Learning to Admit You're Wrong: Statistical Tools for Evaluating Web QA

Mark Dredze
Computer and Information Sciences Department
University of Pennsylvania
Philadelphia, PA 19104
mdredze@seas.upenn.edu

Krzysztof Czuba
Google, Inc.
New York, NY 10011
kczuba@google.com

Abstract

Web search engines provide specialized results to specific queries, often relying on the output of a QA system. However, targeted answers, while helpful, are embarrassing when wrong. Automated techniques are required to avoid wrong answers and improve system performance. We present the Expected Answer System, a statistical data-driven framework that analyzes the performance of a QA system with the goal of improving system accuracy. Our system is used for wrong answer prediction, missing answer discovery, and question class analysis. An empirical study of a production QA system, one of the first such evaluations presented in the literature, motivates our approach.

1 Introduction

Web search has expanded beyond a general presentation of search results and now offers customized results for numerous query types, such as weather, movies, stocks, etc. One such directed system that appears on most major search engines is Question Answering (QA). Ask, Google, Live Search (MSN) and Yahoo, all display a QA system's answer at the top of the results when appropriate. Pushing answers to the top means increased attention making wrong answers particularly embarrassing. Quality control is very difficult since there are an unbounded number of queries that can trigger the QA system. While recent work has used machine learning techniques for system analysis, we consider a new problem: learning when a QA system is wrong.

Information request search queries received by a QA system can be divided into five classes:

1. Good queries (questions for facts) that the QA system responds to correctly.
2. Good queries answered incorrectly.
3. Good queries that are not answered at all.
4. Bad queries that are answered; by definition, any response to these queries is bad.
5. Bad queries that are not answered.

Good queries are the ones the system developers believe their system should address, and good answers are the ones that the system developers believe are helpful to the users.[1] Understanding a system's performance over these classes is necessary to evaluate and improve the system. Consider a QA system that answers as follows:

- how long is spaceballs? (Answer: 96 minutes)

[1]The efforts to define consistent criteria for research prototypes at the TREC conference show that anything beyond this intuitive definition is difficult to enforce [1].

- who was in spaceballs? (Answer: A 1987 science fiction spoof movie)

Both queries are good, but only the first response is correct. Given this evidence, how will the system perform on two related questions, "how long is the princess bride?" and "who was in the princess bride?" Based on the above behavior, the same system is likely to give a correct answer for the first question and a wrong answer for the second. Previous behavior informs evaluation since each of the queries is of the same type as previously answered queries (movie length and cast), where the query type indicates the expected type of answer.

We present the Expected Answer System, a statistical framework for QA system evaluation that learns system behavior from previous queries organized in the above five classes to aid evaluation and improvement. This paper offers a practical perspective, possibly the first evaluation of a large scale production QA system using actual queries. We outline our framework so that others can build and improve QA systems faster and garner insight about practical methods that perform well.

This paper is structured as follows. First, we describe our Expected Answer System, which captures our intuitions and provides a basis for our applications. We then provide context for our methods by describing our QA system and our evaluation techniques. Next, we discuss the implementation of our methods for our QA system. We then explore several applications, including ways to improve precision and recall as well as general techniques for QA system evaluation.

2 Expected Answer System

We evaluate a QA system as a blackbox with the following properties: given a query (good or bad), the system can either return an answer or not. Each query answer pair can then be labeled by an annotator as either right (good query, good answer) or wrong (bad query, or good query and wrong answer). Unanswered good queries are dealt with separately as discussed below. Given a query answer pair and a label (right or wrong), we can evaluate a system and model expected behavior. As we observed, previous behavior is indicative of future performance so labeled examples can *predict* system performance on unlabeled query answer pairs. However, QA systems are complex; a single model cannot accurately predict a system. We use a divide and conquer strategy, dividing the query space into query classes, then modeling each class independently. A query class is a group of queries that all share a common behavior, namely they all have the same answer type. For example, a class may correspond to the template "How tall is T," which only includes queries that expect a measurement of a person's height as an answer. This breakdown is commonly used for processing queries, such as "Qtargets" [2], creating a set of query types such as *who-question*, *what-question*, *how-question*, [3, 4, 5, 6] or classification of queries by type [7]. Since each query class has a single answer type, we can effectively model the answers for all of the queries in the class together. For example, given our knowledge about the "How tall is T" template based on many labeled query answer pairs from that class, we can correctly predict that the query "How tall is Tom Cruise?" should be answered with a measurement of height.

