University of Alberta

Data Extraction from Text Using Wild Card Queries

by

Haobin Li ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**

Department of Computing Science

Edmonton, Alberta
Fall 2006

Your file Votre référence
ISBN: 978-0-494-22305-5
Our file Notre référence
ISBN: 978-0-494-22305-5

Abstract

We propose a domain-independent framework for extracting facts and relations from the Web and text repositories, where an extraction task is expressed in a query using a natural language phrase augmented with some wild cards, and the data that best match the query are extracted. Most existing techniques focus on a more specific task (e.g. job postings) or are only applicable to documents that follow a specific formatting (e.g. wrappers). We show that our querying mechanism, despite being simple, can extract a much wider range of facts. To address the problem that a given phrase query can be too restrictive, we propose a rewriting rule language to express alternative rewritings of the query. Also, to distinguish real facts from false matches, we propose a ranking algorithm that assigns higher weights to promising instances. Extensive experiments show that our approach outperforms other options in terms of both precision and recall.

Acknowledgements

From the very beginning of my research to the final draft of this thesis, I owe an immense debt of gratitude to my supervisor, Dr. Davood Rafiei. His constant support and effective guidance make my graduate study a pleasant and fruitful experience.

To my wife,
you gave me unconditional love and support throughout my study in Canada.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The World Wide Web contains a vast amount of information, which makes it a rich source for data extraction. However, manually extracting data from the Web is a tedious and time consuming process, especially when a large amount of data matches the extraction criteria. Example extraction tasks include compiling a list of Canadian writers, a list of car manufacturers, etc. Unless such lists have already been compiled and made available on the Web, one has to query a search engine, examine the pages returned, and extract a handful of instances from each page (if there is any at all). The problem is further complicated by the flexibility of natural languages. Consider the example of extracting *Canadian writers*; many bona fide writers are not referred to as writers. Instead, they are often coined as *authors*, *novelists*, *journalists*, etc. If only the term "Canadian writers" is used in the query, many qualified instances will not be extracted, thus the extraction quality is compromised. Many previous data extraction systems impose a tight restriction on the type of data that can be extracted. For example, the KnowItAll [14] system can extract hyponyms of a user-specified class. The online prototype of the system is further extended to support a limited form of binary relations such as *X "ceo of" Y*. If a user wants to extract something other than hyponyms and those predefined relations, the system is no longer applicable.

We address some of the aforementioned challenges by introducing a framework that allows an extraction task to be encoded as a simple query. A query is a sentence or phrase[1] with some wild cards, and the result of a query is a ranked list of tuples

[1]The query phrases in this thesis are in English, but our framework should be applicable to other

that match the wild cards. For instance, given the query "% is a car manufacturer", the output is expected to be a ranked list of car manufacturers, preferably the real car manufacturers ranked the highest. This query only uses one wild card, here denoted with %. In general, a query can use more than one % wild card, and the result of the query in this case is a table with one column for each occurrence of the wild card.

Our first contribution is a declarative querying framework that integrates wild cards in natural language phrases. Wild card support has several advantages. In our earlier example about *Canadian writers*, for instance, a user can use one type of wild card to indicate that terms similar to *writers* should also be considered. Another type of wild card may be used to indicate a probable position of the desired data, from which values can be extracted. Combining such wild cards with natural language phrases can provide a simple but powerful interface, which can handle much more extraction tasks than previous systems. There is a close correspondence between our queries and star-free regular expressions; our queries make use of certain abstractions geared toward natural languages which make it simpler to write queries.

Our second contribution is the idea of using query rewritings to improve the coverage of the queries and the quality of the results. This is important because a given query may not retrieve an adequate number of facts without considering possible rewritings. Our experiments, as reported in Chapter 6, show that increasing the number of rewritings can improve both recall and precision.

As our third contribution, we design a new algorithm for ranking extracted tuples and *patterns* that are used to extract those tuples. We use the general term *pattern* to refer to both query and query rewriting. The new ranking algorithm, which exploits the mutual reinforcing relationship between extracted tuples and patterns, ranks the tuples in terms of their relevance to the patterns, and measures the effectiveness of extraction patterns. Since the results are ranked, it is possible to set a cutoff threshold to filter invalid rows from the result, making the final results more accessible to the user.

languages as well.

Finally, as our last contribution, we implement our algorithms in the setting of the Web and report a comparison of our results with previously-proposed alternative data extraction algorithms.

The rest of the thesis is organized as follows. Related work is reviewed in Chapter 2. Chapter 3 describes both the syntax and the semantics of wild cards, the queries supported in our framework and an overview of our query evaluation in the context of the Web. Chapter 4 discusses the details of our rewriting rules and patterns. Our ranking algorithms are discussed in Chapter 5. Experimental results are presented in Chapter 6 and we end the paper with conclusions and future work in Chapter 7.

Chapter 2

Related Work

Automatic data extraction of facts and relations from textual contents has many real-world applications, such as putting together a comprehensive catalog and establishing the semantic Web. The problem is important because once the data is extracted, it can be treated as a table in a traditional database and processed with existing tools and techniques. Data extraction is an active research area and encompasses the fields of information retrieval, databases, machine learning, semantic Web, Web search and ranking, question answering, natural language processing, and so on.

This chapter begins with an overview of the challenges and techniques for extracting data from various sources, and proceeds to review related work in the areas of question answering, query rewriting, ranking algorithms, and indexing.

2.1 The Source of Data Extraction

The source of data extraction can be classified into two categories: natural language text (plain text) collection and the Web. Valuable data can be extracted from some large plain text corpus, such as document collections within a corporation and the Wall Street Journal archive. The Web is the largest knowledge base available, which makes it also a good source for data extraction. The Web is typically orders of magnitude larger than other document collections. Thus, techniques applicable to smaller collections may not be able to scale up to the Web. On the other hand, the phenomena of information redundancy on the Web opens the door to new extraction methods and makes large-scale extraction possible. In addition to plain text, Web pages contain many HTML markups for presentation purpose. Regularities among HTML tags and visual formatting of a Web page can also be used to extract desired data.

Our approach uses the Web as data source and therefore benefits from its large volume of information and redundancy. However, we do not consider the HTML tags within a page nor the page visual formatting. Instead, we remove the tags from a Web page and treat it as a plain text document. Details of our approach can be found in section 3.3.

2.1.1 Data Extraction From Plain Text

Interesting information and semantic relations can be extracted from large plain text corpus. For instance, we can automatically find nouns that satisfy the *is-a* or *part-of* relationship with respect to a given noun. More specifically, given the term "car", lexico-syntactic patterns can be used to automatically identify from the text corpus that "the Honda Accord" is a (kind of) car, and a "speedometer" is part of a car. It is also possible to automatically find lexico-syntactic patterns that encode is-a or part-of relationships between entities.

Challenges in data extraction from plain text arise from the fact that natural languages are intended for human consumption, and they are much more complicated and informal than structured data formats (e.g. XML) in terms of knowledge representation. Therefore, special techniques must be used to extract data from plain text and to evaluate the correctness of extracted results.

Hyponym Patterns

Hearst [23] describes a method to automatically identify hyponym lexical relations from a large text collection. (Hyponym relation is also known as is-a relation. For example, *Canada* is a hyponym of *Country* because we can say Canada is a country.) Her algorithm can be summarized as follows:

1. Pick a semantic relation that is of interest. In this case, it is the hyponym relation.

2. Collect a few pairs of terms that are known to satisfy the hyponym relation, like (Canada, country) or (rose, flower). These pairs of examples can be found by bootstrapping from an existing knowledge base.

3. Find places in the text collection where terms from the same pair appear close to each other, and record the context.

4. The commonalities among the recorded contexts yield patterns that indicate the hyponym relation.

Table 2.1 shows a list of hyponym patterns found using the algorithm above (NP stands for Noun Phrase). There are syntactic patterns that indicate hyponym relations in natural language text but are not recognized by the algorithm. However, the algorithm seems to find frequent and reliable patterns that can be used independent of text domains.

Table 2.1: Hyponym patterns identified by Hearst

NP{,} *such as* {NP,}* {(*or* | *and*)} NP	*Large cities such as Philadelphia and San Francisco* ...
such NP *as* {NP,}* {(*or* | *and*)} NP	... in such Alberta cities as Red Deer, Airdrie and Leduc ...
NP{, NP}*{,} *or other* NP	... *Jasper, Banff, or other National Parks in Canada* ...
NP{, NP}*{,} *and other* NP	... *Heather Mallick, Stuart McLean and other Canadian writers* ...
NP{,} *including* {NP,}* {*or* | *and*} NP	... *top-end car manufacturers, including Porsche and Aston Martin* ...
NP{,} *especially* {NP,}* {*or* | *and*} NP	... *operating systems, especially the MSDOS, OS/MVS, UNIX,* ...

To automatically find hyponyms of a given term, one can match the patterns listed in Table 2.1 to each document in the text corpora. The performance can be an issue if the text corpora is large, say 10GB, and a moderate to large number of patterns need to be checked. To reduce the time for pattern matching or to make online queries responsive, a faster algorithm than naively scanning the text collection is needed. There is already work on fast matching regular expressions on large text [11]. The idea is to automatically select substrings with high selectivity from the corpora and use these substrings to build a tree index on the collection. With the help of an index, only relevant documents, usually a small fraction of the collection, are retrieved and examined.

Our work makes use of the hyponym patterns compiled by Hearst to rewrite queries: if a query matches one of the hyponym patterns, it can be expanded with all other hyponyms patterns. Query expansion can improve both recall and precision, as reported by one of our experiments. Details on rewriting queries can be found in Chapter 4.

Part-of Patterns

Based on Hearst's algorithm for finding hyponyms, Berland and Charniak [6] successfully extract *part-of* relations from the Linguistic Data Consortium's North American News Corpus. More specifically, given a word denoting some entity that has recognizable parts, the algorithm extracts terms that are parts of the given entity from the corpus. For example, given the entity *car*, the algorithm may find terms like speedometer, brake, tire, airbag as parts.

By using Hearst's algorithm, the authors find an initial set of five patterns for identifying "part-whole" relations. The set of patterns are evaluated in an experiment and their respective performances are compared. Three of the five patterns are discarded due to their unsatisfactory results, and the remaining two patterns are shown in Table 2.2.

Table 2.2: Part-of patterns identified by Berland and Charniak

NP*'s* NP
e.g. ... car's airbag ...
NP *of* {*the* | *a*} NP
e.g. ... dashboard of the car ...

Given a word that represents the "whole" in a "part-whole" relation, the algorithm instantiates the two patterns with the word, and then uses the instantiated patterns to find matches from the corpus. At the filtering stage, words ending with "ing", "ness", "ity" are filtered from the result because these suffixes are usually associated with words that denote quality rather than physical objects (e.g. the word "durability" in the phrase "durability of the computer"). At the final stage, results are sorted using some statistical methods.

2.1.2 Data Extraction From the Web

The vast amount of information on the Web makes it a rich source for data extraction. Many data extraction techniques developed for plain text are applicable to Web pages after HTML tags are removed. Meanwhile, some of the unique characteristics of the Web, such as its unprecedented size and semi-structured layout within a Web page, lead to promising opportunities and tough challenges for data extraction. In this section, we review several extraction techniques and systems for the Web, including wrapper construction, extraction by examples, and a system for large-scale data extraction.

Wrappers

A wrapper is a computer program that is designed to extract required data from a set of similarly formatted Web pages hosted on a particular website. Applications of wrappers include meta-search engine [31] (e.g. a search engine built on top of other search engines), news/blog syndication, product shopping comparison, etc. A survey by Florescu, Levy, and Mendelzon [17] reviews the work in the area of Web query languages and wrappers before 1998. The survey observes that many websites can be viewed as a container of structured data. For example, the website of the Internet Movie Database (IMDB) is effectively a database for movies, since it contains hundreds of thousands of movie records; for each movie there are data on movie title, cast names, plot summary, average viewer rating, viewer comments, and so on. Information about each movie (typically stored in one HTML page) is analogous to a tuple in a relational database. However, unlike relational data, data in a Web page may not be structured at all or may be partially structured. Data in a Web page is typically wrapped in natural language text and visual formatting primitives (e.g. HTML tags and graphics). There is little meta-data describing the data in a page. The reason is that HTML pages are intended for human viewing rather than data description or querying. On the other hand, for large and resourceful websites, related web pages are automatically generated on the sever side by the same program (e.g. CGI program), thus there exist regularities among related pages that may indicate how the embedded data is stored. Consider two pages from the IMDB that describe two different movies. Since the pages are generated by the same program, the typesettings of data items are expected to be the same: movie titles from both pages are in 16-point, bold, sans-serif letters, followed by the released year enclosed within parentheses; the cast names are formated as a list; the genre is displayed following the string sequence "Genre:" and separated by "/". Based on the commonalities of related pages, we can construct a wrapper specific to the target website, extract data of interest and transform it to a more structured format for more convenient querying and manipulation. For those dynamic web pages that are only accessible through a form, the wrapper also needs to pose an appropriate query in order to access the Web page.

Defining one wrapper by hand is tedious and time consuming; the task of manually constructing wrappers for an increasing number of websites quickly becomes impractical. Therefore, it is more attractive to build wrappers rapidly with little or no human intervention instead of building them by hand. There is some work on the rapid creation of wrappers. One approach [21, 22] allows the wrapper developer to specify the layout of the target Web pages by using a specialized grammar, and to indicate what data is of interest, thus the required data can be extracted. Another approach employs machine learning algorithms to create wrappers. Under the latter approach, one needs to provide a small set of hand-labeled Web pages as a training set to a machine learning algorithm. The wrapper induced by learning algorithms can take new Web pages and extract the required data. More details on wrapper induction using machine learning techniques can be found in [4, 26].

Recent research on wrapper generation takes advantage of not only HTML tag regularities but also visual clues [43]. After all, HTML pages are designed for human viewing and people usually rely on visual patterns to find relevant information from a web page. Consider the scenario where a user tries to find relevant information from a search engine result page: although the page may contain irrelevant blocks such as the search engine logo and the advertisement block, the user can still easily distinguish the actual result block from other "visual noises". The reason is that each result record (i.e. URL and snippet of the result) is rendered within a (transparent) bounding rectangle; the rectangles often have the same width, and are of an equal distance from their neighbors and the left/right boundary of the web browser window. Such visual commonalities can be exploited to locate and extract the desired data. An algorithm [43] that uses visual content features can be briefly described as follows: First, render a sample page (e.g. a search result page with some records), and for each type of objects,such as text, link, and image, record its bounding box's coordinates relative to the browser window. Second, render an empty page (e.g. a result page with logo and advertisements but without any records), record the coordinates of each object. Third, "subtract" the objects rendered in the second step from those in the first step if they have the same coordinates. The purpose of this step is to remove background noises that are present in both pages. Fourth, by using some heuristics, classify the remaining objects into groups of visually similar blocks. Each group, which may contain several records, becomes a candidate for generating wrappers. Fifth, since there may be multiple groups, the most promising group is chosen by studying both visual and non-visual content features. Ideally, the chosen group is the one that has all and only the result records. The content features that govern which group to be picked include: rendering area, center distance, number of records, and average number of characters. Finally, generate a wrapper for the chosen group of blocks/records based on HTML tag regularities.

