Answer Extraction using Language Models and Data-Mining

Wei Li
Department of Computer Science
University of Massachusetts, Amherst, MA 01003
weili@cs.umass.edu

Abstract

In this report, I present a combined approach of language modeling and an automatically built answer model for finding answers from the relevant documents. We make use of a data-mining technique, Snowball, to collect patterns for the answer model. It only needs a few samples to start with and the patterns are automatically evaluated. As shown by the test results, this model outperforms our current heuristic approach in the QuASM system and the answer model is proven to be helpful.

1. Introduction

The goal of question answering is to provide users with short phrases that explicitly answer their questions. So unlike document retrieval, which returns a list of relevant documents, a QA system needs to perform an additional task called answer extraction. An easy solution to this problem is to do a secondary retrieval on the returned documents. However, the success of traditional IR techniques usually relies on the similarity between the word distributions of queries and relevant documents, which is not always true for questions and answers.

The NLP and IE communities have explored alternative techniques for answer extraction. In our current QA system QuASM, a question is classified according to the type of answer it is seeking, and entity phrases are recognized in the documents. Then we score the answers in a heuristic way, which considers the match between entity types and the question class, as well as context information.

In this report, I present an answer extraction framework that combines both IR and NLP/IE techniques. We start with *language modeling* [1], a statistical approach that has gained more and more successes in the IR area. The basic idea is to build one language model for each answer candidate[1], and rank them according to their probabilities to generate the question.

At the same time, we also want to use the question class information, which has not been utilized by language modeling. Instead of just comparing question class with entity types as QuASM does, we build an *answer model* for every class of questions to capture all kinds of answer patterns.

The intuition behind the answer models is that questions often ask about binary relations. For example, in the TREC-10 evaluation, most "location" questions ask about the following three relations:

<organization, headquarter>,
<person, hometown> and
<country, capital>.

Then an answer should be a piece of text that expresses the corresponding relation. So once we have a collection of text patterns for those relations, we could evaluate answer candidates by comparing them against those patterns.

To automatically collect answer patterns, we borrow a semi-supervised learning technique, Snowball [2], from the data-mining area. We

[1]The current requirement for answer extraction is to restrict the result within a certain length of text, i.e., approximately two sentences. For the rest of the report, the term "answer candidate" is used to refer to any two consecutive sentences.

use weighted vectors to represent the patterns, which allow us to define the degree of match between patterns and answer candidates, and furthermore, to adjust the pattern generality by changing the matching threshold.

The rest of this report is organized as follows: in section 2, I will review some related work; section 3 describes our approach, including the implementation of language modeling for answer extraction, the algorithm for building the answer models, and how to use the answer models and combine them with the language models; section 4 presents evaluation results; section 5 is a discussion about our method and section 6 concludes the report.

2. Related Work

People have long noticed the surface string mismatch between the question formulation and the string containing its answer. Various directions were explored to solve this problem. Berger et al. [3] propose a machine learning approach to answer extraction, which makes use of a training set of answered questions. The learning process is designed to capture the relation between questions and answers with a statistical model. Four strategies have been examined. Adaptive TFIDF is an extension to the traditional tf-idf algorithm, which adjusts the idf weight of each word to maximize the accuracy of answering training questions. The second method, Automatic Query Expansion, learns the correlation between question terms and answer terms from the training data, based on their mutual information. Then the system could automatically expand a query with its related answer words. These two models can be viewed as variants to existing document retrieval techniques, while the other two are quite different. The Statistical Translation Model explicitly addresses the lexical chasm between questions and answers, and treats the answer extraction problem as bilingual text translation. A stochastic matrix is learned to capture the co-occurrences between words in questions and words in answers. The fourth method, Latent Variable Model, attempts to cluster a question and its answer based on their underlying topic. While these machine learning techniques improve the accuracy for finding correct answers, all of them require a large set of training data, which is usually not available for various QA tasks.

Brill et al. [4][5] search the answers on the Web instead of a single, small dataset. The abundance and variation of the data provide better chances to find answers that match the questions. Given a question, they generate multiple queries and send them to a search engine. Then the retrieved summaries are processed to extract answers and score them based on several factors. In the case that it is required to find answers within a given data source, such as the TREC evaluation, they just locate the answers in the dataset using some projection schemes.

Most other systems make use of linguistic resources or NLP/IE/KR techniques such as dictionaries, WordNet, named entity extraction, POS tagging, etc [6][7][8][9][10][11][12][13]. Predictive Annotation (PA), as described in the system GuruQA [12][11], is a technique that applies named entity tagging for answer extraction. The basic idea is to recognize what type of entity the question is looking for and then use it as a guide for the answer. While this approach being explored in many QA systems, PA has implemented it in a different way. Instead of extracting answers from the documents retrieved by a traditional IR system, PA modifies the retrieval process directly. It works as follows. An NE tagger is used on the documents and the entity types (referred to as QA-tokens) are indexed as well as the entity words. Then when a question comes, it will be

classified and converted into a query that includes the corresponding QA-token for the question class. This query is forwarded to the search engine. It is easy to see that the match between QA-tokens automatically contributes to the score of a passage and eventually affects the ranking of answer candidates.

Breck et al. [13] use knowledge representation and natural language processing techniques for answer extraction, treating it as finding variable bindings that satisfy a logical format of questions. After the relevant documents are retrieved using traditional IR methods, they are then processed to dynamically generate a knowledge base. Questions are thus converted into KR queries that are used to find answers in the knowledge base. Note that some of the predicates used for document representation correspond to various types of named entities and are generated using an NE tagger.

