags in the sentences. More specifically, from a user's question input, we will identify the question type, extract the question keywords, and then select the document candidates that are relevant to the keywords. Due to the likely large number of potential document candidates, we will use the classic `td-idf` algorithm (natively-written) to narrow down the subsets. Finally, we will extract likely answer candidates and use another natively-written scoring algorithm to produce the answer.

Although we do not have the ground truth output (the correct answers), based on several trials compared to a simple Google search results, the answers are only mediumly accurate. (**~40%** for CEO's name prediction, as it was the most factoid / searchable question). Time efficiency wise, a simple search such as "Who is the CEO of Salesforce" takes around **15 ~ 20** minutes, which are far more than a Google search, especially considering our document size. Therefore, there is a large room for improvement for this Q&A system developed.

Function Glossary:

	input	output	purpose
`qa_type()`	question (str)	qa type (str)	categorize the question
`q_keywords()`	question (str)	keywords (list of str)	generate keyword query
`doc_select()`	keywords (list of str)	doc_id (list of int)	generate all relevant documents
`doc_top()`	doc_id (list of int) keywords (list of str)	doc_id (list of int)	filter the documents via tf-idf
`ans_extract()`	doc_id (list of int) qa type (str)	ans (list of str) ans_doc (list of int)	generate all potential answers, and their corresponding doc id
`ans_score()`	question (str) keywords (list of str) ans (list of str) ans_doc (list of int)	ans (list of str)	filter the answers via natively-developed scoring algorithm
`generate_output()`	ans(list of str)	ans(list of str)	extract only relevant answer tokens

Overall Methodology

In this section, we will describe the process taken to create the QAS. Overall, in order to generate the answers from a specific question, we need to apply several text analytics techniques. The model created should be applicable to all types of questions listed in this assignment.

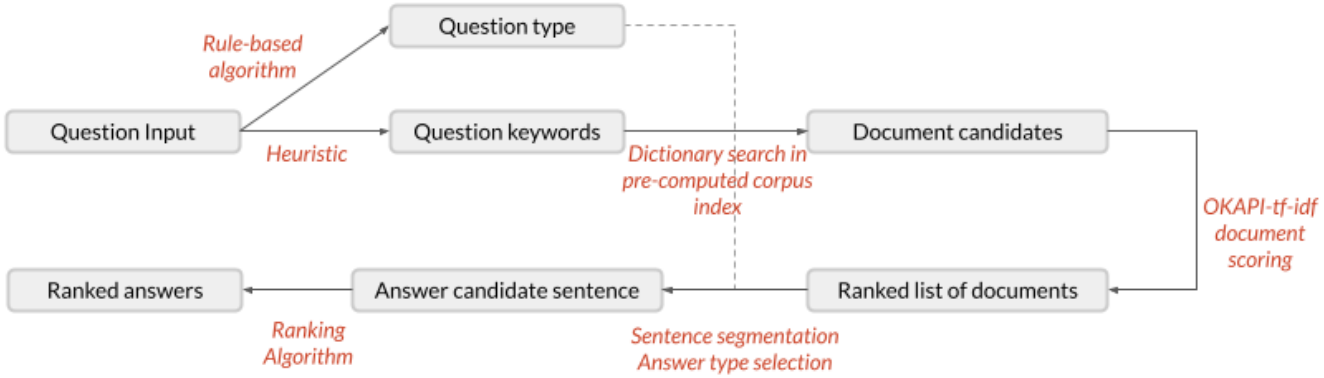

Figure 1: Overview of the question answer system

1. ***Question Type Analysis***
   - 1.1. One of the question types are assigned to each question input from the user: “name” (for questions such as “Who is Apple’s CEO?”); “org” (for questions such as “Which companies went bankrupt in 2019?”); “factor” (for questions such as “What affects GDP?”); and “percent” (for questions such as “What percentage of drop of GDP is associated with unemployment?”).
   - 1.2. A `rule-based model` is used because of its advantages in a domain-specific system. The traditional classification model was not adopted, because of the lack of readily available training datasets, and the fact that it is not very well adapted to the specific domain in this assignment.
   - 1.3. The question was tokenized and converted to all lower cases. The following keywords are searched for for each category of questions:
     - 1.3.1. “name”: “ceo”, “who”
     - 1.3.2. “org”: “which”, “bankrupt”, “bankruptcy”
     - 1.3.3. “factor”: “gdp”, “affect”
     - 1.3.4. “percent”: “percent”, “percentage”, “drop”, “increase”
     - 1.3.5. If there is not a fit, an error message will be displayed.
2. ***Question Keywords Extraction:***
   - 2.1. This step is needed for the following steps in which corpus will be searched and matched.
   - 2.2. The question is tokenized using `word_tokenize()` from the `nltk` package.
   - 2.3. Stop words and any “WH”-type questions are also removed using simple list iteration.
3. ***Corpus Indexing:***
   - 3.1. For every question asked, the first filtering should occur in which the system selects a subset of documents that contain at least one of the keywords.
   - 3.2. Rather than using corpus index packages such as `Lucene`, `Solr`, or `Elasticsearch`, a native document term matrix was built in the following manner:
   - 3.3. The full text is first cleaned and tokenized, with the removal of duplicates. That resulted in around 25K unique tokens of the full Business Insider corpus.
   - 3.4. Each token was set as a key in the corpus dictionary.
   - 3.5. An iterative program was written for each document. Every key was searched in the document, and if the key is found in the document, the document id will be appended to the dictionary.

