LING 573: Initial Project Report

Clara Gordon
University of Washington
Seattle, WA
cgordon1@uw.edu

Claire Jaja
University of Washington
Seattle, WA
cjaja@uw.edu

Andrea Kahn
University of Washington
Seattle, WA
andrea.m.kahn@gmail.com

Abstract

We implemented a question answering system to handle factoid questions from the QA track of the Text Retrieval Conference (TREC), using the AQUAINT Corpus of English News Text as a document collection.

1 Introduction

Question answering (QA) has long been a prominent problem in the field of natural language processing. In contrast to information retrieval (IR) systems, which return relevant documents based on search terms, a question answering system takes a natural-language question as input and outputs a natural-language answer. IR is typically a component of the system, but the addition of question and answer processing prevents users from having to sift through long documents to find the information they are seeking.

We implemented a question answering system to handle factoid questions from the QA track of the Text Retrieval Conference (TREC), using the AQUAINT Corpus of English News Text as a document collection.

2 System Overview

Our system is coded in Python. Third-party modules that we use include Indri/Lemur (for IR), pymur (a Python wrapper for Indri/Lemur), Beautiful Soup (for XML parsing), and NLTK (for tokenization). We chose Indri/Lemur for IR because of its specific handling of TREC-formatted question files. We currently use a stopword list taken from the Indri/Lemur documentation.

We used pymur to index the document collection. We created two indexes using two different stemming methods: Porter stemming, which is more aggressive, and Krovetz stemming. To obtain baseline results, we used the Porter-stemmed index as input to our system.

The core of our system is a three-part pipeline, consisting of modules for question processing, IR, and answer processing, respectively. The system architecture is shown in Figure 1.

3 Approach

The question answering system is called by a wrapper script, question_answering.py, which takes as arguments a file containing questions in TREC QA format, a path to the document index, a run tag, and a path to the desired output file. It uses the third-party module Beautiful Soup to parse the XML in the TREC document and generate a list of questions. It then passes the questions one-by-one to the pipeline described below. For each in the group of answers returned for each question, it prints the question, the run tag, the document ID associated with the answer, and the answer to the output file.

Classes that are used by multiple modules in the pipeline are defined in the module general_classes.py. These include:

- Question class: A Question object stores as attributes the TREC question ID, the question type, the TREC natural-language question stored as a string, and the "target" (the context given for a set of questions in TREC 2004-2006; defaults to None).

- SearchQuery class: A SearchQuery object stores as attributes a dictionary of search terms, each of which can be one or more words, mapped to weights indicating how important those terms are perceived as being, and an overall weight for the query, which will be used to calculate the probability of the corresponding AnswerCandidate.

- AnswerTemplate class: An AnswerTemplate object stores as attributes a set of basic search query terms from the original question and a

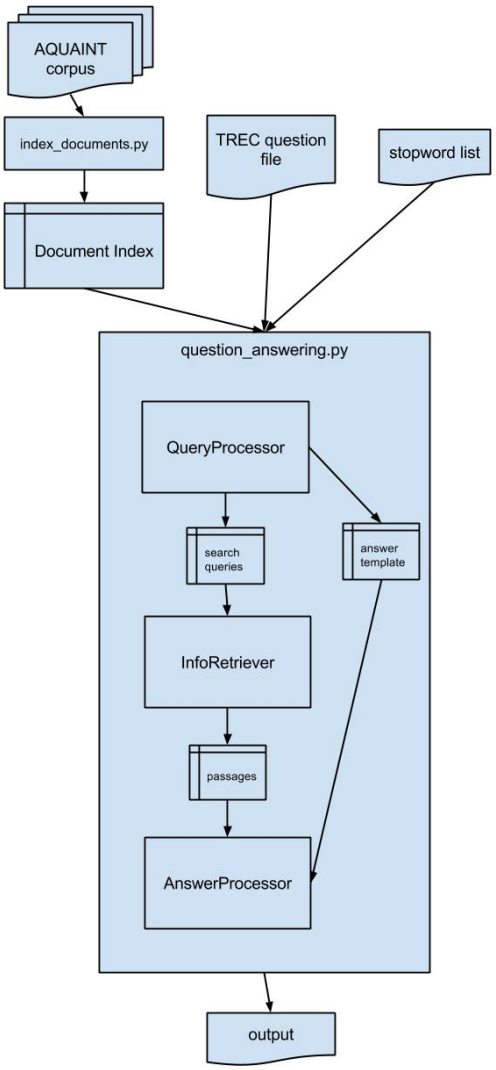

Figure 1: System architecture.

dictionary for the weights of each NE type, where the weights will be used to reweight AnswerCandidate objects during answer processing.

- Passage class: A Passage object stores as attributes the text of a snippet returned by Indri/Lemur, the Indri-assigned weight of the corresponding document, and the document ID.

3.1 System architecture

Our pipeline for processing a single question consists of three components found in three separate modules, described below. The pipeline takes a Question object as input and outputs a list of AnswerCandidate objects.

3.1.1 Query processing

The query processing module is responsible for creating one or more weighted search queries (which are passed to the information retrieval module to be used for passage retrieval) and instantiating an answer template (which is passed to the answer processing module to be used during answer ranking). A QueryProcessor object is initialized with a Question object and generates a vocabulary, or a list of words (i.e., whitespace-tokenized strings) occurring in the question/target and their counts. This vocabulary is used to initialize an AnswerTemplate object. (In subsequent versions of the system, we plan to implement a more sophisticated approach toward answer template creation, including question classification.)