Prediction models require a representation of each example, typically called features. Formally, we define a mapping function $f : A \rightarrow \mathcal{X}$ that maps a QA answer $a \in A$ to a feature vector $\bar{x} \in \mathcal{X}$, where $\mathcal{X}$ is a feature space. These features allow a classifier to identify common patterns in the correct answers. There are many techniques for defining features for QA answers. AskMSR creates numerous features, such as surface string features (capitalization, the presence of digits, etc.) and handcrafted regular expression patterns [5, 8]. [9] use features from a parser run on the query. Features can also come from type information or the source document. Each feature set can be tailored to a specific system; we describe our features below based on our QA system.

Using a division of queries into query classes and a set of features describing each answer, we present our **Expected Answer System** (EAS) which uses labeled queries and answers to model the answer we expect for a new query, allowing for system evaluation and improvement. EAS analyzes a QA system based on just this information (labeled query answer pairs) and the approach is applicable to most QA systems, which easily fit this paradigm, for example, the AskMSR system [9, 5, 8] or MULDER [10]. Research and production QA systems that share these properties are potential beneficiaries of our approach. EAS models each query class separately by constructing a naïve Bayes classifier for each class using labeled question answer pairs from that query class. Since each class has a single answer type, system evaluation is treated as a binary classification problem. Given

a query and answer, EAS selects the classifier for the corresponding class and classifies the answer as right or wrong. Dividing the query space into classes is analogous to the multi-class classification scenario in machine learning, where a single classifier cannot capture multiple decision boundaries, so a separate classifier is learned for each class.

While we treat a QA system as a blackbox for evaluation, we provide an architectural overview of the QA system we evaluated to give context for our empirical study. We then describe our system evaluation, followed by a description of the EAS's implementation for our system.

2.1 QA System Overview

The QA system used in our evaluation is Google's production web based QA system. It receives millions of queries daily and generally provides high precision answers, covering a range of factoid questions. When our system receives a query, it is analyzed to determine if it contains a request for fact based information, such as "What is Tom Cruise's age?" or "Tom Cruise birthday". Opinion and navigational queries are ignored using simple heuristics. If a query contains an appropriate request, our QA system attempts to provide an answer. The user query is transformed into a request to a large fact repository of open domain knowledge. The repository is structured as objects (or entities) with facts describing the object, such that the object with the name *Tom Cruise* will have an attribute *Birthday* and the value *July 3, 1962*. Both the *Tom Cruise* object and the value of the field *Birthday* may have one or more types associated with them, in this case, *person* (or *actor*) and *date*. A repository request returns the *Tom Cruise* object and the *Birthday* attribute indicating that the value of this attribute may contain an answer. The system ranks this and other potential answers and estimates a confidence score for each one. If the system finds a high confidence answer, it displays the repository fact to the user, e.g. "Tom Cruise - Birthday: July 3, 1962".

The fact repository is similar to QA systems that extract answers from text documents and directly analogous to other open domain systems [5] and to other collections or indexes of possible answers [11, 12, 13]. Recent work has focused on building such collections of knowledge [14]. The repository contains a large sampling of facts extracted automatically from structured information on the web, which is common to a wide variety of sites that contain encyclopedic information about people, movies, books, events, places, products, etc. The repository includes an unbounded number of types based on site evidence (e.g., facts collected from the CIA Factbook site are about places). Type assignment is common to many QA systems both for question and answer types [15, 16]. The most common types for attribute values and objects are of the form *person*, *city*, *movie*, etc, and a small number of regular expressions apply types such as *date* or *number*. The construction, coverage, and accuracy, of our repository are beyond the scope of this paper since our evaluation system treats the QA system and repository as a black box.