Due to its practical use and commercial value, rapid wrapper construction has become an active research area. One major drawback of wrappers is that existing wrappers would break when the underlying websites change their presentation formats, which happens frequently in the real world. Our framework does not use wrappers. We use syntactic text patterns to locate target data instead. Since our approach does not depend on HTML tags or other visual features, changes in the formatting of websites will have no negative effects on performance of our system.

Data Extraction by Examples

Although wrappers prove to be useful in real world applications, there are also some limitations. First, a wrapper is specific to one particular website, so it does not work with other websites. If one wants to extract data from hundreds of websites then hundreds of different wrappers have to be built. Even with the state-of-art wrapper generation systems, creating and maintaining a large number of wrappers is costly. Second, building wrappers with automated systems still requires expert knowledge. It is not likely that an average user knows how to generate wrappers using specialized computer programs. Therefore, accessibility to data is limited. Third, even if a large number of wrappers are used, the coverage of these wrappers is still small compared to the whole Web. The vast amount of resources on the Web is not fully explored if information search is only limited to the websites with wrappers.

Brin [8] has proposed an approach to extract patterns and relations which is not limited to a specific page or site. One example of the relations that can be extracted is the pair of *authors* and *titles*. One nice property of Brin's algorithm is that only a small handful of examples of the target relation is required as input, so an average user should find the system easy to use. The algorithm is summarized as follows:

1. Start with a small number of seed tuples of the target relation (e.g. authors and titles).

2. Find all pages that contain the seed tuples on the Web.

3. Generate patterns from the pages retrieved. A pattern may describe the URL prefix of a source Web page, the ordering of the columns in the page, and the common text surrounding the tuples.

4. Search the Web for more tuples of the target relation using the newly generated patterns.

5. Stop if enough tuples are found; otherwise go to step 2.

Brin argues that a pattern cannot be too general because an overly general pattern can extract many false positives. A pattern being too specific is not a problem, however, because its small coverage can be compensated for by information redundancy on the Web.

Pattern relation duality is another observation made in the paper: starting with a good sample of patterns (i.e. those that have high precision and recall), a set of high quality tuples can be found. On the other hand, with a set of good tuples, a set of high quality patterns can be constructed. Therefore, it is possible to expand a small number of examples to a large set of relations and patterns. The Snowball system [1] explores the notion of pattern relation duality further and yields good performance.

Brin's algorithm does a good job when data is structured in a tabular format but is not expected to work on free text. It is generally unlikely to find more than one example of the seed set in a text document such that its surrounding texts are the same. Another pitfall of the algorithm is that the semantic of a query sometimes is not clearly defined by the given examples. In particular, suppose several pairs of *(Canadian author, title)* examples are given, it is still not clear whether the query should be limited to Canadian writers and their books or should include authors from other countries and their work.

Large-Scale Extraction in KnowItAll

The data extraction system KnowItAll [14] takes the description of a concept or class (e.g. cities) as input and extracts instances (e.g. Paris, New York, ...) of the class. The system maintains a set of generic rule templates with some place holders which can be filled with class descriptions in order to identify the instances of the class. KnowItAll uses co-occurrence statistics, in particular mutual information, to assess the relatedness of each instance. One problem that plagues the system is low recall rate. An improved version of KnowItAll [15] adds three new techniques aiming to boost recall while maintaining high precision. The new techniques are:

- Rule Learning: learns domain-specific rules and use them to validate the accuracy of the results.

- Subclass Extraction: given a class (e.g. scientists), the algorithm tries to learn subclasses of the class (e.g. physicists, geologists, ...) and extract instances for the subclasses as well.

- List Extraction: find regularly-formatted lists on the Web that contain a large number of desired data and then extract from the list.

The three techniques are reported to have improved recall significantly while keeping precision comparable to the baseline KnowItAll system. Our approach differs from KnowItAll in several important aspects: First, the query-based interface and the support of wild cards make our approach more adaptive to different extraction tasks. Second, unlike KnowItAll where a concise description of a class must be given, a wild card query may specify only the context in which the instances may appear. This is useful when a concise class description is not available, as shown by some ad hoc queries in our experiments. Last but not the least, our algorithm for assessing the extraction results is novel and performs better than the one used in KnowItAll.

Other-Anaphora Resolution Using the Web

Modjeska et al. [33] propose a machine learning approach that uses the Web for other-anaphora resolution. An anaphor is a phrase that refers to another phrase used earlier. For example, in the sentence "I asked Jim to water the flowers and he did so", the phrase "he" refers to "Jim". In this case, "he" is called an *anaphor* and "Jim" is called an *antecedent*. Other-anaphors are anaphors that contain the modifiers "other" or "another". The goal of other-anaphora resolution is to find what the given other-anaphor is referring to.

Given a set of manually labeled pairs of anaphor and antecedent, the machine learning algorithm for anaphor resolution trains a Naive Bayes classifier based on a few non-Web features, including semantic class for NPs (e.g. person, address, organization), distance between anaphor and antecedent, and so on. In addition to the non-Web features, the training of the classifier also considers one Web feature, which is the Mutual Information between a pair of anaphor and antecedent evaluated on the Web. It is reported that the results (in terms of recall, precision, and F-measure) improve significantly when the Web feature is used together with non-Web features to train the classifier.

2.2 Question Answering Using the Web

A large body of work exists on question answering (QA). For example, the Text Retrieval Evaluation Conference (TREC) [39] has a QA track, where systems compete with each other for retrieving short answers (rather than passages or documents) to a set of questions. QA systems typically decompose the task into two parts: retrieving documents that may contain answers and extracting short answers from those documents [27]. For the first part of the task, systems in the QA track of TREC usually submit a question in its original or modified form to an information retrieval system (e.g. search engine), and retrieve relevant documents from the collection. For the second part, a large number of techniques have been explored, including part-of-speech tagging, named entity extraction, semantic relations, external knowledge base (e.g. dictionaries and WordNet [32]), hand-crafted or automatically generated rules, domain dependent or independent patterns, etc. Some of the aforementioned techniques (such as semantic relations) are computationally intensive and require rather heavy natural language processing; thus they can not easily scale up to large collections.

Our work is related to QA in the following aspects: first, our work makes use of some of the answer extraction techniques described earlier. For those techniques that are not currently used in our approach, it may be possible to incorporate them as enhancements to our existing system. It would be interesting to see how additional extraction techniques can affect the performance of our system. Second, although our system is designed with large-scale data extraction in mind, it is also shown to be effective in answering certain types of questions. Experiments on question answering can be found in Chapter 6.

The AskMSR QA System

The AskMSR system [13] is one of the earlier systems that exploit data redundancy in large corpus (e.g. the Web) for answering natural language questions.

The designers of AskMSR speculate that given a relatively small corpora, finding a correct answer is challenging without computationally intensive NLP techniques. However, if the Web is used as the data source, it is more likely that one can find answers in the proximity to a question text. Therefore, it may be feasible to primarily rely on Web data (with little or no NLP) to perform question answering. AskMSR performs QA in the following four steps:

1. Query reformulation: Given a query, the system generates rewritings that are likely substrings of declarative answers to the question. For instance, "Where is the Louvre Museum located?" is rewritten as "the Louvre Museum is located". The rewritings are generated using hand-crafted rules with the help of a lexicon, and each rewriting has a weight that measures its importance. A *backoff* rewriting is also added in case all other rewritings fail to retrieve any documents. The backoff rewriting is generated by ANDing all non-stop words from the question.

2. N-Gram mining: Each rewriting is formulated as a search query and snippets of search results are downloaded. Unigrams, bigrams, and trigrams are extracted from snippets and weighted according to the rewritings that retrieve them. The final score for an n-gram is based on the weights of the rewritings that retrieve it and the number of unique snippets where it appears

3. N-Gram filtering: The n-grams are filtered and reweighted based on the question type. More specifically, if the question is of *who* type (e.g. "Who killed Abraham Lincoln?"), but an n-gram is not a proper noun (i.e. noun phrase beginning with capital letter), the n-gram should be filtered. The filtering rules are developed manually in advance.

4. N-Gram tiling: n-grams are merged to form longer answers. For example, "University of" and "of Alberta" are merged and become "University of Alberta". Longer n-grams have higher weights than shorter ones.

AskMSR emphasizes the data-redundancy approach to automatic QA. Although the approach works well for TREC questions, and each of which has only one or several correct answers, it still remains unclear whether the approach can be directly applied to large-scale data extraction, where the answer set may contain hundreds or even thousands of facts.

Mining the Web to Play "Who Wants to be a Millionaire?"

The ABC TV show "Who Wants to be a Millionaire?" is a game of multiple choice questions. For a human player, to answer the questions correctly, it requires common sense and the knowledge of popular culture. Lam et al. [27] design and implement a computer program to play the Millionaire game and the computer player is reported to have achieved performance (in terms of average amount of prize won) comparable to human players.

Unlike most QA systems that extract some text as answers, the computer player chooses one of the four choices for each question. Lam's approach exploits the redundancy and volume of information on the Web. More specifically, the approach is based on the idea that words from a question tend to appear in a document containing the answer, and the question words are likely to repeat in that document as well. The idea is especially applicable to large corpus, thus it is natural to use the Web as data source and search engines as document retrieval tools.

The authors' baseline algorithm is as follows. A question and each of its four choices are sent to a search engine and the number of matches are counted. Generally, the response to the question is the choice with the highest search result counts. For inverted questions (e.g. "Which of the following is NOT a city name?"), the response is the choice with lowest result counts. Since some search engines have limits on the maximum number of query terms, stop words from a question are removed, if necessary, to keep the total number of query terms within limits.

The baseline algorithm works reasonably well, but the authors also suggest some heuristics to improve the performance. For example, when a query returns a zero result count, the query can be relaxed by removing some terms from the query until a non-empty set of results is returned. Another heuristic is putting multiple-word choices in quotes to require that they appear as phrases in retrieved documents. In addition, many researchers working in the QA track of TREC believe that answers and questions not only tend to appear in the same document, but they also tend to appear close to each other [37]. Thus, a *proximity* metric, which measures the average distance among question words and answer words, is incorporated into the algorithm of the computer player. Search result counts, proximity metric, and other heuristics are used in combination in the improved version of the computer player, which is shown to be more effective than the baseline system in experiments.

2.3 Query Rewriting

One of the main challenges in extracting data from natural language text is that often desired data appear in different contexts and a user query may give only one of those contexts. As a result, the query may match very few or no results. For example, the query "X is a Canadian writer" can match a writer's name from "Timothy Findley is a Canadian writer." but fails to match anything in the sentence "Canadian writer Margaret Atwood shares her experience ...", which also contains a valid tuple to the query. Query rewriting is an effective technique to improve precision and recall for data extraction and QA systems. In fact, according to the TREC-10 QA evaluation [38], the winning system uses only one resource: a large number of rewritings for each question type [35]. Automatically generating equivalent or similar rewritings for a query still remains an open research area due to its inherent difficulty. To address this problem, we propose a rewriting rule language and compile a set of generic and specific rules for rewriting user queries. A detail discussion on how we address the problem can be found in Chapter 4. This section reviews some methods proposed by the QA research community for automatic query or question rewriting.

Learning Text Surface Patterns

Ravichandran and Hovy investigate the problem of learning surface text patterns that can be used to find answers for various question types [34]. For instance, given a *birthdate* type question, the algorithm can learn text patterns such as `<name> was born in <birthdate>` and `<name> (<birthdate>-`. These patterns can be used to find birthdates in sentences like "Mozart was born in 1756." or "Gandhi (1869-1948) ...". To learn text patterns for a particular question type, say *birthdate*, the algorithm requires an example as input; in this case, the example is chosen as "Mozart 1756". "Mozart" is referred to as the question term and "1756" is referred to as the answer term. Both the question and answer terms are sent to a search engine and the top 1000 result pages are downloaded. Each page is broken into individual sentences, and only those sentences that contain both the question and answer terms are retained. Remaining sentences are passed through a suffix tree constructor. Once the suffix tree is built, it is convenient to find common substrings among sentences along with their lengths and counts (i.e. the number of sentences that contain each substring). All the substrings are filtered so that only those that contain both the question and answer terms are retained. Finally, patterns are generated by replacing the question term with the tag `<name>` and answer term with the tag `<answer>`. The pattern learning procedure is repeated for different examples of the same question type.

The precision of a text pattern is computed using the following algorithm: query the search engine using only the question term (in this example, "Mozart"). Download the top 1000 pages returned, break them into sentences and only retain those that contain the question term. For each text pattern learned, record the number of sentences in which the `<answer>` tag matches by any word (C_o), and record the number of sentences in which the `<answer>` tag matches the correct answer (C_a). The precision of a pattern is calculated as C_a/C_o. Only the patterns with reasonably high precisions are returned.

A large number of high quality text patterns can be compiled for various question types using the algorithms discussed above. Different text patterns for the same question type can be used to generate a query rewriting rule in our data extraction system, as it will be discussed later in Chapter 4.

Using Inference Rules for Query Rewriting

An unsupervised method for discovering *inference rules* is presented by Lin and Pantel [30]. An inference rule is of the form "X wrote Y $\approx$ X is the author of Y". Inference rules can be used to expand a query. In particular, if a given query is "X wrote The Da Vinci Code", it can be rewritten as "X is the author of The Da Vinci Code" according to the inference rule.

To describe the algorithm for discovering inference rules, we need to define what a *path* is. Essentially, a path is a binary relation between two entities. For example, a particular path can represent the semantics that "X wrote Y". Paths can be obtained by parsing a text corpus. Therefore, the problem of discovering inference rules becomes finding paths that are most similar to a given path. According to the Extended Distributional Hypothesis, two paths tend to be similar if they tend to appear in similar contexts. For instance, the two paths "X finds a solution to Y" and "X solves Y" are considered similar if words appearing at X in the first path have a large overlap with the words appearing at X in the second path, and the same holds for words appearing at Y.

