Project 3 Report: Question Answering

Ryan Enderby (rae83), Liane Longpre (lfl42), Jordan Stout (jds459)

Overall Approach

Our final approach is built around the idea that based on question type (who, what, when), we know what kind of named entity should be returned as an answer. Documents can be scored based on the occurrences and co-occurrences of content words from the question, and from the top documents, sentences can be extracted and scored in a similar manner. From here, named entities of the proper type can be extracted from the top scoring sentences, and these will serve as educated guesses to the answer.

Rationale

We made these key design decision for a number of reasons. The decision to assign an appropriate named entity type for the answer based on question word is guided by the logical conclusion that a "when" question will necessitate an answer of type "time," just as "where" questions would need an answer the provides a place and "who" would best be answered by returning a person or organization.

We chose to score documents based on the number of occurrences of content words and co-occurrences. This is based on the rationale that a relevant answer will contain many of the same content words as the question. Further, we chose to weight the co-occurrence of different content words within spans like sentences as higher than just the occurrence of any content word. To illustrate why, consider the example of question 89. We distill this question into the content words 'Belize' and 'located.' We predict that a relevant document is one that mentions 'Belize' and 'located' together a lot, not just one that says 'Belize' a lot. As such, the co-occurrence of different content words is more important than just the absolute count of content words.

The content words used to score documents and sentences are distilled from each question, and we see reducing each question to a small number of content words as an important step for both efficiency and accuracy. Since we have a system that filters documents based on words from the question, not having to search for function words as well as content words will make our system considerably faster. This is because including function words would not only increase the number of words we have to search for, but also because function words occur with much higher frequency in the answer documents than the content words in question. Similarly, since function words are so common, we do not imagine that they would provide enough information in ranking potential answers to be worth the drawbacks.

Using scores based on the content words and methods described above, we extract the top scoring documents for further analysis. From our testing, we found that reducing the number of top documents analyzed from 10 to 5 had minimal impact on MRR. From these top documents, we split each into sentences. Sentences are then NER tagged, and those that do not contain the appropriate named entity tag are removed, as without the proper type of entity, we do not predict the correct answer can exist in the sentence. Remaining sentences are scored using the same co-occurrence rationale as documents. Named entities are extracted from the top 5 scoring sentences so that answers are concise, of the proper entity type, and fit within the 10 word restriction. If, even after extracting only named entities, the answer is still more than 10 words, we just pick the first 10. We do not have a good way of identifying which entity of the proper type is most likely to be an answer, as such, we implemented this naive solution.

We hypothesized that this type of system would be effective and efficient since it identifies the proper type of entity to be returned as an answer, establishes content words to guide the extraction of relevant documents, then extracts entities of the right type from the most relevant sentences.

Final System

The general steps of the system involve processing each question, removing function words so that each question is distilled into a short list of content words. We remove function words by using NLTK's POS tagger, and removing words in the question that are of a function word type. Based on the question word (e.g. who, where or when), we also assign a list of possible named-entity types that our answer should contain. 'Who' questions should return answers with a 'PERSON' or "ORGANIZATION' named entity, 'When' returns a 'TIME' answer, and 'Where' returns a 'LOCATION' or 'GPE' (geo political entity).

We then iterate through each document in the corresponding question number folder, assigning a score based on the number of occurrences of content words and the number of co-occurrences of different content words within a span. As described above, co-occurrence of different content words adds more to the score than just the occurrence of any content word, although both factor in. For the top scoring documents, in this case 5, we split them into sentences, POS tag the sentences, then named-entity chunk the sentences. We filter out sentences that do not contain a named entity of the appropriate answer type, as defined above. We score the remaining sentences based on co-occurrence of different content words from the question, as defined above, but at the sentence level. For the top 5 scoring sentences, we extract the appropriate named-entity as our guess, or, if no sentences meet our criteria, "nil".