A repository allows for a faster response, important in a live system, since answers are preprocessed. Additionally, a repository based approach distinguishes between knowledge discovery and QA, allowing us to focus on QA directly. It results in a highly scalable system that is largely language independent. We focus on English in the remainder of this paper, but both our QA system and the methods presented here have been applied to different languages with little modification.

2.2 System Evaluation

A primary assumption of our approach is that system designers will label some queries for appropriateness (good/bad) and answer correctness (correct/incorrect). While costly, it is necessary; it is unlikely one would launch a new QA service without any measure of quality. As we described above, EAS uses these queries to train a binary classifier for each query class. We label query answer pairs based on our human annotations for binary classification as follows:

- **Positive/Good**: A good query with a correct answer.
- **Negative/Bad**: A bad query with an answer or a good query with a bad answer.
- **Unknown/Unlabeled**: A query with no answer or an unrated answer.

Our (English) evaluation set consists of about 20,000 unique hand rated query and answer pairs (corresponding to a total of about 1.1 million non-unique queries mined from logs), sorted by volume over a given time span. Our evaluations indicate that we answer queries with very high precision,

which is important since a silent lack of answer is preferred to an embarrassing wrong answer. In contrast, it is impossible to get an accurate measure of recall without evaluating a large number of queries not answered by our system, including all queries received by the search engine.

While most QA systems are evaluated based on the TREC QA track dataset [1], we do not find much evidence that these queries are common in our query logs. Additionally, we need multiple queries of the same type to learn behavior e.g., one query about the population of a country is insufficient; we need to have a set that tests different ways users can ask a question.

2.3 Implementation

We now describe our method for creating classes and the features we used for our system. There are many ways to group queries by answer type, including classification [7], general question types [3, 4, 5, 6] or parsing. Additionally, queries with common answers could be group together, such as "Tom Cruise birthday" and "When was Tom Cruise born?" Our system relies on the automatic grouping of queries into these groups. Our initial evaluation used a string matching technique to create templates and assumed that each template corresponded to a single class. However, any of the above methods could be used instead to increase query coverage, find more general classes, or group multiple templates into a single class. The efficacy of each approach depends on the configuration of the QA system.

We resolved a query to a template by replacing any string in the query that matched the object or attribute name from our repository, or synonyms for these fields, that was used to answer the question. The matched tokens were replaced by the "T" token. Since our example query ("Tom Cruise birthday") returns the *Tom Cruise* object, we replace "Tom Cruise" with "T" yielding the query template "T birthday". Additionally, we strip all punctuation from the string, ignore case, and move the "T" token to the end to create larger groupings. This simple extraction method is robust since it could resolve potentially confusing queries, such as "Who stars in the movie who framed roger rabbit?", where the movie name itself is a question. Additionally, we can separate two seemingly similar queries that actually have different answers, such as "what is in T" and "what is T". Despite the simplicity of our initial query class construction, we found that the methods scaled well to new queries not observed in our training dataset. For answer features we relied on structural properties of our repository facts as features. We implemented the following feature rules:

- **Value type**: Type of the answer string (ex. date, city, movie); e.g. "contains=number".
- **Attribute name**: The attribute name used to answer the query; e.g. "attribute name=birthday".
- **Object type**: The type of the object containing the attribute; e.g. "object type=person".

Our EAS system processes queries as follows. For each query answer pair, the system creates a query template by replacing parts of the query string as described above. Once all queries have been grouped into classes, it extracts features for each query answer pair. Using the labeled examples as feature vectors, it trains a separate naïve Bayes classifier for each query class. In total, we were able to find 395 query templates, resolving 85% of our 20,000 labeled queries, yielding 395 unique naïve Bayes models, one for each query class. A sampling of these classes is shown in Table 3. While the scale of our system provides us access to a volume of queries far beyond research systems, we wanted to cover the large range of real user queries, and therefore required a very large and diverse range of data. Our probabilistic modeling is similar to [17], who extract "question contexts" for a question and compute the probability that a given answer is found in the question's context. We use this insight to directly model system behavior.