The QueryProcessor is then used to generate a list of weighted search queries, each of which is in turn a set of weighted search terms. In the current version of the system, the QueryProcessor generates just one search query, which is the set of words occurring in the question and the question target, with the word counts as weights. In subsequent versions of the system, we plan to explore more sophisticated methods of query generation, including query expansion, as well as to test generating multiple weighted search queries for one question.

3.1.2 Information retrieval

The infomation retrieval module uses the Indri/Lemur IR system to retrieve a series of 100 snippets and associated documents for each set of query terms passed to it by the query processing module. Although pymur is used to index

the document collection, this portion of the system instead uses Python's subprocess module to directly query Indri using the IndriRunQuery command and various command line arguments. This is necessary because certain Indri functions, such as snippet retrieval, are not available from pymur. A stopword list, stoplist.drf, is used to improve search results by prompting Indri to ignore these words when performing document retrieval.

Once the snippets and associated document metadata are retrieved from standard out, a series of character-based split commands is used to segment the string into the relevant fields: document ID number, document weight, and snippet text. These fields are used to construct a Passage object for each snippet. A list of these objects is then passed to the answer processing module, which uses the snippets and associated metadata to derive answer candidates.

3.1.3 Answer candidate extraction and ranking

The answer processing module is used to extract and rank answers. An object of this class is initialized with a list of Passage objects, an AnswerTemplate object, and an optional stopword list. This object can then generate and rank answers. This is done in a series of steps.

First, possible answers are extracted from the Passages by generating all unigrams, bigrams, trigrams, and 4-grams from the text of each passage; the score of each of these possible answers is the sum of the negated inverse of the retrieval score of the passage it is found in. If an n-gram appears multiple times in a passage, the n-gram's score is updated each time the n-gram appears, so a possible answer that appears frequently in a passage is scored higher than one that appears just once in the passage. The negated inverse of the retrieval score is used because Indri/Lemur returns a negative score where higher (i.e. closer to zero) is better; we wanted a positive score where higher (i.e. further from zero) is better for our own readability and troubleshooting. While these scores are being calculated, the ID numbers of documents containing the n-grams are also tracked for later use. At the end, a list of AnswerCandidate objects is generated which contains a possible answer, its score, and the documents it is found in.

After this, the AnswerCandidates go through a filtering step. At this step, any answers that start or end with a stopword or contain any words from the original query (retrieved from the AnswerTemplate) or any standalone punctuation tokens are discarded. Then, a combining step updates the score of each answer to be the current score plus the sum of the scores of the unigram answers contained within it. This prevents unigrams from being the highest ranked answers and instead favors longer answers.

Next, the answers are reweighted. At this point, any answers that did not appear in more than one passage are discarded. Additionally, this is the place where constraints on named entity type from the AnswerTemplate can be applied to increase or decrease AnswerCandidate scores; currently, since no constraints are being generated at the query processing stage, there is nothing to apply.

Lastly, the answers are ranked by score, and the top 20 are returned. For each answer generated, one of the document IDs where the n-gram occurred is randomly selected as the source of the answer. In future development, a more clever method for selecting which document ID to use as the answer source will be employed.

4 Results

We evaluated our results using the mean reciprocal rank (MRR) measure with strict and lenient evaluation. At this point, we are reporting only baseline results, shown in Table 1 below.

System	Strict	Lenient
Baseline	0.00511	0.02894

Table 1: Baseline results based on automatic pattern scoring. All scores rounded to five significant digits.

5 Discussion

Our baseline results are very low. Error analysis indicates that this is most likely the result of the low quality of the snippets returned at retrieval time. Additionally, there are some duplicate documents, so an answer may be retained as being from more than one document when in reality, it is from two identical documents and should be treated as being from one document and thus discarded.

In subsequent versions of the system, we hope to address the first issue by augmenting the query processing module to include query expansion. Better search queries (and more of them) will hopefully improve the quality of the snippets returned in the information retrieval module.

In addition, we plan to encode more information in the answer template by implementing question classification in the query processing module. This will hopefully facilitate better answer generation and ranking in the answer processing module.

As mentioned above, we created two document indexes: one using Porter stemming and one using Krovetz stemming, which is less aggressive. Initial testing on a small question set demonstrated better results using the Krovetz-stemmed index. Due to issues we encountered with Indri/Lemur when using the Krovetz-stemmed index with the full question set, we used the Porter-stemmed index instead in the baseline system. In subsequent versions of our system, we plan to explore using the Krovetz-stemmed index to see if we achieve better results.

6 Conclusion

We implemented a baseline question answering system to handle factoid questions from the TREC QA shared task using the AQUAINT Corpus as a document collection. Our baseline results were very low, most likely due to the poor quality of the snippets returned by the IR component of the system, duplicate documents in the document collection, and the primitive methods we used for query processing. In future development, we plan to extend the query processing component of our system to implement query expansion and question classification, as well as to explore using a document index created with a different stemmer, in the hope of achieving improved results.

References

Sanda M Harabagiu, Dan I Moldovan, Christine Clark, Mitchell Bowden, Andrew Hickl, and Patrick Wang. 2005. Employing two question answering systems in trec-2005. In *TREC*.

Lynette Hirschman and Robert Gaizauskas. 2001. Natural language question answering: The view from here. *Natural Language Engineering*, 7(4):275–300.

Jimmy Lin. 2007. An exploration of the principles underlying redundancy-based factoid question answering. *ACM Transactions on Information Systems (TOIS)*, 25(2):6.