3.6. With a completed dictionary, every question keyword can be searched through the dictionary, and a subset of documents will be readily available.

4. Document Candidate Selection:

4.1. For each question keyword, it was searched as the key in the `corpus dictionary`. A non-duplicate list of candidate documents will be generated.

5. Document Scoring:

5.1. For each document, `tf-idf` score was computed to rate the document's relative importance. Higher `tf-idf` score indicates higher relevance.

5.2. `tf` score is computed as 0.5 + 0.5 * frequency of token in document-of-interest / maximum frequency of any words in the doc

5.3. `idf` score is computed as the log value of the total number of documents in corpus, divided by the number of documents with the token in them.

5.4. Rather than using the `sklearn` package, a `native tf-idf algorithm` was written. Only a subset of documents was chosen. (5)

6. Answer Candidate Sentence:

6.1. Out of all the shortlisted documents, every sentence is segmented using `sent_tokenzie()` and evaluated to determine if it can become a potential candidate for answers. In this step, a `rule-based model` was constructed, in addition to the NER algorithm in the `spacy` package.

6.1.1. "name": "ceo", "PERSON" tag, "ORG" tag,

6.1.2. "org": "bankrupt", "bankruptcy", "DATE" tag, "ORG" tag

6.1.3. "factor": "gdp", "GPE" tag, "MONEY" tag, "PERCENT" tag

6.1.4. "percent": "PERCENT" tag

7. Answer Ranked:

7.1. A native scoring algorithm was written to evaluate the answer candidates. The final score for each answer candidate is "`A + B - C`".

7.1.1. A: Number of pair of adjacent tokens appearing in both answer candidates and questions;

7.1.2. B: `tf-idf` sum of words in both question and answer;

7.1.3. C: `tf-idf` sum of words that appear in question but not in answer.

7.2. Only a subset of answers are chosen. in this case, 5.

8. Final Output:

8.1. Finally, for each selected answer, apply the NER model again, and extract "PERSON", "ORG", "PERCENT" etc for the final output.

Performance

A) Run time efficiency

The run time clearly has a large difference in between stages. For the first 3 stages of analyzing question types, extracting keywords, and extracting all relevant documents take almost 0 time. The first 2 is because of an efficient rule-based algorithm, the document extraction is done by a hash table (python dictionary), which was pre-computed with each unique token in the corpus as key, and all document id in a list that such key appears as the value.

Filtering documents using `tf-idf`, extracting answers by filtering only the relevant sentence segments, and finally evaluating all sentences based on 3-way scoring functions (including `tf-idf` calculation) takes significantly longer, due to the large number of candidates, and Big O time.