Srihari and Li [10] have studied IE techniques in different levels for the QA task. They make use of a natural language shallow parser to capture the structural information in questions in order to classify them correctly. Note that they have a larger set of question categories than typically used. For example, there are several sub-types for the “location” class, such as city, country, mountain, river, and so on. In response, they require a more specific named entity tagger that could recognize the above entities. This extended NE tag set matches the expected answer and entities in the documents more accurately, and also provides a better foundation for the next level of IE application, i.e., defining relationships between entities. What they propose is to extract all kinds of relationships for the entities and use them directly as answers when questions ask about such relations. For example, for a “person” entity, the interesting relationships include his name, title, age, gender, birth time, birthplace and so on. This can be viewed as an extension to the conventional MUC relation extraction task in the sense that it covers more types of relations and thus a wide range of questions.

Instead of constructing a large list of entities and their relationships from the documents, another approach tries to maintain a set of answer patterns to capture such relations and applies them to the documents when questions are given. The success of Soubbotin et al. [14] on the TREC-10 evaluation relies much on a large set of such patterns, which are manually defined and very extensive. Actually, this kind of patterns has also been used in other QA systems to some extent [15][16]. Let us revisit Brill et al.’s system [4][5]. When formulating queries from a question, they apply a set of rewriting rules, which essentially correspond to the expected answer contexts. The number of patterns used here is far less than the ones used by Soubbotin, but they still work quite well when searching the Web, where even several simple patterns are very likely to hit the answers.

Another example would be MURAX [6], a QA system that searches answers in an online encyclopedia and uses some simple heuristic patterns to help answer extraction. One family of patterns is defined based on the observation that for some questions, especially for “what” and “which” questions, the target is explicitly specified and the answer should be an instance of it. Suppose a question starts with “What river …?”. Then we know that its answer is a river instance. So MURAX applies a set of patterns indicating this “instance of” relation to answer candidates and gives high credits to those who match the relation with the target phrase. Another type of patterns deals with the situation when target terms are unrecognizable. MURAX then tries to identify the question term, as demonstrated by the underlined part

in question “Who killed Abraham Lincoln?”. In this case, the answer should be related to the question term by the verb. So MURAX simply matches the verb in answer candidates with various patterns corresponding to active and passive forms. Actually, this idea has been extended by following researches, which no longer require the occurrence of the same verb to capture the relation between question term and answer term.

Given the effectiveness of answer patterns, the remaining question is how to obtain them. All the above systems use handcrafted methods, while an alternative is to automatically learn the patterns. In the IE area, people have been studying a similar problem, i.e., how to induce information extraction rules automatically. A popular approach is to apply machine learning techniques to obtain the rules from annotated training data. For example, CRYSTAL [17] is a system that builds a dictionary of concept node definitions and applies them to unseen texts to extract new information. The concept nodes represent the information that users are interested in, and the definitions describe their context restrictions such as sentence structures and occurrences of certain words. It requires a set of parsed, annotated sentences to generate such rules.

RAPIER [18] makes use of a simpler pattern representation that does not include syntactic information. So there is no need to parse the texts. They use some more robust techniques such as POS tagging and WordNet to construct the rules. The learning algorithm is inspired by inductive logic programming and it generates patterns by gradually dropping constraints to cluster the more specific patterns into more general ones.

Craven et al. [19] present a machine learning approach to construct a knowledge base from the Web documents, which includes relations as well as entities. One assumption of this task is that relations among different entities are usually reflected by the link structure of the Web pages, which then becomes an important component of the resulting rules. To achieve this, the training data is composed of not only individual labeled pages but also the relations among them. The learning algorithm employs a first-order representation for the rules and searches them in a general-to-specific way.

As we can see, all of the above methods need a large manually tagged training set in order to obtain reliable patterns. And the rules learned from one dataset are usually not transferable to other datasets. So for different applications, we need different training data. To avoid the large amount of manual labor involved in the training procedure, people try to reduce the required label information. An example would be AutoSlog-TS, as introduced by Riloff [20]. The basic idea is to exhaustively generate an extraction pattern for each noun phrase in the training set and then evaluate all of them. The evaluation method assumes that the training corpus can be divided into two sets: relevant and non-relevant, with regard to a particular domain. Then the relevance rate for a pattern could be calculated as the number of times it being activated in a relevant document divided by the total number of times activated by any document in the corpus. This is a major factor in the evaluation criterion. As we can see, the manual effort has been reduced from tagging the training set to differentiating relevant and non-relevant documents. However, this still requires going through every document in the training corpus to decide its relevancy.

The difficulty to obtain labeled data is not unique to our problem. In many applications, it is easy to get a large number of unlabeled examples, but expensive to get labeled ones.

So the machine learning community has been studying semi-supervised learning techniques, which take advantages of abundant unlabeled data to reduce the need for labeled data. One approach is to apply EM to generative models, treating the labels of unlabeled data as missing values. Nigam et al. [21] propose a combined model of EM and a naïve Bayes classifier for text classification. The basic algorithm trains a classifier using the initial labeled documents. Then it probabilistically labels the unlabeled documents and trains a new classifier based on all documents and their labels. This process is repeated until convergence. The assumption behind this algorithm is that the documents are generated by a mixture model and each class corresponds to a mixture component. To deal with the cases that the assumption is violated, they further propose two extensions: assigning different weights to the labeled and unlabeled data, and modeling each class with multiple mixture components.

A widely used technique for semi-supervised learning is bootstrapping. The basic idea is to start with a very small set of labeled instances and iteratively enlarges it by predicting labels for unlabeled examples. There are many ways to decide which unlabeled instances should be added. A typical example of bootstrapping is Co-Training. It applies to the datasets whose features could be naturally divided into two sets and each of them is sufficient to learn the classifier given enough labeled examples. As there is only a small amount of labeled data, two classifiers are learned independently using the two feature sets. Then they incrementally label unlabeled examples, and at each round, the newly labeled instances will be added to the other classifier's training set. Co-Training was first proposed by Blum and Mitchell [22] for web page classification. Then it has been applied to various text processing tasks, such as statistical parsing [23], statistical machine translation [24] and reference resolution [25].

