MIM QA System: Deliverable 3

Antariksh Bothale
abothale@uw.edu

Julian Chan
jchan3@uw.edu

Yi-shu Wei
yishuwei@uw.edu

Abstract

We detail the work currently done on MIM, our Question Answering system. We report a lenient MRR score of 0.3609 (strict score 0.2433), and detail the improvements made in the system to achieve said score.

1 Introduction

This document reports the creation and performance of an end-to-end Question Answering system. We have a basic setup in place for the overall task and present current baseline results.

2 System Overview

Our system architecture can be broken down into three major classes - MainFacilitator, TaskExecutor, and Session. Our information retrieval system processes queries and retrieves answers in 5 steps, namely Question Classification, Query Formation, Document Retrieval, Passage Retrieval, and Answer Processing. Each of these steps are encapsulated in their own class and derives from the TaskExecutor task. The Session object is used to pass information between these tasks. The implementation of each TaskExecutor is completely transparent to the MainFacilitator, whose job is to simply run the TaskExecutor(s) in the correct order and pass a Session object between them. The QuestionProcessor object is embedded in the Session object and contains methods that allows the TaskExecutor(s) to access the user query.

During start-up, MainFacilitator instantiates the five TaskExecutor(s) and put them in an array. When there is a new query and the AnswerQuestion() method is called, MainFacilitator creates a new Session object and calls Execute() on each of the TaskExecutor(s) sequentially. Barring any execution error, the answer will be written to the Session object.

The SystemEvaluator reads in all the question from the data set and calls MainFacilitator's AnswerQuestion() function. It then reads the answer from the returned Session object and writes to the output file.

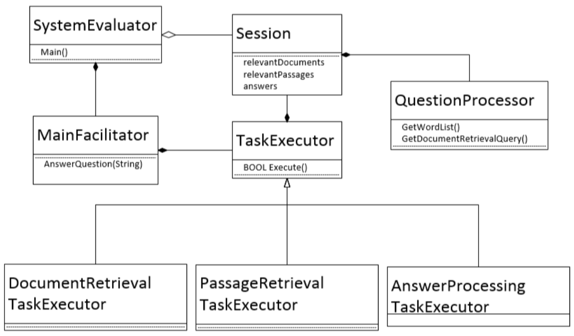

Fig 1: class diagram of the MIM QA system

3 Approach

3.1 Question Classification

We build an SVM classifier that classifies a given question into one of the question types defined in the UIUC question taxonomy [4]. The classifier uses unigrams as well as the *Wh-* words and the headwords as features. (Headwords are extracted from parsing the questions using the Berkeley parser [2]).

We train the classifier on the dataset from [4] with 5500 labeled questions, and test the classifier on TREC 2004 and 2005 question-answering data. Since the test data are only labeled for coarse-grained question types (6 classes), we can only evaluate the classifier with coarse-grained classification, for which our SVM classifier achieves 87% accuracy. However, for our end-to-end system we retrained the classifier for fine-grained question classification (50 classes) because coarse-grained class label does not provide sufficient information for selecting correct answers.

3.2 Query Formation by Redundancy-based Web-boosting

In this step we convert the question into a query that can be used for document retrieval. The query indicates that the subsequent module should search for the question terms (except the *Wh*-words) in the body text, and for the target words either in the title or the body text.

We implemented query expansion using a redundancy-based web-boosting strategy. The original question (together with the target words) is passed to the Ask.com search engine and the snippets of the first 40 results are extracted. From these snippets we obtain bigram counts and keep only the top 10 most frequent ones that do not contain any words from the original question. Often times, we found these top bigrams to contain the correct answer to the questions.

We then harvest unique terms from these top bigrams and add them to the query to the document retrieval system. We were able to increase our accuracy significantly (at least 10 points).

3.3 Document Retrieval

The Document Retrieval Subsystem takes a document retrieval query from the session and populates the session with a list of hits from the indexing and retrieval system, as described below. This list of documents is then used for Passage Retrieval and Answer Processing.

For the purpose of indexing and searching the documents, we use Whoosh [3], which is pure-Python based text indexing, search, and spell checking library. The choice of a Python based library above alternatives such as Lucene and Indri/Lemur was motivated by its ease of integration and use with our Python code-base, and its functionality being comparable to that of its Java counterparts.

3.3.1 Indexing

The Beautiful Soup XML parser [1] was used to parse all the documents and extract the Document ID, Headline and Body Text. Whoosh's in-built indexing engine was used to index all the documents present in the corpus. The current system uses Whoosh's default indexing mechanism, which does not process the text, we plan to improve on this by incorporating tokenization and stemming. The indexing schema was designed to store the Document ID and Headline with the Index, while the Body Text was indexed but not stored as it would have caused unnecessary duplication of data. Instead, the text is retrieved from the document using the Document ID whenever needed. This index is currently stored on Patas.

3.4 Passage Retrieval

This step reads the files retrieved in the last step, extracting all passages in the documents that contain the keywords (the target words, the words in the question, and the words added from web-boosting). A "passage" in our current system is simply defined as a sentence, where sentence segmentation is handled by Whoosh. These passages are then scored based on the density of keywords in the passage. The passage extraction and scoring function are taken from Whoosh, which are originally used for returning snippets for search results.

3.5 Answer Processing

We do re-ranking based on pattern matching, and then chop off anything after 250 characters. If the question is classified to be of type *HUM*, any answer with named entities of type *PERSON* (as classified by NLTK) are bumped above answers without such entities. Likewise, for questions of type *NUM:date*, answers that have a pattern match with a date / year pattern are bumped up. Finally, for questions of type *NUM*, we prioritize answers with numerical patterns in them. If there are multiple answers that satisfy any of the above criteria, we do not break ties between them and return them in the relative order in which they were returned by the passage retrieval system.

4 Results

Under the baseline system, we obtain an aggregate score of 0.2433 under the Strict evaluation scheme, and an aggregate score of 0.3609 under the Lenient evaluation scheme.

The big improvement from last deliverable comes from several sources: First, we include the target words in our query; second, we use the whole sentence rather than automatically generated snippets as our answers; third, we exploit search results from the World Wide Web and use question classification and relevant answer re-ranking.

5 Discussion

We have achieved fairly good results by improving key processes in the QA pipeline. We also realized (from our experience and also from that of other teams') that techniques like n-gram scoring only gave marginal improvement (if any) and we dropped them in favor of our current system. It is encouraging to see that the results are pretty good with the improvements we made and we expect even better results in the final deliverable.

Acknowledgments

We thank C J Hsu for the code snippet for interfacing with Ask.com for the Web Boosting mode that he shared with all students on GoPost.

References

[1] *Beautiful Soup*. URL: `http://www.crummy.com/software/BeautifulSoup/`.

[2] *Berkeley parser*. URL: `http://nlp.cs.berkeley.edu/Software.shtml`.

[3] Matt Chaput. *Whoosh*. URL: `https://pypi.python.org/pypi/Whoosh/`.

[4] Xin Li and Dan Roth. *Learning Question Classifiers*. URL: `http://cogcomp.cs.illinois.edu/Data/QA/QC/`.