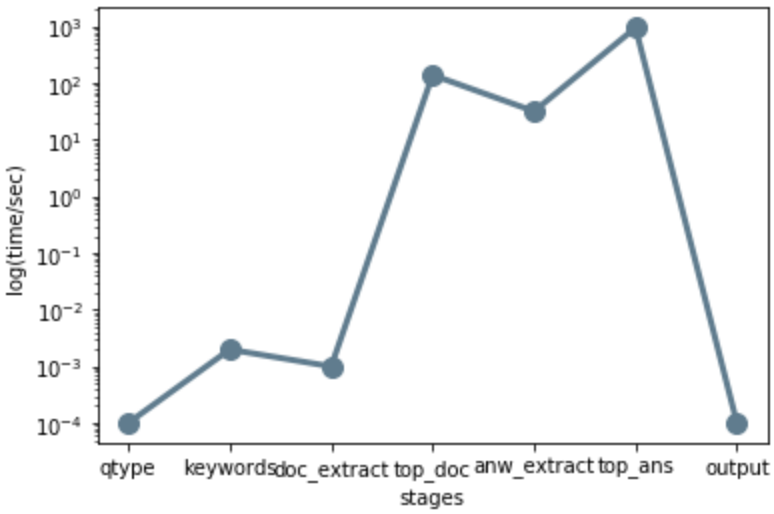

Figure 2: stages against time spent

B) ***Dictionary distribution***

As illustrated in part (A), pre-computed document indexing corpus saves the common algorithm time. However, by evaluating the number of documents per keywords (for example, 3 for `{{"twitter", [0,23,34]}}`, it can be seen that although most keyword tokens appear in less than 10% of all document corpus -- there are some keywords (such as “ceo”, “apple”) appear significantly more frequently, which might significantly slow down the computation time.

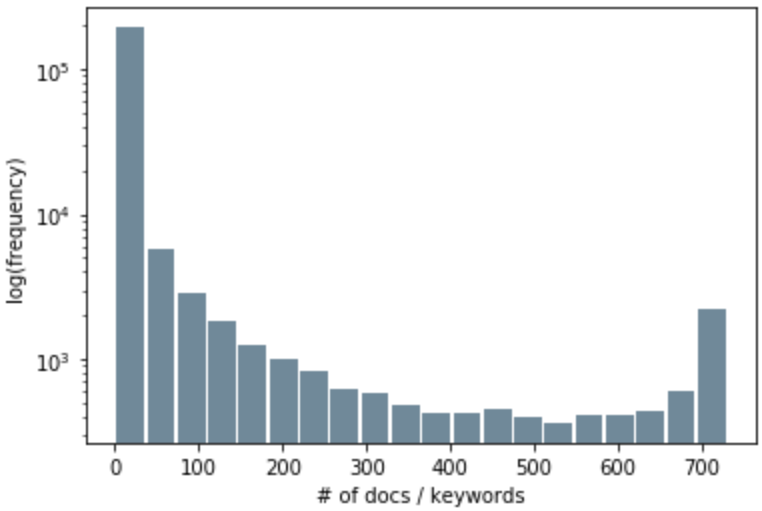

Figure 3: frequency histogram of number of documents / keyword

C) ***TF-IDF computation***

In `tf-idf` function, for each document, the max value -- maximum number of appearances for any tokens in a doc -- is the same. Therefore, such value (appearing in tf denominator) is pre-computed. As seen from the plot below, most of the documents have their score below 200, whereas some documents have their max words appearing over 400 times.

Such results might require more investigation, checking if there are any stop words, symbols that still left in the documents.

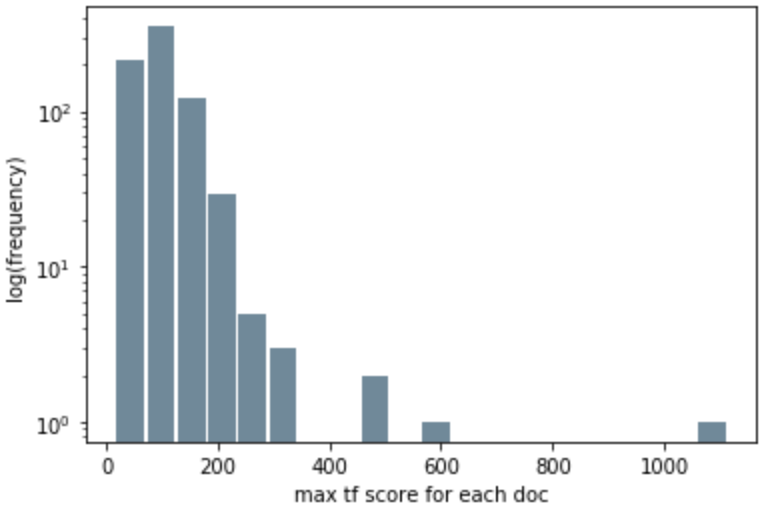

Figure 4: frequency histogram of document max word appearances

D) ***Improvement:***

a) There is a large room for improvement for the speed of the program. More efficient algorithms, or more cleverly pre-computed dictionaries might help speed up the program.
b) Build synonymous keyword libraries to improve potential search results, such as " company = LLC / LTD / INC / CORP" etc.
c) Utilize OKPAI_tf score to replace the TF-IDF score, taking into account the length of each document.
d) Create more efficient algorithms to filter through candidate documents, extract potential answers, and scoring answers as they are the bottleneck.