Bootstrapping has also shown successes in IE area. For example, Riloff et al. [26] present a multilevel mutual bootstrapping technique that builds a semantic lexicon and a dictionary of extraction patterns at the same time. Starting with a handful of seed semantic entities, they alternatively select the best pattern and use it to extract new entities, which will be used to generate the next pattern. The criterion used to evaluate patterns is composed of two factors. One of them is the ratio for a pattern to hit a lexicon entry and the other is the number of such hits.

Another application of bootstrapping methods is to extract relation pairs from text documents, such as DIPRE [27] and Snowball – the one we are using for answer pattern collection. As we will see later, the basic ideas of these two systems are very similar to Riloff's approach. However, as dealing with different tasks, they differ in many implementation aspects such as pattern representation and pattern generation. Bootstrapping has also been applied to answer extraction for the QA task. An example is the system proposed by Ravichandran and Hovy [28]. A comprehensive comparison between this approach and our model will be presented in the discussion section.

3. System Description

3.1. Language Models

A language model is a probability distribution over a vocabulary of words. Given a language model, we can calculate its probability to generate a piece of text as follows:

$$P(w_1,...,w_n) = P(w_1)P(w_2 \mid w_1)..P(w_n \mid w_1,...,w_{n-1})$$

The simplest model is the unigram model, which assumes independence between words. So the above formula becomes:

$$P(w_1,...,w_n) = P(w_1)P(w_2)..P(w_n)$$

To make use of a language model, we usually need to first estimate it from some sample data. The most common method for estimation is the Maximum-Likelihood-Estimation, which is actually counting the frequencies. To avoid overfitting, we need to use some smoothing techniques to make it more general.

As for our answer extraction task, we build one unigram language model for each answer candidate and then calculate its probability to generate the question. Since we are estimating the language model from just two sentences, we have a severe problem of sparse data. So adequate smoothing becomes critical in this case. We have used the whole set of retrieved documents as a background model to smooth the probabilities estimated from the answer candidates:

$$P(w) = \lambda P_1(w) + (1-\lambda)P_2(w)$$

Here $P_1(w)$ is the probability estimated from the answer candidate, $P_2(w)$ is the probability from the background model and

$$\lambda = d / (d + \text{constant}),$$

where d is the length of the answer candidate and the constant is determined empirically.

3. 2. Answer Models

An answer model is a collection of patterns to express several binary relations pertaining to a class of questions. In order to build this model, we actually build one sub-collection for each relation and then merge them together. The method we used is Snowball, a bootstrapping approach that simultaneously extracts relation pairs and patterns from text documents. The algorithm is sketched below. To better explain the procedure for pattern collection, I use the relation of <organization, headquarter> as an example.

3.2.1 Seeds

The first step is to find several samples. One advantage of Snowball is that we don't need a large number of training data. Only five pairs are used as the initial seeds:

- <Microsoft, Redmond>
- <Boeing, Seattle>
- <IBM, Armonk>
- <Intel, Santa Clara>
- <Exxon, Irving>

3.2.2 Occurrences

Then Snowball finds all occurrences of every seed. An occurrence of a relation tuple refers to a sentence that contains the two entities of the pair. Consider the following sentence:

"Microsoft is based in Redmond."

It is an occurrence of <Microsoft, Redmond>.

Each occurrence is divided into five parts:

left + entity-1 + middle + entity-2 + right

Entity-1 and entity-2 are the two components of the tuple and the other three are the strings surrounding them. Note that we also consider punctuations as part of the strings since they were proven to be helpful in the evaluation of Snowball. There are restrictions on the lengths of strings. If the middle part is longer than a certain length, which is 10 tokens in practice, the occurrence is ignored. This is because we don't want to extract too complicated patterns. Furthermore, we limit the left and right parts to a 5-token window surrounding the entity pair. The next step is to convert the context strings into weighted vectors. It will be later discussed that our patterns are represented in the same way. This representation is more flexible to decide if patterns or occurrences match each other. We could define the degree of match between patterns and set a threshold to show how close we require two matching patterns to be. In this way, we won't miss a new tuple even if its context is slight different from our patterns. This flexibility cannot be

achieved by using text strings since it requires exact string match for two patterns to match each other. The weight of a token t in a vector is defined in terms of the number of times it occurs in the corresponding string s (denoted by C(t)):

$$W(t \mid s) = C(t) / \sqrt{\sum_{t' \in s} C^2(t')}$$

This roughly shows how important this token is in this context. As well as tokens, the three vectors also have weights associated with them to show their individual importance. In practice, we set $W_L = W_R = 0.2$ and $W_M = 0.6$. This is because the middle part is generally more useful than the other two to decide the relationship. To distinguish between the two possible orders for the two entities, we use another indicator that is set to be true if the first entity occurs before the second. So the above example occurrence is represented as follows:

order = "true"
left = ""
middle = "<is:0.58>, <based:0.58>, <in:0.58>"
right = "<.:1.0>"

3.2.3 Pattern Generation

This step is to cluster similar occurrences and then calculate their centroids to form patterns. We define the similarity between occurrences or patterns in terms of the inner products of their weighed vectors. The score is calculated as follows:

If p_1.*order* != p_2.*order*, then Sim(p_1, p_2) = 0;
Else Sim(p_1, p_2) =
W_L * InnerProduct(p_1.*left*, p_2.*left*)
+ W_M * InnerProduct(p_1.*middle*, p_2.*middle*)
+ W_R * InnerProduct(p_1.*right*, p_2.*right*)

We also define the similarity between two clusters as the similarity between the centroids. At the beginning of the clustering algorithm, every occurrence is in one individual cluster. We then repeatedly merge similar ones. Let C(k) denote the set of clusters at iteration k, the following rules are used to get C(k+1):

C(k+1) ← empty set;
For each cluster C in C(k):
Let C' be its most similar cluster in C(k+1);
If Sim(C, C') >= threshold, merge C into C';
Else add C to C(k+1);

The similarity threshold is set to be 0.6, and the algorithm repeats until no merge is done. Then we extract all the centroids as patterns.