Phrase Query Expansion

A term query can be expanded or relaxed by removing some of the terms from the query, as is commonly done in search engines. Removing terms from a phrase query, however, can often ruins its meaning. Phrase query expansion is related to our work because we rely on phrase queries to locate not only relevant documents, but also the exact positions from which tuples are extracted. Phrase query expansion has been shown to be a difficult problem [7]. The following are some of the known techniques for phrase query expansion.

The first level of expansion that is applicable to all phrase queries is *proximity search*. Proximity search still enforces the ordering of words to be the same as that of original query, but words can appear within some distance from their neighbors. For instance, the phrase query "is a country" can be expanded to "is NEAR a NEAR country" in a proximity search. Proximity search weakens the restriction of the original query, thus a discount factor (i.e. penalty) is usually assigned to the rewritten query. Special expansion techniques also exist for some limited types of queries, namely those for people's names and *Noun-Noun Collocations*. A person's full name with optional title, say "President John F. Kennedy", can be expanded to the last name (e.g. "Kennedy"). In this case, it is assumed that the full name and the last name are likely to refer to the same person. Noun-noun collocations can be expanded with appropriate pluralization information. For example, "laptop computer" can be expanded to "laptop computers" but not "laptops computer", while "attorney general" should be expanded to "attorneys general" instead of "attorney generals".

The START Natural Language System

The START system [24] parses natural language text to create a structured knowledge base for question answering. Given a sentence containing clauses, appositions, multiple levels of embeddings, etc, the START system breaks the sentence into smaller units such that each unit contains only one verb. Each unit is analyzed and a *ternary expression* of the form `<subject relation object>` is generated. For instance, two ternary expressions `<<Bill surprise Hillary> with answer>` and `<answer related-to Bill>` are constructed from the sentence "Bill surprised Hillary with his answer." The ternary expressions are stored and indexed in a knowledge base. When given a question, say "Whom did Bill surprise with his answer?", ternary expressions are generated from the question and are used to search the knowledge base for answers.

2.4 Ranking

Data extraction systems can automatically extract a large number of tuples from a source collection, but not all extracted tuples may be correct once verified by a user. Errors are almost always present in the extraction result due to the inherent difficulty of the extraction problem and the quality of the data source. The same can be said for QA systems. Although QA systems typically return only one or a few tuples as answers to a given question, most of them (e.g. [13]) still generate many candidate answers before the most probable one(s) are returned. Having many errors in the result set or contradicting answers for a question makes a system much less useful to users. Therefore, it is important to have a ranking function to assess each extracted tuple and then order the result into a sorted list according to the weights assigned by the ranking function. Once the result is sorted, it is possible to filter some of the errors by setting a threshold on the weight, or simply return the top n tuples as the most promising answers.

This section reviews two ranking algorithms. The first one is the Hypertext Induced Topic Selection (HITS) algorithm for ranking Web pages [25]. The HITS algorithm is closely related to our work because our ranking algorithm is an adaptation of HITS. The other ranking algorithm is Mutual Information (MI). MI has been used in many previous studies (e.g. [14, 36]) to rank candidate answers.

Web Page Ranking with HITS

Due to the large volume of information on the Web, a query in a search engine usually returns thousands or even millions of related pages, and a user is unable to digest all of these pages. One solution to the problem is to rank the Web pages according to their quality and relevance to the query. If one can find enough information from the first few results, there is no need to go through all of the pages returned by the search engine. HITS, in particular, is a well-known algorithm for ranking Web pages based on the hyperlink structure of the Web.

HITS is founded on the Web graph model, where each page is a node on the graph, and there is a directional edge from node p (which denotes page p) to node q (which denotes page q) if and only if there exists a hyperlink from page p to page q. HITS identifies two kinds of pages, namely *authorities* and *hubs*. Conceptually, an authority page contains authoritative information on the query topic, whereas a hub page contains links to multiple relevant authority pages. From the perspective of the Web graph, authorities and hubs exhibit mutually reinforcing relationship: a good hub points to many good authorities; a good authority is pointed to by many good hubs. To break the circularity of the authority-hub relationship, an iterative algorithm is proposed to compute authority weights from hub weights, and vice versa, until both kinds of weights converge. Finally, pages with high authority weights are considered good authoritative pages, and those with high hub weights are regarded as good hub pages.

The HITS algorithm cannot be applied directly in our case to rank extracted tuples, since there is no hyperlink environment in the extraction result. However, a modified version of HITS, which is discussed in Chapter 5, can take advantage of the relationship between extracted tuples and extraction patterns and rank the results.

Mutual Information

The history of Mutual Information (MI) can be traced back to as early as the 1960's [16]. MI is a metric that measures the strength of association between two words or phrases. A pair of words that tend to co-occur, such as "hospital" and "doctor", is expected to have a relatively large MI value.

Turney uses MI to automatically answer multiple choice test questions on synonyms [36]. Given a question word and a candidate answer word, their MI value is computed (details can be found in Chapter 5). The question and candidate pair with the largest MI value is chosen as the answer to the question. It is reported that about 74% of the questions can be correctly answered using this approach.

MI is used in the KnowItAll system [14] to assess the likelihood that an extraction result is correct. For example, suppose "Edmonton" is extracted as an instance for the class "City", the MI between "Edmonton" and a discriminative phrase (e.g. "city of Edmonton") is calculated, and the MI value is used as a factor to decide whether "Edmonton" is really a city or not.

Using MI for ranking has some drawbacks, which are discussed in Chapter 5. In our experiments, MI is used as one of the benchmarks to which we compare our ranking algorithm.

2.5 Indexing

We have developed an online prototype that implements our data extraction framework. The prototype is currently built on top of commercial search engines. There are benefits of using search engines as our source of data. First, search engines generally provide more up-to-date information than locally maintained corpus. Second, development time required to develop the prototype is significantly reduced. However, there are also disadvantages of layering our prototype on top of search engines. One disadvantage is the longer response time for queries, since it is relatively slow to access search engines over the network. Should we decide to have a local collection as source data, there is previous work on indexing the local collection for fast retrieval of relevant documents.

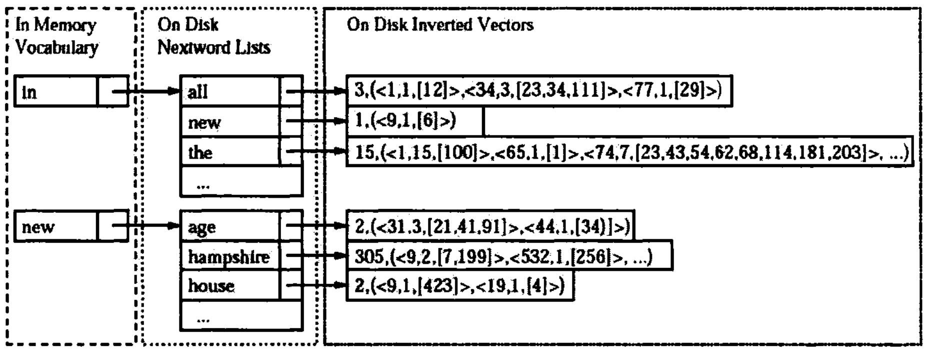


Figure 2.1: A nextword index with two firstwords. The figure is taken from [5]

Inverted index is a standard technique to speed up term queries. However, phrase queries are expensive to evaluate using a regular inverted index. Since the queries we issued are phrase queries (e.g. "countries such as %"), it is desirable to index the local collection using index structures that are efficient for phrase queries.

Nextword index is a special kind of inverted index that is geared towards efficient phrase queries [5, 42]. As shown in Figure 2.1, a nextword index has three levels. The top level contains firstwords, or distinct index terms, in the collection. The middle level stores nextwords (i.e. words appear immediately after the given one) for each index term in the top level. The bottom level stores, for each nextword in the middle level, a posting that denotes the frequency of the firstword-nextword pair across the collection, the document IDs where the pair occurs, and the frequency of the pair inside a particular document, and the position of the pair within that document. For example, the figure shows that the "in-all" pair occurs 3 times in the collection, and it appears within document 1 once with offset 12 w.r.t the beginning of the document. When a phrase query, say "new house", is evaluated against a nextword index, the firstword "new" will be looked up first, followed by the nextword "house", and the posting for "new house" is retrieved. Only documents in the retrieved posting is returned as matches. If a regular inverted index is used, the postings of both "new" and "house" need to be retrieved from disk and merged, and these postings often are much larger in size than the posting of "new house". Therefore, nextword index provides faster phrase query evaluation than regular inverted index.

Cafarella and Etzioni propose a specialized index to support phrase queries with typed variables [9], such as *"powerful <noun>"* and *"Louvre Museum located in <address>"*. Typed variables can be of syntactic types (e.g. noun, verb, adjective) or semantic types (e.g. name, address, phone number). Without a specialized index, an application may have to download a large number of pages containing the phrase query, use a tagger to identify the syntactic and semantic types within the pages, and finally extract those entities that appear next to the given phrase and match the target typed variables. Since many of the returned pages do not have the required typed variables next to the given phrase, a significant fraction of returned pages turn out to be discarded by the application, leading to inefficient usage of network bandwidth and computation resources. If a search engine uses an inverted index that stores the index terms and their neighboring terms along with types, the search engine can return only pages that contain the phrase and matching typed variables. Figure 2.2 illustrates such an index: suppose Document A contains the sentence "... Seattle mayors such as ...", the index entry for "mayors" stores its left neighbor (i.e. "Seattle") along with the left neighbor's types (i.e. NP_{left} and $TERM_{left}$). Similarly, the entry also records the right neighbor "such" and its type. Queries in our framework will also benefit from such NLP application oriented index. Chapter 3 discusses queries that can be accepted by our framework in detail.

2.6 Google's Fill in the Blanks

Google has introduced a new search feature called "fill in the blanks" or wild card search. One can use an asterisk to mark up the *blank*, which indicates the information to search for. Results returned from a wild card search in Google is a list of pages and their snippets. The query terms and the words that match the blank are highlighted within the snippets. For instance, given the query "Edmonton is famous for *", the returned snippets include "**Edmonton is famous for our beautiful river** valley ...", "**Edmonton is famous for its West Edmonton** Mall", etc. It is reported that softer pattern matching (e.g. stemming) and a specialized ranking method are used in Google's "fill in the blanks" [2, 19]. However, the detailed

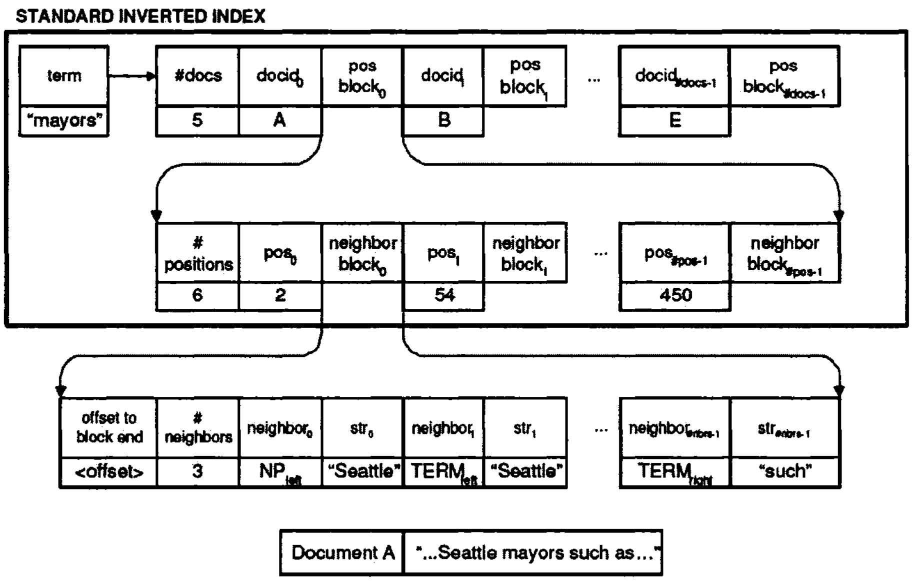


Figure 2.2: An inverted index that stores an index term, the index term's neighbors and the neighbors' syntactic and semantic types. The figure is taken from [9]

algorithm behind the wild card search in Google is not published.

Chapter 3

Query with Wild Cards

To express extraction tasks concisely and in a flexible manner, we make use of wild cards in our queries. This chapter first reviews the common usage of wild cards in computer science, and then introduces the wild cards supported in our framework.

3.1 Wild Cards in Computer Science

The use of wild cards is prevalent in many areas of computer science. Examples are SQL, operating system shells and scripting languages such as Perl, Awk and Python. There are two types of wild cards defined in SQL. The first type (i.e. %) matches any string of zero or more characters. For instance, the following SQL query finds suppliers whose names end with "soft" (e.g. Microsoft).

```
SELECT * FROM supplier
WHERE supplier_name like '%soft';
```

The other type of wild card in SQL (i.e. _) matches exactly one character. The following SQL query is intended to find suppliers whose account numbers consist of six characters and the first five are "263T5".

```
SELECT * FROM supplier
WHERE account_number like '263T5_';
```

Regular expressions, which is a built-in feature for programming languages such as Perl, make extensive use of wild cards. Wild cards in a regular expression mainly serve as the purpose of *quantifiers*. A quantifier after a character or group specifies how frequently the preceded expression can appear [41]. The three common quantifiers are ?, * and +:

- The question mark (?) indicates there is 0 or 1 of the preceded expression. For example, the regular expression `yahoo?` matches "yaho" and "yahoo".
- The asterisk (*) indicates there are any number of the preceeding expression (from 0 to unbounded many). For example, `yahoo*` matches "yaho", "yahoo", "yahooo", etc.
- The plus sign (+) indicates there are 1 or many of the preceeding expression. For example, `yahoo+` matches "yahoo", "yahooo", but not "yaho".

In addition to quantifiers, there are many language-specific wild cards targeting to match subsets of characters. For example, in the Perl programming language, the dot (.) can match any character, `\d` matches numeric character (e.g. 0, 1, ..., 9), and \w matches any alphanumeric character plus the underscore "_".

3.2 Wild Cards Supported by Our Framework

Unlike many of the wild cards we saw in the previous section, wild cards supported by our framework iterate over the domains of *parts of speech* or other meaningful groupings of natural language words. Parts of speech are lexical categories of words. Common parts of speech include nouns, verb, adjectives, adverbs, and so on. In particular, we introduce two types of wild cards, namely * and %. Their syntax and semantics are described as follows.

% wild card

The % wild card represents one or more noun phrases. A noun phrase may consist of one or multiple words, for instance, "movie" and "action movie" are both noun phrases. This wild card, when used in a query, indicates the location of a noun phrase or noun phrases that should be extracted. For example, the query "summer movies such as %" will extract noun phrases *Harry Potter*, *Shrek*, and *Spiderman* from the following sentence: "Popular summer movies such as Harry Potter, Shrek and Spiderman appeal to audience of all ages."