3 Applications

EAS facilitates a targeted evaluation of the system using a set of labeled query and answer pairs as training data. We developed two applications, one to improve precision (accuracy) and one to improve recall (coverage). Additionally, we allowed annotators direct access to the query classes. We now describe these three applications in detail.

Query	*Answer*
faq games	Games: 0
old blade	Age: 15
death time	Time: 1983-1999
who founded peru	Founded: 1922
country girl play	Country: USA
name meaning life	Property:...is a multi-faceted concept.
what ewok are you	Property:...Ewoks are sentient furred bipeds...
what is in a cell	Property:...is the basic structure of organisms.
uae prime minister	Prime Minister: Sheikh Maktoum... [2]
hong kong area code	Area: 1,098 SQ KM
kansas constitution	Constitution: 34 TH State
ringtones australia	Australia: 1900 956 015
what time is it in chile	Time: 28 December 1998...

Table 1: A sample of question answer pairs that were automatically added to our blacklist. The provided answers are either wrong or non-sensical.

3.1 Blacklist Generator

Our first application directly applies our classifiers to find wrong answers for improved precision through the use of a query blacklist. A query blacklist suppresses known wrong answers, a technique not uncommon for live systems. A blacklist ensures a known precision of 100% and prevents system over-tuning since even highly accurate systems are bound to make mistakes. Our blacklist filters based on a specific query or a specific word in the query (such as "my").[3] A blacklist could also block a type of query. We consider correct answer prediction as a binary classification problem, where, for query class i, we calculate $P_i(Y|\bar{x})$ for $Y = 0$ and $Y = 1$, the probabilities that an answer $\bar{x}$ is incorrect and correct. We use Laplacian smoothing to deal with a sparse feature space. Applying our naïve Bayes models as classifiers, we sampled a large number of unlabeled user queries and answers and assigned each answer a classification score defined as $score(a) = \frac{P_i(Y=1|f(a))}{P_i(Y=0|f(a))}$. Answers with lower scores were more likely to be wrong. We chose scores that were below a given threshold $\lambda = .25$. We restricted classification to classes that contained at least 10 labeled queries so that we would have suitable confidence in each classifier. We evaluated around 60,000 unlabeled queries in this manner, to which most did not result in answers that would be displayed to a user, and produced a blacklist of 972 queries, all of which the system answered with high confidence and had been shown to a user (Table 1). An evaluation revealed that the accuracy of the automatic blacklist was 82%, which improved our system since we favor system precision over recall.

We include several observations of these data. A number of queries of the form *big T* were blacklisted, such as "big mo" (answered: 69709 sq. mi) and "big brown" (answered: 6'3"). The first query matched Missouri (abbreviated "MO") and the second matched a person in the repository named "Brown". A similar mistake was made by the queries "tn land", "wa land", "miami area code", and "mexico area codes", which returned the land areas of Tennessee, Washington, Miami and Mexico. The blacklist revealed a problem with synonym matching in our system, where we assumed the word "high" was a synonym for "height", returning the height of Jake Plummer (answered: 6'2") for the query "jake plummer high school". An amusing example was the template *what T are you*, matching queries like *what jedi are you* or *what tree are you*, which returned answers that, while informative about Jedis and trees, were not answers to the query.

The primary advantage to our blacklist approach was that it correctly identified incorrect answers even in normally correctly behaving query classes. For example, we answer 95% of the queries *how old is T* correctly, but the system identified numerous queries of this form that were incorrect, such as "how old is god" (answered: "Date of Birth: 1985-05-17"). In a related class, the query "jesus christ age" returned "5 B.C.". Numerous other such queries were identified because the answer was

[2]While Sheikh Maktoum was the prime minister of the United Arab Emirates, he passed away in early 2006. The blacklist generator noticed that many of our prime minister answers were out of date and therefore identified this correctly as a wrong answer.

[3]We filter queries with the term "my" to prevent answering queries like "what is my age".