3.2.4 Exploring New Tuples

To find new tuples using the patterns, the first problem we need to address is to identify the potential tuples, i.e., entities of the particular types. So we apply a named entity tagger on the documents to recognize entities of interest. As for <organization, headquarter> relation, we construct three weighted vectors for every pair of "organization" and "location" entities that occur in one sentence, the same way as we deal with occurrences in 3.2.2. Then the candidate occurrence is compared to extracted patterns. If it is similar enough to one of them as the threshold equals 0.6, we will extract the <organization, location> pair as a new tuple. Note that we might get one pair using various patterns, possibly with different match scores. We keep all those information because they will help us decide which tuples should be saved and which should be removed.

One important feature of Snowball is that the patterns and new tuples can be automatically evaluated. Along with the flexibility of our pattern matching scheme comes the risk of getting unreliable patterns and tuples. And the erroneous patterns or tuples in early iterations will accumulate and lead the future collection

to a wrong direction. So we need a method to evaluate patterns and tuples, and only keep the high-quality ones. The basic idea is that if a tuple is in the seed set of current iteration, we then believe it is valid and can be used to judge the newly discovered tuples. Here we assume that the first entity is the key for the relation, i.e., no two tuples share the same first entity. For example, <Microsoft, Redmond> is used for the first iteration of <organization, headquarter> collection as a seed, i.e., a valid tuple. Then any <Microsoft, other location> pair we find is considered invalid. This gives us a criterion to judge whether a tuple is valid or not, and furthermore, to score the patterns according to the qualities of the tuples they discover. For each pattern, we define its *belief* as:

$$\text{belief}(p) = \text{positive} / (\text{positive} + \text{negative})$$

Here, *positive* is the number of valid tuples discovered by this pattern and *negative* is the number of invalid tuples. So at the same time that new tuples are extracted, the beliefs of our patterns are also updated.

The above method can only tell which tuples are valid or invalid, but cannot evaluate other tuples that are not seeds and don't share the same first entity with any seed. In this case, we define a tuple's *belief* based on the patterns that discover it:

$$\text{belief}(t) = 1 - \Pi_p(1 - \text{belief}(p) * \text{Sim}(t, p))$$

A tuple will get a high score if it is supported by reliable patterns with high confidences. Naturally, we set the belief to be 1 for a valid tuple and 0 for an invalid tuple.

After the new tuples are evaluated, we can select some good ones to merge to the seed set and start the next iteration. The threshold for the <organization, headquarter> relation is 0.8. It may vary among different relations. Usually we run 2 or 3 iterations to get enough patterns.

3.2.4 Parameter Summarization

As we can see, there are many parameters that need to be empirically set. And I summarize our choices in the following table:

Max Length		Weight		Threshold	
L_L	5	W_L	0.2	T_1	0.6
L_R	5	W_R	0.2	T_2	0.6
L_M	10	W_M	0.6	T_3	0.8

There are three thresholds. T_1 is for clustering similar occurrences, T_2 is used to extract new tuples with collected patterns and T_3 is the score threshold for selecting good tuples. Note that T_3 is not constant and the number listed above is for the <organization, headquarter> relation.

3.3. Combining Two Models

For language modeling, we have a natural way to measure the probability for a candidate C to be the answer to question Q as:

$$P_L(C|Q) \propto P(Q|C),$$

where P(Q|C) is the probability that question Q is generated from the underlying language model of candidate C.

As for the answer model, we should define the probability for a candidate to be the answer in terms of its similarity to the corresponding answer model. Let T(C) be the set of tuples discovered from candidate C using patterns in the question's answer model. Then it is further restricted to the tuples whose first part appears in the question. Only the tuples satisfying this restriction could be the target of the question and thus affect our belief in the candidate. Then we can define

$$P_A(C|Q) \propto \sum_{t \in T(C)} \text{belief}(t)$$

To combine the two models together, we use linear combination, where

$$P(C|Q) = \alpha * P_L(C|Q) + (1-\alpha) * P_A(C|Q)$$

It is the final score used to rank the answer

candidates. Before combination, we need to normalize the results from the two models to the same scale. The value of α is determined empirically.

4. Evaluation

The evaluation has two purposes: the first one is to see if language modeling is adequate for answer extraction, and the other is to test the effectiveness of answer models.

For the first part, we conduct an experiment on the TREC-10 dataset, which contains 687 questions and 11 classes. The performance is measured by MRR, the mean-reciprocal-rank. The table below shows a comparison between QuASM's heuristic approach and language modeling on individual classes and average performance:

Class	# of Q	Heuristic	LM
A	53	0.283	0.232
B	14	0.238	0.179
D	9	0.111	0.204
F	93	0.543	0.598
L	109	0.553	0.446
O	39	0.489	0.685
P	113	0.593	0.582
R	9	0.278	0.167
T	73	0.305	0.288
W	16	0.325	0.469
X	168	0.278	0.407
Avg.	**696**	**0.421**	**0.448**

Although I have listed the experiment results of language modeling for different classes, we know that the class information is not included yet. But even with this restriction, the average performance of language modeling is as good as the heuristic model. So it is reasonable to believe that language modeling is an adequate framework for answer extraction and we can further improve its performance by using the class information.

To test the effectiveness of the answer models, we choose the "location" (L) questions to do the experiments. Based on our previous results, the heuristic model performs much better than the language models for this class. So we are interested to see if adding the answer models helps.

The database used for building the answer models is made up of 16 TREC corpora. For each document, we extract the TEXT part, segment it into sentences and mark out the entities with IdentiFinder [29].