*** wild card**

The * wild card represents a set of phrases similar to a given phrase. Consider the task of finding a listing of summer movies; we may type the query "% is a summer movie". However, some bona fide movies are often referred to as "films", "blockbusters", and so on. In a naive way, one may have to try other terms similar to "movie" manually, save the results each time, and then put the results together at the end. The naive method is tedious and inflexible. In our queries, a term may be enclosed within a pair of * to instruct that the search should be extended to include terms and phrases similar to the given one. For example, the user can re-formulate the query as "% is a summer *movie*", and the query will be automatically expanded to include additional related queries (e.g. "% is a summer film", "% is a summer blockbuster", etc.).

It is feasible to consider other wild cards. For instance, we could have wild cards for verbs, adjectives, or a union of nouns, nouns preceded by adjectives. It is also possible to have a wild card that matches a fixed number of terms. In an attempt to keep the syntax of our queries simple, our queries extend phrase queries of a typical search engine with the two wild cards % and *, as discussed above. The following is a list of example queries:

- % is a *country*
- % is a summer *blockbuster*
- % invented the light bulb
- Google *acquired* %

A query may use any number of % wild cards. Given a query with k % wild cards, the result of the query is a table with k columns, one for each % wild card. Queries with multiple % wild cards can be useful for extracting n-ary relations. For instance, the query "% headquarters in %" can be used to find occurrence of the binary relation `Company - Headquarters`. The result of a query may be ranked to show the fact that some matches are better supported by the text collection and the query rewritings (if available). Therefore, we assume that the result of a query is ranked and each row is assigned a score that shows the level of support it receives; Chapter 5 discusses a few measures to rank the matching tuples of a query.

A query can have any number of * wild cards. Given a query q with some * wild cards, let $q_1, \ldots, q_s$ be the set of queries that are obtained by replacing each * wild card with *similar terms*. A row matches q if it matches at least one of $q_1, \ldots, q_s$; the score of the row for q is an aggregation of the scores of the row for $q_1, \ldots, q_s$. For our purpose, two terms are considered *similar* if they have the same meanings (e.g. synonyms), one is a generalization of the other, or the two terms can be used interchangeably in the same context. The similar terms can be often obtained from dictionaries, thesaurus, online corpus [29], etc. Next we discuss how these queries can be evaluated.

3.3 Evaluating Wild Card Queries on the Web

Our framework for data extraction can be applied to any text repository. When the repository is stored locally, standard techniques can be used to index the collection (e.g. [9, 11]) for fast query processing. In this section, we consider the scenario where data is not stored locally. This is based on the observation that it is not always easy to collect a large text corpora with more up-to-date facts. In particular, we build our data extraction engine on top of a search engine. The response time, of course, will be compromised, but the goal in this thesis is to evaluate the effectiveness of the data extraction framework instead of optimizing the query response time. This section provides an overview of our data extraction algorithm in the context of the Web by using the query "% is a *country*" as an example.

As the first step, the query is analyzed and the words enclosed by pairs of * wild cards (if any) are automatically augmented with their similar terms. The word "country" is enclosed by *'s in the given query. An online system that gives similar terms [28] returns "nation" as one of the similar terms to "country". A new query,

"% is a nation", is formed and added to the expanded query set. More than one query can be added if multiple synonyms are found. The terms returned by the online system may not always be appropriate for expanding the query. In particular, the online system also returns "China" as a similar term to "country", which in turn leads to a new query "% is a China". The meaning of the new query diverges from that of the original one and can have negative effect on the extraction results. To address this problem, we can ask the user to select the terms that are indeed similar from those returned by the online system.

In the next step, each query in the query set is passed to a Part-Of-Speech (POS) tagger. A POS tagger can identify the parts of speech of the words in a sentence or phrase, and optionally "chunk" a group of words into noun phrases, verb phrases, etc. For instance, the query "% invented the light bulb" is processed by a POS tagger and the result is `% _VP(invented) _NP(the light bulb)`, where NP and VP denote noun phrase and verb phrase, respectively. Each tagged query is compared with a set of precompiled patterns for possible rewritings. The result of query tagging is not always reliable, in particular for short queries. To account for those cases, queries are also rewritten using rules that do not require tagging. Let's consider the query "% is a country" first; after tagging, the query conforms to the pattern "NP1 is a(n) NP2". Note that the wild card % matches a noun phrase, as we defined earlier. A pattern may be assigned to a pre-determined class based on its semantic relationship with other patterns. The pattern "NP1 is a NP2", in particular, belongs to the *hyponym class*, since the template indicates that NP1 is a (hyponym of) NP2. Other patterns in the hyponym class include "NP2 such as NP1List", "NP2 including NP1List", etc. All patterns in the matching class (i.e. the hyponym class) are instantiated according to the matched query. Thus, the query set is expanded with extra queries like "countries such as %" and "countries including %". (Chapter 4 discusses our query rewritings in more detail.) Similarly, the query "% is a nation" also matches a pattern in the hyponym class, and the query set is further expanded automatically. If the query cannot match any pattern, no query expansion will occur at this step.

As the third step, all queries in the query set are sent to a search engine. For each query, the matching snippets are downloaded for further processing. When there is a large number of matches, only a fixed number of them are selected.

HTML tags are stripped from downloaded snippets for each query and the remaining text is analyzed to identify the pieces that match the query. Noun phrases (identified by a POS tagger) that appear in the positions of % wild cards of a query are extracted from the text and are saved in the result set. Words other than noun phrases should not be extracted even if they appear in target locations. Suppose the query "% invented the light bulb" is sent to a search engine and the following two snippets are among those returned.

- Thomas Edison is often said to have *invented the light bulb.*

- We all learned in our history classes that Thomas Edison *invented the light bulb* in 1879.

The POS tagger identifies that the word "have" in the first snippet is not a noun phrase, while "Thomas Edison" from the second snippet is. Therefore, the phrase "Thomas Edison" is extracted but "have" is not.

The result of extraction in the previous step is a set of rows; for the given example, each row is a noun phrase. A ranking algorithm is applied to the extracted set. Section 5.3 gives the details of our ranking algorithm. Finally, a sorted list of rows is returned.

Chapter 4

Rewriting Queries

A problem with extracting data from natural language text is that often desired data items appear in different contexts and a user query may give only one of those contexts. As a result, the query may match very few or no items. To address this problem, we propose rewriting rules to express the fact that a query, or in general a phrase, can be rewritten in alternative formats. These alternative forms of a query are expected to return the same or semantically related results but can be syntactically quite different. Our experiments as reported in Chapter 6 confirms that there is a correlation between the number of rewritings and the precision of the retrieved results. Also, using multiple related queries can increase the recall, as it can be seen from the example on *Canadian writers*. It is easy to show that many instances could not have been found if only the original query were used.

4.1 Rewriting Rule Language

The rewriting rule language lists all different ways of rewriting a query. Each rule here is of the form *rule-head* → *rule-body*. A rule head consists of one or more regular expressions, and a rule body consists of one or more rewritings with place holders. Multiple regular expressions in the head or rewritings in the body are separated by newlines. A rule matches a query if any one of the regular expressions in the head matches the query. When a rule matches a query, the query is expanded with all rewritings in the rule body. For each match, keywords from the query may be remembered using *capturing groups*. Capturing group is a feature of regular expressions. To capture a substring for later use, one can put parentheses around the subpattern that matches it. The first pair of parentheses saves its substring in variable `$1`, and second pair saves its substring in variable `$2`, etc [40]. For example, given a string "University of Alberta", the regular expression "`(.*) of (.*)`" captures the substring "University" in variable `$1` and "Alberta" in `$2`. The remembered substrings can be recalled later by referring to the variables. The captured keywords can be transformed (e.g. from a singular noun to a plural noun, or from a past participle to a present participle) before being used in the rewritings. The keywords transformation is done by looking up a dictionary, which is built from the resource files in the Java Extraction Toolkit [20]. This allows us to write generic rewritings that can match a large number of queries. The rewriting rule language should be extendable and we should be able to add more rules as they become available. Here is an example of a rule that makes use of hyponym patterns.

```
(.+),?  such as (.+)
(.+),?  including (.+)
→
$2, and other $1 && plural($1)
$2 is a $1       && singular($1)
```

Given the query "countries such as %", the first regular expression in the rule head matches the query. As a result, "countries" is captured in `$1` and "%" is captured in `$2`. The variables in the rule body are assigned their corresponding values. Also, the value in `$1` is transformed to its plural form for the first rewriting, and to its singular form for the second rewriting in the rule body. At the end, the rule generates "%, and other countries" and "% is a country" as two rewritings to the given query.

In general, a given query can match some of the rules and each rule can give a few rewritings. Hence, each query is mapped to a set of closely related queries which can in turn be used for the extraction.

4.2 Compiling Rewriting Rules

Preparing a set of rewritings for a given query is rather easy, but compiling a set of rewriting rules in advance for any arbitrary query is not straightforward. The effectiveness of a rule usually depends on the precision and the recall for its rewritings and the fraction of queries the rule matches. We group the rewriting rules into two categories: generic and specific. The generic rules can potentially match many queries and can be compiled in advance. Two types of generic rules that we can identify are *hyponyms* and *morphological variants*. A hyponym pattern describes lexico-syntactic relations that can be used to infer one element is a hyponym of another within a sentence. Hearst gives a list of hyponym patterns [23]. A sample of hyponym patterns (the first six from Hearst and the rest put together by us) can be found in Table 4.1.

Table 4.1: Hyponym patterns

NP1 {,} "such as" NP2List
"such" NP1 "as" NP2List
NP1 {,} "especially" NP2List
NP1 {,} "including" NP2List
NP2List "and other" NP1
NP2List "or other" NP1
NP2, "a(n)" NP1
NP2 "is a(n)" NP1
NP1 NP2

The morphological variants of verbs are useful for rewriting many queries that contain verbs. A given query may be rewritten by simply changing its verb tense and without much affecting its meaning. Many extraction tasks are expressed in the form of "Subject transitive-Verb Object" which can be rewritten in a passive form and vice versa. For example, if a user wants to find out who invented the light bulb, she can express the extraction as "% invented the light bulb"; possible rewritings of the query by using morphological variants are "the light bulb was invented by %", "% has invented the light bulb", etc. Our morphological patterns enumerate different verb tenses (e.g. present tense, past tense, ...) and use both active and passive forms. Some of our patterns are presented in Table 4.2 with the objects and the verbs of the patterns instantiated to "the light bulb" and "invent", respectively.

Table 4.2: Morphological patterns

NP1 invent the light bulb
NP1 invents the light bulb
NP1 invented the light bulb
the light bulb is invented by NP1
the light bulb was invented by NP1
the light bulb, invented by NP1

All the relationships described by patterns in the hyponym and morphological classes can be expressed as rules in our rule language.

Although generic patterns and rewritings can be applied to a wide range of queries, it is not hard to find queries where no generic pattern is applicable or sufficient. These queries may be known in advance or may be obtained by examining the query log. In both cases, rewriting rules may be customized to a particular extraction task. Table 4.3 gives some examples of specific patterns targeted to find *discoverers* and their *discoveries*. These specialized rules are likely to match a larger number of high quality tuples, which will lead to improvements in both recall and precision. Manually defining and maintaining specific rewritings can be expensive. There is work on generating specific patterns automatically [30, 34], and the results are very encouraging. For instance, Lin and Pantel [30] automatically compile a list of over 182,000 classes of related patterns. This collection should be used with some care since two patterns in the same class may be loosely related; for instance the patterns "NP1 solves NP2", "NP1 does something about NP2" and "NP1 uses NP2" are returned as related but we may not be able to use one pattern to extract matches for the other.

4.3 Rewriting Quality

Patterns are not equally *strong* in terms of their qualities. For example, "NP1 such as NP2List" is considered a strong pattern because a noun phrase that appears at NP2List is very likely to be a hyponym of the one that appears at NP1. On the other hand, "NP2, a(n) NP1" may be considered a *weak* pattern because sometimes

Table 4.3: Specific patterns for discoverers and their discoveries

NP2 "discovered by" NP1
NP1 "find" NP2
NP2 "found by" NP1
NP1 "uncover" NP2
NP1 "unearth" NP2
NP2List "stumble upon" NP1
NP1 "announce the discovery of" NP2
NP1 "reveal" NP2

the hyponym relation inferred by this pattern is incorrect (e.g. "select a city, a country, and a region from the list."). For the rest of the time, the pattern can be used to extract hyponyms from sentences like "...New York, a city of neighborhoods ...". Similarly, "NP1 NP2" may also be seen as a weak pattern, but we find it very effective in certain applications, such as the names of people. For example, the template can be used to infer from the sentence "Prime Minister Paul Martin attends a Canada Day ceremony..." that "Paul Martin" is a "Prime Minister". The effects of using weak patterns are two folds. First, weak patterns can become strong ones in some cases, and under those circumstances they can improve both recall and precision. Second, weak patterns often introduce more false positives than other patterns. We believe that the negative effects of weak patterns are alleviated since the final results are ranked and the false positives are likely to be removed or assigned very low ranks. Therefore, we use all the patterns returned by the applicable rewriting rules regardless of the patterns' strengths.

Chapter 5

Bringing Order to Results

The data extraction process discussed in Section 3.3 can accumulate a large set of candidates. Some of the candidates are correct, meaning that a user would expect to see them, while the rest are errors.

5.1 Sources of Errors

A query typed by a user can match many more rows than what the user may have had in mind. For instance, the query "% is a country" can match the statement "Joe is a country singer," thus Joe will be added to the list of country names. There are Natural Language Processing (NLP) techniques that are applicable in special cases (such as the given example) and may reduce the number of false positives [14]. More specifically, by requiring "country" to be the head of a noun phrase, the query in the example would not match the sentence because "country" is a modifier instead of the head in the phrase "country singer". Using these techniques can be expensive and sometimes hard to maintain. Also, we are not sure if they can scale up to large volumes of data and queries, for instance on the Web. In general, broad queries are likely to match more false positives. Rewriting queries can also broaden the queries and introduce additional false positives.

False positives also arise when queries are posed to uncontrolled collections such as the Web which contains many incorrect statements. Since the published content may not be verified for correctness, statements, such as "New York is the capital of the United States" are not rare.

One more source of error is due to using the POS tagger. Although the POS tagger produces good results most of the time, it is still a *best-effort* program. Sometimes verbs are misclassified as noun phrases, or vice versa.

