FSC at TREC

Stephen Taylor Orlando Montalvo-Huhn Nikhil Kartha

January 26, 2008

1 Introduction

This is the first time that Fitchburg State College has entered the competition and the first project that the students had worked on of this sort. We decided that we would split our time between building an infrastructure and on specific techniques for information processing, and to use the project to help us understand the problem better for subsequent years. Because of the time we needed to research and understand the problem and when we started we had a real time constraint. We decided to do some fast prototyping and use simple information processing techniques to test the basic infrastructure. To allow for easier insertion of more complex methods we decided on a layered approach to the information processing. A three layer approach seemed to make the most sense. The first layer would find the document that may have the answer. The second would try to find the sentence that answered the question. And the third would try to extract the answer from the sentence. We wanted to try a number of different approaches to processing information at each layer. However because of lack of time, we only got a chance to try a few. We spent much of time experimenting with the document retrieval portion. Even when you find a method, you need time to play with the parameters with the weight multiplier and any number of tweaks. Unfortunately, we didn't get much time to do that.

2 Approach

We decided on a quick prototyping development approach. This would allow us to have something up and running very quickly. We would also be able to stop development at anytime and have a working system (even if it doesn't work very well). We thought this approach would be the one that would most likely give us time for tweaking.

2.1 Brute Force

Brute force as first pass, didn't want to get bogged down We started with a brute force approach, using regular expressions and hand parsers. Since we did not know which areas would be the most likely to be the most important, we decided that we should create a system quickly and then concentrate on the areas that seem like they would be most likely to produce the best results.

2.2 Use packages

Use packages when we could We wanted to be able to build on previous work when possible, so we tried to find a package that would give us plenty of things to play with. For this reason we chose the Python natural language toolkit (nltk)[1] . We were very pleased with its interface to WordNet[2] .

[1] nltk.sourceforge.net
[2] wordnet.princeton.edu

2.3 Use proximity in sentence searches

Although we did not know what document query approach we would be using for searching for the documents, we decided early to use sentence proximity as a way to identify sentences that may contain the answer to questions. We tried a couple of different home grown methods counting the number of words in common and using synonyms. We also attempted a home grown distance method, but finally decided on the cosine distance method.

2.4 Word Types and Tagging

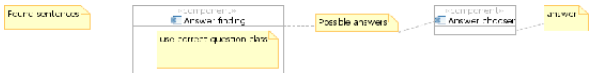
Use word types and tagging, depending on question It seemed to us that the quickest way to get results where you recognize sentence types and certain word patterns was to use regular expressions. Regular expressions can be used to recognize certain text such as dates and numbers, without a lot of work. It can be difficult to cover all cases, but you can get many fairly easily. You can also use regular expressions for recognizing types of questions. We divided the questions in basic types: who, when, how many, etc. Then we developed RE patterns for recognizing these types. We knew that we would only be able to tag a limited number of words using regular expressions and that it would be difficult to find phrases this way, so we use the nltk to help us tag part-of-speech and chunk phrases.

3 Structure

3.1 Deployment diagram

Here is a high level diagram of the whole system. The blogs and website data is used to create an index.

There is a question parser that provides a query for the index and information for finding sentences and answers. The query is used to find the documents that contain the information we need. Those documents are then passed to the sentence finder which finds the sentences that contain the answers. Once we have the sentences that may contain the answer, we still need to extract the answers. The answer finding component looks through each of the sentences and picks out the phrase or word that could be the answer. The answer chooser then looks for the phrase that appears most often and uses that for the output answer.



We built a layered system for finding answers. The idea was to look for documents which may have the answer, then narrow it down to the sentences and finally extracting the answers from those sentences.

3.2 Layers

3.2.1 Search for docs

The first step was to find the documents. Before we do this we need to know for what we are searching. This required us create queries.

3.2.1.1 Creating queries

To create queries we decided on a simple method. We create a list of words contained in the question. We remove the stopwords, then we look for synonyms using WordNet. In creating the search we use the words in the topic, the words in the sentence, and the synonyms, attempting to give words from the different lists different weights. The way the query is created depends on the type of search. This means that the query creation code needed to be part of the query module. We pass the word lists to a method in the module to perform the query creation.

3.2.1.2 Web searches

We built an internet search module which not used used in the final QA processing. We created a module that would take a website and create a query specific to that site. The query would be sent, and the returned html would be scrubbed for the list of websites. Once we got the list of websites, we would asynchronously look up the web pages. This process turned out to be quite easy. Much more difficult was finding the real content on the web pages. Writing this from scratched turned out to be difficult as web pages can have very different formats. We created a table of all the text elements in the document with a count of words in each element. The one with the most words was used. We fiddled with this ignoring some elements like ¡br¿ and ¡p¿ until we found a combination that seemed to work best. This actually gave us pretty good results.

3.2.1.3 Indexing

For the workshop, what we really needed was to search the documents given to us by the the competition, so we needed to index the TREC documents. We used Lucene[3] since it's been out in the in the field for a while. After trying a couple of different distributions, we settled on PyLucene[4] . It was not as fast as some others, but it seemed quite stable. An unexpected result of our going to using an index of the TREC documents, instead of the web search, was the number of correct answers we got dropped by half. We attributed this to the theory that search sites have been tuned over several years. Another issue we encountered was that the synonyms returned by doing a synset closure search gave us synonyms that increased our false hits significantly. On the other hand, not using the synonyms meant that there were questions for which we never got the documents containing the correct answer. In the end, we weeded these out the false hits during the sentence search, which will be described below.