The answer model for the "location" questions is composed of patterns for the following three relations:

- <organization, headquarter>
- <person, hometown>
- <country, capital>

Some of the patterns collected along with their beliefs are listed below:

<organization, headquarter>
order = "false"
left = "<In:1.0>"
middle = "<a:0.58> <for:0.55> <spokesman:0.51>
<spokeswoman:0.05>"
right = "<the: 0.26> <said:0.26> <company:0.13>"
belief = 0.9

order = "true"
left = ""
middle = "<is:0.58> <in:0.58> <based:0.58>"
right = "<.:0.86>"
belief = 0.85

order = "true"
left = ""
middle = "<is:0.51><headquartered:0.51><in:0.51>"
right = "<.:1.0>"
belief = 0.8

<person, hometown>

order = “false”

left = “<the:0.49> <in:0.33>”

middle = “<town:0.69> <of:0.71>”

right = “<.:0.47>”

belief = 0.9

order = “true”

left = “”

middle = “<born:0.60> <in:0.60> <was:0.46>”

right = “”

belief = 0.86

<country, capital>

order = “false”

left = “<In:1.0>”

middle = “<a:0.46> <said:0.46> <official:0.46>”

right = “<to:0.36> <was:0.27>”

belief = 1.0

order = “false”

left = “<in:0.40>”

middle = “<capital:0.71> <of:0.71>”

right = “<the:0.14>”

belief = 0.9

order = “true”

left = “<in:0.20>”

middle = “<capital:0.5> <’:0.5> <s:0.5> <of:0.5>”

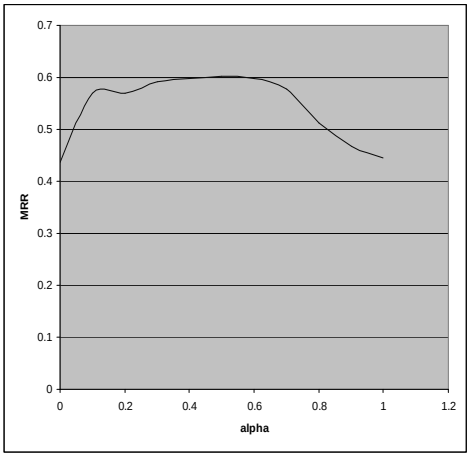
right = “”

belief = 0.76

The individual performances of these relations are as follows:

Relation	# of Q	LM	Combined
<O, H>	21	0.325	0.787
<P, H>	12	0.396	0.667
<C, C>	50	0.545	0.663

The performance of the combined model on the whole “location” set is presented below. Note that there are 26 questions that do not belong to any of the three relations and will not benefit from the answer models.



As we can see, the best result is achieved when $\alpha = 0.5$ and MRR = 0.603 for this value. It is a significant improvement over language modeling only (0.446), and it is also better than the heuristic model (0.553).

5. Discussion

The basic idea of Snowball was inherited from an existing technique called DIPRE [22]: Dual Iterative Pattern Relation Expansion. These two approaches share the same bootstrapping procedure to extract structured relations. They both need very minimal samples to start with and then iteratively discover relation patterns and tuples. As DIPRE searches the World Wide Web, Snowball restricts itself to fixed collections of text documents and makes two major modifications.

The first difference is pattern representation. In DIPRE, a pattern is defined as a five-tuple <order, urlprefix, prefix, middle, suffix>. All of them are kept in Snowball except urlprefix because it is no longer available when we are not searching the World Wide Web. And the three context items prefix, middle and suffix, which are strings in DIPRE, are replaced with weighted vectors in Snowball.

As patterns are represented in different ways,

the operations upon them, such as how to generate patterns from occurrences and how to decide if a pair matches a pattern, are also different. Since DIPRE uses strings for the contextual information, it is simpler to define those operations. To generate patterns, DIPRE divides occurrences into groups based on their orders and middles. Then for each group, one pattern is defined as follows: order and middle are the same as the occurrences in the group, urlprefix is the longest matching prefix of all urls, prefix is the longest matching suffix of all prefixes and suffix is the longest matching prefix of all suffixes. Given the patterns, the problem of matching relation pairs against them is a binary decision of string matching. More specifically, it requires the pair to occur in a document whose URL has the same prefix as the pattern's urlprefix and the context matches its prefix, middle and suffix. This has been implemented using regular expressions. So in DIPRE, a pair either matches a pattern or does not match. There is no state in the middle. But Snowball has introduced the idea of matching degree, which allows us to adjust the patterns' generality. As we can see, DIPRE is very similar to a special case of Snowball, where its matching threshold is set to be 1.

Another modification made by Snowball is to incorporate a method that could automatically evaluate the patterns based on their precisions in finding valid tuples. Furthermore, the newly discovered tuples can also be evaluated and only the good ones are kept as seeds for the next iteration. In DIPRE, there is no such evaluation. All tuples are considered valid. However, it does have some restrictions on the patterns to prevent them from being overly general. For each pattern, there are two factors that affect whether it will be kept. The first one is the specificity (denoted by s), which is calculated as the product of the lengths of the pattern's four strings. The other is the number of seeds with occurrences supporting the pattern (denoted by n). DIPRE gives higher credits to patterns that are more specific and have more supporting seeds. So it requires that $s \times n >$ threshold and $n > 1$. As the threshold is greater than zero, the first requirement implies $s > 0$, i.e., no pattern with empty strings will be accepted. The second requirement excludes those one-seed-based patterns because they are not very reliable. Actually, this idea of considering the number of supporting seeds when evaluating patterns has been explored in our own experiment and I will discuss it in details in the following section.

Besides the two major modifications discussed above, Snowball has also made use of some IE techniques to improve the performance. For example, DIPRE requires users to provide a set of regular expressions to find new tuple candidates. This is not very accurate and will add more noises into the system. So Snowball chooses to apply a named entity tagger and only considers the two types of entities that constitute the relation. Another usage of the named entity tagger in our own experiment is at the finding-patterns stage, though Snowball doesn't use it this way. Consider the following example:

1. … spokesman Jim Desler said in …
2. … spokesman Tom Ryan said in …

The only difference between the above two occurrences is the name of the spokesman. More generally, we do not expect to see the same person name in this context for different tuples. However, they should be viewed as the same pattern. So we replace those specific names with their entity type PERSON and thus collapse them together.