Since correct extractions are intermixed with errors, it is important to rank all candidates in terms of their relevance to the user query. A good ranking algorithm should consistently rank correct matches higher than errors, so that errors are pushed down to the bottom of the sorted list. A good ranking would make it easier to draw a cutoff line somewhere in the sorted list, so that we can trade recall for higher precision.

5.2 Ranking Heuristics

Ranking the result of a natural language question over a text corpora is a problem that arises in any question answering system. The precision of an answer usually depends on factors such as the quality and the size of the corpora and the relationships between the text of a question and possible answers. In our case, given a query and a set of rewritings, we want to find out meaningful ways of ordering the results. We first present a few heuristics that may be used to order the result of a query. Then, we present a novel adaptation of the HITS algorithm [25] to the problem of ranking extractions. A comparison of these heuristics and algorithms can be found in our experiments.

Number of Matched Pages or Documents (NPages)

Correct tuples are likely to appear frequently within a query pattern[1] or one of its rewritings. One heuristic is to rank a tuple based on the number of pages or documents in which it matches the query or one of its rewritings. In particular, if the phrase "Thomas King is a writer" appears in 12 documents in the corpora and "Hugh Cave is a writer" appears in only 2 documents, then "Thomas King" is ranked higher by this heuristic than "Hugh Cave". On the Web, we do not usually have access to all pages, and the numbers could be approximated by sampling from a search engine. There are however some problems with this heuristic. First, all

[1]The query would not have been issued in the first place if it is assumed otherwise.

rewritings of a query are treated equally important. Second, many correct instances may not be frequent and we may not be able to separate them from false positives. Finally, the Web contains many duplicate pages and the scores of rows that appear in those pages can be inflated.

Mutual Information (MI)

Another ranking scheme which has been used previously to quantify the relationship between two random variables is the *Mutual Information* (MI). If we denote the probability that a page contains the query string q with $P(q)$, the probability that a page contains a row r with $P(r)$ (the term *row* and *tuple* are used interchangeably in this thesis), and the probability that a page contains a proper encoding of the row using the query string with $P(q,r)$, then the mutual information between q and r is defined as

$$MI(q,r) = log\frac{P(q,r)}{P(q)P(r)}.$$

Suppose q and r are independently distributed, then we have $P(q,r) = P(q)P(r)$ and $MI(q,r)$ equal zero. On the other hand, if q and r tend to appear together, then $P(q,r) >> P(q)P(r)$, yielding a large positive MI value. In some formulations of the mutual information, the above formula is multiplied by $P(q,r)$ [10]. This measure is used to evaluate the association between words [12], and also between the instances of a class and a discriminative phrase [14]. In our case, since $P(q)$ is fixed for a given query, the score of a tuple can be estimated as the ratio between the number of pages where the tuple appears within the query template and the number of pages that contain all the columns of the tuple. These counts can be obtained from a search engine. For a given query and tuple, the MI measures the conditional probability that the tuple appears with the query template given that all the columns of the tuple appears in a page. However, using the MI also has some drawbacks. First, a tuple may not appear with the query but it may appear with one of its rewritings. In the context of the Web, obtaining hit counts can be costly and may not be reliable (e.g. see [3]).

Number of Matched Patterns (NPatterns)

Another simple ranking is to count for each tuple, the number of different patterns (including the query and its rewritings) that would extract the tuple. Because of the semantic correlations between a query and its rewritings, if a tuple is retrieved by multiple patterns, then there is probably a good indication that the tuple is indeed a good match. This approach also has the drawback that all patterns are treated equally important. To address this problem, the next section presents an adaptation of the HITS algorithm for ranking. Note that the HITS algorithm by Kleinberg, which entirely relies on the hyperlink structure, is not directly applicable to our case because there is no hyperlink structure in our context.

5.3 Reinforcing Relationship between Patterns and Tuples

Our hypothesis is that *good tuples* and *good patterns* exhibit a mutually reinforcing relationship: a good tuple is extracted by many good patterns; a good pattern extracts many good tuples. For example, if *Canada* is indeed a good match for the query "% is a country", it should be extracted by many good related patterns, such as "countries including %", "such countries as %", and so on. Similarly, if "countries such as %" is indeed a good pattern for extracting country names, it should extract many good instances, like *the U.S.*, *Canada*, *China*, etc. This mutually reinforcing relationship between patterns and tuples is illustrated in Figure 5.1.

An Iterative Algorithm: Let's associate weight $w_T(t)$ to each tuple t, and weight $w_P(p)$ to each pattern p. The mutually reinforcing relationship between tuples and patterns can be re-stated as follows: if a tuple is extracted by many good patterns (i.e. those with large w_P-values), then it should receive a large w_T-value; if a pattern extracts many good tuples (i.e. those with large w_T-values), then it should receive a large w_P-value. In an iterative and alternating fashion, the weights can be updated as follows:

$$w_T(t) = \sum_{\{p|p \text{ extracts } t\}} w_P(p) \tag{5.1}$$

$$w_P(p) = \sum_{\{t|t \text{ is extracted by } p\}} w_T(t) \tag{5.2}$$

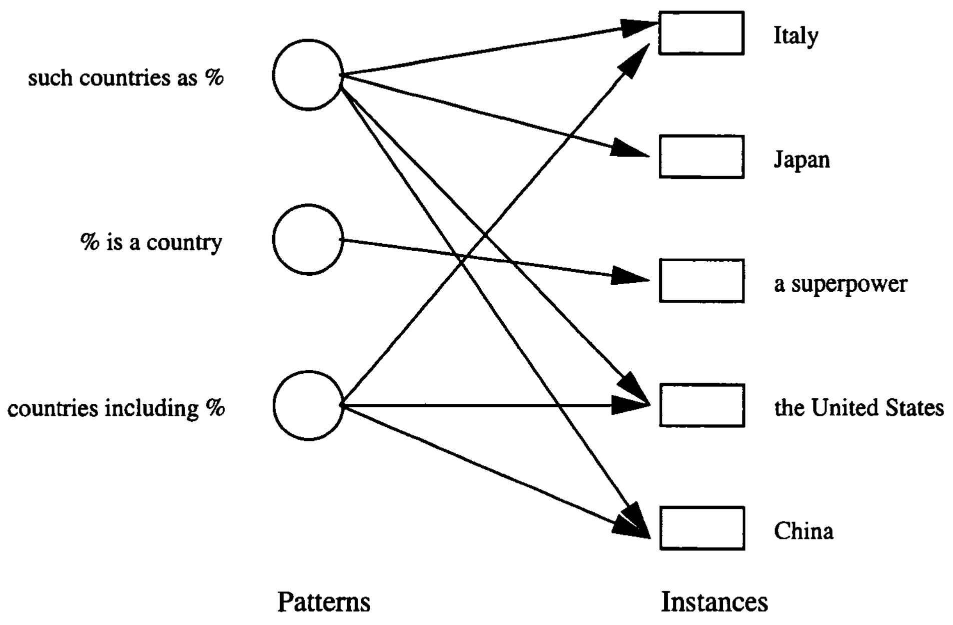


Figure 5.1: Mutually reinforcing relationship between patterns and tuples

The weights initially could be the same for all tuples and the patterns. Based on the reinforcing relationship, we update the weights of tuples and patterns in an alternating and iterative fashion. If we denote the set of tuples with S_T, where $|S_T| = m$, and the set of patterns with S_P, where $|S_P| = n$, then we can normalize the weights of tuples and patterns, so that their squared sums are equal to 1 (i.e. the corresponding vectors have unit lengths): $\sum_{t \in S_T} w_T(t)^2 = 1$, and $\sum_{p \in S_P} w_P(p)^2 = 1$. The normalization is necessary to keep the weights bounded, which is needed to show the convergence of the weights. At the end, tuples with larger weights are considered better, and the same holds for patterns.

Let's represent the set of weights $\{w_T(t)\}$ with vector T such that each coordinate of T corresponds to the w_T-weight of a unique instance t. Similarly, let's represent the set of weights $\{w_P(p)\}$ with vector P such that each coordinate of P corresponds to the w_P-weight of a unique pattern p. The following is the pseudo code for our rank computation.

```
Iterate($S_T$, $S_P$, $k$)
    $S_T$: the set of extracted tuples, $|S_T| = m$
    $S_P$: the set of extraction patterns, $|S_P| = n$
```

```
k: a positive integer
$z^m : (1, 1, 1, \dots, 1) \in R^m$
$z^n : (1, 1, \dots, 1) \in R^n$
Set $T_0 := z^m$
Set $P_0 := z^n$
For $j = 1, 2, \dots, k$
  Apply Eq. 5.1 to $P_{j-1}$, obtaining new $T_j$.
  Apply Eq. 5.2 to $T_j$, obtaining new $P_j$.
  Normalize $T_j$ so that the length of the vector is 1.
  Normalize $P_j$ so that the length of the vector is 1.
End
Return $(T_k, P_k)$
```

Theorem 1 *The sequences $T_1, T_2, T_3, \dots$ and $P_1, P_2, P_3, \dots$ respectively converge T^* and P^*.*

Proof: Let A be the *pattern-tuple matrix* such that the $(i, j)^{th}$ entry of A is 1 if the i^{th} tuple is extracted by the j^{th} pattern, and 0 otherwise. Also let A^{tr} be the transpose matrix of A. It can be verified that Equations 5.1 and 5.2 can be written as $w_T \leftarrow Aw_P$ and $w_P \leftarrow A^{tr}w_T$, respectively. Thus T_k is the unit vector in the direction of $(AA^{tr})^{k-1}Az^n$, and P_k is the unit vector in the direction of $(A^{tr}A)^k z^n$.

A standard result of linear algebra (see [18]) is that if M is a symmetric square matrix, and v is a vector not orthogonal to the principal eigenvector of M, then the unit vector in the direction of $M^k v$ converges to the principal eigenvector of M as k approaches positive infinity. Also, if M has only non-negative entries, then the principal eigenvector of M has only non-negative entries.

In our case, $A^{tr}A$ is a symmetric square matrix with only non-negative entries. Since the principal eigenvector of $A^{tr}A$ contains only non-negative entries and cannot be a zero-vector, z^n is not orthogonal to the principal eigenvector of $A^{tr}A$ (i.e. their dot product does not equal zero). Hence P_k converges to a limit P^* as k increases without bound. Similarly, one can show that T_k converges to a limit T^* as

k increases without bound.

The proof of Theorem 1 leads to the following results (in the above notation).

Theorem 2 *T^* and P^* are the principal eigenvectors of AA^{tr} and $A^{tr}A$ respectively.*

The proofs of both theorems can also be found elsewhere [25]. Theorem 1 shows that the weights of tuples and patterns stabilize when a large enough k is chosen. Alternatively, we can take advantage of Theorem 2 and compute the rankings directly from matrix A, without using the iterative algorithm.

Chapter 6

Experiments

To experiment with our querying interface and to evaluate our algorithms, we built a system called *DeWild* which relies on the Web as its source of data [1]. Using the Web has both benefits and disadvantages. As a benefit, the Web's information redundancy can compensate for the relatively small size and coverage of our rewriting rule set and the lightweight NLP techniques used. However, a challenge is that there are many bogus tuples that need to be filtered.

6.1 System Architecture of DeWild

Figure 6.1 shows the Web query interface of DeWild and Figure 6.2 presents an overview of the architecture of the system. When a query is issued through the Web interface, the *query rewriting module*, which implements the rewriting rule language, rewrites the query to similar or equivalent rewritings. Both the query and its rewritings are passed to the *Extractor module*, which sends each pattern to a Web search engine and retrieves the text snippets from search results. Tuples that match the wild cards in the patterns are extracted from the text snippets. All patterns and their corresponding tuples are taken as input by the *Ranking module*, which produces a ranking of the tuples as well as the patterns. The ranked lists are presented to the user as output. A sample output page is shown by Figure 6.3. One can click on a particular instance to view the URLs from which the instance is extracted (Figure 6.4). Furthermore, by following a URL one can go directly to the

[1]The name *DeWild* stands for Data Extraction using Wild cards. The system is available online at `dewild.cs.ualberta.ca`.

source Web page to examine the instance and the context where it appears (Figure 6.5).

DeWild takes advantage of existing commercial search engines and queries Google and Yahoo (when Google does not respond) via their APIs. It is also possible to access multiple search engines simultaneously via multiple threads in order to reduce turnaround time. In our experiments, 200 snippets are downloaded for each extraction pattern. If there are fewer than 200 snippets found for a pattern, then all available snippets are downloaded[2]. The snippets returned by a search engine typically consist of the search query and its surrounding text. Since the target data appear immediately before or after the user query, they can be often extracted using the snippets only (without downloading the actual pages), hence network and processing costs are significantly reduced.

To find matches for % wild cards, a publicly available POS tagger called NLProcessor[3] is used to identify the part of speech from the text, so that only noun phrases are extracted. For * wilds cards, our system uses a collection of similar terms automatically compiled [29] from Wall Street Journal corpus, but it can equally use other collections as well. Two additional terms are retrieved from the collection for each query term that is enclosed by a pair of * wild cards.

Next we report our experiments with DeWild.

6.2 Recall and Precision

In general, it is difficult to measure recall on the Web since we often do not know the full answer set. The answer set may not be all on the Web, or it can be scattered in many pages of which some may not be crawled or indexed by a search engine. To measure both recall and precision under these constraints, we decided to extract instances of some known classes. To make a comparison with an alternative system, the class names were chosen from those reported for KnowItAll [14].

In our first experiment, we used a list of 191 country names, compiled by the

[2]Our online demo downloads at most 30 snippets for each query and also for each rewriting to keep the response time short.