Intermediate System

We also constructed an intermediate system to help us make informed decisions about the design of our final QA system. This intermediate system used POS tagging to identify content words in the question. Each document in the folder was processed, creating a dictionary of the occurrences of every word (e.g. {word: (document number, index in document), …}). Using this dictionary, we identified spans of text within documents that had the highest occurrence of content words and co-occurrence of different content words. These spans were then returned as answers. This system bears several similarities to our final system, but does not use NER to identify the proper type of answer entity, and does not filter out top documents and then top sentences, which our final does.

This system was an improvement over our baseline and helped us figure out a few important points en route to the final. First, we initially implemented this system where occurrence of any content words was treated the same as the co-occurrence of different content words. This showed us a clear shortcoming, as, going back to the Belize example, a sentence that contains 'Belize' four times was scored the same as one that says 'Belize' twice and 'located' twice. Clearly, if we are looking for the location of Belize, the second would be preferable. Experimenting with this helped us a build a final system that can account for the co-occurrence of different words.

We also saw that many times, multiple guesses came from the same document. As such, we built our final system to extract the top documents, since our evidence suggested that we could ultimately analyze the sentences of only a handful of top documents without sacrificing much accuracy.

Baseline System

For our baseline system, we first pre-process questions in order to search for only the content words within a question. To do this, we manually selected a list of function words to remove from the questions. We then reduce the remaining words to a single keyword by selecting the longest word. As such, each question ends up being represented by the single longest content word from the original prompt. The decision to keep only the longest word was a naive means of trying to get to only the most "important" content word, with important meaning the most specific word to the question, as longer words appear less frequently in text than shorter words. Having a content word that is longer means the word is likely to appear less frequently in the answer documents, and therefore allows us to filter to a more specific set. For a preliminary implementation, this will select a relevant word in almost all questions, even if it gets rid of a lot

of valuable information. We then iterate through the documents for each question and find the first occurrence of the keyword in the question. The system's 5 answer guesses are different 10-word windows surrounding the keyword.

We see that our baseline system most often does not find the answer, but that if the answer is found, it is usually found in the first or second guess. This makes sense because our guesses were simply changes in the window of the same answer. Additionally, we see that some of our answers include metadata from the document.

Specifics of the Final QA System

Question document:

We preprocess the question document by removing any occurrences of '\n', '\r', 'Number: ', and 'Description: '. The file is then split on <top>, <num>, and <desc> tags. We tokenize the document, then part-of-speech tag. A choice of certain tags are removed: ['IN', 'DT', '.', 'WDT', 'MD', 'VBZ', 'TO', 'PRP', 'POS', 'WRB', 'WP']. This ensures that the remaining words are content and not function words. These content words are keywords that we later use to score answer document sentences. We represent the processed question document as a dictionary with keys of question numbers and values of lists of content words.

We also use the first word in the question, "Who," "When," or "Where" to determine the correct named entity types that our answer should return, e.g "PERSON," "TIME," and "LOCATION" or "GPE," respectively.

Answer documents:

For each question, we iterate through each document and assign a score based on the occurrences and co-occurrences of content words. A certain amount of weight (+1) is assigned for each content word from the question that occurs in the document. A higher weight is assigned when different content words both appear in the same sentence (+10). We rank the documents and perform further analysis on only the top N number of ranked documents. We chose the aforementioned weights after testing the system and seeing which additional weight for co-occurrence of different content words ended up giving us answers with a higher MRR. We tested a system that return the top 10 documents and top 5 documents, and found that returning only the top 5 documents made our system almost twice as fast, while only reducing MRR by a small amount (0.005).

Searching/scoring algorithm (for each question):

In each of the top-scoring documents, we split the document by sentence. For each sentence, we perform named-entity tagging and remove any sentences that do not contain the correct NER tag corresponding to the question type. We rank the remaining sentences according to the content words contained in the sentence. We use two different measures. The first adds a certain amount of weight to an occurrence any time it appears. The second measure adds a much higher amount of weight to an occurrence if it overlaps with a different content word within the same sentence. This scoring system is similar to how we score and rank documents earlier. This returns a dictionary of {(document number, sentence): score, ...}.