As we can see from the discussion so far, one important feature of DIPRE is that it only generates very specific patterns in order to improve the precision of the collected relation

pairs. So the contexts are represented with non-empty strings and patterns extracted from different websites are generally not merged. However, the large amount of data on the Web allows DIPRE to retrieve a large set of tuples even with those specific patterns. In one of the experiments, it starts with five samples of the <book, author> relation and finally obtains over 15,000 pairs. According to the evaluation, 19 of 20 randomly picked pairs are valid. This is quite impressive, but it doesn't work so well when we explore a similar approach for the <organization, headquarter> relation on the 16 TREC collections. In our own experiments of DIPRE, we exclude urlprefix from patterns and keep the other four items. At first, we could barely find any pattern that satisfies the specificity requirements. The patterns with at least two supporting seeds usually have empty strings. Then we try to accept patterns that break at most one restriction, but result in many invalid tuples even in early iterations. Our experience shows that the success of DIPRE relies much on a huge amount of data and it works best in an environment like the Web.

On the other hand, Snowball works much better than DIPRE on a fixed data source. It utilizes more general patterns that allow us to get satisfying results in spite of data shortage. However, there are still some difficulties when we apply Snowball to construct our answer models. The first problem is how to define the weighted vectors. Our initial attempt is to use the token frequency as its weight, but then we realize that this definition has a bias toward shorter patterns in terms of similarity. Take the following two cases as an example:

Case 1:

P1 = "<x, 0.5>, <y, 0.5>"

P2 = "<x, 0.5>, <z, 0.5>"

Sim(P1, P2) = 0.25

Case 2:

P1 = "<a, 0.2>, <b, 0.2>, <c, 0.2>, <d, 0.2>, <e, 0.2>"

P2 = "<a, 0.2>, <b, 0.2>, <c, 0.2>, <d, 0.2>, <e, 0.2>"

Sim(P1, P2) = 0.2

Intuitively, the two patterns in Case 2 should have higher similarity score because they are identical and the ones in Case 1 are not. But here we get the opposite result. So we turn to another definition for the weighted vectors in order to solve this problem. Currently, we use the following formula to calculate a token's weight:

$$W(t \mid s) = C(t) / \sqrt{\sum_{t' \in s} C^2(t')},$$

where C(t) is the number of occurrences of token t in string s. Given this definition, the above example becomes:

Case 1:

P1 = "<x, 0.707>, <y, 0.707>"

P2 = "<x, 0.707>, <z, 0.707>"

Sim(P1, P2) = 0.5

Case 2:

P1 = "<a, 0.447>, <b, 0.447>, <c, 0.447>, <d, 0.447>, <e, 0.447>"

P2 = "<a, 0.447>, <b, 0.447>, <c, 0.447>, <d, 0.447>, <e, 0.447>"

Sim(P1, P2) = 1

Now we get the desirable result. Actually, this definition guarantees that if two patterns are identical, their similarity will always be 1 no matter what their lengths are.

Another difficulty with Snowball involves the pattern evaluation method. As we know, the success of our approach relies on effectively recognizing valid and invalid tuples. But the literal comparison might cause trouble. For example, as <Intel, Santa Clara> is a seed, we treat <Intel, other location> as invalid tuples. However, among those other locations, some of them might be the variants of "Santa Clara",

such as “Santa Clara, CA”. The performance will be significantly affected if we are unable to tell that this is also a valid tuple. Currently we are using some simple heuristics to achieve this. For instance, we consider two locations the same if one of them is the prefix of the other. But they cannot deal with all situations. In the future, we will need more sophisticated methods to solve this problem.

As I mentioned before, we have considered various options for scoring patterns, some of which involve the number of supporting seeds for the pattern. We define three belief metrics as follows:

belief(p) = positive / (positive + negative)

belief_1(p) = belief(p) × $\log_2$(positive)

belief_2(p) = belief_1(p) × $\log_2$(# of supporters)

As the first one focuses on the precision of the pattern, the other two also account for its coverage. However, introducing the number of positive matches increases the belief of many overly general patterns and thus depreciates the precision. So we choose the first metric to construct our answer models.

Another factor that can affect the quality of the patterns is how to generate them from occurrences. Currently, we are using a greedy clustering algorithm, in which the merge of two similar clusters is permanent. As most greedy algorithms, the order of merges has a great impact on the final clustering result. However, we have not paid special attention to this issue and set it completely based on the order in which the occurrences are identified in the documents. So improvement might be achieved if we change the ordering to reflect our confidences of merges. For example, we could merge the two clusters that are most similar and repeat until the similarity is below a threshold.

Finally, I want to discuss a little bit about how we combine the answer models with language models. Besides the linear combination, as described in Section 3, we have tried another approach, which is an integrated probabilistic model. The basic idea is as follows. For any given question Q with class C, we want to rank answer candidate A based on P(A|Q,C), which is proportional to

$$P(A,Q \mid C) = P(Q \mid A,C)P(A \mid C)$$

Assuming P(Q|A,C) = P(Q|A), the first factor is naturally captured in language models as the probability of generating the question by the answer candidate. So the remaining task is to use the answer models to calculate the prior P(A|C). This probability is independent from the particular question Q and only shows our belief for the candidate to be an answer to a class of questions.