Query	*Query Class*	*Object Name*	*Type*	*Attribute Name*	*Answer Contains*
quebec anthem	anthem T	Quebec	Country	National Anthem	
texas senators	senators T	Texas	State	Senators	Person
capital navarra	capital T	Navarra	Place	Capital	
how big is 1 mu	how big is T	1 mu		Area, Mass	Measurement
tom cruise wife	wife T	Tom Cruise	Person	Spouse	Person
my fair lady director	director T	My Fair Lady	Movie	Director	Person
height of eiffel tower	height T	Eiffel Tower		Height	Measurement
will smith's real name	real name T	Will Smith	Person	Name at birth	
calvin coolidge nickname	nickname T	Calvin Coolidge	Person	Nickname	
who discovered austraila	who discovered T	Austraila		Discoverer	Person
how big is the planet mars	how big is T	Planet Mars		Area, Mass	Measurement
who discovered king tuts tomb	who discovered T	King Tuts tomb		Discoverer	Person
"goldberg variations" composer	composer T	Goldberg Variations	Album	Composer	

Table 2: Missing facts discovered by our system including the matched class, and, when available, the most likely object name, type, attribute and answer features.

of the wrong form or the object type was not correct, such as for the queries "how old is batman" (answered: 1939), "how old is snoopy" (answered: 1950), "how old is gandalf" (answered: 1937), "how old is chocolate" (answered: 1990-02-20), and "how old is darth vader" (answered: 1977). A similar problem was found in the *who founded T* classes, which used the *founded* attribute but also had the *Contains date* feature, which yielded queries such as "who founded san antonio" (answered: 1731), "who founded ikea" (answered: 1943 in Helsingborg, Sweden), and "who founded harvard" (answered: 1636). This is an advantage of building multiple predictors for the system since they can identify different answer properties for each query class.

3.2 Missing Fact Finder

Next we applied our EAS to improving recall (coverage). To expand the capabilities of a QA system, developers evaluate which queries are unanswered. However, a list of unanswered queries is insufficient since many unanswered queries are not good queries. Only a small subset of unanswered queries should be answered, specifically, queries *like* known good queries. We automatically create a list of facts missing from our system and descriptions about what each fact may look like by examining the unanswered queries using our expected answer framework. Consider the unanswered query "president of turkey", which matches a template of the form *president of T*, something we frequently answer. Our accuracy on this class is high, so there is likely a correct answer to this query. Additionally, we know by examination of predictive features from the classifier for the *president of T* class that the object is named *turkey* (matched the T) of type *country* or *nation*, and has an attribute named *president*, whose value is of type *politician* or *person*.

For each unanswered query, we look for a matching query class that our system usually answers correctly. The manner in which a query is matched to a class relies on the method used for constructing tempaltes. As discussed above, we chose a simple string matching method based on templates for our initial evaluation, so our matching algorithm for unanswered queries is based on a similar approach. Given a query and its matched template, the system emits the string in the query that was replaced by the T token in the template as the object name. Next, the system uses the naïve Bayes classifier that corresponds to the class to select features for the attribute name, object type and attribute value that have a high probability given a correct answer, i.e. the weights of the classifier. For each matched query, the system produces a fact description including the object name, attribute name, object type and value type. We used some simple post processing to clean the extracted object names, such as removing common words from the beginning of the object name ("the", "was", "a", etc.).

Since we analyze unanswered queries, a method for matching queries to classes is needed that does not rely on an answer. There are many ways to sort queries into classes [3, 2, 7] and we used a greedy text alignment algorithm, matching each query to the closest template.

1. For each query, construct every possible query template.

		Good Features		
Query Class	*Sample Queries*	*Object Type*	*Attribute*	*Answer Contains*
who is T	who is tom cruise who is george bush who is thurgood marshall	Person	Property[4]	Date[5]
cast T	”cocoon” cast dukes of hazard cast saving private ryan cast	Movie	Cast	Actor
country code T	country code canada spain country code country code uk	Country	Internet country code	Number
wife T	phil mickelson wife hulk hogan wife kevin costner wife	Person	Spouse	Person
how many people live in T	how many people live in california how many people live in wales how many people live in tunisia	Country	Population	Number
who discovered T	who discovered phosphorus who discovered curium who discovered iron	Element	Discoverer	Scientist, Person

Table 3: Some example query classes, actual queries, and popular features of correct answers.