The final score dictionary is then sorted by score, and the top 5 sentences are selected as our guesses. In order to create our guess, we again perform named-entity tagging and only return the NER tag corresponding to the question type as our guess for each given sentence. In the event that the contents remaining in our answer sentence contains only content words from the question, we return "nil". We reason that if there are no other correct NER tag types in the sentence than what was presented in the question, the answer has not been found. Our 5 guesses are maintained in a list as: [(question number, doc number, string guess),...].

We iterate through this final list for each question in order to generate an appropriately formatted output file with the results.

Example walkthrough: Question 320

- We part-of-speech tag the question "Who made the rotary engine automobile?" and remove the unwanted parts of speech to extract content words, maintained in a list: ["rotary", "engine", "automobile"]. We also assign the appropriate named entity answer type to be 'PERSON' or 'ORGANIZATION.'
- We iterate through the documents in the corresponding answer directory for question 320 and assign a score to each document, +1 for each content word occurrence and +10 for each overlap of different content words within a span of 10 words. We rank the documents accordingly.
- For the top 5 ranked documents, we split the document by sentence and perform NER tagging. For each sentence, we remove every sentence that does not contain the 'PERSON' or 'ORGANIZATION' tag (chosen based on "Who" from the question).
- For the remaining sentences, we assign scores to each sentence based on contents of the sentence in the same manner as documents are scored, but at the sentence level. We create a dictionary mapping each sentence to its score: {(doc number, sentence): score ...}

- We select the top 5 scored sentences in order to generate our guesses in tuple the form (doc number, sentence):
  [('2', 'Wankel began work on a rotary engine in 1926, culminating with production of the new-design motor at the West German NSU Motor Works in 1957.'),
  ('2', 'ID: 320\tRANK: 2\tSCORE: 20.412918 <DOC> <DOCNO> AP881013-0321 </DOCNO> <FILEID>AP-NR-10-13-88 1518EDT</FILEID> <FIRST>r f BC-Obit-Wankel 10-13 0129</FIRST> <SECOND>BC-Obit-Wankel,0139</SECOND> <HEAD>Inventor of Rotary Motor Dies At Age 86</HEAD> <DATELINE>LINDAU, West Germany (AP) </DATELINE> <TEXT> Felix Wankel, inventor of the rotary engine, has died in his native country, friends of the motor designer said Thursday.'),
  ('1', 'Mazda continued work on developing the Wankel rotary engine and made the concept a successful one.'),
  ('95', "The only problem is that the Ford Escort's engine is made by Mazda."),
  ('4', "Mazda Motor Co.'s HR-X coupe (the letters stand for Hydrogen Rotary Experimental) runs on hydrogen gas that is stored in a rectangular tank and fed into a rotary engine.")]
- We again perform NER tagging on our sentences. We extract the 'PERSON' and 'ORGANIZATION' tag from each sentence and enter the result as our answers: [('2', 'Wankel'), ('2', 'Rotary Motor'), ('1', 'Mazda'), ('95', 'Mazda'), ('4', 'Mazda Motor Co.')]

Results & Analysis

Running our baseline system with Perl gives the following result:

Mean reciprocal rank over 232 questions is 0.084
209 questions had no answers found in top 5 responses

Runnin our intermediate system with Perl gives the following result:

Mean reciprocal rank over 232 questions is 0.140
176 questions had no answers found in top 5 responses

Running our final system, pulling top 10 docs for each question gives the following result:

Mean reciprocal rank over 232 questions is 0.200
153 questions had no answers found in top 5 responses.

Running our final system, pulling top 5 docs for each question gives the following result:

Mean reciprocal rank over 232 questions is 0.195
155 questions had no answers found in top 5 responses.

Starting from our baseline system, we achieved significant improvement to the intermediate system and again to our final system in terms of both mean reciprocal rank and finding answers for more questions. Improvement from baseline to intermediate can be attributed to how baseline stops at the first occurrence of a content word from the question, whereas intermediate examines all occurrences, and ranks spans of text, returning the highest scoring ones. Further improvement from intermediate to final system can be attributed to utilizing what we learned about scoring from the intermediate system, but now incorporating NER so that we identify the proper type of entity to return, and only return entities of the appropriate type from top scoring sentences.