As we can see, the answer models are used in different ways in the two approaches we have explored. However, they both need to deal with the same problem, i.e., we cannot classify questions as specific as relations using our currently question classifier. For example, the “location” category involves three relations. While the two questions “Where was George Bush born?” and “Where is Microsoft?” are both “location” questions, they are interested in different relations and thus expect totally different answers. So we need to figure out which pattern set should be used. As for the linear combination method, we do not answer this question directly, but put together all relations pertaining to one class and treat them as one answer model. Obviously, this is very inaccurate. For instance, since the candidate “Microsoft is based on Redmond.” matches one pattern of the <organization, headquarter> relation very well, which is in the “location” answer model, it will get a high score for all “location” questions, including “Where was

George Bush born?". To address this problem, we furthermore require a pattern to affect our belief in a candidate only if it discovers a tuple whose first entity appears in the question. As for the above example, the tuple extracted by the pattern is <Microsoft, Redmond>, whose first entity "Microsoft" is not in the question. So this pattern will just be ignored. However, it will contribute to the question "Where is Microsoft?". Actually, this method does not only reduce errors caused by selecting the wrong relation patterns, but also works well in the situation where the candidate matches the target relation but is not a valid answer. An example would be "Bill Clinton was born in Hope, Arkansas." to the George Bush question. On the other hand, we are not supposed to touch the content of questions in order to calculate the prior P(A|C) in the integrated probabilistic model. Therefore we take another approach to select the pattern set based on the observation that different relations in one class usually have different types for their first entities. This is true for the "location" question, in which the three relations are asking about <organization, location>, <person, location> and <location, location> respectively. So we could guess the target relation of a question by recognizing the type of entities in the question. Apparently, this method works for the two example questions I mentioned before. In the case that there are multiple types of entities in the question, we calculate the expected prior as

$$P(A \mid C) = \sum_{R_i} P(R_i \mid C) P(A \mid R_i),$$

where R_i enumerates all relations in class C and

$$P(R_i \mid C) = \frac{\#\, entities\ \ of\ \ type\ T(R_i)}{\#\, entities}$$

$T(R_i)$ refers to the type of the first entity in R_i. As we can see, the performance of this method depends much on the accuracy of the entity recognizer. Unfortunately, IdentiFinder does not work very well on our test data and thus depreciates the overall performance. Besides, this model could hardly be extended to classes that have relations sharing the same types of first entities. So we finally choose the linear combination method instead. But we are still interested to see how the probabilistic model works when we have a more specific question classifier that could classify questions into relations.

So far, I have discussed several difficulties using relation patterns for answer extraction and possible solutions to them. Some of them are addressed in another approach proposed by Ravichandran and Hovy [23], which is very similar to our method and also shares some features with DIPRE. For each class of questions, this model searches the World Wide Web for occurrences of sample <question term, answer term> pairs and records all substrings containing both terms as patterns. Note that these seeds do not necessarily belong to the same relation set, which means we could use both <Microsoft, Redmond> and <George Bush, New Haven Connecticut> as seeds for the "location" patterns. So the pattern set that is collected for one class of questions might involve multiple relations. After patterns are generated, their precisions are calculated in a similar way as we evaluate our patterns. So they also have the canonicalization problem, i.e., the answer term might have variants and they need to be treated as the same term. While we try to match variants to the original term based on some simple rules at evaluation phase, their solution is to list all possibilities of writing an answer term in advance. Another difference is that when evaluating a pattern based on its precision in finding seed pairs, the one seed that actually obtains the pattern is excluded. This cross-checking method makes the evaluation more reliable.

After patterns are evaluated, they are then used to extract answers, requiring that the first term in the resulting pair match the question term. This is quite similar to the way that our answer models are used in linear combination, but even more accurate if question terms can be well identified. However, its performance is decreased, as there is no restriction on the answer term. For example, while the question is asking for a “location” entity, the model might recommend a non-location phrase as the answer if it matches one of the good patterns. As in our system, we apply an entity tagger to all answer candidates in order to reduce this type of errors.

From the above comparison and the earlier one with DIPRE, we could draw the following conclusions:

1. The tremendous data available on the Web and its great variety makes it easier to collect a large set of specific answer patterns. So it is our future direction to explore the Web for pattern collection. However, as long as we need to find answers in a particular dataset, such as in the TREC evaluation case, we still believe weighted vectors are more appropriate representation than strings. This is because even when collecting patterns from the Web, it is not guaranteed that one of them exactly matches the way that the answer is expressed. So weighted vectors will offer extra flexibility to recognize the answer when it is similar to but not completely the same as the patterns.

2. Pattern evaluation has played an important role to improve performance. To make it even more effective, we need a more sophisticated way to solve the canonicalization problem. Actually, it may not only be applied to the answer terms, but question terms as well. Our current evaluation metric of precision is just one of the various options, and we are going to work on some alternative methods too. For example, once we are able to obtain specific patterns from the Web, we might try again the two metrics I mentioned before, which take into account the coverage of patterns and the number of supporting seeds. Another issue associated with pattern evaluation is how to extend this approach to relations whose first entity does not serve as the key, i.e., different pairs could share the same first entity and they are all valid.

3. When using the collected patterns to extract answers, a good restriction on the resulting pair should be that the first term matches the question term and the second term matches the target entity type. To achieve this, we need a better named entity tagger and a very accurate technique to identify question terms.

6. Conclusion

Question answering differs from information retrieval in that it needs to retrieve specific fact information rather than whole documents. So answer extraction, the process of scanning the relevant documents to find the accurate answer, is an important component of a QA system. In our previous researches, we have found that the class of question is helpful for judging answer candidates, and already made use of it in a heuristic way.

This report has investigated a more principled framework that combines language modeling and answer models. Language modeling is a probabilistic approach that has been widely used for various IR tasks. As for the answer extraction problem, its role is to judge the relevance between answer candidates and the question by their contents. Without using the question class information, language modeling performs as well as the heuristic model. And we expect further improvement by combining the answer models.

An answer model is a collection of patterns to express answers to one class of questions. By introducing such models, we want to capture the difference between answers to different classes of questions. In this report, we have examined a data-mining technique, Snowball, to build answer models. As the experiment for the "location" questions shows, the combined approach of language modeling and answer model significantly improves the performance over language modeling only.

For the future researches, we will try to obtain patterns from the Web. Taking advantage of this tremendous data resource, we believe that our pattern collection process will be more efficient and effective. We are also interested in designing a solution to the canonicalization problem in order to improve pattern qualities. Another direction is to develop a technique for precise recognition of question terms, which could be used along with collected patterns for more accurate answer extraction.