[3]`www.infogistics.com/textanalysis.html`

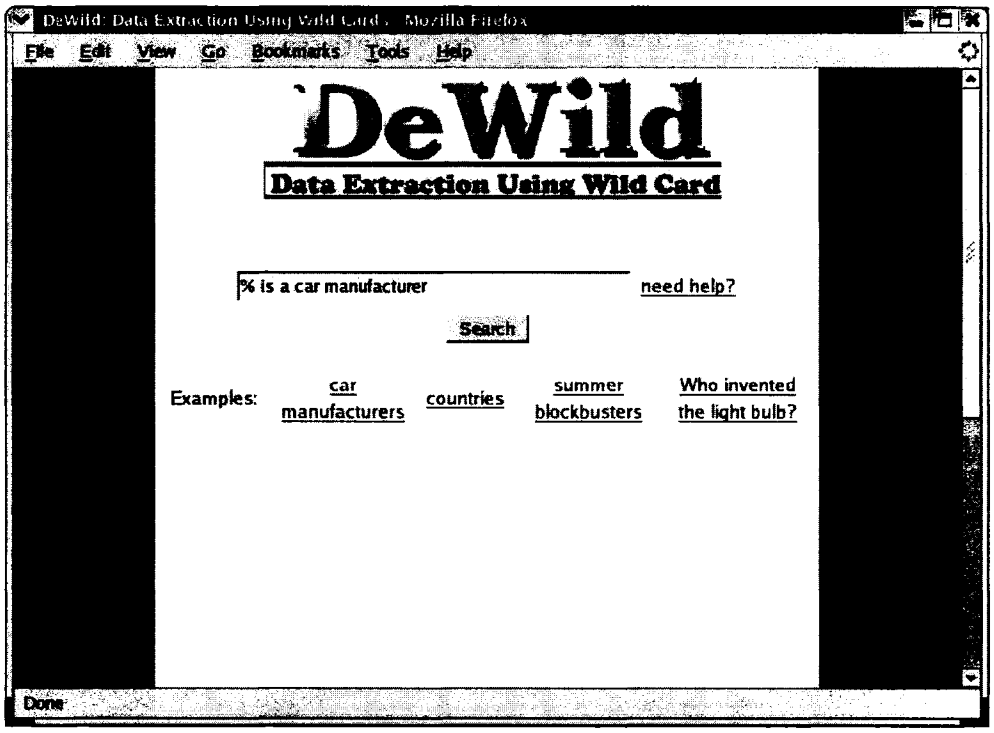

DeWild: Data Extraction Using Wild Card - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
DeWild
Data Extraction Using Wild Card
% is a car manufacturer
need help?
Search
Examples:
car manufacturers
countries
summer blockbusters
Who invented the light bulb?
Done

Figure 6.1: Web interface of DeWild

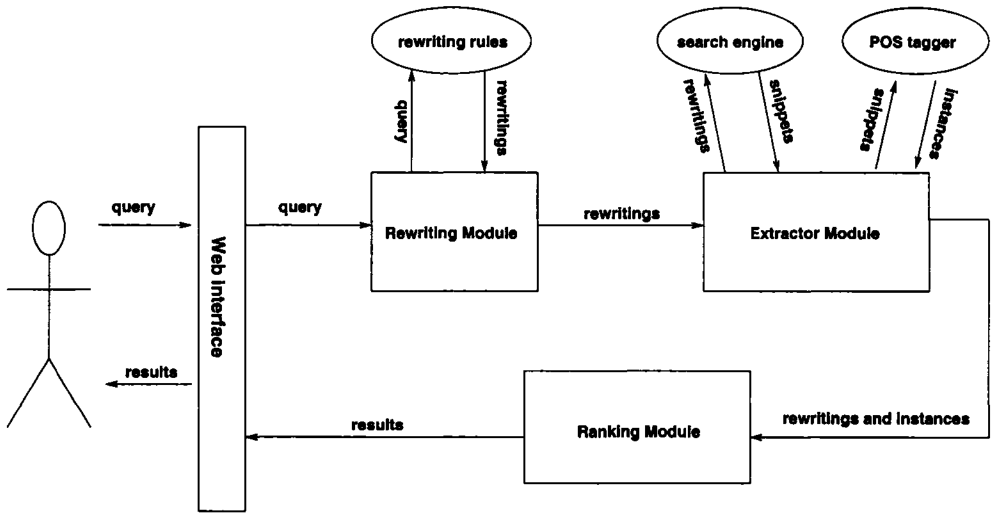
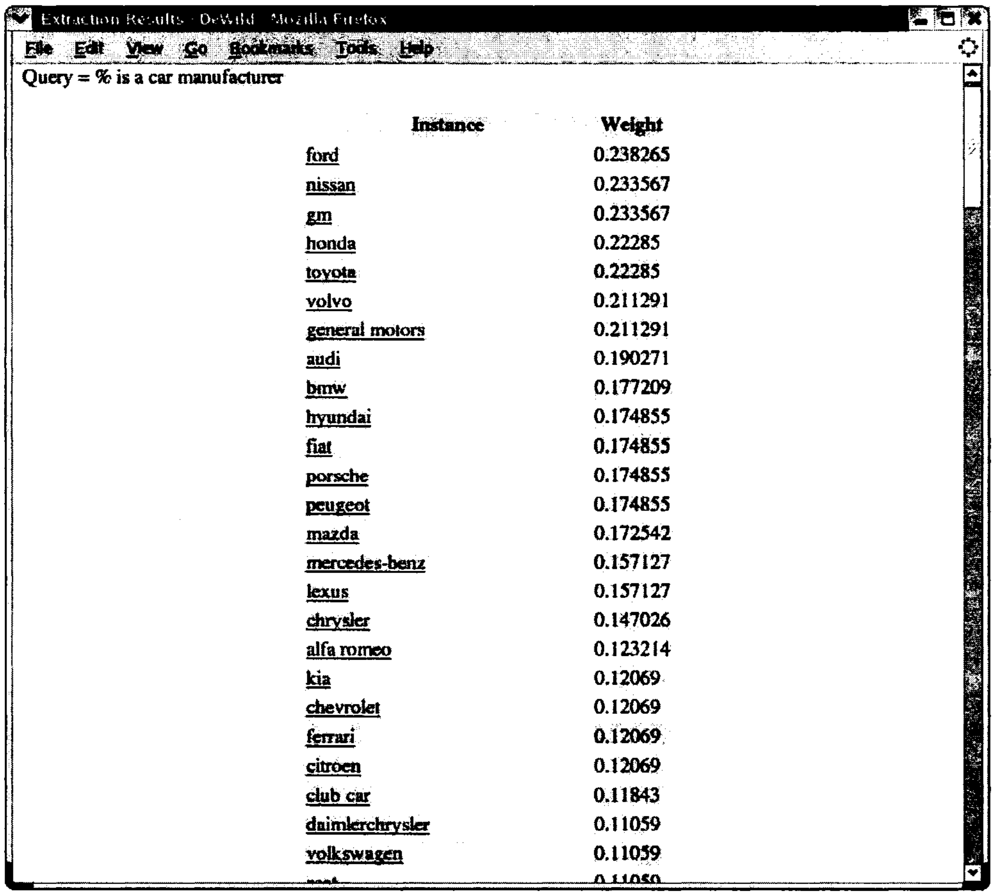


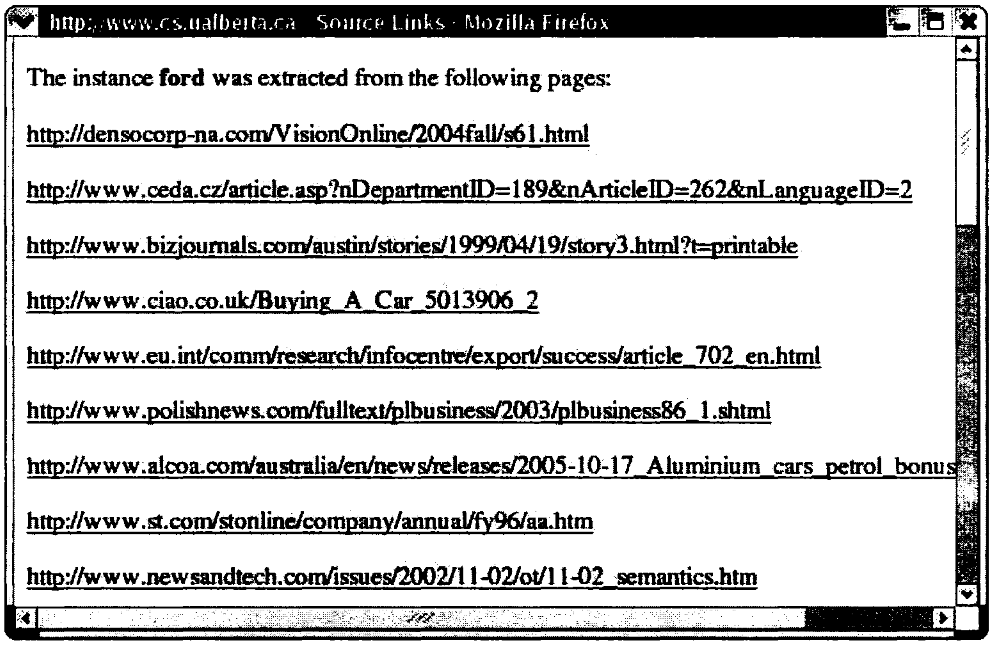
Figure 6.2: System architecture of DeWild



Query = % is a car manufacturer

Instance	Weight
ford	0.238265
nissan	0.233567
gm	0.233567
honda	0.22285
toyota	0.22285
volvo	0.211291
general motors	0.211291
audi	0.190271
bmw	0.177209
hyundai	0.174855
fiat	0.174855
porsche	0.174855
peugeot	0.174855
mazda	0.172542
mercedes-benz	0.157127
lexus	0.157127
chrysler	0.147026
alfa romeo	0.123214
kia	0.12069
chevrolet	0.12069
ferrari	0.12069
citroen	0.12069
club car	0.11843
daimlerchrysler	0.11059
volkswagen	0.11059
[illegible]	[illegible]

Figure 6.3: Instances extracted and their weights for the query "% is a car manufacturer"


http://www.cs.ualberta.ca Source Links - Mozilla Firefox

The instance **ford** was extracted from the following pages:

http://densocorp-na.com/VisionOnline/2004fall/s61.html

http://www.ceda.cz/article.asp?nDepartmentID=189&nArticleID=262&nLanguageID=2

http://www.bizjournals.com/austin/stories/1999/04/19/story3.html?t=printable

http://www.ciao.co.uk/Buying_A_Car_5013906_2

http://www.eu.int/comm/research/infocentre/export/success/article_702_en.html

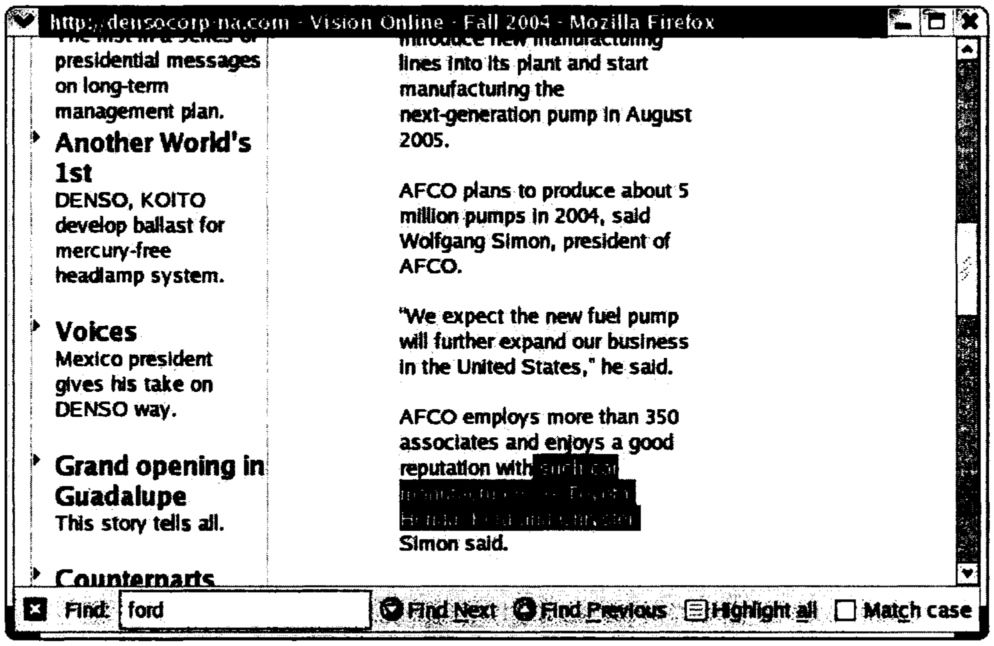
http://www.polishnews.com/fulltext/plbusiness/2003/plbusiness86_1.shtml

http://www.alcoa.com/australia/en/news/releases/2005-10-17_Aluminium_cars_petrol_bonus

http://www.st.com/stonline/company/annual/fy96/aa.htm

http://www.newsandtech.com/issues/2002/11-02/ot/11-02_semantics.htm

Figure 6.4: A list of URLs from which the tuple “Ford” is extracted.


http://densocorp-na.com - Vision Online - Fall 2004 - Mozilla Firefox

presidential messages on long-term management plan.

Another World's 1st
DENSO, KOITO develop ballast for mercury-free headlamp system.

Voices
Mexico president gives his take on DENSO way.

Grand opening in Guadalupe
This story tells all.

Counterparts

lines into its plant and start manufacturing the next-generation pump in August 2005.

AFCO plans to produce about 5 million pumps in 2004, said Wolfgang Simon, president of AFCO.

"We expect the new fuel pump will further expand our business in the United States," he said.

AFCO employs more than 350 associates and enjoys a good reputation with such car manufacturers as Toyota, Honda, Ford and Chrysler, Simon said.

Figure 6.5: A Web page that contains the tuple “Ford”. The Web page is retrieved by clicking on one of the links in Figure 6.4.

US State Department[4], as the ground truth. The query "countries such as %" was used to extract the country names in DeWild. The same query and its rewritings were also used to evaluate the two heuristics NPages and NPatterns; these heuristics are discussed in Chapter 5. As is shown in Figure 6.6, DeWild outperforms both heuristics at almost all recall rates. Note that the recall rate of KnowItAll stops at about 0.7 because that is the recall achieved by the KnowItAll system. Table 6.1 shows the extraction patterns used in the experiment, as well as their weights computed by our ranking algorithm. Any of the patterns in the table could have also been used as a query and the result of DeWild would have been the same. To do a ranking using mutual information (MI), we could use either the query or one of its rewritings. Since it is not clear which one performs the best, we ran the algorithm three times with the discriminative phrases "country of X", "countries such as X", and "X is a country". These variations of MI are respectively referred to as MI-1, MI-2 and MI-3. Figure 6.7 compares DeWild to the online system KnowItAll[5] and MI. We have to point out that there are differences between DeWild and KnowItAll. DeWild uses search engines as its data source; even though the result of a search engine is ranked, we are not making use of this ranking. KnowItAll was originally using Google but it had switched to its own local collection when we tested it. The lack of sufficient details in the KnowItAll paper prevented us from directly implementing it. Since we are comparing precision at each recall, the size of the collection should not have much impact on the comparison. We do not make use of the ranking provided by Google. In fact, out of the first 10 snippets returned by Google for the query "is a country", it is possible to extract valid country names from only 2 snippets. The precision of MI is very poor at low recall rates, which means that the highly ranked instances by MI are mostly errors.