We examined the answers our intermediate system produced in order to identify possible improvements. We noticed that answers often did not contain the correct named entity type for which the question asked. This led us to incorporate identification of the proper entity type in the question and implement NER for answer extraction. We used our methods from our intermediate system to rank documents and sentences, but implemented a final system that identifies the desired NER tag and only considers sentences containing the correct tags. We notice a significant improvement in mean reciprocal rank as well as total questions answered after this alteration of our system.

One clear improvement from both our baseline and intermediate approach is that answers are not only much more accurate, but also much more concise. In many cases, our final answers are only a couple of words, and yet the answers provide a much higher MRR than the two initial systems with almost always return an answer of the maximum length (10 words). This can be attributed to how we extract the appropriate named entity type from possible answer sentences, as opposed to blindly returning a 10 words span that contains content words from the question prompt.

In the end, our program takes around 20 minutes to generate the answer document for the doc_dev folder, most of which is spent iterating through subtrees of named entity tagged chunks. We initially were worried that this was too slow, but after reading Piazza, see that in the context of other systems taking hours to run, this is not too bad. On average, it takes 5 seconds to find an answer. In future potential iterations, we would like to either find a better way to perform this function or find a way to prune more sentences away before performing this function, such that we can safely check fewer sentences without sacrificing MRR.

Work Distribution

Most work was accomplished during team meetings. However, outside of the meetings, we divided work such that Ryan and Liane completed the baseline system and final system, and later Jordan focused on getting our answers to write to a document in the proper format while Ryan and Liane finished the write-up.

Code

Our final system is in the file ner-qa.py. This file is currently set up to run on the test corpus, and assumes that doc_test is in the same directory as this python file. It requires os, csv, re, numpy, operator and nltk packages, running in Python 2.7. It writes to answers_test.txt in the same directory.

Our intermediate system is in the file pos-qa.py. This file is currently set up to run on the test corpus, and assumes that doc_test is in the same directory as this python file. It requires os, csv, re, numpy, operator and nltk packages. It writes to answers_test.txt in the same directory.

Our baseline system is also include in the zip file submitted to CMS. This file is baseline.py and it assumes that doc_dev is in the same directory as this python file, and requires os, csv, re, numpy and nltk packages. It writes to answer.txt.

APPENDIX A:

Extra rationale for final system

Further, we chose word co-occurrences as a feature for scoring spans of text because of our expectation that sentences with the same content words as our question were mostly likely to be appropriate answers. We first implemented a system that treated co-occurrence of all content words the same, we call this our "initial intermediate system" from here on out. Details of our intermediate systems are in appendix A at the end of this design document. For example, our system pulls out "Belize" and "located" as the content words for question 89. This intermediate system would treat a sentence with "Belize" twice and "located" twice the same as a sentence that says "Belize" four times.

We looked at output answers and saw the clear flaw in this system. It was ranking sentences that said "Belize" a number of times higher than ones that might have said "Belize" and "located." Just because a sentence repeats one content a word a lot does not make it more accurate. We therefore modified our QA system to give a much higher weighting to co-occurrences of different content words. This improved the mean rank and number of answers found in our final system.

As such, we built an implementation to test our hypothesis that returning text segments based on the occurrence and co-occurrence of content words from the questions would be an effective QA system.

Intermediate System

Answer documents:

For each question, we iterate through each document and index the words in the documents. We create a dictionary that maps the word to a list of locations the word appears in as so: {word: [(file number, index in file),...]}. We do this so that searching can be done in constant time, and the documents need only be fully iterated through once.