Acknowledgement

This material is based on work supported in part by the Center for Intelligent Information Retrieval and in part by NSF grant #EIA-9983215.

Any opinion, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.

References

[1] J. M. Ponte and W. B. Croft. A Language Modeling Approach to Information Retrieval. In Proceedings of the 21st ACM Conference on Research and Development in Information Retrieval (SIGIR'98), pages 275-281, Melbourne, 1998.

[2] E. Agichtein and L. Gravano. Snowball: Extracting Relations from Large Plaint-Text Collections. In Proceedings of the 5th ACM International Conference on Digital Libraries (DL'00), 2000.

[3] A. Berger, R. Caruana, D. Cohn, D. Freitag and V. Mittal. Bridging the Lexical Chasm: Statistical Approaches to Answer-Finding. In Proceddings of the 23rd ACM Conference on Research and Development in Information Retrieval (SIGIR'00), pages 192-199, 2000.

[4] E. Brill, J. Lin, M. Banko, S. Dumais and A. Ng. Data-Intensive Question Answering. In Proceedings of the Text REtrieval Conference, 2001.

[5] S. Dumais, M. Banko, E. Brill, J. Lin and A. Ng. Web Question Answering: Is More Always Better? In Proceedings of the 25th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2002), Tampere, Finland, 2002.

[6] J. Kupiec. MURAX: A Robust Linguistic Approach For Question Answering Using An On-Line Encyclopedia. In Proceedings of the ACM SIGIR Conference, pages 181-190, 1993.

[7] S. Harabagiu, D. Moldovan, M. Pasca, R. Mihalcea, M. Surdeanu, R. Bunescu, R. Girju, V. Rus and P. Morarescu. FALCON: Boosting Knowledge for Answer Engines. In Proceedings of the ninth Text REtrieval Conference, pages 479-488, 2000.

[8] E. Hovy, U. Hermjakob and C. Lin. The Use of External Knowledge in Factoid QA. In Proceedings of the tenth Text REtrieval Conference, 2001.

[9] E. Hovy, L. Gerber, U. Hermjakob, M. Junk and C. Lin. Question Answering in Webclopedia. In Proceedings of the 9th Text REtrieval Conference, Gaithersburg, MD, Nov. 2000.

[10] R. Srihari and W. Li. Information Extraction Supported Question Answering. In Proceedings of the 8th Text REtrieval Conference, Gaithersburg, MD, Nov. 1999.

[11] J. Prager and E. Brown. One Search Engine or Two for Question-Answering. In

Proceedings of the 9th Text REtrieval conference, 2000.

[12] J. Prager, E. Brown and A. Coden. Question-Answering by Predictive Annotation. In Proceedings of SIGIR'00, Athens, Greece, 2000.

[13] E. Breck, J. Burger, D. House, M. Light and I. Mani. Question Answering from Large Document Collections. In 1999 AAAI Fall Symposium on Question Answering Systems, North Falmouth, MA, 1999.

[14] M. M. Soubbotin and S. M. Soubbotin. Patterns of Potential Answer Expressions as Clues to the Right Answer. In Proceedings of TREC-10 Conference, pages 175-182, 2001.

[15] O. Ferret, B. Grau, M. Hurault-Plantet, G. Illouz, L. Monceaux, I. Robba and A. Vilnat. Finding an answer based on the recognition of the question focus. In Proceedings of the 10th Text REtrieval Conference, page 362, 2001.

[16] G. Lee, J. Seo, S. Lee, H. Jung, B. Cho, C. Lee, B. Kwak, J. Cha, D. Kim, J. An, H. Kim and K. Kim. SiteQ: Engineering High Performance QA System Using Lexico- Semantic Pattern Matching and Shallow NLP. In Proceedings of the 10th Text REtrieval Conference, pages 437-446, Maryland, 2001.

[17] S. Soderland, D. Fisher, J. Aseltine and W. Lehnert. CRYSTAL: Inducing a conceptual dictionary. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, pages 1314-1319, 1995.

[18] M. E. Califf. Relational Learning Techniques for Natural Language Information Extraction. Ph.D. thesis, University of Texas at Austin, August 1998.

[19] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigram and S. Slattery. Learning to Extract Symbolic Knowledge from the World Wide Web. In Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98), pages 509-516, 1998.

[20] E. Riloff. Automatically Generating Extraction Patterns from Untagged Text. In Proceedings of the 13th National Conference on Artificial Intelligence, pages 1044-1049, 1996.

[21] K. Nigam, A. McCallum, S. Thrun and T. Mitchell. Text Classification From Labeled And Unlabeled Documents Using EM. In Proceedings of National Conference on Artificial Intelligence (AAAI), 1998.

[22] A. Blum and T. Mitchell. Combining Labeled and Unlabeled Data with Co-Training. In Conference on Computational Learning Theory 11, 1998.

[23] A. Sarkar. Applying Co-training Methods to Statistical Parsing. In Proceedings of the 2nd NAACL, Pittsburgh, PA, 2001

[24] C. Callison-Burch and M. Osborne. Co-training for Statistical Machine Learning. In Proceedings of the 6th Annual CLUK Research Colloquium, 2003.

[25] C. Mueller, S. Rapp, and M. Strube. Applying Co-Training to Reference Resolution. In Proceedings of ACL 2002, pages 352-359, 2002.

[26] E. Riloff and R. Jones. Learning Dictionaries for Information Extraction by Multi-Level Bootstrapping. In Proceedings of the 16th National Conference on Artificial Intelligence, 1999.

[27] S. Brin. Extracting Patterns and Relations from the World Wide Web. In Proceedings of the International Workshop on the Web and Databases, pages 102-108, Valencia, Spain, 1998.

[28] D. Ravichandran and E. Hovy. Learning Surface Text Patterns for a Question Answering System. In ACL Conference, 2002.

[29] BBN official site about the IdentiFinder: http://www.bbn.com/speech/identifinder.html.