In our second experiment, we used the names of 50 US states as the ground truth and tried to retrieve and rank the same data using DeWild and our other heuristics. Table 6.2 shows the extraction patterns which were used after instantiating "US states" in our generic patterns, as well as their weights computed by DeWild.

[4]`www.state.gov/www/regions/independent_states.html`
[5]`www.cs.washington.edu/research/knowitall`

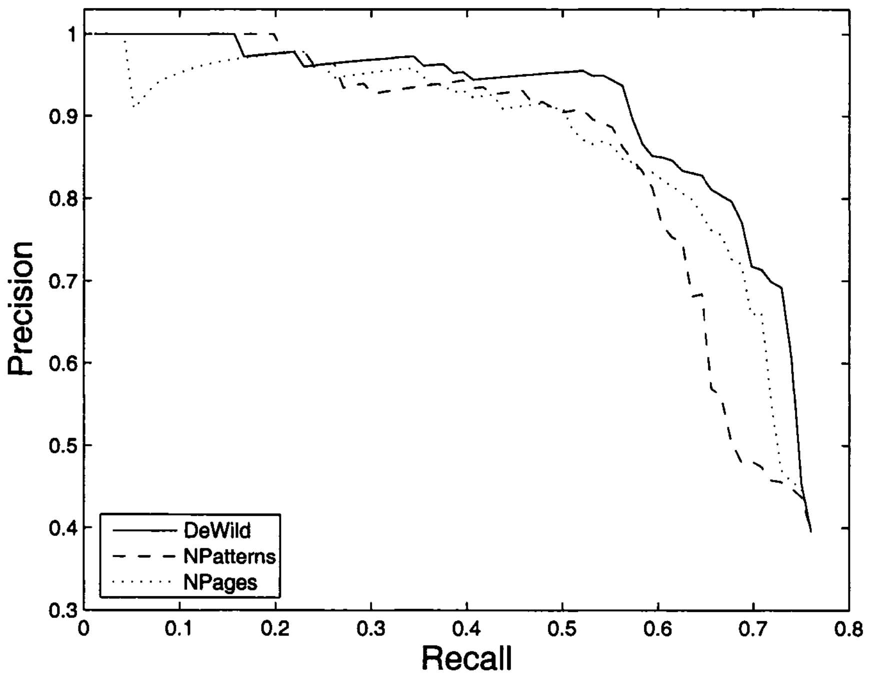


Figure 6.6: Precision and recall for DeWild, NPages and NPatterns. The extraction target are country names.

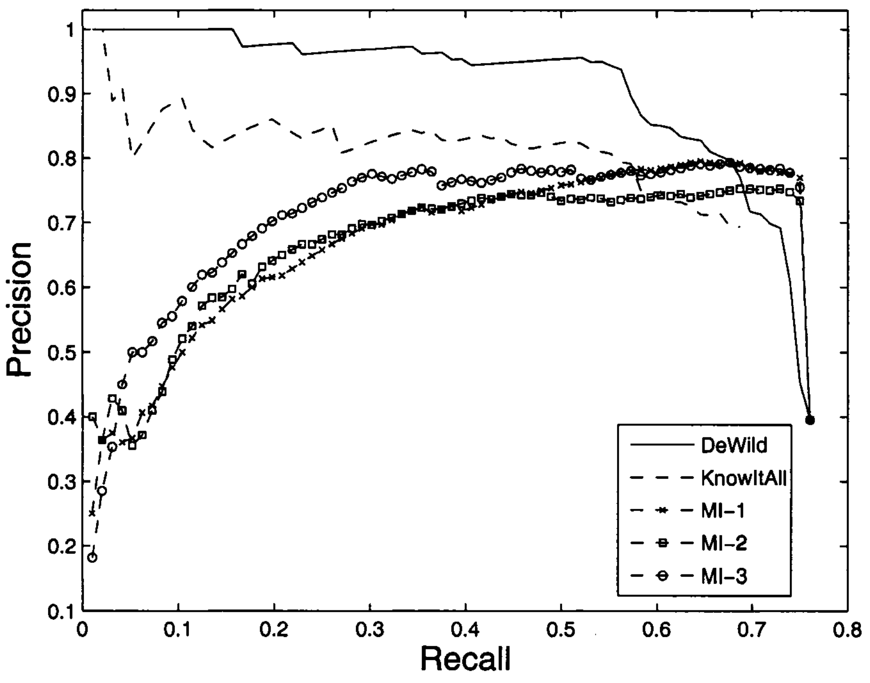


Figure 6.7: Precision and recall for DeWild, MI and KnowItAll. The extraction target are country names.

Table 6.1: Patterns that are used to extract country names, with the weights computed by DeWild in each case

Pattern	Weight
such countries as %	0.645329
countries such as %	0.580203
countries, including %	0.434738
% and other countries	0.158705
countries, especially %	0.127139
% is a country	0.122413
% or other countries	0.038431
%, a country	0.010205
countries %	0

Even though the same set of patterns as the one for country names was used, both the weights and the orderings were different. This is an evidence that the pattern weights are query-dependent and cannot be fixed in advance. Clearly, any of the patterns in the table could have been used as a query and the result returned by DeWild would have been the same. In our evaluation, a retrieved state name was treated "correct" if it was either a full state name or an abbreviation. Figure 6.8 and Figure 6.9 show precision and recall for DeWild, NPages, NPatterns, MI and KnowItAll. Like DeWild, KnowItAll has a precision of 1 when recall is less than 0.35, meaning that the top 35% of the answer is correct. For higher recalls, the precision for KnowItAll drops sharply whereas DeWild has a precision of 1 for all recall rates less than 0.75. Even for higher recall rates, the precision for DeWild does not drop sharply. For our experiments with MI, we used the discriminative phrases "US state of X", "US states such as X" and "X is a US state", which respectively correspond to MI-1, MI-2 and MI-3 in Figure 6.9. Both MI-2 and MI-3 perform poorly in terms of precision for all recall rates. MI-1 performs good at higher recall rates but not so good at smaller recalls, meaning that many incorrect instances show up at the top of the list.

6.3 Number of Rewritings

Adding each query rewriting introduces some cost at the query processing time, and a question is if this additional cost is justified. To evaluate the effect of the

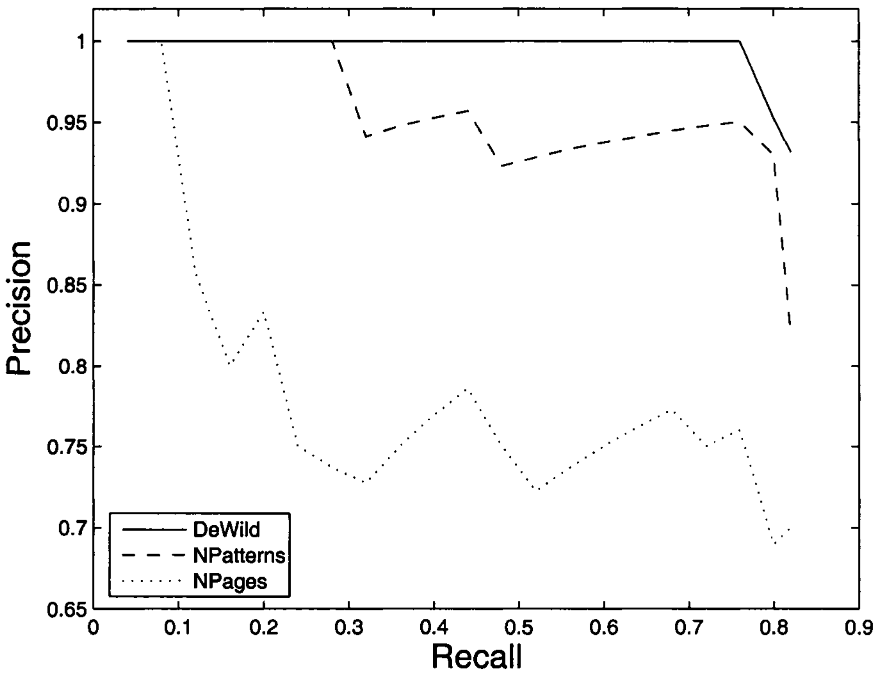


Figure 6.8: Precision and recall for DeWild, NPages and NPatterns. The extraction target are US states.

Table 6.2: Patterns that are used to extract US state names, with the weights computed by DeWild in each case

Pattern	Weight
US states, including %	0.739794
US states such as %	0.526682
% and other US states	0.320306
such US states as %	0.227648
US states, especially %	0.113638
% or other US states	0.074993
% is a US state	0.046522
US states %	0.013729
%, a US state	0

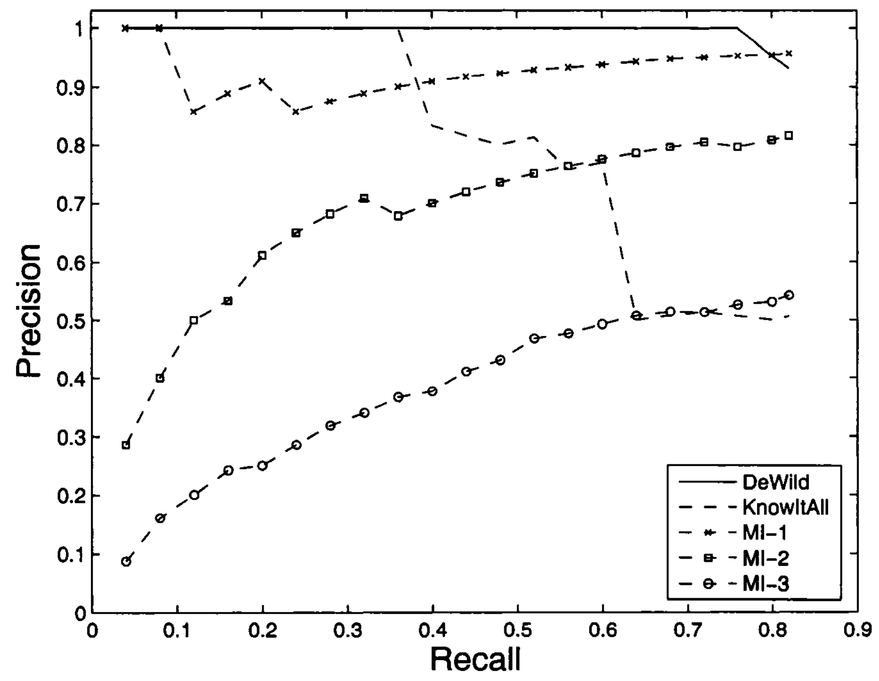


Figure 6.9: Precision and recall for DeWild, MI and KnowItAll. The extraction target are US states.

number of rewritings on the precision and recall, we used DeWild to compile a list of "US states" but varied the number of rewritings that were used. We chose the best sets of 2, 3, and 5 patterns (i.e. those with the highest weights) from Table 6.2 and ran DeWild each time with only one of these sets. The precision-recall curve in each case is shown in Figure 6.10. At the same recall rate, the precision improves significantly when the number of patterns increases from 2 to 3. The precision at higher recalls is further improved when the number of patterns is increased from 3 to 5. We did the same experiment with the country names and the results were the same, hence they are not reported. The results of these experiments show that the performance depends strongly on the number of patterns used.

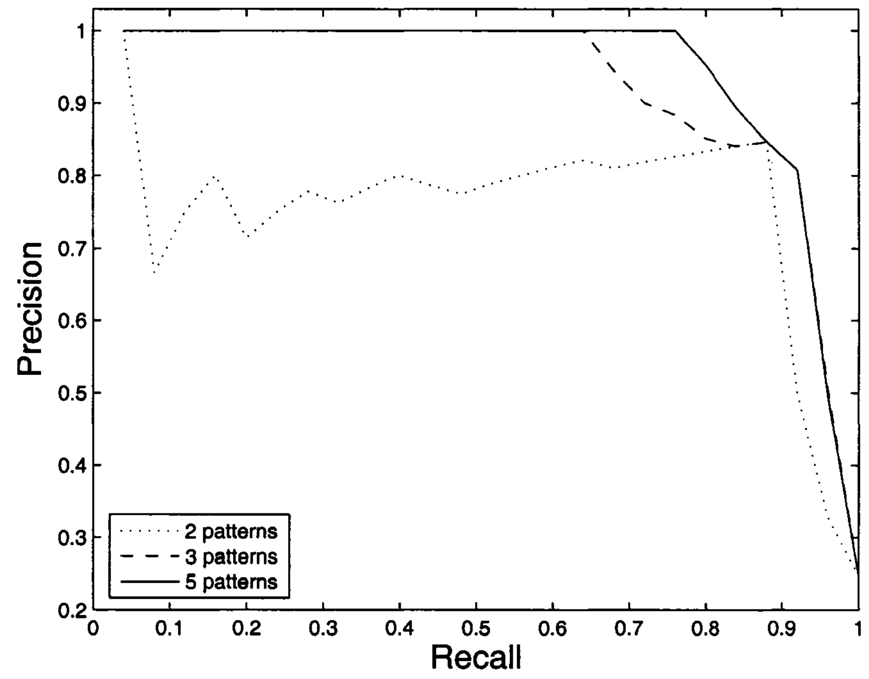


Figure 6.10: The performance of DeWild depends strongly on the number of patterns used.

6.4 Handling Question-Answering Tasks

To do a further evaluation, we tried to use DeWild for question-answering where one of the goals is to return the actual answer to a question, rather than an entire paragraph or a sentence. If a question is formulated as a DeWild query, we can use our approach to locate the answer from the Web. For our evaluation, we took the first five QA targets from the TREC 2004 dataset [39]; since a QA target consisted of multiple questions, we ended up with a total of 22 questions in the experiment (see Table 6.3). For each question, we tried to manually formulate a DeWild query in order to retrieve the answers. We report the number of correct answers given by TREC, the number of answers (correct or otherwise) from DeWild, the number of overlaps between the two, and the number of rewritings used in DeWild. The result of the evaluation is presented in Table 6.4.

Table 6.3: The First 22 Questions from TREC 2004

question id	question
1.1	When was the first Crip gang started?
1.2	What does the name mean or come from?
1.3	Which cities have Crip gangs?
1.4	What ethnic group/race are Crip members?
1.5	What is their gang color?
2.1	What is the name of Durst's group?
2.2	What record company is he with?
2.3	What are titles of the group's releases?
2.4	Where was Durst born?
3.1	When was the comet discovered?
3.2	How often does it approach the earth?
3.3	In what countries was the comet visible on its last return?
4.1	When was James Dean born?
4.2	When did James Dean die?
4.3	How did he die?
4.4	What movies did he appear in?
4.5	Which was the first movie that he was in?
5.1	What does AARP stand for?
5.2	When was the organization started?
5.3	Where is its headquarters?
5.4	Who is its top official or CEO?
5.5	What companies has AARP endorsed?