[3] lucene.apache.org
[4] pylucene.osafoundation.org

3.2.1.4 File Searches

It was our plan to use the PyLucene indexing for the final project, however, when we got the trec documents and began to run tests with them, we found that python xml/html parsers were choking on some of the documents. This was especially true with the blog data. We made changes to the PyLucene modules which fixed a number of problems, but progress really slowed down while we fixed these. In the end, we decided to put aside the indexing and use the top docs, the fifty top documents for each question as provided by the workshop organizers. This required us to create a new module that just to handle the top docs.

3.2.2 Search for sentences

The second layer is the sentence search. After finding the documents, we need to pick out the sentences that contain the answer to the question. Finding the (one) sentence with the answer would be highly unlikely. Instead, we pick out a set of sentences which might have the answer and choose the answer that appears the most often. Since we may have had false positive hits in the document search, a decent method here can cull those documents by not having good sentence matches. We need the sentence search to be fast since we could be looking through a large number of large documents. This could mean a large number of sentences, which we do not want to compare to each other. Instead, we need a way to calculate a suitability number which can then be used to sort the sentences. We use two methods for the calculation. The first method is a word match count. We mentioned earlier that we keep the search words in 3 lists: One for words in the topic, one for words in the question, and one for synonyms of words in the question. Each one of these gets a different weight. So, for each word in the current sentence that matches one of the words in the list, we add a certain amount. In general, the words from the question get the highest weight. The second method was to calculate a sentence distance. After several tries, we settled on using cosine distance. Unfortunately, we did not include syn-

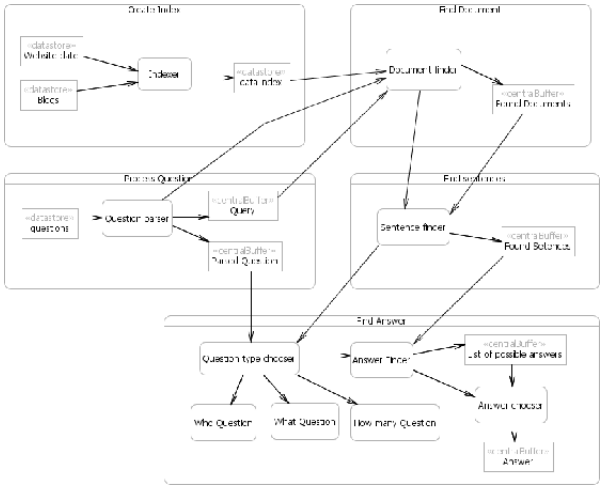
onyms for cosine distance, which meant we would lose the synonym information. We wanted to use both, so we combined the two methods into one number. Once each sentence has a weight or score, we look at the top sentences. The number of sentences can vary, and was set using a parameter. We planned to tweak this number was a last step.

3.2.3 Look for answer

Now that we have a list of sentences that may contain the answers, we'll need to extract the answer from the sentences. For this version, we decided that we would answer with just a factoid; we would not worry about lists. The type of answer we give, depends on the type of question asked. The list of questions can be divided into types. We examine the list of sentences in the found list, and attempt to extract an answer depending on the question type. How we do this for each question type will be discussed below, but for now, let's just assumed that we have extracted a list of answers from the sentences. The code simply counts the number of times each 'answer' appears. The answer with the highest count is the one we use. The most obvious type of question is 'who'. This take a number of forms, not always looking like just a who question. 'Who is ...', 'What person...' are of this type, but there are other forms too, some more difficult to see. Besides who, there are where, when, which, and how questions. We'll discuss each one of the question types. We used a very simple method for determining types. Using lists of sentence beginnings with associated question type, we just check the beginning of the question, if we find a match, we set the it to the question type. If we did not find a type we set it to a default, which is 'who'. The who questions are any question that require a name as an answer. So, for each sentence in our found sentence list, we look for what looks like a proper noun. We look for proper nouns with regular expressions. The code searches for all the proper nouns in the sentences. It then returns the proper noun

that appears the most number of times. The when answers are found by looking for dates. Again, we used regular expressions to look for anything that looks like a date and return a normalized form as the possible answer. The regular expression is pretty complicated, but it still doesn't deal with dates like 'last week', 'next Friday' and so on. Nevertheless, it seems quite good at being able to pick up many dates. The normalized form for dates is formatted as follows: ¡year¿¡era¿ ¡month¿ ¡day¿. ¡era¿ may be BCE or CE. The where answers are treated just like the who questions, except that the answers are checked against a gazetteer. Anything in the gazetteer gets a higher score, after that the names are treated just like those in who. The which answers are found quite differently. The code looks for the category name. For example, a 'which color' question would use the category color. Once we have the category we do a hypernym closure search. The code then searches for those words in the sentences. The words found are then picked out and counted. Unfortunately, this did not seem to work. Although our analysis included other categories, our code did not deal with them.

3.3 Quick flow diagram



4 Conclusions

The layered approach allowed us to make quick changes and also allowed us to test at each layer. We had the ability to work one question at a time,

and to quickly switch out one scheme for another. On the other hand, even a layered approach has its limits to flexibility. There were changes we could not make easily, such as trying now completely new approaches such as using ontologies and it was not easy to share information between the layers.

5 Ideas for future

There are several things we would like to investigate for the future. Going along a similar approach to what we currently have, we want to try better parsing and tagging. Better REs may help us to better identify dates, proper

nouns, numbers, etc. Places in the text where calculations regarding dates are possible need to be recognized and handled. One example is using the document date with relative date information within the document. Another example is using the phrase 'next Friday'. So far, we have made no real attempt to deal with grammar and grammatical elements. Approaches using lemmatization and basic elements seem to hold some promise. Along these same lines, we need to look at handling negation and tenses. We'd also like to look at some very different options like ontologies, datalogs and semantic spaces.