2. Match each generated query template to a known good query template, discarding those that do not match.
3. For each query, keep the query template with lowest alignment penalty, i.e. the template that replaces the fewest number of tokens.

Running our algorithm on 108,728 queries that our system received but produced no answer, we generated 284 new facts. This low number may indicate that a large number of queries do not contain information requests or reflect query types we have never seen. It is impossible to determine the true recall of our methods without examining all of the queries. Examples of discovered missing facts are in Table 2. We evaluated the missing fact list by labeling each fact as either correct (our system should return this fact) or incorrect (there is no such fact). Our missing fact finder was 94% correct in its identification of new facts.

Overall, the quality of the facts was high since the fact finder looked for facts similar to queries we answer well. For example, a number of facts were suggested for “cast” queries, where the user requested a list of the cast in a film. Our system identified the correct attribute of cast for a movie object as well as listing potential types of values, including actor, actress and person.

We envision two uses for our system. First, an automatic fact list gives developers a structured snapshot of user information requests. Rather than looking at raw query volume, a list of new facts users’ are requesting guides the expansion of a QA system. Second, an information extraction system could use this rich information as a guide to extract facts automatically. This enables a system to automatically respond to new trends in user queries by finding information specifically requested by users. We do not intend this method to replace other fact finding systems, such as [13] or [14]. Rather, user queries can supplement more sophisticated methods and direct extraction towards information requested by the user.

3.3 Query Class Analysis

Our final application was an interface that allows an annotator to navigate information about each query class to speedup evaluation. We expose the underlying query class groups, enhancing the annotation interface with a table of all query classes and relevant statistics, including information about system performance on each class and salient features from each class’s classifier. A colored visualization indicates poor performers, query classes that our system frequently answers incorrectly. The

[4]The “property” field of an object contains general descriptions of the object, such as a biography.

[5]Many biographies contain dates of birth or death.

Application	*Description*	*Accuracy*
Blacklist Generator	Find wrong answers in unlabeled queries	82%
Missing Fact Finder	Discover missing facts and descriptions from unanswered queries	94%
Query Class Analysis	Provide annotator with information about system behaviors	–

Table 4: A summary of EAS applications.

system can display a list of queries and answers for each class, as well as the most predictive features for correct and incorrect answers. With this information, an annotator can choose to explore a particular query class in more detail if it performs poorly, which we explore below. Additionally, she can skip unlabeled queries that are likely to be answered correctly based on the accuracy of queries in the same class. Strategic labeling speeds annotation and highlights potential sources of error by indicating which features are common to incorrect answers. Instead of reviewing the system on a per query basis, the annotator sees an overview of system behavior by query class.

A few examples illustrate utility. Our system found the template *bio T*, a very popular query type, to be highly accurate, correctly answering 387 queries (832,997 queries by volume), and incorrectly answering 14 queries (31,662 queries by volume). We decided not to evaluate the additional 971 unlabeled queries (138,805 queries by volume) since the labeled queries perform so well. Next, we found that the query *athletics T* was always answered incorrectly (26 total and 51,224 by volume). An investigation revealed that all such searches are navigational in nature ("lehigh university athletics") but our system incorrectly returned facts about the school's program, like its size or number of teams. Another misbehaving query was *country code* (58% correct), which our system interpreted as the internet country code. This mistake was obvious when we observed that incorrect answers contained the feature *attribute=internet country code*. Finally, the template *holiday T* had no right answers, where T was a country name ("Italy", "Spain", "Barbados"). These navigational queries used the term "Holiday" as "vacation" and not the meaning implied by the answers, which was a day of observance.