For 12 questions, all answers returned by TREC were also returned by DeWild. For four of the questions, we couldn't find a pattern between the question and possible answers; hence we couldn't form a query. These are marked with "na" in the table. We found out that the TREC answers for question 1.3 ("Which cities have Crip gangs?") were not the ground truth on the Web; therefore there was small overlap between TREC and DeWild. For questions 2.3 and 4.4, there were more than one formulation of the query but these different formulations were not in our rewriting set; this explains the small overlap between TREC and DeWild. For questions 4.5, the TREC answer was not supported on the Web and we could only find it in NIST's TREC pages. For question 5.4, which asked for the CEO of AARP, TREC had "Horace Deets" or "Tess Canja" as the correct answer; this was based on the information in year 2004. At the time of running our experiments, the correct answer was "Bill Novelli" or "Marie Smith". DeWild extracted the more up-to-date correct answer.

DeWild sometimes returned additional instances of which some were correct and others were incorrect but appeared with the query and gave additional information. For instance, consider the question "Who discovered prions?" from TREC which has only one correct answer. We transformed the question to "prions are discovered by %" and passed it as a query to DeWild. The top 3 answers returned by DeWild are shown in Table 6.5. The highest ranked instance, "Stanley Prusiner", was the correct answer to the question, and it also received a substantially larger weight than the second best instance. Our system returns other acceptable answers, including the 8th-ranked "Dr. Stanley Prusiner", the 9th-ranked "researcher Stanley Prusiner", and the 12-th ranked "Nobel Prize winner Stanley Prusiner". These other answers show that Stanley Prusiner was a doctor, a researcher, a Nobel prize winner and he was from the University of San Francisco.

We also tried the TREC question "Who are the members of the Rat Pack?", which is a list-type question. The question was transformed to "Rat Pack members such as %" before it was tried. Table 6.6 contains the top 8 rows returned by our system. The correct answer to the question consisted of five names, four of which corresponded to the top four rows in Table 6.6, and the remaining one corresponded

Table 6.4: DeWild Handling the First 22 Questions from TREC 2004

question id	ans. in TREC	ans. in DeWild	overlaps	rewritings
1.1	1	2	1	3
1.2	1	na	na	na
1.3	14	5	2	1
1.4	1	na	na	na
1.5	1	4	1	1
2.1	1	2	1	1
2.2	1	4	1	1
2.3	5	7	3	1
2.4	1	7	1	11
3.1	1	3	1	1
3.2	1	na	na	na
3.3	1	na	na	na
4.1	1	1	1	1
4.2	1	1	1	1
4.3	1	15	1	1
4.4	4	7	3	1
4.5	1	2	0	1
5.1	1	6	1	1
5.2	1	2	1	1
5.3	1	20	1	1
5.4	1	12	0	11
5.5	6	17	2	13

Table 6.5: Top candidate answers for the question "Who discovered prions?"

Result	Weight
Stanley Prusiner	0.507819
Scientists	0.408304
University of San Francisco	0.295243

to the 8th row in the Table. Note that the 5th row in Table 6.6 is also correct since it is a spelling variation of the first row.

Table 6.6: Top candidate answers for the question "Who are the members of the Rat Pack?"

Result	Weight
Sammy Davis Jr.	0.454338
Frank Sinatra	0.454338
Peter Lawford	0.27854
Joey Bishop	0.252512
Sammy Davis Jr	0.252512
Michelle Griffin	0.252512
Sammy Davis Jr. playing pool	0.252512
Dean Martin	0.252512

6.5 Ad Hoc Data Extractions

As our last experiment, we tried to compile useful resource lists which we could not find in a list format anywhere on the Web. In one case, we tried to extract the names of senior researchers working at Google. To the best of our knowledge, no such list exists in the public domain. The query used for this task was "% is a senior research scientist at Google". Due to space limitation, only the top 10 instances and their weights are presented in Table 6.7.

We manually verified the names in the table against resources on the Web, and all of them were bona fide researchers at Google. It is worth to point out that the name "Amit" refers to "Amit Singhal", which appears at the 14th position in the list. Also, "lifelong bharat" refers to "Krishna Bharat". A punctuation mark was missing between the two words in the original sentence, which caused the POS tagger to mistakenly group the two words as a noun phrase.

In another case, we tried to find the names of summer movies. Although some online resources maintain a quite complete list of movies, they don't classify movies as summer movies or otherwise. The pattern "% is a summer *blockbuster*" is used as the query for the task. The term *blockbuster*, which is enclosed by * wild cards in the query, is augmented by two extra related terms: *movie* and *film*. The top 10

Table 6.7: A list of senior research scientists at Google

Result	Weight
Bay-Wei Chang	0.345906
David Cohn	0.345906
Sahami	0.2727
Dr. Mehran Sahami	0.2727
Steve Lawrence	0.2727
Krishna Bharat	0.2727
Amit	0.2727
Mehran Sahami	0.2727
Shumeet Baluja	0.2727
lifelong bharat	0.2727

results are given in Table 6.8.

We manually evaluated the extracted results using the Internet Movie Database (IMDB) and concluded that all the results shown in Table 6.8 were indeed correct movie names, and their release dates were in the summer.

Table 6.8: A List of summer movies

Result	Weight
Star Wars	0.279295
Shrek 2	0.247855
Spider-Man 2	0.225779
Harry Potter	0.204052
Spiderman	0.203189
Men in Black	0.202629
Pearl Harbor	0.177852
Mission Impossible	0.171008
Van Helsing	0.171008
Independence Day	0.165511

In one more experiment, we used the query "% is a Canadian *writer*" to compile a list of Canadian writers. The * wild card expands to include two other words that are similar to "writer", namely "author" and "novelist". This time, we put together a set of rewritings that were specific to the query. Table 6.9 shows the rewritings for "writer". Rewritings for "author" and "novelist" are constructed in a similar way. The query returned over 1300 names. We could verify that 91 of the first 100 rows were real Canadian writers. Of the first 200 rows that we verified, 156 were real Canadian writers. We also compared the first 200 tuples to two of the most comprehensive online lists of Canadian writers that we could find. DeWild retrieved 86 real names which could not be found in one list[6] and 70 names which were not in the other list[7]. After combining the two lists, DeWild still reported 58 names which we couldn't find in the combined list. This experiment shows that our approach is a useful extension to current knowledge bases. By combining our approach with existing resource, more comprehensive knowledge bases can be created.

Table 6.9: A list of hand-picked rewritings for finding Canadian writers

Canadian writers including %
Canadian writers such as %
Acclaimed Canadian writer %
such Canadian writers as %
% is a Canadian writer
Award-winning Canadian writer %
Renowned Canadian writer %
%, and other Canadian writers
Canadian writer %
%, a Canadian writer
%, a well-known Canadian writer
Canadian writers %
%, or other Canadian writers
Canadian writers especially %

[6] `www.track0.com/ogwc/authors`
[7] `www.umanitoba.ca/canlit/authorlist`

Chapter 7

Conclusions

We have presented a framework for large-scale data extraction from natural language text, and have evaluated the effectiveness of our framework within a few data extraction tasks on the Web. Our developed querying interface is both simple and extendable with more wild cards and rewriting rules. In the experiments that measure precision and recall, we show that our ranking algorithm is more effective than other heuristics at almost all recall rates. We also show that there is a correlation between precision and the number of rewritings used: more rewritings lead to higher precision in general. The experiment for extracting names of Canadian writers shows that our approach is useful for compiling new resource lists or extending existing knowledge bases.

Our work leads to a few interesting directions. One issue is extracting *n-ary* relations for $n > 3$; the problem in general is difficult since the columns of target rows can be scattered in multiple sentences. To address this problem, we are looking into the possibility of extending our queries or integrating them inside a relational query language. Another direction is finding other interesting classes of wild cards while keeping the queries simple. To improve the running time and to scale up the system to a large number of queries, we may use indexes and do some query optimization in advance. Ordering rewriting rules to prioritize their evaluations and finding other pruning techniques is also another interesting direction.

Bibliography

[1] Eugene Agichtein and Luis Gravano. Snowball: extracting relations from large plain-text collections. Technical report, Columbia University, 1999.

[2] Hiyan Alshawi. Fill in the blanks. `http://googleblog.blogspot.com/2005/08/fill-in-blanks.html`, 2005. [Online; accessed 8-March-2006].

[3] Aris Anagnostopoulos, Andrei Z. Broder, and David Carmel. Sampling search-engine results. In *Proceedings of the WWW Conference*, pages 245–256, 2005.

[4] Naveen Ashish and Craig Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.

[5] Dirk Bahle, Hugh E. Williams, and Justin Zobel. Efficient phrase querying with an auxiliary index. In *Proceedings of the SIGIR Conference*, pages 215 – 221, 2002.

[6] Matthew Berland and Eugene Charniak. Finding parts in very large corpora. In *Proceedings of the ACL Conference*, pages 57–64, 1999.

[7] Matthew W. Billoti. Query expansion techniques for question answering. Master's thesis, Massachusetts Institute of Technology, 2004.

[8] Sergey Brin. Extracting patterns and relations from the world wide web. In *WebDB Workshop at the EDBT Conference*, pages 172–183, 1998.

[9] Michael J. Cafarella and Oren Etzioni. A search engine for natural language applications. In *Proceedings of the WWW Conference*, pages 442 – 452, 2005.

[10] Soumen Chakrabarti. *Mining the Web: discovering knowledge from hypertext data.* Morgan-Kauffman, 2002.

[11] Junghoo Cho and Sridhar Rajagopalan. A fast regular expression indexing engine. In *Proceedings of the ICDE Conference*, pages 419–430, 2002.

[12] Kenneth Ward Church and Patrick Hanks. Word association norms, mutual information, and lexicography. In *Proceedings of the Computational Linguistics Conference*, pages 76–83, 1989.

[13] Susan Dumais, Michele Banko, Eric Brill, Jimmy Lin, and Andrew Ng. Web question answering: Is more always better? In *Proceedings of the SIGIR Conference*, pages 291–298, 2002.

[14] Oren Etzioni, Michael Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Web-scale information extraction in KnowItAll: (preliminary results). In *Proceedings of the WWW Conference*, pages 100–110, 2004.

[15] Oren Etzioni, Michael Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Methods for domain-independent information extraction from the web: an experimental comparison. In *Proceedings of the National Conference on Artificial Intelligence*, pages 391–398, 2004.

[16] Robert M. Fano. *Transmission of information; a statistical theory of communications.* MIT Press, 1961.

[17] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the World Wide Web: a survey. *ACM SIGMOD Record*, 27(3):59–74, September 1998.

[18] Gene H. Golub and Charles F. Van Loan. *Matrix computations.* Johns Hopkins University Press, 1989.

[19] Google. Does google support wildcard searches? `http://www.google.com/support/bin/answer.py?answer=3178`, 2006. [Online; accessed 8-March-2006].

[20] Ralph Grishman. Jet: the Java Extraction Toolkit. `http://www.cs.nyu.edu/cs/faculty/grishman/jet/doc/Jet.html`, 2005.

[21] Jean-Robert Gruser, Louiqa Raschid, Maria Esther Vidal, and Laura Bright. Wrapper generation for web accessible data sources. In *Proceedings of the CoopIS Conference*, 1998.

[22] Joachim Hammer, Héctor García-Molina, Svetlozar Nestorov, Ramana Yerneni, Marcus Breunig, and Vasilis Vassalos. Template-based wrappers in the tsimmis system. In *Proceedings of the SIGMOD Conference*, pages 532–535, 1997.

[23] Marti A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the COLING Conference*, pages 539–545, 1992.

[24] Boris Katz. Annotating the world wide web using natural language. In *Proceedings of the 5th RIAO Conference on Computer Assisted Information Searching on the Internet*, pages 136–155, 1997.

[25] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[26] N. Kushmerick, D. Weld, and R. S. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the JCAI Conference*, pages 729–737, 1997.

[27] Shyong K. Lam, David M. Pennock, Dan Cosley, and Steve Lawrence. 1 billion pages = 1 million dollars? mining the web to play "who wants to be a millionaire?". In *Proceedings of the UAI Conference*, 2003.

[28] Dekang Lin. Dependency-based word similarity (demo). `www.cs.ualberta.ca/~lindek/demos/depsim.htm`.

[29] Dekang Lin. Using syntactic dependency as local context to resolve word sense ambiguity. In *Proceedings of the ACL Conference*, pages 64–71, 1997.

[30] Dekang Lin and Patrick Pantel. DIRT - discovery of inference rules from text. In *Proceedings of the SIGKDD Conference*, pages 323–328, 2001.

[31] Weiyi Meng, Clement Yu, and King-Lup Liu. Authoritative sources in a hyperlinked environment. *Computing Surveys*, 34(1):48–89, 2002.

[32] George A. Miller. Wordnet: a lexical database for english. *ACM Communications*, 38(11):39, 1995.

[33] Natalia Modjeska, Katja Markert, and Malvina Nissim. Using the web in machine learning for other-anaphora resolution. In *Proceedings of the EMNLP Conference*, pages 176–183, 2003.

[34] Deepak Ravichandran and Eduard H. Hovy. Learning surface text patterns for a question answering system. In *Proceedings of the ACL Conference*, pages 41–47, 2002.

[35] M. M. Soubbotin and S. M. Soubbotin. Patterns of potential answer expressions as clues to the right answers. In *Proceedings of the TREC-10 Conference*, pages 175–182, 2001.

[36] Peter D. Turney. Mining the web for synonyms: Pmi-ir versus lsa on toefl. In *Proceedings of the European Conference on Machine Learning*, pages 491–502, 2001.

[37] Ellen M. Voorhees. The trec-8 question answering track report. In *Text REtrieval Conference*, 1999.

[38] Ellen M. Voorhees. Overview of the question answering track. In *Proceedings of the TREC-10 Conference*, pages 157–165, 2001.

[39] Ellen M. Voorhees. Overview of the TREC 2004 question answering track. In *Text REtrieval Conference*, 2004.

[40] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl (3rd Edition).* O'Reilly, 2000.

[41] Wikipedia. Regular expression — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=4283%4800`, 2006. [Online; accessed 8-March-2006].

[42] Hugh E. Williams, Justin Zobel, and Phil Anderson. Index structures for efficient phrase querying. In *Proceedings of the Australasian Database Conference*, pages 141–152, 1999.

[43] Hongkun Zhao, Weiyi Meng, Zonghuan Wu, Vijay Raghavan, and Clement Yu. Fully automatic wrapper generation for search engines. In *Proceedings of the WWW Conference*, pages 66–75, 2005.