Searching/scoring algorithm (for each question):

Data structures

- We create a document occurrence dictionary that maintains a list of all occurrences of any content words from the question for a given answer document. The dictionary maps

values as: {doc number: [(index of word, word),...]}. This structure is created by calling function get_doc_occurrences.
- We create a score dictionary that maintains the score of each content word occurrence. The dictionary maps values as: {(doc number, (index, word)): score}. Scores are initialized at zero. This structures is created by calling function get_init_scores.

In order to calculate the scores for each content word, we use two different measures. The first adds a certain amount of weight to an occurrence based on the document in which it appears. Weight is given to documents proportional to the total number of occurrences of content words in the document. The second measure adds a certain amount of weight to an occurrence if it overlaps with a different content word (window of 10 both before and after the word occurrence).

The final score dictionary is then converted into a sorted list maintained as [(doc number, (index, word), score),...]. We use the first 5 elements of this list to generate our guesses. We access the 10 words surrounding the index of the word occurrence in question in order to create the string guess. Our generate_guesses function will throw an error if no content words from the question are found in any of the answer documents. We catch this error and return "nil" as the answer when this occurs. Our 5 guesses are maintained in a list as: [(question number, doc number, string guess),...].

We iterate through this algorithm for each question in order to generate guesses into an output file with the results.

Example question: 320

- We part-of-speech tag the question "Who made the rotary engine automobile?" and remove the unwanted parts of speech to extract content words, maintained in a list: ["rotary", "engine", "automobile"]
- We iterate through the documents in the corresponding answer directory for question 223 and index each word to create a dictionary of words, maintained as: {word: (doc number, index of word), ...}
- For each content word from the question, we iterate through the answer word dictionary and create a dictionary only containing content words: {doc number: [(index of word, word), …] ...}
- We create a dictionary mapping each content word occurrence to an initial score of 0: {(doc num, (index, word)) : 0, ...} ie {(84, (155, "rotary")): 0, ...}

- We add weight to each word occurrence proportional to the total number of content word occurrences in the document. For instance, in document 84, there are 7 content word occurrences, so particular instance (84, (565, “engine”)) is assigned 7.
- We add weight to each word occurrence if the occurrence overlaps with a different content word, where overlaps is defined as being within 10 words of the other. For instance, in document 72, “rotary” at index 491 overlaps with “engine” at index 492, so 100 is added to the existing weight of particular instance (72, (451, “rotary”)).
- We sort the dictionary based on scores and return a list of sorted word occurrences: [((doc number, (index, word)), score), ...]
- We take the first 5 elements of the sorted list of word occurrences:
  [((4, (931, “rotary”)), 209),
  ((11, (1134, “rotary”)), 208),
  ((72, (492, “engine”)), 120),
  ((72, (491, “rotary”)), 120),
  ((1, (210, “rotary”)), 118)]
- For each, we find the word at its index in its document and take a window of 10 words around the occurrence as a string, with the occurrence in the middle of the window. We return a list of guesses in the form [question number, doc number, guess]:
  [[320, 4, “. Mazda bills its hydrogen rotary engine as “ the”],
  [320, 11, “HR-X car . A hydrogen rotary engine is a clean”],
  [320, 72, “and objectives of hydrogen rotary engine (RE below)”],
  [320, 72, “background and objectives of hydrogen rotary engine (RE below”],
  [320, 1, “produced the R0 80 Wankel rotary engine car , which”]]

As mentioned in the *Approach* section above, we saw that this was because our initial intermediate system treated the co-occurrence of any content words the same, even treating multiple occurrences of the same word as equivalent to occurrences of different content words. After analysis, this intuitively made sense as a potential issue. For example, our system pulls out “Belize” and “located” as the content words for question 89. This intermediate system would treat a sentence with “Belize” twice and “located” twice the same as a sentence that says “Belize” four times. Just because a sentence repeats one content a word a lot does not make it more accurate or relevant to our question. As such, we saw that this approach did not work well, and instead implemented higher weighting for the co-occurrence of different content words. This led to our intermediate system, which had better performance in both measures. Therefore, we concluded that oc-occurrence of different content words from the question is one of the strongest cues for identifying a relevant answer. Details of our intermediate system implementation are added at the end of this design document.