Additionally, our query templates captured subtle differences in queries. For example, date of death queries did quite well, such as "when did thomas jefferson die" (answer: July 4, 1826), but how questions do poorly ("how thomas jefferson died"). Similarly, the template *who is on T* is misinterpreted as "who is" and queries like "who is on supreme court" produce descriptions of the entity instead of a list of names. Finally, we were able to substantially improve our coverage by finding that we incorrectly answered *time T* queries (329,682 queries by volume); we added support for this question class. Our framework discovered previously unknown problems, resulting from misinterpreted questions and lack of coverage. This new evaluation strategy can be applied to other QA systems and can speed evaluation and system improvement.

4 Conclusion

We presented the Expected Answer System, a framework that aids evaluation and development of QA systems that augment search results. By modeling individual aspects of a QA system we develop automated methods to improve precision and recall, as well as general identification of system weaknesses (Table 4). Since QA evaluation can be complex, we believe this approach can inform the question answering community. Furthermore, we validate our tools on a production web QA system, which we believe is the first such evaluation of its kind, and provide insights from real world experience that are unavailable in research environments.

5 Acknowledgments

The authors would like to thank Casey Whitelaw, Jeff Reynar, Yusuke Shinyama, David Vespe and Daniel Yehuda for their helpful comments and contributions.

References

[1] E.M. Voorhees and H.T. Dang. Overview of the trec 2005 question answering track. In *TREC 2005 Proceedings*, 2005.

[2] Ulf Hermjakob. Parsing and question classification for question answering. In *Proceedings of the Workshop on Open-Domain Question Answering at ACL-2001*, 2001.

[3] Claire Cardie, Vincent Ng, David Pierce, and Chris Buckley. Examining the role of statistical and linguistic knowledge sources in a general-knowledge question-answering. In *Proceedings of the Sixth Applied Natural Language Processing Conference (ANLP-2000)*, pages 180–187, 2000.

[4] Abraham Ittycheriah, Martin Franz, and Salim Roukos. IBM's statistical question answering system - TREC-10. In *Text REtrieval Conference*, 2001.

[5] Eric Brill, Jimmy Lin, Michele Banko, Susan Dumais, and Andrew Ng. Data-intensive question answering. In *TREC 2001 Proceedings*, 2001.

[6] Eduard Hovy, Laurie Gerber, Ulf Hermjakob, Chin-Yew Lin, and Deepak Ravichandran. Towards semantic-based answer pinpointing. In *Proceedings of Human Language Technologies Conference*, pages 339–345, 2001.

[7] Xin Li and Dan Roth. Learning question classifiers. In *COLING*, 2002.

[8] Radu Soricut and Eric Brill. Automatic question answering: Beyond the factoid. In *Human Language Technology and North American Association for Computational Linguistics Conference (HLT/NAACL-2004)*, 2004.

[9] David Azari, Eric Horvitz, Susan Dumais, and Eric Brill. Actions, answers, and uncertainty: A decision making perspective on web-based question asking. *Information Processing and Management*, 40:849–868, 2004.

[10] Cody C. T. Kwok, Oren Etzioni, and Daniel S. Weld. Scaling question answering to the web. In *World Wide Web*, pages 150–161, 2001.

[11] John Prager, Eric Brown, Anni Coden, and Dragomir Radev. Question-answering by predictive annotation. In *23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 184–191. ACM Press, 2000.

[12] Terence Clifton and William Teahan. Bangor at trec 2004: Question answering track. In *Proceedings of the TREC-13 Conference*. NIST, 2004.

[13] Deepak Ravichandran and Eduard Hovy. Learning surface text patterns for a question answering system. In *Proceedings of the ACL Conference*, 2002.

[14] Michele Banko, Michael J Cafarella, Stephen Soderland, Matt Broadhead, and Oren Etzioni. Open information extraction from the web. In *IJCAI*, 2007.

[15] Marius Pasca and Sanda M. Harabagiu. The informative role of wordnet in open-domain question answering. In *NAACL-01 Workshop on WordNet and Other Lexical Resources*, pages 138–143, 2001.

[16] S. Schlobach, M. Olsthoorn, and M. de Rijke. Type checking in open-domain question answering. In *16th European Conference on Artificial Intelligence (ECAI)*, 2004.

[17] Christopher Pinchak and Dekang Lin. A probabilistic answer type model. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006.